AppDynamics App iQ Platform
AppDynamics Platform
Version 4.5.13
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application Monitoring</td>
<td>7</td>
</tr>
<tr>
<td>Overview of Application Monitoring</td>
<td>8</td>
</tr>
<tr>
<td>Install App Server Agents</td>
<td>12</td>
</tr>
<tr>
<td>Agent-to-Controller Connections</td>
<td>15</td>
</tr>
<tr>
<td>Java Agent</td>
<td>20</td>
</tr>
<tr>
<td>Java Supported Environments</td>
<td>22</td>
</tr>
<tr>
<td>Install the Java Agent</td>
<td>32</td>
</tr>
<tr>
<td>Agent Installation by Java Framework</td>
<td>36</td>
</tr>
<tr>
<td>Automate Java Agent Deployment</td>
<td>71</td>
</tr>
<tr>
<td>Deploy Using Containers</td>
<td>72</td>
</tr>
<tr>
<td>Install the AppDynamics Site Extension for Java</td>
<td>74</td>
</tr>
<tr>
<td>Instrument Java System Classes</td>
<td>77</td>
</tr>
<tr>
<td>Instrument JVMs in a Dynamic Environment</td>
<td>78</td>
</tr>
<tr>
<td>Instrument JVMs in Restricted Environments</td>
<td>80</td>
</tr>
<tr>
<td>Instrument JVMs Started by Batch or Cron Jobs</td>
<td>81</td>
</tr>
<tr>
<td>Use Environment Variables for Java Agent Settings</td>
<td>82</td>
</tr>
<tr>
<td>Use System Properties for Java Agent Settings</td>
<td>83</td>
</tr>
<tr>
<td>Administer the Java Agent</td>
<td>85</td>
</tr>
<tr>
<td>Java Agent Directory Structure</td>
<td>87</td>
</tr>
<tr>
<td>Java Agent Logging</td>
<td>88</td>
</tr>
<tr>
<td>Enable SSL for the Java Agent</td>
<td>91</td>
</tr>
<tr>
<td>Upgrade the Java Agent</td>
<td>96</td>
</tr>
<tr>
<td>Uninstall the Java Agent</td>
<td>97</td>
</tr>
<tr>
<td>Tune Java Agent Performance</td>
<td>98</td>
</tr>
<tr>
<td>Troubleshooting Java Agent Issues</td>
<td>100</td>
</tr>
<tr>
<td>Java Agent Configuration Properties</td>
<td>103</td>
</tr>
<tr>
<td>Filter Sensitive Data</td>
<td>113</td>
</tr>
<tr>
<td>IBM Java Agent</td>
<td>116</td>
</tr>
<tr>
<td>Threading and the Java Agent</td>
<td>117</td>
</tr>
<tr>
<td>Constructor Mode Thread Tracing</td>
<td>118</td>
</tr>
<tr>
<td>Executor Mode Thread Tracing (Experimental)</td>
<td>119</td>
</tr>
<tr>
<td>Trace Multithreaded Transactions for Java</td>
<td>121</td>
</tr>
<tr>
<td>Java Agent API User Guide</td>
<td>125</td>
</tr>
</tbody>
</table>

.NET Agent

.NET Supported Environments

Install the .NET Agent for Windows

Configure the .NET Agent

Enable SSL for .NET

Configure the .NET Agent for Windows Services and Standalone Applications

Name .NET Tiers

Name .NET Nodes

Unattended Installation for .NET

Upgrade the .NET Agent for Windows

Resolve .NET Agent Installation and Configuration Issues

Instrumenting SharePoint

Uninstall the .NET Agent

Encrypt Credentials in .NET Agent Configuration

Administer the .NET Agent

.NET Agent Directory Structure

.NET Agent Configuration Properties

Configure Multiple Business Application Support for .NET

Disable Instrumentation for an IIS Application Pool

Configure Application Domain Monitoring

Instrument the DefaultDomain for Standalone Applications

Configure the .NET Machine Agent

Manage Windows Performance Metrics

Enable Correlation for .NET Remoting

Thread Correlation for .NET

Enable Instrumentation for WCF Data Services

Configure Machine Snapshots for .NET

Configure Runtime Reinstrumentation for .NET

Manage Configuration for .NET

.NET Agent for Windows Logging

.NET Microservices Agent

.NET Core Microservices Agent Support

Install the .NET Core Microservices Agent for Windows

AppDynamicsConfig.json File

.NET Core for Linux SDK

.NET Core for Linux SDK Supported Environments

Using .NET Core for Linux SDK

.NET Core for Linux SDK Use Cases

.NET Core for Linux SDK Reference

Page 2

Page 2
C/C++ Application Agent .............................................................. 463
C/C++ Agent Supported Environments ............................................. 465
Using the C/C++ Agent SDK .......................................................... 466
Enable SSL for C++ on Libagent ...................................................... 485
C/C++ Agent SDK Reference ......................................................... 486
Go Language Agent .................................................................... 506
Go Language Supported Environments ............................................. 508
Using Go SDK ........................................................................... 509
Go SDK Reference ...................................................................... 516
IBM Integration Bus Agent ............................................................ 526
IBP Supported Environments .......................................................... 527
Install the IBM Agent ................................................................... 528
Deploy AppDynamics for Azure ...................................................... 531
Install AppDynamics for Azure App Service ...................................... 532
Install the AppDynamics Azure Site Extension for .NET .................... 533
Install the AppDynamics .NET Microservices Agent ......................... 539
Monitor Virtual Applications of Azure Web Apps .............................. 541
Install AppDynamics for Azure Service Fabric ................................. 545
Install AppDynamics for Azure Cloud Services .................................. 548
Upgrade AppDynamics.WindowsAzure NuGet Package ................... 550
Release Notes and PDFs ............................................................... 551
4.4.3 Azure Enhancements and Resolved Issues ............................... 552
Standalone Machine Agents ........................................................... 553
Administer App Server Agents ....................................................... 554
Encrypt Agent Credentials .............................................................. 555
Manage App Agents .................................................................... 560
Agent Log Files ......................................................................... 562
Bytecode Transformer Logging ......................................................... 565
REST Logging ........................................................................... 566
Metrics Limits ............................................................................ 567
Historical and Disconnected Nodes ................................................... 569
Request Agent Log Files ............................................................... 571
App Agent Node Properties ........................................................... 573
App Agent Node Properties Reference .......................................... 575
App Agent Node Properties (A) ....................................................... 579
App Agent Node Properties (B-C) .................................................... 583
App Agent Node Properties (D-E) .................................................... 588
App Agent Node Properties (F-I) ..................................................... 600
App Agent Node Properties (J-L) ..................................................... 602
App Agent Node Properties (M) ....................................................... 608
App Agent Node Properties (N-R) .................................................... 617
App Agent Node Properties (S) ....................................................... 622
App Agent Node Properties (T-Z) .................................................... 626
Dynamic Language Agent Proxy .................................................... 630
Dynamic Agent Proxy Logging ....................................................... 633
Business Applications .................................................................. 634
Flow Maps ................................................................................. 636
Customize Flow Maps ................................................................. 640
Cross Application Flow .................................................................. 647
Configure Cross Application Flow ................................................... 648
Configure Federated Friendship ....................................................... 652
Federated Friendship API Reference .............................................. 658
Export and Import Business Application Settings .............................. 661
Business Transactions ................................................................... 663
Organize Business Transactions ..................................................... 666
Business Transaction Performance ................................................ 671
Transaction Thresholds ................................................................ 673
Dynamic Baselines ..................................................................... 675
Transaction Snapshots ................................................................ 679
Call Graphs ............................................................................... 683
Diagnostic Sessions .................................................................... 687
KPI Analyzer .............................................................................. 689
Transaction Snapshot Collection .................................................... 691
End-to-End Latency Performance ................................................... 693
Monitor Background Tasks ............................................................ 695
Business Transactions Logging ....................................................... 697
Service Endpoints .................................................................... 698
Tiers and Nodes ......................................................................... 700
Moving and Renaming Nodes ......................................................... 701
Troubleshoot Node Problems ....................................................... 704
Monitor JVMs ............................................................................ 705
Automatic Leak Detection for Java ................................................ 707
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configure Instrumentation</td>
<td>709</td>
</tr>
<tr>
<td>Configure Instrumentation Overview</td>
<td>711</td>
</tr>
<tr>
<td>JVM Crash Guard</td>
<td>716</td>
</tr>
<tr>
<td>Garbage Collection</td>
<td>718</td>
</tr>
<tr>
<td>Monitor JMX</td>
<td>721</td>
</tr>
<tr>
<td>Default JMX Metrics for Apache Kafka Backends</td>
<td>724</td>
</tr>
<tr>
<td>Monitor .NET Nodes</td>
<td>725</td>
</tr>
<tr>
<td>Monitor Windows Hardware Resources</td>
<td>726</td>
</tr>
<tr>
<td>Machine Snapshots for .NET</td>
<td>727</td>
</tr>
<tr>
<td>Monitor CLR</td>
<td>730</td>
</tr>
<tr>
<td>Monitor CLR Crashes</td>
<td>731</td>
</tr>
<tr>
<td>Object Instance Tracking for .NET</td>
<td>733</td>
</tr>
<tr>
<td>Monitor IIS</td>
<td>735</td>
</tr>
<tr>
<td>Monitor Node.js Processes</td>
<td>738</td>
</tr>
<tr>
<td>Object Instance Tracking for Node.js</td>
<td>739</td>
</tr>
<tr>
<td>Node.js Metrics</td>
<td>741</td>
</tr>
<tr>
<td>N</td>
<td>Solid Monitoring Data</td>
</tr>
<tr>
<td>Remote Services</td>
<td>744</td>
</tr>
<tr>
<td>Stale Remote Service Removal</td>
<td>746</td>
</tr>
<tr>
<td>Group Remote Services on Flow Maps</td>
<td>747</td>
</tr>
<tr>
<td>Monitor Databases</td>
<td>749</td>
</tr>
<tr>
<td>Resolve Remote Services to Tiers</td>
<td>753</td>
</tr>
<tr>
<td>Troubleshooting Applications</td>
<td>754</td>
</tr>
<tr>
<td>Slow Response Times</td>
<td>755</td>
</tr>
<tr>
<td>Errors and Exceptions</td>
<td>761</td>
</tr>
<tr>
<td>Slow Response Times for .NET</td>
<td>765</td>
</tr>
<tr>
<td>Java Resource Issues</td>
<td>769</td>
</tr>
<tr>
<td>Java Memory Leaks</td>
<td>772</td>
</tr>
<tr>
<td>Java Memory Thrash</td>
<td>775</td>
</tr>
<tr>
<td>Code Deadlocks for Java</td>
<td>778</td>
</tr>
<tr>
<td>Thread Contention</td>
<td>780</td>
</tr>
<tr>
<td>Event Loop Blocking in Node.js</td>
<td>784</td>
</tr>
<tr>
<td>Manage Node.js Process Snapshots</td>
<td>786</td>
</tr>
<tr>
<td>Process Snapshots and Business Transaction Snapshots</td>
<td>787</td>
</tr>
<tr>
<td>Information Points</td>
<td>789</td>
</tr>
<tr>
<td>Java and .NET Information Points</td>
<td>791</td>
</tr>
<tr>
<td>PHP Information Points</td>
<td>793</td>
</tr>
<tr>
<td>Transaction Detection Rules</td>
<td>802</td>
</tr>
<tr>
<td>Automatic Transaction Discovery Rules</td>
<td>815</td>
</tr>
<tr>
<td>Transaction Detection for Apache Web Servers</td>
<td>820</td>
</tr>
<tr>
<td>Validation for Automatic Transaction Discovery Rules</td>
<td>822</td>
</tr>
<tr>
<td>Custom Match Rules</td>
<td>823</td>
</tr>
<tr>
<td>.NET Business Transaction Detection</td>
<td>827</td>
</tr>
<tr>
<td>Java Business Transaction Detection</td>
<td>839</td>
</tr>
<tr>
<td>PHP Transaction Detection</td>
<td>851</td>
</tr>
<tr>
<td>Custom Exclude Rule Examples</td>
<td>855</td>
</tr>
<tr>
<td>Custom Match Rule Live Preview</td>
<td>875</td>
</tr>
<tr>
<td>URI Based Entry Points</td>
<td>883</td>
</tr>
<tr>
<td>Message Queue Entry Points</td>
<td>887</td>
</tr>
<tr>
<td>Business Transaction Discovery Sessions</td>
<td>888</td>
</tr>
<tr>
<td>Backend Detection Rules</td>
<td>893</td>
</tr>
<tr>
<td>Exit Point Detection Rules</td>
<td>898</td>
</tr>
<tr>
<td>Java Backend Detection</td>
<td>900</td>
</tr>
<tr>
<td>Custom Exit Points for Java</td>
<td>907</td>
</tr>
<tr>
<td>Example Message Queue Backend Configuration</td>
<td>912</td>
</tr>
<tr>
<td>Example JDBC Backend Configuration</td>
<td>914</td>
</tr>
<tr>
<td>Apache Kafka Consumer Backends</td>
<td>917</td>
</tr>
<tr>
<td>.NET Backend Detection</td>
<td>919</td>
</tr>
<tr>
<td>Asynchronous Exit Points for .NET</td>
<td>924</td>
</tr>
<tr>
<td>MSMQ Backends for .NET</td>
<td>926</td>
</tr>
<tr>
<td>NServiceBus Backends for .NET</td>
<td>928</td>
</tr>
<tr>
<td>RabbitMQ Backends for .NET</td>
<td>930</td>
</tr>
<tr>
<td>Correlation Over Microsoft BizTalk</td>
<td>933</td>
</tr>
<tr>
<td>Apache Web Server Backend Detection</td>
<td>937</td>
</tr>
<tr>
<td>HTTP Backend Detection</td>
<td>938</td>
</tr>
<tr>
<td>Error Detection</td>
<td>941</td>
</tr>
<tr>
<td>Service Endpoint Detection</td>
<td>945</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Data Collectors</td>
<td>948</td>
</tr>
<tr>
<td>Call Graph Settings</td>
<td>956</td>
</tr>
<tr>
<td>Configure JMX Metrics from MBeans</td>
<td>958</td>
</tr>
<tr>
<td>Exclude JMX Metrics</td>
<td>962</td>
</tr>
<tr>
<td>Exclude MBean Attributes</td>
<td>963</td>
</tr>
<tr>
<td>JMX Logging</td>
<td>964</td>
</tr>
<tr>
<td>Asynchronous Transaction Demarcators</td>
<td>965</td>
</tr>
<tr>
<td>Automatic Instrumentation of Specialist Packages/Frameworks</td>
<td>966</td>
</tr>
<tr>
<td>IBM-BPM Support</td>
<td>967</td>
</tr>
<tr>
<td>Spring Batch Support</td>
<td>978</td>
</tr>
<tr>
<td>OSB Support</td>
<td>979</td>
</tr>
<tr>
<td>Open Tracing Support</td>
<td>980</td>
</tr>
<tr>
<td>Development Level Monitoring</td>
<td>983</td>
</tr>
<tr>
<td>App Server Agents Supported Environments</td>
<td>986</td>
</tr>
</tbody>
</table>
Application Monitoring

AppDynamics Application Performance Monitoring (APM), a component of the AppDynamics platform, gives you end-to-end visibility into the performance of your applications.

AppDynamics works with popular programming languages such as Java, .NET, Node.js, PHP, Python, C/C++, and more, enabling you to:

- Troubleshoot problems such as slow response times and application errors.
- Automatically discover application topology and how components in the application environment work together to fulfill key business transactions for its users.
- Measure end-to-end business transaction performance, along with the health of individual application and infrastructure nodes.
- Receive alerts based on custom or built-in health rules, including rules against dynamic performance baselines that alert you to issues in the context of business transactions.
- Analyze your applications at the code execution level using snapshots.

Agent Installation
- Install the Java Agent
- Install the .NET Agent for Windows
- Install the Node.js Agent
- Install the Python Agent

Configuration
- Transaction Detection Rules
- Backend Detection Rules
- Error Detection
- Data Collectors

Monitoring Applications
- Business Transactions
- Transaction Snapshots
- Tiers and Nodes
- Troubleshooting Applications

Concepts
- Overview of Application Monitoring
- Organize Business Transactions
- Flow Maps
- Remote Services
After you learn the basics of AppDynamics through a hands-on trial, you're ready to learn how AppDynamics models application environments. The model serves as the framework around which AppDynamics organizes and presents performance information.

**Application Model Overview**

A typical application environment consists of the different components that interact in a variety of ways to fulfill requests from the application's users:

- Web applications served from an application server
- Databases or other data stores
- Remote services such as message queues and caches

AppDynamics app agents automatically discover the most common application frameworks and services. Using built-in application detection and configuration settings, agents collect application data and metrics to build flow maps.

A flow map visually represents the components of your application to help you understand how data flows among the application components. For example, the business transaction flow map for a simple e-commerce application below shows data flowing between web services, message queues, and databases:
Automatic detection lets you start exploring AppDynamics features quickly. As your understanding of AppDynamics matures and you identify areas unique to your environment, you can refine your application model.

**Business Transactions**

In the AppDynamics model, a business transaction represents the data processing flow for a request, most often a user request. In real-world terms, many different components in your application may interact to provide services to fulfill the following types of requests:

- In an e-commerce application, a user logging in, searching for items or adding items to the cart
- In a content portal, a user requests content such as sports, business or entertainment news
- In a stock trading application, operations such as receiving a stock quote, buying or selling stocks

AppDynamics app agents discover requests to your application as entry points to a business transaction. Similar requests, such as user login, are treated as multiple instances of the same business transaction. The agents tag the request data and trace the request path as it passes from web servers to databases and other infrastructure components. AppDynamics collects performance metrics for each tier that processes the business transaction.

Because AppDynamics orients performance monitoring around business transactions, you can focus on the performance of your application components from the user perspective. You can quickly identify whether a component is readily available or if it is having performance issues. For instance, you can check whether users able to log in, check out or view their data. You can see response times for users, and the causes of problems when they occur.

**Business Applications**

A business application is the top-level container in the AppDynamics model. A business application contains a set of related services and business transactions.

In a small AppDynamics deployment, only a single business application may be needed to model the environment. In larger deployments, you may choose to divide the model of the environment into several business applications.

The best way to organize business applications for you depends on your environment. A leading consideration for most cases,
however, is to organize business applications in a way that reflects work teams in your organization, since role-based access controls in the Controller UI are oriented by business application.

Nodes

A node in the AppDynamics model corresponds to a monitored server or JVM in the application environment. A node is the smallest unit of the modeled environment. Depending on the agent type, a node may correspond to an individual application server, JVM, CLR, PHP application, Apache Web server.

Each node identifies itself in the AppDynamics model. When you configure the agent, you specify the name of the node, tier, and business application under which the agent reports data to the Controller.

Tiers

A tier is a unit in the AppDynamics model composed of a grouping of one or more nodes. How you organize tiers depends on the conceptual model of your environment.

Often a tier is used to a group of a set of identical, redundant servers. But that is not strictly required. You can group any set of nodes, identical or not, for which you want performance metrics to be treated as a unit into a single tier.

The single restriction is that all nodes in a single tier must be the same type. That is, a tier cannot have mixed types of agents, such as both .NET and Java nodes.

The traffic in a business application flows between tiers, as indicated by lines on the flow map, which are annotated with performance metrics.

In the AppDynamics model:

- There is no interaction among nodes within a single tier
- An application agent node cannot belong to more than one tier

Backends

A backend is a component that is not instrumented by an AppDynamics agent but that participates in the processing of a business transaction instance. A backend may be a web server, database, message queue, or another type of service.

The agent recognizes calls to these services from instrumented code (called exit calls). If the service is not instrumented and cannot continue the transaction context of the call, the agent determines that the service is a backend component. The agent picks up the transaction context at the response at the backend and continues to follow the context of the transaction from there.

Performance information is available for the backend call. For detailed transaction analysis in for the leg of a transaction processed by the backend, you need to instrument the database, web service or other application.

Integration with other AppDynamics Modules

This section describes how other AppDynamics APM Platform products work with Application Monitoring to provide complete, full visibility on application health and user experience.

Application Monitoring and Infrastructure Visibility

Infrastructure Visibility provides end-to-end visibility into the hardware and networks on which your applications run. You can use Infrastructure Visibility to identify and troubleshoot problems that affect application performance such as server failures, JVM crashes, and hardware resource utilization. There are three classes of Infrastructure Visibility functionality:

- You use the Standalone Machine Agent to collect basic hardware metrics. One Machine Agent license is included with each App Agent license that you purchase. You can deploy this Machine Agent only on the same machine where the App Agent is installed. The functionality provided by the Machine Agent includes:
  - Basic hardware metrics from the server OS. For example, %CPU and memory utilization, disk and network I/O
- Custom metrics passed to the Controller by extensions
- Run remediation scripts to automate your runbook procedures. You can optionally configure the remediation action to require human approval before the script is started.
- Run JVM Crash Guard to monitor JVM crashes and optionally run remediation scripts

- If you have a Server Visibility license, the Standalone Machine Agent provides the following additional functionality:
  - Extended hardware metrics such as machine availability, disk/CPU/virtual-memory utilization, and process page faults
  - Monitor application nodes that run inside Docker containers and identify container issues that impact application performance
  - The Tier Metric Correlator, which enables you to identify load and performance anomalies across all nodes in a tier
  - Monitor internal or external HTTP and HTTPS services
  - Group servers together so that health rules can be applied to specific server groups
  - Define alerts that trigger when certain conditions are met or exceeded based on monitored server hardware metrics

- Network Visibility monitors traffic flows, network packets, TCP connections, and TCP ports. Network Agents leverage the APM intelligence of App Server Agents to identify the TCP connections used by each application. Network Visibility includes the following functionality:
  - Detailed metrics about dropped/retransmitted packets, TCP window sizes (Limited / Zero), connection setup/teardown issues, high round trip times, and other performance-impacting issues
  - Network Dashboard that highlights network KPIs (Key Performance Indicators) for tiers, nodes, and network links
  - Right-click dashboards for tiers, nodes, and network links that enable quick drill-downs from transaction outliers to network root causes
  - Automatic mapping of TCP connections with application flows
  - Automatic detection of intermediate load balancers that split TCP connections
  - Diagnostic mode for collecting advanced diagnostic information for individual connections

### Application Monitoring and Browser Real User Monitoring

When you add End-User Monitoring to Application Performance Management, you can correlate business transaction performance to the user experience for those transactions. See [Correlate Business Transactions for Browser RUM](#).

If server app agents run on the applications that serve your browser applications, you can further configure the app server agents to inject JavaScript agent into the code that runs on the browser. Access the settings to configure injection in the Applications Configuration page. For more information, see [Automatic Injection of the JavaScript Agent](#) and [Assisted Injection](#).

### Application Monitoring and Database Visibility

In Application Monitoring, a database called by an instrumented node is considered a remote service. You can get a significant amount of information on the interaction between the application node and database, but not from the database server perspective. When using Database Visibility with Application Monitoring, you can drill down to detailed database performance information directly from application flow maps. For more information see [Access Database Visibility from Application Monitoring Views](#).

### Application Monitoring and Analytics

For those times when tracing application code does not provide enough clues to track down the cause of a problem, AppDynamics provides visibility into the transaction logs that can be correlated to specific business transaction requests. Log correlation visibility requires a license for both Transaction Analytics and Log Analytics. See [Business Transaction and Log Correlation](#).
Install App Server Agents

On this page:

- Agent Installation Quick Start
- Before Starting
- Use the Getting Started Wizard
- Agent Installation by Type
- Tier and Node Naming Guidelines
- App Agent Network Bandwidth Usage
- Agent License Considerations

Related pages:

- Agent and Controller Compatibility

Instrumenting an application adds the AppDynamics application agent (app agent) into the runtime process of the application. This page gives you an overview of installing agents in the application environment.

Agent Installation Quick Start

The Getting Started Wizard in the AppDynamics Controller walks you through the steps to download and configure an agent for your application. This section gives you an overview of how to use the wizard.

The wizard produces a fully configured agent, including a node identity. Accordingly, it is intended to be run for each application instance you want to monitor until you have the hang customizing the configuration manually.

Before Starting

Before installing an agent, make sure that:

- The agent supports your application environment. See App Server Supported Environments.
- You can access the application host with a user account that has sufficient privileges to install the agent and—for certain installation types—restart the application.
- The application host has network connectivity to the Controller. Proxies or firewalls on the network between the agent and Controller may require additional configuration.

Use the Getting Started Wizard

After verifying the requirements, follow the workflow as guided by the wizard:

1. Open the wizard from the home page in the controller UI by clicking Getting Started.
2. Enter the configuration values for the application instance as described in the wizard. You name the node and indicate the tier and business application to which it will belong. If you are not sure of the best values to use, you can use temporary names and change them later.
3. When finished, download the agent. How you download the agent varies by agent type. In most cases, the agent comes as a ZIP file that you extract and install in the startup routine of your server. For other types of agents, you may need to modify the instrumented source code, for example, by including the agent library. The wizard walks you through it for each agent type.
4. Install the agent on your app server.
5. Apply load to your application. If you are instrumenting a production application, this will happen with customer interaction. Otherwise, create some test load on your application. The agent instruments the application code and reports metrics back to the Controller.
6. View your application in the Application Dashboard. For example:
The wizard makes it easy to perform a basic installation of the agent with minimally required settings, such as the Controller host and port, SSL, application name, and tier name. For advanced options or more complicated scenarios, you need to perform a manual installation of the agent. For more information, see the agent-specific link in the following section.

Agent Installation by Type

For detailed installation information by agent type, see the following topics:

- Java Agent
- .NET Agent
- Node.js Agent
- PHP Agent
- Python Agent
- Serverless APM for AWS Lambda
- Apache Web Server Agent
- C/C++ Application Agent
- Go Language Agent
- IBM Integration Bus Agent
- Standalone Machine Agents

For automated deployment guidelines, see Controller Deployment.

Tier and Node Naming Guidelines

The maximum length of a tier name is 100 characters and the maximum length of a node name is 225 characters for Linux and 500 characters for all other operating systems. In your tier, node and application names, you should avoid certain special characters. The characters you can use are listed on the Tiers and Nodes page.

Generally, node names should be unique. However, nodes that reside on different tiers and different machines (hosts) can have duplicate node names.

Within a business application, node names should always be unique in the following use cases:

- If the nodes reside on the same tier, but on different machines
- If the nodes reside on the same machine, but on different tiers
- Node names and machine names must unique. When a node is registered to a controller, it is associated with the machine it is on, and cannot be moved to another machine without changing the node name.

If the nodes names are the same in the aforementioned use cases, the nodes will not register or report successfully.

Nodes on proxy-based agents can have duplicate names on the same tier and same machine.
To rename an application, see Business Applications.

For node naming conventions by agent type, see the installation page for that agent, such as Node.js Agent or PHP Agent.

App Agent Network Bandwidth Usage

The following guidelines can help you estimate how much bandwidth overhead will be added to your environment by deploying AppDynamics agents.

Keep in mind that the exact bandwidth required for a deployment varies greatly depending on the nature of your application, the agent configuration, and the AppDynamics features you use. The best way to determine the bandwidth overhead is to test the AppDynamics deployment in a staging environment that mirrors as closely as possible the live operating environment.

1. The approximate bandwidth used by a single Java Agent with the default configuration is five to eight kilobytes per second.
2. Scaling of additional agents is linear. That is, if the average bandwidth usage for an app agent in a given deployment is 5 kilobytes, adding 10 means that bandwidth usage will be 5 × 10, or 50 kilobytes.
3. While the average bandwidth used is five to eight kbytes per second, agents send data to the Controller in bursts rather than as a steady stream of data. When testing bandwidth usage, to determine the actual kbytes per second used by an agent, you need to observe and average out traffic over the course of at least several minutes.
4. When testing bandwidth usage in the environment, keep in mind that different types of tiers will generate a different amount of load. For instance, a database tier tends to generate more traffic between the agent and Controller than an application server tier. For the best possible estimate, the test should take this into account.

Agent License Considerations

For agent-based license units—including APM, database monitoring, and server monitoring—licenses are allocated to the first agents that register with the Controller up to the licensed limit. For example, with five agent licenses, the first five agents that connect to the Controller are licensed.

Agent licenses are not bound to a particular machine or application. Therefore, a transfer of an agent-based license can be done simply by shutting down the application that runs the licensed agent—uninstalling the agent if the application will need to be restarted—and starting up the new application with the newly installed agent. Once the agent disconnects, a license unit is freed for the second agent.

For application monitoring agents (Java, .NET, Node.js, and so on), a license validation cycle runs every five minutes. It causes the agents to connect and validate that available license units are not exceeded. Historical usage data is captured during this cycle and stored as five-minute usage data. Every hour, the five-minute usage data is rolled up in hour usage data, which includes data on license unit usage. The five-minute data is purged after a few hours.
Agent-to-Controller Connections

Connection Settings

The connection between the agent and Controller is a one-way connection initiated by the agent. Therefore, you only need to configure connection settings in the agent.

Provide the following information to connect an agent to the Controller:

- **Controller host**: The hostname of the Controller to connect to. Agents may connect directly to the Controller or through a proxy.
- **Controller port**: The port on which the Controller listens for agent traffic. Agents use port 443, which is the same port as the browser connection to the Controller UI.
- **Account name**: The name of the account listed in the Controller. A single tenant Controller has two accounts, a default account name, customer1, and an internal system account. For most connections, use the default account name.
- **Global Account name**: The Global Account name is used for certain connections, such as to the Events Service or from the Analytics Agent.
- **Account access key**: A unique key associated with the Controller account. See Finding Your Account and Access Key.
- **SSL enabled**: If the agent should connect using SSL.

If you downloaded the agent through the Agent Download Wizard in the Controller, the Controller host, port, and account settings are already configured for you.

To connect the agents in your environment to a SaaS Controller or an On-premises Controller through a proxy, configure the agent properties to use the host and port settings of the Controller. For an example, see Java Agent Configuration Properties for proxy settings descriptions.

To connect to SaaS Controllers from agents in environments that limit outbound connections, set firewall rules that permit access to AppDynamics SaaS platform components. For a list of SaaS IP addresses, see SaaS Domains and IP Ranges.

Finding Your Account Name and Access Key

To configure the agent manually rather than through the Agent Download Wizard, set the Controller host and port, account name, and account access key settings manually. If you have an Admin account, you can find your account name and access key in the AppDynamics Controller UI. If you are not an admin, ask your administrator for your access key.

To view your account name and default access key:

1. Log in to the Controller UI as a user with view license permissions.
2. Click the gear icon (⚙️) and choose License.
3. Click on the Account tab. The account name appears next to the Name label. The account name is customer1 in a single tenant, on-premises Controller, the account name varies on SaaS Controllers.
4. Click Show next to the Access Key label to reveal the account default access key setting for this instance. Use this value as the Account access key setting in agent settings.
Securing the Connection

The .NET Agent uses the settings in the container to negotiate the SSL protocol with the Controller. Normally you do not need to configure the security protocol for the .NET Agent.

For the Java Agent, see the Agent and Controller Compatibility for a list of the default security protocols for different versions of the Java Agent.

If the default security protocol for your version of the Java Agent is incompatible with the Controller or an intervening proxy, pass the <appdynamics.agent.ssl.protocol> system property to set the protocol to one of the following security protocols:

- SSL
- TLS
- TLSv1.2
- TLSv1.1

Use the following format:

```
java -javaagent:<agent_home>/javaagent.jar ...
-Dappdynamics.agent.ssl.protocol <protocol> ...
```

Controller-specific security considerations vary between SaaS and on-premises Controllers, as described below.

On-premises Controller Secure Connections

An on-premises Controller has both an active secure (HTTPS) port and an HTTP port. Agents can use either port to connect to the Controller. By default, the certificate used for the secure connection is a self-signed certificate. The .NET agents cannot connect on a secure port that uses a self-signed certificate, so you will need to apply your own certificate to the port. App Agents connecting to an AppDynamics SaaS Controller must use an HTTPS connection.

To implement SSL for the Controller-agent connection:

- Set the application server primary port to the SSL port, 8181 by default. See Port Settings.
- Install a trusted certificate, see Controller SSL and Certificates.

SaaS Controller Secure Connections

SaaS Controllers require the use of SSL. Therefore, you only need to enable SSL in the configuration settings for your agents and connect them to the secure Controller port, 443. For more information, see App Agent Security and Machine Agent Security.

Using AWS PrivateLink with a SaaS Controller
Some customers may have a policy where they do not want agent traffic exposed to the public internet even though the agent traffic is encrypted. AppDynamics customers can leverage Amazon Web Services (AWS) inter-region PrivateLink to privately connect their cloud-based applications between Virtual Private Clouds (VPC) and AppDynamics SaaS and On-premises Controllers. In this scenario, all agent traffic flows across Amazon’s private network, even if VPCs reside in different AWS regions. Before you can ingest data from the AppDynamics endpoint to your AWS VPCs and AWS Accounts using AWS PrivateLink, you'll need to perform several set up steps. See How do I use AWS PrivateLink to connect to an AppDynamics SaaS Controller?

**App Agent Security**

To configure your agents for SSL, set these SSL-related properties:

- Set `controller-ssl-enabled` to `true`.
- Set the `controller-port` to the correct value for either on-premises or SaaS Controller.

In multi-tenant and SaaS environments, App Agents authenticate themselves to the Controller using the required account name and account access key values set in the connection properties configuration file.

**Standalone Machine Agent Security**

For information on the security settings related to the Machine Agent connection to the Controller, see Standalone Machine Agent Configuration Properties.

**Verify the Connection**

You can verify that an app agent is reporting to the Controller from the Tiers & Nodes list in the Controller UI. See Tiers and Nodes. You can also verify the connection from the AppDynamics Agents page, under the gear icon, see Manage App Agents.

In the Tiers & Nodes pages, the App Agent Status column indicates the status of the agent connection to the Controller. A green arrow icon indicates active connected agents, a red down arrow indicates an agent that has been previously recognized but is not currently connected.

If the agent is not reporting to the Controller, see troubleshooting information:

- Troubleshooting Java Agent Issues
- Resolve .NET Agent Installation and Configuration Issues
- Dynamic Language Agent Proxy
- Resolve PHP Agent Installation Issues

If traffic is not being properly correlated between tiers, make sure that any network components, such as load balancers or routers that sit between monitored nodes, are preserving the AppDynamics correlation header from HTTP traffic.

**Agent-Controller Communication Intervals**

Each AppDynamics agent has multiple communication channels for different purposes that initiate connections to the Controller independently, and at different time intervals.

- The agent configuration channel queries the Controller for any new configuration changes, and downloads these changes when available, every 60 seconds.
- The agent metric channel posts all new periodic metrics, including JMX, Windows performance counters, and business transaction metrics to the Controller every 60 seconds.
- If there are new business transactions that have not been seen before by the agent, they are posted to the Controller for registration every 10 seconds.
- If the agent has collected any new snapshots or events, they are posted to the Controller every 20 seconds.

**Information Sent to the Controller**

The following table shows the types of information that is collected by an application agent and sent to the Controller.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
</table>
Server Request GUID
A unique GUID identifying a request, known as a Business Transaction, in the form of xxxxxxxx–xxxx–xxxx–xxxx–xxxx–xxxx
where x is a lower-case hexadecimal digit.

Application Entry Points
Transaction entry points are identified among various frameworks and technologies. This includes Servlet URIs, Strut Action and Method name, Spring Bean Name and Method Name, JMS queue destination or listener name, Web Service/WCF action/operation name, PHP Virtual Name, and more.

Application Exit Points
Transaction exit points are identified among various frameworks and technologies. This includes: HTTP URL endpoints, JMS queue/destination, type, and vendor; database URL endpoint and vendor/version, web service Service Name, cache name and URL endpoint.

Performance Metrics
In general, for each monitored metric in AppDynamics, a response time, call rate, and error rate are collected. Each of these metrics also have an automatic baseline derived for each respective metric value.

Thread Stack Traces
Code level method execution metrics that comprise the application request are collected. This data includes the class and method that executed and the line number within the source code. Note that less than 5% of transactions will have stack traces collected. Errors/Exceptions and stack trace of error data will be collected.

SQL Query Values
If assigned with administrative permissions, SQL query variables within a query can be enabled, collected, and viewed. Note that the parameter data is collected for less than 5% of transactions. Configuration changes are logged in an audit log that is available for security review.

Data Collectors
If assigned with administrative permissions, data in the form of HTTP values or method payload can be collected and viewed. Note that a specific data collectors and code payload accessors require explicit configuration to be collected. Note, data is collected for less than 5% of transactions. Configuration changes are logged in an audit log that is available for security review.

Hostname
The DNS hostname of the machine (virtual/physical) from where the agent is installed and reporting monitoring data.

IP Address Internet Protocol (IP)
IP Address Internet Protocol (IP) address of the machine (virtual/physical) where the agent is sending monitoring data from.

CPU Usage
The value of CPU that is consumed on the monitored machine/virtual machine.

Memory Usage
The value of physical memory that is consumed on the monitored/virtual machine.

Network I/O Usage
The value of network I/O that is consumed on the monitored machine/virtual machine.

Disk I/O Usage
The value of disk I/O that is consumed on the monitored machine/virtual machine.

JVM Performance
JVM Heap Usage, JVM Memory Pools Settings, Garbage Collection performance, JVM System/Start-up Options, MBean metric values (for example, connection pool names and metric values, such as active connections, maximum connections, and so on)

Blitz Load Profile
Blitz is a horizontally scalable data processing platform for SaaS deployments. It collects metric data from agents, which it then aggregates and stores.

The 10M metrics/min Blitz load profile includes the following agents and churn information:

- Active load 10MM with 24K nodes. Total registered metrics is 40M.
- Node distribution:
  - DotNet = 3K
  - SIM Machine Agents OR Docker Containers = 30K
  - DBMon Data Collectors = 1K
  - Java Nodes = 16K
  - EUM Browser Apps = 60
- Java Node Churn: 1K/hr
- Either SIM or Docker Churn
  - Sim Node Churn = 40/hr (1% of 4K SIM nodes)
• Docker Churn = 200 containers/hr
• Node purger enabled with hard-limit of 4K and soft-limit of 10K
• SIM node purger enabled with a deletion max limit of 300/hr
Java Agent

On this page:

- Before You Begin
- Install the Agent
- Tier and Node Naming in the Java Getting Started Wizard

Related pages:

- AppDynamics Essentials
- Install the Machine Agent
- Administer App Server Agents

To monitor Java applications in the Controller, you need to install the AppDynamics Java Agent on each server that hosts applications to be monitored.

The following steps provide an overview of how to install the Java Agent using the Agent Download Wizard in the Controller. If you downloaded the agent from the AppDynamics Downloads Center, see Install the Java Agent.

Before You Begin

1. Verify support for your application environment at Java Supported Environments.
2. Confirm you have access to a compatible controller. See Agent and Controller Compatibility.
3. Confirm the connection settings to the Controller where your agent will report data:
   - If you use a SaaS Controller, AppDynamics sent you the Controller hostname in your Welcome Email. Use port 443 for HTTPS or port 80 for HTTP.
   - If you use an on-premises Controller, you supplied hostname and port at install time.
4. Verify you have access the machine where the application runs as a user account with privileges to install the agent software and restart the application.
5. Verify that the machine where the application runs can connect to the Controller. Proxies or firewalls on the network between the agent and Controller may require additional configuration.

Install the Agent

To install the agent using the Agent Download Wizard in the Controller UI:

1. Log in to the Controller UI and access the Getting Started Wizard for Java.
2. Follow the steps in the wizard to configure and download the agent. The agent guides you through some preliminary configuration steps.
3. As an administrator on the machine running your Java application, unzip the AppServerAgent.zip file.
   For example, on Linux unzip the agent to home/appdynamics. This is the <agent_home> directory.

```
unzip AppServerAgent.zip -d /opt/appdynamics/appagent
```

4. Add the Java Agent binary to the application process, typically by passing the agent JAR file as a startup argument to the application. For information by framework and application type, see the following pages:

When done, you are ready to restart your application and view data reported by the agent in the Controller UI. From there, you can install more agents or you can begin monitoring your application environment.

Tier and Node Naming in the Java Getting Started Wizard

Each monitored JVM must have a unique combination of tier and node name in AppDynamics. Naming the components in the wizard varies between the self-service trial edition of AppDynamics and the non-trial edition:
For the self-service Pro Trial edition of AppDynamics Pro, the wizard names the application and tier for you using the format described in the following section. You can always change the names later.

For a non-trial edition, you name the application and tier in the wizard, while the wizard generates the node name.

For self-service trials of AppDynamics Pro, the getting started wizard uses the following naming scheme to identify agents:

- **Node name:** `<app_server_type>@<hostname>:<port>`
  - **For example:** JBoss@appserver1.example.com:8080
- **Tier name:** MyTier
- **Application name:** MyApp

The port number in the node name ensures that each node has a unique name if there is more than one app server on the same machine.

If the application server listens on multiple ports, the node name uses the lowest of the port numbers. Notice that the lowest port number may not be the primary port for the host. For instance, if a server listens for client requests at 8080 but listens for shutdown requests on port 8005, the node will be named with the 8005 port.

The Controller identifies distinct tiers based on traffic flow between nodes. All tiers belong to a single business application, MyApp.
Java Supported Environments

This topic lists the application environments and versions supported by the AppDynamics Java Agent.

Java Agent Supported Platforms

In the following tables, note that:

- A dash (“-”) in a table cell indicates that this column is not relevant or not supported for that particular environment.
- In cases where no version is provided, assume that all versions are supported. Contact AppDynamics Sales for confirmation.
- For environments that require additional configuration, a separate table describing or linking to configuration information follows the support matrix.
- For environments supported by AppDynamics End User Monitoring, see Supported Environments and Versions - Web EUM.
- For environments supported by AppDynamics Server Visibility, Standalone Machine Agent Requirements and Supported Environments.

JVM Support

The AppDynamics Java Agent uses the standard JVM Tool Interface (JVMTI) mechanism allowing it to instrument any software running on a JVM supporting this mechanism.

AppDynamics certifies the successful operation of the basic mechanisms of instrumentation used by the agent on the following Java runtimes. These capabilities are supported on both JRE or full JDK installations.

Where the agent supports the following advanced memory monitoring features, they are listed for the JVM: Object Instance Tracking (OIT), Automatic Leak Detection (ALD), Content Inspection (CI), and Access Tracking (AT).

<table>
<thead>
<tr>
<th>JVM</th>
<th>OS</th>
<th>Memory Monitoring Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon Corretto 8, 11</td>
<td>Linux, Windows</td>
<td></td>
</tr>
<tr>
<td>Azul Zing 15.x</td>
<td>Linux x64</td>
<td>OIT, ALD</td>
</tr>
<tr>
<td>Azul Zulu 1.6, 1.7, 1.8, 9, 10, 11</td>
<td>Linux x64</td>
<td>OIT, ALD</td>
</tr>
<tr>
<td>JDK11 is supported from 4.5.6 onwards</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HP OpenVMS</td>
<td>Linux x64</td>
<td>OIT, ALD</td>
</tr>
<tr>
<td>JVM Language Frameworks Support</td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ALD, CI</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Object instance tracking, automatic leak detection, and custom memory structure monitoring are not supported with the AppDynamics IBM Java Agent. IBM JVMs can be instrumented with the AppDynamics Sun Java Agent to work around this limitation, however this only enables automatic leak detection and custom memory structure monitoring. Object instance tracking is not available. Working around this limitation can result in negative performance impact and is not recommended. In such cases, the IBM JVM needs to be restarted to enable custom memory structure monitoring.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Oracle Rockit JVM 28.1+</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Oracle/BEA JRockit 1.6</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Oracle/Sun JVM 1.6, 1.7, 1.8, 9, 10, 11</th>
</tr>
</thead>
<tbody>
<tr>
<td>JDK11 is supported from 4.5.6 onwards</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Oracle/Sun JVM 1.6, 1.7, 1.8, 9, 10, 11, 12</th>
</tr>
</thead>
<tbody>
<tr>
<td>JDK11 is supported from 4.5.6 onwards</td>
</tr>
<tr>
<td>JDK12 is supported from 4.5.11 onwards</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SAP JDK 6+</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>JDK11 is supported from 4.5.6 onwards</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>JDK12 is supported from 4.5.11 onwards</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Windows, Solaris, Linux, HP-UX, i5/OS, AIX</th>
</tr>
</thead>
</table>

**JVM Application Server and Framework Support**

AppDynamics supports the use of the Java Agent to instrument any application component running on a supported JVM, irrespective of how that component is built. The power of the AppDynamics platform is that it can automatically discover the topology and behavior of complex enterprise applications without requiring deep technical knowledge of the application's underlying code.

Frequently, Java-based systems employ standard framework code to implement business logic. Automatic instrumentation of framework code relies on knowledge of the business logic and programming patterns employed by the framework. AppDynamics instrumentation targets processing hand-offs between components, called `entry points` and `exit points`, either within the JVM or between JVMs. This includes hand-offs between frameworks in cases where multiple frameworks are being used together. This section covers the capabilities for frameworks for which AppDynamics provides automatic detection rules.

Monitoring application components built using frameworks not listed here may require custom configuration. The custom configuration may involve, for example, custom `POJO entry or exit points`. If you understand how the application behaves internally, you can easily configure this type of instrumentation. For more complex configuration tasks, contact your account representative to discuss how to engage the AppDynamics customer success organization.
No additional configuration is required for these frameworks.

<table>
<thead>
<tr>
<th>Vendor</th>
<th>JVM Language Framework</th>
<th>Version</th>
<th>Correlation/Entry Points</th>
<th>Exit Points</th>
<th>Transports</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open Source</td>
<td>Akka Actor</td>
<td>2.1 – 2.5.x</td>
<td>Yes</td>
<td>Yes</td>
<td>Netty</td>
<td>4.3.1 required for 2.4.x 2.5x support includes Persistence Remoting exit/entry supported</td>
</tr>
<tr>
<td>Open Source</td>
<td>Akka HTTP</td>
<td>Akka Actor 2.5.x</td>
<td>Akka HTTP upto 10.1.5 Scala 2.11, 2.12</td>
<td>Yes</td>
<td>Yes</td>
<td>HTTP EUM is supported</td>
</tr>
<tr>
<td>Open Source</td>
<td>Groovy</td>
<td>-</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Open Source</td>
<td>Play for Scala</td>
<td>2.1 – 2.6</td>
<td>Yes</td>
<td>-</td>
<td>HTTP over Netty server Akka HTTP server</td>
<td>Includes framework specific entry and exit points Play EUM-APM correlation supported</td>
</tr>
<tr>
<td></td>
<td>Play for Java</td>
<td>Scala 2.11, 2.12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Open Source</td>
<td>Scala</td>
<td>2.11.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Open Source</td>
<td>Spray toolkit (Spray.io)</td>
<td>1.1.x 1.1.3</td>
<td>Yes</td>
<td>Yes</td>
<td>HTTP</td>
<td>Entry points are detected and configurable as servlet entry point and exit points as HTTP exits</td>
</tr>
<tr>
<td>Pivotal</td>
<td>Grails</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Java Frameworks Support**

The Java Agent supports these Java frameworks. Some require additional configuration as indicated in the Configuration Notes column.

<table>
<thead>
<tr>
<th>Vendor</th>
<th>Framework</th>
<th>Version</th>
<th>SOA protocol (WebServices)</th>
<th>Auto Naming</th>
<th>Entry Points</th>
<th>Exit Points</th>
<th>Detection</th>
<th>Configuration Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adobe</td>
<td>BlazeDS</td>
<td>-</td>
<td>HTTP and JMS adaptor</td>
<td>-</td>
<td>Yes</td>
<td></td>
<td>-</td>
<td>Example Message Queue Backend Configuration</td>
</tr>
<tr>
<td>Adobe</td>
<td>ColdFusion</td>
<td>8.x, 9.x</td>
<td></td>
<td>-</td>
<td>Yes</td>
<td>-</td>
<td>Configuration required for transaction discovery. See:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Java Business Transaction Detection</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Servlet Entry Points</td>
</tr>
<tr>
<td>Apache</td>
<td>Cassandra with Thrift framework</td>
<td>-</td>
<td></td>
<td>-</td>
<td>Yes</td>
<td>Yes</td>
<td>Apache Thrift Entry and Exit points are detected</td>
<td></td>
</tr>
<tr>
<td>Apache</td>
<td>Struts</td>
<td>1.x, 2.x</td>
<td></td>
<td>-</td>
<td>Yes</td>
<td></td>
<td>Struts Actions are detected as entry points;.struts invocation handler is instrumented</td>
<td>Struts Entry Points</td>
</tr>
<tr>
<td>Apache</td>
<td>Tapestry</td>
<td>5</td>
<td></td>
<td>-</td>
<td>Yes</td>
<td>-</td>
<td>Not by default. See:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Java Business Transaction Detection</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Servlet Entry Points</td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>-----------</td>
<td>---------</td>
<td>----------</td>
<td>------</td>
<td>-----------------------------------</td>
<td>---------------------</td>
<td>-------</td>
<td></td>
</tr>
<tr>
<td>Apache</td>
<td>Wicket</td>
<td>-</td>
<td>-</td>
<td>No</td>
<td>Yes</td>
<td>-</td>
<td>Not by default</td>
<td></td>
</tr>
<tr>
<td>Apple</td>
<td>WebObjects</td>
<td>5.4.3</td>
<td>HTTP</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
<td>Yes</td>
<td>See: App WebObjects Startup Settings</td>
</tr>
<tr>
<td>axonframework.org</td>
<td>Axon</td>
<td>2.x, 3.x</td>
<td>-</td>
<td>-</td>
<td>Commands on the Command Bus continue existing Business Transactions</td>
<td>Correlation for Distributed Command Bus on JGroups and for Spring Cloud Connector transport as an exit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Open Source</td>
<td>CometD</td>
<td>2.6</td>
<td>HTTP</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
<td>See also &quot;HTTP Exit Points&quot; on Java Backend Detection.</td>
</tr>
<tr>
<td>Eclipse</td>
<td>RCP (Rich Client Platform)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Google</td>
<td>Google Web Toolkit (GWT)</td>
<td>2.5.1</td>
<td>HTTP</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>JBoss</td>
<td>JBossWS Native Stack</td>
<td>4.x, 5.x</td>
<td>Native Stack</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IBM</td>
<td>IBM-BPM</td>
<td>8.5.7, 8.6</td>
<td>-</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>IBM-BPM Support</td>
<td></td>
</tr>
<tr>
<td>Open Source</td>
<td>Direct Web Remoting (DWR)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Open Source</td>
<td>Eclipse Vert.x Core</td>
<td>3.3.3-3.5.4</td>
<td>HTTP</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>EUM Correlation is supported</td>
</tr>
<tr>
<td>Open Source</td>
<td>Enterprise Java Beans (EJB)</td>
<td>2.x, 3.x</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
<td>EJB Entry Points</td>
</tr>
<tr>
<td>Open Source</td>
<td>Grails</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>-</td>
<td>Not by default</td>
<td></td>
</tr>
<tr>
<td>Open Source</td>
<td>Hibernate JMS Listeners</td>
<td>1.x</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Open Source</td>
<td>Java Abstract Windowing Toolkit (AWT)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Open Source</td>
<td>Java Server Faces (JSF)</td>
<td>1.x, 2.x</td>
<td>-</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
<td>Java Business Transaction Detection and Servlet Entry Points</td>
</tr>
<tr>
<td>Open Source</td>
<td>Java Server Pages</td>
<td>2.x</td>
<td>-</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>Servlet Entry Points</td>
</tr>
<tr>
<td>Open Source</td>
<td>Java Servlet API</td>
<td>2.x, 3.0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Open Source</td>
<td>Jersey</td>
<td>1.x, 2.x</td>
<td>REST, JAX-RS</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Not by default</td>
<td>JAX-RS Support and node properties:</td>
</tr>
<tr>
<td>Open Source</td>
<td>JRuby HTTP</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>-</td>
<td>Not by default</td>
<td>See:</td>
</tr>
</tbody>
</table>

Copyright © AppDynamics 2012-2019
### Application Servers

The Java Agent supports the following application servers. Some require additional configuration. Click the link on the server or OSGi Runtime for information about additional requirements or related configuration topics. The agent usually discovers application servers as an entry point.

<table>
<thead>
<tr>
<th>Vendor</th>
<th>Application Server / OSGi Runtime</th>
<th>Version</th>
<th>SOA Protocol</th>
<th>RMI Supported</th>
<th>JMX</th>
<th>Entry Points</th>
<th>Configuration Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adobe</td>
<td>Cold Fusion</td>
<td>8.x, 9.x</td>
<td>-</td>
<td>No</td>
<td>-</td>
<td>Yes</td>
<td>Requires configuration for transaction discovery; see Servlet Entry Points</td>
</tr>
<tr>
<td>Equinox</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>OSGi Infrastructure Configuration</td>
</tr>
<tr>
<td>Apache</td>
<td>Felix</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>OSGi Infrastructure Configuration</td>
</tr>
<tr>
<td>Apache</td>
<td>Sling</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>OSGi Infrastructure Configuration</td>
</tr>
<tr>
<td>Apache</td>
<td>Tomcat</td>
<td>5.x, 6.x, 7.x, 8.x, 9</td>
<td>-</td>
<td>Yes</td>
<td></td>
<td></td>
<td>Apache Tomcat Startup Settings</td>
</tr>
<tr>
<td>Apache</td>
<td>Resin</td>
<td>1.x - 4.x</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Resin Startup Settings</td>
</tr>
<tr>
<td>Eclipse</td>
<td>Jetty</td>
<td>6.x, 7.x, 8.x, 9x</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Jetty Startup Settings</td>
</tr>
<tr>
<td>IBM</td>
<td>InfoSphere</td>
<td>8.x</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>IBM WebSphere and InfoSphere Startup Settings</td>
</tr>
<tr>
<td>IBM</td>
<td>WebSphere</td>
<td>6.1, 7.x, 8.x, 9x</td>
<td>JAX-WS</td>
<td>Yes, detect and correlate</td>
<td>Yes for WebSphere Perm</td>
<td>IBM WebSphere and InfoSphere Startup Settings</td>
<td></td>
</tr>
<tr>
<td>Open Source</td>
<td>Literay Portal</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Open Source</td>
<td>JBoss Wildfly (formerly JBoss Application Server)</td>
<td>4.x to 14.x</td>
<td>-</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>JBoss and Wildfly Startup Settings</td>
</tr>
<tr>
<td>Sun/Oracle</td>
<td>GlassFish Enterprise Server</td>
<td>2.x</td>
<td>-</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
<td>GlassFish Startup Settings</td>
</tr>
<tr>
<td>Oracle</td>
<td>GlassFish Server and GlassFish Server Open Source Edition</td>
<td>3.x, 4.x</td>
<td>-</td>
<td>Yes for AMX</td>
<td>Yes</td>
<td></td>
<td>GlassFish Startup Settings</td>
</tr>
<tr>
<td>Oracle and BEA</td>
<td>WebLogic Server</td>
<td>9.x+</td>
<td>JAX-WS</td>
<td>Yes, detect and correlate for 10.x</td>
<td>Yes</td>
<td>Yes</td>
<td>Oracle WebLogic Startup Settings</td>
</tr>
<tr>
<td>Software AG</td>
<td>webMethods</td>
<td>9.5, 9.6</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>webMethods Startup Settings</td>
</tr>
<tr>
<td>Tibco</td>
<td>ActiveMatrix BusinessWorks Service Engine</td>
<td>5.x, 6.x</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>Tibco ActiveMatrix BusinessWorks Service Engine Settings</td>
</tr>
</tbody>
</table>
Servlet 3.x detection is not supported.

**PaaS Providers**

<table>
<thead>
<tr>
<th>PaaS Provider</th>
<th>Buildpack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pivotal Cloud Foundry</td>
<td>Java Buildpack 3.4 and higher</td>
</tr>
<tr>
<td></td>
<td>See Using AppDynamics with Java Applications on Pivotal Cloud for more information.</td>
</tr>
<tr>
<td>Red Hat Openshift 3</td>
<td>JBoss EAP 6.4 and WildFly 8.1 Docker images</td>
</tr>
<tr>
<td></td>
<td>For documentation and download information, see the AppDynamics Java APM Agent page on the Red Hat Customer Portal.</td>
</tr>
</tbody>
</table>

**Message Oriented Middleware Support**

The Java Agent supports the following message oriented middleware environments. Some require additional configuration as indicated in the Configuration Notes column. Message oriented middleware servers are usually found by the Java Agent as an entry point.

<table>
<thead>
<tr>
<th>Vendor</th>
<th>Messaging Server</th>
<th>Version</th>
<th>Protocol</th>
<th>Correlation/Entry Points</th>
<th>Exit Points</th>
<th>JMX</th>
<th>Configuration Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon</td>
<td>Simple Queue Service (SQS)</td>
<td>-</td>
<td>-</td>
<td>Yes (correlation only)</td>
<td>Yes</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Amazon</td>
<td>Simple Notification Service (SNS)</td>
<td>-</td>
<td>-</td>
<td>No</td>
<td>Yes</td>
<td>-</td>
<td>See &quot;Amazon Simple Notification Service Backends&quot; on Java Backend Detection</td>
</tr>
<tr>
<td>Apache</td>
<td>ActiveMQ</td>
<td>5.x+</td>
<td>JMS 1.x</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Apache</td>
<td>ActiveMQ</td>
<td>5.x+</td>
<td>STOMP</td>
<td>No</td>
<td>-</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Apache</td>
<td>ActiveMQ</td>
<td>5.8.x+</td>
<td>AMQP 1.0</td>
<td>No</td>
<td>-</td>
<td>Yes</td>
<td>Example Message Queue Backend Configuration</td>
</tr>
<tr>
<td>Apache</td>
<td>Axis</td>
<td>1.x, 2.x</td>
<td>JAX-WS</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
<td>Default exclude rules exist for Apache Axis, Axis2, and Axis Admin Servlets. See also &quot;Web Service Entry Points&quot; on Java Backend Detection.</td>
</tr>
<tr>
<td>Apache</td>
<td>Apache CXF</td>
<td>2.1</td>
<td>JAX-WS</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
<td>To enable correlation, set node property enable-soap-header-correlation=true.</td>
</tr>
<tr>
<td>Apache</td>
<td>Kafka</td>
<td>0.9.0.0 to 2.0.0</td>
<td>-</td>
<td>Yes</td>
<td>Yes</td>
<td>Kafka consumer entry points are disabled by default. Correlation is supported. See Apache Kafka Consumer Backends.</td>
<td></td>
</tr>
<tr>
<td>Apache</td>
<td>Synapse</td>
<td>2.1</td>
<td>HTTP</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
<td>To enable correlation, set node property enable-soap-header-correlation=true.</td>
</tr>
<tr>
<td>Fiorano</td>
<td>Fiorano MQ</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>IBM</td>
<td>IBM Web Application Server (WAS)</td>
<td>6.1+, 7.x</td>
<td>Embedded JMS</td>
<td>-</td>
<td>Yes</td>
<td>Example Message Queue Backend Configuration</td>
<td></td>
</tr>
<tr>
<td>IBM</td>
<td>IBM MQ (formerly IBM WebSphere MQ)</td>
<td>6+</td>
<td>JMS</td>
<td>Yes</td>
<td>Yes</td>
<td>Example Message Queue Backend Configuration</td>
<td></td>
</tr>
<tr>
<td>Mulesoft</td>
<td>Mule ESB</td>
<td>3.4, 3.6, 3.7, 3.8, 3.9, 4.1x</td>
<td>HTTP, JMS</td>
<td>Yes</td>
<td>Yes</td>
<td>Mule ESB Startup Settings</td>
<td></td>
</tr>
<tr>
<td>Open Source</td>
<td>Eclipse Vert.x verticles</td>
<td>3.3.x, 3.4.x, 3.5.0, 3.6.0</td>
<td>-</td>
<td>Yes (correlation only)</td>
<td>Yes</td>
<td>The Java Agent detects messaging exit calls between verticles.</td>
<td></td>
</tr>
<tr>
<td>Open Source</td>
<td>Open MQ</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Oracle</td>
<td>Java Message Service</td>
<td>2.0</td>
<td>JMS</td>
<td>Correlation of the listener is disabled by default</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Oracle</td>
<td>Oracle AQ</td>
<td>-</td>
<td>JMS</td>
<td>-</td>
<td>Yes</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>
### Oracle OSB deployed on WebLogic 12.2.1
- HTTP
- JMS
- Yes
- Yes
- OSB Support

<table>
<thead>
<tr>
<th>Oracle / BEA</th>
<th>WebLogic</th>
<th>9.x+</th>
<th>JMS 1.1</th>
<th>Yes</th>
<th>Yes</th>
<th>Yes</th>
<th>Yes</th>
<th>Oracle WebLogic Startup Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Progress</td>
<td>SonicMQ</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>See &quot;RabbitMQ Backends&quot; on Java Backend Detection</td>
</tr>
<tr>
<td>Pivotal</td>
<td>RabbitMQ</td>
<td>-</td>
<td>HTTP</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>See &quot;RabbitMQ Backends&quot; on Java Backend Detection</td>
</tr>
<tr>
<td>Rabbit</td>
<td>RabbitMQ Spring Client</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>See &quot;RabbitMQ Backends&quot; on Java Backend Detection</td>
</tr>
<tr>
<td>Red Hat</td>
<td>HornetQ (formerly JBoss Messaging and JBoss MQ)</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Spring Integration Support</td>
</tr>
<tr>
<td>Red Hat</td>
<td>JBoss A-MQ</td>
<td>4.x+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>Spring Integration Support</td>
</tr>
<tr>
<td>Spring</td>
<td>Spring Integration</td>
<td>2.2.0+, 4.0+</td>
<td>JMS</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>See also &quot;Java Message Service Backends&quot; on Java Backend Detection</td>
</tr>
<tr>
<td>WSO2</td>
<td>ESB</td>
<td>4.7.0</td>
<td>-</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>EUM Correlation is not supported</td>
</tr>
</tbody>
</table>

### JDBC Drivers and Database Servers Support

The Java Agent supports these JDBC driver and database server environments. AppDynamics can follow transactions using these drivers to the designated database.

<table>
<thead>
<tr>
<th>JDBC Vendor</th>
<th>Driver Version</th>
<th>Driver Type</th>
<th>Database Server</th>
<th>Database Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>10.9.1.0</td>
<td>Embedded or client</td>
<td>Derby</td>
<td>-</td>
</tr>
<tr>
<td>Apache</td>
<td>-</td>
<td>-</td>
<td>Cassandra</td>
<td>-</td>
</tr>
<tr>
<td>Progress</td>
<td>DataDirect</td>
<td>data connectivity for ODBC and JDBC driver access, data integration, and SaaS and cloud computing solutions</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>IBM</td>
<td>JDBC 3.0 version 3.57.82 or JDBC 4.0 version 4.7.85</td>
<td>DB2 Universal JDBC driver</td>
<td>DB2</td>
<td>9.x</td>
</tr>
<tr>
<td>IBM</td>
<td>JDBC 3.0 version 3.66.46 or JDBC 4.0 version 4.16.53</td>
<td>DB2 Universal JDBC driver</td>
<td>DB2</td>
<td>10.1</td>
</tr>
<tr>
<td>IBM</td>
<td>-</td>
<td>Type IV</td>
<td>Informix</td>
<td>-</td>
</tr>
<tr>
<td>Microsoft</td>
<td>4</td>
<td>Type II</td>
<td>MS SQL Server</td>
<td>2012</td>
</tr>
<tr>
<td>Oracle MySQL, MySQL Community</td>
<td>5.x</td>
<td>Type II, Type IV</td>
<td>MySQL</td>
<td>5.x</td>
</tr>
<tr>
<td>Oracle</td>
<td>RAC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Oracle</td>
<td>9.x</td>
<td>Type II, Type IV</td>
<td>Oracle Database</td>
<td>8i+</td>
</tr>
<tr>
<td>Open Source Postgres</td>
<td>42.2.5</td>
<td>Type IV</td>
<td>Postgres</td>
<td>8.x, 9.x, 11x</td>
</tr>
<tr>
<td>Sybase</td>
<td>jConnect</td>
<td>Type IV</td>
<td>Sybase</td>
<td>-</td>
</tr>
<tr>
<td>Teradata</td>
<td></td>
<td></td>
<td>Teradata</td>
<td>-</td>
</tr>
</tbody>
</table>

**Notes:**
- Type II is a C or OCI driver
- Type IV is a thin database client and is a pure Java driver

### NoSQL/Data Grids/Cache Servers Support

The Java Agent supports these NoSQL, data grids and cache server environments. Some require additional configuration. Click the link on the database, data grid or cache name in the following support matrix for information about additional configuration required or related configuration topics.
<table>
<thead>
<tr>
<th>Vendor</th>
<th>Database/Data Grid/Cache</th>
<th>Version</th>
<th>Correlation/Entry Points</th>
<th>JMX</th>
<th>Configuration Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon</td>
<td>DynamoDB</td>
<td>-</td>
<td>Exit Points</td>
<td>-</td>
<td>See &quot;Amazon Web Services&quot; on Java Backend Detection.</td>
</tr>
<tr>
<td>Amazon</td>
<td>Simple Storage Service (S3)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>&quot;Amazon Simple Storage Service Backends&quot; on Java Backend Detection.</td>
</tr>
<tr>
<td>Apache</td>
<td>Cassandra</td>
<td>1.x, 2.x</td>
<td>Correlation for Thrift drivers only</td>
<td>Yes</td>
<td>• &quot;Cassandra Backends&quot; on Java Backend Detection.</td>
</tr>
<tr>
<td>Apache</td>
<td>Lucene - Apache Solr</td>
<td>1.4.1</td>
<td>Entry Points</td>
<td>Yes</td>
<td>Apache Solr Startup Settings</td>
</tr>
<tr>
<td>JBoss</td>
<td>Cache TreeCache</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>JBoss Startup Settings</td>
</tr>
<tr>
<td>JBoss</td>
<td>Infinispan</td>
<td>5.3.0+</td>
<td>Correlation</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Open Source</td>
<td>Memcached</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Memcached Exit Points</td>
</tr>
<tr>
<td>Open Source</td>
<td>MongoDB</td>
<td>3.1</td>
<td>-</td>
<td>-</td>
<td>See &quot;MongoDB Backends&quot; on Java Backend Detection</td>
</tr>
<tr>
<td>Oracle</td>
<td>Coherence</td>
<td>3.7.1</td>
<td>Custom-Exit</td>
<td>Yes</td>
<td>Coherence Startup Settings</td>
</tr>
<tr>
<td>Red Hat</td>
<td>JBoss DataGrid</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>JBoss Startup Settings</td>
</tr>
<tr>
<td>Red Hat</td>
<td>JBoss Cache TreeCache</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Red Hat</td>
<td>JBoss Infinispan</td>
<td>5.3.0+</td>
<td>Correlation</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Terracotta</td>
<td>EhCache</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>EhCache Exit Points</td>
</tr>
</tbody>
</table>

**RPC/Web Services API/HTTP Client Support**

The Java Agent supports these RPC, web services or API framework types. Some require additional configuration as indicated in the Configuration Notes column.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>Apache CXF</td>
<td>2.1</td>
<td>JAX-WS</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>See &quot;HTTP Backends&quot; ava Backe Detection</td>
</tr>
<tr>
<td>Apache</td>
<td>Apache HTTP Client</td>
<td>-</td>
<td>HTTPClient (now in Apache HTTP Components)</td>
<td>Yes</td>
<td>Yes (correlation only)</td>
<td>Yes</td>
<td>-</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Apache</td>
<td>Apache Async HTTP Client</td>
<td>4.1.x</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Apache</td>
<td>Ribbon HTTP Client</td>
<td>2.1.0</td>
<td>HTTPClient</td>
<td>Yes</td>
<td>Yes (correlation) Entry - NA</td>
<td>Yes</td>
<td>NA</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Apache</td>
<td>Apache Thrift</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Binary Per Entry Poin Apache T</td>
</tr>
<tr>
<td>Eclipse</td>
<td>Jetty</td>
<td>8.x, 9.x</td>
<td>HTTP Client</td>
<td>Yes</td>
<td>Yes (correlation only)</td>
<td>Yes</td>
<td>(ART supported)</td>
<td>-</td>
<td>See &quot;HTTP Backends&quot; ava Backe Detection</td>
</tr>
<tr>
<td>IBM</td>
<td>WebSphere</td>
<td>6.x, 7.x, 8.x</td>
<td>JAX-RPC</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>IBM WebS and InfoSp Startup Se also see D configurat excludes WebSphere classes</td>
</tr>
<tr>
<td>IBM</td>
<td>Websphere</td>
<td>7.x, 8.x</td>
<td>IIOP</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>IBM WebSphere and InfoSphere Startup Settings excludes WebSphere classes</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-----------</td>
<td>----------</td>
<td>------</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>Open Source</td>
<td>java.net.Http</td>
<td>-</td>
<td>HTTP</td>
<td>Yes</td>
<td>-</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>See &quot;HTTP Backends&quot; ava Back Detection.</td>
</tr>
<tr>
<td>Open Source</td>
<td>HTTPClient</td>
<td>0.3-3</td>
<td>Oracle SOA and potentially others that embed this library</td>
<td>-</td>
<td>Correlation: Yes; Entry: No</td>
<td>Yes</td>
<td>-</td>
<td>Yes</td>
<td>Oracle WebLogic Startup Settings excludes WebLogic classes</td>
</tr>
<tr>
<td>Open Source</td>
<td>Grizzly</td>
<td>Grizzly Async HTTP Client (com.ning.http-client 1.8.x, 1.9.x, grizzly-http-client 1.1x)</td>
<td>HTTP</td>
<td>-</td>
<td>Correlation: Yes; Entry: No</td>
<td>Yes</td>
<td>-</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Oracle</td>
<td>GlassFish Metro</td>
<td>-</td>
<td>JAX-RPC</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Oracle</td>
<td>GlassFish Metro with Grails</td>
<td>-</td>
<td>JAX-RPC</td>
<td>-</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
<td>Not by Default</td>
<td></td>
</tr>
<tr>
<td>Oracle</td>
<td>Oracle Application Server</td>
<td>ORMI</td>
<td>-</td>
<td>no</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Oracle</td>
<td>WebLogic</td>
<td>10.x</td>
<td>T3, IIOP</td>
<td>Yes</td>
<td>Correlation: Yes; Entry: No</td>
<td>Yes</td>
<td>-</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Oracle</td>
<td>WebLogic</td>
<td>9.x, 10.x</td>
<td>JAX-RPC</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Oracle/Sun</td>
<td>Java</td>
<td>11</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Yes (ART supported)</td>
<td>-</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Oracle/Sun</td>
<td>Sun RMI</td>
<td>-</td>
<td>IIOP</td>
<td>-</td>
<td>Not by Default</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Oracle/Sun</td>
<td>Sun RMI</td>
<td>-</td>
<td>JRMP</td>
<td>-</td>
<td>No</td>
<td>Yes</td>
<td>host/port</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Red Hat</td>
<td>JBoss A-MQ</td>
<td>4.x+</td>
<td>RMI</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>JBoss and Wildfly Settings</td>
</tr>
<tr>
<td>Square</td>
<td>OkHttp</td>
<td>-</td>
<td>HTTP</td>
<td>Yes</td>
<td>Correlation: Yes; Entry: No</td>
<td>Yes</td>
<td>-</td>
<td>Synchronous only</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>Web Services</td>
<td>-</td>
<td>SOAP over HTTP</td>
<td>-</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
<td>Create Match Rules for Web Services</td>
</tr>
<tr>
<td>jersey.github.io</td>
<td>Reactive JAX-RS client API</td>
<td>2.25+</td>
<td>HTTP Client</td>
<td>Yes</td>
<td>Yes (correlation)</td>
<td>Entry – NA</td>
<td>Yes</td>
<td>NA</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Business Transaction Error Detection**

The Java Agent supports the following logging frameworks for business transaction error detection:

- Apache Log4j and Log4j 2
java.util.logging
- Simple Logging Facade for Java (SLF4J)
  Support for the following method has been added: `public void error(String format, Object... argArray)`
- Logback

To instrument other types of loggers, see Error Detection.
Install the Java Agent

On this page:

- Prepare to Install the Java Agent
- Java Agent Resource Overhead
- Download and Unzip the Java Agent Distribution
- Configure the Java Agent
- Load the Java Agent in a JVM
- Attaching the Java Agent to a Running JVM Process
- Verify Java Agent Installation

Related pages:

- Java Agent Configuration Properties
- Enable SSL for the Java Agent

To begin monitoring a Java application using AppDynamics, you install the AppDynamics Java Agent into the application JVM:

1. Download the agent distribution to the machine where your Java application runs.
2. Configure the Java Agent settings. This configuration section of this topic describes manual configuration. For more options, see Administer the Java Agent.
3. Add the agent to the JVM process.

Prepare to Install the Java Agent

Consider the following as you decide where and how to deploy the AppDynamics Java Agent:

- The user under which the JVM runs must have write privileges to the conf and logs directories in the Java Agent home. To achieve this, you can install the agent as the same user that owns the JVM or as an administrator on the host machine. You can restrict the remaining contents of the agent directory to read only access.
- If you want to secure communications between the Java Agent and your controller, see Enable SSL for the Java Agent.
- If the agent connects to the Controller via a local proxy, you must configure proxy settings for the agent. See "Proxy Properties for the Controller" in Java Agent Configuration Properties for more information.
- The Java Agent supports sharing a single agent between multiple JVMs running on the same machine. If you choose this deployment scenario, you will likely want to specify settings in a combination of system properties and the versioned configuration file. See Instrument Multiple JVMs on a Single Machine for example configurations.

The AppDynamics Java Agent is one type of bytecode injection (BCI) agent. To prevent unforeseen issues with other BCI agents, AppDynamics only supports environments running the AppDynamics Java Agent as the sole BCI agent on the JVM.

Java Agent Resource Overhead

The Java Agent typically adds between 0% to 2% additional CPU consumption.

However, certain factors can increase CPU overhead from the agent beyond 2%. These include the use of resource-intensive AppDynamics features, such as asynchronous transaction tracking. Very active environments or configuration settings that result in a high number of metrics or snapshots reported per minute can also affect agent resource consumption.

In all cases, AppDynamics recommends that you test the agent in a staging environment, and monitor resource consumption of your application to ensure that it remains within proper operating parameters.

If your application operates within a small margin of its existing memory resource allocation, you may choose to increase the allocation for the application. AppDynamics recommends allocating the following amounts of additional Heap and PermGen space to accommodate the agent:

- Maximum heap size (-Xmx): 100 MB in addition to the amount required by the application
- Maximum PermGen (permanent generation) heap size (-XX:MaxPermSize): 20 MB in addition to the amount required by the application
For network bandwidth consumption, see Administer App Server Agents.

Download and Unzip the Java Agent Distribution

You can get the agent from the Agent Download Wizard. If you have never installed an agent before, the wizard is a good place to start. The wizard populates the configuration file in the agent you download with Controller connection settings and identifying settings for the agent. After you download the agent, you can install it in the JVM, as described ...

Alternatively, you can download the agent manually, as follows:

1. Download the Java Agent ZIP file from AppDynamics Download Center.
2. Extract the ZIP file to the destination directory as the same user or administrator of the JVM. Note the following:
   - Extract the Java Agent to a directory that is outside of your container or application server runtime directories, such as to \usr\local\appdynamics\appagent.
   - All files should be readable by the user under which the JVM runs. The user must have write privileges to the conf and logs directories in the Java Agent home. One way to achieve this is to install the agent as the same user that owns the JVM or as an administrator on the host machine.
   - The application server's runtime directory should be writable by the Java Agent as well.

For information on the contents of the Java Agent home directory, see Java Agent Directory Structure.

Configure the Java Agent

If you downloaded the agent from the Agent Download Wizard in the Controller, you can jump ahead to the next section, as the agent is already configured.

To configure the settings manually (or verify the wizard settings):

1. Edit the versioned configuration file:
   
   `<agent_home>/<version_number>/conf/controller-info.xml`

   The controller-info.xml is one of several approaches available for supplying configuration settings. For others, see Java Agent Configuration Properties.

2. Modify the connection settings to the Controller:
   - controller-host: Set to the IP address or hostname of the Controller. If the agent needs to connect through a proxy, see “Proxy Properties for the Controller” in Java Agent Configuration Properties.
   - controller-port: Set to the primary listening port number on the Controller. By default:
     - For a SaaS Controller, use 80 for HTTP or 443 for HTTPS
     - For an on-premises Controller, use 8090 for HTTP or 8181 for HTTPS

3. Direct the agent to connect to the Controller by SSL (HTTPS) by setting the controller-ssl-enabled value to true. See Enable SSL for the Java Agent.

4. Identify the business application, tier, and node that this the monitored JVM belongs to in the AppDynamics application model using these settings:
   - application-name
   - tier-name
   - node-name

   In a self-service Trial edition of AppDynamics Pro, the agent uses a default naming scheme, see Java Agent. You can use automatic naming with a standard edition of AppDynamics Pro by adding the following property:
   
   `<auto-naming>true</auto-naming>`

5. If the agents connects to a SaaS Controller or other multi-tenant Controller, configure the Account Name. For all Controllers, configure the Account Access Key.
   - account-name
   - account-access-key

   This information is provided in the Welcome email from the AppDynamics Team when you acquired the Controller. For a multi-tenant on-premises Controller, you can find this information in
   
   `<controller_home>/initial_account_access_info.txt`

6. See Java Agent Configuration Properties and configure an additional properties required in your environment.

The following shows a controller-info.xml file with sample configuration values:
<controller-info>
    <controller-host>192.168.1.20</controller-host>
    <controller-port>8090</controller-port>
    <application-name>ACMEOnline</application-name>
    <tier-name>InventoryTier</tier-name>
    <node-name>Inventory1</node-name>
</controller-info>

Load the Java Agent in a JVM

After configuring the agent settings, you can add the agent to the JVM. The exact steps for doing so vary by framework. The general approach, however, involves specifying the agent as a javaagent argument to the startup command for the JVM.

The argument should indicate the location of the Java Agent JAR file:

```
-javaagent:<agent_home>/javaagent.jar
```

On Windows, include the drive letter in the path to the agent:

```
-javaagent:C:<agent_home>\javaagent.jar
```

Adding the javaagent to the startup script requires a restart of the JVM. If it's not possible to restart the JVM when you are installing the agent and modifying the JVM startup script, you can attach the agent dynamically to the running Java process, as described next.

See Agent Installation by Java Framework for more information on how to install Java Agent by Java framework or technology.

Attaching the Java Agent to a Running JVM Process

Attaching the agent to a running JVM allows you to install the Java Agent without requiring a JVM restart. This approach would normally be used alongside adding the javaagent argument to the JVM startup script, or some other persistent approach to ensure that the agent is loaded again at the next JVM restart. However, dynamic attachment allows you to install the agent when restarting the JVM is not possible or convenient.

Dynamic agent attachment works if:

- The JVM is version 1.6 or later.
- The JVM is an Oracle (HotSpot) JVMs (unavailable for IBM or JRockit JVMs).

Other considerations include:

- Do not attach the agent dynamically to an environment that is already instrumented (either by the AppDynamics Java Agent or another type of agent). Doing so can cause unforeseeable issues and errors.
- Attaching the AppDynamics Java Agent to a running environment will impact the performance of the application while the agent performs the class retransformation needed to instrument the application. The agent overhead will return to its normal operating level when it finishes the process, but it is important to consider the potential performance impact to production services.

To attach the agent to the JVM, follow these steps:

1. Determine the PID of the JVM to which you want to attach. For example, on Linux, use:

```
ps -A | grep java
```
On Windows, use:

```
jps -l
```

2. Run the following command, replacing the placeholders for the path to the `tools.jar` file in your JDK, path to the AppDynamics Java Agent home directory, and the JVM process ID with values appropriate for your environment:

```
java -Xbootclasspath/a:<path_to_jdk>/lib/tools.jar -jar
<agent_home>/javaagent.jar <jvm_process_id>
appdynamics.controller.hostName=<controller_hostname>,appdynamics.controller.port=<controller_port_no>,appdynamics.controller.ssl.enabled=false,appdynamics.agent.applicationName=<app_name>,appdynamics.agent.tierName=<agent_tier_name>,appdynamics.agent.nodeName=<agent_node_name>
```

Use the equivalent paths for Windows, including drive letter. The following shows an example with system output included:

```
[appduser@my_centos6 ~]$ ps -A | grep java
   6780 pts/1    00:00:04 java
[appduser@my_centos6 ~]$ java
-Xbootclasspath/a:/usr/java/jdk1.7.0_79/lib/tools.jar -jar
/home/appduser/appagent/javaagent.jar 6780
Attaching to VM [6780]
agent path >>>/home/appduser/appagent/javaagent.jar
```

### Verify Java Agent Installation

After a installation, the agent log in `<agent_home>/logs` will contain the following message:

```
Started AppDynamics Java Agent Successfully
```

If the agent log file is not present, the Java Agent may not be accessing the `javaagent` command properties. To troubleshoot, check the application server log file where STDOUT is logged. It will have the fallback log messages, useful for troubleshooting the agent.

Also, verify that the agent is able to connect to the Controller in the Controller UI. To verify, log in to the Controller UI and click the Settings cog icon at the top right of the screen, and then **AppDynamics Agents**. In the list, look for the agent in the list by machine hostname.
Agent Installation by Java Framework

The following pages describe individual considerations and instructions for installing the Java Agent for some of the application servers supported for the Java Agent.

- Apache Cassandra Startup Settings
- Apache Tomcat Startup Settings
- Apple WebObjects Startup Settings
- Coherence Startup Settings
- GlassFish Startup Settings
- IBM WebSphere and InfoSphere Startup Settings
- JBoss and Wildfly Startup Settings
- Jetty Startup Settings
- Mule ESB Startup Settings
- Oracle WebLogic Startup Settings
- OSGI Infrastructure Configuration
- Resin Startup Settings
- Apache Solr Startup Settings
- Standalone JVM Startup Settings
- Tanuki Service Wrapper Settings
- Tibco ActiveMatrix BusinessWorks Service Engine Settings
- webMethods Startup Settings
- Java Security Manager Configuration
Apache Cassandra Startup Settings

On this page:
- Instrument Cassandra in a Windows Environment
- Instrument Cassandra in a Linux Environment

The Java Agent bootstraps using the javaagent command line option. Add this option to the cassandra (Linux) or cassandra.bat (Windows) file.

Instrument Cassandra in a Windows Environment

1. Open the apache-cassandra-x.x.x\bin\cassandra.bat file.
2. Add the Java Agent javaagent path to the JAVA_OPTS variable. Make sure to include the drive in the full path to the Java Agent directory.

   ```
   -javaagent:<agent_home>\javaagent.jar
   ```

   For example:

   ```
   set JAVA_OPTS=-ea
   -javaagent:C:\appdynamics\agent\javaagent.jar
   -javaagent:"%CASSANDRA_HOME%\lib\jamm-0.2.5.jar
   . . .
   ```

   3. Restart the Cassandra server. The Cassandra server must be restarted for the changes to take effect.

Instrument Cassandra in a Linux Environment

1. Open the apache-cassandra-x.x.x/bin/cassandra.in.sh file.
2. Add the javaagent argument at the top of the file:

   ```
   JVM_OPTS=-javaagent:<agent_home>/javaagent.jar
   ```

   For example:

   ```
   JVM_OPTS=-javaagent:/home/software/appdynamics/agent/javaagent.jar
   ```

3. Restart the Cassandra server for the changes to take effect.
Apache Tomcat Startup Settings

To instrument applications on Apache Tomcat, add the Java Agent JAR location as a Catalina environment, or CATALINA_OPTS variable, as described here.

Instrument Apache Tomcat

1. Open (or create if it doesn't already exist) the setenv.sh (Linux) or setenv.bat (Windows) file. For Tomcat 6 and later, you can put the file in the CATALINA_BASE/bin directory. For previous versions of Tomcat, put the file in the CATALINA_HOME/bin directory.
2. Add the -javaagent argument to the file as a Catalina environment variable, as follows:

   **On Linux:**
   ```
   export CATALINA_OPTS="$CATALINA_OPTS -javaagent:<agent_home>/javaagent.jar"
   ```

   Replace `<agent_home>` with the full path to the Java Agent JAR file.

   **For example:**
   ```
   export CATALINA_OPTS="$CATALINA_OPTS -javaagent:/home/appserver/appagent/javaagent.jar"
   ```

   **On Windows:**
   ```
   set CATALINA_OPTS=%CATALINA_OPTS%
   -javaagent:"Drive:<agent_home>\javaagent.jar"
   ```

   **For example**
   ```
   set CATALINA_OPTS=%CATALINA_OPTS%
   -javaagent:C:\appagent\javaagent.jar
   ```

3. Restart the application server. The application server must be restarted for the changes to take effect. For more information on running Tomcat as a service, see Setting Properties and Options on Startup.

Instrument Tomcat When Running as a Windows Service

When running Tomcat as a Windows service, add the javaagent argument to your Tomcat startup properties. These instructions apply to Apache Tomcat 6.x or later versions.

To install the Java agent in Tomcat running as a Windows service:

1. Ensure that you are using administrator privileges.
2. Run the Apache `tomcat<version>.exe` utility to configure your tomcat service to load the appdynamics agent (where `<version>` represents the major version number of the tomcat being instrumented).

   **For example**
3. Click the **Java** tab and in the **Java Options** add:

```
-javaagent:"<agent_home>\javaagent.jar"
```

4. Restart the Tomcat service to have the changes take effect.
Apple WebObjects Startup Settings

This topic helps you instrument applications written with WebObjects 5.4.3 on OSX 10.9 systems.

We will use one of the developer examples to illustrate how to instrument an application created with Apple WebObjects. After installing WebObjects, you can find most of the artifacts in the following directories:

```
/Developer/Examples/JavaWebObjects
/Developer/Applications/WebObjects
```

When you run the HelloWorld application at `/Developer/Examples/JavaWebObjects/HelloWorld`, a script file is generated:

```
/Developer/Examples/JavaWebObjects/HelloWorld/dist/legacy/HelloWorld.woa/HelloWorld
```

Open the generated script file for editing. Towards the end of the file, line 310 in the following example, appears the Java execute line:

```
304 #
305 # Launch the application.
306 #
307 echo $(SCRIPT_NAME).woa ...
308
309 echo $(JAVA_EXECUTABLE) $(JAVA_EXECUTABLE_ARGS) -classpath WBootstrap.jar com.webobjects.bootstr.WBootstrap $(COMMAND_LINE_ARGS)
310 eval exec $(JAVA_EXECUTABLE) $(JAVA_EXECUTABLE_ARGS) $classpath WBootstrap.jar com.webobjects.bootstr.WBootstrap $(COMMAND_LINE_ARGS)
```

Add the standard Java Agent arguments to the Java execution script for the HelloWorld application:

```
0707 echo $(SCRIPT_NAME).woa ...
0708
0709 echo $(JAVA_EXECUTABLE) $(JAVA_EXECUTABLE_ARGS) -classpath WBootstrap.jar com.webobjects.bootstr.WBootstrap $(COMMAND_LINE_ARGS)
0710 eval exec $(JAVA_EXECUTABLE) $(JAVA_EXECUTABLE_ARGS) $classpath WBootstrap.jar com.webobjects.bootstr.WBootstrap $(COMMAND_LINE_ARGS)
```

You can configure business transaction name using getter-chains. For more information, see

- Using Getter Chains
- See "Split by POJO Method Call" on Split Servlet Transaction by Payload Examples
Coherence Startup Settings

To add the javaagent command in Oracle Coherence:

1. In the `<coherence_home>/bin/cache-server.sh` file, update the following:

   ```bash
   $JAVAEXEC -server -showversion $JAVA_OPTS
   -javaagent:<agent_home>/javaagent.jar -cp
   "$COHERENCE_HOME/lib/coherence.jar"
   com.tangosol.net.DefaultCacheServer $1
   ```

2. Restart the application server to have the changes take effect.
GlassFish Startup Settings

On this page:
- Instrument Oracle GlassFish
- About Glassfish AMX Support

Oracle GlassFish is an OSGi-based application container. Accordingly, you need to set configure class bootloader settings as well as load the Java Agent in the runtime JVM, as described here.

Instrument Oracle GlassFish

1. Add the location of the Java Agent JAR file as a -javaagent JVM option in the GlassFish domain. You can add the agent using the asadmin tool as follows:

   For Windows:
   ```
glassfish4\bin\asadmin.bat create-jvm-options
'\-javaagent:<Drive_letter>:<agent_home>\javaagent.jar'
```

   For Linux:
   ```
glassfish4\bin\asadmin create-jvm-options
"-javaagent:/home/appduser/javaagent/javaagent.jar"
```

   Ensure you escape the colon character on Linux.

2. Configure the boot delegation entry for the AppDynamics package, as follows:
   - In GlassFish 3.x through 3.1.2, open the config.properties file located at `<glassfish_home>/glassfish/osgi/felix/conf` and add the following package prefix to the config.properties file:
   ```
org.osgi.framework.bootdelegation=com.singularity.*
```
   - In GlassFish 3.1.2 and later, in the boot delegation list in the `<glassfish_home>/glassfish/config/osgi.properties` file, add `com.singularity.*`

   For example:
   ```
eclipselink.bootdelegation=oracle.sql, oracle.sql.*
```

   # There is no need to use bootdelegation except for the following issues:
   # 1. EclipseLink
   # 4. NetBeans profiler packages exist in parent class loader (see issue #6112)
   # 5. MTrace exists in bootclasspath.
   # org.osgi.framework.bootdelegation=\{eclipselink.bootdelegation\}, \com.sun.brace, com.sun.btrace,*, \ore.orgnetbeans.lib.profiler, org.netbeans.lib.profiler.*, \com.singularity.*

   # The OSGi R4.2 spec says boot delegation uses the boot class loader by default. We need
   # 3. Restart the application server.

To verify the configuration, look at the domain.xml file located at `<glassfish_home>\domains\<domain_name>\config`. The domain.xml file should have an entry for the -javaagent option.

About Glassfish AMX Support

AppDynamics supports Glassfish AMX MBeans.

Set the boot-amx node property to enable AMX MBeans. See `boot-amx`.

You will see the AMX domain in the MBean Browser in the JMX tab of the node dashboard.
IBM WebSphere and InfoSphere Startup Settings

On this page:

- Security Requirements and Configuration
- Instrument WebSphere 7.x, 8.x, and 9.x or InfoSphere 8.x
- Instrument WebSphere 6.x
- Instrument WebSphere 5.x
- Verifying the Java Agent Configuration

The Java Agent passes the -javaagent argument command line option to WebSphere to ensure the App Server Agent has the correct permissions to monitor your app.

Security Requirements and Configuration

Full permissions are required for the agent to function correctly with WebSphere. Grant all permissions on both the server level and the profile level.

Running WebSphere with Security Enabled

If you want to run WebSphere while J2EE security or Global security is enabled, you need to make changes to WebSphere's server.policy file to prevent problems with the interaction between WebSphere and the Java Agent.

A codeBase value indicates the location of the source code. You must grant permissions to the code from that location. The codeBase is a URL value, and depends on the characters at the end. A codeBase with a trailing "/-" matches all files, both class and JAR files, in the directory and recursively all files in subdirectories contained in the directory.

1. Navigate to the server.policy file, located in <websphere_home>/properties or in <websphere_profile_home>/properties.
2. Add the following block to the WebSphere server.policy file:

   Syntax

   ```
   grant codeBase "file:/full/path/to/agent_install_directory/-" {
   permission java.security.AllPermission;
   }
   ```

   Example

   ```
   grant codeBase
   "file:/opt/appdynamics/javagent/agent4.5.1.23676/-" {
   permission java.security.AllPermission;
   }
   ```

3. Save the file.

Performance Monitoring Statistic

Under Application servers > <server_name> > Performance Monitoring Infrastructure (PMI), set a Currently monitored statistic set to an option other than None for the JMX functionality to work.

Instrument WebSphere 7.x, 8.x, and 9.x or InfoSphere 8.x

1. Log in to the administrative console for the WebSphere node where you want to install the App Server Agent.
2. In the administrative console select Servers > Server Types > WebSphere application servers.
3. Select the name of your server.
4. In the Server Infrastructure section, select **Java and Process Management > Process Definition**.

5. Under the Additional Properties section, select **Java Virtual Machine**.

6. Enter the javaagent option with the full path to the AppDynamics javaagent.jar file in the Generic JVM arguments field.
WebSphere uses Equinox as its OSGi container. In some cases you may also need to add the Java agent packages to the OSGi bootdelegation system property as follows:

```
-Dorg.osgi.framework.bootdelegation=META-INF.services,com.singulari
ty.*,com.ibm.*
```

Instrument WebSphere 6.x

1. Log in to the administrative console for the WebSphere node where you want to install the Java Agent.
2. In the left navigation tree, select Servers > Application servers.
3. Click the name of your server in the list of servers.
4. In the Configuration tab, select **Java and Process Management**.
5. Enter the javaagent option with the full path to the Java Agent javaagent.jar file in the Generic JVM arguments field.

   For Windows:
   ```
   -javaagent:<Drive Letter>:<agent install location>\javaagent.jar
   ```

   For Linux:
   ```
   -javaagent:<agent install location>/javaagent.jar
   ```

   For example:
   ![Diagram of AppDynamics Interface]

6. Click OK.

**Instrument WebSphere 5.x**

1. Log in to the administrative console of the WebSphere node where you want to install the App Server Agent.
2. In the administrative console, click Servers.
3. Click Application Servers.
4. Click the name of your server.
7. Enter the javaagent option with the full path to the Java Agent javaagent.jar file in the Generic JVM arguments field.
For Windows:

```
-javaagent:<Drive Letter>:<agent install location>/javaagent.jar
```

For Linux:

```
-javaagent:<agent install location>/javaagent.jar
```

8. Click **OK**.

**Verifying the Java Agent Configuration**

Verify the configuration settings by checking the server.xml file of the WebSphere node where you installed the Java Agent. The server.xml file should have this entry:

```
<jvmEntries ...
genericJvmArguments='-javaagent:E:\test1\AppServerAgent\javaagent.jar' disableJIT="false"/>
```
JBoss and Wildfly Startup Settings

On this page:

- Before You Install
- Domain Mode Agent Installation
- Troubleshooting JBoss Startup Issues
- Troubleshooting JBoss Shutdown Issues

This topic describes how to install the AppDynamics Java Agent on Red Hat JBoss Enterprise Application Server and JBoss Wildfly.

Before You Install

At a high level, to install the agent on JBoss or Wildfly, you add the Java Agent and log manager packages to the server startup routine.

The location you configure this varies depending on your version of the framework:

- In Linux, add the settings to standalone.conf or standalone.sh.
- In Windows, add the settings to standalone.conf.bat.
- If using JBoss 4.x or 5.x, add the configuration to run.sh for Linux or run.bat for Windows.

Standalone Mode Agent Installation

The following instructions assume the use of Red Hat JBoss (Application Server 7.x or Enterprise Application Server 6.x or later).

To install the Java Agent on JBoss Standalone:

1. Open the bin/standalone.conf file.
2. In the standalone.sh file, add the following javaagent argument.

   ```bash
   export JAVA_OPTS="${JAVA_OPTS}
   -javaagent:/agent_install_dir/javaagent.jar"
   ...
   while true;do
   if [ "x$LAUNCH_JBOSS_IN_BACKGROUND" = "X" ]; then
   # Execute the JVM in the foreground
   eval """JAVA" " -D""[Standalone]\""$JAVA_OPTS \""-Dorg.jboss.boot.log.file=$JBOSS_LOG_DIR/boot.log" \""-Dlogging.configuration=file:$JBOSS_CONFIG_DIR/logging.properties" \""-jar \""$JBOSS_HOME/jboss-modules.jar" \"
   -Djava.util.logging.manager=org.jboss.logmanager.LogManager
   -Xbootclasspath/p: <path_to_jboss_logmanager>/jboss-logmanager-<version>.jar"
   ```

3. Put the argument above the following section of standalone.sh

4. Restart the application server.
5. Add the following to the end of the standalone.conf file in the JAVA_OPTS section.

   ```bash
   -Djava.util.logging.manager=org.jboss.logmanager.LogManager
   -Xbootclasspath/p: <path_to_jboss_logmanager>/jboss-logmanager-<version>.jar
   ```

   **JDK9 and above**, the `-Xbootclasspath/p` option has been removed; use `-Xbootclasspath/a` instead.
5. `java.util.logging.manager=org.jboss.logmanager.LogManager`  
   `-Xbootclasspath/a:`  
   `<path_to_jboss_logmanager>/jboss-logmanager-<version>.jar`

6. Replace `<path_to_jboss_logmanager>` and `<version>` with the path and log manager JAR filename for your system. See [Making the LogManager Location Dynamic](#) for information on making the path dynamic.

**Dynamic LogManager Location**

On standalone JBoss instances, instead of hard coding the path and name of the log manager JAR, you can use glob pattern matching techniques to make the path to the log manager file dynamic, so that it is resilient to change or variances among systems.

The exact steps to accomplish this varies by environment. The following sections provide an example of this configuration on Windows and Linux systems, and are meant as a starting point for your own implementation.

**Windows**

In Windows, the standalone.conf.bat gets this additional snippet:

```bash
...  
rem jboss.modules.system.pkgs
set JAVA_OPTS=%JAVA_OPTS%
-Djboss.modules.system.pkgs=org.jboss.byteman,com.singularity,org.jboss.logmanager

rem java.util.logging
set JAVA_OPTS=%JAVA_OPTS%
-Djava.util.logging.manager=org.jboss.logmanager.LogManager

rem bootclasspath
set LOGMANAGER=
for /f %%i in ('dir /b "%JBOSS_HOME%\modules\system\layers\base\org\jboss\logmanager\main\jboss-logmanager-*.*.jar"') do (
set LOGMANAGER_JAR=%JBOSS_HOME%\modules\system\layers\base\org\jboss\logmanager\main\%%i
)
set JAVA_OPTS=%JAVA_OPTS% -Xbootclasspath/p:%LOGMANAGER_JAR%
```

The path to the LogManager JAR file under the JBoss home can vary by JBoss version. Be sure to check your system and adjust the path as shown in the example accordingly.

**Linux**

In Linux, you can populate the path dynamically with the following code:

```bash
...  
```

**Note**  
Step 5 and Step 6 are required only if the log manager errors manifest. Also, repeat Step 4 after this change.
JBoss Modules System PKGS
="org.jboss.byteman, com.singularity, org.jboss.logmanager"

JAVA_OPTS="$JAVA_OPTS
-Djava.util.logging.manager=org.jboss.logmanager.LogManager"
JAVA_OPTS="$JAVA_OPTS -Xbootclasspath/p:${(JBOSS_HOME)/modules/system/layers/base/org/jboss/logmanager/main/
jboss-logmanager-*.jar}"

If using the $(JBOSS_HOME) variable, as in the example, be sure to set the variable to the directory to the JBoss installation directory on your system.

The path to the LogManager JAR file under the JBoss home can vary by JBoss version. Be sure to check your system and adjust the path as shown in the example accordingly.

Domain Mode Agent Installation

For domain mode, the location in which you need to configure the settings depends upon specifics for your environment. Keep in mind that a domain is made up of these components:

- A domain controller, the administration and configuration server. The domain.xml configuration file is the global configuration for the managed hosts.
- Host controllers, which manage a particular host containing one or more application server nodes. There can be any number of host controllers and nodes. The hosts.xml file contains settings for the nodes on that host machine.

Where you put the configuration, therefore, varies as follows:

- domain.xml: Settings that can be identical for all hosts in the managed domain (i.e., the log manager and agent files are at the same location on all machines) can go into the domain.xml file for the Domain Controller.
- host.xml: Settings that need to be specialized for individual hosts (for example, if the paths to relevant files on hosts vary) need to go into the host.xml file.

You can add configuration settings to both domain.xml and host.xml, depending on which are global and which are host-specific. The following sections show an example of adding general settings to the domain configuration and node name setting to the host configuration.

Domain.xml Configuration

1. Locate and edit domain.xml for the domain. This is usually located under $JBOSS_HOME/domain/configuration/.
2. Find the the system-properties element and add a property named jboss.modules.system.pkgs with a value of com.singularity to the existing system properties. For example:
2. Under the server group name where you want to enable your agents, add the JVM options using the appropriate values for your agent location, JBoss application name, and tier name.

3. Changes to the `domain.xml` file require a restart of the management host to take effect. The changes are not propagated to server hosts until they are restarted as well.

**Host.xml Configuration**

For each host, specify the AppDynamics node name in the `host.xml` file, usually located under `$JBOSS_HOME/domain/configuration`.

Add the `-Dappdynamics.agent.nodeName` JVM option to specify the node name for this instance:
<servers>
  <server name="server-one" group="main-server-group">
    <jvm name="default">
      <jvm-options>
        <option value="-agentlib:jdwp=transport=dt_socket,address=8787,server=y,suspend=n"/>
        <option value="-Dappdynamics.agent.nodeName=JBOSS-EAP-NODE-1"/>
      </jvm-options>
    </jvm>
  </server>
  <server name="server-two" group="main-server-group" auto-start="true">
    <socket-bindings port-offset="150"/>
    <jvm name="default">
      <jvm-options>
        <option value="-Dappdynamics.agent.nodeName=JBOSS-EAP-NODE-2"/>
      </jvm-options>
    </jvm>
  </server>
  <server name="server-three" group="other-server-group" auto-start="false">
    <socket-bindings port-offset="250"/>
  </server>
</servers>

For brevity, comments have been removed from the sample.

**Troubleshooting JBoss Startup Issues**

Most issues installing the Java Agent on JBoss result from conflicts between startup arguments. That is, settings you add for the Java Agent may be overridden or conflict in other ways with existing arguments. Such issues are not always easy to detect.

The best way to begin troubleshoot such startup issues is to print and inspect the startup arguments that JBoss actually gets when attempting to start. To do this, view JBoss process information during startup using the following command. You need to issue this command while the start up attempt is occurring but before it fails.

```
ps -ef | grep [o]rg.jbossas | tr ' ' '
' | sed -e '/^$/d'
```

**Troubleshooting JBoss Shutdown Issues**

When enabling the JMXremote features of the JVM on a JBoss 5.1.2 server configuration, the following error is generated when calling the /bin/shutdown.sh
javax.management.JMRuntimeException: Failed to load MBeanServerBuilder class
org.jboss.system.server.jmx.MBeanServerBuilderImpl:
java.lang.ClassNotFoundException:
org.jboss.system.server.jmx.MBeanServerBuilderImpl

Resolution

Sometimes, there is JAVA_OPTS parameter in the environment, and it affects to a java process when the JVM start. From the reason that EAP uses JAVA_OPTS parameter internally, it should not be in the environment.

To clear the parameter in the environment, executing the following from a console:

1. export JAVA_OPTS=""
2. env|grep JAVA_OPTS
   JAVA_OPTS= <You will see this message>
3. shutdown.sh -s jnp://localhost:1099 -u USER -p PASSWORD

Root Cause

JMX related options should not be set in JAVA_OPTS when calling shutdown.sh. Ensure JAVA_OPTS is only set when starting JBoss EAP, not when running.
Jetty Startup Settings

On this page:
- Instrument Jetty Version 8.x or 9.x
- Instrument Jetty Version 6.x or 7.x
- Custom Exclude Rule for WebAppContext

The details for instrumenting the Jetty web server with the AppDynamics Java agent vary depending on the version of Jetty you are using, as described below.

Instrument Jetty Version 8.x or 9.x

You can instrument Jetty with the AppDynamics agent either from the server startup command or by editing the Jetty startup configuration file.

- To add the agent to Jetty at the command line, pass the javaagent argument with the fully qualified location of the Java agent JAR file when starting the Jetty server. For example:

  ```
  java -javaagent:/<agent_home>/javaagent.jar -jar start.jar
  ```

- To use the startup configuration file, edit the start.ini file in the Jetty base directory by adding the following lines:

  ```
  --exec
  -javaagent:/<agent_home>/javaagent.jar
  ```

  Be sure to specify the location of the AppDynamics javaagent.jar file in the javaagent argument as appropriate for your system. Restart the Jetty server after modifying the configuration file to have your changes take effect.

Instrument Jetty Version 6.x or 7.x

For Jetty version 6.x or 7.x, you can add the javaagent command line option to your jetty.sh file, as follows:

1. Open the jetty.sh start script file.
2. Add the following javaagent argument to the beginning of the script.

  ```
  java -javaagent:/<agent_home>/javaagent.jar
  ```

3. Save the script file.
4. Restart the application server for the changes to take effect.

Custom Exclude Rule for WebAppContext

For the Eclipse version of Jetty, to enable AppDynamics to detect business transactions based on web services, you need to create a Servlet exclude rule for the default Jetty servlet org.eclipse.jetty.webapp.WebAppContext.

This ensures that AppDynamics can detect business transactions based on the web services provided by your web applications while ignoring unmapped URL or URLs for the underlying framework.
To load the Java Agent in Mule ESB, pass the Java Agent JAR location as a JVM argument to Mule.

Mule ESB 3.X or later uses the Tanuki configuration environment. To specify JVM arguments in your Mule ESB environment, you need to configure them as additional parameters to the Tanuki Java Service Wrapper configuration file, wrapper.conf, as described below.

Configuring the Tanuki Service Wrapper

1. Open the Java Service Wrapper configuration file:
   `<MULE_HOME>/conf/wrapper.conf`
2. Find the location indicated for Java Additional Parameters:

   ```
   # Java Additional Parameters
   wrapper.java.additional.1=
   ```
3. Add the path to the Java Agent JAR file as a JVM argument using a `wrapper.java.additional.n` parameter, as follows.

   ```
   wrapper.java.additional.n="-javaagent:/path_to_appagent/javaagent.jar"
   wrapper.java.additional.n.stripquotes=TRUE
   ```

   Where "n" is the next available integer among the `wrapper.java.additional` parameters already in the wrapper.conf file, if any. The numbers serve to identify each Java Additional Parameter in the file. Do not skip numbers when adding the property. Replace `path_to_appagent` to the path to the `javaagent.jar` file in your system. The `stripquotes` parameter is necessary only if there are spaces in the path or filename, but is safe to include if not.

   For example, on a Linux system and with seven Java parameters already in the file, add the following properties:

   ```
   wrapper.java.additional.8="-javaagent:/opt/AppDynamics/Agent/app_agent/javaagent.jar"
   wrapper.java.additional.8.stripquotes=TRUE
   ```

   **Note**: There may be additional `wrapper.java.additional` properties defined in other Mule files. These are generated by the system. As indicated by comments preceding these properties, you should not make changes directly to the properties. Mule automatically auto-increments the index number for these properties based on the highest integer number used in `wrapper.conf`, so you do not need to modify or otherwise account for the index numbers of auto-generated configuration properties.
Oracle WebLogic Startup Settings

On this page:
- Instrument Oracle WebLogic for Windows
- Instrument Oracle WebLogic for an Application Running as a Windows Service
- Instrument Oracle WebLogic for Linux
- Instrument Clustered WebLogic Servers
- Java 2 Security Configuration

To enable the AppDynamics Java Agent add the javaagent command line option to the WebLogic startup file. If you start and stop clustered WebLogic servers using Node Manager, configure server startup in the WebLogic Server Administration Console.

Instrument Oracle WebLogic for Windows

1. Open the startWebLogic.cmd file, located at <weblogic_version_install_dir>/user_projects/domains/<domain_name>/bin.
2. Add following javaagent argument to the application server start script.

   ```
   set JAVA_OPTIONS=% JAVA_OPTIONS%
   -javaagent:"<drive>:\<agent_home>\javaagent.jar"
   ```

   - The javaagent argument must reference the full path of the agent installation directory, including the drive letter.
   - The command must precedes the WebLogic start commands, for example:

   ```
   @REM Enable the AppDynamics Java Agent
   set JAVA_OPTIONS=% JAVA_OPTIONS%
   -javaagent:"E:\AppDynamics\AppServerAgent\javaagent.jar"
   @REM AppDynamics Java Agent END
   @REM START WEBLOGIC
   echo starting weblogic with Java version:
   %JAVA_HOME%\bin\java %JAVA_VM% -version
   ```

3. Restart the application server. The application server must be restarted for the changes to take effect.

Instrument Oracle WebLogic for an Application Running as a Windows Service

Some applications have a pre-compiled startup method that installs WebLogic as a Windows service. Follow these steps to add the agent to the service.

1. Open the script file that starts the application service, such as install_XXXX_Server_Start_Win_Service.cmd.
2. Add the javaagent command before the line starting with "set CMDLINE=%JAVA_VM%..." such as
set CLASSPATH=%MYSERVER_CLASSPATH%;%PRE_CLASSPATH%;%WEBLOGIC_CLASSPATH%;%POST_CLASSPATH%;%WLP_POST_CLASSPATH%

set JAVA_VM=%JAVA_VM% %JAVA_DEBUG% %JAVA_PROFILE%

set WLS_DISPLAY_MODE=Production

@REM Enable the AppDynamics Java Agent
set JAVA_OPTIONS=% JAVA_OPTIONS% -javaagent:"<drive>:\<agent_home>\javaagent.jar"
@REM AppDynamics Agent END

set CMDLINE=%JAVA_VM% %MEM_ARGS% -classpath %CLASSPATH% %JAVA_OPTIONS% weblogic.Server"

3. Open a command prompt and run the following script to remove the existing Windows Service for your application:

```
install_XXXXX_Server_Start_Win_Service.cmd
XXXXX_xxxxx_Production_Server  R
```

4. Install the updated Windows Service for your application:

```
install_XXXXX_Server_Start_Win_Service.cmd
XXXXX_xxxxx_Production_Server  I
```

5. From the WebLogic web console, stop your application.
6. Start your application (which also starts WebLogic) from the Windows Services application, where the Windows service name = XXXX_xxxx_Production_Server.
7. Ensure that your application is working properly.
   For more information, see Creating a Server-Specific Script in the Oracle documentation.

**Instrument Oracle WebLogic for Linux**

1. Edit `startWebLogic.sh` located at `<weblogic_<version#>_install_dir>/user_projects/domains/<domain_name>/bin/startWebLogic.sh`.
2. Add the following to the beginning of your application server start script:

```
export JAVA_OPTIONS="$JAVA_OPTIONS -javaagent:/agent_home/javaagent.jar"
```

- The javaagent argument must reference the full path of the agent installation directory.
- The export command precedes the WebLogic start commands, for example:
# Enable the AppDynamics Java Agent

```bash
export JAVA_OPTIONS="\$JAVA_OPTIONS
-\javaagent:/opt/AppDynamics/AppServerAgent/javaagent.jar"

# START WEBLOGIC
 echo "starting weblogic with Java version:"
 $(JAVA_HOME)/bin/java $(JAVA_VM) \-version
```

3. Restart the application server to have the change take effect.

**Instrument Clustered WebLogic Servers**

For clustered WebLogic servers you start and stop using Node Manager, configure server startup in the WebLogic Server Administration Console.

1. Open the WebLogic Server Administration Console. See [Overview of the Administration Console](#).
2. Navigate to Environment > Servers and click your server in the Server List.
3. Click the Server Start tab.
4. Add the `javaagent` argument and set the value to the path to the Java Agent:

   ```
   -javaagent:/<agent_home>/javaagent.jar
   ```

5. Restart the application server.

**Java 2 Security Configuration**

If you have enabled Java 2 security on your WebLogic server, you need to add the following code block to the weblogic.policy file:

```java
grant codeBase "file:/<agent_home>/-"
{ permission java.security.AllPermission; };
```
OSGi Infrastructure Configuration

On this page:
- Configure Eclipse Equinox
- Configure Apache Sling
- Configure JIRA or Confluence
- Configure Other OSGi-based Containers

Related pages:
- GlassFish Startup Settings

The GlassFish application server versions 3.x and later uses OSGi architecture. By default, OSGi containers follow a specific model for bootstrap class delegation. Classes that are not specified in the container's CLASSPATH are not delegated to the bootstrap classloader; therefore, you must configure the OSGi containers for the Java Agent classes.

For more information, see GlassFish OSGi Configuration per Domain.

To ensure that the OSGi container identifies the Java Agent, specify the following package prefix:

```
org.osgi.framework.bootdelegation=com.singularity.*
```

This prefix follows the regular boot delegation model so that the Java Agent classes are visible.

If you already have existing boot delegations, add "com.singularity.*" to the existing path separated by a comma. For example:

```
```

Configure Eclipse Equinox

If running Eclipse Equinox under Glassfish:

1. Open the config.ini file located at <glassfish install directory>/glassfish/osgi/equinox/configuration.
2. Add the following package prefix to the config.ini file:

```
org.osgi.framework.bootdelegation=com.singularity.*
```

If running Eclipse Equinox under the WebSphere Application Server Liberty profile:

1. Open the bootstrap.properties file in the JVM directory <WLP_home>/usr/servers/<server_name> for editing.
2. Add the following line:

```
org.osgi.framework.bootdelegation=com.singularity.*
```

For more information see Getting Started with Equinox.

Configure Apache Sling

1. Open the sling.properties file. The location of the sling.properties file varies depending on the Java platform. In the Sun/Oracle implementation, the sling.properties file is located at <java.home>/lib.
2. Add following package prefix to the sling.properties file:

```
org.osgi.framework.bootdelegation=com.singularity.*
```

Configure JIRA or Confluence
For JIRA 5.1.8 and newer and Confluence 5.3 and newer:

1. Open the startup script (that is, catalina.sh) for editing.
2. Look for the `start` command block (look for "elif \[ "$1" = "start" \] ; then")
3. Add the following Java system property alongside the existing properties in both `else` blocks:

   ```java
   ```

4. Add the property to the run command block as well to instrument the application started in run mode. For example:

   ```bash
   if [ "$1" = "-security" ] then
     echo "Using Security Manager"
   fi
   ```

5. Before the comment line 'Execute The Requested Command', add the `-javaagent` argument to the file as a new Java option as follows:

   On Linux:
   ```bash
   JAVA_OPTS="$JAVA_OPTS -javaagent:<agent_home>/javaagent.jar"
   ```
   
   On Windows:
   ```bash
   set JAVA_OPTS=%JAVA_OPTS%
   -javaagent:"Drive:<agent_home>\javaagent.jar"
   ```

6. Restart the application.

**Configure Other OSGi-based Containers**

For other OSGi-based runtime containers, add the following package prefix to the appropriate OSGi configuration.

```java
file.org.osgi.framework.bootdelegation=com.singularity.*
```
Resin Startup Settings

On this page:
- Instrument Resin 1.x - 3.x for Windows
- Instrument Resin 1.x - 3.x for Linux
- Instrument Resin 4.x

The Java Agent bootstraps using the javaagent command line option. Add this option to the resin.sh or resin.bat file.

Instrument Resin 1.x - 3.x for Windows

1. Open the resin.bat file, located at <resin_home>/bin.
2. At the beginning of your application server start script, add the javaagent argument with the full path (including drive) to the javaagent.jar file on your system:

   ```
   exec JAVA_EXE -javaagent:"<drive>:\<agent_home>\javaagent.jar"
   ```

3. Restart the application server to have the changes take effect.

Instrument Resin 1.x - 3.x for Linux

1. Open the resin.sh file, located at <resin_home>/bin.
2. At the beginning of your application server start script, add the javaagent argument with the full path to the javaagent.jar file on your system:

   ```
   exec $JAVA_EXE -javaagent:"<agent_home>/javaagent.jar"
   ```

   For example:

   ```
   /mnt/resin-pro-4.0.18/bin/resin.sh -root @ export TERM=xterm-256color
   ```

3. Restart the application server. The application server must be restarted for the changes to take effect.

Instrument Resin 4.x

1. To install the Java Agent into Resin 4.X or later, edit the ./conf/resin.xml file and add:
2. Restart the application server. The application server must be restarted for the changes to take effect.

```xml
<jvm-arg>-Xmx512m</jvm-arg>
<jvm-arg>-javaagent:<agent_home>/javaagent.jar</jvm-arg>
```
Apache Solr Startup Settings

On this page:
- Instrument Solr in a Linux Environment
- Instrument Solr in a Windows Environment

This page describes how to instrument Apache Solr with the Java Agent.

Instrument Solr in a Linux Environment

1. Open the terminal.
2. Execute the following commands to add the javaagent argument to the Solr server:

   ```bash
   >cd $Solr_Installation_Directory
   >java -javaagent:"<agent install directory>/javaagent.jar" -jar
   start.jar
   ```

   For the argument value, reference the full path to the Java Agent installation directory.

Instrument Solr in a Windows Environment

1. Add the Java Agent to the Solr configuration file, solr.in.
   You can find the file at:
   `<solr-install-dir>\bin\solr.in`
2. Add the following line just below "REM set SOLR_OPTS=%SOLR_OPTS%":

   ```plaintext
   set SOLR_OPTS=%SOLR_OPTS% -javaagent:"<java_agent_location>\javaagent.jar"
   ```

3. Restart the Solr service.
Standalone JVM Startup Settings

AppDynamics works just as well with applications running outside of an application servers or container.

To install the agent on a standalone JVM, pass the path to the Java Agent when running the Java application in the following form:

```
java -javaagent:"/<agent_home>/javaagent.jar" <classname>
```

The javaagent argument value must reference the full path to the Java Agent home directory, including the drive on Windows. The classname should be the fully qualified class name with main method.

For example, the following is an example of the command on Windows systems:

```
java -javaagent:"C:\AppDynamics\agentDir\javaagent.jar"
com.main.HelloWorld
```

On Linux, it would be:

```
java -javaagent:"/mnt/AppDynamics/agentDir/javaagent.jar"
com.main.HelloWorld
```
The Java Agent bootstraps using the javaagent command line option. Add this option to the Tanuki Service wrapper.conf file.

**Configure the Tanuki Service Wrapper**

1. Open the wrapper.conf file for editing.
2. Add a `wrapper.java.additional` property that contains the javaagent option with the location of the Java Agent JAR file.
   For example:
   ```
   wrapper.java.additional.6=-javaagent:C://agent//javaagent.jar
   ```
With Tibco ActiveMatrix BusinessWorks Service Engine, each application is started in its own JVM. Therefore, for each application you want to monitor, you need to configure the Java agent in the startup settings for the configuration file of that application.

The file to configure is usually the TRA file named for the application. For example:

- `<application_name>.tra`

To instrument the Tibco application

1. Open the TRA file for the application you want to monitor and add the following to the file:

   ```
   java.extended.properties=-javaagent:/opt/appagent/javaagent.jar
   ```

   AppDynamics recommends you include all `-D` system property arguments in a single `java.extended.properties` line. Otherwise, the `-javaagent` property may be overwritten.

2. Restart the application to have the changes take effect using the command line tool `bwengine.sh` (or `exe`) or the Tibco Admin UI.

Since Tibco traffic commonly takes the form of incoming SOAP actions sent via HTTP requests to a single URL (i.e., `/BusinessServices/WebGateway`), it's likely that you will need to create a POJO split rule in the AppDynamics configuration to differentiate business transactions for Tibco. For information, see See “Split by POJO Method Call” on Split Servlet Transaction by Payload Examples.
webMethods Startup Settings

On this page:
- Instrumenting webMethods with the Tanuki Wrapper

Related pages:
- Tanuki Service Wrapper Settings

You can instrument the webMethods Integration Server or My webMethods Server by adding the Java agent to the startup script of the server.

The Java agent settings can go in the runtime.bat/runtime.sh file or the server.bat/server.sh file under the bin subdirectory of the server home.

Instrumenting webMethods with the Tanuki Wrapper

For webMethods servers that use the Tanuki Java service wrapper for start up, you need to configure the agent in the wrapper.conf file.

Note that the order of arguments in the file is important. Specifically, when instrumenting an OSGI-based platform, the agent configuration must precede the bootclasspath argument.

For example, the following listing shows the Java options used in a sample wrapper.conf file:

```
# Java Additional Parameters
...
wrapper.java.additional.4=-XX:MaxPermSize=256M
wrapper.java.additional.5=-javaagent:/opt/appd/appagent/javaagent.jar
wrapper.java.additional.6=-Xbootclasspath/a:"%OSGI_INSTALL_AREA%/lib/runtime/platform-jaasproxy.jar"
```
Java Security Manager Configuration

Running in a JVM with Security Enabled

If you want to enable a java security manager and instrument the JVM, you need to make changes to the active security policy file to prevent problems within the interaction between JVM and the Java Agent. The agent requires the change listed below. The policy file can be located by inspecting the value of the java.security.policy system property.

Add the following block to the JVM server.policy file:

```java
grant codeBase "file:/* AGENT_DEPLOYMENT_DIRECTORY */-"
{
    permission java.java security manager.AllPermission;
};
```
Automate Java Agent Deployment

**Related pages:**
- https://github.com/edmunds/cookbook-appdynamics
- Deploy Multiple Machine Agents From a Common Directory

This topic lists sample tools created by AppDynamics users for deploying Java Agents automatically. You can use these samples for ideas on how to automate AppDynamics agent deployment for your own environment.

In the samples, the agents are deployed independently of the application deployment:

- JavaExample1 uses a script, configuration file and package repository.

Click below to download the samples:

- ChefExample1.tar
- ChefExample2.tar
- JavaExample1.tar
Deploy Using Containers

On this page:

- Before You Start
- Use an Official AppDynamics Image
- Best Practices for Configuring App Agents in Containers

Related pages:

- Instrument JVMs in a Dynamic Environment
- Monitoring Applications in Docker Containers
- Monitoring Docker Containers
- Java Agent Configuration Properties
- Configure the Standalone Machine Agent
- Machine Agent Settings for Server Visibility

You can deploy and run AppDynamics App Agents inside Docker containers with no additional configuration. You can also use orchestration frameworks such as Kubernetes and Ansible to set up containers with the desired agents.

Before You Start

Note the following:

- AppDynamics recommends that you use Docker CE/EE v17.03 (or higher) or Docker Engine v1.13 (or higher).
- Before you deploy an agent, review the Best Practices for Configuring App Agents in Containers (below).

Use an Official AppDynamics Image

The quickest and easiest way to run a container with an App Agent installed is to use one of the official AppDynamics Images from the Docker Store. These images are produced by AppDynamics and based on certified base images from the Docker community. You can run these images directly or use them as base images for your own application containers.

If you prefer to build your base images, you can use the full source code on GitHub as a pattern for your own builds: https://github.com/Appdynamics/appdynamics-docker-images.

Best Practices for Configuring App Agents in Containers

You should set the following properties for App Agents running in containers. See Java Agent Configuration Properties for details of how to configure these properties.

- Reuse Node Name and Reuse Node Name Prefix – With these settings specified, the Controller reuses the node name after a container is shut down and marked as historical. See the Configuring the Agent for Dynamic Environments section.
- Agent Runtime Dir – The runtime directory where the agent stores agent logs and config files. This can be a shared volume (Docker) or persistent storage (Kubernetes), mounted into the container. Specifying a runtime directory outside the container ensures that agent logs are accessible after the container is stopped.
- Mark Node Name as Historical – This setting is important to ensure that an agent does not keep consuming a license after the node shuts down. See the Marking Dynamic Nodes as Historical at Shutdown section.

Example: Run the Java Agent using Docker from the Command Line
Redirecting Agent Output to Stdout

It is normally best practice to have all logging output redirected to stdout in containerized environments. To redirect the output of the AppDynamics APM agent to stdout, edit the log4j.xml configuration file and add the following line:

```xml
<appender-ref ref="ConsoleAppender"/>
```

under the `<logger name="com.singularity" additivity="false">`
Install the AppDynamics Site Extension for Java

You can use the Windows Azure Portal to add the AppDynamics Azure Site Extension to your Azure App Service web app. Azure Site Extension is used by Ops teams that may not have access to source files, or would not like to modify or recompile them, yet still want to monitor their Azure projects and solutions.

Prepare to Install

To install the AppDynamics for Windows Azure Site Extension, you need the following:

- Connection information for your AppDynamics Controller. See also Agent and Controller Compatibility.
- A Windows Azure account.
- An Azure web app to monitor.

If you are upgrading from a previous version, see Upgrade the AppDynamics Azure Site Extension.

Add the AppDynamics Azure Site Extension

Add the AppDynamics Azure Site extension as you would any site extension for any Azure web app.

1. Log in to the Windows Azure Portal.
2. Browse to your web app.
   - If you want to configure the Java agent using environment variables, add the environment variables before you install the AppDynamics Azure Site Extension. See configure the agent using environment variables.
3. From the DEVELOPMENT TOOLS list, click Extensions.
4. Click +Add to install the version of the AppDynamics Java Agent you want to add to your web app.
   - After you install the AppDynamics Azure Site Extension, it appears in the installed extensions list.

Configure the Controller Connection

You have the following options to configure the Java Agent to connect to the AppDynamics Controller:

- Configure the Controller connection using the Kudu console.
- Configure the Controller connection settings using environment variables.

For more detail, see Agent-to-Controller Connections.

Configure the Agent with the Kudu Console

When you add the AppDynamics Azure Site Extension to your web app, you can interactively configure the Java agent using the Kudu console.

1. Navigate to the AppDynamics Controller Configuration page in the Kudu Console:
   - http://(web app).scm.azurewebsitesJava/appdynamics/
   - For example: https://myazureexample.scm.azurewebsitesJava/appdynamics/
2. On the AppDynamics Controller Configuration page, enter your Controller connection information. For example:
AppDynamics Controller Configuration

Enter the following information to monitor your Azure application with AppDynamics. You should have received this from your sign up email or from your AppDynamics sales representative.

Controller Host
mycompany.seas.appdynamics.com

Port
443

Enable SSL

Account Name
MyAccount

Access Key
123dd45d-6fc7-8901-a234-567e0890bd12

Application Name
MyAzureApp

Validate

3. Click Validate to test the connection to the AppDynamics Controller and save your settings.
4. Restart your web app.
   After you apply some load to your web app, you can view it on flow maps in the AppDynamics Controller UI.

Configure the Agent Using Environment Variables

Configuring the Java agent using environment variables allows for unattended configuration. To configure agents in this manner, add the environment variables before you install the AppDynamics Azure Site Extension, as follows:

1. Navigate to SETTINGS > Application Settings for your web app.
2. Add the Java agent environment variables under App settings:
   - APPDYNAMICS_AGENT_ACCOUNT_ACCESS_KEY: A unique key associated with the Controller account
   - APPDYNAMICS_CONTROLLER_HOST_NAME: The hostname of the AppDynamics Controller
   - APPDYNAMICS_CONTROLLER_SSL_ENABLED: Set it to 'True' to enable SSL connection to the Controller. Otherwise set it to 'False'.
   - APPDYNAMICS_AGENT_ACCOUNT_NAME: The account name you use to log on to the Controller
   - APPDYNAMICS_AGENT_ACCOUNT_ACCESS_KEY: The account key you use to log on the Controller
   - APPDYNAMICS_AGENT_APPLICATION_NAME: the business application name in the Controller
   - APPDYNAMICS_AGENT_TIER_NAME: The tier name in the Controller
   - APPDYNAMICS_AGENT_NODE_NAME: The node name in the Controller
3. Restart your web app. After you apply some load to your web app, you can view it on flow maps in the AppDynamics Controller UI.

Upgrade the AppDynamics Azure Site Extension

When you click on the Extensions tab for your web app, the Microsoft Azure Portal displays the currently installed version of the AppDynamics Azure Site Extension. The Update Available column of the installed extensions list indicates if there is a more recent minor release of the Java Agent available. If so, you can click to update the extension from the list.

Upgrade a Major Version of the Java Agent

AppDynamics maintains major release versions of the Java Agent as separate site extensions. Therefore you need to uninstall the installed version of the AppDynamics Azure Site Extension before you upgrade to a new major release.
1. Log in to the Windows Azure Portal.
2. Stop your web app.
3. Click on the AppDynamics Azure Site Extension from the list of installed extensions and click Delete to uninstall it.
4. Install the new version of the AppDynamics Azure Site Extension as normal.
Instrument Java System Classes

The Java Agent excludes system classes like java.lang.* from instrumentation by default. To enable instrumentation for a system class, use an agent configuration property as described here:

Instrumented system classes add to the resource overhead introduced by the agent. The amount of overhead depends on the number of calls to the classes. AppDynamics recommends that you instrument a few nodes first and monitor the performance for these nodes before configuring all nodes in your system.

To instrument a Java system class:

1. Open the <agent_home>/conf/app-agent-config.xml file for the node where you want to enable instrumentation.
2. Add an <override-system-exclude> element with the fully qualified system class name to be instrumented in the <bci-processing-excludes> element. <override-system-exclude> element is used to whitelist classes (for example, 'com.sun.jersey') for instrumentation despite a blacklist ('com.sun.*').

For example:

```xml
<override-system-exclude filter-type="equals" filter-value="com.sun.jersey"/>
```

3. Restart the JVM for which you have modified the XML file.
   Once you restart the JVM, the package will be included in the instrumentation.
Instrument JVMs in a Dynamic Environment

Considerations for a Dynamic Environment

- Ensure that the agents can reach your Controller. Particularly if your Controller operates on-premises while the agents run on hosted servers, you need to ensure that the agents can access the Controller through the firewalls in your environment.
- Configure custom node expiration handling. By default, the Controller manages the lifecycle for a node based on default timeout settings. You will likely want to reduce the default expiration times for nodes, as described in Historical and Disconnected Nodes. To manage this directly, rather than relying on activity timeouts, a strategy would be to run a script in the JVM before it shuts down that invokes the mark-nodes-historical resource in the AppDynamics REST API to declare itself historical to the Controller.
- Depending on the nature of your environment, it may not make sense to track each JVM instance as a distinct node in AppDynamics, since the set of nodes in this case would be boundless. Also, since the JVMs are identical, representing them as different nodes in AppDynamics does not best reflect the logical model of the environment. The following section provides more information.

Configuring the Agent for Dynamic Environments

You can configure AppDynamics to reuse node names for these types of environments. With node name reuse enabled, after a node is shut down and marked as historical in AppDynamics, the name of the node becomes available for reuse in subsequently registered nodes. You can also specify a node name prefix when reusing node names to indicate the association between all concurrently running JVMs for the dynamic application.

To configure the agent for this environment, use the following properties:

- Reuse Node Name Property to enable re-use of node names, and
- Auto Node Name Prefix Property to set the prefix used for automatically-named nodes

When enabled, the Controller manages name assignments for the agents. It forms node names by appending a sequentially incremented number as the suffix to the prefix you specify. Be sure to avoid specifying the node name using any other mechanism (such as in controller-info.xml file or via a system property).

The following listing shows sample settings for the properties as they may appear in a startup script:

```
-Dappdynamics.agent.reuse.nodeName=true
-Dappdynamics.agent.reuse.nodeName.prefix=CloudActivator_
```

When you have enabled reuse node name, the Java Agent logs to standard output after starting up and before registration with the Controller. To troubleshoot agent registration failure, you may want the log files to be written to a file instead.

To configure the agent to write logs to a file, edit the log4j-unknown.xml at <agent_home>/<version_number>/conf/logging and uncomment the AgentLog Appender, as follows:
<!-- To log files to an "unknown" folder before the agent registers with the Controller, uncomment the line below -->
<appender-ref ref="AgentLogAppender"/>

This change causes the log files of an agent prior to registration to be written to a directory named "unknown" rather than standard output.

**Marking Dynamic Nodes as Historical at Shutdown**

As indicated in the Configuring the Agent for Dynamics Environments section, setting the node name reuse flag causes the JVM to report graceful shutdowns, allowing the controller to mark the corresponding node as historical. If you are not reusing node names, you can use one of the following alternative methods to have a node marked as historical:

- Set the following system property at JVM startup:

  ```
  -Dappdynamics.jvm.shutdown.mark.node.as.historical=true
  ```

- Set the following property in `app-agent-config.xml` to `true`, as shown:

  ```
  <app-agent-configuration>
  ...
  <configuration-properties>
  ...
  <property
  name="appdynamics.jvm.shutdown.mark.node.as.historical"
  value="true"/>
  ```

**Note**

If your application environment abortively terminates JVMs rather than allowing them to shut down gracefully, arrange for your orchestration framework to use the controller API (Configuration API > Mark Nodes as Historical) to mark the node as historical at the same time as it terminates the node.

**Controller-side Handling of Historical Nodes**

If a node has been out of contact with the Controller for a certain amount of time, the Controller marks the node as a historical node. The Controller suspends certain types of processing activities for the node, such as rule evaluation. For more information, see Historical and Disconnected Nodes.
Instrument JVMs in Restricted Environments

Some restricted environments do not allow any changes to the JVM startup script. For these environments, you can use the environmental variable JAVA_TOOLS_OPTIONS in a wrapper script that allows you to specify the initialization of tools, specifically, -javaagent.

To create the wrapper script:

1. Create a wrapper script in the following format:
   ```bash
   #!/bin/bash
   JAVA_TOOLS_OPTIONS="-javaagent:/opt/appdynamics/appserveragent/javaagent.jar -Dappdynamics.controller.hostName=<controller_host> -Dappdynamics.controller.ssl.enabled=false -Dappdynamics.controller.port=<port> -Dappdynamics.agent.applicationName=<application_name> -Dappdynamics.agent.tierName=<tier_name> -Dappdynamics.agent.nodeName=<node_name>"
   export JAVA_TOOLS_OPTIONS
   /path/to/original/script/startup.sh $*
   
   If you need to specify spaces for the application, tier, or node names, use backslashes and quotes around the value, for example:
   ```bash
   -Dappdynamics.agent.applicationName="$APPLICATION_NAME"
   ```
```

2. Add the original startup script to the wrapper script.
3. Make the new script executable, for example `chmod 750 wrapperscript.sh`.
4. Run your new wrapper startup script.

Full example: `startApplicationWithAppDynamics.sh`

```bash
#!/bin/bash
JAVA_TOOLS_OPTIONS="-javaagent:/opt/appdynamics/appserveragent/javaagent.jar -Dappdynamics.controller.hostName=<controller_host> -Dappdynamics.controller.ssl.enabled=false -Dappdynamics.controller.port=<port> -Dappdynamics.agent.applicationName=<application_name> -Dappdynamics.agent.tierName=<tier_name> -Dappdynamics.agent.nodeName=<node_name>"
export JAVA_TOOLS_OPTIONS
/path/to/original/script/startup.sh $*
```
Instrument JVMs Started by Batch or Cron Jobs

On this page:
- Configure the Java Agent
- Use the Script Name as the Node Name

You can configure the Java Agent for those JVMs that run as cron or batch jobs where the JVM runs only for the duration of the job.

**Configure the Java Agent**

1. Add the application and tier name to the controller-info.xml file.
2. Add the appdynamics.cron.vm property to the AppDynamics javaagent command in the startup script of your JVM process:

   ```
   -javaagent:<agent_home>/javaagent.jar
   -Dappdynamics.agent.nodeName=${NODE_NAME}
   -Dappdynamics.cron.vm=true
   ```

   The appdynamics.cron.vm property creates a delay between the end of the main method and the JVM exit so that the Agent has time to upload metrics to the Controller.

**Use the Script Name as the Node Name**

You can use the name of the script that executes a cron or batch job in the node name, as follows.

The following commands set the value of variable NODE_NAME using the combination of the script and host name. Add these commands to the startup script of the JVM.

```bash
# Use the name of the script (no path, no extension) as the name of
# the node.
NODE_NAME=sample
NODE_NAME="${NODE_NAME%.}
 echo $NODE_NAME
# Localize the script to the host.
NODE_NAME="$NODE_NAME@$HOSTNAME"
```
Use Environment Variables for Java Agent Settings

On this page:
- About Environment Variables
- Environment Variables Used by the Agent

As an alternative to using system properties or controller-info.xml to configure the agent, you can use environment variables. Environment variables allow agent identity settings to be populated with dynamic environments which may be unique for the JVM.

**About Environment Variables**

To configure the agent with environment variables, set the value of the environment variables described below. Environment variables exist for most of the agent settings you can configure in the controller-info.xml file, such as the application name, node name, and connection settings.

For example, set the node name for the agent using an environment variable using APPDYNAMICS_AGENT_NODE_NAME. On Linux, you would enter this command:

```plaintext
export APPDYNAMICS_AGENT_NODE_NAME=node23
```

When the agent subsequently starts up, the agent takes node23 as its node name. Typically the environment variable would be set as part of the start up routine of the JVM, and populated with values determined at startup.

**Environment Variables Used by the Agent**

The Java Agent takes configuration settings from the following environment variables, when set:

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Equivalent property</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPDYNAMICS_CONTROLLER_HOST_NAME</td>
<td>&lt;controller-host&gt;</td>
</tr>
<tr>
<td>APPDYNAMICS_CONTROLLER_PORT</td>
<td>&lt;controller-port&gt;</td>
</tr>
<tr>
<td>APPDYNAMICS_CONTROLLER_SSL_ENABLED</td>
<td>&lt;controller-ssl-enabled&gt;</td>
</tr>
<tr>
<td>APPDYNAMICS_AGENT_APPLICATION_NAME</td>
<td>&lt;application-name&gt;</td>
</tr>
<tr>
<td>APPDYNAMICS_AGENT_TIER_NAME</td>
<td>&lt;tier-name&gt;</td>
</tr>
<tr>
<td>APPDYNAMICS_AGENT_NODE_NAME</td>
<td>&lt;node-name&gt;</td>
</tr>
<tr>
<td>APPDYNAMICS_AGENT_ACCOUNT_NAME</td>
<td>&lt;account-name&gt;</td>
</tr>
<tr>
<td>APPDYNAMICS_AGENT_ACCOUNT_ACCESS_KEY</td>
<td>&lt;account-access-key&gt;</td>
</tr>
<tr>
<td>APPDYNAMICS_AGENT_UNIQUE_HOST_ID</td>
<td>Unique host name</td>
</tr>
<tr>
<td>APPDYNAMICS_AGENT_BASE_DIR</td>
<td>&lt;agent-runtime-dir&gt;</td>
</tr>
<tr>
<td>APPDYNAMICS_JAVA_AGENT_REUSE_NODE_NAME</td>
<td>-Dappdynamics.agent.reuse.nodeName</td>
</tr>
<tr>
<td>APPDYNAMICS_JAVA_AGENT_REUSE_NODE_NAME_PREFIX</td>
<td>-Dappdynamics.agent.reuse.nodeName.prefix</td>
</tr>
</tbody>
</table>

Notice that not all configurable agent settings are configurable through environment variables. For those settings, you need to use system properties or controller-info.xml, as described on Java Agent Configuration Properties.
Use System Properties for Java Agent Settings

On this page:
- Referencing System Properties
- Combining Properties

Related pages:
- Install the Java Agent
- Java Agent Configuration Properties

The Java Agent can accept configuration settings specified as system properties. This is useful when you have more than one JVM running on a machine and need to set node properties dynamically.

The JVMs may be in the same tier or on different tiers in the AppDynamics model. In either case:
- All of the common information should be configured using controller-info.xml.
- All of the information unique to a JVM should be configured using the system properties (-D option) in the startup script.
- Information in the startup scripts always overrides the information in the controller-info.xml file.

*Java Agent Configuration Properties* describes the system properties available for the Java Agent.

**Referencing System Properties**

You can reference system properties from the command line or in the agent configuration file to specify the node name, tier name, or other agent configuration properties dynamically.

For example, consider an Java application started with a script named startserver.sh. The script defines a system property as follows:

```
-Dserver.name=$1
```

The property `server.name` will get the value of the first argument passed when starting the script. For example, the following command to start the JVM would bind the `server.name` property to the value `ecommerce01`:

```
startserver.sh ecommerce01
```

You can use this value in your agent configuration by setting the `-Dappdynamics.agent.nodeName` property at startup as follows:

```
-Dappdynamics.agent.nodeName=$server.name
```

Or, you can use the value in the controller-info.xml file, using the following format:

```
${system_property_name}
```

For example:

```
<controller-info>
  ...
  <node-name>${server.name}</node-name>
</controller-info>
```
Note that `server.name` is simply the name of a sample system property used here to identify the node. Use an appropriate system property that is available in your environment for the Java agent node name.

To have tier names assigned dynamically as well, supply a dynamic value to the tier name property, as follows:

```
java -javaagent:<agent_home>/javaagent.jar
-Dappdynamics.agent.tierName=$tierName
-Dappdynamics.agent.nodeName=$nodeName
```

**Combining Properties**

You can combine multiple system properties to name nodes or tiers. For example, in controller-info.xml, combine properties as follows:

```
${system_property_name_1}${system_property_name_2}
```

You can combine system properties with literals. In the following example “_” and “inventory” are literals.

```
${myhost.name}_${myserver.name}.inventory
```

You can use existing system properties for the Controller host and port settings as well; however, combining properties as shown here is not supported for those settings.
Administer the Java Agent

On this page:
- Configuration Options and Precedence
- Choose an Effective Configuration Strategy
- System Properties Sample Configuration
- Agent Properties Sample Configuration
- controller-info.xml Sample File

The topics in this section describe how to maintain the AppDynamics Java Agent and how to tune its configuration for your requirements. These include, for example, how to upgrade the agent to a new version and how configure SSL for the agent.

Configuration Options and Precedence

You can control many aspects of the operation of the Java Agent through its configuration settings. The agent provides several approaches for specifying these settings, as listed below.

The order of the list matches the priority of the settings. The agent applies the first non-empty value for a configuration property it encounters, given the following order.

1. Environment variables. See Use Environment Variables for Java Agent Settings.
2. System properties passed in the start command for the JVM.

Not all properties are available as environment variables, system properties, or XML elements for the controller-config.xml. Property references include only the available configuration methods.

Choose an Effective Configuration Strategy

- Use versioned agent properties and versioned configuration files before using global agent properties and global configuration files. This minimizes any impact of configuration format changes in future agent releases.
- The agent only reads versioned agent properties and versioned configuration files from one versioned directory. During upgrades, manually migrate configurations from directories for previous versions. See Upgrade the Java Agent.
- The Agent Download Wizard automatically configures the agent using a versioned configuration file. The configuration in controller-info.xml statically defines the agent identity.
- For dynamic or elastic environments that require flexible agent identity, use an approach that provides for the dynamic node identification. For example, pass node identify or other agent configuration settings dynamically using environment variables or system properties.
- For shared binaries among multiple JVM instances, AppDynamics recommends you use a combination of configuration files and startup properties to configure the app agent. In this case, you configure properties common to all JVMs in the controller-info.xml file. Then specify the properties unique to each JVM using environment variables or system properties. See Instrument Multiple JVMs on a Single Machine for example configurations.
- For some properties, you can use system properties already defined in the startup script as the Java Agent property values. For more information, see Use System Properties for Java Agent Settings.

System Properties Sample Configuration

The following command demonstrates a startup script that includes system properties to start and configure the Java Agent. The application is "ACMEOnline", the tier is "Inventory", and the node is "Inventory1". SampleApplication is the application file.
java -javaagent:/home/appdynamics/agent/javaagent.jar
-Dappdynamics.controller.hostName=mycontroller.example.com
-Dappdynamics.controller.port=8090
-Dappdynamics.agent.applicationName=ACMEOnline
-Dappdynamics.agent.tierName=Inventory
-Dappdynamics.agent.nodeName=Inventory1 MyApplication.jar

System property values are case-sensitive.

Agent Properties Sample Configuration

The Java Agent doesn't include an agent.properties file by default. The file is a list of key-value pairs of system properties. For example:

appdynamics.controller.hostName=mycontroller.example.com
appdynamics.controller.port=8090

- The JVM treats properties from the agent.properties file the same as system properties. Values in agent.properties do not override any properties passed explicitly in the JVM startup script.
- You can also include system properties not specific to AppDynamics in the agent.properties file. Values from agent.properties do not override values passed in the JVM startup script.
- Do not include -D switch for keys in the agent.properties file.

controller-info.xml Sample File

The controller-info.xml file resides in <agent_home>/<version_number>/conf. Comments in the file from the agent distribution describe the settings. They have been removed in the following example for brevity.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<controller-info>
  <controller-host>192.168.1.20</controller-host>
  <controller-port>8090</controller-port>
  <controller-ssl-enabled>false</controller-ssl-enabled>
  <application-name>ACMEOnline</application-name>
  <tier-name>InventoryTier</tier-name>
  <node-name>Inventory1</node-name>
  <agent-runtime-dir></agent-runtime-dir>
  <enable-orchestration>false</enable-orchestration>
  <account-name>customer1</account-name>
  <account-access-key>341bf72e-7d7a-1234-b33d-9n712nn574</account-access-key>
  <force-agent-registration>false</force-agent-registration>
</controller-info>
```

For a complete list of configuration settings, see Java Agent Configuration Properties.
Java Agent Directory Structure

On this page:

- Java Agent Directory Contents
- Contents of the conf Directory

Unzipping the Java Agent distribution archive extracts the following files and directories to the current directory.

**Java Agent Directory Contents**

- **conf**: Common configuration files. The configuration files in this directory are used by an agent if a more specific version of the configuration file does not exist in the version specific conf directory, `ver<version_number>/conf`.
- **javaagent.jar**: The common JAR file used to bootstrap the Java agent. To enable the agent, pass the fully qualified location of this file as the `--javaagent` argument value to the JVM at startup.
- **readme.txt**: Instructions and notes for installing the Java agent.
- **utils**: Binary files for utilities shared across versions of the agent.
- **ver<version_number>**: The `javaagent.jar` file, configuration files, and other resources specific for this version of the Java agent. This directory is named to reflect the Java Agent version number, such as `ver4.4.0.5`. Among other things, the directory contains:
  - **conf**: The configuration files used by this version of the Java agent, including `controller-info.xml`, `app-agent-config.xml`, and more. These settings in these files take precedence over configuration file in the `<agent_home>/conf` directory.
  - **external-services**: Dynamically loaded modules that extend the agent, such as the Analytics Data Sources dynamic service.
  - **javaagent.jar**: A version-specific JAR file that serves as the Java agent binary. Do not use this Java agent JAR when configuring instrumenting directly. Use `javaagent.jar` in the root directory instead.
  - **lib**: Libraries that support the operation of this version of the Java agent.
  - **logs**: Log files written by the agent.
  - **sdk**: Samples, APIs, and Javadoc for extending the capabilities of the AppDynamics Java agent.
  - **utils**: Binary files for utilities that are specific to this version of the agent.

**Contents of the conf Directory**

The files located in this directory are commonly used for agent configuration and deployment.

- **transactions.xml**: Configuration settings for business transactions identified by the agent.
- **controller-info.xml**: Configuration settings that identify the node, tier, and application associated with the data that this agent reports to the AppDynamics Controller, along with connection settings for the Controller. For more information, see [Java Agent Configuration Properties](#).
- **app-agent-config.xml**: Local configuration settings for this agent. The settings in this file override any equivalent settings specified globally for the AppDynamics deployment. This file is typically used for short-term properties settings or for debugging agent issues.
- **jmx**: Files for configuring the JMX and Websphere PMI metrics.
- **logging/log4j.xml**: Flags to control logging levels for the agent. It is highly recommended not to change the default logging levels.
Java Agent Logging

On this page:
- Configure Appender Attributes
- Configure Log File Sizes and Rollovers
- Modifying the Log Directory Location
- Set the Agent Log Level
- Log to Syslog

Related pages:
- Agent Log Files
- Troubleshooting Java Agent Issues

By default, the AppDynamics Java Agent writes log files to the `<agent_home>/logs/<node_name>` directory. See Agent Log Files for information about how the logs are organized into sets that roll over.

**Configure Appender Attributes**

Each logger has one or more appenders. Each appender specifies where and in what format the data is being logged. For each appender, you can configure the following attributes.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Type</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>fileName</td>
<td>The name of the log file.</td>
<td>String</td>
<td>-</td>
</tr>
<tr>
<td>name</td>
<td>The name of the appender.</td>
<td>String</td>
<td>-</td>
</tr>
<tr>
<td>directory</td>
<td>The directory that the log file is saved to.</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>immediateFlush</td>
<td>Whether or not to immediately flush log data. Options: true, false</td>
<td>String</td>
<td>true</td>
</tr>
<tr>
<td>bufferSize</td>
<td>The size, in MB, of the event buffer. The buffer temporarily stores messages before they are written to disk.</td>
<td>String</td>
<td>256 KB</td>
</tr>
</tbody>
</table>

**Configure Log File Sizes and Rollovers**

You may want to limit the length of your agent log files to make troubleshooting easier. To do this, you can specify a maximum size for your log files. When a log reaches that size, it is then rolled over: the log file gets zipped, and a new blank log file is generated to store the next batch of log data. By default, a log file has a maximum size of 20 MB and can be rolled over four times.

You can configure log rollover attributes in the log4j2.xml file.

1. In log4j2.xml, find the ADRRAFAppender element that you want to configure rollover attributes for.
2. Modify the ADRolloverStrategy element and set values for the following attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>max</td>
<td>The maximum number of log files. When the maximum number of files is reached, the oldest log file after the initial log file is deleted.</td>
<td>String</td>
</tr>
<tr>
<td>compressionLevel</td>
<td>The degree of file compression, with 0 being uncompressed and 9 being the most compressed.</td>
<td>String</td>
</tr>
<tr>
<td>format</td>
<td>The file format that the log is saved in. Options: zip, gz</td>
<td>String</td>
</tr>
</tbody>
</table>

For example:
In the above example, "5" is the total number of log files with backups per appender type per set that we keep. In essence, each agent restart will create (number of appenders) files as a set. Then each file can grow up to "5" files total consisting of the first file and 4 backup log files, before it rolls over.

3. Modify the SizeBasedTriggeringPolicy element and set values for the following attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>The maximum log file size, in MB, before a log is rolled over. By default, the size is 20 MB before the rollover, because the logs are compressed.</td>
<td>String</td>
</tr>
</tbody>
</table>

**Modifying the Log Directory Location**

To specify a different log directory, use the following system property:

```
-Dappdynamics.agent.logs.dir
```

The default logging directory is `<agent_home>/ver<version_number>/logs`.

**Set the Agent Log Level**

The default logging level for most log files is INFO. Higher logging levels consume more disk space; you can change the logging level to WARN or ERROR to reduce the amount of logging. You can control the logging level for the Java Agent by changing the value of the “level value” parameter in the log4j.xml file in the versioned logging configuration file directory: `<agent_home>/ver<version_number>/conf/logging`. For example, to set the log level to DEBUG:

```xml
<AsyncLogger name="com.singularity" level="debug" additivity="false">
  <AppenderRef ref="Default"/>
  <AppenderRef ref="RESTAppender"/>
</AsyncLogger>
```

**Log to Syslog**

Instead of having the Java Agent write to the default log directory in the agent home directory, you can configure the agent to direct logging output to syslog. The agent supports syslog-based logging through log4j SyslogAppender.

To configure the agent to send logs to syslog:

1. Open the following configuration file in the agent home for editing:
   ```
   <agent_home>/ver<version_number>/conf/logging/log4j2.xml
   ```
2. Add the following section to the configuration file:
2. Configure the agent to redirect its logs to this appender. Find and replace this section of the file:

```xml
<Syslog name="Syslog" facility="LOCAL1" host="localhost" port="514" protocol="TCP">
    <PatternLayout pattern="[%t] %d{DATE} %5p %c - %m%n/>
</Syslog>
```

3. With the following:

```xml
<!-- to control the logging level of the agent log files, use the level attribute below.
value="all|trace|debug|info|warn|error"-->
<AsyncLogger name="com.singularity" level="info" additivity="false">
    <AppenderRef ref="Default"/>
    <AppenderRef ref="RESTAppender"/>
</AsyncLogger>
```

With the following:

```xml
<!-- to control the logging level of the agent log files, use the level attribute below.
value="all|trace|debug|info|warn|error"-->
<AsyncLogger name="com.singularity" level="info" additivity="false">
    <AppenderRef ref="SyslogAppender"/>
    <AppenderRef ref="RESTAppender"/>
</AsyncLogger>
```
Enable SSL for the Java Agent

On this page:
- Gather SSL Configuration Details
- Establish Trust for the Controller SSL Certificate
- Enable SSL between the Java Agent and the Controller
- Establish Trust for the Analytics Agent SSL Certificate
- Enable SSL between the Java Agent and the Analytics Agent
- Secure the Java Agent Truststore
- Keystore Certificate Extractor Utility
- Resolve SSL Issues

Related pages:
- Secure the Platform
- Controller SSL and Certificates
- Encrypt Agent Credentials
- Install the Java Agent

This topic covers how to secure communication between the AppDynamics Java Agent and the following AppDynamics components using SSL:

- AppDynamics Controller. Before you configure the agent to communicate with the Controller via SSL, you must either use a SaaS Controller or configure the on-premises Controller to use SSL.
- AppDynamics Analytics Agent. Before you configure the agent to communicate with the Analytics Agent via SSL, you must Enable SSL for the Analytics Agent.

The Java Agent supports extending and enforcing the SSL trust chain when in SSL mode.

Gather SSL Configuration Details

In preparation to secure Java Agent communications via SSL, you need information about the SSL configuration of the Controller or the Analytics Agent:

- The SSL port.
  - For SaaS Controllers the SSL port is 443.
  - For on-premises Controllers the default SSL port is 8181, but you may configure the Controller to listen for SSL on another port.
  - You configure the port for the Analytics Agent using the ad.dw.http.port property. See Enable SSL for the Analytics Agent.

- The signature method for the certificates:
  - A publicly known certificate authority (CA) signed the certificate. This applies for DigiCert, Verisign, Thawte, and other commercial CAs.
  - A CA internal to your organization signed the certificate. Some companies maintain internal certificate authorities to manage trust and encryption within their domain.
  - The Controller or Analytics agent uses a self-signed certificate.

Establish Trust for the Controller SSL Certificate

To establish trust between the Java Agent and the AppDynamics Controller, you must import the root certificate for the authority that signed the Controller's certificate into the agent truststore.

If you secured your on-premises Controller with a self-signed certificate, see Keystore Certificate Extractor Utility for instructions to create the agent truststore.

1. Obtain the root certificate for the authority that signed the certificate for the Controller.
2. Run the Java keytool command to import the root certificate. The command creates the keystore in the versioned agent configuration directory if it doesn’t exist:
keytool -import -alias rootCA -file
<root_certificate_file_name> -keystore
<agent_home>/<version_number>/conf/cacerts.jks -storepass
<truststore_password>

For example:

keytool -import -alias ControllerRootCA -file
/usr/home/appdynamics/DigicertGlobalRootCA.pem -keystore
/usr/local/appagent/4.3.0.0/conf/cacerts.jks -storepass
MySecurePassword

Make note of the truststore password, you need it to configure the Java Agent.

**Enable SSL between the Java Agent and the Controller**

Configure the following system properties in the versioned controller-info.xml: <agent_home>/<version_number>/conf/controller-info.xml. See "SSL Configuration Properties" on Java Agent Configuration Properties for full details on each property:

- **Controller Port**: the SSL port for the controller. 443 for AppDynamics SaaS.
  
  <controller-port>443</controller-port>

- **Controller SSL Enabled**: true.
  
  <controller-ssl-enabled>true</controller-ssl-enabled>

- **Controller Keystore Password**: the plain text password for the agent truststore.
  
  <controller-keystore-password>MySecurePassword</controller-keystore-password>

If you have enabled the Secure Credential Store, encrypt the password and enter it here. See Encrypt Agent Credentials.

- **Controller Keystore Filename**: path of the agent truststore relative to <agent_home>/<version>/conf. Required if you use a truststore other than the default <agent_home>/<version_number>/conf/cacerts.jks.
  
  <controller-keystore-filename>..//conf/cacerts.jks</controller-keystore-filename>

You can specify the Controller port and enable SSL for the Controller in the JVM startup script, but you must specify the truststore password and filename in the versioned controller-info.xml file.

Restart the JVM after you complete the configuration changes.

Sample controller-info.xml with SSL and Secure Credential Store encryption enabled
Establish Trust for the Analytics Agent SSL Certificate

To establish trust between the Java Agent and the Analytics Agent, you must import the root certificate for the authority that signed the Analytics Agent's certificate into the agent truststore.

1. Obtain the root certificate for the authority that signed the certificate for the Analytics Agent.
2. Run the Java keytool command to import the root certificate into the JRE truststore.

```
keytool -import -trustcacerts -alias analytics-agent -file <root_certificate_file_name> -keystore <agent_home>/<version_number>/conf/cacerts.jks
```

For example:

```
keytool -import -trustcacerts -alias analytics-agent -file /usr/home/appdynamics/MyAnalyticsCert.crt -keystore /usr/local/appagent/4.3.0.0/conf/cacerts.jks
```

Enable SSL between the Java Agent and the Analytics Agent
To enable the Java Agent to access the Analytics Agent over SSL, configure the `appdynamics.analytics.agent.url` system property for the JVM:

- Use the HTTPS protocol.
- Use the Analytics Agent SSL port. The port should be the same value as the Analytics Agent `ad.dw.http.port` property. See Enable SSL for the Analytics Agent.

```
-Dappdynamics.analytics.agent.url=https://<analytics_agent_host>:<analytics_agent_port>/v2/sinks/bt
```

For example:

```
java -javaagent:/home/appdynamics/agent/javaagent.jar
-Dappdynamics.controller.hostName=mycontroller.example.com
-Dappdynamics.controller.port=8090
-Dappdynamics.analytics.agent.url=https://my.analytics.host.example.com:9090/v2/sinks/bt
-Dappdynamics.agent.applicationName=ACMEOnline
-Dappdynamics.agent.tierName=Inventory
-Dappdynamics.agent.nodeName=Inventory1 ACMEOnline.jar
```

Restart the JVM after you complete the configuration changes.

**Secure the Java Agent Truststore**

Take the following security measures to prevent tampering with the Java Agent truststore:

- Secure the truststore file through filesystem permissions:
  
  - Make the agent truststore readable by any user.
  - Make the truststore owned by a privileged user.
  - Make the truststore writable only by the specified privileged user.

- Secure the agent configuration files so that they are only readable by the agent runtime user and only writable by a privileged user:
  
  - Versioned configuration file: `<agent_home>/<version_number>/conf/controller-info.xml`
  - Global configuration file: `<agent_home>/conf/controller-info.xml`

**Keystore Certificate Extractor Utility**

The Keystore Certificate Extractor Utility exports certificates from the Controller's Java keystore and writes them to an agent truststore. It installs to the following location:

```
<agent_home>/<version_number>/utils/keystorereader/kr.jar
```

To avoid copying the Controller keystore to an agent machine, you can run this utility from the Controller server. Access the agent distribution on the Controller at the following location:

```
<controller_home>/appserver/glassfish/domains/domain1/appagent
```

1. Execute `kr.jar` and pass the following parameters:

   - The full path to the Controller's keystore:
1. The truststore output file name. By default, the agent looks for a Java truststore file named cacerts.jks in the `<agent_home>/<version>/conf` directory in the agent home.

2. The password for the Controller's certificate, which defaults to "changeit". If you don't include a password, the extractor applies the password "changeit" to the output truststore.

   ```
   java -jar kr.jar
   <controller_home>/appserver/glassfish/domains/domain1/conf/keystore.jks cacerts.jks
   <controller_certificate_password>
   ```

2. Install the agent trust store to the versioned agent configuration directory:

   ```
   <agent_home>/<version_number>/conf/
   ```

**Resolve SSL Issues**

- If you run into problems with the version of TLS/SSL, see "SSL Compatibility between Java Agent and Controller" on Agent and Controller Compatibility and “AppDynamics Agent SSL Protocol Property” on Java Agent Configuration Properties.
Upgrade the Java Agent

To upgrade the AppDynamics Java Agent, you simply copy the existing agent directory to a backup location and replace it with the new agent directory. You then copy configuration file changes made in the old directory to the new agent directory and restart the application server.

This topic provides additional background information and step-by-step instructions.

**About the Upgrade**

Before you begin, review the Release Notes for changes that affect your environment. If you are upgrading both the Controller and agents, first upgrade the Controller and then upgrade the Java Agents.

Also, if upgrading multiple agents in your monitored environment, upgrade the agents for the tiers on which business transactions are originated last. For more information about this requirement, along with Controller and agent compatibility information, see Agent and Controller Compatibility.

Upgrading the agent requires a restart of the application server.

Before starting, download the latest version of the Java Agent for your JVM from the AppDynamics Download Center.

**Upgrade the Java Agent**

1. Shut down the application server where the Java Agent is installed.
2. Create a backup copy of the current agent installation directory.
3. Extract the Java Agent archive to a new directory.
4. Rename the existing agent directory.
5. Rename the new directory to the original name of the old agent directory. The new agent directory and its containing files should have the same directory path as the original one. Using the same directory path avoids the task of manually changing the agent-related configurations in your JVM startup script.
6. Copy controller-info.xml and, if necessary, agent.properties from the old agent config directory to the following location in the new directory:
   
   ```
   <agent_home>/<version>/conf
   ```
   
   For example:
   
   ```
   appd_javaagent/ver4.0.0.0/conf
   ```
   
7. If you previously made changes to the app-agent-config.xml or other configuration files, copy those changes to the new file as well.
8. Restart the application server.

When done you can archive the original directory and remove it from the application server home.
Uninstall the Java Agent

Related pages:
- Manage App Agents
- Install the Java Agent
- Java Agent Configuration Properties

If you delete an app agent from the Controller UI, as described in Manage App Agents, but do not shut down the JVM that the Java Agent runs on, the Java Agent will reappear in the UI the next time it connects to the Controller.

To prevent a Java Agent from connecting to the Controller, you need to remove the Java Agent settings from the JVM configuration. This frees the license associated with the agent in the Controller and makes it available for use by another app agent. This topic describes how to uninstall a Java Agent from the JVM configuration.

To uninstall the Java Agent

1. Stop the application server on which the Java Agent is configured.
2. Remove the -javaagent argument in the startup script of the JVM.
3. Remove any system properties configured for the Java Agent from the startup script of your JVM.
4. Restart the application server.
Tune Java Agent Performance

On this page:
- Viewing Agent Diagnostic Data
- Business Transaction Thresholds
- Snapshot Collection Thresholds
- Tuning Call Graph Settings
- Memory Monitoring
- Agent Heap Storage Monitoring and Shutdown

Related pages:
- Transaction Thresholds
- Call Graph Settings
- Java Memory Leaks

This topic discusses how to get the best performance from Java Agents. You can view performance statistics for deployed app agents in the diagnostic data window.

**Viewing Agent Diagnostic Data**

The agent diagnostic data shows you information about the performance of App Server agents. You can view the data from the Agents page in the Controller UI.

From the Node Dashboard of the node that contains the agent you want to view, click the subtab. Click View Diagnostics for all Agents to see stats for all agents.

**Business Transaction Thresholds**

AppDynamics determines whether transactions are slow, very slow, or stalled based on the thresholds for Business Transactions. AppDynamics recommends using standard deviation based dynamic thresholds. See Transaction Thresholds.

**Snapshot Collection Thresholds**

Snapshot collection thresholds determine when snapshots are collected for a Business Transaction. Too many snapshots can affect performance and therefore snapshot collection thresholds should be considered carefully in production or load testing scenarios. See Transaction Thresholds.

**Suggested Scheduling Settings**

- 10 minutes for deployments with less than 10 BTs
- 20 minutes for deployments from 10 to 50 BTs
- 30 minutes for deployments from 50 to 100 BTs
- 60 minutes for deployments larger than 100 BTs

**Suggested Diagnostic Session Settings**

- Settings for Slow requests (%value): 20 – 30
- Settings for Error requests (%value): 10 – 20
- Click Apply to all Business Transactions.

**Tuning Call Graph Settings**

You can tune call graph settings from the Call Graph Settings page, as described in Call Graph Settings.

**Suggested SQL Query Time and Parameters**

- Increase the Minimum SQL query time to 100 ms from the default of 10 ms.
- Enable Filter SQL Parameter Values.

**Memory Monitoring**

Memory monitoring features such as leak detection, object instance tracking, and custom memory should be enabled only for a specific
node or nodes when debugging a memory problem. Automatic leak detection is on-demand, and therefore, the leak detection will execute only for the specified duration.

When you observe periods of growth in the heap utilization (%), you should enable on-demand memory leak capture. See Java Memory Leaks.

Agent Heap Storage Monitoring and Shutdown

The Java Agent monitors the heap storage used by the instrumented application and shuts itself down when conditions indicate excessive utilization of Java heap storage by the application. The agent takes this measure not necessarily because it is likely to be the source of the memory pressure—in fact, it can help you diagnose the root cause of the issue—but as a final protective measure to help relieve the pressure on available memory for the application.

Typical reasons for excessive heap utilization include:

- Java heap space is too small compared to your application traffic and footprint
- Java heap memory leak
- A rogue thread is consuming a large amount of memory in a short amount of time

Java Agent monitoring of heap storage works as follows. The agent check heap utilization every 30 seconds. When it detects that heap utilization has exceeded 95%, the Java Agent checks for a garbage collection event. When garbage collection occurs, it checks heap utilization again. If usage still exceeds the threshold, the agent shuts itself down and logs the following agent event:

ERROR HeapShortageMonitor - Agent shutdown triggered because max heap usage percentage reached

If heap memory utilization exceeds the threshold, but garbage collection never occurs, the agent resumes normal monitoring after 30 seconds. If the heap usage remains above the threshold, the heap shortage monitor resumes checking for garbage collection, as it did previously.

You can use a Java Agent node property to control whether heap storage monitoring is enabled and the action that results from excessive heap utilization. The heap-storage-monitor-enabled controls whether it is enabled (true, by default), and the heap-storage-monitor-shutdown-action property determines the triggered action, between:

- true: If the heap threshold is exceeded, the agent is disabled.
- false: If the threshold is exceeded, a warning-level message is written to the agent log and warning event is sent to the Controller (of type AGENT_EVENT with the text: “AppDynamics agent has detected shortage of JVM Heap Storage”).
Troubleshooting Java Agent Issues

On this page:

- Resolving Java Agent Startup Issues
- Locating the Java Agent Log Files
- Resolving Incomplete Agent Configuration Issues
- Unblocking the Controller Port
- Correcting File Permission Issues
- Maximum Async Handoff Call Graph Samples Error

Related pages:

- Troubleshoot Controller Issues

This topic discusses techniques for troubleshooting agent installation and operation. In particular, it describes how to find and interpret information in agent log files.

Resolving Java Agent Startup Issues

The first thing the Java agent does upon application startup is register with the Controller. Once registered, the agent should appear in the Settings > AppDynamics Agents list.

If you do not see the agent in the list within a few minutes, check the following:

1. Make sure you have restarted the application server.
2. Verify that the javaagent argument has been added to the startup script of your JVM.
3. Verify that you configured the agent-controller communication properties and agent identification properties in the controller-info.xml file or as system properties in the startup script of your JVM. See Java Agent Configuration Properties.
4. Check the Agent logs directory located at <agent_home>/logs/<Node_Name> for the agent.log file.
5. Verify that the Agent is compatible with the Controller. For details see Agent and Controller Compatibility.

Locating the Java Agent Log Files

Agent log files are located in the <agent_home>/logs/<node_name> folder.

The agent.log file is the recommended file to help you with troubleshooting. This log can indicate the following:

- Incomplete information in your Agent configuration
- The Controller port is blocked
- Incorrect file permissions

Error messages related to starting the Java Agent use this format:

```
ERROR com.singularity.JavaAgent - Could Not Start Java Agent
```

Resolving Incomplete Agent Configuration Issues

The following table lists the typical error messages for incomplete Agent configuration:

<table>
<thead>
<tr>
<th>Error Message</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cannot connect to the Agent - ERROR com.singularity.XMLConfigManager -</td>
<td>This indicates that the value for the Controller port in controller-info.xml is missing. Add</td>
</tr>
<tr>
<td>Incomplete Agent Identity data, Invalid Controller Port Value</td>
<td>the Controller port and host value to resolve:</td>
</tr>
<tr>
<td></td>
<td>• For on-premises Controller installations: 8090 for HTTP and 8181 for HTTPS.</td>
</tr>
<tr>
<td></td>
<td>• For Controller SaaS service, use the default HTTPS port 443.</td>
</tr>
</tbody>
</table>
Caused by: com.singularity.ee.agent.configuration.a: Could not resolve agent-controller basic configuration
This is usually caused because of incorrect configuration in the Controller-info.xml file. Ensure that the information for agent communication (Controller host and port) and agent identification (application, tier and node names) is correctly configured. Alternatively, you can also use the system properties (-D options) or environment variables to configure these settings.

**Unblocking the Controller Port**

The following table lists the typical error message when the Controller port is blocked in your network:

<table>
<thead>
<tr>
<th>Error Message</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERROR com.singularity.CONFIG.ConfigurationChannel - Fatal transport error:</td>
<td>Try pinging the Controller from the machine where you have configured the</td>
</tr>
<tr>
<td>Connection refused</td>
<td>application agent. To check if a port is blocked in the network, use the</td>
</tr>
<tr>
<td>WARN com.singularity.CONFIG.ConfigurationChannel - Could not connect to the</td>
<td>commands:</td>
</tr>
<tr>
<td>controller/invalid response from controller, cannot get initialization</td>
<td>• netstat -an for Windows</td>
</tr>
<tr>
<td>information, controller host \x.x.x.x, port 8090, exception Fatal transport</td>
<td>• nmap for Linux.</td>
</tr>
<tr>
<td>error: Connection refused</td>
<td>The default ports are:</td>
</tr>
<tr>
<td></td>
<td>• For on-premises Controller installations: 8090 for HTTP and 8181 for</td>
</tr>
<tr>
<td></td>
<td>• For Controller SaaS service, use the default HTTPS port 443.</td>
</tr>
</tbody>
</table>

**Correcting File Permission Issues**

Following table lists the typical error message when the file permissions are not correct:

<table>
<thead>
<tr>
<th>Error Message</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERROR com.singularity.JavaAgent - Could Not Start Java Agent</td>
<td>This is usually caused because of incorrect permissions for log files.</td>
</tr>
<tr>
<td>com.singularity.ee.agent.appagent.kernel.spi.c: Could not start services&quot;</td>
<td>To troubleshoot:</td>
</tr>
<tr>
<td></td>
<td>Confirm whether the user who is running the server has read and write</td>
</tr>
<tr>
<td></td>
<td>permission on the agent directories. If the user has chmod a=r equivalent</td>
</tr>
<tr>
<td></td>
<td>permission, change the permission to chmod a+r &quot;&lt;agent_home&gt;&quot;</td>
</tr>
</tbody>
</table>

**Maximum Async Handoff Call Graph Samples Error**

The following error indicates that the number of handoffs in an asynchronous has exceeded the limit:

"WARN AsyncHandOffIdentificationInterceptor - Reached maximum limit 500 of async handoff call graph samples. No more samples will be taken" Error

This can result from transactions being misidentified as async transactions. In AppDynamics 3.6 and later, all Runnables, Callables and Threads are instrumented by default except those that are excluded by the agent configuration in app-agent-config.xml.

In some environments, this may result in too many classes being instrumented, or cause common classes in a framework that implements the Runnable interface to be mistaken for asynchronous activity when it is not, for example Groovy applications using Clojure.

To debug, check the call graph for asynchronous activities that are misidentified as asynchronous activities. If found, exclude the
packages that are not really asynchronous activities.

See Threading and the Java Agent.
Java Agent Configuration Properties

On this page:
- Agent-Controller Communication Properties
- SSL Configuration Properties
- Secure Credential Store Properties
- Agent Identification Properties
- Account Properties
- Proxy Properties for the Controller
- Other Properties

Related pages:
- Administer the Java Agent
- Use System Properties for Java Agent Settings
- Instrument JVMs in a Dynamic Environment

This topic is a reference for the configuration properties for the AppDynamics Java Agent. If you are also installing a Machine Agent on the same machine with the Java Agent, see Standalone Machine Agent Installation Scenarios.

Agent-Controller Communication Properties

Controller Host

The host name or the IP address of the AppDynamics Controller. Example values are 192.168.1.22 or myhost or myhost.example.com. This is the same host that you use to access the AppDynamics browser-based user interface. For an On-premises Controller, use the value for Application Server Host Name that was configured when the Controller was installed. If you are using the AppDynamics SaaS Controller service, see the Welcome email from AppDynamics.

Element in controller-info.xml: `<controller-host>`

System Property: Dappdynamics.controller.hostName

Environment Variable: APPDYNAMICS_CONTROLLER_HOST_NAME

Type: String

Default: None

Required: Yes, if the Enable Orchestration property is false.

If Enable Orchestration is true, and if the app agent is deployed in a compute cloud instance created by an AppDynamics workflow, do not set the Controller host unless you want to override the auto-detected value. See Enable Orchestration Property.

Controller Port

The HTTP(S) port of the AppDynamics Controller. This is the port used to access the AppDynamics browser-based user interface. If the Controller SSL Enabled property is set to true, specify the HTTPS port of the Controller; otherwise specify the HTTP port. See Controller SSL Enabled Property.

Element in controller-info.xml: `<controller-port>`

System Property: Dappdynamics.controller.port

Environment Variable: APPDYNAMICS_CONTROLLER_PORT

Type: Positive Integer

Default: For On-premises installations, port 8090 for HTTP and port 8181 for HTTPS are the defaults. For the SaaS Controller Service, use port 443 for HTTPS connections.

Required: Yes, if the Enable Orchestration property is false.

If Enable Orchestration is true, and if the app agent is deployed in a compute cloud instance created by an AppDynamics workflow, do not set the Controller port unless you want to override the auto-detected value. See Enable Orchestration Property.
SSL Configuration Properties

Controller SSL Enabled

If true, specifies that the agent should use SSL (HTTPS) to connect to the Controller. If SSL Enabled is true, set the Controller Port property to the HTTPS port of the Controller. See Controller Port Property.

Element in controller-info.xml: <controller-ssl-enabled>

System Property: -Dappdynamics.controller.ssl.enabled

Environment Variable: APPDYNAMICS_CONTROLLER_SSL_ENABLED

Type: Boolean
Default: False
Required: No

Controller Keystore Password

The plain text value of the Controller certificate password. If Use Encrypted Credentials is true, encrypt the password. See Encrypt Agent Credentials.

Element in controller-info.xml: <controller-keystore-password>

Type: String
Default: None
Required: No

Controller Keystore Filename

By default, the agent looks for a Java truststore file named cacerts.jks in the <agent_home>/<version>/conf directory in the agent home. Use this property to enable full validation of Controller SSL certificates with a different Java truststore file. See Enable SSL for the Java Agent.

Element in controller-info.xml: <controller-keystore-filename>

Type: String
Default: None
Required: No

Force Default SSL Certificate Validation

Used to override the default behavior for SSL validation. The property can have three states:

- **true**: Forces the agent to perform full validation of the certificate sent by the controller, enabling the agent to enforce the SSL trust chain. Use this setting when a public certificate authority(CA) signs your Controller SSL certificate. See Enable SSL On-Premises with a Trusted CA Signed Certificate.
- **false**: Forces the agent to perform minimal validation of the certificate. This property disables full validation of the Controller’s SSL certificate. Use this setting when full validation of a SaaS certificate fails.
- **unspecified**: The validation performed by the agent depends on the context:
  - If the agent is connecting to a SaasS controller, full validation is performed.
  - If the agent is connecting to an On-premises controller, and the cacerts.jks file is present, then full validation is performed using the cacerts.jks file.
  - If the agent is connecting to an On-premises controller, and there is no cacerts.jks file, then minimal validation is performed

System Property: -Dappdynamics.force.default.ssl.certificate.validation

Type: Boolean
Default: None
Required: No
AppDynamics Agent SSL Protocol

The SSL compatibility table in Agent and Controller Compatibility lists the default security protocol for the different versions of the Java Agent. If the default security protocol for your version of an agent is incompatible with the Controller or it is incompatible with an intervening proxy, pass the `-Dappdynamics.agent.ssl.protocol` system property to configure one of the following security protocols:

- SSL
- TLS
- TLSv1.2
- TLSv1.1

**System Property:** `-Dappdynamics.agent.ssl.protocol`

**Type:** String

**Default:** See Agent and Controller Compatibility

**Required:** No

**Secure Credential Store Properties**

**Use Encrypted Credentials**

Before you enable Use Encrypted Credentials, see Encrypt Agent Credentials for instructions on how to initialize the Secure Credential Store.

Set Use Encrypted Credentials to True to configure the agent to use credentials encrypted with the Secure Credential Store. When you enable Use Encrypted Credentials, you must supply the Credential Store Filename and the obfuscated Credential Store Password. For more information, see Encrypt Agent Credentials.

When Use Encrypted Credentials is true, encrypt the following:

- Account Access Key
- Controller Keystore Password
- Proxy Password

**Element in controller-info.xml:** `<use-encrypted-credentials>`

**Type:** Boolean

**Default:** False

**Required:** No

**Credential Store Filename**

The absolute path to the Secure Credential Store keystore. For more information, see Encrypt Agent Credentials.

**Element in controller-info.xml:** `<credential-store-filename>`

**Type:** String

**Default:** None

**Required:** If Use Encrypted Credentials is set to True

**Credential Store Password**

The obfuscated keystore password for the Secure Credential Store. For instructions on how to obfuscate the password, see Encrypt Agent Credentials.

**Element in controller-info.xml:** `<credential-store-password>`

**Type:** String

**Default:** None

**Required:** If Use Encrypted Credentials is set to True
**Agent Identification Properties**

**Automatic Naming**

If enabled and other agent identification properties are not specified in other settings, the tier and application for the agent are automatically named. The default names are in the format MyApp and MyTier.

**Element in controller-info.xml:** `<auto-naming>

- **Type:** Boolean
- **Default:** None
- **Required:** No

**Application Name**

The name of the logical business application that this JVM node belongs to. Note that this is not the deployment name (ear/war/jar) on the application server.

If a business application of the configured name does not exist, it is created automatically.

**Element in controller-info.xml:** `<application-name>

- **System Property:** -Dappdynamics.agent.applicationName
- **Environment Variable:** APPDYNAMICS_AGENT_APPLICATION_NAME
- **Type:** String
- **Default:** None
- **Required:** Yes

**Tier Name**

The name of the tier that this JVM node belongs to. Note that this is not the deployment name (ear/war/jar) on the application server.

If the JVM or application server startup script already has a system property that references a tier, such as `-Dserver.tier`, you can use `${server.tier}` as the tier name. For more information, see [Use System Properties for Java Agent Settings](#).

The agent registers the named tier with the Controller, if the tier does not already exist, the first time it connects with the Controller. If a tier with the name already exists in the Controller model, the agent is associated with the existing tier.

**Element in controller-info.xml:** `<tier-name>

- **System Property:** -Dappdynamics.agent.tierName
- **Environment Variable:** APPDYNAMICS_AGENT_TIER_NAME
- **Type:** String
- **Default:** None
- **Required:** Yes

**Node Name**

The name of the node. Where JVMs are dynamically created, use the system property to set the node name.

If your JVM or application server startup script already has a system property that can be used as a node name, such as `-Dserver.name`, you could use `${server.name}` as the node name. You could also use expressions such as `${server.name}_${host.name}` to define the node name. See [Use System Properties for Java Agent Settings](#) for more information.

In general, the node name must be unique within the business application and physical host. If you want to use the same node name for multiple nodes on the same physical machine, create multiple virtual hosts using the Unique Host ID property. See [Unique Host ID](#).

**Element in controller-info.xml:** `<node-name>

- **System Property:** -Dappdynamics.agent.nodeName
Environment Variable: APPDYNAMICS_AGENT_NODE_NAME

Type: String
Default: None
Required: Yes

Reuse Node Name

Set this property to true to reuse node names in AppDynamics. When you set the property to true, you don’t need to supply a node name, but you do need to provide a node name prefix using -Dappdynamics.agent.reuse.nodeName.prefix.

This property is useful for monitoring environments where there are many JVMs with short life spans. When true, AppDynamics reuses the node names of historical JVMs for new JVMs. This avoids a proliferation of differently named nodes in AppDynamics over time, particularly when the nodes are essentially identical processes that run over different times. An example of this environment is a z/OS Dynamic Workload Manager based-environment where new JVMs are launched and shut down based on actual workload.

AppDynamics generates a node name with App, Tier and Sequence number. The node names are pooled. For example, the sequence numbers are reused when the nodes are purged (based on the node lifetime).

When the Java Agent starts up, it logs output to the console until it registers with the Controller and the Controller generates the node name. To configure the agent to write logs to a file, edit the log4j-unknown.xml at <agent_home>/<version_number>/conf/logging. See Instrument JVMs in a Dynamic Environment.

The Controller reuses node names based on the node retention period property.

System Property: -Dappdynamics.agent.reuse.nodeName

Environment Variable: APPDYNAMICS_JAVA_AGENT_REUSE_NODE_NAME (New in 4.5.8)

Type: Boolean
Default: False
Required: No

Example: With the following configuration, the Controller generates a node name with the prefix "reportGen". Node names will have suffixes -1, -2, and so on, depending on the number of nodes are running in parallel. The name of a node that is shut down and qualifies as a historical node may be reused by a new node.

-Dappdynamics.agent.reuse.nodeName=true -Dappdynamics.agent.reuse.nodeName.prefix=reportGen

Reuse Node Name Prefix

When you configure the agent to reuse node names, use this property to specify the prefix the Controller uses to generate node names dynamically.

System Property: -Dappdynamics.agent.reuse.nodeName.prefix

Environment Variable: APPDYNAMICS_JAVA_AGENT_REUSE_NODE_NAME_PREFIX (New in 4.5.8)

Type: String
Default: None
Required: When -Dappdynamics.agent.reuse.nodeName=true

Example: Using the following property specifications, the agent directs the Controller to generate a node name with the prefix "reportGen". Node names will have suffixes --1, --2, and so on, depending on how many nodes are running in parallel.

-Dappdynamics.agent.reuse.nodeName=true -Dappdynamics.agent.reuse.nodeName.prefix=reportGen

Self Service

Configure the Java Agent to automatically name nodes based upon the platform. For automatic node naming to work, you must specify an application name and a tier name.

System Property: -Dappdynamics.agent.selfService

Type: String
**Values:**

- **tibco**: The Java Agent names nodes for the TIBCO process name. See [Configure the Java Agent for TIBCO BusinessWorks](#) for more information.

**Default**: None  
**Required**: No

### Account Properties

If the AppDynamics Controller is running in multi-tenant mode or if you are using the AppDynamics SaaS Controller, specify the account name and key for the agent to authenticate with the Controller.

If you are using the AppDynamics SaaS Controller, the account name is provided in the Welcome email sent by AppDynamics. You can also find this information in the `<controller_home>/initial_account_access_info.txt` file.

If the Controller is running in single-tenant mode, you only need to configure the account access key. You can find the unique access key for your Controller instance from the License Management page in the UI.

**Account Name**  
The account name used to authenticate with the Controller.  
Element in `controller-info.xml`: `<account-name>`  
*System Properties*: `Dappdynamics.agent.accountName`  
*Environment Variable*: `APPDYNAMICS_AGENT_ACCOUNT_NAME`  
**Type**: String  
**Default**: None  
**Required**: Yes for AppDynamics SaaS Controller and other multi-tenant users; no for single-tenant users.

**Account Access Key**  
The account access key used to authenticate with the Controller. If Use Encrypted Credentials is true, encrypt the account access key.  
See [Encrypt Agent Credentials](#).  
Element in `controller-info.xml`: `<account-access-key>`  
*System Properties*: `Dappdynamics.agent.accountAccessKey`  
*Environment Variable*: `APPDYNAMICS_AGENT_ACCOUNT_ACCESS_KEY`  
**Type**: String  
**Default**: None  
**Required**: Yes

If you provide application keys through JVM system properties or environment variables, ensure that you use quotes to wrap any shell special characters that may be contained within application keys to prevent the Shell from interpreting them. See the shell documentation for more detail.

### Proxy Properties for the Controller

Use the proxy properties to configure the agent to connect to the Controller through a proxy.

**Proxy Host**  
The proxy host name or IP address.

Proxy authentication cannot be used in conjunction with agent SSL. To connect the agent through a proxy via SSL, the proxy must be open (not require the agent to authenticate).
**System Property:** -Dappdynamics.http.proxyHost

**Type:** String  
**Default:** None  
**Required:** No

**Proxy Port**

The proxy HTTP(S) port.

**System Property:** -Dappdynamics.http.proxyPort

**Type:** Positive Integer  
**Default:** None  
**Required:** No

**Proxy User Name**

The name of the user that is authenticated by the proxy host.

**System Property:** -Dappdynamics.http.proxyUser

**Type:** String  
**Default:** None  
**Required:** No

**Proxy Password**

The absolute path to the file containing the password of the user that is authenticated by the proxy host. The password must be the first line of the file.

If Use Encrypted Credentials is false, enter the password in plain text. If Use Encrypted Credentials is true, encrypt the password. See Encrypt Agent Credentials.

**System Property:** -Dappdynamics.http.proxyPasswordFile

**Type:** String  
**Default:** None  
**Required:** No

**Example:** -Dappdynamics.http.proxyPasswordFile=/path/to/file-with-password

**Other Properties**

**Enable Orchestration**

When set to true, enables auto-detection of the controller host and port when the app server is a compute cloud instance created by an AppDynamics orchestration workflow. See Controller Host Property and Controller Port Property.

In a cloud compute environment, auto-detection is necessary for the Create Machine tasks in the workflow to run correctly.

If the host machine on which this agent resides is not created through AppDynamics workflow orchestration, this property should be set to false.

**Element in controller-info.xml:** <enable-orchestration>

**Type:** Boolean  
**Default:** False  
**Required:** No

**Agent Runtime Directory**
Sets the directory under which all files the agent writes at runtime. If this property is specified, all agent logs are written to
<Agent-Runtime-Directory>/logs/node-name and transaction configuration is written to the <Agent-Runtime-Directory>/conf/node-name
directory. The log folder location can be overridden with the appdynamics.agent.logs.dir property.

Element in controller-info.xml: <agent-runtime-dir>

System Property: Dappdynamics.agent.runtime.dir

Environment Variable: APPDYNAMICS_AGENT_BASE_DIR

Type: String

Default: <agent_home>/nodes

Required: No

Redirect Logfiles

Sets the destination directory to which the logs will be written to.

System Property: Dappdynamics.agent.logs.dir

Type: String

Default: <agent_home>/logs/<Node_Name>

Required: No

Custom path for agent conf directory

Sets a custom path for the agent conf directory. This is where the agent reads its static config files from. If you need to change
custom-activity-correlation.xml or app-agent-config.xml and the agent installation is read-only, this instructs the agent to read the static
config files from elsewhere.

System Property: Dappdynamics.agent.conf.dir

Type: String

Default: <agent_home>/ver4.5.x.x.x/conf

Required: No

Force Agent Registration

Set to true only under the following conditions:

- The agent has been moved to a new application or tier from the UI, and
- You want to override that move by specifying a new application name or tier name in the agent configuration

Element in controller-info.xml: <force-agent-registration>

Type: Boolean

Default: False

Required: No

Auto Node Name Prefix

Set this property if you want the Controller to generate node names automatically using a prefix that you provide.

The Controller generates node names by concatenating the specified prefix with a UUID suffix. For example, if you set the prefix as
follows:

-Dappdynamics.agent.auto.node.prefix=JoannaAutoNode

The generated node name is

JoannaAutoNode_d39dbf1c-6f4b-4eb7-a788-c1c0135b6bcb

This property provides a similar function to the Reuse Node Name Prefix Property property. However, this property is not meant to be
used in combination with reusing node names; use Reuse Node Name Prefix Property for those cases instead.

**Element in controller-info.xml:** Not applicable

**System Property:** `-Dappdynamics.agent.auto.node.prefix=<your_prefix>`

**Type:** String

**Default:** Serial number maintained by the Controller appended to the tier name

**Required:** No

### Cron/Batch JVM

Set this property to true if the JVM is a batch/cron process. This property can be used to stall the shutdown to allow the agent to send metrics before shutdown.

**Element in controller-info.xml:** Not applicable

**System Property:** `-Dappdynamics.cron.vm`

**Type:** Boolean

**Default:** False

**Required:** No

### Unique Host ID

Logically partitions a single physical host or virtual machine such that it appears to the Controller that the application is running on different machines. Set the value to a string that is unique across the entire managed infrastructure. The string may not contain any spaces. If you have a machine agent associated with the application monitored by the app agent, then this property must be set on the machine agent to the same value. See [Standalone Machine Agent Installation Scenarios](#).

**System Property:** `-Dappdynamics.agent.uniqueHostId`

**Environment Variable:** `APPDYNAMICS_AGENT_UNIQUE_HOST_ID`

**Type:** String

**Default:** None

**Required:** No

### Agent Meta Info

Allows you to associate arbitrary information with a node, which can then be used as a basis for applying health rules or policies by node. For example, you can exclude a health rule from applying to agents tagged as test agents based on a meta-info property. Pass the property in key;value format (for example, "key1;value1;key2;value2"). Do not use semicolons as value(s) as it is used as a delimiter.

For information on using the properties in health rules or policies (along with built-in meta-info properties), see [Configure Health Rules](#) or [Configure Policies](#).

**System Property:** `-Dappdynamics.agent.node.metaInfo`

**Type:** String

**Default:** None

**Required:** No

### Low Entropy

Addresses agent startup issues in systems with low to zero entropy available for seeding the PRNG algorithm. We use the NativePRNGNonBlocking algorithm via SecureRandom if the system property `appdynamics.low.entropy` is set.

**System Property:** `appdynamics.low.entropy=true`

**Element in controller-info.xml:** No

**Environment Variable:** No
Type: Boolean
Default: False
Required: No

If `appdynamics.low.entropy=true` then the agent takes measures to ensure it does not block when generating random values, even in the absence of entropy.

**Analytics Agent**

For use with the transaction analytics feature with a remote (or non-default) Analytics agent. See [Enable the App Server Agent for a Remote Analytics Agent](#) for details.

**System Property:** `-Dappdynamics.analytics.agent.url`

**Element in controller-info.xml:** No

**Environment Variable:** No

**Type:** String

**Default:** `http://localhost:9090/v2/sinks/bt`

**Required:** No

**Use Simple Hostname**

By default (unless overridden with the `uniqueHostId` system property), the agent determines the host name of the OS it is running in by reverse DNS lookup. In some circumstances, this host name may be set as the fully qualified domain name of the host name. If this property is set to true, the agent removes any domain name and uses the simple hostname to identify the host. In cases where the host name is an IP address (which happens if the DNS lookup fails) the full IP address in string form is used. The host name is used in mapping metrics gathered by the machine agent to application nodes (see **Unique Host ID Property**).

**Element in controller-info.xml:** `<use-simple-hostname>`

**Type:** Boolean

**Default:** False

**Required:** No

**For example:** If this property is set to true `server.mydomain.com` becomes `server`.
Filter Sensitive Data

By default, the AppDynamics Java Agent sends transaction data to the Controller that your organization may classify as privileged information. Although such data is useful for diagnosis and troubleshooting, security considerations may require you to filter certain information from view in the Controller. You can use:

- Sensitive data filters to exclude environment variable, system property, and JMX data.
- Sensitive URL filters to exclude sensitive information from a URL in snapshot details.
- Sensitive message filters to exclude sensitive data that the application may place in log messages or exception detail messages.

Default Sensitive Data Filters

When you enable a sensitive data filter, the Controller displays asterisks for the values of matching environment variables or system properties. By default, the Java Agent enables two sensitive data filters in the app-agent-config.xml:

- Environment variables or system properties that contain the case insensitive substring “password”.
- Environment variables or system properties that contain the case insensitive substring “key”.

```
<sensitive-data-filters>
  <sensitive-data-filter
    applies-to="environment-variables,system-properties"
    match-type="CONTAINS"
    match-pattern="password"/>

  <sensitive-data-filter
    applies-to="environment-variables,system-properties"
    match-type="CONTAINS"
    match-pattern="key"/>
</sensitive-data-filters>
```

Add a Sensitive Data Filter

2. Add a sensitive data filter element as a child of the Sensitive Data Filters element using one of the following attributes.
   - Specify a comma separated list in the `applies-to` attribute to filter the following:
     ```
     environment-variables
     system-properties
     jmx-mbeans
     ```
   - Set the `match-type` attribute as follows:
     ```
     EQUALS
     CONTAINS
     STARTSWITH
     ENDSWITH
     ```
   - Specify a string to match for the `match-pattern` attribute. String matches are case insensitive. The pattern matches...
against the environment variable and system property names, not values.

3. Restart the JVM.

In the example below, the Java Agent checks for system properties and environment variables beginning with the string "DB_." The Controller displays the values of matching environment variables and system properties as asterisks. For instance, an environment variable "DB_USER" is replaced with an asterisk.

```xml
<sensitive-data-filter
applies-to="environment-variables,system-properties"
   match-type="STARTSWITH"
   match-pattern="DB_"/>
```

### Add a Sensitive URL Filter

You can use sensitive URL filters to configure the agent to obfuscate sensitive information from the URLs in transaction snapshot details.

2. Add a sensitive URL filter element as a child of the sensitive URL filters element:

```xml
<sensitive-url-filter delimiter=""
   segment=""
   match-filter="EQUALS|INLIST|STARTSWITH|ENDSWITH|CONTAINS|REGEX|NOT_EMPTY"
   match-pattern="pattern"
   param-pattern=""/>
```

- **delimiter:** Specify the character that you want to use as URL segment endpoints. The agent splices the URL at each delimiter instance to create the segments. For HTTP, use the forward slash character "/". In the case of a forward slash, the agent does not split on the slashes immediately following the protocol. For example, "https://myapp.example.com" constitutes a single segment. By default, the delimiter is "/".
- **segment:** Specify a comma separated list to indicate the segments that you want the agent to filter. Segment numbering starts from 1.
- **match-filter:** Specify the string that you want to be filtered by the match-filter.
- **match-pattern:** Specify the regular expression matching the query parameters to filter.
- **param-pattern:** Specify the regular expression matching the query parameters to filter.

For example, the following configuration splits the URL on the "/" character and masks the second segment and the param-pattern in the third segment of the URL. In this case, the segmentation and obfuscation apply only to URLs containing "myapp".

```xml
<sensitive-url-filters>
   <sensitive-url-filter delimiter="/"
      segment="2"
      match-filter="CONTAINS"
      match-pattern="myapp"
      param-pattern="[a-z]+_name"/>
</sensitive-url-filters>
```

The exit call to https://myapp.example.com/sensitive/data?first_name=abc&last_name=xyz breaks down to three segments: "https://myapp.example.com", "sensitive", and "data?first_name=abc&last_name=xyz". The Controller shows the masked values of the URL and the param-pattern display https://myapp.example.com/*****/data?first_name=***&last_name=*** in the snapshot details.

In case you do not use any values for the query parameters, the Controller does not mask any query parameters in the URL.
**Add a Sensitive Message Filter**

You can use sensitive message filters to configure the agent to obfuscate sensitive information contained within text messages collected by the agent from log messages, or detail messages from exceptions.

2. Add a sensitive message filter element as a child of the sensitive message filters element:

```xml
<sensitive-message-filter
message-type="throwable,logger-message,all"
match-type="EQUALS|CONTAINS|STARTSWITH|ENDSWITH|REGEX"
match-pattern="CASESENSITIVE_PATTERN"
redaction-regex="SENSITIVE_INFO_REGEX_GROUP"/>
```

- message-type specify throwable, logger-message or all
- match-type specify the type of match that should be used to opt-in messages for redaction
- match-pattern specify the pattern that, when matched, opts the message in for redaction
- redaction-regex specify a regular expression identifying data that should be redacted from the opted-in messages

For example, if an application logs SQL queries including secret numeric values, the following configuration would remove all the numeric data from the logged messages:

```xml
<sensitive-message-filter message-type="logger-message"
match-type="CONTAINS"
match-pattern="SELECT"
redaction-regex="[0-9]+"/>
```

Resulting in –

"SELECT name from customer WHERE customer.id - 773"

Being collected as –

"SELECT name from customer WHERE customer.id - *****"
IBM Java Agent

Under most circumstances, the IBM Java Agent works the same as the Java Agent. This topic gathers information specific to the IBM Java Agent.

Supported JVMs

IBM JVM 1.5.x, 1.6.x, 1.7.x, 1.8.x

Instrumenting the IBM Java Agent

To change instrumentation for the IBM Java Agent, the IBM JVM must be restarted. By default the IBM Java Agent does not apply BCI changes without restarting the JVM. This is because in the IBM VM (J9 1.6.0) the implementation of re-transformation affects performance (changes the JIT behavior such that less optimization occurs).

The following changes require that you restart the IBM JVM.

- Automatic leak detection
- Custom memory structures
- Information points
- Custom POJO rules for transaction detection
- Custom exit point rules
- End user experience monitor (EUM), when you enable it and/or disable it after first enabling it

Instrumenting WebSphere/InfoSphere with WebSphere Security Enabled

If your WebSphere/InfoSphere environment includes a security-enabled WebSphere Application Server, several InfoSphere Master Data Management (MDM) Server clients require security configuration.

For more information, see IBM WebSphere and InfoSphere Startup Settings.
Threading and the Java Agent

Related pages:
- Constructor Mode Thread Tracing
- Executor Mode Thread Tracing (Experimental)

Introduction

A pivotal part of the functionality of the Java agent is tracing activity within the JVM such that all the activity (both internal and external to the JVM) that results from servicing any given inbound request can be associated with the request itself. In simple cases, each request is processed by a single Java thread, making it only necessary to track the processing of that single thread, in a pattern widely known as 'thread per request'. This was the predominant case for JEE applications prior to the introduction of version 3.0 of the servlet specification (introduced in Java EE 6). In systems where a significant amount of time is spent waiting for external resources, the thread per request model can be seen as inefficient since many application threads spend most of their lifetime blocked awaiting external responses. This observation motivated the asynchronous servlet capabilities provided as part of servlet 3, and also motivates many of the increasingly prevalent reactive frameworks that are in common usage (for example, Play, Akka, Vert.x and Spring Reactor). In order for the agent to correctly correlate transactions end to end in applications using this architectural approach, it is necessary for the agent to follow transactions from thread to thread to allow it to measure and link all the elements of work performed on behalf of each request, irrespective of which thread that work is executed on.

The agent provides 2 mechanisms for thread tracking, known as 'Constructor Mode' and 'Executor Mode' (a new Experimental Feature added in Java agent release 4.5.11)

The following pages describe these 2 alternative approaches to correlation of cross-thread transaction activity:
- **Constructor Mode Thread Tracing**
  This is the mode that the agent has used for several years. It instruments Threads, Runnables and Callables based on their creation.
- **Executor Mode Thread Tracing (Experimental)**
  This new experimental tracking mode is based on instrumenting executor frameworks which are commonly used by the majority of applications and frameworks. This mode is expected to make agent operation more efficient, especially in cases where heavy use of threading is in place, for example in the presence of the above mentioned reactive frameworks.

Choosing the Thread Correlation Mode

The thread correlation mode used by the agent is selected using the async-instrumentation-strategy property within app-agent-config.xml configuration file. To use the new (Experimental), executor-based instrumentation, modify app-agent-config.xml by changing the value async-instrumentation-strategy property from constructor to executor before starting the agent. You can also set the instrumentation strategy by setting the appdynamics.async.instrumentation.strategy system property on the JVM command line, for example,
```
Dappdynamics.async.instrumentation.strategy=executor
```

The system property takes precedence over the configuration file. The thread instrumentation mode that is used cannot be changed at runtime. Threading behaviour is tracked using Constructor Mode by default.
Constructor Mode Thread Tracing

When the agent is in Constructor Mode, transaction activity is tracked from thread to thread by instrumenting the constructors of Runnable objects, and their run methods, such that the agent can link the creation point of the object with their actual execution. Since in many cases the work executed by Runnable objects does not relate to the Business Transaction context within which they were created, the agent provides for exclude rules to limit which threads are tracked. A default set of these is maintained within the agent configuration, and this can be fine-tuned by the user where necessary.

Constructor Mode Configuration

Classes for multithreaded correlation are configured in the <excludes> child elements of the <fork-config> element in the <agent _home>/conf/app-agent-config.xml file.

The default configuration excludes the Java, Org, WebLogic and WebSphere classes:

```
<fork-config>
  <!-- exclude java and org -->
  <excludes filter-type="STARTSWITH" filter-value="com.singularity/">/
    <excludes filter-type="STARTSWITH" filter-value="java/,javax/,com.sun/,sun/,org/"/>
  </excludes>
  <!-- exclude weblogic and websphere -->
  <excludes filter-type="STARTSWITH" filter-value="com.bea/,com.weblogic/,weblogic/,com.ibm/,net /nf/,com/mchange/"/>
  ...
```

The agent supports package names where the levels in the hierarchy are either separated by dots (.) or slashes (/). The agent converts the dots to slashes internally.

Custom Configuration

You can edit the app-agent-config.xml file to exclude additional classes from thread correlation. All classes not excluded are by default included.

Use the <excludes> element to specify a comma-separated list of classes or packages to be excluded. Use the singular form, <excludes>, to specify a single class or a package.

For includes, you must specify each package or class individually using the <include> element. You can include a sub-package or subclass of an excluded element by specifying the sub-package or subclass with an <include> element. If a classname is matched by both an exclude and include pattern, the more specific pattern matched prevails. That is, an include match with org.example.myclas prevails over an exclude match for org.example.

Configuration Using Node Properties

You can also configure which classes or packages to include or exclude using a node property. See, thread-correlation-classes and thread-correlation-classes-exclude

Disable Thread Correlation at the Agent

To disable thread correlation (which disables end-to-end asynchronous transaction monitoring as well) at the agent, use one of the following:

- Set the Java agent node property thread-correlation-classes-exclude to disable asynchronous monitoring for all the relevant classes.
  - thread-correlation-classes-exclude=a,b,c,d,e,f,...z
- Add the following line under the fork-config section of the app-agent-config.xml file.
  - <exclude filter-type="REGEX" filter-value=".*"/>
Executor Mode Thread Tracing (Experimental)

The Executor mode is an Experimental Feature. It is an early iteration of a generally available product feature. That means it may not be entirely feature complete, but it is generally available, fully supported by AppDynamics in accordance with your relevant support agreement(s), production-ready, and available for use via end user opt-in. If you have any questions of feedback about this Experimental Feature, reach out to your AppDynamics account team.

When the agent is in Executor Mode, transaction activity is tracked from thread to thread by instrumenting `Executor.execute()` and similar method(s) such that the agent can identify application work to track as that work is being scheduled.

**Benefits**

- Reduces the agent's resource consumption for most use-cases
- Enables tracking thread hand-offs using the new agent API
- Improves reliability of last thread on tier asynchronous transaction demarcation
- Allows Service Endpoints to measure response time of services implemented using asynchronous frameworks

**Behaviour of Service Endpoints in Executor Mode**

The behaviour of Service Endpoints differs between Constructor and Executor Modes. In constructor mode, each asynchronous transaction segment is represented by its own Service Endpoint, and the Service Endpoint corresponding to the transaction entry point shows the execution time of the initiating thread as shown below.

<table>
<thead>
<tr>
<th>Name</th>
<th>Response Time (ms)</th>
<th>Calls</th>
<th>Calls / min</th>
<th>Errors</th>
<th>Errors / min</th>
<th>Type</th>
<th>Tier</th>
</tr>
</thead>
<tbody>
<tr>
<td>AsyncServletOne/one</td>
<td>112</td>
<td>15</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>Servlet</td>
<td>WebServer</td>
</tr>
<tr>
<td>RequestProcessor</td>
<td>1,126</td>
<td>15</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>Async</td>
<td>WebServer</td>
</tr>
<tr>
<td>AsyncServlet#RequestProcessor$Lambda$7</td>
<td>119</td>
<td>15</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>Async</td>
<td>WebServer</td>
</tr>
<tr>
<td>AsyncServlet#RequestProcessor$Lambda$8</td>
<td>95</td>
<td>15</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>Async</td>
<td>WebServer</td>
</tr>
</tbody>
</table>

With the executor instrumentation strategy, metrics for a single Service Endpoint are reported, with a response time corresponding to the execution time of the entire asynchronous transaction as shown below.

<table>
<thead>
<tr>
<th>Name</th>
<th>Response Time (ms)</th>
<th>Calls</th>
<th>Calls / min</th>
<th>Errors</th>
<th>Errors / min</th>
<th>Type</th>
<th>Tier</th>
</tr>
</thead>
<tbody>
<tr>
<td>AsyncServletOne/one</td>
<td>684</td>
<td>15</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>Servlet</td>
<td>WebServer</td>
</tr>
</tbody>
</table>

Currently, each service entrypoint must match a transaction entry point. Configuration of Service Endpoints at other points within execution of asynchronous transactions is currently not supported in the Executor Mode. Additionally, Service Endpoints only report if placed within the context of an executing a Business Transaction when the agent is in Executor Mode.

**Support for Raw Threads**

Support for raw threads differs between executor and constructor modes. While constructor mode supports tracking of all threads created in the context of a transaction, the Executor Mode supports only non-daemon threads which are started in the context of a transaction. The daemon status is inherited from the parent thread which can cause unexpected results. For example, the worker threads in an application server are often daemon threads, so any thread started directly within servlet code is not tracked in the Executor Mode unless the application explicitly unsets its daemon status.

**Node Properties**

The following table presents the node properties which are specific for the executor-based instrumentation strategy, or which work differently than for the constructor-based instrumentation strategy (default).

<table>
<thead>
<tr>
<th>Property</th>
<th>Default value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread-correlation-classes</td>
<td>none</td>
<td>These properties have the same effect in Executor Mode as in Constructor mode, however their use is discouraged in Executor Mode, especially when used to limit the costs of thread tracking. Due to the differences in</td>
</tr>
</tbody>
</table>
implementation of the async hand-offs, any performance gains achieved using these settings in constructor mode are likely to be available using the out of the box settings in Executor Mode.

<table>
<thead>
<tr>
<th>thread-correlation-classes-exclude</th>
<th>implementation of the async hand-offs, any performance gains achieved using these settings in constructor mode are likely to be available using the out of the box settings in Executor Mode.</th>
</tr>
</thead>
<tbody>
<tr>
<td>min-transaction-stall-threshold-in-seconds</td>
<td>60</td>
</tr>
</tbody>
</table>

**Excluding Selected Activities from Transactions**

Asynchronous frameworks use lower level asynchronous mechanisms which are not related to the processing of user transactions (for example, a thread pool might grow and shrink automatically, depending on the load or a framework might initialise its thread pool upon the first use). It is necessary to ignore threads used in these contexts since their lifecycle is not bound to any individual application transaction.

To facilitate exclusion of such asynchronous components (usually Threads and other Runnables), executor mode offers a mechanism called **capture suppression**. The suppression is tied to a specific method by a suppression rule in the async-config section of `app-agen t-config.xml`. Consider the following example of such a rule provided by default:

```
<job>
  <match-class type="matches-class"><name filter-type="EQUALS" filter-value="java.util.concurrent.ThreadPoolExecutor"/></match-class>
  <match-method><name filter-type="EQUALS" filter-value="addWorker"/></match-method>
  <action type="suppression"/>
</job>
```

The `addWorker` method on the `java.util.concurrent.ThreadPoolExecutor` class is a private method invoked by `ThreadPoolExecutor` whenever a new worker thread is to be created, started and added to the pool. Since such threads are not directly associated with any individual transaction at the point of creation, a suppression rule is used in order that any asynchronous task hand-offs occurring within this method will not get associated with the transaction. Any asynchronous tasks subsequently executed by these threads will be associated with the transaction that created the task.

**Known Limitations**

Executor mode is introduced in version 4.5.11 as an **Experimental Feature**. You should validate the behaviour of executor mode in instrumented applications prior to production deployment and report any apparent anomalies.
Trace Multithreaded Transactions for Java

On this page:
- Monitoring Asynchronous and Thread Processing Activity
- Viewing Threads and Thread Tasks in the Metric Browser
- Thread Metrics in Health Rules

Related pages:
- End-to-End Latency Performance
- Threading and the Java Agent

The Java Agent can automatically track multithreaded applications. It correlates threads from classes that implement `Runnable` or `Callable` interfaces or extend the `Thread` class. Processes with such threads are identified in AppDynamics as asynchronous.

Monitoring Asynchronous and Thread Processing Activity

You can monitor performance around asynchronous calls and multithreaded activity in a variety of ways, as described here. It's helpful to understand what AppDynamics considers to be asynchronous activity, what information it provides on that activity, and how that information relates to thread execution at the code level.

The following sections describe the type of information available given various thread and exit call execution scenarios.

**Asynchronous Exit Call**

In the most common case, an asynchronous exit call is an exit call made by a thread that is spawned by a thread other than the primary business transaction thread.

The primary thread, in this case, is the one on which an entry point call is received. A common pattern for this case is when the `main()` function in the application spawns multiple threads as request handlers to receive and process requests. For example:

![Flow map image]

Flow maps in the Controller UI show this connection as a dotted line (asynchronous connection).

AppDynamics considers the call to be an asynchronous call only due to the context in which it was invoked. If the exit call is made from thread 2, as shown in the following illustration, the call may be considered asynchronous from the perspective of the overall design of the application, but is not considered asynchronous in the context of the business transaction:
In other words, without a business transaction, AppDynamics does not track asynchronous activity.

AppDynamics does not differentiate between the two common patterns in asynchronous processing, Wait-for-Completion (in which the threads report back into the primary thread) and Fire-and-Forget (in which the primary thread may finish processing before spawned threads do). Both are represented the same way in the Controller UI, although they would usually be distinguished by their execution time, in which the primary thread for a Wait-for-Completion transaction would exceed any of the child processes, whereas Fire-and-Forget the primary thread may be shorter than child processes.

**Logical Asynchronous Exit Calls**

An exception to the scenario described in the previous section exists for exit calls to backend systems which—from a conceptual point of view—are considered asynchronous in the AppDynamics model.

Message queue tiers such as JMS are considered logically asynchronous, so flow maps show connections to message queues as dotted lines (asynchronous connections) as well. Therefore, the following JMS exit call would be shown as an asynchronous call in a flow map, while the JDBC call would not be.

**Time on Tier**

Exit calls are one consideration in understanding multithreaded application performance; another consideration is the actual time spent processing a request.

For example, consider the following processing scenario. This occurs on a single tier. In fulfilling a request, the entry point thread spawns multiple child threads that report back to the primary thread to assemble the response.
For the transaction, the response time metric represents the time from when the request arrived at the entry point to the point the response was sent. However, while this may accurately represent the user experience of the transaction, it does not reflect the processing burden on the system, since many threads participate in the transaction. A slow performing thread may or may not be evident based on the user experience of the transaction.

To better understand the complete cost of the transaction in terms of time spent processing, you can use the time on tier metric. This metric shows the aggregate processing time for each of the threads involved in processing a request. In other words, the total execution time of threads 2, 3 and 4 from the figure, along with the processing time between the entry point and response for thread 1.

The value of the time on tier metric is indicated by the async tag in business transaction dashboards.

**End to End Processing**

A response from an entry point thread may not best represent the logical end of a business transaction.

For example, consider an application in which an entry point method in a request handler spawns multiple threads, including one to serve as the final response handler. The request handler then returns a preliminary response to the client. By default, this stops the clock for purposes of measuring the response time for that business transaction. Meanwhile the spawned threads continue to execute until completion, at which point the response handler generates the final response to the client.

In this case, the initial response time is much shorter than the full logical transaction execution time.
End-to-end metrics let you monitor logical transactions that use this programming pattern. End-to-end metrics include the end-to-end transaction time, the number of end-to-end transactions per minute, and the number of slow end-to-end message processing events.

To enable end-to-end metrics, you configure an Asynchronous Transaction as described in Asynchronous Transaction Demarcators.

**Viewing Threads and Thread Tasks in the Metric Browser**

In a multithreaded transaction, AppDynamics reports key business transaction performance metrics for individual threads in a Thread Tasks branch of the tier in the Metric Browser. The Thread Tasks branch is present in the Metric Browser only for multithreaded transactions. The Metric Browser path is Business Transaction Performance > Business Transactions > tier-name > business-transaction-name > Thread Tasks. Thread Tasks also appear for tiers under Overall Application Performance, where you can see metrics on specific calls made by each thread in a node or in a tier.

**Thread Metrics in Health Rules**

For each asynchronous thread spawned in the course of executing a business transaction, AppDynamics collects and reports these metrics:

- Average response time
- Calls per minute
- Errors per minute

You can create custom health rules based on the performance metrics for a thread task. When you click the metric icon in the Health Rule Wizard, the embedded metric browser includes the Thread Tasks if the entity for which you are configuring the health rule spawns multiple threads. See Configure Health Rules.
Java Agent API User Guide

On this page:

- About the Java Agent API
- Installing the Dependency
- Common Use Cases
- Starting and Ending an Asynchronous Business Transaction

Related Pages

- AppDynamics APIs
- Data Collectors
- Add Custom Fields to Business Transactions Using the Java SDK

This topic introduces the Java Agent API and describes the common use cases for the API.

About the Java Agent API

Introduction

AppDynamics automatically detects an application activity out-of-the-box. The agent ships with out-of-the-box configuration that - for supported application frameworks - tells it where Business Transaction activity starts, where thread handoffs happen and where calls to downstream systems are made and how to inject correlation headers into outbound messages and retrieve them from inbound messages such that AppDynamics can establish the end-to-end transaction flow through the application architecture.

For frameworks that do not benefit from out-of-the-box support, the agent provides a set of APIs allowing the application developer to make code changes to add calls to the agent to allow it to identify and follow Business Transactions as they execute, providing end-to-end visibility for any application.

Use of the agent API can be seamlessly combined with the out-of-the-box instrumentation to support situations where a mixture of supported and unsupported frameworks are in use (for example, a standard servlet making an external call using a proprietary API).

The Java Agent API enables you to programatically:

- Define Business Transactions
- Define Exit Calls
- Inform the agent when the application hands off transaction processing between threads
- Add application data to snapshots or transaction analytics
- Report custom metrics

When instrumenting any application, the agent API design prioritises the success of the application transactions over instrumentation. For this reason, if any of the agent API calls fail, they do not throw exceptions disrupting the transaction flow, but rather log messages to aid diagnosis. This also means that there is no hard dependency between the application and the presence of the AppDynamics Java agent within the JVM.

Installing the Dependency

The agent API jar can be accessed directly or downloaded from Maven Central, or it can be downloaded from the AppDynamics portal. The library version changes with each new API release, and is not tightly coupled to the version of the underlying agent, which must be a minimum of version 4.5.11.

For use with Maven Central, add the dependency to your build files in Gradle:

```groovy
dependencies {
    compile group: 'com.appdynamics', name: 'agent-api', version: '
    '4.5.12.26900'
}
```

or Maven:

Copyright © AppDynamics 2012-2019
See, [https://docs.appdynamics.com/javadocs/java-agent-api/v4.5/](https://docs.appdynamics.com/javadocs/java-agent-api/v4.5/) for Javadoc reference for the agent API.

Common Use Cases

**Starting and Ending a Synchronous Business Transaction**

The following example shows code that starts a Business Transaction called 'Checkout' whenever the method `checkout()` is called. The Business Transaction ends when the method does. Encapsulating the method body in a try/finally block ensures that you end the Business Transaction even if the method itself throws an exception or otherwise terminates without reaching the end.

```java
public String checkout(List<ItemOrders> orders) {
    Transaction transaction = null;
    try {
        transaction = AppdynamicsAgent.startTransaction("Checkout", null, false);

        /* Method Body Here */
    } finally {
        if (transaction != null) {
            transaction.end();
        }
    }
}
```

Alternatively you can use try-with-resources pattern:

```java
public String checkout(List<ItemOrders> orders) {
    try (Transaction transaction = AppdynamicsAgent.startTransaction("Checkout", null, false)) {
        /* Method Body Here */
    }
}
```

In this case, the Business Transaction ends when the try block closes.
Starting and Ending an Asynchronous Business Transaction

This example shows code that starts a Business Transaction called 'CheckoutAsync' whenever the method `checkoutAsync()` is called. The originating segment created for the Business Transaction ends when the method `endSegment()` is called on the Business Transaction, or when it is closed if used in a try-with-resources construct. Encapsulating the method body in a try/finally block ensures that we end the segment even if the method itself throws an exception or otherwise terminates without reaching the end.

```java
//The thread where the Business Transaction starts
public String checkoutAsync(List<ItemOrders> orders) {
    Transaction transaction = null;
    try {
        transaction = AppdynamicsAgent.startTransaction("CheckoutAsync", null, true);
        //mark handoff to link this segment with the end segment
        transaction.markHandoff(commonPayload);

        /*******************
        * Method Body Here
        *******************/
    } finally {
        if (transaction != null) {
            transaction.endSegment();
        }
    }
}
```

Alternatively, try-with-resources pattern is supported:

```java
//The thread where the Business Transaction starts
public String checkoutAsync(List<ItemOrders> orders) {
    try (Transaction transaction = AppdynamicsAgent.startTransaction("CheckoutAsync", null, true)) {
        //mark handoff to link this segment with the end segment
        transaction.markHandoff(commonPayload);

        /*******************
        * Method Body Here
        *******************/
    }
}
```

This ends the segment in the thread where the Business Transaction was started when the try block closes. The Business Transaction itself needs to be ended in the method where async Business Transaction ends.
//The thread where the Business Transaction ends
public String checkoutAsyncEnd(List<ItemOrders> orders, Transaction transaction, Object commonPayload) {
    //link to the originating segment
    Transaction transactionSegment = AppdynamicsAgent.startSegment(commonPayload);

    ****************************
    * Method Body Here
    ****************************
    if (transactionSegment != null) {
        transactionSegment.endSegment();
    }
    if (transaction != null) {
        transaction.end();
    }
}

**Defining an Exit Call**

Given an inventoryServer.verifyQuantities(orders) which makes a request to another process, you can monitor that request as an Exit Call to continue monitoring the Business Transaction through the call to the downstream server, and identify the time spent in the remote service. You can do this by modifying the method as follows:

```java
public void verifyQuantities(List<ItemOrders> orders) {
    ExitCall exitCall = null;
    try {
        exitCall = AppdynamicsAgent.getTransaction().startExitCall("Quantity Check", "Inventory Server", false);
        ****************************
        * Method Body
        ****************************
    } finally {
        if (exitCall != null) {
            exitCall.end();
        }
    }
}
```

The above code modifications defines the Exit Call that manifests it as a remote service in the controller. To tag and follow the request into an instrumented downstream tier, add a correlation header:
public void verifyQuantities(List<ItemOrders> orders) {
    ExitCall exitCall = null;
    try {
        exitCall = AppdynamicsAgent.getTransaction().startExitCall("Quantity Check", "Inventory Server", false);

        // Generate the appdynamics correlation header
        String correlationHeader = exitCall.getCorrelationHeader();

        // ... Method code including request creation

        // Modify the request/payload to include the correlation header
        inventoryRequest.addHeader(AppdynamicsAgent.TRANSACTION_CORRELATION_HEADER, correlationHeader);
    } finally {
        if (exitCall != null) {
            exitCall.end();
        }
    }
}

Defining an Asynchronous Thread Handoff

If your checkout method also does a thread handoff and executes some business logic you would be interested in monitoring in a separate thread, register the worker thread with the Business Transaction.
public String checkout(List<ItemOrders> orders) {
    Transaction transaction = null;
    try {
        transaction = AppdynamicsAgent.startTransaction("Checkout", null, false);

        // ... Method code

        // Custom thread handoff using custom queue
        asyncTaskQueue.add(task);
    }
    finally {
        if (transaction != null) {
            transaction.end();
        }
    }
}

To instrument this, modify the add method to mark a thread handoff and then start a new segment where the thread begins running.

```java
public class AsyncTaskQueue {
    public void add(Task task) {
        AppdynamicsAgent.getTransaction().markHandoff(task);

        /*******************
        * Method Body
        *******************/
    }
}
public class Task {
    public void run() {
        Transaction transaction = null;
        try {
            transaction = AppdynamicsAgent.startSegment(this);
            
            /*******************
            * Method Body
            *******************/
            } finally {
            if (transaction != null) {
                transaction.endSegment();
            }
        }
    }

    public void cancel() {
        AppdynamicsAgent.cancelHandoff(this);
        
        /*******************
        * Method Body
        *******************/
    }
}

The task object is used by the agent to link the segments. Correlating thread segments using the agent API requires that the agent is running in Executor mode.

**Adding Data to Snapshot or Analytics**

Often there are values of interest in the code that are helpful to add to snapshots to aid in root cause diagnosis of issues, or to send to the AppDynamics Business Transaction analytics to help answer real time business-oriented questions about your application. Data reported using this API appears in the same way as if it had been collected with a Method Invocation Data collector.

To report a total checkout amount to Business Transaction analytics and have it present in APM snapshots, use the following code:
private static final Set<DataScope> dataScopeSet = new HashSet(Arrays.asList(DataScope.ANALYTICS, DataScope.SNAPSHOTS));

public String checkout(List<ItemOrders> orders) {
    Transaction transaction = null;
    try {
        transaction = AppdynamicsAgent.startTransaction("Checkout", null, false);

        // ... Method code
        double shoppingCartTotal = total(orders);
        transaction.collectData("cart total", Double.toString(shoppingCartTotal), dataScopeSet);
    } finally {
        if (transaction != null) {
            transaction.end();
        }
    }
}

**Defining a Custom Metric or Event**

It can also be useful to report a value as a custom metric:

```java
public String checkout(List<ItemOrders> orders) {
    // ... Method code
    double shoppingCartTotal = total(orders);
    AppdynamicsAgent.getMetricPublisher().reportSumMetric("Cart Total", (long) shoppingCartTotal);
}
```

Reporting custom metrics and events is possible irrespective of the Business Transaction context.
To monitor .NET applications with AppDynamics, you need to install the .NET Agent on the servers where the applications run, as described here.

You only need to install the agent once per server even if you want to monitor more than one application on the server.

**About these Instructions**

The steps below help you to install the agent for IIS applications using the Getting Started wizard in the Controller. For alternative approaches, see:

- [Install the .NET Agent for Windows](#), if you downloaded the agent directly from the AppDynamics Download Center.
- [Configure the .NET Agent for Windows Services and Standalone Applications](#), if you are monitoring Windows services or standalone applications.

**Before You Begin**

1. Verify support for your environment at [.NET Supported Environments](#).
2. Confirm you have access to a compatible Controller. See [Agent and Controller Compatibility](#).
3. Confirm the connection settings to the Controller where your agent will report data:
   - If you use a SaaS Controller, AppDynamics sent you the controller hostname in your Welcome Email. Use port 443 for HTTPS or port 80 for HTTP.
   - If you use an on-premises Controller, you supplied the hostname and port at install time.
4. Verify you have access to the machine where the application runs as a user account with privileges to install the agent software and restart the application.
5. Verify that the machine where the application runs can connect to the Controller. Proxies or firewalls on the network between the agent and Controller may require additional configuration.

**Install the .NET Agent**

To install the agent for IIS applications:

1. Log in to the Controller UI and access the Getting Started Wizard for .NET.
2. Follow the steps in the wizard to configure and download the agent. The agent guides you through some preliminary configuration steps.
   When finished, the wizard lets you download the agent as a ZIP archive named in the format of `dotNetAgent-Portal-<architecture>-<version>.zip`.
3. Extract the agent archive on the destination computer.
4. Launch an elevated command prompt with full administrator privileges.
5. Execute the Installer.bat file you extracted.
   The batch file installs the agent and starts the AppDynamics Agent Coordinator service.
6. Restart IIS.
7. If you are in a testing environment, apply load to your application to see the activity in the Controller UI.
8. Log on to the AppDynamics Controller to see your application in action. You should see something like this:
That is it! Next, you can install more agents, or you can begin monitoring your application. See AppDynamics Essentials.
.NET Supported Environments

On this page:
- Supported Runtime Environments
- Automatically Discovered Business Transactions
- Supported Loggers for the .NET Agent
- Remote Service Detection
- Supported Windows Azure Remote Services
- Cache Clients
- Data Storage Detection

Related pages:
- Browser RUM Supported Environments

.NET Agent Support

Supported Runtime Environments

This section lists the environments where the .NET Agent does some automatic discovery after little or no configuration.

OS Versions

- Microsoft Windows Server 2008 (32-bit and 64-bit)
- Microsoft Windows Server 2008 R2
- Microsoft Windows Server 2012
- Microsoft Windows Server 2012 R2
- Microsoft Windows Server 2016
- Microsoft Windows 7, 8, 8.1, 10

Microsoft .NET Frameworks

Microsoft .NET Framework versions 2.0, 3.0, 3.5, 4.0, 4.5, 4.5.2, 4.6, 4.7 on the following runtime environments:

- Microsoft IIS versions 6.0, 7.0, 7.5, 8.0, 8.5, 10
- Managed Windows Services
- Managed Standalone Applications
- Microsoft SharePoint 2010, 2013 as services running inside IIS
- Microsoft .NET Core 2.0/2.1 for Windows
- Microsoft .NET Core 2.2 for Windows is supported for .NET Agent versions 4.5.7 and later

Microsoft Windows Azure

- Azure App Services for .NET 4.6 environments in the Azure Portal
  - Web Apps
  - Web Jobs
  - API Apps
  - Container Services
  For Azure App Services, the .NET Machine Agent disables certain .NET Machine Agent infrastructure monitoring features: CLR crash reporting, machine snapshots, and Windows performance counter monitoring.
- Azure Cloud Services
  - Web Roles
  - Worker Roles

Unsupported Frameworks

- Microsoft .NET versions 1.0, 1.1
- Unmanaged native code

Automatically Discovered Business Transactions
The .NET Agent discovers business transactions for the following frameworks by default. The agent enables detection without additional configuration.

<table>
<thead>
<tr>
<th>Type</th>
<th>Custom Configuration Options?</th>
<th>Downstream Correlation?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASP.NET*</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ASP.NET MVC 2</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ASP.NET MVC 3</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ASP.NET MVC 4</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ASP.NET MVC 5</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ASP.NET Core on the full framework</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Open Web Interface for .NET (OWIN) web API</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>.NET Remoting</td>
<td>No</td>
<td>See Enable Correlation for .NET Remoting.</td>
</tr>
<tr>
<td>Windows Communication Foundation (WCF)</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Web Services including SOAP</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Message Queues**

<table>
<thead>
<tr>
<th>Type</th>
<th>Custom Configuration Options?</th>
<th>Downstream Correlation?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache ActiveMQ NMS framework and related MQs</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>IBM WebSphere MQ</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Microsoft Message Queuing (MSMQ)</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Microsoft Service Bus / Windows Azure Service Bus</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>NServiceBus over MSMQ or RabbitMQ transport</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>RabbitMQ</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>TIBCO Enterprise Message Service</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>TIBCO Rendezvous</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Windows Azure Queue</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

* The .NET Agent automatically discovers entry points for ASP.NET web forms with the Async property set to "true" in the Page directive.

**Supported Loggers for the .NET Agent**

- Log4Net
- NLog
- System Trace
- Windows Event Log

If you are using a different logger, see Error Detection.

**Remote Service Detection**

The .NET Agent automatically detects the following remote service types. The agent enables detection by default. You do not need to perform extra configuration.

<table>
<thead>
<tr>
<th>Type</th>
<th>Custom Configuration Options?</th>
<th>Async Detection?*</th>
<th>Downstream Correlation?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Directory Services, including LDAP</td>
<td>No</td>
<td>No</td>
<td>N/A</td>
</tr>
</tbody>
</table>
### HTTP
- **HTTP**: Yes
- **See Asynchronous Exit Points for .NET**: Yes
- **MongoDB: C# and .NET MongoDB Driver version 1.10, 2.0**: No
- **See Asynchronous Exit Points for .NET**: N/A
- **.NET Remoting**: Yes
- **See Asynchronous Exit Points for .NET**: No
- **WCF**: Yes
- **See Asynchronous Exit Points for .NET**: Yes
- **WCF Data Services**: Yes
- **Web Services, including SOAP**: Yes
- **See Asynchronous Exit Points for .NET**: No
- **Azure Service Fabric Remoting v1 and v2—for the .NET Microservices Agent**: -
- **See Asynchronous Exit Points for .NET**: -

### Data Integration
- **Microsoft BizTalk Server 2010, 2013**: No
- **See Correlation Over Microsoft BizTalk**: Yes

### Message Queues
- **Apache ActiveMQ NMS framework and related MQs**: Yes
- **IBM WebSphere MQ (IBM XMS)**: Yes
- **Microsoft Message Queuing (MSMQ)**: Yes
- **Microsoft Service Bus / Windows Azure Service Bus**: No
- **NServiceBus over MSMQ or RabbitMQ transport**: No
- **RabbitMQ**: No
- **TIBCO Enterprise Message Service**: Yes
- **TIBCO Rendezvous**: Yes
- **Windows Azure Queue**: No

### Cache Clients
- **StackExchange.Redis**: No

### Data Storage Detection
The .NET Agent automatically detects the following data storage types. The agent enables detection by default. You do not need to perform extra configuration.
<table>
<thead>
<tr>
<th>Type</th>
<th>Customizable Configuration?</th>
<th>Async Detection?*</th>
<th>AppD for Databases?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADO.NET (see supported clients below)</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Windows Azure Blob Storage</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Windows Azure File Storage</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Windows Azure Table Storage</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

* The agent discovers asynchronous transactions for the Microsoft .NET 4.5 framework. See Asynchronous Exit Points for .NET

Supported ADO.NET Clients

AppDynamics can monitor any ADO.NET client version and type. Clients we've tested include the following:

<table>
<thead>
<tr>
<th>Database Name</th>
<th>Database Version</th>
<th>Client Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oracle</td>
<td>10, 11, 12</td>
<td>ODP.NET</td>
</tr>
<tr>
<td>Oracle</td>
<td>10, 11, 12</td>
<td>Microsoft Provider for Oracle</td>
</tr>
<tr>
<td>MySQL</td>
<td>5.x</td>
<td>Connector/Net and ADO.NET</td>
</tr>
</tbody>
</table>

* Microsoft, SQL Server, and Windows are registered trademarks of Microsoft Corporation in the United States and other countries.
Install the .NET Agent for Windows

To monitor IIS applications, Windows services or standalone applications, install the AppDynamics .NET Agent once on each machine that hosts managed .NET applications. At startup, the agent initializes an individual instance of itself for each application running in the CLR.

This topic describes a new installation for the .NET Agent using the MSI package.

- To install from the command line, see Unattended Installation for .NET
- To install the AppDynamics Site Extension for Windows Azure, see Install the AppDynamics Azure Site Extension for .NET
- To install the AppDynamics .NET Microservices Agent using one of the new AppDynamics NuGet Packages, see .NET Microservices Agent
- To upgrade, see Upgrade the .NET Agent

Installation Overview

1. Prepare to Install.
2. Install the agent.
3. Configure the agent.
4. Restart instrumented applications.
   - For Microsoft IIS, the configuration utility gives you the option to restart IIS or not.
   - If you do not restart IIS, monitoring does not begin until the next time IIS restarts.
   - You must restart Windows services and standalone applications manually.

Prepare to Install

Review the following requirements before you install the .NET Agent:

- Windows Management Instrumentation service must be enabled on the machine where you are installing the .NET Agent and the service must be running.
- To run the .NET Agent and the Standalone Machine Agent on the same machine, .NET Compatibility Mode must be enabled on both the machine and the Controller. For more information, see .NET Compatibility Mode.

Install the .NET Agent

1. Download the MSI installer package from the AppDynamics Download Center.
2. Run the MSI installer package.
3. Read the End User Agreement and click to accept. Click Next.
4. Optional. On the Confirm location window, change the .NET Agent installation directories, including:
   - the destination directory for the .NET Agent executables and supporting files.
   - the parent directory for local data including agent configuration files and log files.
   See .NET Agent Directory Structure for the default directories.

You can also change the destination of the logs directory later using the AppDynamics Agent Configuration utility.
5. On the User Account Control window, click **Yes** to allow the installer to make changes to the computer. If the current account does not have administrator privileges, the installer prompts you to supply the password for an administrator account.
6. Wait for the installation to complete.
7. For new installs, AppDynamics recommends you launch the AppDynamics Agent Configuration utility.

If you encounter problems installing, see Resolve .NET Agent Installation and Configuration Issues

**Configure the .NET Agent**

Launch the AppDynamics Agent Configuration utility to configure the .NET Agent.

- For Microsoft IIS applications, see Configure the .NET Agent
- For Windows services or standalone applications, see Configure the .NET Agent for Windows Services and Standalone Applications
Configure the .NET Agent

On this page:
- Prepare to Configure the .NET Agent
- Configure the .NET Agent

Related pages:
- Install the .NET Agent for Windows
- Name .NET Tiers
- Configure the .NET Agent for Windows Services and Standalone Applications
- Unattended Installation for .NET

After you install the AppDynamics .NET Agent, configure the agent according to the types of applications you want to monitor:

- For Microsoft IIS applications, use the configuration utility with either automatic or manual tier naming. See the instructions in this topic.
- For Windows services or standalone applications, use the configuration utility, then manually update the config.xml. See Configure the .NET Agent for Windows Services and Standalone Applications

You can instrument any combination of IIS Applications, Windows services, and standalone applications on a single server. Run the configuration utility to configure IIS, then follow the instructions for Windows services and Standalone applications.

Use the AppDynamics Agent Configuration Utility to configure the agent immediately after installation, or to make changes to existing agent configurations. The utility configures the agent for one machine at a time.

Prepare to Configure the .NET Agent

- Install the .NET Agent for Windows.
- For account credentials, you need the following depending on your installation type:
  - Single-tenant: Membership in a role with the View License account level permission. See Roles and Permissions.
  - Multi-tenant or SaaS: Your AppDynamics Welcome Email that contains your account credentials.
- Uninstall any pre-existing profiler, such as Ant, VS 2010 Performance Tools, or others. The configuration utility alerts you if it finds a pre-existing profiler.
- Decide your tier naming scheme:
  - Automatic
  - Manual
  - Pre-configured using Config Management in the Controller

The configuration utility must restart IIS to apply configurations. The utility offers you the option to restart IIS or not. If you choose not to restart, configurations apply the next time IIS restarts.

File System Security Settings

The following Windows accounts require specific file system permissions:

- The account you use to run your web application as defined by its application pool or the Windows service account.
- The account you use to run the AppDynamics Agent Coordinator, by default the Local System account.

The required permissions are as follows:

- Write permission to the .NET App Agent logs directory. By default:
  - Windows Server 2008 and later: %ProgramData%\AppDynamics\DotNetAgent\Logs
- Read and Execute permissions to the .NET App Agent install directory, by default C:\Program Files\AppDynamics\AppDynamics .NET Agent
- Read and Execute permissions the web application installation directory, for example C:\inetpub\wwwroot\myapp

Configure the .NET Agent

   If the Warning: 3rd Party Profiler installed message displays, click Yes to exit and uninstall any pre-existing profiler.
Check the registry to make sure that the uninstall process cleaned up the registry entries. Use the warning message to identify any undeleted profiler environment variables.

2. If the configuration utility detects legacy agent configurations from the .NET Agent version 3.7.7 or earlier, it displays the Upgrade page. Choose either:
   • Yes to remove legacy configurations. See ‘Clean up legacy configurations’ on Upgrade the .NET Agent for Windows.
   • No to leave legacy configurations in place.
   When the utility discovers no further profiler conflicts or after any configuration clean up it displays the welcome page.
3. On the Log directory permissions page, you can optionally change the location of the log directory, click Change and select a new location.
4. If needed, enter an account to grant log directory permissions. Click Add. If you see a warning message, make sure that the account is valid on the system.
   The wizard confirms the list of accounts.
5. On the Controller Configuration page, enter the Controller access information and credentials.
   • The AppDynamics Agent Configuration utility only supports configuration of one Controller and business application per server.
     • Use tiers to organize different applications you instrument on a single server, or
     • Manually configure support for multiple business applications, see Configure Multiple Business Application Support for .NET.
   • Enter the Controller server name or IP and port number.
   • For single-tenant accounts, enter your Account Access Key. Find your credentials under Settings > License in the Controller.
   • For multi-tenant accounts, click Multi-Tenant Controller. Enter the Account Name and Account Access Key as provided to you by AppDynamics.
   • For a secure connection, click Enable SSL.
     • Make sure you have secured the Controller with a trusted certificate. See Enable SSL for .NET.
     • When you enable SSL, the agent secures communication to the Controller using the protocols set for ServicePointManager.SecurityProtocol in your application.
     • By default, the configuration utility enables TLS 1.2, making it the first option in the list of secure protocols. To disable TLS 1.2, click to deselect this option.
   • If needed, fill in the HTTP proxy information. Proxies that use authentication require additional configuration.
5. Click Test Controller Connection to verify the connection.
7. On the Application Configuration page, AppDynamics displays existing business application information from the Controller and displays it in the left column. Controller connection status displays on the right.
8. Configure the business application for the Controller as follows:
   • If you already have a business application on the Controller, click Existing Applications from the Controller and click the appropriate business application. If you haven't defined business applications in the Controller, the utility displays an empty list.
   • Click New Application to define a new business application. Be careful about spellings and capitalization and note down the exact name. Ampersands are not supported in application names.
   • If you want to use Config Management in the Controller, click Register Machine with No Application.
9. Click Next to advance to Assign IIS applications to tiers where you can use one of the tier naming options below.

Automatically Name IIS Tiers

1. In the Assign IIS applications to tiers window, click Automatic.
2. If prompted, click OK to confirm the Automatic configuration.
   The configuration utility summarizes the configuration settings.
3. By default when you click Next the configuration utility restarts IIS.

If you do not want to apply the configuration right away, uncheck the box. The Agent Configuration Utility saves the information and applies it the next time you restart IIS.
4. If you proceed and click **Next**, the configuration utility logs its activities, including stopping and restarting IIS, and reports any problems. Review the summary for any issues in red font. Green font indicates the more interesting logged events. The summary shows any Warnings (W) or Errors (E).

5. When you're finished, click **Done** to close the Configuration Utility.

**Manually name IIS tiers**

1. On the Assign IIS applications to tiers window, click **Manual**, then click **Next**.
2. Assign IIS Applications to AppDynamics tiers by selecting a tier on the right and a business application on the left. The utility highlights the assigned tier in boldface.

   For large IIS installations, use the Max IIS tree depth pulldown to display all the projects. A large tree depth may take some time to view.

To create new tiers, enter a name and click **Add Tier**.

3. When you are done click **Next**.
4. On the **Configuration Summary** page, uncheck **Restart IIS** if you don't want to immediately restart IIS.
   You may restart later to apply your changes, or they will take effect after a reboot.
5. If you proceed and click **Next**, the Configuration Utility logs its activities, including stopping and restarting IIS, and reports any problems.
6. Review the configuration log summary.
   As it applies the configuration, AppDynamics generates a log of the configuration activities and displays a summary. Review the summary for any issues in red font. Green font indicates the more interesting logged events. The summary shows any Warnings (W) or Errors (E).
7. When you are done, click **Next**. The wizard completes.

**Use Config Management**

1. On the Assign IIS applications to tiers page, click **Register Machine with No Application** and then **Next**.
2. On the Configuration Summary page, uncheck **Restart IIS** if you don't want to immediately restart IIS.
   You may restart later to apply your changes, or they will take effect after a reboot.
3. Follow the instructions on **Manage Configuration for .NET** to assign a configuration from the Controller.
   The .NET Agent and .NET Machine Agent do not report metrics to the Controller until after you assign a configuration using the Config Management tool.

For troubleshooting information see **Resolve .NET Agent Installation and Configuration Issues**.
Enable SSL for .NET

On this page:
- Requirements
- Establish Trust for the Controller's SSL Certificate
- Enable SSL for the .NET Agent
- Troubleshooting

Related pages:
- Secure the Platform
- Administer the .NET Agent
- .NET Agent Configuration Properties

This topic covers how to configure the AppDynamics .NET Agent to connect to the Controller using SSL.

Requirements

Before you configure the agent to enable SSL, gather the following information:

- Identify the Controller SSL port:
  - For SaaS Controllers the SSL port is 443.
  - For on-premises Controllers the SSL port is 8181 by default, but it is possible for on-premises Controllers to be configured to use other ports at installation time.
- Identify the signature method for the Controller's SSL certificate:
  - A publicly known certificate authority (CA) signed the certificate. This applies for Verisign, Thawte, and other commercial CAs.
  - A CA internal to your organization signed the certificate. Some companies maintain internal certificate authorities to manage trust and encryption within their domain.
  - The Controller uses a self-signed certificate. In this case, you cannot enable SSL for the .NET Agent because the agent does not support self-signed certificates on the Controller. You can only use SSL with the .NET Agent when the Controller uses a certificate signed by a trusted CA signing authority or an internal trusted root CA. See Secure the Platform.

Establish Trust for the Controller's SSL Certificate

The .NET Agent requires that the Common Name (CN) on the Controller certificate matches the DNS name of the Controller. Additionally, certificates for the root CA that signed the Controller’s SSL certificate must reside in the Windows Trusted Root Certification Authorities store for the Local Computer.

Certificates Signed by a Publicly Known Certificate Authority

The root certificates for most publicly trusted CA signing authorities, such as DigiCert, Verisign, Thawte, and other commercial CAs, are in the Trusted Root Certification Authorities store by default.

Certificates Signed by an Internal Certificate Authority

If your organization uses internal CA to sign certificates, you may need to obtain the root CA certificate from your internal security management resource. To import the root certificate, see Adding certificates to the Trusted Root Certification Authorities store for a local computer.

This example shows how to use the Certificate snap-in for the Microsoft Management Console to import a certificate for a Trusted Root Certification Authority:
This example shows the Intermediate Certification Authorities store:

If an intermediate CA signed the Controller certificate, you must import the certificate for the intermediate CA in addition to the one for the root CA that signed the intermediate CA certificate. If your Controller is publicly accessible, you can use a certificate checker to identify the certificates required to complete the trust chain. See the certificate checker from Thawte.
Certificate Management Tips

- If you imported certificates for a root or intermediate CA, verify the certificate store where you imported them. Import them to **Certificates (Local Computer)**.

- The AppDynamics SaaS Controller uses certificates signed by DigiCert. In some cases, SaaS customers must import the DigiCert root certificates into the **Windows Trusted Root Certification Authorities** store.
- In some cases system administrators set up group policies that require external certificates be imported to the **Third-Party Root Certification Authorities** store. If importing the certificate for the root CA to the Windows Trusted Certification Authorities store doesn’t work, try the Third-Party Root Certification Authorities store.

**Enable SSL for the .NET Agent**

There are two ways to update the SSL settings for the agent, either using the AppDynamics Agent Configuration Utility or by editing the settings directly in the **config.xml**.
When you enable SSL for the .NET Agent, you automatically enable SSL for the .NET Machine Agent.

**Configure SSL Using the AppDynamics Agent Configuration Utility**

1. Launch the AppDynamics Agent Configuration utility.
2. In the Controller Configuration window, set the **Port Number** to the SSL port for the Controller.
   - For a SAAS Controller, set the **Port Number** to 443.
   - For an on-premises Controller, set the **Port Number** to the on-premises SSL port. The default is 8181.
3. Click **Enable SSL**.
   - When you enable SSL, the agent secures communication to the Controller using the protocols set for `ServicePointManager.SecurityProtocol` in your application.
   - By default, the configuration utility enables TLS 1.2, making it the first option in the list of secure protocols. This affects all secure communications from your application, not just requests to the AppDynamics Controller. If you want to disable TLS 1.2, click to deselect this option.
4. Click **Next** and proceed with the rest of the windows to complete the configuration.
5. Restart instrumented applications: IIS applications or application pools, Windows services, and standalone applications.

If you use automatic tier configuration, restart IIS. For example, open a command prompt and run:

```
iisreset
```

Upon restart, the agent connects with the Controller via SSL.

**Configure SSL Using config.xml**

1. Open the config.xml file as administrator. See [Administer the .NET Agent](#).
2. Update the following SSL settings:
   - Controller port attribute: set to the on-premises SSL port. The default is 8181. See [Controller port attribute](#).
   - Controller SSL attribute: set to true. See [Controller ssl attribute](#). When you enable SSL, the agent secures communication to the Controller using the protocols set for `ServicePointManager.SecurityProtocol` in your application.
   - Controller enable TLS 1.2 attribute: Optionally set to true to add TLS 1.2 as the first option in the list of protocols. This affects all secure communications from your application, not just requests to the AppDynamics Controller.
3. Save your changes.
5. Restart instrumented applications: IIS applications or application pools, Windows services, and standalone applications.

If you use Automatic configuration, restart IIS. For example, open a command prompt and run:

```
iisreset
```

Upon restart, the agent connects with the Controller via SSL.

**Sample SaaS SSL config.xml Configuration**
Sample On-Premises SSL config.xml Configuration

```xml
<?xml version="1.0" encoding="utf-8"?>
<appdynamics-agent
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <controller host="mycontroller.mycompany.com" port="8181"
    ssl="true" enable_tls12="true">
    <application name="MyDotNetApplication" />
  </controller>
...
</appdynamics-agent>
```

**Troubleshooting**

If you have verified all prerequisites, but have communication issues verify the default ciphers are enabled in Windows Server:

Check the following registry key:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\SecurityProviders\SCHANNEL\Ciphers
```

If subkeys exist, your operations team may have disabled certain ciphers.
Configure the .NET Agent for Windows Services and Standalone Applications

On this page:
- Before Starting
- Manually Configure the .NET Agent
- Sample Configuration File
- Troubleshooting

Related pages:
- Configure the .NET Agent
- .NET Agent Configuration Properties

The .NET Agent automatically instruments IIS applications only. However, you can follow the instructions in this topic to manually configure the .NET Agent to instrument Windows services or standalone .NET applications.

**Before Starting**

To instrument Windows services and standalone .NET applications, you edit the .NET Agent configuration file, `config.xml`.

Before starting, verify that the services or applications you want to instrument are .NET applications rather than native applications or another type of application. From a command line, run the following command:

```
tasklist /m "mscor*"
```

The output lists the processes that have DLLs starting with `mscor*`, indicative of .NET processes. Processes that are not on the list are not .NET processes and cannot be instrumented with the .NET Agent.

If you have previously instrumented IIS applications on the server that hosts the Windows services and standalone applications, the server should already have a `config.xml` file that you can edit. If not, perform the following steps to generate one:

1. Install the .NET Agent for Windows.
2. Run the AppDynamics Agent Configuration utility.

If you want to avoid instrumenting IIS applications, choose the manual tier naming approach and omit the step of assigning tiers for the IIS application. This disables instrumentation for the IIS applications, allowing you to instrument only the intended Windows services or standalone applications.

The utility performs these configuration tasks:

1. Changes the location of the logs directory and assign permissions.
2. Configures and tests connectivity to the Controller.
3. Sets the business application for the agent.

**Manually Configure the .NET Agent**

Once you have configured the Controller properties for the .NET Agent, instrument your Windows service or standalone application by adding the Standalone Applications element to the `config.xml`.

1. Edit the `config.xml` file as an administrator. See **Administer the .NET Agent**.
If you have not instrumented any IIS applications, the file contains the minimal configuration for the Controller connectivity and the machine agent. Verify the Controller properties and the Business Application name, as in the following example:

```xml
<?xml version="1.0" encoding="utf-8"?>
<appdynamics-agent
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <controller host="mycontroller.example.com" port="8090"
ssl="false">
      <application name="My Business Application" />
   </controller>
...
</appdynamics-agent>
```

If you have already instrumented IIS applications, those configurations appear under the IIS element.

2. Add a single standalone applications element, standalone-applications under the app-agents element, and under the standalone-applications element, add a standalone application element, standalone-application for each Windows service or standalone application to instrument. For example:

```xml
...<app-agents>
   ...
   ...
   <standalone-applications>
      <standalone-application executable="MyStandaloneApp.exe">
         <tier name="Standalone Tier 1" />
      </standalone-application>
      <!-- Instrument a standalone application using a partial path. -->
      <standalone-application executable="MyApplication\MyOtherStandaloneApp.exe">
         <tier name="Standalone Tier 2" />
      </standalone-application>
      <!-- Instrument a Windows service using arguments. -->
      <!-- The following example matches the command "MyWindowsService.exe -d -x -r". -->
      <standalone-application executable="MyWindowsService.exe" command-line="-x">
         <tier name="Windows Service Tier" />
      </standalone-application>
   </standalone-applications>
</app-agents>
```

In the standalone application element configuration:
- Use the tier element to assign the instrumented application to a tier in the Controller. See .NET Agent Configuration Properties.
- Identify the executable file of the application in the Standalone Application element executable attribute using one of the following formats:
  - Executable name: For example, MyStandaloneApp.exe or MyWindowsService.exe. The file extension is optional, so MyStandaloneApp also works.
  - Full or partial path to the executable: For example, C:\Program
Files\MyApplication\MyStandaloneApp.exe or MyApplication\MyStandaloneApp.exe. Use the full or partial path when you want to assign different AppDynamics tiers to separate instances of the same executable file running from different paths.

- To differentiate between two instances of the same executable, specify any unique portion of the command line invocation format of the application, such as an argument, in the Standalone Application command-line attribute.

You can discover the path to a Windows service executable in the Services panel of the administrative tools. In Services, click on the service and choose Properties. The path appears in the General tab.

4. Restart the Windows service or standalone application.
5. If your Windows service or standalone application does not implement an auto-detected framework, you must configure a POC O entry point for a class/method in your service for the agent to begin instrumentation.

Sample Configuration File

This sample config.xml demonstrates instrumentation for a Windows service and standalone application:

```xml
<?xml version="1.0" encoding="utf-8"?>
<appdynamics-agent
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <controller host="mycontroller.mycompany.com" port="8090"
 ssl="false">
    <application name="My Business Application" />
  </controller>
  <machine-agent />
  <app-agents>
    <IIS>
      <applications />
    </IIS>
    <standalone-applications>
      <standalone-application executable="MyWindowsService.exe"
 command-line="-x">
        <tier name="Windows Service Tier" />
      </standalone-application>
      <standalone-application executable="MyStandaloneApp.exe">
        <tier name="Standalone Tier" />
      </standalone-application>
    </standalone-applications>
  </app-agents>
</appdynamics-agent>
```

Troubleshooting

Intermittent Loss of Windows Services Instrumentation

When instrumenting Windows Services, there may be cases where the instrumented service initializes before the AppDynamics.Agent.Coordinator service. When this happens the Profiler shuts down and no instrumentation happens. This problem may present intermittently and is not common.

The following describes the procedures required to overcome this problem for Windows 2012 and later, and for Windows 2008.
Windows 2012 and later

To ensure the AppDynamics.Agent.Coordinator service initializes before the instrumented Windows Service, modify the registry as follows:

Edit the Reg key `EarlyStartServices @HKLM:\SYSTEM\CurrentControlSet\Control` and add `AppDynamics.Agent.Coordinator_service` to the list of early start services.

Windows 2008

If you are using Windows 2008, please contact AppDynamics Support for assistance.

Troubleshooting

Intermittent Loss of Windows Services Instrumentation

When instrumenting Windows Services, there may be cases where the instrumented service initializes before the AppDynamics.Agent.Coordinator service. When this happens the Profiler shuts down and no instrumentation happens. This problem may present intermittently and is not common.

The following describes the procedures required to overcome this problem for Windows 2012 and later, and for Windows 2008.

Windows 2012 and later

To ensure the AppDynamics.Agent.Coordinator service initializes before the instrumented Windows Service, modify the registry as follows:

Edit the Reg key `EarlyStartServices @HKLM:\SYSTEM\CurrentControlSet\Control` and add `AppDynamics.Agent.Coordinator_service` to the list of early start services.
Name .NET Tiers

On this page:

- Name IIS Tiers Automatically
- Name Azure Tiers Automatically
- Name IIS Tiers Manually
  - Use the configuration utility
  - Edit the config.xml directly
- Name Windows Service or Standalone Application Tiers

Related pages:

- Configure the .NET Agent
- Name .NET Nodes

In AppDynamics, a tier represents a service in your application environment, such as an ASP.NET front end, WCF service, .NET web service or standalone application. Tiers appear on flow maps, so you should name your tiers in a way that is logical and easy to understand for your users.

Use the AppDynamics .NET Agent Configuration Utility to map IIS sites to tiers, as described in Configure the .NET Agent. Some manual configuration options require you to edit the config.xml file, as described in 'Where to Configure Agent Properties' on Administer the .NET Agent.

Name IIS Tiers Automatically

Name IIS tiers automatically using the configuration utility. Choose Automatic on the Assign IIS applications to tiers window. The .NET Agent instruments all IIS sites except the Default Web Site and names the tiers using the following scheme:

IIS site/app

The agent omits app when the application is the root application for the IIS site.

Use this option in the following cases:

- you are new to AppDynamics and the .NET Agent.
- you want to instrument all IIS applications and your team understands the form of IIS application names.

Automatic Tier Naming for the .NET Agent doesn't automatically assign a virtual application to a tier. If you want to map virtual applications to tiers, see Name IIS Tiers Manually.

Name Azure Tiers Automatically

The .NET Agent automatically names Azure tiers using the following schemes:

- Cloud Services: Azure role name
- App Services: Azure site name

If you want to customize tier naming for Azure, see Edit the config.xml directly under Name IIS Tiers Manually.

Name IIS Tiers Manually

The .NET Agent offers two options for naming IIS tiers manually, using the configuration utility of editing the config.xml file directly.

Use the configuration utility

On the Assign IIS applications to tiers page, choose Manual. The configuration utility lets you create new tiers and assign IIS applications to tiers.
Use this option in the following cases:

- You do not want to instrument all IIS applications on the server
- You want custom tier names
- You want to assign multiple applications to a single tier

**Edit the config.xml directly**

Use this option to customize Azure tier names or in cases where it is not feasible to use the configuration utility.

For each IIS tier you want to instrument, add an application element as a child element of the IIS applications element in the config.xml file. You can specify a static IIS tier name or a regular expression which is helpful for variable Azure site names. For the full syntax and an example, see 'IIS Applications Element' on .NET Agent Configuration Properties.

For example, to add all Azure sites that begin with MvcWebRole to a tier named My Azure Tier using a regular expression:

```xml
<?xml version="1.0" encoding="utf-8"?>
<appdynamics-agent xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <controller host="nativedemocon1.cloudapp.net" port="8090" ssl="false">
    <application name="AzureEmailService" />
    <account name="customer1" password="SJ5b2m7d1$354" />
  </controller>
  <machine-agent />
  <app-agents azure="true" azure-role-name="" azure-role-instance-id="">
    <IIS>
      <applications>
        <!-- Configure IIS tier names with a regular expression. -->
        <application path="/" site="MvcWebRole.*" site-regex="true">
          <tier name="My Azure Tier" />
        </application>
      </applications>
    </IIS>
    <standalone-applications>
      <standalone-application executable="WaWorkerHost.exe">
        <tier name="" />
      </standalone-application>
    </standalone-applications>
  </app-agents>
</appdynamics-agent>
```

**Name Windows Service or Standalone Application Tiers**

See Configure the .NET Agent for Windows Services and Standalone Applications for information on naming tiers for each instrumented Windows service or standalone application manually in the config.xml.
Name .NET Nodes

On this page:
- IIS Node Naming
- IIS Web Gardens
- Windows Service or Standalone Application Nodes Naming

Related pages:
- Configure the .NET Agent
- Name .NET Tiers

This topic describes naming for .NET Agent nodes.

**IIS Node Naming**

By default, the AppDynamics .NET Agent names nodes using a combination of the Windows machine name, the tier name, and the name of the .NET application, as in the following format:

<machine NetBIOS name>--<tier>--<IIS site>/<app>

However, elements in the name may be omitted in these conditions:
- The app name is omitted when the application is the root application for the IIS site.
- The tier name is omitted when the tier name is the same as the IIS site name.

The following examples illustrate IIS node naming.

**Example 1**

WIN-86M7CEJO6P5-Order Server-OrderSvc

- WIN-86M7CEJO6P5 is the machine NetBIOS name.
- Order Server is the tier name.
- OrderSvc is the IIS site name. The application is the site root, so the agent omits the application name.

**Example 2:**

WIN-86M7CEJO6P5-Order Server-Store/ProcessOrder

- Store is the IIS site name.
- ProcessOrder is the application name within the site.

Different .NET versions of the same application have their own versions of the CLR and run on independent processes. Therefore the agent identifies the two processes as different nodes.

**IIS Web Gardens**

The syntax for web gardens is the same as IIS Nodes, except that the agent appends a zero-based process index to differentiate the worker processes. That is:

<machine NetBIOS name>--<tier>--<IIS site>/<app>--<process index>

When IIS first launches web garden processes, the agent assigns a sequential index to each process. However, when IIS recycles a process, the agent re-uses the available index freed by the terminated process. Therefore there is no correlation between the index sequence and the chronological start of the process.

Sometimes you may see more nodes than the maximum number of worker processes. This can happen when a long-running request prevents a process from shutting down before its replacement launches.

**Windows Service or Standalone Application Nodes Naming**

By default, the agent names Windows service and standalone application nodes as follows:

<machine NetBIOS name>--<tier>--<Windows service name or executable name>

The agent omits tier when the tier name is the same as the service name or executable name.

The following examples illustrate Windows service and standalone application naming.
Example 1:

WIN-86M7CEJ06P5-MyWindowsService

- WIN-86M7CEJ06P5 is the machine name.
- MyWindowsService is the Windows service name.

Example 2:

WIN-86M7CEJ06P5-MyStandaloneApp.exe

- WIN-86M7CEJ06P5 is the machine name.
- MyStandaloneApp.exe is the executable file name.
Unattended Installation for .NET

On this page:

- Create a Setup Configuration File
- Sample Setup Configuration File
- Unattended Installation
- Setup Configuration File Properties

Related pages:

- Configure the .NET Agent
- .NET Agent Configuration Properties
- Configure the .NET Agent for Windows Services and Standalone Applications

The .NET Agent provides a command-line unattended installation procedure for cases where you have multiple servers that require the same AppDynamics configuration. Using unattended installation, you only need to configure once, then use the command line scripts to automate installation and instrumentation on multiple servers.

For more detail about how to install and configure the agent manually, see Install the .NET Agent for Windows.

Create a Setup Configuration File

The .NET Agent MSI installer package allows you to specify the path to a setup configuration file to perform an unattended installation. The setup configuration file contains all the properties you need to enable instrumentation for your .NET applications.

You must run the .NET Agent MSI installer package on one machine before you can use the AppDynamics Agent Configuration utility to create a setup configuration file. See Install the .NET Agent for Windows.

Setup configuration files created in previous versions of the AppDynamics Agent Configuration utility work with the 4.0 installer.

1. Launch the AppDynamics Agent Configuration utility from the command line. Use the -s parameter to specify the setup configuration file destination.

```
%ProgramFiles%\AppDynamics\AppDynamics .NET Agent\AppDynamics.Agent.Winston.exe -s <path to setup configuration file>
```

For example:

```
%ProgramFiles%\AppDynamics\AppDynamics .NET Agent\AppDynamics.Agent.Winston.exe -s "c:\temp\configurationSavedSetupConfiguration.xml"
```

2. Go through the configuration wizard normally.

The configuration utility saves the setup configuration file to the path you specified.

```
The configuration utility only configures instrumentation for IIS applications.
```

3. Optional. To perform the unattended installation for Windows services or for standalone applications, you must edit the setup configuration file manually. See Configure the .NET Agent for Windows Services and Standalone Applications.

Sample Setup Configuration File

The following example shows a setup configuration file that instruments: two IIS Applications, MainBC and SampleHTTPService; a Windows service, MyWindowsService.exe; and a standalone application, MyStandaloneApp.exe.

The configuration file sets the log directory as `C:\ProgramData\AppDynamics\DotNetAgent\Logs` and grants write permission to
four accounts.

```xml
<winston>
  <logFileDirectory
directory="C:\ProgramData\AppDynamics\DotNetAgent\Logs" />
  <logFileFolderAccessPermissions defaultAccountsEnabled="false">
    <account name="NT AUTHORITY\LOCAL SERVICE" displayName="LOCAL SERVICE" />
    <account name="NT AUTHORITY\SYSTEM" displayName="SYSTEM" />
    <account name="NT AUTHORITY\NETWORK SERVICE" displayName="NETWORK SERVICE" />
    <account name="IIS_IUSRS" displayName="ApplicationPool Identity" />
  </logFileFolderAccessPermissions>
  <appdynamics-agent
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <controller host="mycontroller.mycompany.com" port="8090" ssl="false">
      <application name="My Business Application" />
    </controller>
    <machine-agent />
    <app-agents>
      <IIS>
        <applications>
          <application path="/" site="MainBC">
            <tier name="Main Site" />
          </application>
          <application path="/" site="SampleHTTPService">
            <tier name="HTTP Services" />
          </application>
        </applications>
      </IIS>
      <standalone-applications>
        <standalone-application executable="MyStandaloneApp.exe">
          <tier name="Standalone App" />
        </standalone-application>
        <standalone-application executable="MyWindowsService.exe">
          <tier name="Windows Service" />
        </standalone-application>
      </standalone-applications>
    </app-agents>
  </appdynamics-agent>
</winston>

Unattended Installation

Requirements
Microsoft Distributed Transaction Coordinator (MSDTC): MSDTC must run under the NT Authority\NetworkServices account. See ‘Verify MSDTC’ on Resolve .NET Agent Installation and Configuration Issues.

Windows Management Instrumentation (WMI)

Perform unattended installation

1. Launch an elevated command prompt with full administrator privileges. See Start a Command Prompt as an Administrator. Logging on to Windows as a member of the Administrators group does not grant sufficient permissions to run the installer.
2. Run the agent MSI installer package from the elevated command prompt. Use the AD_SetupFile parameter to pass the absolute file path to the setup configuration file.

   msiexec /i dotNetAgentSetup64.msi /q /norestart /lv %TEMP%\AgentInstaller.log AD_SetupFile=<absolute path to setup config.xml>

Optional Parameters

INSTALLDIR: Specify the directory to install the .NET Agent executables and supporting files.

DOTNETAGENTFOLDER: Specify the parent directory for local data including agent configuration files and log files.

For example:

   msiexec /i dotNetAgentSetup64.msi /q /norestart /lv %TEMP%\AgentInstaller.log AD_SetupFile=C:\temp\SetupConfig.xml INSTALLDIR=D:\AppDynamics DOTNETAGENTFOLDER=D:\AppDynamicsData

If you specify the DOTNETAGENTFOLDER on the command line and a logFileDirectory in the setup configuration file, the agent creates config and data folders under DOTNETAGENTFOLDER but writes the log files to the logFileDirectory specified in the setup configuration file.


   net start AppDynamics.Agent.Coordinator

4. Restart applications you have instrumented: IIS services, Windows services, and standalone applications.

   For example, to restart IIS:

   iisreset

Setup Configuration File Properties

Winston element

The Winston element is the root element of the configuration file.

Required element: <winston>

Log File Directory element

The Log File Directory element is a child element of the Winston element. Use the directory attribute to specify the log directory. If
you omit the logFileDirectory element, AppDynamics uses the default directory:

**Windows Server 2008 and later:** %ProgramData%\AppDynamics\DotNetAgent\Logs

**Optional element:** <logFileDirectory directory="C:\ProgramData\AppDynamics\DotNetAgent\Logs" />

**Log File Folder Access Permissions element**

The Log File Folder Access Permissions is a child element of the Winston element. Unless you set the defaultAccountsEnabled attribute to false, AppDynamics grants write access to the Logs folder for the default accounts:

- LOCAL SERVICE
- SYSTEM
- NETWORK SERVICE
- ApplicationPool Identity

**Optional element:** <logFileFolderAccessPermissions defaultAccountsEnabled="false" />

**Account element**

The Account element is a child element of the Log File Folder Access Permissions element. Create an Account element for the Windows account you use to run your application.

Set the name attribute to the name of the account you use to run your application, that is the account for the application pool for IIS or the Windows service account.

The display name attribute is a user-friendly name you choose for the account. The display name shows up in log entries about assigning permissions for the account.

**Optional element:** <account name="MyAppPoolIdentity" displayName="Custom ApplicationPool Identity" />

For example, if you run a Windows service under a domain account:

```
<account name="MYDOMAIN\service_acct" displayName="Domain Service Account" />
```

**AppDynamics Agent element**

The AppDynamics Agent element is a child of the Winston element. It follows the same format as the config.xml file to define the agent configuration for all your .NET applications. See .NET Agent Configuration Properties.

Upgrade the .NET Agent for Windows

On this page:

- Requirements
- Upgrade the .NET Agent from Version 3.9 or Newer
- Upgrade the .NET Agent from Versions 3.7.8 through 3.8.6
- Upgrade the .NET Agent from Version 3.7.7 or Earlier

Related pages:

- Agent and Controller Compatibility
- Install the .NET Agent for Windows
- Configure the .NET Agent
- .NET Agent Directory Structure
- Resolve .NET Agent Installation and Configuration Issues

The MSI installer package for the new version of the AppDynamics .NET Agent installs updated agent files and maintains legacy configurations. When you upgrade from versions of the .NET Agent older than 3.9, you must first uninstall the existing .NET Agent.

If you use the AppDynamics Azure Site Extension, see Install the AppDynamics Azure Site Extension.

Requirements

- Before you begin, review the Release Notes for changes that affect your environment.
- AppDynamics requires an account access key for agent connections to single-tenant Controller accounts. Versions of AppDynamics before 4.1 only required an account access key for multi-tenant Controller accounts.
  - Show how to find the access key...

Single-tenant Controller customers can find the account access key in the Controller under Settings > License > Account. You must be a member of a role with the View License account level permission, see Roles and Permissions.

The .NET Agent only supports in-place upgrade for the following versions:

- 3.9 or later.
- Major and minor releases after the currently installed version. For patch releases, you must uninstall the existing agent before upgrading.

For example, to install the patch release 4.1.0.1 over 4.1.0.0, uninstall the existing agent before upgrading.

Upgrade the .NET Agent from Version 3.9 or Newer

Except for patches to the current version, you do not need to uninstall the old agent first when you upgrade from the .NET Agent version 3.9 or later.
1. Stop w3wp processes for instrumented IIS applications. Stop instrumented Windows services or standalone applications.
2. Download the MSI installer package from the AppDynamics Download Center.
3. Launch an elevated command prompt with full administrator privileges. See Start a Command Prompt as an Administrator. Logging on to Windows as a member of the Administrators group does not grant sufficient permissions to run the installer.
4. If you use a single-tenant Controller account, in the Controller click Settings > License > Account to view your access key.
5. Run a command-line agent install. For single-tenant Controller accounts, specify your account access key using the AD_CONTROLLER_ACCOUNT_ACCESS_KEY parameter. For example:

```
msiexec /i "%USERPROFILE%\Downloads\dotNetAgentSetup.msi" /l log.txt /q AD_CONTROLLER_ACCOUNT_ACCESS_KEY=changeme
```

The installation runs silently in the background.

If you forget to add the account access key for a single-tenant Controller account, you can run the Agent Configuration Utility or manually add it to the config.xml later, see ‘Account Element’ on .NET Agent Configuration Properties.

6. Restart IIS:
   - Launch the AppDynamics Agent Configuration utility and click Restart IIS on the Configuration Summary window. OR
   - Execute iisreset from the command line.
   Restart Windows services and standalone applications.

Upgrade the .NET Agent from Versions 3.7.8 through 3.8.6

Uninstall the old version of the agent

1. Stop IIS, instrumented Windows services, and instrumented standalone applications.

   If you shut down IIS but continue to see active IIS Worker Processes, check the Application Pools pane in the IIS Manager and stop any started application pools.

   Failing to stop instrumented applications before uninstalling the .NET Agent may require you to reboot the machine.

2. Stop the AppDynamics.Agent.Coordinator service.
3. In the Control Panel, select Add/Remove Programs. Remove the AppDynamics .NET Agent.

   In some cases, another process interferes with the .NET Agent uninstallation process by locking the profiler.dll. If uninstallation fails, use a utility such as Process Explorer to see if a process is using profiler.dll. If so, terminate the process. Otherwise, try rebooting the machine. Then retry the uninstallation.

Install the new version of the agent

1. Stop w3wp processes for instrumented IIS applications. Stop instrumented Windows services or standalone applications.
2. Download the MSI installer package from the AppDynamics Download Center.
3. Launch an elevated command prompt with full administrator privileges. See Start a Command Prompt as an Administrator. Logging on to Windows as a member of the Administrators group does not grant sufficient permissions to run the installer.
4. If you use a single-tenant Controller account, in the Controller click Settings > License > Account to view your access key.
5. Run a command-line agent install. For single-tenant Controller accounts, specify your account access key using the AD_CONTROLLER_ACCOUNT_ACCESS_KEY parameter. For example:

```
msiexec /i "%USERPROFILE%\Downloads\dotNetAgentSetup.msi" /l log.txt /q AD_CONTROLLER_ACCOUNT_ACCESS_KEY=changeme
```

The installation runs silently in the background.
6. Restart IIS:
   - Launch the AppDynamics Agent Configuration utility and click **Restart IIS** on the Configuration Summary window.
   - OR
   - Execute `iisreset` from the command line.

   Restart Windows services and standalone applications.

**Upgrade the .NET Agent from Version 3.7.7 or Earlier**

Identify the right upgrade path based upon the method of tier naming and assignment (manual or automatic) and the type of application you instrument:

- If you use manual tier naming and assignment, the installer package upgrades configurations for IIS applications and Windows services.
- If you used automatic tier naming and assignment, run the configuration utility to update configurations.
- If you used standalone applications with 3.7.7 or earlier, follow the steps for standalone applications on [Configure the .NET Agent for Windows Services and Standalone Applications](#).

After installation, you may need to run the configuration utility to update your configuration and optionally remove legacy configurations.

**Uninstall the old version of the .NET Agent**

1. Stop IIS and instrumented Windows services.

   If you shut down IIS but continue to see active IIS Worker Processes, check the Application Pools pane in the IIS Manager and stop any started application pools.

   Failing to stop instrumented applications before uninstalling the .NET Agent may require you to reboot the machine.

2. Stop the `AppDynamics.Agent.Coordinator` service.
3. In the Control Panel, select Add/Remove Programs. Remove the `AppDynamics .NET Agent`.

   In some cases, another process interferes with the .NET Agent uninstallation process by locking the profiler.dll. If uninstallation fails, use a utility such as **Process Explorer** to see if a process is using profiler.dll. If so, terminate the process. Otherwise, try rebooting the machine. Then retry the uninstallation.

**Install the new version of the .NET Agent**

1. Stop `w3wp` processes for instrumented IIS applications. Stop instrumented Windows services or standalone applications.
2. Download the MSI installer package from the [AppDynamics Download Center](#).
3. Launch an elevated command prompt with full administrator privileges. See [Start a Command Prompt as an Administrator](#). Logging on to Windows as a member of the Administrators group does not grant sufficient permissions to run the installer.
4. Optional, if you use a single-tenant Controller account, in the Controller click **Settings > License > Account** to view your access key.
5. Run a command-line agent install. For single-tenant Controller accounts, specify your account access key using the `AD_CONTROLLER_ACCOUNT_ACCESS_KEY` parameter. For example:

   ```
   msiexec /i "%USERPROFILE%\Downloads\dotNetAgentSetup.msi" /l log.txt /q AD_CONTROLLER_ACCOUNT_ACCESS_KEY=changeme
   ```

   The installation runs silently in the background.

   If you forget to add the account access key for a single-tenant Controller account, you can run the Agent Configuration Utility or manually add it to the config.xml later, see 'Account Element' on [NET Agent Configuration Properties](#).
6. Restart IIS:
   - Launch the AppDynamics Agent Configuration utility and click **Restart IIS** on the Configuration Summary window.
   - Execute `iisreset` from the command line.
   - Restart Windows services.

If you used the following environment variables with the earlier version, the MSI installer migrates the configurations to the new configuration file:

- `AppDynamicsAgent_CallGraphOptions`
- `AppDynamicsAgent_DisableAppPools`
- `AppDynamicsAgent_EnableInProcesses`
- `AppDynamicsAgent_IgnoreCLREnv`
- `AppDynamicsAgent_Profiler_Classes`

Configure the .NET Agent

Configure the agent based on your method of tier generation and assignment: automatic or manual.

The .NET Agent configuration utility only supports configuration of one Controller per server. To configure multiple business applications, see Configure Multiple Business Application Support for .NET.

Configure the agent using automatic tier generation and assignment

If you used automatic configuration with the earlier version of the .NET Agent, run the configuration utility to configure the agent:

1. Use the .NET Agent Configuration utility to reconfigure instrumentation for IIS applications. Choose **Automatic** for the method of tier generation and assignment. See Configure the .NET Agent.
2. Configure instrumentation for Windows services manually. See Configure the .NET Agent for Windows Services and Standalone Applications.

Configure the agent using manual tier generation and assignment

For agents using manual tier generation and assignment, the installer package migrates the configurations for IIS applications and for Windows services to the config.xml. At this stage, the configuration for IIS applications and Windows services is complete.

Clean up legacy configurations

You can clean up legacy configurations by launching the AppDynamics Agent Configuration utility. When the utility detects agent settings from a previous version, it offers you the option to clean up.

The cleanup procedure modifies the web.config files causing an IIS restart.

1. Launch the AppDynamics Agent Configuration utility.
2. Answer **Yes** to clean up old AppDynamics configurations.
3. Proceed through the wizard normally.
   - Verify or update the log directory, and grant write permissions to it.
   - Verify the controller connection information.
   - Verify or update manual tier assignment.

The utility removes the following configurations:
- AppDynamics configSections from web.config files for IIS applications and application.config files for Windows services.
  Environment variables:
  - AppDynamicsAgent_IgnoreCLREnv
  - AppDynamicsAgent_CallGraphOptions
  - AppDynamicsAgent_EnableInProcesses
  - AppDynamicsAgent_DisableAppPools
  - AppDynamicsAgent_Profiler_Classes

Resume monitoring
Start IIS and instrumented Windows services.
Resolve .NET Agent Installation and Configuration Issues

On this page:
- Verify Agent Controller Communication
- Check Internet Explorer proxy settings
- Checklist for Resolving .NET Agent Installation Issues
- Resolve .NET Agent Installation Issues
- Resolve Configuration Errors
- Known Issues

Related pages:
- Install the .NET Agent for Windows
- Configure the .NET Agent
- Instrumenting SharePoint
- .NET Instrumentation Topics on the Community Knowledge Base

Verify Agent Controller Communication

Use the AppDynamics UI to verify that the agent can connect to the Controller.

1. In a browser open:

   http://<controller-host>:<controller-port>/controller

   If you cannot connect to the controller in Internet Explorer, see Check Internet Explorer proxy settings.
2. Log in to the AppDynamics UI.
3. Select the application to open the Application Dashboard.
4. In the left navigation panel click Servers > Tiers & Nodes and open the Health tab.

The Health tab lists the tiers, their nodes, and App Agent Status. When an agent successfully reports to the Controller, you see an up row symbol.

For details see Agent-to-Controller Connections.

- When deploying multiple agents for the same tier, determine whether you get the correct number of nodes reporting into the same tier.
- After sending a request to your web application, data should appear on the AppDynamics UI. The agents should display in the Application Flow Map of the Application Dashboard.

If no data appears after a few minutes:

- Verify that the Agent is writing its log files:
  Windows Server 2008 and later: %ProgramData%\AppDynamics\DotNetAgent\Logs\AgentLog.txt
- If the log file exists, open it and review it for errors.
If the log file does not exist, run the Windows Event Viewer and see the application messages.
If there are no AppDynamics event messages, look for messages from the .NET Runtime.

**Check Internet Explorer proxy settings**

This section is only for resolving issues connecting to the AppDynamics Controller when step 1 of 'Verify Agent Controller Connection' above fails. To configure the .NET Agent to work through a proxy, see 'Controller Element' on .NET Agent Configuration Properties.

Misconfigured proxy settings in Internet Explorer may cause the App Agent for .NET to fail to connect to the controller. If Test Controller connection fails on the Controller Configuration window in the AppDynamics Agent Configuration utility, do the following:

1. Verify the Controller host and port settings are correct.
2. In Internet Explorer, open:

   http://<controller-host>:<controller-port>/controller

3. If the connection also fails in Internet Explorer, check the proxy settings. See Change IE Proxy Settings.
4. Correct or remove any incorrect proxy settings.

**Checklist for Resolving .NET Agent Installation Issues**

<table>
<thead>
<tr>
<th>Item</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔️ Run the installer as Administrator.</td>
<td>Verify Administrative privileges</td>
</tr>
<tr>
<td>✔️ Verify that MSDTC is enabled and that it is running under the correct account.</td>
<td>Verify MSDTC is enabled and running under the correct account</td>
</tr>
<tr>
<td>✔️ Verify permissions for Agent directory.</td>
<td>Verify that the .NET Agent directory has the correct permissions based on the site’s application pool identity.</td>
</tr>
<tr>
<td>✔️ Verify that the Agent is compatible with the Controller.</td>
<td>Agent and Controller Compatibility</td>
</tr>
<tr>
<td>✔️ Verify the correct settings in the config.xml:</td>
<td>Update the config.xml file to include the .NET Agent Configuration Properties.</td>
</tr>
</tbody>
</table>

**Windows Server 2008 and later:**

%ProgramData%\AppDynamics\DotNetAgent\Config\config.xml

**Resolve .NET Agent Installation Issues**

If the Agent installation is failing, check the following configurations in your environment:

**Verify administrative privileges**

Ensure that you have the administrative privileges when you launch the installer. If the current user doesn't have sufficient privileges, the installer prompts you for an administrator password.

**Verify MSDTC is enabled and running under the correct account**

If you encounter an error that MSDTC is not enabled or it is running under the wrong account, launch an elevated command prompt with full administrative privileges and execute the following command:

msdtc -install

Even if MSTDC is already installed, this command resets the service to run using the "NT Authority\NetworkServices" account.
Generate a log for agent installation failures

If the installer fails, use the command line utility to launch the installer.

```
msiexec /i $Path_to_the_MSI_File /l*v verbose.log
```

A verbose log for the .NET Agent is created at the same location where you saved the installer file.

Correct failed installation caused by other APM products

The .NET Agent installation may fail if there are other Application Performance Management (APM) products installed in the same managed environment. Remove the associated Environment subkey for certain services for the installed APM products.

Remove associated Environment subkey for W3SVC and WAS services in the registry:

1. Run Regedit or regedit32.
2. In regedit.exe, locate the following registry keys:
   - HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\services\W3SVC
   - HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\services\WAS
3. Expand the keys.
4. Modify the Environment subkey to delete the following values:

   ```
   COMPLUS_ProfAPI_ProfilerCompatibilitySetting=EnableV2Profiler
   COR_ENABLE_PROFILING=1
   COR_PROFILER= {a GUID}
   ```
5. Restart the services. For more details see How to restart the W3SVC and WAS services?

Resolve Configuration Errors

- Set your IIS applications to a Full trust level. .NET Agent reporting will not work if an IIS application is set to anything other than a Full trust level.
- Ensure that you have correctly configured the config.xml file for the App Agent for .NET. For more detail, see .NET Agent Configuration Properties.
- If you made manual edits to the config.xml file, check the AgentLog.txt and file WarnLog.txt file for errors. Invalid XML shows in the log as follows:

```
2014-03-13 10:49:18.7199 1232 dllhost 1 1 Error
ConfigurationManager Error reading the configuration file
```

- Resolve Log Issues

The .NET Agent writes logs to the following directories:

```
%ProgramData%\AppDynamics\DotNetAgent\Logs
```

The agent will not generate logs if the agent directory does not have sufficient permissions.

Verify that the .NET Agent directory has the correct permissions

1. Click IIS > Application pools.
   IIS displays the list of application pools for your machine.
1. Right-click a particular application pool.

2. Click **Advanced Settings**.

3. Click **Advanced Settings** again.

IIS displays the Application Pool Identity for that application.
3. Ensure that your Agent Directory also has the same permissions as your site application pools.
   - Navigate to AppDynamics .NET App Server Agent directory location.
   - Right-click on the logs directory for the App Server Agent and select Properties.
4. Click the **Security** tab and verify that the same Application Pool Identity is specified for the .NET Agent directory.
If the Agent logs directory does not have the required permissions:

1. In the Security tab, click **Edit**.
2. Click **Add** to add new permissions to the Agent directory.
3. Click **Advanced**.
4. Click **Find Now** to find all the users, groups, or built-in security principals on your machine.
5. Select the required group from the list and click **OK**. See the information that follows for Allowed groups.
6. Provide the read and write permissions for the selected user/group/security principal to the Agent directory and click **OK**.
7. Click **Apply**.

**Allowed groups for different IIS versions**

For IIS v6.x, following settings are applicable for Application Pool Identities:

<table>
<thead>
<tr>
<th>Application Pool Identity</th>
<th>Permission Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>LocalService</td>
<td>LOCAL SERVICE</td>
</tr>
<tr>
<td>LocalSystem</td>
<td>SYSTEM</td>
</tr>
<tr>
<td>NetworkService</td>
<td>NETWORK SERVICE</td>
</tr>
<tr>
<td>Custom Account</td>
<td>Provide the exact name of the account.</td>
</tr>
</tbody>
</table>

For IIS v7.0 and later, following settings are applicable for Application Pool Identities:

<table>
<thead>
<tr>
<th>Application Pool Identity</th>
<th>Permission Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>LocalService</td>
<td>LOCAL SERVICE</td>
</tr>
<tr>
<td>LocalSystem</td>
<td>SYSTEM</td>
</tr>
<tr>
<td>NetworkService</td>
<td>NETWORK SERVICE</td>
</tr>
<tr>
<td>ApplicationPoolIdentity</td>
<td>Provide the group level permissions for IIS_IUSRS Group</td>
</tr>
<tr>
<td></td>
<td>See the screenshot that follows</td>
</tr>
<tr>
<td>Custom Account</td>
<td>Provide the exact name of the account.</td>
</tr>
</tbody>
</table>

For example, if your application has the identity `ApplicationPoolIdentity`, you must provide the permissions for `IIS_IUSRS` group to your agent directory.
**Known Issues**

You may ignore the following error from the AppDynamics.Agent.Coordinator service in the agent.log or warn.log:

```
2015-12-01 10:12:03.0435 3872 AppDynamics.Coordinator 1 7 Warn MaxQueueItemAgePolicy MaxQueueItemAge is beyond the integer limit: 13093467123
```
Instrumenting SharePoint

Related pages:
- Resolve .NET Agent Installation and Configuration Issues

Business operations must ensure their internal and external teams have full access to the content and collaborative services that SharePoint offers. To do that, administrators need an APM solution with total visibility across the SharePoint environment. AppDynamics works with SharePoint to give you the details for complex tasks such as end-user monitoring, log analytics, and server support.

Here are the benefits of using AppDynamics to monitor SharePoint:

- Using AppDynamics, you can monitor most SharePoint components (web applications, search components, workflow and Office Web Apps), using a mixture of .NET APM agents, EUM agents, analytics agents, and machine agents.
- AppDynamics can help you understand what is happening on the end user page and document navigation/browsing using End User Monitoring.
- AppDynamics APM partitions user activity by content web applications.
- AppDynamics APM can provide insights into Office Web Apps document rendering behavior.
- You can get more details on search and query usage.

AppDynamics is not a replacement for the Systems Center Operations Management (SCOM) pack or a well-planned IBM Tivoli implementation, particularly when just interpreting PerfMon counters. SharePoint Server 2013 integrates well with SCOM, which is very useful but still optional. Finally, AppDynamics is not meant to be a replacement for native monitoring tools but rather works together with tools from Microsoft and its robust partner ecosystem.

Download and read this eBook, and work with an AppDynamics expert in your organization, or AppDynamics Professional Services, before implementing this solution.
Uninstall the .NET Agent

Related pages:
- Upgrade the .NET Agent for Windows
- Release Notes

This topic describes how to do a complete uninstall of the .NET Agent. This procedure frees the license associated with the agent and makes it available for use by another app agent.

⚠️ Do not follow these instructions if you are doing an upgrade. If you want to upgrade to a new version see Upgrade the .NET Agent for Windows.

Completely uninstall the .NET Agent

1. Stop IIS, instrumented Windows services, and instrumented standalone applications.
   Failing to stop instrumented applications before uninstalling the agent requires you to reboot the machine to complete the uninstall.
2. Stop the AppDynamics.Agent.Coordinator service.
3. In the Control Panel, select Add/Remove Programs. Remove the AppDynamics .NET Agent.
4. The uninstall procedure does not remove configuration files. Delete the configuration directory: %ProgramData%\AppDynamics
5. Restart IIS, Windows services, and standalone applications.

✅ In some cases, another process interferes with the .NET Agent uninstallation process by locking the profiler.dll. If uninstallation fails, use a utility such as Process Explorer to see if a process is using profiler.dll. If so, terminate the process. Otherwise, try rebooting the machine. Then retry the uninstallation.
Encrypt Credentials in .NET Agent Configuration

By default the AppDynamics .NET Agent stores the following credentials in the config.xml file:

- Controller account authentication
- Proxy server authentication

For environments where security policies require you to secure credentials stored on disk, you can run an unattended installation that encrypts the credentials for the .NET Agent and writes them to the Windows Credential Manager.

**Requirements**

- You run the AppDynamics.Agent.Coordinator service as the LocalSystem account.
- To modify credentials after you install, you need Windows Sys internals.

**Setup Configuration File**

You must generate a setup configuration file in order to run an unattended installation. For a detailed description, see 'Setup Configuration File Properties' on Unattended Installation for .NET.

**New Installation**

For new installations, use one of the following methods to create the setup configuration file:

- Run the AppDynamics Agent Configuration utility from the command line and pass the -s parameter to specify the setup configuration file destination. For this options, you must execute the .NET Agent MSI installer package on one machine before running the configuration utility.

  ```
  %ProgramFiles%\AppDynamics\AppDynamics .NET Agent\AppDynamics.Agent.Winston.exe -s <path to setup configuration file>
  ```

- Manually create a setup configuration file from a sample template.

Remove any plain-text authentication elements from the setup configuration file. You pass the credentials as part of the unattended installation command:

- Controller Account element:
  ```
  <account name="myaccount" password="myaccesskey"/>
  ```
- Proxy Authentication element: If you are using a proxy authentication, use the following format in the setup configuration file:

  ```
  <proxy authentication="true" server="proxyserver" port="8080" username="username" password="password"/>
  ```

Storing credentials for the .NET Agent using the Windows Credential Manager updates the config.xml file to use a schema the AppDynamics Agent Configuration utility doesn't currently support.

If you follow these instructions, you can't use the configuration utility to make configuration changes afterward. If you launch the configuration utility on a server where you have stored credentials in the Windows Credential Manager, the utility will prompt you to delete the configurations.

**Related pages:**

- Unattended Installation for .NET
- Upgrade the .NET Agent for Windows
<proxy host="myproxy.example.com" port="3128" enabled="true">
  <authentication enabled="true" domain="mydomain.com"/>
</proxy>

Upgrade

If your upgrade meets the criteria for an in-place upgrade on Upgrade the .NET Agent for Windows, you can encrypt the credentials for the .NET Agent and upgrade the agent at the same time.

Copy the AppDynamics Agent element from your existing config.xml file to the setup configuration file. Remove any plain-text authentication elements from the setup configuration file. You pass the credentials as part of the unattended installation command:

- **Controller Account element:**
  <account name="mycontroller.saas.appdynamics.com" password="myaccesskey"/>

- **Proxy Authentication element:**
  <authentication enabled="true" user_name="my_proxy_user" password="password" domain="my_windows_domain"/>

Sample Setup Configuration File

The following example shows a setup configuration file that instruments: two IIS Applications, MainBC and SampleHTTPService; a Windows service, BasicWindowsService; and a standalone application, MyStandaloneApp.exe.
Install from the Command Line

To install the .NET Agent from the command line:
1. Download the .NET Agent MSI Installer Package from the AppDynamics Download Center.
2. Launch an elevated command prompt with full administrator privileges. See Start a Command Prompt as an Administrator.

   Logging on to Windows as a member of the Administrators group does not grant sufficient permissions to run the installer.

3. Stop IIS and, if you're upgrading, stop instrumented Windows services and Standalone applications.
4. Run the following command to install the agent with encrypted credentials. See below for a description of command line options.

   msiexec /i <path_to_MSI_installer_package> /l log.txt /q
   AD_SETUPFILE=<path_to_setup_configuration_file>
   AD_SECURED_CREDENTIALS=true AD_CONTROLLER_ACCOUNT_NAME=<SaaS or multi-tenant account> AD_CONTROLLER_ACCOUNT_ACCESS_KEY=<access key> AD_PROXY_USERNAME=<proxy user name>
   AD_PROXY_PASSWORD=<proxy password>

   For example:

   msiexec /i "%USERPROFILE%\Downloads\dotNetAgentSetup.msi" /l log.txt /q
   AD_SETUPFILE="%USERPROFILE%\Documents\SetupConfig.xml"
   AD_SECURED_CREDENTIALS=true
   AD_CONTROLLER_ACCOUNT_NAME=MyAppDynamicsAccount
   AD_CONTROLLER_ACCOUNT_ACCESS_KEY=changeme
   AD_PROXY_USERNAME=MyProxyUser AD_PROXY_PASSWORD=ProxyPass

   The MSI installer package installs the .NET Agent and encrypts the credentials and writes them to the Windows Credential Store. It adds the Controller secure attribute to the Controller element in the config.xml file and sets the value to true:

   <controller host="mycontroller.appdynamics.com" port="443"
   ssl="true" secure="true">

5. Start IIS. Restart or start instrumented Windows services and standalone applications.

**Update Credentials in the Windows Credential Manager**

The .NET Agent includes a Credentials Tool so you can modify credentials stored in the Windows Credential Manager. To change credentials under the Local System account, you need to use PsExec to launch the command prompt.

1. If you have not already, download and install Windows Sysinternals.
2. Use PsExec to launch a command prompt as the Local System account.

   psexec -i -s cmd.exe

3. Run the Credentials Tool and pass the updated credentials.
"%programfiles%\AppDynamics\AppDynamics .NET Agent\AppDynamics.CredentialsTool.exe"
AD_CONTROLLER_ACCOUNT_NAME=<SaaS or multi-tenant account>
AD_CONTROLLER_ACCOUNT_ACCESS_KEY=<access key>
AD_PROXY_USERNAME=<proxy user name> AD_PROXY_PASSWORD=<proxy password>

**Command Line Options**

- **AD_SECURED_CREDENTIALS**: Set to `true` to encrypt credentials to the Windows Credential Store and configure the agent to use the encrypted credentials.
- **AD_CONTROLLER_ACCOUNT_NAME**: The account name for the SaaS or multi-tenant Controller.
- **AD_CONTROLLER_ACCOUNT_ACCESS_KEY**: The account access key for the SaaS or multi-tenant Controller.
- **AD_PROXY_USERNAME**: The proxy server user account.
- **AD_PROXY_PASSWORD**: The password for the proxy server user account.
Administer the .NET Agent

On this page:
- Where to Configure Agent Properties
- Customize .NET Agent Behavior in config.xml
- Sample Minimal config.xml
- Agent Log Files

Related pages:
- .NET Agent Configuration Properties
- Manage Configuration for .NET

The .NET Agent uses a single configuration file to control agent behavior and .NET Machine Agent behavior. The configuration file specifies Controller connectivity, .NET Machine Agent operations, and app agent functionality for IIS applications, Windows services, and standalone applications. The single configuration file lets you:

- Maintain agent configurations separately from web.config files
- Enable instrumentation of Windows services and standalone applications without environment variables
- Control agent behavior for specific applications with hierarchical configuration

The AppDynamics Controller allows you to upload and manage .NET Agent configuration files. From the Controller, you can deploy a configuration to a machine or multiple machines where the agent is installed. See Manage Configuration for .NET.

Where to Configure Agent Properties

Configure the agent properties in the config.xml file in the agent Config directory. To edit the config.xml, you must launch the editor as an administrator. When you run the .NET Agent Configuration Utility, it writes the config.xml file to one of the following locations:

Windows Server 2008 and later

%ProgramData%\AppDynamics\DotNetAgent\Config\config.xml

Windows Azure

For Windows Azure deployments, the .NET Agent NuGet package contains the config.xml file.

Sample config.xml files install to the following location:

%ProgramFiles%\AppDynamics\AppDynamics .NET Agent\SampleConfigurations

After you edit the config.xml file, you must restart the AppDynamics.Agent.Coordinator service. Then restart your IIS services, Windows services or standalone applications for your instrumentation changes to take effect.

Customize .NET Agent Behavior in config.xml

Some .NET Agent configurations require that you edit the config.xml:

2. Edit the config.xml file as an administrator.
3. Modify the XML file according to the configuration instructions.
4. Save the config.xml file.
5. Start the AppDynamics.Agent.Coordinator service.
6. In some cases, you may need to restart IIS, instrumented Windows services, or instrumented Standalone applications. See individual .NET Agent administration topics.

Sample Minimal config.xml

The most basic configuration demonstrates the required sections for agent configuration. This sample instruments all IIS applications
using the automatic element, `<automatic />`. AppDynamics does not instrument Windows services or standalone applications in this case.

```xml
<?xml version="1.0" encoding="utf-8"?>
<appdynamics-agent
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <controller host="mycontroller.mycompany.com" port="8090"
  ssl=false>
    <application name="MyDotNetApplication" />
  </controller>
  <account name="customer1" password="changeme" />
  <machine-agent />
  <app-agents>
    <IIS>
      <automatic />
    </IIS>
    </app-agents>
</appdynamics-agent>
```

Agent Log Files

The configuration file that controls log files for the .NET Agent is located at:

```plaintext
%ProgramFiles%\AppDynamics\AppDynamics .NET Agent\AppDynamicsAgentLog.config
```

The configuration file uses NLog configuration format.
The following applies to Windows Server 2008 and later.

**Executables**

The .NET Agent executables and supporting files install to the AppDynamics .NET Agent directory:

```
%ProgramFiles%\AppDynamics\AppDynamics .NET Agent
```

**Log files**

The Logs directory defaults to the following location:

```
%ProgramData%\AppDynamics\DotNetAgent\Logs
```

**Agent Configuration Files**

The .NET Agent writes the config.xml configuration file to the Config directory:

```
%ProgramData%\AppDynamics\DotNetAgent\Config
```

If you use Config Management, the .NET Agent stores backup configuration files in the Config\Backup directory.

**Coordinator Service Configuration Files**

The AppDynamics Agent Coordinator service writes configuration files from the Controller to the Data directory:

```
%ProgramData%\AppDynamics\DotNetAgent\Data
```

**About Windows Server System Directory Variables**

%ProgramData% is located at <system drive>\Program Data.
.NET Agent Configuration Properties

On this page:
- AppDynamics Agent Element
- Controller Element
- Machine Agent Element
- App Agents Element
- Default Profiler Element
- App Agents - IIS Element
- App Agents - Standalone Applications Element

Related pages:
- Administer the .NET Agent
- Configure the .NET Agent
- Name .NET Tiers
- Configure the .NET Agent for Windows Services and Standalone Applications
- .NET Agent for Linux Environment Variables

This topic is a reference for the configuration properties for the .NET Agent that you set in the agent config.xml file. Some of these properties can be set using Windows System Environment Variables. See .NET Agent Environment Variables. For information about how to edit the file and apply your changes, see Administer the .NET Agent.

Reference of Environment Variables
You can use environment variables in the agent config.xml file for node, tier, and application names. To include an environment variable, reference it by specifying %<variable>%, which can be combined with the other characters; for example, Web-%COMPUTERNAME% will be translated to Web-HOST23.

AppDynamics Agent Element
The appdynamics-agent element is the root container element for configurations in the config.xml.

Required Element:

```xml
<appdynamics-agent
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

Controller Element
The controller element is a child of the appdynamics-agent element. It specifies the connection information for the AppDynamics Controller.

The .NET Agent configuration utility only supports configuration of one Controller per server.

Required Element:

```xml
<controller host="mycontroller.mycompany.com" port="8090"
  ssl="false" enable_tls12="false"
  high_availability="false" secure="false" enable_config_deployment="true"`
```
Controller host attribute

The controller host attribute indicates the hostname or the IP address of the AppDynamics Controller. For an on-premises Controller, use the value for Application Server Host Name you provided when you installed the Controller. If you use the AppDynamics SaaS Controller, see the Welcome email from AppDynamics.

Type: String
Default: None
Required: Yes

Environment Variable: APPDYNAMICS.CONTROLLER.HOSTNAME

Controller port attribute

The controller port attribute specifies the HTTP or HTTPS port of the AppDynamics Controller. If the controller ssl attribute is set to true, specify the HTTPS port of the Controller; otherwise, specify the HTTP port.

Type: Positive Integer
Default: 8090

- For On-premises installations, the defaults are port 8090 for HTTP and port 8181 for HTTPS.
- For the SaaS Controller, use port 80 for HTTP or port 443 for HTTPS.

Required: Yes

Environment Variable: APDDYNAMICS.CONTROLLER.PORT

Controller ssl attribute

To enable encryption over SSL between the agent and the Controller, set the controller ssl attribute to true.

Type: Boolean
Default: false
Required: No

Environment Variable: APDDYNAMICS.CONTROLLER.SSL.ENABLED

Controller enable TLS 1.2 attribute

When you enable SSL, the agent secures communication to the Controller using the protocols set for ServicePointManager.SecurityProtocol in your application. Set the controller enable TLS 1.2 attribute to true to add TLS 1.2 as the first option in the list of protocols. This affects all secure communications from your application, not just requests to the AppDynamics Controller.

Type: Boolean
Default: false
Required: No

Controller high availability attribute

If you have your Controller setup for high availability, set the controller high availability attribute to true.

Type: Boolean
Default: false
Required: No

Controller secure attribute

If you set up the .NET Agent to encrypt credentials, the MSI installer package automatically sets the controller secure attribute to true. When controller secure is true, the .NET Agent ignores any credentials in the Account element or the Proxy element and uses the credentials from the Windows Credential Store. See Encrypt Credentials in .NET Agent Configuration

Type: Boolean
Default: false
Required: No

Controller enable config deployment attribute
By default, the .NET Agent checks for configuration updates to the config.xml from the Controller. If you want to disable the agent from config.xml configuration files from the Controller set the controller enable config deployment attribute to false. See Manage Configuration for .NET

Type: Boolean  
Default: true  
Required: No

Controller Application Element

The controller application element is a child of the controller element. It indicates the name of the logical business application you see in the Controller interface.

The .NET Agent configuration utility only supports one business application per server. Use one of the following methods to support multiple applications:

- Use tiers to organize different applications you instrument on a single server.
- Or manually configure support for multiple business applications, see Configure Multiple Business Application Support for .NET.

Multiple application support has requires different configuration elements, including the <applications> container element.

Required Element: <application name="MyDotNetApplication"/>

Application name attribute

Set the application name attribute to the business application you use in the Controller. If the application name does not exist, the Controller will create it when the agent registers. All instrumented applications in the config.xml register with the same business application in the Controller. See Overview of Application Monitoring.

Type: String, may also reference an environment variable. See ‘Reference of Environment Variables’  
Default: None  
Required: Yes

Environment Variable: APPDYNAMICS_AGENT_APPLICATION_NAME

You specify a Tier for individual applications in the App Agents Element.

Account Element

The account element is a child of the controller element. If the AppDynamics Controller runs in multi-tenant mode or if you use the AppDynamics SaaS Controller, specify the account name. For single-tenant accounts the default account name is customer1. If you are using the AppDynamics SaaS Controller, the account name is provided in the Welcome email from AppDynamics.

The agent requires to use your account access key as a password to authenticate with the Controller. For single-tenant accounts, you can view the access key in the Controller under Settings > License. If you are using the AppDynamics SaaS Controller, the account access key is provided in the Welcome email from AppDynamics.

Required Element: <account name="mycompany" password="myaccesskey"/>

Account name attribute

The account name attribute indicates the account name for the SaaS or multi-tenant Controller.

Type: String  
Default: For single-tenant Controllers, the agent assumes the default of customer1 if you do not specify an account name.  
Required: Only for SaaS or multi-tenant Controllers.

Environment Variable: APPDYNAMICS.AGENT.ACCOUNTNAME

Account password attribute

The account password attribute indicates the account access key for the Controller.

Type: String  
Default: None  
Required: Yes

Environment Variable: APPDYNAMICS.AGENT.ACCOUNTACCESSKEY
Proxy Element
The proxy element is a child of the controller element. Use it to configure the connection to the Controller through a proxy server with no authentication.

Optional Element: `<proxy host="proxy-name" port="3128" enabled="true"/>

Proxy host attribute
The proxy host attribute indicates the proxy server hostname or IP address.
Type: String
Default: None
Required: host is required for the proxy element

Proxy port attribute
The proxy port attribute indicates the proxy server port.
Type: Positive Integer
Default: None
Required: port is required for the proxy element

Proxy enabled attribute
To enable Controller access through a proxy server, set the proxy enabled attribute to true.
Type: Boolean
Default: true
Required: No

Proxy - Authentication Element
The authentication element is a child of the proxy element. Use the authentication element to enable proxy authentication and to supply the credentials for your proxy server. For environments where security policies require you to secure credentials stored on disk, you can encrypt the credentials and store them in the Windows Credential Manager.

Optional Element: `<authentication enabled="true" user_name="my_proxy_user" password="password" domain="my_windows_domain"/>

Proxy authentication enabled attribute
Set the proxy authentication enabled attribute to true to configure the agent to send credentials to a proxy server.
Type: Boolean
Default: false
Required: No

Proxy authentication username attribute
The proxy authentication username attribute indicates the name of the proxy user.
Type: String
Default: None
Required: The username attribute is required for the Authentication element

Proxy authentication password attribute
The proxy authentication password attribute indicates the password for the proxy user.
Type: String
Default: None
Required: No

Proxy authentication domain attribute
The proxy authentication domain attribute specifies the domain or realm where the username is located. This could be the host computer name, an Active Directory domain or DNS domain.
Sample Controller Configuration with Proxy Authentication enabled

In this example, the agent accesses the controller through the proxy, myproxy.example.com using proxy authentication. The .NET Agent uses the credentials MyProxyUser@mydomain.com with password password to access the proxy server.

```xml
<controller host="mycontroller.example.com">
  <application name="MyDotNetApplication" />
  <proxy host="myproxy.example.com" port="3128" enabled="true">
    <authentication enabled="true" user_name="MyProxyUser" password="password" domain="mydomain.com"/>
  </proxy>
</controller>
```

**Machine Agent Element**

The `machine-agent` element is a child of the AppDynamics Agent element. An empty `machine-agent` element enables the default instrumentation for the .NET machine agent. See Monitor CLRs and Monitor IIS.

Enable optional additional Microsoft Performance Counters or .NET Agent instrumentors as children of the Machine Agent element.

Required Element: `<machine-agent/>`

**Machine Snapshot Element**

The `machine-snapshot` element is a child of the `machine-agent` element. Use it to tune the settings for machine snapshots in your environment. If you do not specify an attribute, the agent uses the default values for that attribute. See Machine Snapshots for .NET and Configure Machine Snapshots for .NET.

Optional Element: `<machine-snapshot enabled="true" window-size="600" samples-per-window="60" violations-per-window="6" max-percent-cpu="80" max-percent-memory="80" max-queue-item-age="100" periodic-collection="600"/>

**Machine Snapshot enabled attribute**

Set the `machine-snapshot enabled` attribute to `false` to disable machine snapshots.

Type: Boolean
Default: true
Required: No

**Machine Snapshot window size attribute**

Specify the window size time range in seconds for the .NET Machine Agent to take samples. During a window, the agent takes samples and checks them for breached thresholds: max percent CPU, max percent memory, and max queue item age.

Type: Integer
Default: 600
Required: No

**Machine Snapshot samples per window attribute**

...
Specify the number of samples the .NET Machine Agent takes during the specified window. For example, if the window size is 600 and the samples per window is 60, the agent takes a sample once every 10 seconds.

Type: Integer
Default: 60
Required: No

**Machine Snapshot violations per window attribute**

When the .NET Machine Agent detects the number of violations per window for one threshold, it takes a snapshot. The agent only takes one snapshot per window for thresholds exceeded. For example, if violations per window is set to 6, and six samples show memory usage at 80% or greater, the agent takes a snapshot. The counters for each threshold are separate.

Type: Integer
Default: 6
Required: No

**Machine Snapshot max percent memory attribute**

When the .NET Machine agent detects memory usage on the machine equals or exceeds the max percent memory value, it flags the sample as a violation. The minimum value is 20. The maximum value is 100.

Type: Integer
Default: 80
Required: No

**Machine Snapshot max percent cpu attribute**

When the .NET Machine agent detects cpu usage on the machine equals or exceeds the max percent cpu value, it flags the sample as a violation. The minimum value is 20. The maximum value is 100.

Type: Integer
Default: 80
Required: No

**Machine Snapshot max queue item age attribute**

When the .NET Machine agent detects the oldest item in the IIS queue equals or exceeds the max queue item age value in milliseconds, it flags the sample as a violation.

Type: Integer
Default: 100
Required: No

**Machine Snapshot periodic collection attribute**

The .NET Machine agent takes one snapshot per periodic collection time range. Specify the value in seconds. The minimum is 60 seconds.

Type: Integer
Default: 600
Required: No

**CLR Crash Reporting Element**

The `clrcrash-reporting` element is a child element of the `machine-agent` element. Use the `clrcrash-reporting` element to control whether or not the .NET Machine Agent reports CLR crash events to the Controller. See Monitor CLR Crashes.

Optional Element: `<clrcrash-reporting enabled="true"/>`

**CLR Crash Reporting enabled attribute**

Set the `clrcrash-reporting enabled` attribute to false to stop reporting CLR Crash events to the controller.

Type: Boolean
Default: true
Required: No
Process Monitor Element

The `process-monitor` element is a child element of the `machine-agent` element. By default, the agent enables process monitoring for all IIS processes.

Optional Element: `<process-monitor report-all-iis-processes="true"/>`

**Process Monitor report all IIS processes attribute**

Set the `report-all-iis-processes` attribute to `false` to enable process monitoring only for instrumented IIS processes.

Type: Boolean  
Default: true  
Required: No

Metrics Element

The `metrics` element is a child element of the `machine-agent` element. By default, the machine agent registers a maximum of 200 metrics. See Metrics Limits. Use the `metrics` element to increase the number of metrics the .NET Machine Agent can register.

Use caution when increasing the metric registration limits. Increasing the limit can increase the resource overhead for agents and Controller.

Optional Element: `<metrics max-metrics="200"/>`

**Metrics max-metrics attribute**

Specify the maximum number of metrics the .NET Machine Agent can register.

Type: Integer  
Default: 200  
Required: No

Perf-metrics Element

The `perf-metrics` element is a child element of the `metrics` element. By default, the machine agent collects and reports a full set of performance metrics. You can modify the priority level for the performance metrics to limit the metrics the agent collects and thereby decrease agent overhead. Refer to Manage Windows Performance Metrics for a full list of metrics and their priorities.

Optional Element: `<perf-metrics priority-level="3"/>`

**Perf-metrics priority-level attribute**

Specify the set of performance metrics for the machine agent to collect as follows:

- 0 - disable metric collection  
- 1 - collect only high priority metrics  
- 2 - collect high and medium priority metrics  
- 3 - collect all metrics: high, medium, and low priority

Type: Integer in the range 0 - 3  
Default: 3  
Required: If you use the `perf-metrics` element.

Perf-metric Element

The `perf-metric` element is a child element of the `perf-metrics` element. If you set the `perf-metric` priority to 0 to disable general collection of performance metrics, you can use the `perf-metric` element to enable individual performance metrics. You cannot enable individual metrics with `perf-metric` priority greater than 0.

Optional Element: `<perf-metric name="<metric_path>"/>`

**Perf-metric name attribute**

Specify the full path of the performance metric. See the example below for the % Busy metric and the Errors Unhandled During Execution metric.

Type: String: metric path
Default: N/A
Required: If you use the perf-metric element.

**Sample Machine Agent Configuration with Individual Performance Metrics**

```xml
<machine-agent>
  <metrics>
    <!-- Disable collection of performance metrics in general. -->
    <perf-metrics priority-level="0">
      <!-- Enable collection of individual performance metrics. -->
      <perf-metric name="Hardware Resources|CPU|%Busy"/>
      <perf-metric name="ASP.NET Applications|Errors Unhandled During Execution"/>
    </perf-metrics>
  </metrics>
</machine-agent>
```

**Performance Counters Element**

The `perf-counters` element is a child of the `machine-agent` element. It is a container for all performance counters.

Optional Element: `<perf-counters>`

**Performance Counter element**

The `perf-counter` element is a child of the `perf-counters` element. For a list of performance counters to enable, see Performance Counters in the .NET Framework.

Optional Element: `<perf-counter cat="category" name="name" instance="instance"/>`

**Performance Counter `cat` attribute**

The `cat` attribute indicates the performance counter category.

Type: String
Default: None
Required: Category is required for the `perf-counter` element.

**Performance Counter `name` attribute**

The `name` attribute indicates the performance counter name.

Type: String
Default: None
Required: Name is required for the `perf-counter` element.

**Performance Counter `instance` attribute**

The `instance` attribute is the performance counter instance value.

Type: String
Default: None
Required: Instance is required for the `perf-counter` element.

**Sample Machine Agent Configuration with Performance Counters**
<machine-agent>
  <!-- Additional machine level Performance Counters -->
  <perf-counters>
    <perf-counter cat="Network Interface" name="Bytes Sent" instance="Local Area Connection"/>
  </perf-counters>
</machine-agent>

Instrumentation Element

The instrumentation element is a child of the machine-agent element. It allows you to enable additional .NET Agent instrumentors such as thread correlation or correlation for .NET remoting.

Optional Element: <instrumentation>

Instrumentor Element

The instrumentor element is a child of the instrumentation element. The instrumentor element specifies the .NET Agent instrumentor to implement.

Optional Element: <instrumentor name="instrumentor name" enabled="true"/>

Instrumentor name attribute

The instrumentor name attribute indicates the instrumentor name.

Type: String
Default: None
Required: Name is required for the Instrumentor element.

Instrumentor enabled attribute

Set the instrumentor enabled attribute to true to enable instrumentation.

Type: Boolean
Default: false
Required: No.

The current configuration syntax is enabled="true". Versions prior to 3.7.8 used disabled="false".

Sample Machine Agent Configuration with Thread Correlation Instrumentors

<machine-agent>
  <!--Enable thread correlation-->  
  <instrumentation>
    <instrumentor name="ThreadCorrelationThreadPoolCLR2Instrumentor" enabled="true"/>
    <instrumentor name="ThreadStartCLR2Instrumentor" enabled="true"/>
    <instrumentor name="ThreadStartCLR4Instrumentor" enabled="true"/>
  </instrumentation>
</machine-agent>

Additional .NET Agent Instrumentors
See Thread Correlation for .NET
See Enable Correlation for .NET Remoting
See Enable Instrumentation for WCF Data Services

App Agents Element

The app-agents element is a child of the appdynamics agent element. It is a container for app agent configurations for IIS applications, Windows services, and standalone applications.

Required Element: <app-agents enabled="true"/>

App agents enabled attribute

To disable application monitoring on the server, set the app-agents enabled attribute to false.

Type: Boolean
Default: true
Required: No

Default Profiler Element

The default profiler element is a child of the app-agents element. It defines customizations to the default profiler behavior for all instrumented .NET applications on the machine: IIS applications, application pools, Windows services, and standalone applications.

Optional Element: <profiler>

Profiler - Disabled Features Element

The disabled-features element is a child of the profiler element. Use this property to disable data collection mechanisms at the agent level for security or privacy reasons. This configuration overrides any configuration set by the Controller.

Optional Element: <disabled-features value="NONE"/>

Disabled features value attribute

The disabled-features value attribute is a comma-separated list of features to disable. The available values are as follows:

- LOG_PAYLOAD: Override the log-request-payload node property to suppress logging HTTP request payloads
- RAW_SQL: Override the capture-raw-sql node property to suppress logging raw sql output
- CUSTOM_EXIT_SNAP_DATA: Suppress snapshot data from custom exits points
- METHOD_INV_DATA_COLLECTOR: Suppress method invocation data collector user data
- HTTP_DATA_COLLECTOR: Suppress HTTP request data collector user data
- INFO_POINT: Suppress information point metrics
- ALL: Disable all the available features
- NONE: Don't disable features

Type: String
Default: NONE
Required: No

Profiler - Successful Exit Code Element

The successful-exit-code element is a child of the profiler element. The default successful exit code determines whether or not the agent flags a CLR restart event as graceful or not for Windows services or standalone applications. This configuration does not apply to IIS.

Optional Element: <successful-exit-code value="0"/>

Successful exit code value attribute

Type: Integer
Default: 0
Required: No
Profiler - Runtime Reinstrumentation Element

The `runtime-reinstrumentation` element is a child of the `profiler` element. Use this property to configure runtime reinstrumentation. See Configure Runtime Reinstrumentation for .NET

Optional Element: `<runtime-reinstrumentation enabled="true" interval="60000" optimize="true" />`

Runtime reinstrumentation enabled attribute

Set to `true` to enable runtime reinstrumentation.

Type: Boolean
Default: false
Required: No

Runtime reinstrumentation interval attribute

The frequency in milliseconds that the agent checks for configuration updates that initiate runtime reinstrumentation.

Type: Integer
Default: 60000 milliseconds
Minimum: Because runtime reinstrumentation adds a small amount of system overhead, AppDynamics recommends a minimum interval of 1 minute or 60000 milliseconds.
Required: No

Sample Default Profiler Configuration

```xml
<app-agents>
  <profiler>
    <disabled-features value="LOG_PAYLOAD,RAW_SQL,CUSTOM_EXIT_SNAP_DATA"/>
    <!-- Set the successful exit code for Windows services and standalone applications to "1." -->
    <successful-exit-code value="1"/>
    <!-- Enable Runtime reinstrumentation -->
    <runtime-reinstrumentation enabled="true" interval="60000"/>
  </profiler>
  ...
</app-agents>
```

App Agents - IIS Element

The `IIS` element is a child of the `app-agents` element. There are three options to configure IIS applications:

- Automatic configuration
- Application pool configuration
- Application configuration

The settings for any application pool apply to all applications within the app pool unless the individual application has a specific configuration.

Explicit child-level configurations override parent-level configurations. Otherwise, children inherit parent configurations.

Optional Element: `<IIS>`

Exclude child applications attribute

By default, when you instrument an IIS application, the .NET Agent instruments any child applications and assigns them to the same tier as the parent. To prevent the agent from automatically instrumenting child applications, set `exclude-child-applications` to `true`. For instance: `<IIS exclude-child-applications="true"/>`
If you disable instrumentation for child applications in general, you can instrument specific child applications using the IIS Application element.

Type: Boolean
Default: false
Required: No

**IIS Automatic Instrumentation Element**

The `automatic` element is a child of the IIS element. Use the `automatic` element to enable or disable automatic instrumentation for all IIS apps. You can configure automatic instrumentation and manual instrumentation both. Manual configurations override automatic ones.

Optional Element: `<automatic enabled="false" />`

**Automatic enabled attribute**

Set the `automatic` `enabled` attribute to `true` to enable instrumentation for all IIS applications. This is the default setting if you use the .NET Agent Configuration Utility automatic configuration option. To disable automatic instrumentation for all IIS applications, set the value to `false`.

Type: Boolean
Default: true
Required: No

**IIS Application name enabled attribute**

By default, the agent does not report an IIS application name. To view the IIS application name in the CLR metadata, set the `iis-application-name-enabled` attribute to `true`. For example, `<automatic iis-application-name-enabled="true" />`. When enabled, the name appears as the `iis-application-name` value in the CLR metadata.

Type: Boolean
Default: false
Required: No

**IIS Application Pools Element**

The `application-pools` element is a child of the IIS element. It is a container element for all the IIS application pools you configure for instrumentation.

Optional Element: `<application-pools>`

**IIS Application Pool Element**

The `application-pool` element is a child of the `application-pools` element. You may have multiple application pool elements distinguished by the name attribute. Use the application pool element to configure the app agent for all applications within an application pool. For more information on IIS application pools, see [Managing Application Pools in IIS](#).

⚠️ Application-specific configurations in the IIS Applications element override application pool configurations.

Optional Element: `<application-pool name="DefaultAppPool" enabled="false"/>

**Application pool name attribute**

The `application-pool` `name` attribute indicates the name of the IIS Application Pool.

Type: String
Default: None
Required: Name is required for the application-pool element.

**Application pool enabled attribute**

Set the `application-pool` `enabled` attribute to `false` to disable instrumentation for all applications in the application pool. Set the value to `true` to instrument all applications in the application pool.

Type: Boolean
Default: None. Defaults to `true` if not specified.
Required: No
Application Pool Tier Element

The `tier` element is a child of the `application-pool` element. If you enable instrumentation for an application pool, you must use a `tier` element to assign the applications of a pool to a tier in the Controller. See Overview of Application Monitoring

Required Element: `<tier name="Inventory" />`

**Tier name attribute**

Use the `name` attribute to specify the tier.

Type: String, may also reference an environment variable. See ‘Reference of Environment Variables’
Default: None
Required: Yes

IIS Applications Element

The IIS `applications` element is a child of the `IIS` element. It is a container element for all the IIS applications you configure for instrumentation.
Optional Element: `<applications>`

Application Element

The `application` element is a child of the `applications` element. Use multiple application elements to instrument different sites and applications. To learn more about IIS sites and applications, see Understanding Sites, Applications, and Virtual Directories on IIS 7 and Above

Optional Element: `<application path="/" site="FirstSite" port="8008" site-regex="false">`

**Application site attribute**

The `site` attribute indicates the root site in IIS for the application. The site name accepts a regular expression for cases like Windows Azure where you may only know a partial site name. If you use a regular expression, set the Application site-regex attribute to true.

Type: String
Default: None
Required: Site is required for the Application element.

**Application site-regex attribute**

Set the `site-regex` attribute to true to treat the value of the Application site attribute as a regular expression.

Type: Boolean
Default: false
Required: No

**Application path attribute**

The `path` attribute indicates the path of the application, relative to the root site. Use the forward slash to indicate the root site and instrument all children applications. Use the path to an application to instrument the specific application and any children.

For example, Site1 hosts two applications AppX and AppY. To instrument Site 1, AppY and AppZ, set the path to "/". To instrument AppY, but not AppZ, set the path to /AppY.

Type: String
Default: /
Required: path is required for the application element.

**Application port attribute**

For cases where two or more sites in IIS 6 have the same site name, set the site `port` attribute to differentiate between the sites.

Type: Positive Integer
Default: None
Required: No

**Application enabled attribute**
In certain cases you may want to enable instrumentation for a parent application, but disable it for a child application. In this case create an application element for the child application to disable and set the application enabled attribute to false.

Type: Boolean  
Default: true  
Required: No

**Application Tier Element**

The tier element is a child of the application element. If you enable instrumentation for an application, you must use a tier element to assign the application to a tier in the Controller. See Overview of Application Monitoring

Required Element: `<tier name="Consumer" />`

**Tier name attribute**

The tier name attribute indicates the business application tier.

Type: String, may also reference an environment variable. See 'Reference of Environment Variables'

Default: None  
Required: Yes

**Sample IIS Application Configuration**
<IIS>
  <!-- Automatic instruments all IIS applications when enabled. -->
  <automatic enabled="false" />

  <!-- Application Pool agent configurations -->
  <application-pools>
    <!-- Do not instrument applications in DefaultAppPool when "enabled" attribute is set to false. -->
    <application-pool name="DefaultAppPool" enabled="false">
      <tier name="Tier Name"/>
    </application-pool>

    <!-- Instrument applications in the OtherAppPool and assign them to the Inventory tier. -->
    <application-pool name="OtherAppPool">
      <tier name="Inventory"/>
    </application-pool>
  </application-pools>

  <applications>
    <!-- Instrument all applications in the First Site. -->
    <application path="/" site="FirstSite">
      <tier name="Order"/>
    </application>

    <!-- Instrument the /app application and child apps in the Second Site -->
    <application path="/app" site="SecondSite">
      <tier name="Consumer"/>
    </application>

    <!-- Regular expression for site name assigns all sites beginning with "MyRole" to the Credit Services tier. -->
    <application path="/\w+" site-regex="true">
      <tier name="Credit Services"/>
    </application>
  </applications>
</IIS>

**App Agents - Standalone Applications Element**

The `standalone-applications` element is a child of the `app-agents` element. It is a container element for all the Windows services and standalone applications you configure for instrumentation. See [Configure the .NET Agent for Windows Services and Standalone Applications](#) for detailed instructions.

Optional Element: `<standalone-applications>`

**Standalone Application Element**
The standalone-application element is a child of the standalone-applications element. It specifies a Windows service or standalone application to instrument.

Optional Element: `<standalone-application executable="MyWindowsApplication.exe" command-line=""/>

**Standalone Application executable attribute**

The standalone-application executable attribute specifies the file name of the Windows application to instrument. Set the Standalone Application element executable attribute to one of the following:

- **Executable name:** for example, MyStandaloneApp.exe or MyWindowsService.exe. The file extension is optional, so MyStandaloneApp also works.
- **Full path to the executable:** for example, C:\Program Files\MyApplication\MyStandaloneApp.exe
- **Partial path to the executable:** for example, MyApplication\MyStandaloneApp.exe. Use the full or partial path to the executable when you want to assign different tiers to separate instances of the same executable running from different paths.

Type: String  
Default: None  
Required: Yes

**Standalone Application command-line attribute**

To differentiate between two instances of the same executable, specify any unique portion of the application command line, such as an argument, in the standalone-application command-line attribute.

Type: String  
Default: None  
Required: No

**Standalone application app-domain-name attribute**

For applications with multiple application domains, the app-domain-name attribute enables you to limit instrumentation to specific application domains. See Configure Application Domain Monitoring

Type: String  
Default: None  
Required: No

**Standalone Application Tier Element**

The tier element is a child of the standalone-application element. If you enable instrumentation for an application, you must use a tier element to assign the application to a tier in the Controller. See Overview of Application Monitoring

Required Element: `<tier name="Consumer"/>

**Tier name attribute**

The tier name attribute indicates the business application tier.

Type: String, may also reference an environment variable. See 'Reference of Environment Variables'  
Default: None  
Required: Yes

Environment Variable: APPDYNAMICS_AGENT_TIER_NAME

Sample Windows Service and Standalone Application Configuration
<standalone-applications>
  <standalone-application executable="ExampleApplication.exe">
    <tier name="Standalone Application Tier"/>
  </standalone-application>

<!-- Instrument a Windows service using arguments. -->
<!-- The following example matches the command "MyWindowsService.exe -d -x -r". -->
  <standalone-application executable="MyWindowsService.exe"
    command-line="-x">
    <tier name="Windows Service Tier"/>
  </standalone-application>
</standalone-applications>
.NET Agent Environment Variables

### On this page:
- Agent Account Name
- Agent Application Name
- Agent Account Password
- Agent Node Name
- Agent Reuse Node Name
- Agent Reuse Node Name Prefix
- Agent Tier Name
- Agent Unique Host ID
- Controller Hostname
- Controller Port Number
- Controller SSL Enabled
- Reference of Environment Variables

### Related pages:
- .NET Agent Configuration Properties
- Administer the .NET Agent
- Configure the .NET Agent
- Name .NET Tiers
- Configure the .NET Agent for Windows Services and Standalone Applications

This topic is a reference for the configuration properties for the .NET Agent that can be set using Windows System Environment Variables.

**Agent Account Name**

Specifies the account name for the SaaS or multi-tenant Controller.

- **Environment Variable:** APPDYNAMICS.AGENT.ACCOUNTNAME
- **Type:** String
- **Default:** For single-tenant Controllers, the agent assumes the default of "customer1" if you do not specify an account name.
- **Required:** Only for SaaS or multi-tenant Controllers.

**Agent Application Name**

Specifies the business application you use in the Controller. If the application name does not exist, the Controller will create it when the agent registers. See Overview of Application Monitoring.

- **Environment Variable:** APPDYNAMICS_AGENT.APPLICATION_NAME
- **Type:** String
- **Default:** None
- **Required:** Yes

**Agent Account Password**

Specifies the account access key for the Controller.

- **Environment Variable:** APPDYNAMICS.AGENT.ACCOUNTACCESSKEY
- **Type:** String
- **Default:** None
- **Required:** Yes

**Agent Node Name**

The name of the node.

In general, the node name must be unique within the business application and physical host. If you want to use the same node name for multiple nodes on the same physical machine, create multiple virtual hosts using the Unique Host ID property. See Unique Host ID.

- **Environment Variable:** APPDYNAMICS_AGENT_NODE_NAME
- **Type:** String
- **Default:** None
- **Required:** Yes

**Agent Reuse Node Name**
Set this environment variable to true to reuse node names in AppDynamics. When you set the property to true, you don’t need to supply a node name, but you do need to provide a node name prefix using APPDYNAMICS_AGENT_REUSE_NODE_NAME_PREFIX.

This property is useful for monitoring environments where there are many CLRs with short life spans. When true, AppDynamics reuses the node names of historical CLRs for new CLRs. This avoids a proliferation of differently named nodes in AppDynamics over time, particularly when the nodes are essentially identical processes that run over different times.

AppDynamics generates a node name with App, Tier, and Sequence number. The node names are pooled. For example, the sequence numbers are reused when the nodes are purged (based on the node lifetime).

When the .NET Agent starts up, it logs output to the console until it registers with the Controller and the Controller generates the node name.

The Controller reuses node names based on the node.retention.period property.

**Environment Variable:** APPDYNAMICS_AGENT_REUSE_NODE_NAME  
**Type:** Boolean  
**Default:** False  
**Required:** No

Example: Using the following environmental variable specifications, the Controller generates a node name with the prefix "reportGen". Node names will have suffixes --1, --2, and so on, depending on the number of nodes are running in parallel. The name of a node that is shut down and qualifies as a historical node may be reused by a new node.

- APPDYNAMICS_AGENT_REUSE_NODE_NAME=true
- APPDYNAMICS_AGENT_REUSE_NODE_NAME_PREFIX=reportGen

**Agent Reuse Node Name Prefix**

When you configure the agent to reuse node names, use this property to specify the prefix the Controller uses to generate node names dynamically.

**Environment Variable:** APPDYNAMICS_AGENT_REUSE_NODE_NAME_PREFIX  
**Type:** String  
**Default:** None  
**Required:** When APPDYNAMICS_AGENT_REUSE_NODE_NAME=true

Example: Using the following environmental variable specifications, the agent directs the Controller to generate a node name with the prefix "reportGen". Node names will have suffixes --1, --2, and so on, depending on how many nodes are running in parallel.

- APPDYNAMICS_AGENT_REUSE_NODE_NAME=true
- APPDYNAMICS_AGENT_REUSE_NODE_NAME_PREFIX=reportGen

**Agent Tier Name**

Specifies the name of the tier that this .NET node belongs to. Note that this is not the deployment name on the application server.

**Environment Variable:** APPDYNAMICS_AGENT_TIER_NAME  
**Type:** String  
**Default:** None  
**Required:** Yes

**Agent Unique Host ID**

Logically partitions a single physical host or virtual machine such that it appears to the Controller that the application is running on different machines. Set the value to a string that is unique across the entire managed infrastructure. The string may not contain any spaces. If you have a machine agent associated with the application monitored by the app agent, then this property must be set on the machine agent to the same value. See Standalone Machine Agent Installation Scenarios.

**Environment Variable:** APPDYNAMICS_AGENT_UNIQUE_HOST_ID  
**Type:** String  
**Default:** None  
**Required:** No

**Controller Hostname**

Specifies the hostname or the IP address of the AppDynamics Controller. For an on-premises Controller, use the value for Application Server Host Name you provided when you installed the Controller. If you use the AppDynamics SaaS Controller, see the Welcome email from AppDynamics for the name of your Controller.
Environment Variable: APPDYNAMICS.CONTROLLER.HOSTNAME
Type: String
Default: None
Required: Yes, may also be set in the agent config.xml file or using the agent configuration utility.

Controller Port Number
Specifies the HTTP(S) port of the AppDynamics Controller. If the APPDYNAMICS.CONTROLLER.SSL.ENABLED environment variable is set to true, specify the HTTPS port of the Controller; otherwise, specify the HTTP port.

Environment Variable: APPDYNAMICS.CONTROLLER.PORT
Type: Positive Integer
Default: 8090
For On-premises installations, the defaults are port 8090 for HTTP and port 8181 for HTTPS.
For the SaaS Controller, use port 80 for HTTP or port 443 for HTTPS.
Required: Yes, may also be set in the agent config.xml file or using the agent configuration utility.

Controller SSL Enabled
When set to "true" enables encryption over SSL between the agent and the Controller.

Environment Variable: APPDYNAMICS.CONTROLLER.SSL.ENABLED
Type: Boolean
Default: false
Required: No, may also be set in the agent config.xml file or using the agent configuration utility.

Reference of Environment Variables
The .NET Agent (via config.xml) can reference any environment variables when configuring Agent Node Name, Agent Tier Name, or Controller Application Name.

These environment variables can be referenced in the configuration file or in the agent environment variables for the node, tier, and application names listed above. To include an environment variable, reference it by specifying %<variable>% which can be combined with the other characters; for example, Web-%COMPUTERNAME% will be translated to Web-HOST23.
Configure Multiple Business Application Support for .NET

By default, applications on a single Windows host are mapped to a single business application in the Controller. If needed, you can manually configure the .NET Agent to map different applications on the same Windows host to different business applications in the Controller.

To map applications on a single host to a different business application, you need to edit the config.xml file manually. The AppDynamics Agent Configuration utility does not offer this configuration option.

After configuring multiple application support as described here, you cannot use the configuration utility to make configuration changes afterward. If you launch the configuration utility on a server where you've configured multiple application support, the utility will prompt you to delete the configurations.

Prepare to Configure Multiple Business Applications

Before you configure the .NET Agent, you must install the agent. Use the AppDynamics Agent Configuration utility to perform basic configuration tasks.

1. Identify how you want to organize your business applications and identify the business application names.

   For example, consider a Windows host running IIS. The IIS instance serves two applications for two separate customers: Ticket Search and Travel Search. The applications perform similar functions, but it makes sense to monitor them separately because they function independently. In this case, create two business applications based upon the application name: Ticket Search Engine and Travel Search Engine.

2. Map your IIS applications or application pools, Windows services, and standalone applications to tiers in the different business applications.

   For example, map the TicketSearch site to the Ticket Search Web tier in the Ticket Search Engine business application. Map the TravelSearch site to the Travel Search Web tier in the Travel Search Engine business application.

3. If you have not already done so, install the .NET Agent. See Install the .NET Agent for Windows

4. Run the AppDynamics Agent Configuration utility to generate a config.xml and configure the Controller connection. See Configure the .NET Agent

Manually Configure the .NET Agent

Once you have configured the Controller properties for the .NET Agent, instrument your .NET Applications in the config.xml.

1. Open the config.xml file as administrator and edit the file. See ‘Where to Configure Agent Properties’ on Administer the .NET Agent.
2. Copy the controller applications block and paste it as a child element of the controller element. Replace any existing <applications> or <application> elements:

   ```xml
   <!--Configure multiple business applications-->
   <applications>
     <application name="ApplicationName1" default ="true"/>
     <application name="ApplicationName2"/>
   </applications>
   ```

3. Add an application element for each of the business applications in the Controller.
   - Edit the name attribute for the application elements to match the business application names in the Controller. If the application does not exist yet, the Controller creates it.
   - Set the default attribute to true for one application element. If the agent cannot find a match for the business application name in the IIS application, Windows service or standalone application configuration, the tier reports to the default business application.

   In the following example, Ticket Search is the default business application:

   ```xml
   <appdynamics-agent
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:xsd="http://www.w3.org/2001/XMLSchema">
     <controller host="mycontroller.mycompany.com" port="8090"
                 ssl="false">
       <account name="customer1" password="changeme" />
       <!--Configure multiple business applications-->
       <applications>
         <application name="Ticket Search" default ="true"/>
         <application name="Travel Search"/>
       </applications>
     </controller>
   </appdynamics-agent>
   ...
   ```

4. Add configuration elements for the IIS applications or application pools, Windows services, or standalone applications to instrument.

   For IIS applications, add the applications block as a child of the IIS element. Replace the existing applications element. For more information, see IIS Applications Element.

   ```xml
   <!--Show IIS Application configuration.
   ```
For IIS application pools, add the `application pools` block as a child of the `IIS` element. Replace the existing `<application-pools>` element. See [IIS Application Pools Element](#).

```
<application-pools>
  <application-pool name="MyAppPool1"
    controller-application="Application1">
    <tier name="App1 AppPool Tier"/>
  </application-pool>
  <application-pool name="MyAppPool2"
    controller-application="Application2">
    <tier name="App2 AppPool Tier"/>
  </application-pool>
</application-pools>
```

For Windows services or standalone applications, add the `standalone-applications` block as a child of the `app-agents` element. See [Standalone Applications Element](#).

```
<standalone-applications>
  <standalone-application executable="MyStandaloneApp.exe"
    controller-application="ApplicationName1">
    <tier name="Standalone App Tier"/>
  </standalone-application>
  <standalone-application executable="MyWindowsService.exe"
    command-line="-x" controller-application="ApplicationName2">
    <tier name="Windows Service Tier"/>
  </standalone-application>
</standalone-applications>
```

5. Configure your application elements as follows:
   - Add the corresponding element for each IIS application or application pool, Windows service or standalone application to instrument. For information on specific elements, see [.NET Agent Configuration Properties](#).
   - For each application element set the `controller-application` attribute to the name of the corresponding business application. If you omit the `controller-application` attribute, the agent adds the application to a tier in the configured default business application.
   - Set the `tier` element `name` attribute to the business application tier name.

6. After you complete configuration, save the changes to the `config.xml` file.
8. Restart IIS applications or application pools, Windows services, and standalone applications.
9. As your applications begin processing traffic, the agent registers them with the Controller. Log on to the Controller to see that your applications have registered with the corresponding business application.

**Sample Configuration**

This sample config.xml file demonstrates configuration for multiple business applications in the Controller. Because the Windows service `TicketService` does not specify a `controller-application` attribute, it reports to the default business application, `Ticket Search Engine`. All applications in the `TravelAPIPool` pool report to the `Travel Search Engine`.

Show sample config.xml
<?xml version="1.0" encoding="utf-8"?>
<appdynamics-agent
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<controller host="mycontroller.mycompany.com" port="8090"
ssl="false">
<!--Configure multiple business applications-->  
    <applications>
        <application name="Ticket Search Engine" default="true"/>
        <application name="Travel Search Engine"/>
    </applications>
</controller>
</app-agents>

<!--IIS-->
    <IIS>
        <automatic enabled="false" />
        <application-pools>
            <application-pool name="TravelAPIPool" controller-application="Travel Search Engine">
                <tier name="Travel APIs"/>
            </application-pool>
        </application-pools>
        <applications>
            <application path="/" site="TicketSearch" controller-application="Travel Search Engine">
                <tier name="Ticket Search Web"/>
            </application>
            <application path="/" site="TravelSearch" controller-application="Travel Search Engine">
                <tier name="Travel Search Web"/>
            </application>
        </applications>
    </IIS>

<!--Standalone Applications-->
    <standalone-applications>
        <standalone-application executable="StandaloneApp.exe" controller-application="Ticket Search Engine">
            <tier name="Ticket Standalone Tier"/>
        </standalone-application>
        <standalone-application executable="WindowsService.exe" command-line="--x" controller-application="Travel Search Engine">
            <tier name="Travel Windows Service Tier"/>
        </standalone-application>
    </standalone-applications>

<!--Windows Services-->
    <windows-services>
        <windows-service name="TicketService">
            <tier name="TicketService Tier"/>
        </windows-service>
    </windows-services>
</app-agents>
</appdynamics-agent>
Agent Configuration Properties for Multiple Application Support

Multiple business application support includes configuration properties for the .NET Agent. These configuration properties supersede the ones documented in .NET Agent Configuration Properties.

Controller Applications Element

The <applications> element is a child of the <controller> element. It is a container element for all controller applications elements that map to business applications in the Controller.

Required Element: <applications>

Controller Application Element

The <application> element is a child of the <controller applications> element. It indicates the name of the logical business application you see in the Controller. When you have more than one Controller Application element, you must set the default attribute to true for one of them.

Required Element: <application name="MyDotNetApplication" default="true"/>

Application name attribute

Set the application name attribute to the business application name in the Controller. If the application name does not exist, the Controller creates it when the agent registers.

Type: String
Default: None
Required: Yes

Application default attribute

Set the application default attribute to true for one controller application element. Instrumented applications without the controller application attribute register with the default business application in the Controller.

Type: Boolean
Default: false
Required: For one application in multiple application configurations

Controller-Application Attribute

The IIS application, IIS application-pool, windows-service, and standalone-application elements accept the controller-application attribute. Set the value to the controller application element name. If you do not include a controller-application attribute, the application registers with the default business application.

For example, an IIS application:

<application path="/" site="MySite" controller-application="My Business Application"/>

Type: String
Default: None
Required: No
Disable Instrumentation for an IIS Application Pool

Related pages:
- Administer the .NET Agent
- .NET Agent Configuration Properties

When you install the .NET Agent on a machine and use automatic tier naming, the agent instruments every IIS application by default. If you do not need to monitor all application pools, disable monitoring for selected pools.

1. Open the config.xml file for editing as administrator. See 'Where to Configure Agent Properties' on Administer the .NET Agent
2. Add the application-pools block as a child of the IIS element. See .NET Agent Configuration Properties

   ```xml
   <application-pools>
     <!-- Do not instrument applications in DefaultAppPool when "enabled" attribute is set to false -->
     <application-pool name="DefaultAppPool" enabled="false">
       <tier name="Disabled App Pool" />
     </application-pool>
   </application-pools>
   ```

   tier is a required element even if you are disabling an app pool.

3. Set the application-pool element name attribute to the application pool name. This example disables instrumentation for the DefaultAppPool. You may add multiple application-pool elements.

5. Restart IIS.
Configure Application Domain Monitoring

On this page:
- Overview of AppDomains in .NET
- Configure Monitoring for Multiple Application Domains

Related pages:
- Instrument DefaultDomain for Standalone Applications
- Application Domains
- .NET Agent Configuration Properties
- Configure the .NET Agent for Windows Services and Standalone Applications

You can configure the .NET Agent to monitor ASP.NET applications with multiple Application Domains (AppDomains). This topic assumes you have a working knowledge of AppDomains and that you are familiar with the AppDomain implementation in your application.

This topic does not cover the System Domain, Shared Domain, or DefaultDomain AppDomains the CLR instantiates before it executes the managed code. If your standalone application runs in the DefaultDomain, see Instrument the DefaultDomain for Standalone Applications.

Overview of AppDomains in .NET

Windows uses processes to manage security and performance isolation between running applications. Process isolation ensures one application's running code doesn't interfere with another application. However, applications that share data, making calls between Windows processes can introduce complications and performance issues. AppDomains enable developers to create several applications that run inside a single process but maintain application isolation.

Single Application Domain

In the case of a single application running inside its own process, the runtime host typically manages the AppDomain. The application executable and the AppDomain have the same name. The .NET Agent installs itself inside the single AppDomain and creates a node for the application.

Multiple Application Domains

When developers include multiple AppDomains in an application, all the AppDomains run inside a single process. The application executable may have the same name as one AppDomain, but there are other, uniquely-named AppDomains. By default, the agent installs itself inside all the AppDomains and creates nodes for them.
Configure Monitoring for Multiple Application Domains

If the application you monitor contains multiple AppDomains, the App Agent for .NET automatically instruments each AppDomain and creates a node. You can configure the .NET Agent to instrument only the AppDomains you specify. This is useful to exclude AppDomains you do not want to monitor and to limit the number of nodes in a tier.

You can configure application domain monitoring for:

- Windows Services
- Standalone Applications

Configure all instrumentation settings for the .NET Agent in the config.xml file. See ‘Where to Configure Agent Properties’ on Administer the .NET Agent

1. Identify the name of the AppDomains you want to instrument.

   If you have already instrumented your application, you can see the AppDomain names in the Node Dashboard.

   Click the node in the left navigation pane, then click CLR.
2. Launch a text editor as administrator.
3. Edit the config.xml file as an administrator. See Administer the .NET Agent.
4. Find the element that corresponds to your application with multiple AppDomains:
   Standalone Application element: `<standalone-application executable="MyWindowsApplication.exe">`
5. Add the app-domain-name attribute to the element.

For example, to instrument the MyApp.exe AppDomain for the MyApp.exe standalone application:

```xml
<standalone-application executable="MyApp.exe"
  app-domain-name="MyApp.exe">
  <tier name="StandaloneApplication Tier"/>
</standalone-application>
```

As soon as you instrument one AppDomain in the config.xml file, the agent instruments only the AppDomains you specify. Other AppDomains are not instrumented.

6. To instrument additional AppDomains, add an element for each AppDomain as if they were separate applications. For example, to instrument MyAppDomain1 in MyApp.exe:

```xml
<standalone-application executable="MyApp.exe"
  app-domain-name="MyAppDomain1">
  <tier name="StandaloneApplication Tier"/>
</standalone-application>
```

7. Save the config.xml file.
9. Restart instrumented applications: Windows services or standalone applications.

Sample Standalone Application configuration with multiple AppDomains

This sample config.xml file shows the configuration for the application, MyApp.exe. Instrumentation only applies to the AppDomains specified in the config.xml file: MyApp.exe and MyAppDomain2.

```xml
<?xml version="1.0" encoding="utf-8"?>
<appdynamics-agent
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <controller host="mycontroller.mycompany.com" port="8090"
 ssl=false">
 <account name="customer1" password="changeme" />
 <application name="MyDotNetApplication" />
 </controller>
 <machine-agent />
 <app-agents>
 <standalone-applications>
 <standalone-application executable="MyApp.exe"
 app-domain-name="MyApp.exe">
 <tier name="StandaloneApplication Tier"/>
 </standalone-application>
 <standalone-application executable="MyApp.exe"
 app-domain-name="MyAppDomain2">
 <tier name="StandaloneApplication Tier"/>
 </standalone-application>
 </standalone-applications>
 </app-agents>
</appdynamics-agent>
```
Instrument the DefaultDomain for Standalone Applications

On this page:

- Check if Your Application Runs in the DefaultDomain
- Instrument the DefaultDomain

Related pages:

- Configure the .NET Agent for Windows Services and Standalone Applications
- POCO Entry Points
- .NET Agent Configuration Properties

By default, the .NET Agent does not instrument the .NET DefaultDomain AppDomain. Before you instrument the DefaultDomain:

- Follow the instructions to instrument a standalone application
- Create a POCO entry point for a class/method in the application

If you complete those steps and still do not see business transactions in the Controller, check if your managed code runs in the DefaultDomain. If so, you must configure the agent to instrument the DefaultDomain.

**Check if Your Application Runs in the DefaultDomain**

If you are unfamiliar with your application's managed code, you can use the agent logs to identify the AppDomain.

1. Open the agent log:
   - Windows Server 2008 and later: %ProgramData%\AppDynamics\DotNetAgent\Logs\AgentLog.txt
2. Search the agent log for AppDomain.

   Few log entries contain AppDomain when the agent starts up. Look for an entry by dllhost or your instrumented application similar to the following:

   2013-12-16 08:23:02.3120 3068 MYPROGRAM 1 1 Info Configuration
   appDomainName=DefaultDomain appDomainId=1 iis-app=null
   site=null port=null appPoolId=
   2013-12-16 08:23:02.6240 3192 dllhost 1 17 Info
   ConfigurationManager Not instrumenting DefaultDomain for pid 3068

   In this example, MYPROGRAM is the name of the instrumented standalone application. You can see the name of the AppDomain in the log entry, appDomainName=DefaultDomain.

**Instrument the DefaultDomain**

1. Open the config.xml file for editing as administrator. See ‘Where to Configure Agent Properties’ on Administer the .NET Agent
2. Copy the code block below to a child element of the standalone-application element. See .NET Agent Configuration Properties

   ```xml
   <profiler>
       <instrument-defaultdomain enabled="true"/>
   </profiler>
   ```

   The Profiler element must follow the Standalone Application Tier element.

   For example:
<xml version="1.0" encoding="utf-8"?
<appdynamics-agent
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xxmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <controller host="mycontroller.mycompany.com" port="8090"
ssl="false">
    <application name="My Business Application" />
    <account name="customer1" password="changeme" />
    </controller>
    <machine-agent />
    <app-agents>
        <IIS>
            <applications />
        </IIS>
        <standalone-applications>
            <standalone-application executable="MyStandaloneApp.exe">
                <tier name="Standalone Tier" />
                <profiler>
                    <instrument-defaultdomain enabled="true"/>
                </profiler>
            </standalone-application>
        </standalone-applications>
    </app-agents>
</appdynamics-agent>

3. Save the config.xml file.
5. Restart the standalone application for your changes to take effect.
Configure the .NET Machine Agent

On this page:
- Customize .NET Machine Agent Behavior
- .NET Machine Agent Configuration Options
- Configure the .NET Machine Agent Without App Agents

Related pages:
- Administer the .NET Agent
- .NET Agent Configuration Properties
- Monitor Windows Hardware Resources

Do not confuse the .NET Machine Agent with the Standalone Machine Agent, a Java application. The Standalone Machine Agent provides the capability to use extensions such as plugins, metric listeners, and orchestrations. See Standalone Machine Agent.

The AppDynamics .NET Agent includes an embedded .NET Machine Agent that runs as part of the AppDynamics.Agent.Coordinator service. Among other things, the Machine Agent regularly gathers system performance data and reports it back to the Controller as metrics.

To manage the .NET Machine Agent from the Controller, see 'Manage the .NET Machine Agent' on Monitor Windows Hardware Resources.

Customize .NET Machine Agent Behavior

Customize instrumentation settings for the Machine Agent element in the config.xml file. See 'Where to Configure Agent Properties' on Administer the .NET Agent.

2. Edit the config.xml file as an administrator. See 'Where to Configure Agent Properties' on Administer the .NET Agent.
3. Modify the machine-agent element and add any children elements according to the .NET Machine Agent configuration topics.
4. Save the config.xml file.
5. Start the AppDynamics.Agent.Coordinator service.
6. In some cases, you may need to restart IIS, instrumented Windows services or instrumented Standalone applications. See individual .NET Machine Agent configuration topics.

.NET Machine Agent Configuration Options

The following topics cover specific customizations for the .NET Machine Agent:

- Manage Windows Performance Metrics
- Thread Correlation for .NET
- Enable Correlation for .NET Remoting
- Enable Instrumentation for WCF Data Services
- Configure Machine Snapshots for .NET

Configure the .NET Machine Agent Without App Agents

If you want to monitor the Windows hardware performance data, but do not want to monitor any .NET applications, you can configure the .NET Machine agent to run without the .NET Agent.

1. Install the .NET Agent for Windows.
2. Launch the AppDynamics Agent Configuration utility and follow the steps until you reach the Assign IIS applications to tiers window.
3. Click Manual for the method of tier generation and assignment and click Next.
4. Do not assign any IIS applications to tiers, click **Next**.

![Assign IIS applications to tiers](image)

**Do not assign IIS apps to tiers**

*If you configured any Windows services or standalone applications, manually disable those agents in the `config.xml` file.*

5. Continue with the remaining steps and click **Done**.

Monitoring resumes for the .NET Machine Agent only. Metrics appear in the Controller under the Machine Agent tier, see **Monit or Windows Hardware Resources**.
Manage Windows Performance Metrics

On this page:

- Tune the Default Performance Metrics for the .NET Agent
- Configure Additional Performance Counters for .NET
- Sample .NET Machine Agent Configuration with Performance Counters

The AppDynamics .NET Machine Agent uses Microsoft Performance Counters to gather and report .NET metrics. You can manage the metrics that the agent collects as follows:

- Tune the set of metrics that the agent automatically collects
- Configure the agent to collect metrics from additional counters that are not part of the default settings

**Tune the Default Performance Metrics for the .NET Agent**

The .NET Agent prioritizes the default set of Performance Counters into three categories: low, medium, and high. By default, the agent is set to collect metrics for all three categories.

Click here to view the default metric priorities...

<table>
<thead>
<tr>
<th>Metric Browser Location</th>
<th>Metric</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASP.NET</td>
<td>Application Restarts</td>
<td>3 - Low</td>
</tr>
<tr>
<td>ASP.NET</td>
<td>Applications Running</td>
<td>3 - Low</td>
</tr>
<tr>
<td>ASP.NET</td>
<td>Request Wait Time</td>
<td>2 - Medium</td>
</tr>
<tr>
<td>ASP.NET</td>
<td>Requests Disconnected</td>
<td>3 - Low</td>
</tr>
<tr>
<td>ASP.NET</td>
<td>Requests Queued</td>
<td>1 - High</td>
</tr>
<tr>
<td>ASP.NET</td>
<td>Requests Rejected</td>
<td>3 - Low</td>
</tr>
<tr>
<td>ASP.NET</td>
<td>Worker Process Restarts</td>
<td>3 - Low</td>
</tr>
<tr>
<td>ASP.NET Applications</td>
<td>Anonymous Requests</td>
<td>3 - Low</td>
</tr>
<tr>
<td>ASP.NET Applications</td>
<td>Anonymous Requests/Sec</td>
<td>3 - Low</td>
</tr>
<tr>
<td>ASP.NET Applications</td>
<td>Cache API Entries</td>
<td>3 - Low</td>
</tr>
<tr>
<td>ASP.NET Applications</td>
<td>Cache API Hit Ratio</td>
<td>3 - Low</td>
</tr>
<tr>
<td>ASP.NET Applications</td>
<td>Cache API Turnover Rate</td>
<td>3 - Low</td>
</tr>
<tr>
<td>ASP.NET Applications</td>
<td>Cache Total Entries</td>
<td>3 - Low</td>
</tr>
<tr>
<td>ASP.NET Applications</td>
<td>Cache Total Hit Ratio</td>
<td>3 - Low</td>
</tr>
<tr>
<td>ASP.NET Applications</td>
<td>Cache Total Turnover Rate</td>
<td>3 - Low</td>
</tr>
<tr>
<td>ASP.NET Applications</td>
<td>Errors During Compilation</td>
<td>3 - Low</td>
</tr>
<tr>
<td>ASP.NET Applications</td>
<td>Errors During Execution</td>
<td>3 - Low</td>
</tr>
<tr>
<td>ASP.NET Applications</td>
<td>Errors During Preprocessing</td>
<td>3 - Low</td>
</tr>
<tr>
<td>ASP.NET Applications</td>
<td>Errors Total</td>
<td>2 - Medium</td>
</tr>
<tr>
<td>ASP.NET Applications</td>
<td>Errors Total/sec</td>
<td>2 - Medium</td>
</tr>
<tr>
<td>ASP.NET Applications</td>
<td>Errors Unhandled During Execution</td>
<td>3 - Low</td>
</tr>
<tr>
<td>ASP.NET Applications</td>
<td>Errors Unhandled During Execution/Sec</td>
<td>3 - Low</td>
</tr>
<tr>
<td>Category</td>
<td>Metric</td>
<td>Grade</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>---------------------------------------------</td>
<td>-------</td>
</tr>
<tr>
<td>ASP.NET Applications</td>
<td>Output Cache Entries</td>
<td>3</td>
</tr>
<tr>
<td>ASP.NET Applications</td>
<td>Output Cache Hit Ratio</td>
<td>3</td>
</tr>
<tr>
<td>ASP.NET Applications</td>
<td>Output Cache Turnover Rate</td>
<td>3</td>
</tr>
<tr>
<td>ASP.NET Applications</td>
<td>Pipeline Instance Count</td>
<td>3</td>
</tr>
<tr>
<td>ASP.NET Applications</td>
<td>Requests Executing</td>
<td>3</td>
</tr>
<tr>
<td>ASP.NET Applications</td>
<td>Requests Failed</td>
<td>3</td>
</tr>
<tr>
<td>ASP.NET Applications</td>
<td>Requests in Application Queue</td>
<td>1</td>
</tr>
<tr>
<td>ASP.NET Applications</td>
<td>Requests Not Authorized</td>
<td>3</td>
</tr>
<tr>
<td>ASP.NET Applications</td>
<td>Requests Not Found</td>
<td>3</td>
</tr>
<tr>
<td>ASP.NET Applications</td>
<td>Requests Succeeded</td>
<td>3</td>
</tr>
<tr>
<td>ASP.NET Applications</td>
<td>Requests Timed Out</td>
<td>2</td>
</tr>
<tr>
<td>ASP.NET Applications</td>
<td>Requests Total</td>
<td>2</td>
</tr>
<tr>
<td>ASP.NET Applications</td>
<td>Requests/sec</td>
<td>2</td>
</tr>
<tr>
<td>ASP.NET Applications</td>
<td>Session SQL Server Connections Total</td>
<td>3</td>
</tr>
<tr>
<td>ASP.NET Applications</td>
<td>Session State Server Connections Total</td>
<td>3</td>
</tr>
<tr>
<td>ASP.NET Applications</td>
<td>Sessions Abandoned</td>
<td>3</td>
</tr>
<tr>
<td>ASP.NET Applications</td>
<td>Sessions Active</td>
<td>3</td>
</tr>
<tr>
<td>ASP.NET Applications</td>
<td>Sessions Timed Out</td>
<td>3</td>
</tr>
<tr>
<td>ASP.NET Applications</td>
<td>Sessions Total</td>
<td>3</td>
</tr>
<tr>
<td>ASP.NET Applications</td>
<td>Transactions Aborted</td>
<td>3</td>
</tr>
<tr>
<td>ASP.NET Applications</td>
<td>Transactions Committed</td>
<td>3</td>
</tr>
<tr>
<td>ASP.NET Applications</td>
<td>Transactions Pending</td>
<td>3</td>
</tr>
<tr>
<td>ASP.NET Applications</td>
<td>Transactions Total</td>
<td>3</td>
</tr>
<tr>
<td>ASP.NET Applications</td>
<td>Transactions/sec</td>
<td>3</td>
</tr>
<tr>
<td>CLR</td>
<td>Process CPU Burnt %</td>
<td>2</td>
</tr>
<tr>
<td>CLR -&gt; Classes</td>
<td>Current Loaded Class Count</td>
<td>3</td>
</tr>
<tr>
<td>CLR -&gt; Classes</td>
<td>Total Classes Loaded</td>
<td>3</td>
</tr>
<tr>
<td>CLR -&gt; Locks and Threads</td>
<td>Contention Rate Per Sec</td>
<td>2</td>
</tr>
<tr>
<td>CLR -&gt; Locks and Threads</td>
<td>Current Logical Threads</td>
<td>2</td>
</tr>
<tr>
<td>CLR -&gt; Locks and Threads</td>
<td>Current Physical Threads</td>
<td>2</td>
</tr>
<tr>
<td>CLR -&gt; Locks and Threads</td>
<td>Sink Blocks in Use</td>
<td>2</td>
</tr>
<tr>
<td>CLR -&gt; Memory -&gt; Heap</td>
<td>Committed (bytes)</td>
<td>1</td>
</tr>
<tr>
<td>CLR -&gt; Memory -&gt; Heap</td>
<td>Current Usage (bytes)</td>
<td>1</td>
</tr>
<tr>
<td>CLR -&gt; Memory -&gt; Heap</td>
<td>Gen 0 Usage (bytes)</td>
<td>1</td>
</tr>
<tr>
<td>CLR -&gt; Memory -&gt; Heap</td>
<td>Gen 1 Usage (bytes)</td>
<td>1</td>
</tr>
<tr>
<td>Resource Type</td>
<td>Metric Name</td>
<td>Value</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
<td>-------</td>
</tr>
<tr>
<td>CLR -&gt; Memory -&gt; Heap</td>
<td>Gen 2 Usage (bytes)</td>
<td>1 - High</td>
</tr>
<tr>
<td>CLR -&gt; Memory -&gt; LOH</td>
<td>Current Usage (bytes)</td>
<td>1 - High</td>
</tr>
<tr>
<td>Hardware Resources -&gt; CPU</td>
<td>%Busy</td>
<td>1 - High</td>
</tr>
<tr>
<td>Hardware Resources -&gt; CPU</td>
<td>%Idle</td>
<td>3 - Low</td>
</tr>
<tr>
<td>Hardware Resources -&gt; Disks</td>
<td>%Free</td>
<td>1 - High</td>
</tr>
<tr>
<td>Hardware Resources -&gt; Disks</td>
<td>KB read/sec</td>
<td>3 - Low</td>
</tr>
<tr>
<td>Hardware Resources -&gt; Disks</td>
<td>KB written/sec</td>
<td>3 - Low</td>
</tr>
<tr>
<td>Hardware Resources -&gt; Disks</td>
<td>MB Free</td>
<td>1 - High</td>
</tr>
<tr>
<td>Hardware Resources -&gt; Disks</td>
<td>Reads/sec</td>
<td>1 - High</td>
</tr>
<tr>
<td>Hardware Resources -&gt; Disks</td>
<td>Writes/sec</td>
<td>1 - High</td>
</tr>
<tr>
<td>Hardware Resources -&gt; Memory</td>
<td>Free %</td>
<td>3 - Low</td>
</tr>
<tr>
<td>Hardware Resources -&gt; Memory</td>
<td>Free (MB)</td>
<td>3 - Low</td>
</tr>
<tr>
<td>Hardware Resources -&gt; Memory</td>
<td>Total (MB)</td>
<td>1 - High</td>
</tr>
<tr>
<td>Hardware Resources -&gt; Memory</td>
<td>Used %</td>
<td>1 - High</td>
</tr>
<tr>
<td>Hardware Resources -&gt; Memory</td>
<td>Used (MB)</td>
<td>2 - Medium</td>
</tr>
<tr>
<td>Hardware Resources -&gt; Network</td>
<td>Incoming KB/sec</td>
<td>2 - Medium</td>
</tr>
<tr>
<td>Hardware Resources -&gt; Network</td>
<td>Incoming packets/sec</td>
<td>3 - Low</td>
</tr>
<tr>
<td>Hardware Resources -&gt; Network</td>
<td>Outgoing KB/sec</td>
<td>2 - Medium</td>
</tr>
<tr>
<td>Hardware Resources -&gt; Network</td>
<td>Outgoing packets/sec</td>
<td>3 - Low</td>
</tr>
<tr>
<td>Hardware Resources -&gt; System</td>
<td>RQ</td>
<td>3 - Low</td>
</tr>
<tr>
<td>IIS</td>
<td>CPU %</td>
<td>1 - High</td>
</tr>
<tr>
<td>IIS</td>
<td>Number of Working Processes</td>
<td>3 - Low</td>
</tr>
<tr>
<td>IIS</td>
<td>Working Set</td>
<td>1 - High</td>
</tr>
<tr>
<td>IIS -&gt; Application Pools -&gt; &lt;name&gt;</td>
<td>CPU %</td>
<td>1 - High</td>
</tr>
<tr>
<td>IIS -&gt; Application Pools -&gt; &lt;name&gt;</td>
<td>Number of Working Processes</td>
<td>3 - Low</td>
</tr>
<tr>
<td>IIS -&gt; Application Pools -&gt; &lt;name&gt;</td>
<td>Working Set</td>
<td>1 - High</td>
</tr>
<tr>
<td>IIS -&gt; INFO</td>
<td>VersionMajor</td>
<td>3 - Low</td>
</tr>
<tr>
<td>IIS -&gt; INFO</td>
<td>VersionMinor</td>
<td>3 - Low</td>
</tr>
</tbody>
</table>

If you want to reduce the number of metrics that the agent sends to the Controller, or if your are not interested in the full set of metrics, you can configure that agent to collect fewer performance metrics. You can also set the agent to only collect specific metrics from the list. For specific instructions about how to configure the agent, see 'Performance Metrics Element' on .NET Agent Configuration Properties.

**Configure Additional Performance Counters for .NET**

If the agent does not monitor performance counters you are interested in, you can add additional performance counters.

2. Open the config.xml file for editing as administrator. See 'Where to Configure Agent Properties' on Administer the .NET Agent.
3. Add the Performance Counters block as a child of the Machine Agent element.

```xml
<perf-counters>
    <perf-counter cat="" name="" instance=""/>
</perf-counters>
```

4. Create a Performance Counter element for each performance counter you want to add. Use any of the performance counters as specified in Performance Counters in .NET Framework.
   - Set the `cat` attribute to the category of the performance counter.
   - Set the `name` attribute to the performance counter name.
   - Set the `instance` attribute to the instance of the performance counter.
     If the counter does not have an instance name, leave an empty string "".
     If a particular performance counter has many instances you can specify the following options:
     - `instance ="*" OR instance ="all"` (This reports the sum of all instances)

   For example, to add the performance counter for measuring CPU Idle time(%), add the following element in the `<perf-counters>` block:

```xml
<perf-counter cat="Processor" name="% Idle Time" instance="_Total"/>
```

5. Save the `config.xml` file.


After you enable Performance Counter metrics, they appear under the Custom Metrics tree in the Metric Browser.

**Sample .NET Machine Agent Configuration with Performance Counters**

```xml
<machine-agent>
    <!-- Additional machine level Performance Counters -->
    <perf-counters>
        <perf-counter cat="Processor" name="% Idle Time" instance="_Total"/>
        <!-- Perf counter with no instance name -->
        <perf-counter cat="Memory" name="Available MBytes" instance=""/>
    </perf-counters>
</machine-agent>
```
Enable Correlation for .NET Remoting

On this page:
- Instrument Applications That Use .NET Remoting
- Specify an Agent Trigger

Related pages:
- .NET Backend Detection
- Remote Services

Developers use .NET remoting to build distributed applications that share objects across processes or across application domains running in the same process. AppDynamics disables correlation for .NET remoting functions by default.

**Instrument Applications That Use .NET Remoting**

You can configure the .NET Agent to discover .NET remoting entry and exit points.

1. Open the **config.xml** file for editing as administrator. See 'Where to Configure Agent Properties' on Administer the .NET Agent
2. Copy the code block below to a child element of the Machine Agent element. See .NET Agent Configuration Properties

```xml
<instrumentation>
  <instrumentor name="RemotingMscorlibEntryInstrumentor" enabled="true"/>
  <instrumentor name="RemotingExitInstrumentor" enabled="true"/>
</instrumentation>
```

For example:

```xml
<?xml version="1.0" encoding="utf-8"?>
<appdynamics-agent xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  ...
  <machine-agent>
    <!--Enable correlation for .NET remoting-->
    <instrumentation>
      <instrumentor name="RemotingMscorlibEntryInstrumentor" enabled="true"/>
      <instrumentor name="RemotingExitInstrumentor" enabled="true"/>
    </instrumentation>
  </machine-agent>
  ...
</appdynamics-agent>
```

3. Save the **config.xml** file.
5. Restart instrumented applications for your changes to take effect.

If the agent does not discover the entry points after configuration, specify an agent trigger.
Specify an Agent Trigger

.NET remoting entry point functions execute in low-level .NET libraries that may not trigger automatic agent instrumentation. If the agent does not discover the .NET remoting entry points after configuration you can specify a function that triggers the agent to begin instrumentation.

1. Identify a function to trigger the agent to begin instrumentation. The function can be any function that executes as part of the application process.

   For example, consider the following code for a MovieTicket remoting object. In this case, use the function GetTicketStatus to trigger the agent.

   ```csharp
   using System;
   namespace MovieGoer
   {
       public class MovieTicket : MarshalByRefObject
       {
           public MovieTicket()
           {
           }
           public string GetTicketStatus(string stringToPrint)
           {
               return String.Format("Enquiry for {0} -- Sending back status: {1}", stringToPrint, "Ticket Confirmed");
           }
       }
   }
   ```

2. Edit the config.xml file as an administrator. See Administer the .NET Agent

3. Update the Instrumentation element to include the AgentTriggerInstrumentor attribute. See .NET Agent Configuration Properties

   ```xml
   <instrumentation>
     <instrumentor name="AgentTriggerInstrumentor" enabled="true" args="" />
     <instrumentor name="RemotingMscorlibEntryInstrumentor" enabled="true" />
     <instrumentor name="RemotingExitInstrumentor" enabled="true" />
   </instrumentation>
   ```

4. Set the AgentTriggerInstrumentor args value to the name of the trigger function, using information in the code block in step 1. The args value should be formatted as follows: namespace.public_class.public_string.

   For example:
5. Save the config.xml file.
6. Restart instrumented applications for your changes to take effect.
By default, the AppDynamics .NET Agent enables multi-threaded correlation for the following patterns on the Common Language Runtime (CLR) 4.x and later:

- Task.Start
- Task.Run
- TaskFactory.StartNew

The .NET Agent also supports thread correlation for the following patterns which are disabled by default:

- Thread.Start on the CLR 2.x and CLR 4.x
- ThreadPool.QueueUserWorkItem on the CLR 2.x and CLR 4.x

In versions of AppDynamics prior to 4.3, for the .NET Agent, you did not need to configure correlation for ThreadPool.QueueUserWorkItem on the CLR 4 in the config.xml file. If you explicitly call ThreadPool.QueueUserWorkItem for the CLR 4, when you upgrade to the .NET Agent 4.5 from a previous version, you must enable the instrumentor for ThreadPool.QueueUserWorkItem on the CLR 4 in the config.xml file.

**Configure Thread Correlation for .NET**

Configure all instrumentation settings for the .NET Agent in the config.xml file. See [Administer the .NET Agent](#).

1. Open the config.xml file for editing as an administrator. See 'Where to Configure Agent Properties' on [Administer the .NET Agent](#).
2. Copy the code block below to a child element of the Machine Agent element. See [Machine Agent Element](#).

```xml
<instrumentation>
  <instrumentor name="ThreadCorrelationThreadPoolCLR2Instrumentor" enabled="true"/>
  <instrumentor name="ThreadCorrelationThreadPoolCLR4Instrumentor" enabled="true"/>
  <instrumentor name="ThreadStartCLR2Instrumentor" enabled="true"/>
  <instrumentor name="ThreadStartCLR4Instrumentor" enabled="true"/>
</instrumentation>
```

For example:
<appdynamics-agent
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 ...
 <machine-agent>
  <!--Enable thread correlation-->
  <instrumentation>
   <instrumentor
    name="ThreadCorrelationThreadPoolCLR2Instrumentor"
    enabled="true"/>
   <instrumentor
    name="ThreadCorrelationThreadPoolCLR4Instrumentor"
    enabled="true"/>
   <instrumentor
    name="ThreadStartCLR2Instrumentor"
    enabled="true"/>
   <instrumentor
    name="ThreadStartCLR4Instrumentor"
    enabled="true"/>
   </instrumentation>
  </machine-agent>
 ...
</appdynamics-agent>

The configuration syntax is enabled="true".
3. Save the config.xml file.
5. Restart instrumented applications for your changes to take effect.
Enable Instrumentation for WCF Data Services

Related pages:
- .NET Business Transaction Detection
- .NET Agent Configuration Properties

This topic describes how to configure the AppDynamics .NET Machine agent to enable instrumentation for Windows Communication Foundation (WCF) Data Services including WCF RIA Services for Microsoft LightSwitch.

1. Open the `config.xml` file for editing as administrator. See 'Where to Configure Agent Properties' on Administer the .NET Agent
2. Copy the code block below to a child element of the Machine Agent element. See .NET Agent Configuration Properties

```xml
<instrumentation>
    <instrumentor name="WCFDSEntryInstrumentor" enabled="true" />
</instrumentation>
```

For example:

```xml
<?xml version="1.0" encoding="utf-8"?>
<appdynamics-agent
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
...
<machine-agent>
    <!--Enable instrumentation for WCF Data Services correlation-->
    <instrumentation>
        <instrumentor name="WCFDSEntryInstrumentor" enabled="true" />
    </instrumentation>
</machine-agent>
...
</appdynamics-agent>
```

The configuration syntax is `enabled="true"`.  
3. Save the `config.xml` file.  
5. Restart instrumented applications for your changes to take effect.
Configure Machine Snapshots for .NET

Default Settings and Configuration Considerations

By default the .NET Machine Agent takes machine snapshots under the following conditions:

- Periodic collection: The agent takes one snapshot every 10 minutes.
- Breached thresholds: The .NET Machine agent takes samples of machine statistics every 10 seconds within a 10-minute window. For each sample, the agent checks the CPU percent usage, the memory percent usage, and the oldest item in the IIS application pool queue. The agent flags a sample as a violation when the current usage meets or exceeds one of the following thresholds:
  - CPU at 80% or higher
  - Memory at 80% or higher
  - IIS application pool queue item older than 100 milliseconds

The agent takes a snapshot when it identifies 6 violations of a single type, such as CPU usage, within the window. The agent only takes one snapshot per window for breached thresholds.

With the default window size of 10 minutes and the violations per window of six, the sixth violation of a single type triggers a machine snapshot:
Before you change the machine snapshot settings, decide which configuration options work best for your environment. The following questions and considerations should help you decide how to set your thresholds:

- What percentage CPU or memory usage might flag the beginnings of an issue in your environment?
- How long is too long for items to wait in the IIS queue?
- Do you expect occasional CPU or memory spikes?
- Are periodic collections every ten minutes frequent enough?
- Make sure the value for violations is larger than the number of samples per window.

For example, if you decrease the window size to 60 seconds and take six samples per window, the agent takes samples at the same frequency as the default settings, once every 10 seconds. However, you are likely to get more snapshots because the agent only takes one snapshot per window. If you set the violations per window to five, the agent takes a snapshot any time half the samples in the window violate a specific threshold.

**Configure machine snapshots for the .NET Machine Agent**

Configure all instrumentation settings for the .NET Machine Agent in the config.xml file. See "Where to Configure Agent Properties" on Administer the .NET Agent

1. Open the config.xml file for editing as an administrator. See 'Where to Configure Agent Properties' on Administer the .NET Agent
2. Copy the code block below to a child element of the AppDynamics Agent element. See AppDynamics Agent Element

```xml
<machine-agent>
  <machine-snapshot enabled="true" window-size="600"
    samples-per-window="60" violations-per-window="6"
    max-percent-cpu="80" max-percent-memory="80"
    max-queue-item-age="100" periodic-collection="600" />
</machine-agent>
```
2. If you have already customized the .NET Machine Agent, only copy the `machine-snapshot` element and paste it as a child of the `machine-agent` element.

3. Edit the values for the `machine-snapshot` element attributes:

   ```xml
   <?xml version="1.0" encoding="utf-8"?>
   <appdynamics-agent
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   ...
   <machine-agent>
     <!--Configure machine snapshots-->
     <machine-snapshot enabled="true" window-size="60"
      samples-per-window="10" violations-per-window="5"
      max-percent-cpu="80" max-percent-memory="80"
      max-queue-item-age="100" periodic-collection="600"/>
     </machine-agent>
   ...
   </appdynamics-agent>
   
   4. Save the `config.xml` file.
   5. Restart the AppDynamics.Agent.Coordinator service.
Configure Runtime Reinstrumentation for .NET

By default, you must restart your monitored application after you create or update:

- POCO transaction detection entry points
- POCO service endpoints
- custom exit points
- method invocation and HTTP data collectors
- information points

When you enable runtime reinstrumentation, the .NET Agent detects the above types of instrumentation changes and automatically requests the CLR to recompile the affected code. With runtime reinstrumentation enabled you do not need to restart your application for the instrumentation changes to take effect.

**Requirements**

Runtime reinstrumentation works for .NET Framework 4.5.2 and greater. You can verify the framework version in the `web.config` file of your application as follows:

```xml
<system.web>
    <compilation targetFramework="4.5.2" />
    <httpRuntime targetFramework="4.5.2" />
</system.web>
```

**Configure Runtime Reinstrumentation for .NET**

Configure all instrumentation settings for the .NET Agent in the `config.xml` file. See [Administer the .NET Agent](#).

1. Open the `config.xml` file for editing as an administrator. See 'Where to Configure Agent Properties' on [Administer the .NET Agent](#)
2. Copy the code block below to a child element of the `app-agents` element. See [Default Profiler Element](#)

```
<profiler>
    <runtime-reinstrumentation enabled="true" interval="60000" />
</profiler>
```

For example:
3. If appropriate configure the runtime-reinstrumentation options:
   - enabled: Set to true to enable runtime reinstrumentation. Default: false
   - interval: The frequency the agent checks for configuration updates in milliseconds. Default: 60000.

   Because runtime reinstrumentation adds a small amount of system overhead, AppDynamics recommends a minimum interval of 1 minute or 60000 milliseconds.

   The .NET Agent checks for configuration updates every minute, so it may take up to 2 minutes for reinstrumentation to take effect with the default setting of 60000 milliseconds.

4. Save the config.xml file.
5. Restart the AppDynamics.Agent.Coordinator service.
6. Restart instrumented applications for your changes to take effect.
Manage Configuration for .NET

On this page:
- Requirements
- Identify the Configuration File
- Import Configuration
- Assign Configuration to Machines
- Opt Out of Configuration Management
- Restore an Old Configuration

Related pages:
- Administer the .NET Agent
- .NET Agent Configuration Properties

Highly-scaled environments frequently contain multiple machines that run the same .NET applications and therefore use the same AppDynamics .NET Agent configuration. Manual configuration updates to several machines can be tedious and error-prone.

Use the Config Management tool in the AppDynamics Controller to upload and manage .NET Agent config.xml files. By default the .NET Agent checks for configuration updates from the controller every minute. Use the Config Management tool to assign configurations to machines where the .NET Agent is running.

Click Settings > AppDynamics Agents > Config Management to manage all the .NET Agent configurations files for your machines.

Requirements
- Membership in a role with 'Administer user, groups, roles, authentication, and so on' account level permission.
- A valid config.xml configuration file. The Config Management tool does not validate the config.xml file.
- .NET Agent 4.1 or later installed on machines where you plan to deploy configuration.

.NET agents deployed on Azure cloud services using the AppDynamics NuGet package do not register for Config Management because web role and worker role machine names are not always unique. Duplicate machine names could lead to unexpected consequences.

Identify the Configuration File

To start managing your configurations in the Controller, identify a configuration file or set of configuration files to upload. If you are not familiar with .NET Agent configuration, AppDynamics recommends you use the Agent Configuration Utility on each machine to configure the agent. For more information, see Administer the .NET Agent and Configure the .NET Agent.

For a full list of configuration properties, see .NET Agent Configuration Properties.

- Use config.xml files configured with the AppDynamics Agent Configuration Utility to help ensure the XML is valid.
- Use the config.xml file from a previously-configured machine as a template for similar machines that need the exact same configuration.

Import Configuration

The Import Config dialog lets you upload a config.xml file to the Controller and specify a unique name for the configuration.

- Use a naming convention that helps you identify the type of machine for the configuration.
- Consider using a date or version scheme for the configuration name.
- Click a config then click Edit Config to modify the config name.
- Double-click a config to display the XML. You can copy the XML, but you cannot modify it.

Assign Configuration to Machines
The Config Management tool lets you assign machines to a configuration or change the configuration from one to another. After you assign the configuration in the Controller, the .NET Agent automatically downloads the configuration file.

- If a machine doesn't currently have a managed configuration, the Config Management tool displays it following Machine with no assigned templates.
- Drag and drop to assign a machine to a configuration.
- The .NET Agent checks for updated configurations every minute.
- The agent backs up the old config.xml file as config-<date>.xml in the backup subdirectory of the econfig directory. See .NET Agent Directory Structure
- If the agent detects an invalid config.xml file, it does not apply the managed configuration and it raises an Agent Configuration Error event in the Controller. The event summary indicates the affected tiers and nodes.
- Regardless of the Controller connection information in the managed configuration, the agent always retains existing Controller connection information.
- After the agent receives and validates the assigned configuration, any changes take effect after the app pools recycle or IIS restarts. You must restart Windows services or standalone applications. It is unnecessary to restart the AppDynamics Agent Coordinator service.

Opt Out of Configuration Management

To configure an agent so that it does not receive configuration files from the Controller, set the enable_config_deployment Controller attribute to false in the config.xml file. See 'Controller Element' on .NET Agent Configuration Properties

For example:

```xml
<controller host="mycontroller.saas.appdynamics.com" port="443" ssl="true" enable_config_deployment="false">
```

Restore an Old Configuration

If you mistakenly assign a configuration, use the following steps to restore a previous version of the config.xml:

1. On the Config Management window, drag the machine from its currently assigned config to Machine with no assigned templates.
2. On the machine itself, rename the existing configuration file, config.xml, to a name such as config.old.xml.
3. Copy the backup config file from the Backup subdirectory to the Config directory. Backup files are named in the following format: config-<date>.xml.
4. Rename the backup config file to config.xml.
5. Optionally edit the config.xml to opt out of Config Management.
6. Restart the AppDynamics Agent Coordinator service.
7. Restart Windows services, instrumented Windows services, and instrumented standalone applications.
The location for the .NET Agent log files is %ProgramData%\AppDynamics\DotNetAgent\Logs.

See Agent Log Files for a description of how the logs are organized into sets that rollover.

Log File Size

Within a set, the first file for the agent log can reach 5MB. This file never gets deleted, because it contains valuable information about the context in which the agent was started.

Each of the remaining logs in the set can reach a maximum of 5MB.

Allowing for several restarts and all possible logs that could be generated—.NET Agent, Business Transaction, and REST—the maximum number of rollover logs is five. The maximum size per set is 5MB + (5 X 5MB) = 30MB.

Some log files, such as SamplingInfo and Winston are configured differently and can be 9MB in size with only two archive files.

Profiler logging is different, we log to the Profiler folder and one file per instrumented process is created. This is done by a different logging framework called Boost. Boost uses the archive files and max size, however, it also has two other limits, one to limit the number of files in the Profiler folder (defaults to 50) and the other max size (defaults to 30MB).

Controlling the .NET Agent Logging Level

You control the logging level for the .NET Agent by changing the value of the `min level` parameter in the `AppDynamicsAgentLog.config` file found in the .NET Agent logs directory. The log configuration is per server. The .NET Machine Agent and the .NET Agent read the same configuration file and write to the same set of output files.
<rules>
  <logger name="com.appdynamics.REST.HeartBeatLog" minLevel="Info" writeTo="RESTHearbeat" final="true" />
  <logger name="*" minLevel="Warn" writeTo="Warnfile" final="true" />
  <logger name="com.appdynamics.BusinessTransactions" minLevel="Info" writeTo="btlog" final="true" />
  <logger name="com.appdynamics.bci.*" minLevel="Info" writeTo="bcifile" final="true" />
  <logger name="com.appdynamics.REST.*" minLevel="Info" writeTo="RESTfile" final="true" />
  <logger name="AppDynamics.Agent.Coordinator.Recovery.*" minLevel="Info" writeTo="CoordinatorRecovery" final="true" />
  <logger name="com.appdynamics.METRICS.MetricSender" minLevel="Info" writeTo="RESTfile" final="true" />
  <logger name="AppDynamics.Winston.*" minLevel="Info" writeTo="Winston" final="true" />
  <logger name="com.appdynamics.tm.AFastBackendResolver" minLevel="Warn" writeTo="logfile" final="true" />
  <logger name="com.appdynamics.ManagedAgentAPI.DumpStats" minLevel="Trace" writeTo="SamplingInfo" final="true" />
  <logger name="*" minLevel="Info" writeTo="logfile" />
</rules>
.NET Microservices Agent

To monitor standalone applications on different platforms, install the .NET microservices agent on the solution of your choice. Where and how you install the .NET microservices agent depends on your platform—for example, Azure App Service or Service Fabric—and what applications you want to monitor. For example, you can now monitor applications developed using .NET Core for Windows using the .NET microservices agent.

Before You Begin

To accurately determine the steps you need to take to deploy a .NET microservices agent to your solution, review the following decision matrix to pick the appropriate path for your platform and applications.

Install the .NET Microservices Agent

Review the following table to determine which NuGet package you should use to install the .NET microservices agent. For details about managing NuGet packages, see the documentation for your version of Visual Studio.

<table>
<thead>
<tr>
<th>NuGet Package</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AppDynamics.Agent.Distrib.Micro.Windows</td>
<td>AppDynamics NuGet package for .NET. This package should not be installed directly and is intended for download and file distribution. This package is used for Azure Service Fabric deployments. See Install AppDynamics for Azure Service Fabric for instructions.</td>
</tr>
<tr>
<td>AppDynamics.Agent.Windows</td>
<td>AppDynamics .NET Core microservices agent for Windows. Recommended for standalone installations. See Install the .NET Core Microservices Agent for Windows for deployment instructions. NOTE: This package does not support .NET Framework, only .NET Core for Windows.</td>
</tr>
</tbody>
</table>
.NET Core Microservices Agent Support

Supported Runtime Environments

This section lists the environments where the .NET Microservices Agent does some automatic discovery after little or no configuration.

The .NET microservices agent works with .NET Core 2.0 and 2.1 on the following operating systems:

- Microsoft Windows Server 2008 R2
- Microsoft Windows Server 2012
- Microsoft Windows Server 2012 R2
- Microsoft Windows Server 2016
- Microsoft Windows 7, 8.1, 10

Microsoft Windows Azure

- Azure App Services for .NET Core 2.0 and 2.1 environments in the Azure Portal:
  - Web Apps
  - API Apps
  - Container Services

Automatically Discovered Business Transactions

The .NET Microservices Agent discovers business transactions for the following frameworks by default. The agent enables detection without additional configuration.

<table>
<thead>
<tr>
<th>Type</th>
<th>Custom Configuration Options?</th>
<th>Downstream Correlation?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASP.NET Core</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Web Services including SOAP</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Message Queues

<table>
<thead>
<tr>
<th>Type</th>
<th>Custom Configuration Options?</th>
<th>Downstream Correlation?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microsoft Service Bus</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>RabbitMQ</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Supported Loggers for the .NET Agent

- Log4Net
- NLog
- System Trace
- Windows Event Log

If you are using a different logger, see Error Detection.

Remote Service Detection

The .NET Agent automatically detects the following remote service types. The agent enables detection by default. You don't need to perform extra configuration.

<table>
<thead>
<tr>
<th>Type</th>
<th>Custom Configuration Options?</th>
<th>Async Detection?</th>
<th>Downstream Correlation?</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP</td>
<td>Yes</td>
<td>See Asynchronous Exit Points for .NET</td>
<td>Yes</td>
</tr>
<tr>
<td>MongoDB: C# and .NET MongoDB Driver version 1.10, 2.0</td>
<td>No</td>
<td>See Asynchronous Exit Points for .NET</td>
<td>N/A</td>
</tr>
<tr>
<td>Web Services, including SOAP</td>
<td>Yes</td>
<td>See Asynchronous Exit Points for .NET</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Data Integration
Microsoft BizTalk Server 2010, 2013 | No | Yes | See Correlation Over Microsoft BizTalk

<table>
<thead>
<tr>
<th>Message Queues</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache ActiveMQ NMS framework and related MQs</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>RabbitMQ</td>
<td>See RabbitMQ Backends for .NET</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

* The agent discovers asynchronous transactions for the Microsoft .NET 4.5 framework. See Asynchronous Exit Points for .NET

**Data Storage Detection**

The .NET Agent automatically detects the following data storage types. The agent enables detection by default. You don’t need to perform extra configuration.

<table>
<thead>
<tr>
<th>Type</th>
<th>Customizable Configuration?</th>
<th>Async Detection?*</th>
<th>AppD for Databases?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADO.NET (see supported clients below)</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

* The agent discovers asynchronous transactions for the Microsoft .NET 4.5 framework. See Asynchronous Exit Points for .NET

**Supported ADO.NET Clients**

AppDynamics can monitor any ADO.NET client version and type. Clients we have tested include the following:

<table>
<thead>
<tr>
<th>Database Name</th>
<th>Database Version</th>
<th>Client Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL</td>
<td>5.x</td>
<td>Connector/Net and ADO.NET</td>
</tr>
</tbody>
</table>

Microsoft, SQL Server, and Windows are registered trademarks of Microsoft Corporation in the United States and other countries.
Install the .NET Core Microservices Agent for Windows

If the application you want to monitor is executed on Windows, install the AppDynamics .NET Core microservices agent for Windows to your project. Then, set the environment variables described below for your Controller.

Before you Start

Before you install the .NET Core microservices agent, make sure you have access to the AppDynamics Controller where your application metrics will appear.

Install the .NET Core Microservices Agent

1. Install the AppDynamics.Agent.Windows NuGet package in your project. See your IDE documentation.
2. Set up the following environment variables for your system application:

   ```
   CORECLR_ENABLE_PROFILING=1
   CORECLR_PROFILER={39AEABC1-56A5-405F-B8E7-C3668490DB4A}
   CORECLR_PROFILER_PATH_32=<actual_path>\AppDynamics.Profiler_x86.dll
   CORECLR_PROFILER_PATH_64=<actual_path>\AppDynamics.Profiler_x64.dll
   ```

   Where `<actual_path>` is the complete path to the AppDynamics.Profiler.dll.
3. Open the AppDynamicsConfig.json file and update the file with your AppDynamics Controller information.

4. You have the following options to configure the .NET Agent to connect to the AppDynamics Controller:
   - During development: As shown in the previous screenshot, you can enter your environment variables in the `AppDynamicsConfig.json` file, and save it in source control.
   - During build: Define your `msbuild` parameters or environment variables that are passed to the `AppDynamicsConfig.json` file at build time. The `AppDynamicsConfig.json` file does not exist at build time, so if you are defining your `msbuild` parameters or environment variables at build time, you will need to ignore it in source control so that the new `AppDynamicsConfig.json` file can be created.
   - During runtime: Enter your environment variables in Azure.
5. Deploy the application.
AppDynamicsConfig.json File

You use the AppDynamicsConfig.json file to:

- Configure the connection settings to your Controller after installing your .NET microservices agent.
- Disable instrumentation for troubleshooting.
- Assembly whitelisting.
- Disable inject fields.
- Include blacklists and whitelists.

You can use environment variables in the AppDynamicsConfig.json file for node, tier, and application names. To include an environment variable, reference it by specifying %<variable>%, which can be combined with the other characters; for example, Web-%COMPUTERNAME% will be translated to Web-HOST23.

The following example.json file is just a sample and should not be copied/pasted into your AppDynamicsConfig.json file.

```json
{
  "controller": {
    "host": "<<host_name>>",
    "port": 8080,
    "ssl": false,
    "enable_tls12": false,
    "account": "customer1",
    "password": "password",
    "proxy": {
      "host": "test-host",
      "port": 8080,
      "authentication": {
        "username": "test-username",
        "password": "test-password",
        "domain": "test.com"
      }
    }
  },

  "application": {
    "name": "Dotnet",
    "tier": "tier1",
    "node": "node2"
  },

  "instrumentors": {
    "customCorrelationConfig": "custom-correlation.xml",

    "enable": [
      "ThreadStartCLR2Instrumentor",
      "ThreadCorrelationThreadPoolCLR2Instrumentor"
    ],

    "disable": [
      "ASPdotNETInstrumentor"
    ]
}
```
"WebApiInstrumentor"

},
"assemblyWhitelist": [ "^mscorlib$",
  "^EQUALS$",
  "^App_Web_",
  "^STARTS_WITH$",
  "foo$",
  "ENDS_WITH$",
  "bar",
  "CONTAINS",
  "Or any other regex"
],
"disableInjectFields": false
"profiler": {
  "attachBlackList": [ "ProcessNameToBeBlackListed",
  "ProcessNameToBeBlackListed", ... ],
  "attachWhiteList": [ "ProcessNameToBeWhiteListed",
  "ProcessNameToBeWhiteListed", ... ]
},
"analytics": {
  "host": "localhost",
  "port": 9090
}

Note:

In 4.4.3, disableInjectFields are supported in .NET framework applications only.
In 4.5, disableInjectFields are supported in .NET core applications as well.

attachBlackList excludes a process if it is on the profiler list.
attachWhiteList excludes all processes except those listed in the list. If it is empty, it will include all processes.
Both lists relate to whether the profiler is detached from the process when the process launches, assuming the environment variables for the AppDynamics profiler are set correctly.

See Configure Custom Correlation for .NET Applications
.NET Core for Linux SDK

The AppDynamics .NET Core for Linux SDK lets you monitor the performance of .NET Core applications running on Linux.

The SDK supports the following features:

- Business transaction registration and error reporting
- Exit call with correlation and error reporting
- Metrics
- Snapshots with user properties but without callgraphs

Together, these tools give you visibility on application load and response times, and any custom metrics you define.

You monitor .NET Core applications by deploying the .NET Core for Linux SDK to your application directory and project.

Once running, the agent registers business transactions with the AppDynamics Controller. You can then see your application flow map and monitor performance in the Controller.

Before You Begin

1. Confirm you have access to a compatible Controller. See Agent and Controller Compatibility
2. Confirm the connection settings to the Controller where your agent will report data:
   - If you use a SaaS Controller, AppDynamics sent you the Controller host in your Welcome Email. Use port 443 for HTTPS or port 80 for HTTP.
   - If you use an on-premises Controller, you supplied the host and port at install time.
3. Verify you have access the machine where the application runs as a user account with privileges to install the agent software and restart the application.
4. Verify that the machine where the application runs can connect to the Controller. Proxies or firewalls on the network between the agent and Controller may require additional configuration.

Prepare to Install

Ensure you have the following:

- .NET Core SDK
- .NET Core application running on Linux
- AppDynamics.AgentSDK NuGet package
- Visual Studio or other IDE
- The .NET Core command-line interface (CLI)

Install the SDK

Complete the following steps manually, using Visual Studio or another integrated development environment (IDE).

1. To add the AppDynamics.AgentSDK NuGet package in the project, from the command line enter
dotnet add package AppDynamics.AgentSDK_x64

Once the command is run, there's a compatibility check to ensure the package is compatible with the frameworks in the project. If the check passes, a `<PackageReference>` element is added to the project file and `dotnet restore` is run implicitly.

2. Modify your project to use the AppDynamics SDK. See [.NET Core for Linux SDK Reference](#)

Add the SDK namespace and SDK code to the application. For example:

```csharp
using AppDynamics;
...
namespace coremvc.Controllers
{
    public class HomeController : Controller
    {
        public IActionResult SomePage()
        {
            var currCtxId = AgentSDK.StartBusinessTransaction("DemoBT", "ASP_DOTNET", "");
            // Code here calls to other ASP_DOTNET component
            ... 
            AgentSDK.StopBusinessTransaction(currCtxId);
            return View();
        }
    }
}
```

3. Run `dotnet build` or `dotnet publish`.

A warning message appears stating the AppDynamics configuration has not been updated. This is normal.

- Running `dotnet build` adds the following AppDynamics configuration file to your application:
  `<application_directory>
  AppDynamicsConfig.json`
- Running `dotnet publish` renames the `AppDynamicsConfig.json` file and copies it to the publish directory for the `.NET Core for Linux SDK` to reference and use:
  `<publish_directory>
  <project_name>.AppDynamicsConfig.json`

4. Proceed to [Using .NET Core for Linux SDK](#) for instructions on updating the `AppDynamics.Config.json` file and connecting the agent to the Controller.

Known Issues

**The requirement to disable loading of native images (NGEN) for self-contained apps compiled with .NET Core 2.0**

Due to a bug in the .NET Core runtime (see [https://github.com/dotnet/coreclr/issues/13021](https://github.com/dotnet/coreclr/issues/13021)), the command the .NET Core for Linux SDK profiler sends to the runtime to disable loading of native images is ignored. This prevents AppDynamics from instrumenting any methods in these assemblies which include most of the .NET Standard libraries.

A workaround for this bug is to disable using native images by setting the following environment variable:

```bash
COMPLUS_ReadyToRun=0
```
.NET Core for Linux SDK Supported Environments

On this page:
- Operating Systems
- Entry Point Types
- Exit Point Types

Related pages:
- .NET Core for Linux SDK

You must use the `StartBusinessTransaction` and `CreateExitCall` methods with the entry and exit point types listed below in order for the .NET Core for Linux SDK to recognize them. See .NET Core for Linux SDK Reference.

.NET Core 1.x, 2.0.x, and 2.1.x for Linux SDK Support

Operating Systems
- CentOS 7
- Ubuntu 14.04

Entry Point Types
- ASP.NET
- POCO

Exit Point Types
- HTTP
  - HOST
  - PORT
  - URL
  - QUERY STRING
- WCF
  - URL
  - OPERATION
- ADO.NET
  - HOST
  - PORT
  - DATABASE
  - VENDOR
  - CONNECTION STRING
Using .NET Core for Linux SDK

Before you can start instrumenting your applications with the .NET Core for Linux SDK, you need to get the current version of the .NET Core for Linux SDK. The topic describes how to connect the .NET Core for Linux Agent in your application to the Controller.

Once you have the deployed the SDK, you can customize the instrumentation of your application as described here. The SDK provides routines for creating and managing business transactions, transaction snapshots, backends, exit points and collecting custom metrics. This topic provides an overview of these concepts.

Configuration

1. Edit the `AppDynamicsConfig.json` file with the following information:

```json
{
    "controller": {
        "host": "<controller_host_name>",
        "port": <controller_port_name>,
        "account": "<controller_account_name>",
        "password": "<controller_account_key>",
        "ssl": <true if using https controller, or false>
        "proxy": {
            "host": "proxy-host",
            "port": 9090,
            "authentication": {
                "username": "proxy-user",
                "password": "proxy-password"
            }
        } },
    "application": {
        "name": "<application_name>",
        "tier": "<tier_name>",
        "node": "<node_name>"
    },
    "log": {
        "directory": "<log_folder_path>",
        "level": "<log_level>"
    }
}
```

2. Add the following environment variables to your application, then rebuild or republish your application.

<table>
<thead>
<tr>
<th>Environment Variable Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORECLR_PROFILER</td>
<td>{57e1aa68-2229-41aa-9931-a6e93bbc64d8}</td>
</tr>
<tr>
<td>CORECLR_ENABLE_PROFILING</td>
<td>1</td>
</tr>
<tr>
<td>CORECLR_PROFILER_PATH</td>
<td>Path to the libappdprofilerdynamic library. For example, &lt;application_folder_path&gt;/libappdprofiler.so</td>
</tr>
</tbody>
</table>

The location of the libappdprofiler.so library is determined by which .NET command you used to install the SDK. AppDynamics recommends using `dotnet build`, which is the first deployment command option in the following table.
<table>
<thead>
<tr>
<th>Type of Deployment</th>
<th>Command</th>
<th>Work Directory</th>
<th>Profiler Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Development</td>
<td>dotnet</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>build,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>dotnet</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>run</td>
<td>&lt;project_path&gt;</td>
<td>~/.nuget/packages/appdynamics.agentsdk/4.5.0/runtimes/linux-x64/r</td>
</tr>
<tr>
<td>Framework Dependent</td>
<td>dotnet</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>publish</td>
<td>&lt;project_path&gt;</td>
<td>&lt;project_path&gt;/bin/&lt;Flavor&gt;/publish/runtimes/linux-x64/native/lib</td>
</tr>
<tr>
<td>Self-contained</td>
<td>dotnet</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>publish</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-r</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>linux_x64</td>
<td>&lt;project_path&gt;</td>
<td>/&lt;project_path&gt;/bin/&lt;Flavor&gt;/linux_x64/publish/libappdprofiler.so</td>
</tr>
</tbody>
</table>

<project path> is the root directory of the application.

After you rebuild or republish your application, the agent registers to the Controller, and AppDynamics starts instrumenting your application. See .NET Core for Linux SDK Use Cases
**Business Transaction Definition**

You define a custom business transaction using the `StartBusinessTransaction` and `StopBusinessTransaction` SDK calls.

A downstream transaction in a distributed transaction cannot start until its upstream transaction has made its exit call. The order of transactions respects the nonlinear nature of .NET applications. An upstream transaction might end immediately after its exit call completes or sometime later. It is not necessary for an upstream transaction to end before or after its downstream transaction starts or ends. A downstream transaction can end before its upstream transactions have ended.

**Sample Business Transaction Creation**

This example shows how you use the `StartBusinessTransaction` and `StopBusinessTransaction` SDK calls to define the entry point and exit point of a business transaction.

```csharp
using AppDynamics;
...
namespace coremvc.Controllers
{
    public class HomeController : Controller
    {
        public IActionResult SomePage()
        {
            // define start of an ASP_DOTNET type business transaction named DemoBT
            var currCtxId = AgentSDK.StartBusinessTransaction("DemoBT", "ASP_DOTNET", ";
            // Code here calls to other ASP_DOTNET component
            ...
            AgentSDK.StopBusinessTransaction(currCtxId);
            return View();
        }
        ...
    }
}
```

**Exit Call Management**

An exit call is either a custom exit call or an automatically detected exit call. A transaction needs its exit call object to create correlation information to provide to a downstream transaction that needs to correlate with it.

**Custom Exit Calls**
If you want an exit call to be reported to the Controller, and it is not automatically detected by the agent, create a custom exit call with `createExitCall()` and then define the start and end of the exit call — `startExitCall()` and `endExitCall()`.

Use `AddIdentifyingPropertyToExitCall` to help you to identify this exit call in the transaction snapshot.

See .NET Core for Linux SDK Supported Environments for the list of backends that are automatically detected by the .NET Core for Linux SDK. If the exit call invokes a backend not listed on this page, you probably need to create a custom exit call.

Sample Exit Call Creation
using AppDynamics;
...
namespace coremvc.Controllers
{
    public class HomeController : Controller
    {
        public IActionResult SomePage()
        {
            // define start of an ASP_DOTNET type business transaction called DemoBT. StartBusinessTransaction statement equals the value of the currCtxId variable
            var currCtxId = AgentSDK.StartBusinessTransaction("DemoBT", "ASP_DOTNET", "");

            // define HTTP type exit call named CorrelationExit for the DemoBT business transaction. CreateExitCall statement equals the value of the exitCall variable
            var exitCall = AgentSDK.CreateExitCall(currCtxId, "HTTP", "CorrelationExit");

            // add identifying properties Host and Port, along with their respective values to the exit call
            AgentSDK.AddIdentifyingPropertyToExitCall(exitCall, "Host", request.RequestUri.Host);
            AgentSDK.AddIdentifyingPropertyToExitCall(exitCall, "Port", request.RequestUri.Port.ToString());

            // define the beginning of the exit call
            AgentSDK.StartExitCall(exitCall);

            // code here to make exit call
            ...

            // define end of the exit call
            AgentSDK.StopExitCall(exitCall);

            // code here calls to other ASP_DOTNET component
            ...

            // define end of the DemoBT business transaction
            AgentSDK.StopBusinessTransaction(currCtxId);
            return View();
        }
    }
}

Transaction Correlation Management
Transaction correlation is the functionality that maintains the business transaction context across all tiers (servers) traversed by a distributed transaction. The tiers may be built on platforms other than .NET. For example, Java, Node.js, and PHP tiers may participate in a distributed business transaction in which a .NET tier participates as the originating, continuing or terminating tier in the transaction.

The .NET Core for Linux SDK provides facilities for managing transaction correlation among various transactions, which can be custom or automatically-detected.

While the AppDynamics default auto-detection mechanism obtains transaction correlation information from an incoming HTTP request, the .NET Core for Linux SDK allows you to obtain this information from other sources, such as a custom field in a Redis cache entry, and to supply it to your custom transactions. You can use the Agent SDK classes to support forward correlation by supplying transaction correlation information on exit calls that you make to downstream transactions.

A downstream transaction starts after its immediate upstream transaction makes its exit call. That exit call could be a direct exit call to the next tier or a call to a backend service that does some processing, such as publishing a message onto a queue.

Sample Transaction Correlations Between Upstream and Downstream Tiers

Use Case 1: .NET Core on Linux application calling an upstream .NET Core on Linux application. Both applications are monitored by AppDynamics.
namespace coremvc.Controllers
{
    public class HomeController : Controller
    {
        public IActionResult SomePage()
        {
            var currCtxId = AgentSDK.StartBusinessTransaction("DemoBT", "ASP_DOTNET", ",");
            var client = new HttpClient();
            var request = new HttpRequestMessage()
            {
                RequestUri = new Uri("http://localhost:8080"),
                Method = HttpMethod.Get,
            }
            var exitCall = AgentSDK.CreateExitCall(currCtxId, "HTTP", "CorrelationExit");
            AgentSDK.AddIdentifyingPropertyToExitCall(exitCall, "Host", request.RequestUri.Host);
            AgentSDK.StartExitCall(exitCall);
            var correlationHeader = AgentSDK.GetCorrelationHeader(exitCall);
            request.Headers.Add("singularityheader",correlationHeader);
            await client.SendAsync(request);
            AgentSDK.StopExitCall(exitCall);
            AgentSDK.StopBusinessTransaction(currCtxId); 
        }
    }
}

Use Case 2: .NET Core on Linux application calling a downstream .NET Core on Linux application. Both applications are monitored by AppDynamics.
using AppDynamics;
...
namespace coremvc.Controllers
{
    public class HomeController : Controller
    {
        public IActionResult SomePage()
        {
            bool hasCorrelationHeader =
            Request.Headers.TryGetValue("singularityheader", out var correlationHeader);

            var currCtxId = AgentSDK.StartBusinessTransaction("DemoBT", "ASP_DOTNET",
            hasCorrelationHeader ? (string)correlationHeader:"");

            // Code here calls to other ASP_DOTNET component

            AgentSDK.StopBusinessTransaction(currCtxId);

            return View();
        }
    }

Use Case 3: Non .NET Core on Linux application monitored by AppDynamics calling a .NET Core on Linux application.
In order to correlate, the .NET Core on Linux application will have to include code such as the following:
using AppDynamics;  
...  
namespace coremvc.Controllers  
{  
    public class HomeController : Controller  
    {  
        public IActionResult SomePage()  
        {  
            // Obtain the HTTP header value of the page via the TryGetValue method and use that value to define correlationHeader  
            bool hasCorrelationHeader = Request.Headers.TryGetValue("singularityheader", out var correlationHeader);  

            // Define the start of the ASP_DOTNET type business transaction DemoBT to include the correlation header if available.  
            var currCtxId = AgentSDK.StartBusinessTransaction("DemoBT", "ASP_DOTNET",  
                hasCorrelationHeader ? (string)correlationHeader:"");  

            // Code here calls to other ASP_DOTNET component  
            ...  

            // Define the end of the DemoBT business transaction  
            AgentSDK.StopBusinessTransaction(currCtxId);  
            return View();  
        }  
    ...  
    }  
}  

Use Case 4: .NET Core on Linux application calling a non .NET Core on Linux application monitored by AppDynamics.  

In order to correlate, the .NET Core on Linux application will have to include code such as the following:
using AppDynamics;
...
namespace coremvc.Controllers
{
    public class HomeController : Controller
    {
        public IActionResult SomePage()
        {
            var currCtxId = AgentSDK.StartBusinessTransaction("DemoBT", "ASP_DOTNET", ");

            var client = new HttpClient();

            var request = new HttpRequestMessage()
            {
                RequestUri = new Uri("http://localhost:8080"),
                Method = HttpMethod.Get,
            };

            var exitCall = AgentSDK.CreateExitCall(currCtxId, "HTTP", "CorrelationExit");
            AgentSDK.AddIdentifyingPropertyToExitCall(exitCall, "Host", request.RequestUri.Host);
            AgentSDK.StartExitCall(exitCall);
            var correlationHeader = AgentSDK.GetCorrelationHeader(exitCall);
            request.Headers.Add("singularityheader", correlationHeader);
            await client.SendAsync(request);
            AgentSDK.StopExitCall(exitCall);
            AgentSDK.StopBusinessTransaction(currCtxId); } 
...
}
To instrument .NET Core for Linux applications, you use the .NET Core for Linux SDK. Once you have deployed the .NET Core for Linux SDK into your application, AppDynamics automatically instruments the application. You can customize the application instrumentation using the following business transaction and exit call methods.

This page describes the methods defined in the SDK.

**Using the SDK**

**Namespace:** AppDynamics.AgentSDK  
**Assembly:** AppDynamics.Agent.SDK.dll

**Business Transaction Methods**

### StartBusinessTransaction

Starts the Business Transaction.

**Format:** `StartBusinessTransaction(<name>, <entryPointType>, <correlationHeader>)`

**Parameters:**

- **name** – Business Transaction name.
- **entryPointType** – Type of entry point, such as ASP_DOTNET and POCO
- **correlationHeader** – AppDynamics correlation header. Optional.

**Return Type:** AppDynamics.BusinessTransaction

### StopBusinessTransaction

Stop the Business Transaction

**Format:** `AgentSDK.StopBusinessTransaction(<bt>)`

**Parameters:** **bt** – Business transaction object returned by the StartBusinessTransaction method.

**Return Type:** void
AddDataToTransactionSnapshot

Add identifying information to the transaction snapshot.

Format: AgentSDK.AddDataToTransactionSnapshot(<bt>, <key>, <value>)

Parameters:

- bt – The business transaction for which AppDynamics creates a snapshot. Returned by the StartBusinessTransaction method.
- key – Key in key/value pair
- value – Value in key/value pair

Return Type: void

Exit Call Methods

CreateExitCall

Create an Exit Call

Format: AgentSDK.CreateExitCall(<bt>, <exitPointType>, <name>, <isAsync>)

Parameters:

- bt – BusinessTransaction on which to create an exit call.
- exitPointType – Type of exit call, such as HTTP, WCF, and ADO.NET
- name – Name of exit call.
- isAsync – Boolean value to identify exit call as asynchronous. Optional. Default value: false

Return Type: AppDynamics.ExitCall

AddIdentifyingPropertyToExitCall

Add properties to the exit call.

Format: AgentSDK. AddIdentifyingPropertyToExitCall(ExitCall <exitCall>, <string_property>, <string_value>)

Parameters:

- exitCall – Business transaction for which AppDynamics creates a the snapshot. Returned by StartBusinessTransaction.
- string_property – Name of the property. For example, the URL.

Return Type: void

StartExitCall

Start an exit call to the specified backend as part of a business transaction

Format: AgentSDK.StartExitCall(ExitCall <exitCall>)

Parameters: exitCall – ExitCall type returned by CreateExitCall.

Return Type: void

StopExitCall

Complete the exit call

Format: AgentSDK.StopExitCall(ExitCall <exitCall>)

Parameters: exitCall – ExitCall type returned by CreateExitCall.

Return Type: void

AddErrorToTransaction

Register an error for the business transaction
Format: AgentSDK. AddErrorToTransaction (<bt>, <name>, <message>, <errorCode>, <markBtAsError>)

Parameters

- `<name>` – Name of Error.
- `<message>` – Error message.
- `<errorCode>` – Integer value of the error.
- `<markBtAsError>` – Boolean value to qualify the BusinessTransaction as an error. (optional. Default Value: false)

Return Type: void

**AddErrorToExitCall**

Register an error for the exit call

Format: AgentSDK.AddErrorToExitCall(ExitCall <exitCall>, string <errorName>, string <errorMessage>, bool <markBtAsError>)

Parameters

- `<exitCall>` – ExitCall on which the error should be registered.
- `<errorName>` – Name of Error
- `<errorMessage>` – Error message
- `<markBtAsError>` – Boolean value to qualify the ExitCall as an error.

Return Type: void

**GetCorrelationHeader**

Sets the AppDynamics correlation header at the upstream tier. The downstream tiers read this value to continue correlation. Example available below.

Format: AgentSDK.GetCorrelationHeader(<exitCall>)

Parameters: `<exitCall>` – Exit Call created for business transaction to downstream tier monitored by AppDynamics.

Return Type: string
The AppDynamics .NET Agent for Linux lets you monitor the performance of .NET Core applications running on Linux OS.

The agent supports the following features:

- ASP.NET MVC transactions  See Name MVC Transactions by Area, Controller, and Action
- ASP.NET WebAPI transactions
- ASP.NET Razor Pages transactions
- HttpClient backend calls
- ADO.NET backend calls such as SqlClient, MySql, SQLite, and PostgreSQL
- EntityFramework
- Upstream and downstream correlation using HTTP
- Snapshot waterfall view with async support
- Redis exit calls

The AppDynamics .NET Agent for Linux can be run on microservices platforms such as those based on Docker containers. Support for .NET Core applications will soon be included into PCF. See AppDynamics Pivotal Cloud Foundry and .NET Agent for Linux Advanced Configuration Options

The .Net Agent for Linux is supported on AppDynamics Controller version 4.4.1+. See .NET Agent for Linux Supported Environments
Install the .NET Agent for Linux

On this page:
- Getting Started
- Deployment
- Startup Flow

Related pages:
- .NET Agent for Linux
- Create an ASP.NET Core web app in Azure
- Overview of Application Monitoring
- Business Transactions

Getting Started

Before You Begin

Review the following prerequisites before you begin:

1. Confirm you have access to a compatible Controller. See Agent and Controller Compatibility
2. Confirm the connection settings to the Controller where your agent will report data:
   - If you use a SaaS Controller, AppDynamics sent you the Controller host in your Welcome Email. Use port 443 for HTTPS or port 80 for HTTP.
   - If you use an on-premises Controller, you supplied the host and port at install time.
3. Verify you have access the machine where the application runs as a user account with privileges to install the agent software and restart the application.
4. Verify that the machine where the application runs can connect to the Controller. Proxies or firewalls on the network between the agent and Controller may require additional configuration.
5. Verify that the application environment meets the requirements on .NET Agent for Linux Supported Environments prior to running the agent.

Applications and Licenses Overview

Ensure you have the following applications and licenses:

- .NET Core application running on Linux
- .NET license for each .NET Core application running on Linux. See License Management

Binaries Overview

Down the agent binaries from the AppDynamics Download Center and then extract them from the zip file into the desired folder.

The archive contains the following files:

- AppDynamics.Agent.netstandard.dll
- AppDynamicsConfig.json.template
- libappdprofiler.so
- README.md

Deployment

There are two options for starting your deployment using Docker:

- Create Your Own Image, start your agent from scratch
- Run an Existing Image, if you already have the .NET agent enabled

Create Your Own Image

You can use the ASP.NET sample application from Microsoft and a Dockerfile to build your own image and get started:

1. Download the aforementioned binaries.
2. Update the Dockerfile variables to configure the connection to the Controller and your application identity in AppDynamics.
3. Create the Docker image.

The following is an example Dockerfile with commented instructions:

```bash
FROM mcr.microsoft.com/dotnet/core/samples:aspnetapp

####### Requirements
# Have the following files alongside the Dockerfile:
# * libappdprofiler.so
# * AppDynamics.Agent.netstandard.dll

####### Instructions
# Building image: docker build --rm -t appdynamicstest:latest .
# Running container: docker run --rm -p 8000:80 appdynamicstest:latest
# Open the application using http://localhost:8000/

# Copy agent binaries to the image from current folder
RUN mkdir -p /opt/appdynamics/dotnet
ADD libappdprofiler.so /opt/appdynamics/dotnet/
ADD AppDynamics.Agent.netstandard.dll /opt/appdynamics/dotnet/

# Mandatory settings required to attach the agent to the .NET application
ENV CORECLR_PROFILER={57e1aa68-2229-41aa-9931-a6e93bbc64d8} \ 
    CORECLR_ENABLE_PROFILING=1 \ 
    CORECLR_PROFILER_PATH=/opt/appdynamics/dotnet/libappdprofiler.so

# Configure connection to the controller
ENV
APPDYNAMICS_CONTROLLER_HOST_NAME=controller.saas.appdynamics.com
ENV APPDYNAMICS_CONTROLLER_PORT=443
ENV APPDYNAMICS_CONTROLLER_SSL_ENABLED=true
ENV APPDYNAMICS_AGENT_ACCOUNT_NAME=account-name
ENV APPDYNAMICS_AGENT_REUSE_NODE_NAME=true
ENV APPDYNAMICS_AGENT_REUSE_NODE_NAME_PREFIX="Instance"

# Configure application identity in AppDynamics
ENV APPDYNAMICS_AGENT_APPLICATION_NAME="My Application"
ENV APPDYNAMICS_AGENT_TIER_NAME="Sample Tier"
ENV APPDYNAMICS_AGENT_REUSE_NODE_NAME=true
ENV APPDYNAMICS_AGENT_REUSE_NODE_NAME_PREFIX="Instance"

# It is possible to configure .NET agent using
AppDynamicsConfig.json configuration file instead of environment variables
# ADD AppDynamicsConfig.json /opt/appdynamics/dotnet/
```

**Configuring using AppDynamicsConfig.json:** If you are configuring your agent using environment variables along with the json configuration file or just with the json configuration file, include the FULL_AGENT feature in the json file. Otherwise, the
Run an Existing Image

You can run an existing image with an agent enabled. This is the simplest way to get started as it does not require building a new image:

1. Download the aforementioned binaries.
2. Update the Docker command variables to configure the connection to the Controller and your application identity in AppDynamics.
3. Run the command in the desired folder where you extracted the agent binaries from the zip file.
   You can replace $(pwd) in the command with this folder.

The following example runs the sample application. It assumes you have AppDynamics binaries in the current directory:

```bash
docker run \
  -p 8000:80 \
  -e CORECLR_PROFILER={57e1aa68-2229-41aa-9931-a6e93bbc64d8} \
  -e CORECLR_ENABLE_PROFILING=1 \
  -e CORECLR_PROFILER_PATH=/opt/appdynamics/dotnet/libappdprofiler.so \
  -e APPDYNAMICS_CONTROLLER_HOST_NAME=controller.saas.appdynamics.com \
  -e APPDYNAMICS_CONTROLLER_PORT=443 \
  -e APPDYNAMICS_CONTROLLER_SSL_ENABLED=true \
  -e APPDYNAMICS_AGENT_ACCOUNT_NAME=account-name \
  -e APPDYNAMICS_AGENT_ACCOUNT_ACCESS_KEY=access-key \
  -e APPDYNAMICS_AGENT_APPLICATION_NAME="My Application" \
  -e APPDYNAMICS_AGENT_TIER_NAME="Sample Tier" \
  -e APPDYNAMICS_AGENT_REUSE_NODE_NAME=true \
  -e APPDYNAMICS_AGENT_REUSE_NODE_NAME_PREFIX="Instance" \
  -v $(pwd):/opt/appdynamics/dotnet/ \
  mcr.microsoft.com/dotnet/core/samples:aspnetapp
```

The command mounts agent files as a volume and sets environment variables required for the agent to attach.
**Startup Flow**

During the application startup, the AppDynamics agent writes messages to the console as well as the application and .NET framework.

The following sample Agent startup console output indicates proper agent initialization:

```
profiler(Info): Starting .NET Agent. More info can be found from /tmp/appd/dotnet
profiler(Info): .NET Agent (4.5.7.0) starts successfully
profiler(Info): Agent has registered on the platform using app:My Application/tier:Sample Tier/node:Instance1
```

This is a sample view of the application in the Controller:

See .NET Agent for Linux Troubleshooting
.NET Agent for Linux Supported Environments

On this page:
- Supported Environments
- Limitations of the Current Version

Related pages:
- .NET Supported Environments

Supported Environments

The AppDynamics .Net Agent for Linux is supported on the following environments.

Operating Systems

- CentOS 7+
- Red Hat Enterprise Linux 7
- Debian 9+
- Ubuntu 14+
- Alpine 3.7+

.NET Core

- 2.0+
- 2.2 (recommended)

ASP.NET core

- MVC
- Razor
- WebAPI

Backend Calls

- HttpClient backend calls
- ADO.NET backend calls (SqlClient, MySql, SQLite, PostgreSQL)
- EntityFramework

Application environment:

- Applications running in a container (needs to match OS)
- Applications running on a host/VM (needs to match OS)

AppDynamics Controller version 4.4.1+

New in 4.5.9 AppDynamics Controller environment

The ability to customize Business Transaction detection that is now supported by .NET Agent for Linux 4.5.9 requires Controller versions 4.5.2 and higher. The .NET Agent 4.5.9 works with Controller v4.4.1+, but the customizable transaction detection and configuration capabilities require v4.5.2+.

Limitations of the Current Version

There are a few locations in the Controller which do not support the .NET Agent for Linux. These screens appear empty in such cases:

- Node memory
- Slow snapshot drill down
- Tier IIS AppPools
- Tier network dashboard
.NET Agent for Linux Environment Variables

On this page:

- Agent Account Name
- Agent Application Name
- Agent Account Password
- Agent Node Name
- Agent Reuse Node Name
- Agent Reuse Node Name Prefix
- Agent Tier Name
- Analytics SSL Enabled
- Analytics Hostname
- Analytics Port
- Controller Hostname
- Controller Port Number
- Controller SSL Enabled
- Controller SSL Certfile
- HTTP Proxy Host
- HTTP Proxy Port
- HTTP Proxy User
- HTTP Proxy Password

Related pages:

- .NET Agent Configuration Properties
- Administer the .NET Agent
- Configure the .NET Agent
- Name .NET Tiers

This topic is a reference for the configuration properties for the .NET Agent for Linux that can be set using system environment variables.

**Agent Account Name**

Specifies the account name for the SaaS or multi-tenant Controller.

Environment Variable: `APPDYNAMICS_AGENT_ACCOUNT_NAME`

Type: String

Default: For single-tenant Controllers, the agent assumes the default of `customer1` if you do not specify an account name.

Required: Only for SaaS or multi-tenant Controllers.

**Agent Application Name**

Specifies the business application you use in the Controller. If the application name does not exist, the Controller will create it when the agent registers. See *Overview of Application Monitoring*

Environment Variable: `APPDYNAMICS_AGENT_APPLICATION_NAME`

Type: String

Default: None

Required: Yes

**Agent Account Password**

Specifies the account access key for the Controller.

Environment Variable: `APPDYNAMICS_AGENT_ACCOUNT_ACCESS_KEY`

Type: String

Default: None

Required: Yes

**Agent Node Name**

The name of the node.

In general, the node name must be unique within the business application and physical host. To use the same node name for multiple nodes on the same physical machine, create multiple virtual hosts using the Unique Host ID property. See Unique Host ID.
Environment Variable: **APPDYNAMICS_AGENT_NODE_NAME**
Type: String
Default: None
Required: Yes

**Agent Reuse Node Name**

Set this environment variable to true to reuse node names in AppDynamics. When you set the property to true, you don't need to supply a node name, but you do need to provide a node name prefix using `APPDYNAMICS_AGENT_REUSE_NODE_NAME_PREFIX`.

This property is useful for monitoring environments where there are many CLRs with short life spans. When true, AppDynamics reuses the node names of historical CLRs for new CLRs. Reusing node names avoids a proliferation of differently named nodes in AppDynamics over time, particularly when the nodes are essentially identical processes that run over different times.

AppDynamics generates a node name with App, Tier, and Sequence number. The node names are pooled. For example, the sequence numbers are reused when the nodes are purged (based on the node lifetime).

When the .NET Agent for Linux starts up, it logs output to the console until it registers with the Controller and the Controller generates the node name.

The Controller reuses node names based on the `node.retention.period` property. See Historical and Disconnected Nodes.

Environment Variable: **APPDYNAMICS_AGENT_REUSE_NODE_NAME**
Type: Boolean
Default: False
Required: No

Example: Using the following environmental variable specifications, the Controller generates a node name with the prefix `reportGen`. Node names will have suffixes `-1`, `-2`, and so on, depending on the number of nodes are running in parallel. The name of a node that is shut down and qualifies as a historical node may be reused by a new node.

```
APPDYNAMICS_AGENT_REUSE_NODE_NAME=true
APPDYNAMICS_AGENT_REUSE_NODE_NAME_PREFIX=reportGen
```

**Agent Reuse Node Name Prefix**

When you configure the agent to reuse node names, use this property to specify the prefix the Controller uses to generate node names dynamically.

Environment Variable: **APPDYNAMICS_AGENT_REUSE_NODE_NAME_PREFIX**
Type: String
Default: None
Required: When `APPDYNAMICS_AGENT_REUSE_NODE_NAME=true`

Example: Using the following environmental variable specifications, the agent directs the Controller to generate a node name with the prefix `reportGen`. Node names will have suffixes `-1`, `-2`, and so on, depending on how many nodes are running in parallel.

```
APPDYNAMICS_AGENT_REUSE_NODE_NAME=true
APPDYNAMICS_AGENT_REUSE_NODE_NAME_PREFIX=reportGen
```

**Agent Tier Name**

Specifies the name of the tier that this .NET node belongs to. Note that this is not the deployment name on the application server.

Environment Variable: **APPDYNAMICS_AGENT_TIER_NAME**
Type: String
Default: None
Required: Yes

**Analytics SSL Enabled**

Specifies whether or not the .NET Agent for Linux sends the default transaction data to the Analytics Agent via SSL.

Environment Variable: **APPDYNAMICS_ANALYTICS_SSL_ENABLED**
Type: String
Default: False
Required: No
**Analytics Hostname**

Specifies the hostname or IP address of the analytics host.
Environment Variable: APPDYNAMICS_ANALYTICS_HOST_NAME
Type: String
Default: None
Required: No

**Analytics Port**

Specifies the port number of the analytics port.
Environment Variable: APPDYNAMICS_ANALYTICS_PORT
Type: Positive integer
Default: None
Required: No

**Controller Hostname**

Specifies the hostname or the IP address of the AppDynamics Controller. For an on-premises Controller, use the value for Application Server Host Name you provided when you installed the Controller. If you use the AppDynamics SaaS Controller, see the Welcome email from AppDynamics for the name of your Controller.
Environment Variable: APPDYNAMICS_CONTROLLER_HOST_NAME
Type: String
Default: None
Required: Yes, may also be set in the agent AppDynamicsConfig.json file or using the agent configuration utility.

**Controller Port Number**

Specifies the HTTP(S) port of the AppDynamics Controller. If the APPDYNAMICS.CONTROLLER.SSL.ENABLED environment variable is set to true, specify the HTTPS port of the Controller; otherwise, specify the HTTP port.
Environment Variable: APPDYNAMICS_CONTROLLER_PORT
Type: Positive Integer
Default: 8090
- For On-premises installations, the defaults are port 8090 for HTTP and port 8181 for HTTPS.
- For the SaaS Controller, use port 80 for HTTP or port 443 for HTTPS.
Required: Yes, may also be set in the agent AppDynamicsConfig.json file or using the agent configuration utility.

**Controller SSL Enabled**

When set to true enables encryption over SSL between the agent and the Controller.
Environment Variable: APPDYNAMICS_CONTROLLER_SSL_ENABLED
Type: Boolean
Default: false
Required: No, may also be set in the agent AppDynamicsConfig.json file or using the agent configuration utility.

**Controller SSL Certfile**

The path to the folder containing certificate files used to connect to the ssl-enabled Controller.
Environment Variable: APPDYNAMICS_CONTROLLER_SSL_CERTDIR
Type: String
Default: None
Required: No, may also be set in the agent AppDynamicsConfig.json file or using the agent configuration utility.

**HTTP Proxy Host**

The name of the proxy host.
Environment Variable: APPDYNAMICS_PROXY_HOST_NAME
Type: String
Default: None

Required: No, may also be set in the agent AppDynamicsConfig.json file or using the agent configuration utility.

**HTTP Proxy Port**

The port number of the proxy.

Environment Variable: APPDYNAMICS_PROXY_PORT
Type: Positive integer
Default: None

Required: No, may also be set in the agent AppDynamicsConfig.json file or using the agent configuration utility.

**HTTP Proxy User**

The name of the user that connects to the proxy.

Environment Variable: APPDYNAMICS_PROXY_AUTH_USER
Type: String
Default: None

Required: No, may also be set in the agent AppDynamicsConfig.json file or using the agent configuration utility.

**HTTP Proxy Password**

The password of the user that connects to the proxy.

Environment Variable: APPDYNAMICS_PROXY_AUTH_PASSWORD
Type: String
Default: None

Required: No, may also be set in the agent AppDynamicsConfig.json file or using the agent configuration utility.
Configure Analytics for .NET Agent for Linux

Related pages:
- Application Analytics
- .NET Agent for Linux Environment Variables

AppDynamics Application Analytics enables you to do real-time analysis of business performance that is correlated with the performance of your application software.

Configure Analytics Settings

If you are configuring the .NET Agent for Linux to send the default transaction data to the Analytics Agent, add a setting for the analytics host and port using either environment variables or in the agent json configuration file, config.json.

Configure Analytics Settings via JSON

The following json syntax shows the analytics configuration settings:

```json
"analytics: { "host": <analyticsHostName>, "port": <analyticsPort>, "ssl": <true || false> }
```

Configure Analytics Settings via Environment Variables

The following are the environment variables that you can use to configure Analytics:

- `APPDYNAMICS_ANALYTICS_SSL_ENABLED`
- `APPDYNAMICS_ANALYTICS_HOST_NAME`
- `APPDYNAMICS_ANALYTICS_PORT`

See .NET Agent for Linux Environment Variables
This topic walks you through the required configuration properties for the AppDynamics .NET Agent for Linux, which you can use to enable the agent to work without Docker. Before you can start instrumenting your applications with the .NET Agent, you need to get the current version of the .NET Core 2.0+. See .NET Agent Configuration Properties

**Configuration**

The agent configuration file is similar to the one used to configure .NET Core agent on Windows.

Once you have deployed the agent, you can customize the instrumentation of your application:

Edit the AppDynamicsConfig.json file with the following information:
The .NET Agent for Linux supports reuse node name configuration to help manage monitoring environments where there are many CLRs with short life spans. The CLR node names are reused to avoid a proliferation of differently named nodes in AppDynamics. Include the "reusenodename": "true" configuration, with "nodenameprefix": "prefix" to enable reuse node name without having to supply a specific node name.

You can also use environment variables to configure reuse node name, such as `ENV APPDYNAMICS_AGENT_REUSE_NODE_NAME=true` and `ENV APPDYNAMICS_AGENT_REUSE_NODE_NAME_PREFIX=<prefix>`.

See Agent Reuse Node Name for more information.

**Configuring using AppDynamicsConfig.json:** If you are configuring your agent using environment variables along with the json configuration file or just with the json configuration file, include the FULL_AGENT feature in the json file. Otherwise, the agent will not work. See the AppDynamicsConfig.json file example on .NET Agent for Linux Advanced Configuration Options.

```json
{
    "controller": {
        "host": "controller.saas.appdynamics.com",
        "port": 443,
        "account": "account-name",
        "password": "access-key",
        "ssl": true
    }
}
{
    "application": {
        "name": "My Application",
        "tier": "Sample tier",
        "node": "Instance1",
        "reusenodename": "true",
        "nodenameprefix": "prefix"
    },
    "analytics": {
        "host": "localhost",
        "port": 9090
    },
    "feature": [
        "FULL_AGENT"
    ]
}
```
.NET Agent for Linux does not support extended functionality like specifying instrumentors and whitelisting. Do not specify these if copying configuration from .NET Core agent on Windows.

See AppDynamicsConfig.json File

Other Configuration Options

It is possible to enable the agent to work without Docker. However, you will need to ensure that the OS is supported by the current agent.

1. Make the following files available to the application. They can be located next to the application files or in a separate folder:
   - AppDynamics.Agent.netstandard.dll
   - libappdprofiler.so
2. Create the following environment variables by adding them to your application:

<table>
<thead>
<tr>
<th>Environment Variable Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORECLR_PROFILER</td>
<td>(57e1aa68-2229-41aa-9931-a6e93bbc64d8)</td>
</tr>
<tr>
<td>CORECLR_ENABLE_PROFILING</td>
<td>1</td>
</tr>
<tr>
<td>CORECLR_PROFILER_PATH</td>
<td>Path to the libappdprofilerdynamic library. For example, &lt;application_folder&gt;/libappdprofiler.so</td>
</tr>
</tbody>
</table>

Env vars can be created on multiple levels, it is important that they are set in the context of the monitored application:
- Environment variables are inherited from global host level.
- Environment variables are parent process or service.
- Environment variables are set prior to the start of the application using the shell script or similar.

3. Place the agent configuration file next to the appropriate binaries.

   There are two supported options:

   **Global agent configuration file**
   The AppDynamicsConfig.json file can be placed next to the agent binaries.
   This way, it would be the default option for monitoring for each application configured to use the agent.

   The .NET Agent for Linux does not support overriding node name in the agent configuration. Therefore, monitoring multiple applications on the same host using global configuration is not supported. Only the first application will be reporting.

   Local agent configuration
   The [appname].AppDynamicsConfig.json file can be placed next to the application binaries, where [appname] should match your application DLL/EXE name.
   Using this option, you can attach AppDynamics configuration to the application package and deploy it as a bundle. It will be activated when the right environment variables are set and agent binaries are present.

   Local agent configuration has higher priority than global configuration.
Enable Preview Features

Preview features have been thoroughly tested inhouse at AppDynamics but because of the wide variety of possible use cases, AppDynamics releases these features for you to test and use at your discretion. We do not recommend that you use preview features in a production environment.

Enable Preview Features

To use features released under the preview feature flag, you must first enable them.

There are two ways to turn on the preview feature flag:

- Add the flag `PREVIEW_FEATURE` in the feature section of the `AppDynamicsConfig.json` file. For example:

  ```json
  "feature": [ "FULL_AGENT", "PREVIEW_FEATURE" ],
  ```

- Define the environment variable `APPDYNAMICS_PREVIEW_FEATURE_ENABLED` as `true`.

Enabling Preview Features enables both HTTP Custom Discovery Rules and ASP.NET Business Transaction Naming Configuration for the .NET Agent for Linux.
.NET Agent for Linux Troubleshooting

On this page:
- Agent Startup Situations
- Accessing Logs While Running Docker Container in the Background
- Default Agent Log Files Location

This topic describes general techniques when troubleshooting the AppDynamics .NET Agent for Linux deployment problems, along with a few specific scenarios you may encounter with workarounds.

Agent Startup Situations

The agent startup is dependent on three basic steps:

1. .NET Core loads the agent using profiler settings in the environment variables.
2. AppDynamics agent loads the configuration and additional library.
3. AppDynamics agent registers on the Controller using connection details found in the configuration file.

The sample startup console output on .NET Agent for Linux indicates a successful startup when going through these steps.

The following situations may occur:

<table>
<thead>
<tr>
<th>Symptom</th>
<th>Probable root causes</th>
</tr>
</thead>
</table>
| No console output from AppDynamics was written to the console | • Environment variables were not set correctly  
|                                                  | • Agent binaries do not match the OS version             |
| Agent complains about failing to load configuration file | AppDynamicsConfig.json or [appname].AppDynamicsConfig.json were not found in the corresponding folders. |
| Agent complains about HTTP errors                  | Agent failed to communicate to the Controller using connection settings provided in the configuration file. |

Accessing Logs While Running Docker Container in the Background

The default agent configuration redirects all agent output to the console. This is enabled for troubleshooting purposes.

If you are running the container in the background—d for detach—it is possible to access all output using the following Docker command:

```
docker logs <container id>
```

Default Agent Log Files Location

The agent log files location is at /tmp/appd/dotnet by default.

You can change this setting to redirect all logs to the console output for single view troubleshooting by adding the following configuration to the AppDynamicsConfig.json file as follows:

```
"log": [  
  {  
    "outputtype": "console"  
  }  
]
```
ASP.NET Business Transaction Configuration

Define Custom ASP.NET Business Transactions Using the Controller UI

You can create ASP.NET business transactions detection include and exclude rules with criteria that match HTTP Requests against:

- Method
- URI
- HTTP parameter
- Header
- Cookie

Transaction splitting is supported for the above HTTP request criteria.

Although the following HTTP request options appear in the dropdown list, they are not supported at this time by the .NET Agent for Linux. There is no effect if you configure them in the UI:

- Hostname
- Port
- Class name

You customize business transactions for the .NET Agent for Linux in a manner similar to other agents. Click Configure > Instrumentation > Add Rule and then choose to create either an include or exclude rule.

Example ASP.NET Business Transaction Detection Rule

The following screenshots show how to create a .NET Agent custom match rule that uses an ASP.NET entry point to exclude all requests whose URI equals /healthcheck.
In this screenshot of the Rule Summary, we specify that this is an *exclude* match rule. You must create an *include* rule before you can create an *exclude* rule to omit some of the transactions monitored by the include rule.
Here we specify the HTTP request criteria by choosing to use the URI attribute of the HTTP request to compare against. Valid criteria are Method, URI, HTTP parameter, Header, and Cookie. You can split the transactions based on these request match criteria.
The following screenshot specifies the HTTP request match criteria where the URI equals /healthcheck.
POCO Business Transaction Configuration

Define Custom POCO Business Transactions Using the Controller UI

Using the .NET Agent for Linux, you can create POCO business transaction rules that match synchronous or asynchronous transactions by:

- Method name
- Class name

Although the following Match Classes options appear in the dropdown list, they are not supported at this time. There is no effect if you configure them in the UI:

- that implements an interface which
- that extends a super class that
- that has an Annotation which

Also, the following are not yet supported:

- Transaction splitting using method/class data
- Background jobs
- Interface, base class or annotation POCO conditions

Example POCO Business Transaction Detection Rule

The following screenshots show how to create a .NET Agent custom match rule that uses a POCO entry point to include all requests whose class equals MyCompanyJobs.JobProcessor and the method name equals OnEvent.
In this screenshot of the Rule Summary, we specify that this is an *include* match rule. You must create an include rule before you can create an exclude rule to omit some of the transactions monitored by the include rule.
Here we configure the rule to match classes using the *with a Class Name that Match Classes* option, which is the only Match Classes option currently supported by the .NET Agent for Linux.
The following screenshot specifies the Match Class & Method match criteria to match classes with a class name that equals `MyCompanyJobs.JobProcessor` and the method name equals `OnEvent`. 
Transaction splitting is not currently supported for the .NET Agent for Linux.
Upgrade the .NET Agent for Linux

Before you begin, review the Release Notes for changes that affect your environment. AppDynamics requires an account access key for agent connections to single-tenant Controller accounts. Versions of AppDynamics before 4.1 only required an account access key for multi-tenant Controller accounts.

Single-tenant Controller customers can find the account access key in the Controller under Settings > License > Account. You must be a member of a role with the View License account level permission, see Roles and Permissions.

Binaries Overview

After downloading the agent binaries, you will need to extract them from the zip file into the desired folder.

The archive contains the following files:
- AppDynamics.Agent.netstandard.dll
- AppDynamicsConfig.json.template
- libappdprofiler.so
- README.md

Upgrade the .NET Agent for Linux
1. Download and unzip the .NET Agent for Linux installation package, AppDynamics-DotNetCore-linux-x64-4.5.x.x.zip to a temporary directory.

2. Replace the existing agent binaries with their latest versions. The files you need to replace are:
   - libappdprofiler.so
   - AppDynamics.Agent.netstandard.dll

3. Restart your application, or rebuild and then run your container image with the new versions of the agent binaries.

AppDynamics recommends that you include the new agent binaries into your Continuous Integration/Continuous Delivery (CI/CD) pipeline.
Node.js Agent

On this page:

- Node Naming for Node.js Agents
- Node.js Agent and the Java Proxy
- Instrument a Node.js Application Overview

Related pages:

- Dynamic Language Agent Proxy
- Monitor Node.js Processes

Search Node.js Agent topics:

This page introduces the Node.js Agent. With the Node.js agent and AppDynamics, you get automated transaction detection and correlation across your Node.js enabled environment.

Node Naming for Node.js Agents

Each instrumented Node.js process corresponds to a node in the AppDynamics model.

The nodes are named by combining the prefix that you specify for the nodeName in the requires statement that you add to the application source code when you install the Node.js agent with a hyphen and a digit.

For example, if you designate a prefix of MyNode for the nodes in the MyTier tier, the nodes in that tier are named MyNode-0, MyNode-1, MyNode-2 and so on.

Node.js Agent and the Java Proxy

The agent communicates with a Java proxy that handles the communication between the Node.js Agent and the AppDynamics Controller. The proxy reports the performance metrics to the Controller, where the data is stored, baselined, and analyzed. You can access this performance data interactively using the Browser UI or programmatically using the AppDynamics REST API.

The proxy component is automatically started when you start the Node.js Agent.
Instrument a Node.js Application Overview

The Getting Started Wizard walks you through configuration steps and helps you download the agent. To access it, in the Home page of the Controller click Getting Started and then Node.js.

The wizard provides the minimum information that the agent needs to communicate with the AppDynamics Controller: Controller host and port, optionally SSL, application name and tier name.
Node.js Supported Environments

On this page:
- Node.js Versions
- Operating Systems
- Transaction Naming
- HTTP Exit Points
- Database Exit Points
- Cache Exit points

Node.js Agent Support

Node.js Versions
- The 4.5.11 Node.js agent supports Node.js v0.8, v0.10, v0.12, v4, v5, v6, v7, v8, v9, v10, and v11.

For agent versions 4.5.12 and on the npm install will stop and print a message for the two scenarios listed:
- nodejs version less than 6
  - This version of AppDynamics agent supports Node.js versions 6.0 and above.
  - For older versions of Node.js use the the AppDynamics agent 4.5.11 by installing with 'npm install appdynamics@4.5.11'
- nodejs less than 8 on mac
  - This version of AppDynamics agent on Mac OS supports Node.js versions 8.0 and above.
  - For older versions of Node.js use the the AppDynamics agent 4.5.11 by installing with 'npm install appdynamics@4.5.11'

Operating Systems
- The agent is compatible with any Linux distribution based on glibc 2.5+
- Mac OS X 10.8+ (Agent / Node.js version specific compatibility as noted)
- Windows Server 2008R2+ and newer for 64 bit applications for Node.js versions 0.12.0 and higher (Agent / Node.js version specific compatibility as noted)

Transaction Naming

<table>
<thead>
<tr>
<th>Entry Type</th>
<th>Default Transaction Naming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node.js Web</td>
<td>URI</td>
</tr>
</tbody>
</table>

HTTP Exit Points

Supported HTTP Exit Points
- Node.js HTTP client library


Database Exit Points
- MongoDB
- MySQL
- PostgreSQL
- Riak - will be displayed as HTTP backends
- Couchbase
- DynamoDB using the official AWS SDK driver
• Cassandra

*Cache Exit points*

• Memcached
• Redis
Install the Node.js Agent

**On this page:**
- Prepare to Install the Node.js Agent
- About Node.js Agent Node Identity
- Running the Machine Agent on a Node.js Node
- Installing the Node.js Agent
- Stopping the Node.js Application
- Running with other profiling tools
- Instrumenting a Node.js Cluster
- Set User Agent for Node.js 0.8.1 through 0.8.18
- Resolving Installation Issues for Node.js

**Related pages:**
- Install the Node.js Agent on Windows

The Agent Download Wizard in the Controller provides the easiest way to get started quickly with the AppDynamics Node.js Agent. The wizard constructs a pre-configured require statement for the agent based on your input.

This page provides additional information and options for installing the Node.js Agent.

**Prepare to Install the Node.js Agent**

You must add a require statement to the source code of your Node.js application to install the Node.js agent. Therefore, in addition to a running AppDynamics Controller, you need to have write access to the application source code and the ability to restart the Node.js application to install the agent.

The Getting Started wizard in the AppDynamics controller can generate the require statement for you. It populates the statement with the connection settings for the controller and the values you provide to the wizard for modeling the Node.js application in AppDynamics.

**For Installations on Windows:**

- Verify support for your version of Windows.
- Install the Visual C++ Redistributable for Visual Studio 2015.

**About Node.js Agent Node Identity**

You configure the identify of a node.js application instance using the node name setting. By default, the agent uses the value configured for `nodeName` as a prefix and adds a dash and number as a suffix. For example, given the following setting:

```text
nodeName=MyNode
```

The first node to start with this configuration would be named `MyNode-0`. Auto-numbered suffixes on node names are most useful for machines with more than one worker process.

If you are not running multiple worker processes, you can prevent auto-numbering in node names by setting `noNodeNameSuffix` to `true`. In the following example, the node would be named `MyNode`.

```text
nodeName=MyNode
noNodeNameSuffix=true
```

If you are instrumenting worker processes on different machines, keep in mind that each application and node name combination must be unique. Therefore, be sure to specify different node name prefixes for each server, for example, by configuring `nodeName=Server1` for the first server and `nodeName=Server2` for the second server.
If you are instrumenting a Node.js application that uses the PM2 process manager, set the node suffix name to process.env.pm_id.

**Running the Machine Agent on a Node.js Node**

If you install the Standalone Machine Agent on the machine hosting the instrumented Node.js node and you specify the tier and node name in the controller-info.xml file of the machine agent, the Node.js Agent will fail to register.

To avoid this problem:

- Install the Node.js Agent before you install the Standalone Machine Agent.
- If you install the machine agent on the machine hosting the instrumented Node.js node, do not specify the application, tier or node name in the machine agent's controller-info.xml file. If you do, the Node.js Agent may fail to register.

**Installing the Node.js Agent**

To install the agent, run the npm installation command in each directory you want to monitor and then add a require command to add the agent module to your application.

If you are using Node.js 0.8.1 through 0.8.18, see [Set User Agent for Node.js 0.8.1 through 0.8.18](#).

**Installing via npm**

Refer to [Agent and Controller Compatibility](#) to determine which versions of the AppDynamics Node.js agent are compatible with your controller.

To install the latest 4.5 agent, run the following command:

```
npm install appdynamics@next
```

If you know which specific version of the Node.js agent you want to install, you can specify it:

```
npm install appdynamics@<x.y.z>
```

If you are using npm 5, you must disable the lock file by setting `package-lock=false` in the npm configuration settings.

**Installing Node.js with Java proxy**

To install Node.js with Java proxy run the following command at the time of agent install:

```
npm install appdynamics@<x.y.z> --appd_include_java_proxy=true
```

If the `--appd_include_java_proxy` npm configuration option is not provided, the npm install command installs only the libagent dependencies.

**Add the Require Statement**

Paste the following require statement as the very first line of your application source code, before any other require statement. Replace the variables with the values for your setup. To find your account name and access key, click the settings (🔍) icon in the upper right corner of the AppDynamics UI, and then click License.
require("appdynamics").profile({
    controllerHostName: '<controller host name>',
    controllerPort: <controller port number>,
    controllerSslEnabled: false, // Set to true if controllerPort is SSL
    accountName: '<AppDynamics_account_name>',
    accountAccessKey: '<AppDynamics_account_key>', //required
    applicationName: 'your_app_name',
    tierName: 'choose_a_tier_name',
    nodeName: 'choose_a_node_name'
});

For reference information on the settings, along with other settings you can use, see Node.js Settings Reference.

You can place the require statement as the first line in the require statement of another module that appears as the first line of code.

In this case, you would need to modify your point-of-entry source file; this can be just a single line to the require() the file that you place the call to the agent into; for example, require("<script-that-initializes-the-agent>"). You could also parameterize the profile() call to name different instances without having to have multiple versions of the agent initialization script.

If it is not possible to place the require statement as the first line of code, you can insert the statement elsewhere, but it must occur before the require() of any core or third party module that needs to be instrumented. In general, the require("appdynamics") statement should occur as early as possible in the code.

**Test the Configuration**

To verify the installation, restart the application and put load on it. The new node should appear in the flow map for the business application you specified in the configuration.

**Stopping the Node.js Application**

The script or other mechanism used to stop an instrumented Node.js instance should use the SIGTERM signal if possible.

Using SIGKILL (kill -9) will prevent resources for the agent's Java proxy to be properly released.

**Running with other profiling tools**

The agent is incompatible with other profiling tools, such as running the node process with the --inspect flag.

**Instrumenting a Node.js Cluster**

If your application uses the cluster module, place the appdynamics.profile require statement in both the master and worker processes. This scenario does not require a manual launch of the proxy.

If your application uses an external process manager, such as PM2, you need to launch the proxy component manually. See Sharing a Proxy Among Node.js Agents

**Set User Agent for Node.js 0.8.1 through 0.8.18**

If running a Node.js version between 0.8.1 and 0.8.18, inclusive, you need to set the user-agent npm property before you run the npm install command.

The syntax is:
For example:

```
npm config set user-agent "node/v0.8.14 linux x64"
```

**Resolving Installation Issues for Node.js**

If you are trying to install the Node.js Agent on a version of Node.js that the agent does not support, the installation will fail and you will see a message similar to the following:

```
npm ERR! notsup Unsupported
npm ERR! notsup Not compatible with your version of node/npm: appdynamics@3.9.3
npm ERR! notsup Required: {"node":">=0.8.0 <=0.10.31"}
npm ERR! notsup Actual: {"npm":"1.4.28","node":"0.10.32"}
...
```

See Node.js Versions in [Node.js Supported Environments](#)
This topic describes the settings that are available for configuring the Node.js Agent. To apply the settings, add them to the require statement for the Node.js Agent in the monitored application.

You can use environment variables for some of the settings, as described below. See Install the Node.js Agent for more information.

**General Settings**

This is the complete list of settings in the require statement that you insert into your application code. Not all these settings are required.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>controllerHostName</td>
<td>The IP address or hostname of your controller. SaaS customers receive this URL in the customers set them when they install the controller.</td>
</tr>
<tr>
<td>controllerPort</td>
<td>The port to which the agent connects to the Controller. It is 8090 for an on-premises Controller, and 443 for a SaaS Controller.</td>
</tr>
<tr>
<td>controllerSslEnabled</td>
<td>Set this property to true if connecting to the controller via SSL.</td>
</tr>
<tr>
<td>accountName</td>
<td>The account name on the Controller to which this agent will report. See Account Name.</td>
</tr>
<tr>
<td>accountAccessKey</td>
<td>Account access key on the controller.</td>
</tr>
<tr>
<td>applicationName</td>
<td>Name that represents the entire application in the AppDynamics console.</td>
</tr>
<tr>
<td>tierName</td>
<td>Name that represents your Node.js app or service in the flow maps</td>
</tr>
<tr>
<td>nodeName</td>
<td>Name of the Node.js process to be monitored by this agent. See Node Names</td>
</tr>
<tr>
<td>noNodeNameSuffix</td>
<td>Optional, set to true if you do not want the agent to add a suffix, such as -0, -1, -2, and so on</td>
</tr>
<tr>
<td>proxyHost, proxyPort</td>
<td>Set these options to route data to the controller through a proxy server. The proxyHost is the hostname or IP address of the proxy server. The proxyPort is the proxy server's HTTP or HTTPS port, whichever you are using. If you set the host you must set the port as well.</td>
</tr>
<tr>
<td>proxyUser, proxyPasswordFile</td>
<td>Configure the proxy username and password file if the proxy server requires credentials.</td>
</tr>
<tr>
<td>proxy</td>
<td>Set this property to true if you want to use the Java proxy version of the Node.js agent.</td>
</tr>
<tr>
<td>alwaysAddEumMetadataInHttpHeaders</td>
<td>Set this property to true to have the Node.js Agent write business transaction metadata to the XHR header and in a cookie even if the request is considered cross-origin. See the description for the equivalent setting in alwaysAddEumMetadataInHttpHeaders for more information.</td>
</tr>
<tr>
<td>btEntryPointDelayDisabled</td>
<td>true/false - Optional, defaults to false. Setting this property to true can accelerate the start-up of business transactions, but it can adversely affect the drill-down in distributed transactions.</td>
</tr>
<tr>
<td>debug</td>
<td>Set this property to true to enable debug level logging for the agent. The default is false.</td>
</tr>
<tr>
<td>logging</td>
<td>The location, level, and other settings related to agent logging. See Node.js Agent Log</td>
</tr>
<tr>
<td>maxProcessSnapshotsPerPeriod</td>
<td>Optional. The default is 2. The number of automatic process snapshots allowed in processSnapshotCountResetPeriodSeconds seconds.</td>
</tr>
<tr>
<td>processSnapshotCountResetPeriodSeconds</td>
<td>Optional, defaults to 60. Frequency, in seconds, at which the automatic process snapshot is triggered.</td>
</tr>
<tr>
<td>autoSnapshotDurationSeconds</td>
<td>Optional, defaults to 10. Length, in seconds, of automatically-triggered process snapshots.</td>
</tr>
</tbody>
</table>
proxyAutolaunchDisabled | true/false - Optional, defaults to false. Set this property to true if you need to manually launch the Java Proxy for Node.js Agents.

proxyCtrlDir | Directory path for the directory containing the domain control socket, which the agent uses to start an AppDynamics node. Set manually if you are setting up a multi-tenant proxy. See Sharing a Proxy.
ootTmpDir | Directory path for the root of the directory that stores the agent files. Optional, defaults to /tmp/appds.

tmpDir | Directory path for the subdirectory of the root directory for the monitored node. Optional, defaults to a hash of the Controller info for the instrumented node.

reuseNode | Set this property to true to enable reusing node names. This property is useful for monitoring environments where there are many nodes with short life spans. The default is false.

reuseNodePrefix | Set this property to the string that prefixes the node name when reuseNode is set to true. The Controller uses this prefix to generate node names dynamically.

certificateFile | Directory path for the directory containing the SSL certificate.

uniqueHostId | The value of the unique host ID. This setting is only available for the proxyless version of the Node.js agent.

**Analytics Settings**

If you are configuring the Node.js Agent to send the default transaction data to the Analytics Agent, add a setting for the analytics host and port using the following format:

```json
analytics: {
    host: <analyticsHostName>,
    port: <analyticsPort>,
    SSL: <true || false>
}
```

The Analytics Agent may report on the same or different host and port numbers from the Node.js agent. In either case, you need to specify it here.

For more information see Collect Transaction Analytics Data.

**Windows Settings**

On Windows, the Node.js Agent requires three additional ports. See Install the Node.js Agent on Windows for more information.

- proxyCommPort: **default is** 10101
- proxyRequestPort: **default is** 10102
- proxyReportingPort: **default is** 10103

**Filter Sensitive Data**

If your application contains sensitive data that should not be displayed in the Controller, you can apply the following data filters to the `require` statement for the Node.js Agent:
// Add a sensitive data filter
dataFilters: [{
  "appliesTo": "http-headers",
  "matchPattern": "host"
}],

// Add an environment variable filter
dataFilters: [{
  "appliesTo": "env-vars",
  "matchPattern": "password|key"
}],

// Add a sensitive URL filter
urlFilters: [{
  "delimiter": "/",
  "segment": "1",
  "matchPattern": "a",
  "paramPattern": "bar"
}],

// Add a sensitive message filter
messageFilters: [{
  "messageType": "throwable",
  "matchPattern": "Error.*proxy.*",
  "redactionRegex": "proxy"
}]

The matchpattern/parampattern is a standard regex.

**Environment Variables**

If you have not provided the required settings in the require statement, the agent uses the values of the following environment variables if those variables are set. If both are set, the require statement value takes precedence over the environment variable.

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Maps to</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPDYNAMICS_CONTROLLER_HOST_NAME</td>
<td>controllerHostName</td>
</tr>
<tr>
<td>APPDYNAMICS_CONTROLLER_PORT</td>
<td>controllerPort</td>
</tr>
<tr>
<td>APPDYNAMICS_CONTROLLER_SSL_ENABLED</td>
<td>controllerSslEnabled</td>
</tr>
<tr>
<td>APPDYNAMICS_AGENT_ACCOUNT_NAME</td>
<td>accountName</td>
</tr>
<tr>
<td>APPDYNAMICS_AGENT_ACCOUNT_ACCESS_KEY</td>
<td>accountAccessKey</td>
</tr>
<tr>
<td>APPDYNAMICS_AGENT_APPLICATION_NAME</td>
<td>applicationName</td>
</tr>
</tbody>
</table>
You can set the unique host ID to any value that you want to use to identify the host. If you do not specify a value, the agent will generate the host ID for you.

This environment variable is only supported in the proxyless version of the Node.js agent.
Install the Node.js Agent on Windows

On this page:

- Proxy Ports
- Auto-Launch Windows Environments
- Manual-Launch Windows Environments
- Execute runProxy Manually on Windows

Related pages:

- Install the Node.js Agent
- Start the Proxy Manually for Node.js
- Sharing a Proxy Among Node.js Agents
- Dynamic Language Agent Proxy

You can use the instructions in Install the Node.js Agent to install the Node.js Agent on Node.js applications running on Windows. However, there are a few additional considerations that apply to Windows environments, as described here.

Proxy Ports

Depending on your Node.js environment on Windows, you may need to configure three ports in the Node.js require statement. These port settings are:

- `proxyCommPort`: Default is 10101
- `proxyRequestPort`: Default is 10102
- `proxyReportingPort`: Default is 10103

All three ports must be in the range `1024-65535` inclusive.

Auto-Launch Windows Environments

The default behavior is for the Agent to auto-launch the proxy, where `proxyAutolaunchDisabled` is false.

If you allow the Agent to auto-launch, and you have one Node.js process per machine, the Agent will automatically use the default port values if they are available. In this case, you do not have to configure these settings. If the default ports are not available, in the `require` statement, you must configure these settings to ports that are available.

If you have multiple Node.js processes on a single machine each reporting to its own proxy, each process needs its own set of ports. In this case, in the `require` statement for each process, configure unique settings for all three ports.

Manual-Launch Windows Environments

Some environments have multiple Node.js processes on the same machine, all reporting to a single proxy. In these environments, it is necessary to launch the proxy manually, where `proxyAutolaunchDisabled` is true and to configure the three port settings.

The `proxyCommPort` setting in the `require` statement and the `commPort` option to the `runProxy.cmd` must be the same value. The agent and the proxy must use the same port.

The `proxyRequestPort` and the `proxyReportingPort` are two unused TCP ports that the user running the proxy has permission to bind to on Windows.

Each distinct app-tier-node combination on a single machine must use different request and reporting ports. Two processes reporting as the same app-tier-node may use the same request and reporting ports. Set these ports in the `require` statement for each app-tier-node combination.

Execute runProxy Manually on Windows

By default, the proxy component is automatically started when you start the agent. Use the command described here only if you have configured the agent for manual proxy launch.

Before any traffic is run on the instrumented server, execute `runProxy.cmd` to start the proxy using the following syntax. If you use the optional `-d` option, it must precede the `-j` option.
Usage: runProxy.cmd [-d <proxyDir>] -j <jreDir> [logDir] [commPort]

Options:
   -d <proxyDir> the directory where the proxy is installed; optional, defaults to the directory containing the runProxy.cmd script
   -j <jreDir> the directory where the JRE is installed; required
   logDir the directory the proxy should log to; optional, defaults to <proxyDir>\logs
   [commPort] port used for the proxy control channel on Windows; defaults to 10101; if specified it must match the proxyCommPort setting in the require statement

Example

This example assumes that the command is run from the root of the Node.js application instrumented by the agent.

```cmd
.
ode_modules\appdynamics\node_modules\appdynamics-proxy\proxy\runProxy.cmd
-j .\node_modules\appdynamics\node_modules\appdynamics-jre\jre
/var/log/proxy
.node_modules\appdynamics\node_modules\appdynamics-proxy\proxy\logs
10101
```

Execute the runProxy Command Manually on Windows Task Scheduler

You can schedule the runProxy command in the Windows Task Scheduler.

1. Open Task Scheduler.
2. Click Create Basic Task.
3. In the General tab, enter a name for your task.
4. Select Run whether user is logged on or not.
5. Check Run with highest privileges.
6. In the Triggers tab, add a new trigger. Set the trigger to when the computer starts.
7. In the Actions tab, create a new action. Set the action to Start a program.
8. In the Program/script field, enter the following:
   ```cmd
   <app_root>/node_modules/appdynamics-proxy/proxy/runProxy.cmd
   -j <app_root>/node_modules/appdynamics-jre/jre
   %TEMP% 10101
   ```
9. In the Add arguments field, enter the following:
   ```cmd
   -d <app_root>/node_modules/appdynamics-proxy -j <app_root>/node_modules/appdynamics-jre/jre
   %TEMP% 10101
   ```
10. In the Start in field, enter the following:
    ```cmd
    <app_root>/node_modules/appdynamics-proxy/proxy
    ```
11. Click OK to finish the wizard.
12. Reboot the system, and check the Task Manager for a Java process.
Node.js Agent Logging

On this page:
- Node.js Agent Logging Location
- Configure Node.js Agent Logging
- Proxy Logging
- Debug Logging

Related pages:
- Proxy Logging
- Dynamic Language Agent Proxy

By default, the Node.js Agent generates log information at the informational level. You can increase or decrease the logging verbosity, change the log file location, and configure other logging settings.

Node.js Agent Logging Location

The log file appears at the following location on the machine where the Node.js Agent runs:

```
/tmp/appd/<hash>/appd_node_agent.log
```

Where `<hash>` is the portion of the path as used for the Java proxy.

On Windows, you can find the tmp directory with:
```
cd %tmp%
```
and then look for the `<hash>` directory.

Configure Node.js Agent Logging

To configure logging, use the logging module configuration in the Node.js require statement.

The following sample shows the logging settings when you are in libagent mode:

```javascript
{
  ...

  logging: {
    'logfiles': [
      {
        'root_directory': '/tmp/appd',
        'filename': 'echo_%N.log',
        'level': 'TRACE',
        'max_size': 5242880,
        'max_files': 10,
        'outputType': 'console' // Set this parameter if you want to log to STDOUT/STDERR. Omit this parameter if you want to log to a file.
      }
    ]
  }

  ...
}
```

The following sample shows the logging settings when you are in proxy mode:
As illustrated, you can change the level, maximum size or number of files, and mode.

The level determines the verbosity of logging output. It can have one these values, as listed from least to most verbose:

- FATAL
- ERROR
- WARN
- INFO
- DEBUG
- TRACE

By default, once the size of the logging files on disk reaches the maximum size, old logs are purged. You can reduce or increase the size, depending on your needs and the disk space available on the machine.

**Proxy Logging**

The proxy logs the transactions that are accepted from the agent and sent to the Controller. The proxy generates logs independently of the Node.js Agent logging level or activity.

When the Node.js agent launches the proxy, it prints the directory path at which the proxy is logging to its own log. By default, the proxy log is in /tmp/appd/logs.

See Dynamic Agent Proxy Logging

**Debug Logging**

Instead of using the logging configuration shown above, you can use the shortcut of setting DEBUG to true, as follows:

debug : true,

This results in debug level logging, with the location of the debug log file written to standard output.

See Node.js Agent Settings
Sharing a Proxy Among Node.js Agents

This topic describes how to configure multiple Node.js Agent on a machine to share a single, multi-tenant Java proxy.

**About Proxy Sharing**

Each Node.js agent on a machine normally launches its own proxy instance, having created the control directory for its proxy based on a unique combination of application, tier, and node name for the agent. The control directory contains the configuration for the agent as well as the domain control socket, which the agent uses to start an AppDynamics node.

If running multiple Node.js agents on a single machine, you can configure them to use a shared Java proxy. A shared proxy (also called multi-tenant proxy) reduces the overhead of additional proxies.

A shared proxy is required configuration if you use an external process manager, such as PM2.

**Configure a Shared Proxy**

1. Stop the Node.js application.
2. Create directories for the agent and the agent logs and for the proxy control directory. The permissions on this directory must be readable and executable by the application user that the application and writable by the proxy user. For example,

   ```
   mkdir /tmp/appd /tmp/appd/logs /tmp/appd/proxy_ctrl_dir
   ```

   You may want to devise a technique for creating these directories automatically whenever you upgrade the Node.js agent.
3. Disable automatic proxy launching for each agent. In the AppDynamics `require.profile()` block, set `proxyAutolaunchDisabled` to true and set the `proxyCtrlDir` to the directory that you created. For example,

   ```
   ...
   proxyAutolaunchDisabled: true,
   proxyCtrlDir: '/tmp/appd/proxy_ctrl_dir',
   ...
   ```

4. Ensure that each agent that reports to the multi-tenant proxy is configured with a unique node name.
5. Start the shared Java proxy before you start the agents, passing the agent control directory as the `proxyCommunicationDir` argument to the `runproxy` script. See Run the Node.js Proxy Daemon Manually. AppDynamics recommends that you configure the proxy to start on system startup.
6. Restart the Node.js application. In a few minutes, the nodes should appear in the Controller UI flow maps.

**Proxy Sharing Limits**

In its default configuration, a single proxy can handle up to ten agents. If you need to run more, you will likely need to adjust the `maxHeapSize` and `maxPermSize` settings in the `runproxy` script.

When adding agents, use debug mode and monitor the `proxy.out` file, which indicates whether the proxy is running out of heap. The file is visible when running in debug mode.
Start the Proxy Manually for Node.js

When the Node.js Agent starts up, it automatically executes the `runproxy` shell script, which starts the Java proxy.

This arrangement is suitable for most situations. However, there are cases when you may want to suppress automatic proxy startup. For example, if you are running a Node.js cluster on a machine and you want to reduce overhead of multiple proxies or if you are using an external process manager.

The following shows an example of the configuration:

```
nohup
  /<application_root_dir>/node_modules/appdynamics-proxy/proxy/runProxy -j /<application_root_dir>/node_modules/appdynamics-jre/jre --
  /tmp/appd/proxy_ctrl_dir /tmp/appd/logs &
```

You need to launch the proxy manually as the Node.js process owner.

For the full syntax of the `runProxy` script, see the description of the command format for PHP applications in Start the PHP Agent Proxy Manually.

For information about running the proxy manually for the Node.js Agent on Windows, see Install the Node.js Agent on Windows.
Upgrade the Node.js Agent

If you are upgrading both the Controller and agents, first upgrade the Controller and then upgrade the agents.

Also, if you are upgrading multiple agents in your monitored environment, upgrade the agents for the tiers on which business transactions originate last. For more information about this requirement, along with Controller and agent compatibility information, see Agent and Controller Compatibility.

To upgrade the Node.js agent:

1. Stop the Node.js application.
2. From the root directory of the application for which you want to upgrade the agent, uninstall the old version of the agent by entering:

   ```
   npm uninstall appdynamics
   ```

3. From the same directory install the new version by entering:

   ```
   npm install appdynamics@<version>
   ```

   The version is the three-digit version number of the new version that you are installing; for example, 4.1.1.
4. Restart the Node.js application.
Uninstall the Node.js Agent

To uninstall the Node.js Agent:

1. From the application root directory of the application from which you want to uninstall the agent enter:

   ```bash
   npm uninstall appdynamics
   ```

2. Remove the "require("appdynamics")" statement from your Node.js applications.
3. Restart the Node.js application.
The AppDynamics Node.js Agent APIs enable the Node.js Agent to monitor business transactions that a web application performs outside the context of an HTTP request or an application that is not web-based.

**Business Transaction Definition**

You define a custom business transaction using `appd.startTransaction()` and `end()`.

If the custom transaction runs for a long time or if it crosses an asynchronous boundary, you can use `resume()` to rejoin the running transaction.

A downstream transaction in a distributed transaction cannot start until its upstream transaction has made its exit call.

The order of transactions respects the nonlinear nature of Node.js applications. An upstream transaction might end immediately after its exit call completes or sometime later. It is not necessary for an upstream transaction to end before or after its downstream transaction starts or ends. It is possible for a downstream transaction to end before its upstream transactions have ended.

**Marking a Transaction as an Error Transaction**

An error transaction is one in which an exception occurred. It is reported as an error transaction and not as a slow, very slow or stalled transaction, even if the transaction in which the error occurred was also slow or stalled.

To attach a JavaScript error object to a transaction using the Node.js APIs do one of the following:

- Use `trx.markError()` passing in the JavaScript error object.
- Attach the error object to the HTTP response associated with the transaction as `response.error`.

When an error is marked, the status code defaults to 500, unless you set it using one of the following techniques:

- Use `trx.markError()` passing in the JavaScript error object and the optional status code.
- Set the `statusCode` property in the error object passed to `trx.markError()` as `error.StatusCode`.
- Set the `statusCode` property on the `response.error` object as `response.error.StatusCode`.

**Transaction Correlation Management**

Transaction correlation is the functionality that maintains the business transaction context across all tiers—servers—traversed by a distributed transaction. The tiers may be built on platforms other than Node.js. For example, Java, .NET, and PHP tiers may participate in a distributed business transaction in which a Node.js tier participates, as originating, continuing or terminating tiers in the transaction.

The Node.js Agent APIs provide facilities for managing transaction correlation among various transactions, which may be custom or automatically-detected.
While the AppDynamics default auto-detection mechanism obtains transaction correlation information from an incoming HTTP request, the Node.js Agent APIs allow you to obtain this information from other sources, such as a custom field in a Redis cache entry, and to supply it to your custom transactions. You can use the APIs to support forward correlation by supplying transaction correlation information on exit calls that you make to downstream transactions.

A downstream transaction starts after its immediate upstream transaction makes its exit call. That call could be a direct exit call to the next tier or a call to a backend service that does some processing, such as publishing a message onto a queue.

Exit Call Management

An exit call is either a custom exit call or an automatically-detected exit call. A transaction needs its exit call object to create correlation information to provide to a downstream transaction that needs to correlate with it.

**Custom Exit Calls**

If you want an exit call to be reported to the controller, and it is not automatically detected by the agent, create a custom exit call with `startExitCall()` and end it with `endExitCall()`.

See [Node.js Supported Environments](#) for the list of backends that are automatically detected by the Node.js Agent. If the exit call invokes a backend not listed on this page, you probably need to create a custom exit call.

You can also use a `beforeExitCall()` callback function to check for the existence of an automatically-detected exit call.

Processing a custom exit call is straightforward. First, create the exit call and then pass its returned `ExitCall` object to `createCorrelationInfo()` to create a string-encoded correlation information object that a downstream transaction can fetch later.
Workflow for a custom exit call

```javascript
// start a custom transaction
var trx = appd.startTransaction(. . .)

// start a custom exit call
exit = trx.startExitCall(. . .);

// create correlation information
cinfo = trx.createCorrelationInfo(exit)
//store cinfo where it can be made available to a downstream transaction

// make the exit call
.trx.endExitCall(. . .)
```

**Automatically Detected Exit Calls**

The code executing in your custom transaction might make an exit call that is automatically detected by the Node.js Agent.

In this case, you can provide a `beforeExitCall()` callback function to process that exit call. When your code makes an automatically-detected exit call, the agent intercepts the exit call request and invokes the callback, if you have provided one. Then it allows the application to proceed with the exit call request.

You would install a `beforeExitCall()` callback function for the following purposes:

**To test for the existence of an automatically-detected exit call**

If you are not uncertain if the exit call will be automatically detected, you can install a callback function to test for the exit call. If the callback is not invoked, the exit call is not automatically detected.
Test for existence of automatically-detected exit call

```javascript
// start a custom transaction
var trx = appd.startTransaction(. . . )

var detected = false;
trx.beforeExitCall = function(call) { detected = true; return call; }

// make the exit call

// test whether the exit call was automatically detected
if (detected == false)

If detected is still false after the exit call, you need to create a custom exit call if you want the exit call to be reported to the controller. If detected is true, you can capture the exit call and create the correlation information using the callback as shown in the next example.

To capture the ExitCall object to create correlation information

```javascript
// start a custom transaction
var trx = appd.startTransaction(. . . )

trx.beforeExitCall = function getExitInfo(exitCall) {
  // create the correlation header from the captured exit call
  var c = trx.createCorrelationInfo(exitCall);
  // store c somewhere
  // ... (c)
  return exitCall;
}

// make the exit call
```

To modify the exit call that is reported to the Controller
Modfy exit call reported to controller

```javascript
// start a custom transaction
var trx = appd.startTransaction( . . . )

trx.beforeExitCall = function customExitCallHandler(exitCall) {
    // don't report database access for this transaction
    if (exitCall.isSql) return;
    // customize label for all other exit calls in this transaction
    exitCall.label += " (my custom transaction)";
    return exitCall;
}

// make the exit call
```

Workflow: Create Originating Transaction with Correlation Information

In this scenario, you create a custom originating transaction and the correlation header that a downstream transaction will need to correlate with it.

1. Start a custom transaction: var trx=appd.startTransaction(...).
2. Either start a custom exit call using e=trx.startExitCall(einfo) or capture a detected exit call by installing a callback function to e=trx.beforeExitCall(e).
3. Create correlation information from the exit call: c=trx.createCorrelationInfo(e).
4. Store the correlation information.
5. Make the exit call.
6. Optionally provide a trx.exitCallCompleted(e) callback function, if you want to modify what is reported to the controller based on information obtained from the execution of the exit call.
7. If you created a custom exit call, end the exit call after you have processed the exit call response: trx.endExitCall(e).
8. End the transaction: trx.end().

Workflow: Create Terminating Transaction Correlated with Upstream Transaction

In this scenario, you create a custom transaction, which needs to correlate with an upstream transaction but not with another downstream transaction.

1. Retrieve the correlation information stored by the previous transaction.
2. Parse the retrieved correlation information into a correlation header: ch=appd.parseCorrelationInfo(c).
3. Start a custom transaction using the correlation header: trx=appd.startTransaction(ch).
4. Optionally make an exit call.
5. End the transaction: trx.end().

Workflow: Create Continuing Transaction Correlated with Upstream and Downstream Transactions

In this scenario, you create a continuing transaction that needs to correlate both with an upstream originating transaction and a downstream transaction. It uses the correlation techniques described in both of the previous workflows.

1. Retrieve correlation information stored by the previous transaction.
2. Parse the retrieved correlation information into a correlation header: ch=appd.parseCorrelationInfo(c1).
3. Start a custom transaction using the correlation header: trx=appd.startTransaction(ch).
4. Either start a custom exit call using e=trx.startExitCall(einfo), or capture a detected exit call by installing a callback function to e=trx.beforeExitCall(e).
5. Create correlation information from the exit call in this transaction: `c2=trx.createCorrelationInfo(e)`.
6. Store the correlation information for the next downstream transaction.
7. Make the exit call.
8. Optionally provide a `trx.exitCallCompleted(e)` callback function, if you want to modify what is reported to the controller based on information obtained from the execution of the exit call.
9. If you created a custom exit call, end the exit call after you have processed the exit call response: `trx.endExitCall(e)`.
10. End the transaction: `trx.end()`.
Node.js Agent API Reference

On this page:
- API Call List
- Loading the AppDynamics Module
- Basic Business Transaction Management
- Exit Call Management
- Transaction Correlation Management
- Exit Call Properties
- Backend Identifying Properties

Related pages:
- Node.js Agent API User Guide
- Backend Detection Rules

The Node.js Agent APIs includes methods for managing business transactions and exit points in Node.js applications. This topic describes the Node.js Agent API.

API Call List

- appd.startTransaction(transactionInfo)
- TimePromise.prototype.resume()
- TimePromise.prototype.end(error)
- appd.getTransaction(request)
- appd.addAnalyticsData (key, value)
- appd.addSnapshotData (key, value)
- appd.markError(error, [statusCode])
- txn.markError(error, [statusCode])
- txn.addSnapshotData(key, value)
- txn.onSnapshotCaptured(txn)
- txn.addAnalyticsData(key, value)
- txn.onResponseComplete(req, res)
- TimePromise.prototype.startExitCall(exitCallInfo)
- TimePromise.prototype.endExitCall(exitCall, error)
- TimePromise.prototype.beforeExitCall(exitCall)
- TimePromise.prototype.exitCallCompleted(exitCall)
- appd.parseCorrelationInfo(source)
- TimePromise.prototype.createCorrelationInfo(exitCall, doNotResolve)

Loading the AppDynamics Module

You can use the APIs in your application by adding the "appdynamics" Node.js module in a require statement, after appd.profile().

```javascript
var appd = require('appdynamics');
appd.profile({ . . . })
...

var transaction = appd.startTransaction(...);
...

transaction.end();
```
Basic Business Transaction Management

`appd.startTransaction(transactionInfo)`

- `transactionInfo` string, `HttpRequest` or `CorrelationHeader`
  
  The `transactionInfo` parameter can be one of the following:
  
  - A String specifying the name of the custom transaction. The following characters cannot be used in transaction names: `{ } [] | & ;
  - An `HttpRequest` object. Normal transaction matching and naming rules are applied, and any included RUM and/or transaction correlation information is picked up automatically.
  - A `CorrelationHeader` object providing information to correlate with a transaction that is upstream to this one. See `parseCorrelationInfo()`.

Creates and starts a custom business transaction and returns a transaction handle as a `TimePromise` instance, which is used for subsequent operations on this transaction.

`TimePromise.prototype.resume()`

Re-joins a long-running or asynchronous transaction.

Call `resume()`:

- When a long-running transaction needs to be kept alive beyond the default transaction timeout of five minutes. This prevents the agent from assuming the transaction has failed and then deleting the transaction.
- When the code being instrumented crosses an async boundary. This re-associates the current execution context with the correct transaction and ensures any exit calls performed are charged to the right transaction.

`TimePromise.prototype.end(error)`

- `error`: any; optional

Ends the transaction and reports transaction information to the controller.

If an `error` is passed, the error is associated with the transaction and the transaction is flagged as an error transaction.

`appd.getTransaction(request)`

- `request`: HTTP request object

Returns a handle to the business transaction associated with the specified request, if one exists. If no associated business transaction exists, returns false.

Use `appd.getTransaction()` to access both custom and automatically detected business transactions.

`appd.addAnalyticsData (key, value)`

- `key`: key to custom data being added
- `value`: value being added for this key

Attaches custom data to analytics generated for the current transaction.

To enable this feature, the analytics settings in the require statement must be set. See 'Node.js Agent Settings' in Install the Node.js Agent

See for Analytics Data Sources for information about AppDynamics Analytics.

`appd.addSnapshotData (key, value)`

- `key`: key to custom data being added
- `value`: value being added for this key

Attaches custom data to snapshots generated for the current transaction.
addSnapshotData() can be called on a transaction at any time, but the added data is used only when a transaction snapshot is generated. See Transaction Snapshots for information on when transaction snapshots are generated.

The following sample adds session count to USER DATA in snapshot:

```javascript
txn.addSnapshotData("active sessions", my.api.getSessionCount());
```

If the data to be added to the snapshot is available after the transaction completes, it can be attached using a `txn.onSnapshotCaptured(txn)` callback. This technique avoids overhead for collecting the data when no snapshot has been generated. The custom data is exposed in the USER DATA tab of the transaction snapshot details in the controller UI.

### appd.markError(error, [statusCode])

- **error**: JavaScript error object
- **statusCode**: optional transaction status code

This value can also be passed as a statusCode property on the error object. If neither a statusCode parameter nor an error.statusCode property in the error parameter is provided, the status code defaults to 500. See also TimePromise.prototype.end(error) for another way to attach an error object to a business transaction.

Marks the transaction as an error transaction. The marked transaction is reported as an error transaction and not as a slow, very slow or stalled transaction, even if the transaction was also slow or stalled.

```javascript
txn.markError(error, [statusCode])
```

- **error**: JavaScript error object
- **statusCode**: optional transaction status code. This value can also be passed as a statusCode property on the error object.

If neither a statusCode parameter nor an error.statusCode property in the error parameter is provided, the status code defaults to 500.

See also TimePromise.prototype.end(error) for another way to attach an error object to a business transaction.

Marks the transaction as an error transaction. The marked transaction is reported as an error transaction and not as a slow, very slow or stalled transaction, even if the transaction was also slow or stalled.

```javascript
txn.addSnapshotData(key, value)
```

- **key**: key to custom data being added
- **value**: value being added for this key

Attaches custom data to snapshots generated for this transaction.

addSnapshotData() can be called on a transaction at any time, but the added data is used only when a transaction snapshot is generated. See Transaction Snapshots for information on when transaction snapshots are generated.

The following sample adds session count to USER DATA in snapshot.

```javascript
txn.addSnapshotData("active sessions", my.api.getSessionCount());
```

If the data to be added to the snapshot is available after the transaction completes, it can be attached using a `txn.onSnapshotCaptured(txn)` callback. This technique avoids overhead for collecting the data when no snapshot has been generated. The custom data is exposed in the USER DATA tab of the transaction snapshot details in the controller UI.

### txn.onSnapshotCaptured(txn)

- **txn**: transaction

Callback method that can be set on a transaction to add custom data to the transaction snapshot. Used with `txn.addSnapshotData(key, value)`.
When the `callback` function is supplied, the agent invokes this API immediately after capturing a transaction snapshot. If specified, the `callback` is invoked on transaction completion, if the transaction triggers a snapshot.

You must define a function to call and assign it to the `callback`.

```javascript
mytxn.onSnapshotCaptured = customTitle (txn) {
    // get book title for current transaction instance
    title = getBookTitle();
    txn.addSnapshotData ("book title", title);
}
```

Note that `txn === mytxn`. It is passed into the callback so the callback does not need to be defined in the same scope as `mytxn`.

**txn.addAnalyticsData(key, value)**

- `key`: key to analytics data being added
- `value`: value being added for this key

Attaches custom data to analytics generated for this transaction.

To enable this feature, the analytics settings in the require statement must be set. See ‘Node.js Agent Settings’ in Install the Node.js Agent

See for Analytics Data Sources for information about AppDynamics Analytics.

**txn. onResponseComplete(req, res)**

- `req`: HTTP request for the transaction
- `res`: HTTP response for the transaction

Callback method that can be set on a transaction to be notified when the associated HTTP response has been completed. This can be used with `txn.addAnalyticsData(key, value)` to add data from response after it is complete, before the transaction completes. When the callback function is supplied, the agent invokes this API immediately after the HTTP response completes, before closing out the transaction. You must define a function to call and assign it to the callback.

```javascript
mytxn.onResponseComplete = function customTitle (req, res) {
    // get status code of HTTP response
    code = res.statusCode;
    txn.addAnalyticsData ("http response code", code);
};
```

Exit Call Management

**TimePromise.prototype.startExitCall(exitCallInfo)**

- `exitCallInfo`: object

  The `exitCallInfo` object has the following required properties:

  - `exitType`: string—the type of exit call, one of "EXIT_HTTP", "EXIT_DB", "EXIT_CACHE", "EXIT_RABBITMQ", "EXIT_WEBSERVICE"
  - `label`: string—label for the exit call in the AppDynamics UI
  - `backendName`: string—name of the backend remote system or service
  - `identifyingProperties`: Object—a hash of name/value pairs uniquely identifying the remote system or service
being called. See Backend Identifying Properties

Creates and starts a custom exit call, described by the exitCallInfo object.

You can supply more identifying properties in addition to the required ones.

Returns an ExitCall. See Exit Call Properties

Sample exitCallInfo Object for MySQL

```json
{
    "exitType": "EXIT_DB",
    "label": "New SQL",
    "backendName": "NewDB"
    "identifyingProperties": {
        "HOST": "database-server",
        "PORT": "12345",
        "DATABASE": "my-database",
        "VENDOR": "MYSQL"
    }
}
```

TimePromise.prototype.endExitCall(exitCall, error)

- exitCall: ExitCall instance returned by startExitCall()
- error: any; optional

Ends a custom exit call and attaches it to this transaction.

If an error is passed, the error is associated with the exit call, and the exit call is flagged as being in an error state.

TimePromise.prototype.beforeExitCall(exitCall)

- exitCall: ExitCall

Returns an ExitCall.

Optional callback for modifying an automatically-detected (not custom) exit call.

The agent invokes this API immediately before your application makes the exit call when you supply a callback function. You can use the callback to:

- Modify the exit call that is reported to the controller.
- Suppress all reporting of the exit call by returning nothing.
- Capture the exit call to create correlation information for a downstream transaction.

The following is an example of a modify exit call reported to the Controller.
txn.beforeExitCall = function customExitCallHandler(exitCall) {
    // don't report database access for this transaction
    if (exitCall.isSql) return;
    // customize label for all other exit calls in this transaction
    exitCall.label += " (my custom transaction)";
    return exitCall;
}

TimePromise.prototype.exitCallCompleted(exitCall)

• exitCall: ExitCall

Returns an ExitCall.

Optional. Callback for modifying either a custom exit call created by startExitCall() or an automatically-detected exit call on completion of the exit call.

The agent invokes this API immediately after an exit call completes when you supply a callback function.

Use exitCallCompleted() to modify how the exit call is reported to the Controller by returning a modified exit call.

Do not modify the backend identifying properties using this callback.

The following code is an example that modifies the exit call reported to the controller

txn.exitCallCompleted = function customExitCallPostProcessor(exitCall) {
    // report MongoDB read and write operations as distinct exit calls
    if (exitCall.backendName === "MongoDB")
        exitCall.label += " " + exitCall.category;
}

Transaction Correlation Management

appd.parseCorrelationInfo(source)

• source: String or HttpRequest

Describes the upstream transaction with which to correlate.
If an HttpRequest object is passed, it must have a transaction correlation header attached.

On success, returns a CorrelationHeader object that can be used by startTransaction() to create a transaction that is correlated with the upstream transaction described by source.

Returns false if source could not be parsed. This can occur if the source is an HTTP request with no correlation header attached or the string parameter is not recognized as a correlation header.

HTTP requests made through the standard http.request() API have correlation information added automatically. You can correlate a custom transaction created in response to an HTTP request made from another Node.js process by passing the incoming request as source.

For other exit call types, you need to define how correlation information is attached to the originating transaction and retrieved in the downstream transaction.

TimePromise.prototype.createCorrelationInfo(exitCall, doNotResolve)
The input `exitCall` is one of the following:

- For a custom exit call, value returned from `startExitCall()`
- For an automatically-detected exit call, input parameter to the `beforeExitCall()` callback

Set the `doNotResolve` parameter to `true` if you do not want the backend to be resolved to a tier. It defaults to `false`, which means that the backend is resolved to the calling tier. You may want to set this for an exit call to a service, such as a messaging queue, that does not have a 1:1 relationship between the consumer and producer of the service. For more information see Resolve Remote Services to Tiers.

Returns a string-encoded correlation header, which a downstream transaction can use to correlate with this transaction.

### Exit Call Properties

This table describes the properties of the exit call object for each detected exit call type.

A dash (`–`) in a table cell indicates that the property is present in a discovered exit call but its format is unspecified. This allows you to set your own values when you create custom exit calls.

N/A indicates that the property is not used with that exit call type.

Discovering exit calls may have additional properties not documented here.

<table>
<thead>
<tr>
<th>Property</th>
<th>Http</th>
<th>MySQL</th>
<th>Postgres</th>
<th>MongoDB</th>
<th>Memcached</th>
<th>Redis</th>
</tr>
</thead>
<tbody>
<tr>
<td>backendName</td>
<td>&quot;HTTP&quot;</td>
<td>&quot;MySQL&quot;</td>
<td>&quot;PostgresSQL&quot;</td>
<td>&quot;MongoDB&quot;</td>
<td>&quot;Memcached&quot;</td>
<td>&quot;Redis&quot;</td>
</tr>
<tr>
<td>label</td>
<td>String</td>
<td>String</td>
<td>String</td>
<td>String</td>
<td>String</td>
<td>String</td>
</tr>
<tr>
<td>method</td>
<td>String</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>requestHeaders</td>
<td>Object</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>responseHeaders</td>
<td>Object</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>statusCode</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>category</td>
<td>&quot;read&quot;</td>
<td>&quot;write&quot;</td>
<td>N/A</td>
<td>N/A</td>
<td>&quot;read&quot;</td>
<td>&quot;write&quot;</td>
</tr>
<tr>
<td>user</td>
<td>N/A</td>
<td>String</td>
<td>String</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>command</td>
<td>String (URL)</td>
<td>SQL query executed</td>
<td>SQL query executed</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>commandArgs</td>
<td>N/A</td>
<td>SQL query positional parameters</td>
<td>SQL query positional parameters</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>isSql</td>
<td>N/A</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>stackTrace</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>exitType</td>
<td>&quot;EXIT_HTTP&quot;</td>
<td>&quot;EXIT_DB&quot;</td>
<td>&quot;EXIT_DB&quot;</td>
<td>&quot;EXIT_DB&quot;</td>
<td>&quot;EXIT_CACHE&quot;</td>
<td>&quot;EXIT_CACHE&quot;</td>
</tr>
<tr>
<td>identifyingProperties</td>
<td>See Backend Identifying Properties and TimePromise.prototype.startExitCall(exitCallInfo)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* host, port and path; does not include query string

### Backend Identifying Properties

This table lists the identifying properties for the backends that are automatically detected by the Node.js Agent.

In the controller UI, these properties are visible in the upper right panel of the backend dashboards.

<table>
<thead>
<tr>
<th>Backend Service</th>
<th>Exit Call Type</th>
<th>Identifying Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP Server</td>
<td>&quot;EXIT_HTTP&quot;</td>
<td>&quot;HOST&quot;, &quot;PORT&quot;</td>
</tr>
<tr>
<td>MySql Server</td>
<td>&quot;EXIT_DB&quot;</td>
<td>&quot;Host&quot;, &quot;Port&quot;, &quot;Database&quot;, &quot;Vendor&quot; = &quot;MYSQL&quot;</td>
</tr>
<tr>
<td>Server Type</td>
<td>EXIT_FUNCTION</td>
<td>EXIT_ARGUMENTS</td>
</tr>
<tr>
<td>------------------</td>
<td>---------------</td>
<td>-------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Postgres Server</td>
<td>&quot;EXIT_DB&quot;</td>
<td>&quot;Host&quot;, &quot;Port&quot;, &quot;Database&quot;, &quot;Vendor&quot; = &quot;POSTGRESQL&quot;</td>
</tr>
<tr>
<td>MongoDB Server</td>
<td>&quot;EXIT_DB&quot;</td>
<td>&quot;Server Pool&quot;, &quot;Database&quot;, &quot;Vendor&quot; = &quot;MONGODB&quot;</td>
</tr>
<tr>
<td>Memcached</td>
<td>&quot;EXIT_CACHE&quot;</td>
<td>&quot;EXIT_CACHE&quot;, &quot;Server Pool&quot;</td>
</tr>
<tr>
<td>Redis Server</td>
<td>&quot;EXIT_CACHE&quot;</td>
<td>&quot;Server Pool&quot;</td>
</tr>
</tbody>
</table>

1 "n" separated list of server addresses in <host>:<port> format
2 Single server address in <host>:<port> format
Node.js Agent Node Properties

This reference page contains information about app agent node properties. The properties are listed in alphabetical order.

### max-continuing-concurrent-snapshots
A limit for the number of concurrent snapshots continuing on this tier or node.
For example, if too much overhead is being taken by snapshots, then tweak this value to reduce the snapshots.

<table>
<thead>
<tr>
<th>Type</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default Value</td>
<td>200</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Node.js</td>
</tr>
</tbody>
</table>

### max-continuing-snapshots-per-minute
A limit for the number of snapshots continuing on this tier or node per minute.
For example, if too much overhead is being taken by snapshots, then tweak this value to reduce the snapshots.

<table>
<thead>
<tr>
<th>Type</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default Value</td>
<td>100</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Node.js</td>
</tr>
</tbody>
</table>

### max-error-snapshots-per-minute
A limit for the number of snapshots on this tier or node per minute to capture errors. For example, if too many error snapshots are being seen, then tweak this value to reduce noise in the snapshots.

<table>
<thead>
<tr>
<th>Type</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default Value</td>
<td>5</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Node.js</td>
</tr>
</tbody>
</table>
### max-originating-concurrent-snapshots

A limit for the number of concurrent snapshots originating on this tier or node. For example, if too much overhead is being taken by snapshots, then tweak this value to reduce the snapshots.

<table>
<thead>
<tr>
<th>Type</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default Value</td>
<td>10</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Node.js</td>
</tr>
</tbody>
</table>

### max-originating-snapshots-per-minute

A limit for the number of snapshots originating on this tier or node per minute. For example, if too much overhead is being taken by snapshots, then tweak this value to reduce the snapshots.

<table>
<thead>
<tr>
<th>Type</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default Value</td>
<td>20</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Node.js</td>
</tr>
</tbody>
</table>
This topic introduces the PHP Agent, the application monitoring agent for web and CLI PHP applications in AppDynamics.

**About the PHP Agent**

The PHP agent component discovers and monitors business transactions, application services, and backends in your PHP application. It injects AppDynamics instrumentation into the application at runtime.

The AppDynamics agent proxy is a Java process that handles the communication between the PHP agent and the Controller. The proxy reports performance metrics to the Controller, where the data is stored, baselined, and analyzed. You can access this performance data interactively using the Controller console or programmatically using the AppDynamics REST API.

The proxy component starts automatically when you start the PHP Agent. For some applications, such as for PHP CLI applications, you need to start the proxy separately from the agent. You need to provide for proxy startup either manually or in scripts.

**Deployment Overview**

The proxy is the component that connects the agent and the controller. If you can ping the Controller but the agent is not connecting, it is possible that the proxy did not start or that the proxy is not configured correctly. See Resolve PHP Agent Installation Issues and Dynamic Language Agent Proxy.

When the connection is successfully established, log in to the Controller to monitor your application. See AppDynamics Essentials to help you get started.

**Single Application Per Web Server Instance**

This is the default setup assumed by the installer—multiple servers, each with a single application.
Each application has its own agent and its own agent and proxy. There is a single tier and a single node for each application.

**Single Application Per Web Server Instance with Multi-Tenant Proxy**

This scenario also assumes multiple servers, each with a single application.
In this case, each application has its own agent component but all the agents share a single proxy. There is a single tier and a single node for each application. Sharing a proxy can save some overhead in some situations. See Use a Shared Proxy for PHP Agents.

Multiple Applications on a Single Server

This scenario assumes multiple applications running in a single server.

There is one agent and one proxy supporting multiple applications. By default, the proxy automatically starts when the server starts. The applications can be configured with different tiers and nodes. See Multiple PHP Apps on a Single Server for information on how to configure the Apache vhosts or FPM pool config blocks.

Getting Started with the Download Wizard

If you have never installed the agent, the best way to start is with the Getting Started Wizard in the Controller. The Getting
Started Wizard walks you through configuration steps and helps you download the agent. To access the wizard, in the Home page of the controller click Getting Started and then PHP.

The wizard provides the minimum information that the agent needs to communicate with the AppDynamics Controller: controller host and port, optionally SSL, application name and tier name.

**Node Naming in a PHP Environment**

A PHP runtime instance maps to a node. Your naming convention may depend on your exact environment. Use a name that clearly identifies the Web service that corresponds to the node. Some options are:

- `hostName-appName-nodeName`
- `hostName-tierName-nodeName`
- `appName-nodeName`
- `tierName-nodeName`
- IP address
- fully qualified domain name
PHP Supported Environments

On this page:
- PHP Versions
- PHP Web Servers
- Operating Systems
- PHP Frameworks and Protocols
- Transaction Naming
- PaaS Providers
- Exit Points
- Opcode Cache Compatibility

Related pages:
- PHP Transaction Detection

PHP Agent Support

**PHP Versions**

The PHP agent supports PHP 5.6, 7.0, 7.1, 7.2, and 7.3.

The PHP agent does not:
- Monitor PHP applications in Zend Thread Safety (ZTS) mode. If you are using ZTS, AppDynamics suggests that you review your dependencies on ZTS to confirm that you actually need it, and if you do not, to switch to non-ZTS mode
- Support Zend Monitor
- Officially support plugins that encrypt and/or obfuscate PHP code, such as Zend Guard or IonCUBE Loader

**PHP Web Servers**

- Apache 2.2 and 2.4 in the following modes:
  - prefork mode using mod_php
  - worker MPM mode using mod_fastcgi with php-fpm or mod_fcgid with php-cgi
  - Any web server compatible with php-fpm.

**Operating Systems**

- Any Linux distribution based on glibc 2.5+
- Mac OS X 10.9+

**PHP Frameworks and Protocols**

<table>
<thead>
<tr>
<th>Framework/Protocol</th>
<th>Version</th>
<th>Entry Point Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Drupal</td>
<td>7, 8</td>
<td>Drupal</td>
</tr>
<tr>
<td>WordPress</td>
<td>3.4+, 4.x, 5.x</td>
<td>Wordpress</td>
</tr>
<tr>
<td>Zend</td>
<td>1, 2</td>
<td>PHP MVC</td>
</tr>
<tr>
<td>CodeIgniter</td>
<td>2.x, 3.x</td>
<td>PHP MVC</td>
</tr>
<tr>
<td>FuelPHP</td>
<td>1.5x, 1.6x</td>
<td>PHP MVC</td>
</tr>
<tr>
<td>Magento</td>
<td>1.5, 1.6, 1.7</td>
<td>PHP MVC</td>
</tr>
<tr>
<td>Symfony</td>
<td>1, 2</td>
<td>PHP MVC</td>
</tr>
<tr>
<td>CakePHP</td>
<td>2.x, 3.x</td>
<td>PHP MVC</td>
</tr>
</tbody>
</table>
If your PHP framework is not listed here, the agent detects your entry points as PHP Web and names the business transactions based on the first two segments of the URI — the default naming convention for PHP Web transactions. So it is still possible to monitor applications on unsupported frameworks. Laravel BTs are detected as symfony, as laravel itself is built on top of symfony.

### Transaction Naming

<table>
<thead>
<tr>
<th>Framework/Environment</th>
<th>Default Transaction Naming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Drupal</td>
<td>page callback name</td>
</tr>
<tr>
<td>Wordpress</td>
<td>template name</td>
</tr>
<tr>
<td>PHP MVC Frameworks</td>
<td>controller:action</td>
</tr>
<tr>
<td>PHP Modular MVC Frameworks</td>
<td>module:controller:action</td>
</tr>
<tr>
<td>PHP Web</td>
<td>URI</td>
</tr>
<tr>
<td>PHP Web Service</td>
<td>service name.operation name</td>
</tr>
<tr>
<td>PHP CLI</td>
<td>last two segments of the script's directory path plus the name of the script</td>
</tr>
</tbody>
</table>

Virtual host prefixing is available for all supported entry point types except PHP CLI.

### PaaS Providers

<table>
<thead>
<tr>
<th>PaaS Provider</th>
<th>Buildpack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pivotal Cloud Foundry</td>
<td><a href="https://github.com/Appdynamics/php-buildpack">https://github.com/Appdynamics/php-buildpack</a></td>
</tr>
</tbody>
</table>

See [http://docs.pivotal.io/appdynamics/index.html](http://docs.pivotal.io/appdynamics/index.html) for information about integration with PCF.

### Exit Points

#### Supported HTTP Exit Points

- curl/curl-multi*
- drupal_http_request()  
- fopen(), file_get_contents()  
- Zend_HTTP_Client::request()

*The total time reported for a curl/multi_curl request in the Controller is the same as reported by the function curl_getinfo. Also, we report the following execution metrics in the exit call details for the curl/multi_curl request which are included in the total time:

- namelookup_time
- connect_time
- pretransfer_time
- redirect_time

#### Supported Database Exit Points

- MySQL old native driver (removed for PHP 7)  
- MySQLi Extension*
- OCI8
PDO
PostgreSQL accessed via PDO and pgsql extensions

*mysqli_multi_query is not supported.*

**Supported Cache Exit Points**

- Memcache
- Memcached
- Predis 0.8.5 and 1.1.1, on PHP versions 5.6 and higher
- Phpredis 4.1

Although Predis is a full PHP client library, the PHP Agent supports Predis as an exit point only, not as an entry point.

**Supported Web Service Exit Points**

- PHP SOAPClient
- NuSOAP 0.9.5

**Supported Message Queue Exit Points**

- RabbitMQ

RabbitMQ support requires the `amqp` extension.

**Opcode Cache Compatibility**

- Alternative PHP Cache (APC)
Install the PHP Agent

On this page:

- Installation Overview
- Before Starting
- Installing the PHP Agent
- Files Added by Installation
- Using a Machine Agent on a PHP Node

This topic provides an overview of AppDynamics PHP Agent installation.

Installation Overview

The Getting Started Wizard in the Controller UI walks you through the configuration and installation steps for the PHP Agent.

The installer that you get from the wizard uses the PHP directory specified in your environment PATH variable to determine where to install the agent. If you are instrumenting a PHP installation not specified in your PATH, you can manually configure and invoke the RPM or script installer, as described here.

The PHP Agent runs on Linux and MacOS machines. The PHP Agent is designed for PHP operating environments in which:

- There is a single PHP installation
- PHP is running in a single Apache or FPM pool
- Standard packages have been used to install PHP, Apache and/or PHP-FPM
- No customizations have been made to the PHP configuration

The PHP Agent may work in environments that do not meet all assumptions. However, you should use extra care in installing and thoroughly testing the PHP agent and application in a staging environment.

Before Starting

Before attempting to install the AppDynamics PHP Agent, confirm that you use a supported PHP version. You can check your PHP version using the following command:

```
php -i
```

The PHP Agent does not work with PHP applications that are built with the enable-debug configuration option or in any build otherwise compiled with debugging symbols. To determine whether the application was built with debugging symbols, you can use the following command:

```
php -i | grep -e "Debug Build"
```

The response should be:

```
Debug Build => no
```

You can get the PHP Agent distribution from the AppDynamics download site, https://appdynamics.com/download. Be sure to get the distribution file appropriate for your system, as follows:

- For RedHat or CentOS, use the RPM Package Manager distribution, such as PHP Agent—64-bit RPM.
- For all other Linux distributions, use the distribution identified PHP Agent—64-bit Linux.
- For MacOS, use PHP Agent—64-bit OSX
At installation time, you will need to provide Controller connection settings and identifying information for this PHP node. The Getting Started Wizard walks you through this configuration, so if it is your first time installing an AppDynamics Agent, we recommend using the Getting Started Wizard. For general information about the Controller connection settings, see Agent-to-Controller Connections.

To complete the instrumentation, you will need to stop and start the Apache Server. Be sure to perform the installation in a way that minimizes disruption to service users, if applicable.

Do not install the PHP Agent along with other non-AppDynamics Application Performance Management (APM) tools, especially in a production environment. The PHP Agent installation may fail if there are other APM products installed in the same managed environment.

Installing the PHP Agent

The following topics provide installation details for various scenarios:

- To use install.sh, see Install the PHP Agent by Shell Script. Use install.sh to install on OSX as well as Linux.
- To use RPM, see Install the PHP Agent by RPM.
- To instrument PHP CLI, see also Configure the Agent for PHP CLI Applications.
- To instrument an application on MacOS, see MacOS X Installation Considerations.
- To instrument multiple PHP applications running on the same server, see Multiple PHP Apps on a Single Server.

Files Added by Installation

**PHP configuration files**

For AppDynamics, the PHP configuration files of interest are the `php.ini` and `appdynamics_agent.ini` fragment. AppDynamics settings can be found in either .ini file, depending on the operating system under which your PHP is installed.

The PHP Agent installer adds the `appdynamics_agent.ini` file to the directory that contains your `php.ini` file. You can find this directory using the following command:

```
php -i | grep -e "Additional .ini files parsed"
```

If the installer is not able to determine the directory where the ini fragments for your PHP deployment live, it displays the required AppDynamics ini fragment and prompts you to copy and paste it into your main `php.ini` file.


**.so files**

The installer also installs the `appdynamics_agent.so` file in your PHP extensions directory. You can find this directory using the following command:

```
php -i | grep  extension_dir
```

**Logs**

There is an agent log and a proxy log for each application.

By default, the agent log is written to `$<php_agent_install>/logs/agent.log`. The log contains the transactions that the agent processes and then sends to the proxy. The default pattern for agent log naming is:

- `agent.log`: the current log
- `agent.log.1`: most recent log
- `agent.log.2`: second most recent log
- `agent.log.3`: third most recent log
- agent.log.4: fourth most recent log
- agent.log.5: fifth recent log

By default, the proxy log is written to $<php_agent_install>/logs/proxy_$date.log. This log contains the transactions that the proxy accepts from the agent and then sends to the Controller.

If you configure the logs directory with the --log-dir option, the proxy logs are written to the same directory as the agent logs.

For information about the location of log files generated by an RPM installation, see RPM Log file in Install the PHP Agent by RPM.

Using a Machine Agent on a PHP Node

You can install a machine agent on a node that runs the PHP Agent. However, note that if you install the machine agent on a PHP Agent node and you specify the tier and node name in the controller-info.xml file of the machine agent, the PHP Agent will not successfully register with the Controller.

To avoid this problem:

- Install the PHP Agent before you install the machine agent
- If you install the machine agent on the machine hosting the instrumented PHP node, do not specify the application, tier or node name in the controller-info.xml file of the machine agent. If you do, the PHP Agent may fail to register.
Install the PHP Agent by Shell Script

On this page:
- Installing the PHP Agent
- Installation Command Samples

This topic describes how to install the PHP Agent using the included install.sh script.

**Installing the PHP Agent**

The following steps describe how to install the PHP Agent with the installation script, install.sh.

1. **Verify support and system requirements for your environment** as described in Install the PHP Agent.
2. **Create the php-agent installation directory**. The directory should be owned by the same user that runs Apache or PHP-FPM (FastCGI Process Manager). AppDynamics recommends naming the directory /opt/appdynamics/php-agent.

   This document refers to the PHP Agent installation directory as: `<php_agent_install>`. Replace the `<php_agent_install>` syntax with the full path to your agent installation directory. For example, change `<php_agent_install>` to `/opt/appdynamics/php-agent`.

3. **If there is more than one Apache instance on a machine**, run install.sh once for each Apache instance, each time with the appropriate node, php_ini dir and php_ext dir options. In this case, see also Run the PHP Proxy Daemon Manually. See Files Added to Your Installation for information about the default installation directories. From the command line, change directories to the PHP agent installation directory and untar the agent distribution tarball, as in the following example:

   ```
   cd <php_agent_install>
   tar -xvjf appdynamics-php-agent-x64-linux.tar.bz2
   ```

4. **Set the following permissions on your `<php_agent_install>` directory.**
   - php: Make every directory that leads to the PHP agent logs directory readable and executable by all and writable by the directory owner:

     ```
     chmod -R 755 <php_agent_install>
     ```
   - logs: If possible, make the logs subdirectory readable/writable/executable by all:

     ```
     chmod 777 <php_agent_install>/logs
     ```

   Directory access permissions of 777 may be too permissive for some organizations. In this case, simply make sure that the directory is owned by the apache/php/proxy user.

5. Run the installation script using this syntax:

   ```
   ```
The command arguments are described as follows:

- **-s** option: You can optionally specify the **-s** option if you want the agent to use SSL (HTTPS) to connect to the controller. In this case, set the Controller port to the HTTPS port of the controller.
- **-a** account_name@account_access_key: The required controller account name and account key. To find your account name and access key, click the gear (⚙️) icon in the upper right corner of the AppDynamics UI, then click License.
- **http-proxy-host** and **http-proxy-port**: Set the **http-proxy-host** and **http-proxy-port** to route data to the controller through a proxy server. The **http-proxy-host** is the hostname or IP address of the proxy server. The **http-proxy-port** is the proxy server HTTP or HTTPS port, whichever you are using. If you set the http-proxy-host you must set the http-proxy-port as well.
- **http-proxy-user** and **http-proxy-password-file**: These are required if using a proxy and the proxy server requires credentials.
- **-e** option: Extensions directory for the `appdynamics_agent.so` file. Needed on Ubuntu as well as when the default PHP CLI binary cannot be determined.
- **-i** option: ini directory for the `appdynamics_agent.ini` file. Needed on Ubuntu as well as when the default PHP CLI binary cannot be determined.
- **-v** option: Version of PHP that you are instrumenting. Valid formats are version numbers to one or two decimal positions, for example, 5.4 and 5.4.21 are both valid. Needed only when the default PHP CLI binary cannot be determined or there is no PHP CLI binary.
- **-p** option: Path to the PHP binary
- **enable-cli-long-running**: Set to **true** to defend PHP in long-running CLI applications. Defaults to **false**. See Long-Running CLI Applications with the Suhosin Patch in Configure the Agent for PHP CLI Applications.
- **--log-dir=<dir>**: The directory to which the agent and proxy logs should be written. Defaults to the `<php_agent_install>/logs` directory.
- **--proxy-ctrl-dir=<dir>**: The proxy control directory. If not specified, the installer creates a temporary directory.

If all options are used, the **-e**, **-i** and **-v** options have precedence over the **-p** option.

On Ubuntu, the installation needs to be performed as the root user. Also, you need to use the **-e** option to indicate the correct extensions directory for the `appdynamics_agent.so` file and the **-i** option for the correct ini directory for the `appdynamics_agent.ini` file.

6. Restart the web server, unless you are installing an agent to monitor PHP-CLI only.

If your installation failed, see Resolve PHP Agent Installation Issues.

**Installation Command Samples**

The following illustrates a sample command to install the agent:

```
install.sh controller 8090 -a=PHPCust@XC6v2n8m2$543 myApp myTier myNode
```

To install the agent using SSL, use the **-s** switch, as follows:
```
install.sh -s -a=PHPCust@XC6v2n8m2$543 controller1.appdynamics.com
8818 myApp myTier myNode
```

The following sample command illustrates installation with a proxy server present:

```
install.sh --http-proxy-host=myproxyhost --http-proxy-port=8099
-a=PHPCust@XC6v2n8m2$543 controller 8090 myApp myTier myNode
```
Install the PHP Agent by RPM

On this page:
- About the RPM Installer
- Installing the PHP Agent with RPM
- RPM Environment Variables
- Updating the Installation
- RPM Installation Log File
- Uninstall the PHP Agent using RPM

Related pages:
- Resolve PHP Agent Installation Issues

You can use the AppDynamics PHP Agent RPM to install the PHP Agent on RedHat or CentOS Linux systems, as described here.

**About the RPM Installer**

You must use the PHP Agent RPM installer as the root user.

You pass configuration settings for the agent to the installer using environment variables rather than command-line arguments. Before starting—or as part of your installation script—you need to set the environment variables. You can configure settings such as node name, tier name, business application name, and others for the agent. However, the installer will provide sensible defaults if not provided.

The RPM installer determines the location of your PHP installation based on the PATH environment variable. It uses the first PHP installation that it encounters in the PATH to configure the installer. If you have installed PHP in a non-standard location, you must provide the directory of your PHP binary in the `APPD_PHP_PATH` environment variable.

Before starting, get the RPM package from the AppDynamics Download Center.

**Installing the PHP Agent with RPM**

To install PHP Agent with RPM:

1. Verify support and system requirements for your environment as described in Install the PHP Agent.
2. Set environment variables with the configuration settings for the agent if needed. The account key is a required setting. Other settings may be required depending on your environment. See RPM Environment Variables for more information. Alternatively, you can pass environment variables to the installation script directly.
3. Run the installer.

```
sudo rpm -i <package_name>
```

Replace `<package_name>` with the name of the package you downloaded.

If you are using `sudo` to pass the environment variables to the installation script you can use:

```
sudo APPD_PHP_PATH=/opt/php rpm -i <package_name>
```
or

```
APPD_PHP_PATH=/opt/php sudo -E rpm -i <package_name>
```

If the install reports errors, check the log file as described in RPM Log File.

4. Restart Apache. Not required if you are installing an agent to monitor PHP CLI only.

The RPM installs a single agent. If you have more than one PHP installation on the machine, run the RPM once for each PHP installation, each time with the appropriate `APPD_PHP_PATH` and `APPD_CONF_NODE` settings.
## RPM Environment Variables

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Description</th>
<th>Default</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPD_PHP_PATH</td>
<td>Directory containing the PHP binary</td>
<td>None</td>
<td>If your PHP binary is not in a std location. By default, the installer PHP CLI binary to determine where to install the app agent.</td>
</tr>
<tr>
<td>APPD_PHP_CONFIGURATION_DIR</td>
<td>INI directory in which to install the appdynamics_agent.ini file. Takes precedence over the APPD_PHP_PATH setting.</td>
<td>Directory containing your php.ini file. See information on files added to your installation in Install the PHP Agent.</td>
<td>If your PHP binary is not in the std location or if no PHP CLI binary available.</td>
</tr>
<tr>
<td>APPD_PHP_EXTENSION_DIR</td>
<td>Extensions directory in which to install the appdynamics_agent.so file. Takes precedence over the APPD_PHP_PATH setting.</td>
<td>Your PHP extensions directory. See information on files added to your installation in Install the PHP Agent.</td>
<td>If your PHP binary is not in the std location or if no PHP CLI binary available.</td>
</tr>
<tr>
<td>APPD_PHP_VERSION</td>
<td>Version of PHP that you are instrumenting. Valid formats are version numbers to one or two decimal positions, for example, 5.4 and 5.4.21 are both valid. Takes precedence over the APPD_PHP_PATH setting.</td>
<td>Version used by your PHP CLI binary</td>
<td>If no PHP CLI binary is available, along with the APPD_PHP_CONFIGURATION_DIR and APPD_PHP_EXTENSION_DIR variables.</td>
</tr>
<tr>
<td>APPD_CONF_CONTROLLER_HOST</td>
<td>Controller hostname</td>
<td>localhost</td>
<td></td>
</tr>
<tr>
<td>APPD_CONF_CONTROLLER_PORT</td>
<td>Controller port</td>
<td>8080</td>
<td></td>
</tr>
<tr>
<td>APPD_CONF_APP</td>
<td>Application name</td>
<td>MyApp</td>
<td></td>
</tr>
<tr>
<td>APPD_CONF_TIER</td>
<td>Tier name</td>
<td>Hostname of the machine running the script—same as the node name</td>
<td></td>
</tr>
<tr>
<td>APPD_CONF_NODE</td>
<td>Node name</td>
<td>Hostname of the machine running the script</td>
<td></td>
</tr>
<tr>
<td>APPD_CONF_ACCOUNT_NAME</td>
<td>Account name</td>
<td>Defaults to customer1.</td>
<td></td>
</tr>
<tr>
<td>APPD_CONF_ACCESS_KEY</td>
<td>Account key</td>
<td>None</td>
<td>Yes. To find your access key, click the gear ( ) icon in the upper right corner of the AppDynamics UI, then click License.</td>
</tr>
<tr>
<td>APPD_CONF_SSL_ENABLED</td>
<td>True to enable SSL communication with the controller, false otherwise</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>APPD_CONF_HTTP_PROXY_HOST</td>
<td>Hostname or IP address of the http proxy server</td>
<td>None</td>
<td>If you want to route data to the controller through a proxy server.</td>
</tr>
</tbody>
</table>
**APPD_CONF_HTTP_PROXY_PORT**

HTTP or HTTPS port of the http proxy server; must be set if `APPD_CONF_HTTP_PROXY_HOST` is set

None

If you want to route data to the controller through a proxy server.

**APPD_CONF_HTTP_PROXY_USER**

Username on the http proxy server

None

If you want to route data to the controller through a proxy server that requires a username.

**APPD_CONF_HTTP_PROXY_PASSWORD_FILE**

Password on the http proxy server

None

You want to route data to the controller through a proxy server that requires a password.

**APPD_PROXY_CTRL_DIR**

Initial control communication directory between the agent and the Java proxy

None

**APPD_CONF_LOG_DIR**

Directory to which the agent writes the agent and proxy logs

<php_agent_install>/logs

**APPD_CONF_CLI_LONG_RUNNING_ENABLE**

Defends PHP in long-running CLI applications.

false

Advisable if instrumenting long-running CLI applications. See Long-Running CLI Applications with the Suhosin Patch in Configure the Agent for PHP CLI Applications.

---

**Updating the Installation**

Any changes that you made to the configuration files are preserved when you re-run the installer. RPM saves your original settings and `appdynamics_agent_log4cxx.xml` files with the settings from the previous installation.

**RPM Installation Log File**

If errors occurred during installation, a message displays the location of the log file generated in the `/tmp` directory. Examine this log file to identify the cause of the problem. A successful installation does not produce a log file.

**Uninstall the PHP Agent using RPM**

If you installed the agent using RPM, use RPM to uninstall it, as follows:

```
sudo rpm -e appdynamics-php-agent-<version>
```

Notice that you need to identify the version of the package. To find the version of the package that you installed, you can enter:

```
rpm -qa | grep appdynamics-php-agent
```

The existing configurations are saved in a tar file in `/tmp`, as indicated upon completion of the uninstall process.
Configure the Agent for PHP CLI Applications

On this page:

- Instrumenting PHP CLI Applications
- PHP CLI and the Apache Web Server
- Long-Running CLI Applications with the Suhosin Patch

Related pages:

- Start the PHP Agent Proxy Manually
- PHP Agent Configuration Settings

If you are instrumenting a PHP CLI Application with the AppDynamics PHP Agent, after installing the agent you need to take a few additional configuration steps described here.

**Instrumenting PHP CLI Applications**

When instrumenting a PHP CLI application, you need to provide for the startup of the proxy daemon, as follows:

1. Create the proxy control directory, which is used for agent/proxy communication.
2. In your PHP configuration file—php.ini or appdynamics_agent.ini depending on your environment. Include the following settings:
   
   ```ini
   agent.cli_enabled = 1
   agent.auto_launch_proxy = 0
   agent.proxy_ctrl_dir  = <proxy control directory>
   ```
   
   If you are using the RPM installer to install the agent you may have configured the proxy control directory using the AP_PD_PROXY_CTRL_DIR environment variable. See Install the PHP Agent by RPM. This environment variable takes precedence over the setting in the ini file.
3. Before running any traffic through the CLI, run the proxy from the directory into which you installed the PHP agent, passing the proxy control directory and proxy log directory as arguments.

   ```
   proxy/runProxy <proxy_control_dir> <log_dir>
   ```

   For example:

   ```
   proxy/runProxy /tmp/proxy.communication /tmp/agentLogs
   ```

   For the full set of options to runProxy, see Start the PHP Agent Proxy Manually.

**PHP CLI and the Apache Web Server**

If you have PHP CLI applications and an Apache web server on the same machine, your setup depends on whether you want all the traffic reported against a single AppDynamics node or separate nodes. A separate proxy is required for each AppDynamics node that you want to monitor in the controller.

If you want all the CLI traffic to be reported against one node and all the web traffic to be reported against a different node, configure Apache to auto-launch the proxy (the default) and configure CLI to use a manually-launched proxy. This requires separate .ini files—one for the web PHP with `agent.auto_launch_proxy` set to 1 and another for PHP CLI with `agent.auto_launch_proxy` set to 0.

If you want the web traffic and the CLI traffic to be reported against the same node, configure both Apache and CLI to use the same manually launched proxy.

**Long-Running CLI Applications with the Suhosin Patch**

A side effect of the Suhosin patch is that it prevents the PHP Agent from ensuring cleanup in long-running CLI applications.

If your PHP has the Suhosin patch, it is possible that resources will not be freed in long-running applications. Thus memory leaks could result if the application itself does not explicitly free these resources.
The long-running-cli feature defends PHP applications in an environment in which both of the following conditions exist:

- PHP with the Suhosin patch is running on Debian or Ubuntu. It is common for Debian and Ubuntu PHPs to have this patch. This feature is not needed for PHPs with only the Suhosin extension, which is different from the patch. Be aware that some PHPs use both the extension and the patch.
- Using the PHP Agent API, you are instrumenting a CLI application that has multiple unbounded business transactions running on the same process.

How the Long-Running CLI Feature Works

At installation time, if the installer determines that PHP has the Suhosin patch and CLI is enabled—agent.cli_enabled=1—the value of the installer option results as follows:

- If true, a fatal error is generated and the installer terminates. With the option set, the installer refuses to instrument a long-running CLI application on a PHP installation with the Suhosin patch.
- If false—the default—the installation continues and warns that memory leaks could occur in long-running CLI processes.

If the installer determines that PHP does not have the Suhosin patch, the installation continues. Long-running CLI processes are supported by the agent, since there is no Suhosin patch.

If the agent could not determine whether your PHP has the Suhosin patch at installation but it does detect the patch at runtime, having set the installer option to 'true' prevents the agent from instrumenting any CLI processes, not just long-running ones. This prevents the Suhosin-patched PHP from exiting.

If CLI is enabled and the installer did not terminate because of the detection of the Suhosin patch, AppDynamics recommends that you install the agent with the enable-cli-long-running option—shell script installs—or the APPD_CONF_CLI_LONG_RUNNING_ENABLED environment variable—for RPM installs—set to true. This will defend your PHP if the patch is detected at runtime.

If the CLI part of your application does not get instrumented, because the installer detected the Suhosin patch, you can unset the option by setting the agent.cli_long_running option in the PHP ini file to off. Or alternatively, you can re-install with the installer.
Multiple PHP Apps on a Single Server

If you follow only the default installation instructions, the PHP Agent treats your entire PHP deployment as a business application with a single tier with a single node. It models your AppDynamics environment this way even if in reality you have multiple PHP applications running in the same Apache or PHP-FPM pool.

You may have configured your PHP applications as different virtual hosts or FPM pools. With multi-application/single-server (MASS) support, you can monitor several PHP applications running on the same server as separate AppDynamics entities, represented as different business applications, different tiers or different nodes. Multiple applications running in the same PHP server can have separate metrics, dashboards, health rules, events, and so on. You can monitor them on different controllers, even from different AppDynamics accounts. You can mix and match the settings to configure the AppDynamics model that works best for monitoring your entire PHP application environment.

AppDynamics recommends that you not instrument more than ten applications on a single server.

Download and Install

To set up your PHP installation for multiple nodes, download a single agent from AppDynamics.

Then follow the instructions in Install the PHP Agent by RPM—RedHat and CentOSX—and Install the PHP Agent by Shell Script—all other Linux and Mac OS—for instrumenting the agent. Configure the required settings using an app/tier/node configuration that fits one of the applications you are instrumenting. Make sure you configure the required settings listed below.

Then, for each additional application that you want the agent to monitor, add an AppDynamics configuration with different settings that reflect how you want the performance metrics to be reported in your AppDynamics model.

Required Configuration Settings for Multi-Node Support

The required settings for multi-node support for each monitored app are:

- agent.controller.hostName
- agent.applicationName
- agent.tierName
- agent.nodeName
- agent.accountName

It may be necessary to configure additional settings depending on your environment, for example, if you need to route data to the AppDynamics controller through a proxy server.

Where to Configure Settings

There are three files from which the PHP Agent configuration settings are read:

1. appdynamics_agent.ini
2. Apache (vhost) or FPM (fpm-pool) configuration (overrides 1)
3. .htaccess for Apache or .user.ini for FPM (overrides 1 and 2)

When you instrument your first application, passing the required settings on the command-line—shell script install—or through environment variables (RPM install), those settings are written to appdynamics_agent.ini.

To apply different sets of settings for the different applications, as best practice specify the settings in each vhost or FPM pool config block.

Sample Apache Configuration
Here are sample virtual host configurations for two applications running on the same server.

**Virtual Host Configuration for the api App**

```
<VirtualHost *:80>
  ServerName api.myhost.com
  DocumentRoot "/var/www/api"
  ErrorLog "/private/var/log/apache2/api_error.log"
  CustomLog "/private/var/log/apache2/api_access.log" common

  php_value agent.applicationName API
  php_value agent.tierName api-tier1
  php_value agent.nodeName api-node1
  php_value agent.controller.hostName appd1.saas.appdynamics.com
  php_value agent.controller.port 8080
  php_value agent.accountName customer1
  php_value agent.accountAccessKey 123456789
</VirtualHost>
```

**Virtual Host Configuration for the frontend App**

```
<VirtualHost *:80>
  ServerName www.myhost.com
  DocumentRoot "/var/www/frontend"
  ErrorLog "/private/var/log/apache2/www_error.log"
  CustomLog "/private/var/log/apache2/www_access.log" common

  php_value agent.applicationName FrontEnd
  php_value agent.tierName frontend-tier1
  php_value agent.nodeName frontend-node1
  php_value agent.controller.hostName appd1.saas.appdynamics.com
  php_value agent.controller.port 8080
  php_value agent.accountName customer1
  php_value agent.accountAccessKey 123456789
</VirtualHost>
```

**Sample FPM Configuration**

Here are sample FPM configurations for two applications running on the same server.

```
.conf Configuration for the "api" App
```
api.conf:

php_value[agent.applicationName] = API
php_value[agent.tierName] = api-tier1
php_value[agent.nodeName] = api-node1
php_value[agent.controller.hostName] = appd1.saas.appdynamics.com
php_value[agent.controller.port] = 8080
php_value[agent.accountName] = customer1
php_value[agent.accountAccessKey] = 123456789

.www.conf Configuration for the FrontEnd App

www.conf:

php_value[agent.applicationName] = FrontEnd
php_value[agent.tierName] = frontend-tier1
php_value[agent.nodeName] = frontend-node1
php_value[agent.controller.hostName] = appd1.saas.appdynamics.com
php_value[agent.controller.port] = 8080
php_value[agent.accountName] = customer1
php_value[agent.accountAccessKey] = 123456789
MacOS X Installation Considerations

If you are instrumenting an application running on MacOS X, you need to instrument it manually using the shell script.

If you are installing the PHP Agent on MacOS X, set `max_execution_time` to 0 in the `appdynamics_agent.ini` file.

If you have another INI file that is loaded after `appdynamics_agent.ini`, and sets `max_execution_time` to a non-zero value, set `max_execution_time` to 0 in the `php.ini` file instead.
Use a Shared Proxy for PHP Agents

On this page:
- Setting up a Multi-Tenant Proxy

Related pages:
- Install the PHP Agent

By default, if you are running multiple agents, each agent automatically launches its own Java proxy to communicate with the controller. However, if you are running multiple PHP Agents on the same machine, you can reduce your overhead by setting up the agents to report to a single shared or multi-tenant proxy. In this case, you would need to start that proxy manually.

The number of nodes that can report to a single proxy is limited by the size of the heap given to the proxy. You may need to adjust the maxHeapSize and maxPermSize settings in the `runproxy` script if you have a large number of agents reporting to a single proxy.

**Setting up a Multi-Tenant Proxy**

1. Configure each agent for manual launch of the proxy. To do this, in the PHP configuration file—`php.ini` or `appdynamics_agent.ini` depending on your setup—for each agent, set the `agent.auto_launch_proxy` value to 0.
2. Configure a single proxy control directory for all the agents that will share the proxy. They must all be on the same machine. To do this, in the PHP configuration file for each agent, set `agent.proxy_ctrl_dir` to the same proxy control directory. The permissions on this directory should be readable and executable by the process that runs Apache and writable by the process that runs the proxy.
3. Before you start the agents, arrange to launch the proxy manually, passing the proxy control directory configured in step 2 as the `proxyCommunicationDir` argument to the `runProxy` script. See information on executing the `runProxy` in Start the PHP Agent Proxy Manually. AppDynamics recommends launching the proxy on system startup.
4. Verify that each agent reporting to the multi-tenant proxy is configured with a unique `app_name/node_name` combination. The `app_name` and `node_name` are arguments to the agent install script.
Start the PHP Agent Proxy Manually

On this page:
- Configure the Agent for Manual Proxy Launch
- Run the Proxy

Related pages:
- Use a Shared Proxy for PHP Agents
- Configure the Agent for PHP CLI Applications

When the PHP agent starts, it automatically starts the Java proxy that handles communication between the PHP agents and the controller.

Automatic startup of the proxy works for the great majority of situations. However, you can suppress the automatic startup of this script and run it manually. You would do this if:

- You plan to instrument a PHP CLI entry point. The PHP CLI entry point requires manual startup of the proxy and immediate creation of the node on startup.
- You have multiple Apache or FPM pools on the same machine reporting to the same proxy.

To do this, you first need to configure the agent for manual proxy launch. Then you need to launch the proxy manually.

**Configure the Agent for Manual Proxy Launch**

The PHP Agent configuration needs to be modified to reflect manual launching.

To configure the agent for manual launching:

1. Open the PHP configuration file for editing. The file is php.ini or appdynamics_agent.ini, depending on your setup.
2. Set the agent.auto_launch_proxy setting to 0.
3. Set the agent.proxy_script to the path of the runproxy that you want to use. The file in the script is relative to the root of the PHP agent. You can specify the absolute path if you prefer.
4. Set the agent.proxy_ctrl_dir directory to use for initial control communication. This directory contains the domain control socket, which the agent uses to start an AppDynamics node. This directory is where the agent gets the configuration for the node. The application user must have read permission on the proxy_ctrl_dir.

Whenever you install the PHP agent, the installer overwrites the runproxy script and the appdynamics_agent.ini file, but not the php.ini. If you re-install, you need to reset the agent.auto_launch_proxy setting in the appdynamics_agent.ini file before you restart the server.

**Run the Proxy**

Before any traffic runs on the instrumented server, execute the runProxy script to start the proxy.

The following lists the full set of options for the runProxy script. The proxy control directory is required.

```bash
Usage: runProxy options -- proxyCommunicationDir logDirectory [jvmOption [ jvmOption [...] ]

Options:
- -r <dir>, --proxy-runtime-dir=<dir>  Specifies proxy runtime directory
- -d <dir>, --proxy-dir=<dir>        Specifies root proxy directory
- -j <dir>, --jre-dir=<dir>         Specifies root JRE directory
- -v, --verbose                      Enable verbose output
- -h,--help                          Show this
```
The following shows an example:

```
./proxy/runProxy -d ./proxy -r <agent.proxy_ctrl_dir>
/tmp/proxy.communication /tmp/agentLogs
```

A node is created when the agent first detects traffic on it.

Every time you reboot the server, you need to execute the `runProxy` script if you have opted to start the proxy manually.
Resolve PHP Agent Installation Issues

On this page:
- Ensure that the Agent is Installed in the Correct Directory
- Check the AppDynamics Settings Block in the Configuration
- Confirm User Permissions
- Verify that the Proxy is Running
- Check the Configuration Properties

If traffic doesn't appear in the Controller after you have installed the AppDynamics PHP Agent and started the instrumented server, try the following troubleshooting steps.

**Ensure that the Agent is Installed in the Correct Directory**

It is possible that the agent was installed in the wrong directory. Verify the location of your PHP installation.

- Verify the location of your PHP by running phpinfo. See [http://us1.php.net/phpinfo](http://us1.php.net/phpinfo)

Then check where the installer actually installed the agent files.

- The `appdynamics_agent.ini` file should be in the same directory that contains the `php.ini` file for your PHP installation.
- The `appdynamics_agent.so` file should be in the extensions directory for your PHP installation.

See [Files Added to Your Installation](#) for information about how to locate these directories.

In addition, on Linux you can use `pstree` to locate the agent. The `pstree` command displays the AppDynamics agent running under Apache if the agent is installed properly. See [http://freecode.com/projects/pstree](http://freecode.com/projects/pstree)

If the agent files are not in the correct directories, re-install the agent with the `-i` and `-e` options.

**Re-install the agent**

If the app agent is not installed in the right directory, re-install the agent using the `install.sh` installer with the `-i` and `-e` options. Use the `-i` to install the `appdynamics_agent.ini` file in the same directory as your `php.ini` file and the `-e` option to install the `appdynamics_agent.so` file in the same directory as your PHP extensions directory. See [Install the PHP Agent by Shell Script](#).

If you initially installed the agent using the RPM installer, you can find the shell script installer at `/usr/lib/appdynamics-php5/install.sh`.

**Check error messages in the installation output**

When you reinstall, examine carefully any error messages in the output of the install script, especially those that direct you to copy some settings into your `php.ini` file.

If necessary copy those settings into the `php.ini` file. See [PHP Agent Configuration Settings](#) for information on common settings that may be missing.

**Check the AppDynamics Settings Block in the Configuration**

Run this command:

```bash
php -i | less
```

and examine the output. You should see an `appdynamics_agent.ini` file and a configuration block listing `appdynamics` `ini` values.

**Confirm User Permissions**

Check that the following permissions are set:
chown -R <apacheuser>:<apacheuser> <php_agent_install>
chmod -R 755 <path_to_php_agent_install_logs>
chmod 777 <php_agent_install>/logs

**Verify that the Proxy is Running**

The Java proxy is the part of the agent that communicates with the Controller. If the agent is installed in the right place, confirm that the Java proxy is running.

1. From the command line enter, `ps aux | grep java`.
2. Inspect the list. You should see output similar to the following if the proxy is running:

```bash
/usr/lib/appdynamics-php5/proxy/jre/bin/java -server -Xmx120m -classpath
-Dappdynamics.agent.logs.dir=/usr/lib/appdynamics-php5/logs
-Dcomm=/tmp/ad-siJ4rp -DagentType=PHP_APP_AGENT
-Dappdynamics.agent.runtime.dir=/usr/lib/appdynamics-php5/proxy
com.appdynamics.ee.agent.proxy.kernel.Proxy
```

If you are instrumenting a PHP CLI script, you need to start the proxy manually. You may also need to start the proxy manually if you have special requirements for running Java processes. See Start the PHP Agent Proxy Manually.

**Check the Configuration Properties**

It is possible that the properties that the proxy uses to communicate with the Controller were not set properly.

You can modify these properties in the `php.ini` or `appdynamics_agent.ini` file, wherever they are set in your environment. See PHP Agent Configuration Settings.
Upgrade the PHP Agent

If you are upgrading both the Controller and agents, first upgrade the Controller and then upgrade the agents.

Also, if you are upgrading multiple agents in your monitored environment, upgrade the agents for the tiers on which business transactions originate last. For more information about this requirement, along with Controller and agent compatibility information, see Agent and Controller Compatibility.

Upgrade the PHP Agent

1. Shut down the web server or `php-fpm`.
2. Copy the controller host, controller port, application name, tier name and node name property values from your `ini` file. If you are running in multi-tenant mode, also copy the account name and account access key property values.
3. Recursively remove or rename the old AppDynamics PHP installation directory.
4. Download and extract the most recent agent tarball.
5. Run the installation script.
   Beginning in 4.1, you must pass the controller access key as one of the options to the installer.
   To find your access key, click the gear (🔧) icon > License > Account.
6. Restart Apache or `php-fpm`.

   If you are using the agent to monitor PHP CLI without running a web server, you can omit steps 1 and 6.
Uninstall the PHP Agent

On this page:

- Uninstall an RPM-Installed PHP Agent
- Uninstall a Script-Installed PHP Agent
- Checking the Installation Method

Related pages:

- Install the PHP Agent by Shell Script
- Install the PHP Agent by RPM

The method you use to install the agent varies depending on the installation mechanism you used, as described here. If you are not sure which method was used, see Checking the Installation Method.

Uninstall an RPM-Installed PHP Agent

1. Shut down the web server, if one is running. This may not apply for a PHP CLI application.
2. Run the following command:

   ```bash
   rpm -e appdynamics-php-agent-<version>
   ```

   where `<version>` is the package you installed. You can find the version using:

   ```bash
   rpm -qa | grep appdynamics-php-agent
   ```

Uninstall a Script-Installed PHP Agent

If you installed the agent using `install.sh`, uninstall that agent as follows:

1. Shut down the web server, if one is running. This may not apply for PHP CLI application.
2. From the PHP agent install directory, run the PHP installer with the `-u` option:

   ```bash
   install.sh -u
   ```

3. Delete the `<php_agent_install>` directory.

Checking the Installation Method

If you used the Agent Download Wizard, it might not be obvious which installer was used since the installer is called by the `runme.sh` wizard script.

Normally, if RHEL and CentOS was selected in the wizard, the RPM installer was used. If All Other Linux OS was selected in the wizard, the shell script installer was used. So use the appropriate uninstall command based on the command that was used to install the agent.

If you do not know which selection was checked in the wizard, run
rpm -qa | grep appdynamics-php-agent

If this does not list any results, assume that the shell script installer was used to install the agent.
PHP Agent Configuration Settings

On this page:

- agent.controller.hostName
- agent.controller.port
- agent.applicationName
- agent.tierName
- agent.nodeName
- agent.accountName
- agent.accountAccessKey
- agent.controller.ssl.enabled
- agent.proxy_ctrl_dir
- agent.http_error_detection=true

The PHP installers write information that the agent uses to communicate with the Controller to the AppDynamics Agent section of the PHP configuration files.

This is different from some of the other AppDynamics application agents, which write this information to an XML file called controller-info.xml.

If you re-install the agent, the installer overwrites your settings only if you are using a fragments directory. It never overwrites the php.ini file. You can edit the PHP configuration files after installation to add, delete or modify these settings.

If a setting documented as required is not provided, the agent will not start. In this case, the agent logs the error to the web server error log. For example, if the controller.Hostname is not set you would see the following message in the Apache error log:

```
[AD agent] agent.controller.hostName is not set. Agent is disabled.
```

If after successful startup you remove a required setting or set it to an empty value, the change is ignored as long as the application is running, using the original value of the setting as it was at startup. If you change a required setting to a different valid value, restart the web server to apply the change.

The Controller information settings are described below:

**agent.controller.hostName**

This is the host name or the IP address of the AppDynamics Controller. Example values are 192.168.1.22, myhost or myhost.abc.com. This is the same host that you use to access the AppDynamics browser-based user interface. For an on-premises Controller, use the value for application server hostname that was configured when the Controller was installed.

This setting is required.

**agent.controller.port**

This is the HTTP(S) port of the AppDynamics Controller. This is the same port that you use to access the AppDynamics browser-based user interface.

If agent.controller.ssl.enabled is true, specify the HTTPS port of the Controller; otherwise, specify the HTTP port. See agent.controller.ssl.enabled.

For on-premises installations, port 8090 for HTTP and port 8181 for HTTPS are the defaults.

This setting is required.

**agent.applicationName**

This is the name of the logical business application that the instrumented node belongs to.
If a business application of the configured name does not exist, it is created automatically.
This setting is required.

**agent.tierName**

This is the name of the logical tier that this node belongs to.
This setting is required.

**agent.nodeName**

This is the name of the instrumented node.
This setting is required.

**agent.accountName**

This is the account name used to authenticate with the Controller.
This setting is required if the AppDynamics Controller is running in multi-tenant mode or if you are using the AppDynamics SaaS Controller. It specifies the account name for the agent to use to authenticate with the Controller. If you are using the AppDynamics SaaS Controller, the Welcome email sent by AppDynamics provides the account name.
This setting is not required if the Controller is running in single-tenant mode.

**agent.accountAccessKey**

This is the account access key used to authenticate with the Controller.
This setting is required. To find your access key, click the gear (⚙️) icon in the upper right corner of the AppDynamics UI, then click **License**.

**agent.controller.ssl.enabled**

When set to `true`, this setting specifies that the agent should use SSL (HTTPS) to connect to the Controller. If `agent.controller.ssl.enabled` is `true`, set the `agent.controller.port` to the HTTPS port of the Controller.

**agent.proxy_ctrl_dir**

This specifies the initial control communication directory between the agent and the proxy. Needed only for a manual proxy start. See **Start the PHP Agent Proxy Manually**

**agent.http_error_detection=true**

When adding this setting to `php.ini`, the PHP agent, marks HTTP error response, such as status code 4xx response as an error.
This topic introduces the PHP Agent API and provides example use cases for the API.

### About the PHP Agent API

The PHP Agent API enables you to:

- Define custom business transactions programmatically
- Provide correlation headers for entry points not supported by default detection
- Create custom exit calls to discover backends that are not automatically detected by the PHP Agent

### Include the AppDynamics API Header

You should include the `appdynamics_api_header.php` file in your application to ensure that it works correctly if the agent is uninstalled or temporarily disabled. This file contains empty API functions that will prevent the application from throwing errors if the agent is not present.

The `appdynamics_api_header.php` file is located in the PHP Agent package in the same directory as the `install.sh` script.

To include the header file:

1. Copy `appdynamics_api_header.php` to where you keep the header files for the monitored application.
2. Make sure that `appdynamics_api_header.php` is in your include path.
3. Then add the following to your script:

   ```php
   require 'appdynamics_api_header.php';
   ```

### Backend Detection with the MySQLi Driver

As noted on PHP Supported Environments, the PHP Agent does not work with PHP 5.2 applications that use the `new` keyword to instantiate a database backend with the MySQLi database driver. For example, AppDynamics does not detect the MySQLi backend created as follows in a PHP 5.2 application:

```php
$db = new mysqli("localhost", "user", "password", "database");
```

You can work around this by using `mysqli_connect()` instead:
Parts of a Script are Business Transactions

If you have a long PHP script application that performs a number of discrete tasks, but you want the agent to detect only one or more of them as business transactions, enclose the code for each of those tasks within `appdynamics_start_transaction()` and `appdynamics_end_transaction()` calls. The agent detects those blocks as separate business transactions. Otherwise, the agent detects the entire script as a single business transaction.

Handling Business Transaction Code Executing in a Loop

For a script-based CLI application that executes in a loop, perhaps fetching items from a database or remote service, you may want the agent to detect every iteration of the loop as a separate business transaction. In this case, enclose the code inside the loop within `appdynamics_start_transaction()` and `appdynamics_end_transaction()` calls.

If you do not do this, the agent will aggregate each iteration through the loop into a single business transaction.

In the following example, the agent detects a business transaction named `getItem` for every iteration.

```php
while (true){
    appdynamics_start_transaction("getItem", AD_CLI);
    //your code goes here
    . . .
    appdynamics_end_transaction();
}
```

Correlating with an Upstream Service

If you have a distributed business transaction in which a tier needs to correlate with an upstream service that is not an entry point supported by the PHP Agent, you can maintain transaction correlation using `appdynamics_continue_transaction()` in the downstream tier, passing it the correlation header from the service.

You need to extract the correlation header from the service as shown in the following sample. The sample function extracts the correlation header from each message in an AMQP message queue and passes it to `appdynamics_continue_transaction()`.

After processing the message, it calls `appdynamics_end_transaction()`, which ends the continuation of the transaction on the calling tier. The `appdynamics_end_transaction()` call does not end the entire distributed transaction in the case where that tier makes a distributed call to another downstream tier.
function amqp_receive($exchangeName, $routingKey, $queueName) {
    $amqpConnection = amqp_connection();
    $channel = new AMQPChannel($amqpConnection);
    $queue = new AMQPQueue($channel);
    $queue->setName($queueName);
    $queue->bind($exchangeName, $routingKey);
    while($message = $queue->get()) {
        // Extracting the correlation header.
        echo("Message #".$message->getDeliveryTag()." \\
            ".$message->getBody()." \\
            "Correlation header: " .
            $message->getHeader("singularityheader"));
        // Passing correlation header to API.
        appdynamics_continue_transaction($message->getHeader("singularityheader"));
        doStuff($message);
        // End transaction.
        appdynamics_end_transaction();
    }
    if(!$amqpConnection->disconnect()) {
        throw new Exception("Could not disconnect !");
    }
}

If the service is not a supported entry point and you do not do this, the tier will not be correlated with the upstream transaction.

Make Socket-based HTTP Calls Example

While the API does not include built-in calls for socket-based HTTP call, you can implement monitoring of socket-based HTTP exit calls yourself as shown in the following example:
<?php

function doSocketHTTPCall($url, $corrHeader = null)
{
    $parts = parse_url($url);
    $fs = @fsockopen($parts['host'], isset($parts['port']) ?
        $parts['port'] : 80, $errno, $error);
    if (!$fs)
        return null;
    $send = "GET {$parts['path']} HTTP/1.1\r\n" .
        "Host: {$parts['host']}\r\n" .
        "Connection: Close\r\n";
    if ($corrHeader)
        $send .= "singularityheader: $corrHeader\r\n";
    $send .= "\r\n";
    fwrite($fs, $send);
    $data = stream_get_contents($fs);
    fclose($fs);
    return $data;
}

$url = 'http://httpstat.us/200';
$parts = parse_url($url);

$exitCall = appdynamics_begin_exit_call(
    AD_EXIT_HTTP,
    'HTTP Status Service',
    array('HOST' => $parts['host'],
        'PORT' => (string)$parts['port'])
);

doSocketHTTPCall($url);

appdynamics_end_exit_call($exitCall);
?>

Inject a Correlation Header into an HTTP Payload Example

The next example injects a correlation header into the socket-based HTTP payload.
The next example shows how to use a non-exclusive flag to start an exit call that may be wrapping other exit calls. The outer socket-HTTP call is started, then the `file_get_contents()` call is processed by the agent normally, and finally, the outer call is finished. We also pass the exception object to report any errors. The end result is that both backends are displayed on the flowmap.
<?php

class SocketHTTPException extends Exception
{
}

$url = 'http://httpstat.us/200';
$url = 'http://myhost.mydomain/process.jsp';
$parts = parse_url($url);

$exitCall = appdynamics_begin_exit_call(
   AD_EXIT_HTTP,
   'HTTP Status Service',
   array('HOST' => $parts['host'],
       'PORT' => (string)$parts['port']),
   false
);
$contents = file_get_contents($url);

if (doSocketHTTPCall($url) == null) {
   $error = new SocketHTTPException("something bad happened");
}
appdynamics_end_exit_call($exitCall, $error);
?>
The PHP Agent APIs support custom business transaction definition and correlation. They provide a way to generate multiple business transactions in a single PHP request. They also enable you to monitor exit calls that are not automatically detected by the PHP Agent.

API Reference Overview

The operations in the API fall into two categories, business transaction management and exit call management.

Business transaction management methods are:

- bool appdynamics_start_transaction($transaction_name, $entry_point_type)
- bool appdynamics_continue_transaction($correlation_header, $entry_point_type)
- bool appdynamics_end_transaction()

Exit call management methods are:

- ADExitCall appdynamics_begin_exit_call($type, $label, $properties, $exclusive=true)
- void appdynamics_end_exit_call(ADExitCall $exitCall, $exception = null)
- string ADExitCall::getCorrelationHeader()
- void ADExitCall::setSQLQueryInfo($querystring, $boundparams)

The following sections detail the methods in the API.

Start Transaction

Starts a custom business transaction

Format

bool appdynamics_start_transaction($transaction_name, $entry_point_type)

Description

If the business transaction initiated by this call is not matched by an appdynamics_end_transaction() call, the transaction terminates at the end of the request or script.

Custom business transactions cannot be nested. If you call appdynamics_start_transaction() multiple times before calling appdynamics_end_transaction(), the last appdynamics_start_transaction() is used and the previous calls are discarded.

Parameters

$transaction_name: The name used for the transaction in the controller. The following characters are not allowed in transaction names: ()[]|& ;
$entry_point_type: Indicates the framework or protocol of the entry point. Valid entry point types are provided as PHP extension
constants, shown below:

- AD_WEB
- AD_MVC
- AD_DRUPAL
- AD_WORDPRESS
- AD_CLI
- AD_WEBSERVICE

Entry point types are case sensitive.

Returns

True on success, false on failure.

Failure conditions are reported in the Apache log. Reasons for failure include:

- Invalid transaction name, contains disallowed characters
- Invalid entry point type
- Agent not initialized
- EUM headers were sent prior to the `appdynamics_start_transaction()` call.
- Correlation headers were sent prior to the `appdynamics_start_transaction()` call.

Continue Transaction

Correlates a custom business transaction with an upstream service.

Format

```php
bool appdynamics_continue_transaction($correlation_header, $entry_point_type)
```

Description

Used by a downstream tier to correlate with a service that is not an entry point supported by the PHP Python Agent.

Parameters

- **$correlation_header**: Correlation header of the upstream service with which to correlate.
  
  It is the developer's responsibility to extract the correlation information from the service to provide the $correlation_header. See `getCorrelationHeader()`.

- **$entry_point_type**: Optional parameter that indicates the framework or protocol of the entry point for the continued transaction. By default, the value is automatically set to AD_CLI when the application is running in CLI mode, and AD_WEB otherwise.

  Use this parameter to indicate a different originating PHP application type for continuing transactions that cross PHP types, such as for a transaction originating at a PHP web application that crosses to a PHP CLI leg of the transaction.

Valid entry point types are provided as PHP extension constants:

- AD_WEB
- AD_MVC
- AD_DRUPAL
- AD_WORDPRESS
- AD_CLI
- AD_WEBSERVICE

Entry point types are case sensitive.

Return

Returns true on success, false on failure.

End Transaction
Ends a transaction created by a previous call to `appdynamics_begin_transaction()` or continued by `appdynamics_continue_transaction()`.

**Format**

```c
bool appdynamics_end_transaction()
```

**Description**

When paired with an `appdynamics_continue_transaction()` call, this call ends the transaction on the tier being continued but does not end any subsequent calls downstream from that tier that are part of the distributed transaction.

If there is no previous `appdynamics_begin_transaction()` or `appdynamics_continue_transaction()` calls in the request/script, this function returns false and does not change the transaction.

**Begin Exit Call**

Marks the start of an exit call.

**Format**

```c
ADExitCall appdynamics_begin_exit_call($type, $label, $properties, $exclusive=true)
```

**Parameters**

- **$type**: The type of exit call. Must be one of the following:
  - `AD_EXIT_HTTP`
  - `AD_EXIT_DB`
  - `AD_EXIT_CACHE`
  - `AD_EXIT_RABBITMQ`
  - `AD_EXIT_WEBSERVICE`

- **$label**: Label for the exit call in the AppDynamics UI. Use a label under 40 characters long so that it fits in flowmaps.

- **$properties**: An associative array of identifying properties—name/value pairs—for the exit call. Property names and values must be strings.

  Each exit call type has its own properties. There is no validation of property names, but each exit type has traditionally used the names listed below.

<table>
<thead>
<tr>
<th>type</th>
<th>HOST</th>
<th>PORT</th>
<th>DATABASE</th>
<th>VENDOR</th>
<th>URL</th>
<th>QUERY_STRING</th>
</tr>
</thead>
<tbody>
<tr>
<td>AD_EXIT_HTTP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AD_EXIT_DB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AD_EXIT_CACHE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AD_EXIT_RABBITMQ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AD_EXIT_WEBSERVICE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For `AD_EXIT_DB` exit call type it is advisable to specify at least `HOST`, `PORT`, and `VENDOR`, as these properties are used by the AppDynamics DB integration. The `VENDOR` property for `AD_EXIT_DB` backends should be one of the following:

- `MYSQL`
- `POSTGRESQL`
- `SQLSERVER`
- `ORACLE`
- `SYBASE`
- `DB2`

- **$exclusive**: Boolean that Indicates whether the exit call is exclusive.
Only one exclusive exit call can be in progress at a time. For example, if this API is used to start an HTTP exit call and there is a
mysql_connect() call immediately following it, the MySQL call will not be detected while the HTTP call is in progress. An exclusive exit
call has to be explicitly ended before subsequent exit calls can be detected or initiated.

Exit calls are exclusive by default, but you can make them non-exclusive by setting this parameter to false.

See the last code sample in the 'Scenario: Application makes socket-based HTTP calls' in PHP Agent API User Guide for an example
of how setting this flag to false can be used to support nested exit calls.

Return
Returns an instance of the ADExitCall class on success, NULL on error.

End Exit Call
Marks the end of an exit call.

Format
void appdynamics_end_exit_call(ADExitCall $exitCall, $exception = null)

Parameters
$exitCall: An object representing the exit call to be ended, and returned from its appdynamics_begin_exit_call().
$exception: An exception object—either Exception class or one derived from it—specifying if an error occurred during the exit call.

Get Correlation Header
Returns the correlation header for this exit call. The returned correlation header can be passed to appdynamics_continue_transaction to correlate with this exit call.

See also the 'Scenario: Application makes socket-based HTTP calls' ample in PHP Agent API User Guide for an example of injecting
the correlation header into an HTTP payload.

Format
string ADExitCall::getCorrelationHeader()

Set SQL Query Info
Sets the SQL query for this exit custom call.

Format
void ADExitCall::setSQLQueryInfo($querystring, $boundparams)

Parameters
$querystring: String containing the SQL query. Use question marks to indicate the bound parameters.
$boundparams: Optional comma-separated array of bound parameters as quoted strings.

Example
```
$exitCall->setSQLQueryInfo("SELECT * FROM mytable where id1=? and id2=?;", ('Susie', '12345'));
```
For the PHP Agent there is an agent log and a proxy log for each instrumented application.

**Agent Log**

The agent log is located at $<php_agent_install>/logs/agent.log. The log contains the transactions that the agent processes and then sends to the proxy.

The default pattern for agent log naming is the following:

- agent.log: the current log
- agent.log.1: most recent log
- agent.log.2: second most recent log
- agent.log.3: third most recent log
- agent.log.4: fourth most recent log
- agent.log.5: fifth recent log.

The agent creates and rotates a maximum of six log files. Maximum log size is 20MB, which gives you a maximum of the most recent 120MB of log data at one time.

**Proxy Log**

The proxy log is located $<php_agent_install>/logs/proxy_$date.log. This log contains the transactions that the proxy accepts from the agent and then sends to the Controller. See Proxy Logging for details.
Python Agent

On this page:
- Python Agent Architecture
- Install the Agent Overview
- Getting Started with the Download Wizard

Related pages:
- Dynamic Language Agent Proxy

The topics in this section describe how to install and administer the Python Agent, the runtime agent used to monitor Python with AppDynamics Application Monitoring.

Python Agent Architecture

The Python Agent discovers, and maps and tracks metrics for business transactions, app services, and backends in your web application by injecting instrumentation into the Python application at runtime. The agent operates from inside the WSGI server that contains the instrumented application.

The agent proxy is a Java process that handles the communication between the Python Agent and Controller. The proxy reports performance metrics to the Controller, where the data is stored, analyzed, and presented.

As shown in the following figure, the proxy can serve multiple Python Agents simultaneously.

In pure Python environments, the proxy is automatically started when you start the Python Agent. In other types of environments, you need to start the proxy manually. See Instrument the Application in Python Agent for more information.
However the proxy is started, the commands that start the agent check whether the proxy is running before attempting to start it.

Install the Agent Overview

If you have never installed the agent, the best way to start is with the Getting Started Wizard in the Controller. This installer configures your agents with the values that you supply in the wizard.

If you downloaded the agent from PyPI, use the instructions in Install the Python Agent to instrument your application manually. In this case, see Python Agent Settings for a complete list of all the Python Agent settings that you can use to instrument the agent.

Getting Started with the Download Wizard

The Getting Started Wizard walks you through configuration steps and helps you download the agent. To access the wizard, from the home page of the Controller, click Getting Started and then Python.

The wizard provides for a minimally configured agent, with settings for the Controller connection, the business application name, and tier name. For advanced scenarios, see Install the Python Agent.
Python Supported Environments

Python Agent Support

* **Python Versions**
  - The Python agent supports CPython 2.6, 2.7, 3.4, 3.5, 3.6, and 3.7.

* **Operating Systems**
  - Any Linux distribution based on glibc 2.5+
  - Mac OS X 10.8+

* **Python Frameworks and Protocols**

<table>
<thead>
<tr>
<th>Framework/Protocol</th>
<th>Version</th>
<th>Entry Point Type</th>
<th>Default Transaction Naming</th>
</tr>
</thead>
<tbody>
<tr>
<td>WSGI</td>
<td>1.0</td>
<td>Python Web</td>
<td>First two segments of URI</td>
</tr>
<tr>
<td>Tornado</td>
<td>3.2 - 4.5</td>
<td>Python Web</td>
<td>First two segments of URI</td>
</tr>
</tbody>
</table>

AppDynamics has tested the Python Agent on Tornado, Django, Flask, CherryPy, Bottle, and Pyramid.

You can configure the agent to instrument any WSGI-based application or framework as Python Web, including but not limited to those listed below.

At present, the Python agent fully supports exception detection in Django, Flask, CherryPy, Bottle, Pyramid, and Tornado frameworks. Other WSGI frameworks and custom WSGI applications may install exception handlers that effectively hide some exceptions from the agent. In such cases, the agent will only detect exceptions during exit calls, uncaught exceptions which are propagated to the WSGI server, and exceptions reported via the custom business transaction API.

<table>
<thead>
<tr>
<th>WSGI-Based Frameworks</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bottle</td>
<td></td>
</tr>
<tr>
<td>CherryPy</td>
<td></td>
</tr>
<tr>
<td>Django</td>
<td></td>
</tr>
<tr>
<td>Flask</td>
<td></td>
</tr>
<tr>
<td>PasteDeploy</td>
<td></td>
</tr>
<tr>
<td>Pyramid</td>
<td></td>
</tr>
</tbody>
</table>

**Database Exit Points**

<table>
<thead>
<tr>
<th>Supported Database Exit Points</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>cx_Oracle</td>
<td>5.1.x</td>
</tr>
<tr>
<td>MongoDB</td>
<td>3.1+</td>
</tr>
<tr>
<td>MySQL-Python</td>
<td></td>
</tr>
</tbody>
</table>
### HTTP Exit Points

**Supported HTTP Exit Points**

<table>
<thead>
<tr>
<th>Library</th>
</tr>
</thead>
<tbody>
<tr>
<td>httplib</td>
</tr>
<tr>
<td>httplib2</td>
</tr>
<tr>
<td>requests</td>
</tr>
<tr>
<td>urllib</td>
</tr>
<tr>
<td>urllib2</td>
</tr>
<tr>
<td>urllib3</td>
</tr>
<tr>
<td>tornado.httpclient</td>
</tr>
</tbody>
</table>

* The agent detects calls to any external library built on top of httplib. Therefore, backend calls to such services, such as boto, dropbox, python-twitter, and so on are detected and displayed as HTTP exit calls.

### Cache Exit Points

**Supported Cache Exit Points**

<table>
<thead>
<tr>
<th>Cache Exit Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memcache</td>
</tr>
<tr>
<td>Redis-py</td>
</tr>
</tbody>
</table>
Install the Python Agent

Before You Begin

1. Verify support for your application environment at Python Supported Environments.
2. Verify that your Python environment meets the following requirements:
   - CPython Versions 2.6, 2.7, 3.4, 3.5, or 3.6
   - pip 1.4 or later
   - Access to the Python Package Index, https://pypi.python.org/ by the machine on which you will install the agent
3. Provide a WSGI-based application to monitor.
4. Verify that you can access the machine where the application runs as a user with privileges to install the agent software and restart the application. Verify that you have a user account with these privileges.
5. If you are using uWSGI, set enable-threads=1 in the uWSGI configuration file. The agent requires multi-threading.
   There is a known incompatibility between the Python Agent and versions of uWSGI installed via OS packages managers such as ‘apt-get’. For this reason, AppDynamics recommends installing uWSGI from pip to avoid this issue.
6. If the application to monitor runs in a virtual environment, activate the virtual environment. For example, the following source command activates a virtual environment:

   ```
   source /<path_to_virtual_environment>/bin/activate
   ```

   Activating a virtual environment is not necessary if the application runs in the global Python environment.

Install the Agent

Log in to the machine on which the Python application runs using appropriate user credentials, as follows:

- For a virtual environment, you need to be the user who owns the virtual environment.
- For the global Python environment, you need to run the install command as root.

To install or upgrade to the latest version of the agent, run the pip install command as follows:

```
pip install -U appdynamics\<4.5
```

When there are multiple packages, you can locate the agent using the pip list command or using the List packages facility at https://pypi.python.org/pypi and then find appdynamics in the output. Here is a sample output:
For production deployments, AppDynamics recommends that you freeze the versions of dependencies so that they are not upgraded in production without first being deployed to your test/staging environments. The command to install or upgrade to a particular version of the AppDynamics Python Agent is:

```
pip install -U appdynamics==<released_agent_version>
```

For example:

```
pip install -U appdynamics==4.4.0.0
```

This command always installs the exact same version of the agent each time it is run.

**Configure the Agent**

Provide a configuration file that specifies the required AppDynamics settings for agent-Controller communication. The file should be in **Python ConfigParser format**. The Python application user must have read access on the configuration file.

Note that lines in the configuration file must not start with spaces. Lines that begin with a # are comments and are ignored by the agent.

The following is a simple sample configuration file with minimum required settings.

```
[agent]
app = <app_name>
tier = <tier_name>
node = <node_name>

[controller]
host = <controller_host>
port = <controller_port>
ssl = on|off
account = <your AppDynamics controller account name>
accesskey = <your AppDynamics controller account access key>
```

Note the following points in the configuration:

1. The ssl settings determines whether the agent connects to the Controller by SSL. This is required for SaaS Controllers.
2. The account value is required if you are using a SaaS account or a multi-tenant on-premises controller. It defaults to `custom` for a single tenant controller.
3. The accesskey is required for all controllers. To find your account name and access key, click the gear (⚙️) icon in the upper right corner of the AppDynamics UI, then click License.
4. Other settings, such as ssl, http-proxy or wsgi_module, may be required for your environment. See Python Agent Settings for a complete list of settings.

When you instrument an application using `pyagent run`, you pass the configuration file path as a parameter to the `pyagent run` command. In other deployments set the `APPD_CONFIG_FILE` environment variable as illustrated below in the samples for uWSGI with Emperor and Apache with mod_wsgi.

Instrument the Application

Which instrumentation instructions to use depends on how the application is deployed, from these deployment options:

- `pyagent run`
- uWSGI with Emperor
- Apache with mod_wsgi

Irrespective of your Python environment, if you built your application using PasteDeploy, you can install the Python Agent by modifying the PasteDeploy configuration. See PasteDeploy.

**pyagent run**

If you can control the way your WSGI server is invoked, you can instrument the application using `pyagent run`. This command runs your WSGI server with the Python agent enabled. This option is generally possible if you use a process launcher/manager that takes a command to execute. For example, frameworks managed by Supervisor, uWSGI without Emperor, init.d scripts, and so on.

To use the `pyagent run` command, prepend to your existing application run command the `pyagent run` command, passing the AppDynamics configuration file described in Configure the Agent as a parameter.

Do not overwrite `PYTHONPATH` for any reason. Doing so will prevent the `pyagent run` command from loading the agent. If you need to add to `PYTHONPATH`, use the pythonpath configuration variable. For example, these commands add `/foo` and `/bar` to the `PYTHONPATH` instead of overwriting it:

Correct way to add to `PYTHONPATH`:

```python
pythonpath = /foo
pythonpath = /bar
```

Do not add the values to `PYTHONPATH` using the following syntax. This is the wrong way to add to `PYTHONPATH`:

```python
env = PYTHONPATH=/foo:/bar
```

Using supervisorctl

If you use `supervisorctl`, after updating your Supervisor configuration you must use the `supervisorctl reload` command to have the Python agent loaded. Supervisor does not re-read its configuration files when you use the `supervisorctl restart` command.

To verify that the agent was loaded, look for the Python agent log file. Its default location is `/tmp/appd/logs/<app_name>-<node_name>.log`. For example, if your application name is `myapp` and your node name is `mynode` as specified in the agent configuration file, and you have not changed the location of the log file, the log file will be `/tmp/appd/logs/myapp-mynode.log`.

If the log file exists, the agent was loaded. If the log file does not exist, the agent was not loaded, in which case you should try reloading the Supervisor configuration with `supervisorctl reload`.

Django and Flask

If your framework is Django or Flask, simply prepend `pyagent run` to your run command. For example, if your current run command looks like this:
gunicorn -w 8 -b '0.0.0.0:9000' example.app:application

Replace it with the following:

```
pyagent run -c <path_to_appdynamics_config_file> -- gunicorn -w 8 -b '0.0.0.0:9000' example.app:application
```

Other Pure Python WSGI-Based Frameworks

If you use a WSGI-based framework that is not Django or Flask:

1. In the AppDynamics configuration file, specify your WSGI application by setting the APPD_WSGI_MODULE directive to point to your app module. See Python Agent Settings.
2. Prepend `pyagent run` to your run command.
3. Run the AppDynamics-generated application.

For example, if your run command looks like this:

```
gunicorn -w 8 -b '0.0.0.0:9000' example.app:application
```

Replace it with these two commands:

```
pyagent run -c /path/to/appdynamics.cfg -- gunicorn -w 8 -b '0.0.0.0:9000' appdynamics.scripts.wsgi:application
```

**uWSGI with Emperor**

If your environment is uWSGI with Emperor, you need to modify your WSGI configuration files and then manually launch the proxy.

uWSGI Emperor is a process manager specific to the uWSGI server. It does not allow you to control how the uWSGI processes that it manages are launched and therefore cannot be used with the `pyagent run` command.


**To instrument an application for uWSGI with Emperor:**

1. Create the configuration file described in Configure the Agent.
2. Modify the uWSGI configuration file. Do one of the following, depending on whether the configuration uses a module directive or a wsgi-file directive:
   - **Module Directive**
     
     If the uWSGI configuration has a module directive such as the following:

     ```
     module = yourcompany.sample:app
     ```

     modify that configuration by changing the module setting and adding the APPD_WSGI_MODULE and APPD_CONFIG_FILE settings to look like this, assuming that you have stored the configuration file in `/etc/appdynamics.cfg`:
2. Create the configuration file described in Configure the Agent.

3. Modify the mod_wsgi configuration file.
   If the mod_wsgi configuration file has an entry like this:

   ```
   WSGIScriptAlias /books /var/www/acme/bookstore/app.wsgi
   WSGICallableObject application
   ```

   modify it to look like this, assuming that you have stored the configuration file in /etc/appdynamics.cfg:

```python
env = APPD_CONFIG_FILE=/etc/appdynamics.cfg
env = APPD_WSGI_MODULE=yourcompany.sample:app
module = appdynamics.scripts.wsgi:application
```

**WSGI-File Directive**

If the uWSGI configuration has a `wsgi-file` directive:

```python
wsgi-file = /var/www/yourcompany/sample.py
callable = app
```

Modify the configuration to look like the following, assuming you have stored the configuration file in /etc/appdynamics.cfg:

```python
env = APPD_CONFIG_FILE=/etc/appdynamics.cfg
env = APPD_WSGI_SCRIPT_ALIAS=/var/www/yourcompany/sample.py
env = APPD_WSGI_CALLABLE_OBJECT=app
module = appdynamics.scripts.wsgi
```

3. Before running any traffic through the instrumented application, manually launch the proxy by executing:

```bash
pyagent proxy start
```
SetEnv APPD_CONFIG_FILE /etc/appdynamics.cfg
SetEnv APPD_WSGI_MODULE acme.bookstore:app
WSGIScriptAlias /books
/<path_to_virtualenv>/lib/python2.7/site-packages/appdynamics/scripts/wsgi.py

3. Before running any traffic through the instrumented app, manually launch the proxy by executing:

```
pyagent proxy start
```

**PasteDeploy**

You can instrument a Python application built with PasteDeploy by modifying your PasteDeploy configuration to use a composite factory supplied by AppDynamics. This feature can be used to instrument applications described by the other deployment options if they were built with PasteDeploy.

The AppDynamics composite factory is named `egg:appdynamics#instrument`. It requires a parameter named `target` that points to the application to the original application and the full path to the `APPD_settings`.

To instrument an application built with PasteDeploy:

1. Manually launch the AppDynamics proxy:

```
pyagent proxy start
```

2. In the PasteDeploy configuration file, rename the existing composite to a unique name. For example, if the existing composite configuration for an application named `metadata` is:

```
[composite:metadata]
use = egg:Paste#urlmap
/: meta
```

you could rename it:

```
[composite:_orig_metadata]
use = egg:Paste#urlmap
/: meta
```

3. Create a new composite section for the `metadata` application above the original one that you just renamed, as follows:
   a. Give the name of the old renamed application to the new composite application.
   b. Configure it to use the AppDynamics composite factory: `egg:appdynamics#instrument`.
   c. Set its target to the renamed application.
   d. Set the AppDynamics configuration file environment variable, `APPD_CONFIG_FILE`, to the path of your configuration file. For example:
[composite:metadata]
use = egg:appdynamics#instrument
target = orig_metadata
APPD_CONFIG_FILE = /etc/appdynamics.cfg

You can also set other APPD configuration variables here. For example, APPD_LOGS_DIR=/var/log/appdynamics.

4. Restart the application.
Start the Python Agent Proxy Manually

If you use the `pyagent run` command to instrument your application, the proxy is automatically started when you start the agent. However, you may need to arrange to start the proxy separately from the agent. The most likely reason you would need to do this is because your Python environment uses uWSGI with Emperor or Apache with mod_wsgi. You can do so using the `pyagent run` command described here. Only use this command only if you need to launch the proxy manually.

Directory Configuration Setting

The command you use to start the proxy needs to read the directory configuration setting, APPD_DIR, the base directory for the AppDynamics Python Agent configuration. Make sure it is correctly set. The default is `/tmp/appd/`.

You can make this setting available to the proxy in any of the following ways:

- Setting the environment variable `APPD_DIR` manually.
- Setting the `APPD_CONFIG_FILE` environment variable to point to the configuration file and then setting the base directory in the configuration file. See Install the Python Agent.
- Passing in the path to the configuration file to the pyagent proxy command using `-c (--config-file)` option as illustrated below.

Start, Stop and Restart Options for the pyagent proxy

To start, stop, or restart the proxy, use the `pyagent run` command, passing the desired operation and any options. The options include:

- `--debug (-d)`: starts the proxy in debugging mode
- `--no-watchdog`: disables the proxy watchdog (not recommended)
- `--config-file (-c)`: path to the Python Agent configuration file

Usage:

```
pyagent proxy start|stop|restart start_options  -- [jvmOption [jvmOption [...] ]
```

Examples:

```
pyagent proxy start -c appdynamics.config -d
pyagent proxy restart -c appdynamics.config -d
pyagent proxy stop -c appdynamics.config
```
Upgrade the Python Agent

If you are upgrading both the Controller and agents, first upgrade the Controller and then upgrade the agents.

To upgrade the Python to the latest version of the 4.5 agent, run this command:

```
pip install -U appdynamics<4.5
```

The upgrade will take effect the next time you restart the agent.
Uninstall the Python Agent

You can remove Python Agent instrumentation from your application by reverting deployment script changes made when installing the agent to remove all pyagent commands and APPD environment variables.

Removing instrumentation is generally sufficient for most purposes when it comes to uninstalling the Python Agent, however, you can additionally remove the Python Agent packages using pip.

Because pip install appdynamics installed a separate dependencies package as well as the agent, two pip uninstall commands are required:

```
pip uninstall appdynamics
pip uninstall <python_agent_package>
```

If you do not know which package is installed, you can find out using pip freeze and grep:

```
pip freeze | grep appdynamics
```
Python Agent Settings

On this page:
- [agent]
- [wsgi]
- [log]
- [controller]
- [controller:http-proxy]
- [eum]
- [services:snapshot]
- [services:transaction-monitor]

You can configure operating settings for the Python Agent using a configuration file or by setting environment variables in the application environment. This topic lists the Python Agent settings.

See ‘Configure the Agent’ in Install the Python Agent for information about the configuration file.

[agent]

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
<th>Example</th>
<th>Default</th>
<th>Environment Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>app</td>
<td>App Name</td>
<td>MyApp</td>
<td>Required</td>
<td>APPD_APP_NAME</td>
</tr>
<tr>
<td>tier</td>
<td>Tier Name</td>
<td>web-fe</td>
<td>Required</td>
<td>APPD_TIER_NAME</td>
</tr>
<tr>
<td>node</td>
<td>Node Name</td>
<td>web-fel</td>
<td>Required</td>
<td>APPD_NODE_NAME</td>
</tr>
<tr>
<td>dir</td>
<td>Base directory for files related to the AppDynamics agent</td>
<td>/mysite/appd/agent/python/</td>
<td>/tmp/appd/</td>
<td>APPD_DIR</td>
</tr>
</tbody>
</table>

[wsgi]

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
<th>Example</th>
<th>Default</th>
<th>Environment Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>script</td>
<td>Path to WSGI script file</td>
<td>/var/www/acme/bookstore.py</td>
<td>n/a</td>
<td>APPD_WSGI_SCRIPT_ALIAS</td>
</tr>
<tr>
<td>callable</td>
<td>Name of WSGI callable in script/module</td>
<td>app</td>
<td>application</td>
<td>APPD_WSGI_CALLABLE_OBJECT</td>
</tr>
<tr>
<td>module</td>
<td>Fully-qualified name of app module</td>
<td>acme.bookstore:app</td>
<td>n/a</td>
<td>APPD_WSGI_MODULE</td>
</tr>
</tbody>
</table>

If both the script and module directives are specified, the module directive takes precedence.

The module value may be the fully-qualified name of a module, or it may be the fully-qualified name of a module followed by a colon and the name of the WSGI callable in that module. In the latter form, the module directive overrides the callable directive.

Both the long-form of the module directive and the callable directive may take either the name of a symbol, or the name of a symbol followed by an empty pair of parentheses. In the latter form, the callable is taken to be the result of calling the callable specified by the directive. This latter form can be used with Django, for example:

```python
module = django.core.handlers.wsgi:WSGIHandler()
```

[log]

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
<th>Example</th>
<th>Default</th>
<th>Environment Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>dir</td>
<td>The directory to write proxy and agent logs to</td>
<td>/var/log/appdynamics</td>
<td>/tmp/appd/logs</td>
<td>APPD_LOGS_DIR</td>
</tr>
</tbody>
</table>
The level to log at one of: warning, info, or debug

debugging
On to write DEBUG level logs to stderr and log files

APPD_LOGGING_LEVEL
APPD_DEBUG_LOG

[controller]
This section specifies configuration for the AppDynamics controller.

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
<th>Example</th>
<th>Default</th>
<th>Environment Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>host</td>
<td>Controller host</td>
<td>mycontroller.example.org</td>
<td><strong>Required</strong></td>
<td>APPD_CONTROLLER_HOST</td>
</tr>
<tr>
<td>port</td>
<td>Controller port</td>
<td>9000</td>
<td>8090 for HTTP on-prem 443 for HTTPS, in which case SSL must also be set</td>
<td>APPD_CONTROLLER_PORT</td>
</tr>
<tr>
<td>ssl</td>
<td>Is SSL set be used to talk to the controller? on or off</td>
<td>on</td>
<td>off</td>
<td>APPD_SSL_ENABLED</td>
</tr>
<tr>
<td>account</td>
<td>AppDynamics controller account</td>
<td>user1</td>
<td>For a single-tenant controller defaults to customer1, Otherwise required.</td>
<td>APPD_ACCOUNT_NAME</td>
</tr>
<tr>
<td>accesskey</td>
<td>AppDynamics controller account access key</td>
<td>XC6v2n8m2$543</td>
<td><strong>Required</strong>. To find your account name and access key, click the gear ( ) icon in the upper right corner of the AppDynamics UI, then click License.</td>
<td>APPD_ACCOUNT_ACCESS_KEY</td>
</tr>
</tbody>
</table>

[controller:http-proxy]
If you need to use an HTTP proxy to talk to your controller, use this section to configure the HTTP proxy.

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
<th>Example</th>
<th>Default</th>
<th>Environment Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>host</td>
<td>HTTP proxy host</td>
<td>proxy.example.org</td>
<td>n/a</td>
<td>APPD_HTTP_PROXY_HOST</td>
</tr>
<tr>
<td>port</td>
<td>HTTP proxy port</td>
<td>8090</td>
<td>80</td>
<td>APPD_HTTP_PROXY_PORT</td>
</tr>
<tr>
<td>user</td>
<td>HTTP proxy user</td>
<td>proxyuser</td>
<td>n/a</td>
<td>APPD_HTTP_PROXY_USER</td>
</tr>
<tr>
<td>password-file</td>
<td>HTTP proxy password file</td>
<td>/etc/http-proxy.passwd</td>
<td>n/a</td>
<td>APPD_HTTP_PROXY_PASSWORD_FILE</td>
</tr>
</tbody>
</table>

[eum]

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
<th>Example</th>
<th>Default</th>
<th>Environment Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>disable-cookie</td>
<td>If set, the agent does not add EUM correlation data to WSGI response headers.</td>
<td>on</td>
<td>off</td>
<td>APPD_EUM_DISABLE_COOKIE</td>
</tr>
</tbody>
</table>
### user-agent-whitelist

If specified overwrites the default whitelist for user agent added as EUM correlation data headers. Use this setting to specify alternate user agents as a comma separated list. Use '*' to allow all user agents.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>'iPad, Android'</td>
<td>'Mozilla, Opera, WebKit, Nokia'</td>
</tr>
</tbody>
</table>

APPD_EUM_USER_AGENT_WHITELIST

### [services:snapshot]

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
<th>Example</th>
<th>Default</th>
<th>Environment Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>exit-call-details-length</td>
<td>Specifies the number of characters in the details string describing exit calls in transaction snapshots.</td>
<td>200</td>
<td>100</td>
<td>APPD_EXIT_CALL_DETAILS_LENGTH</td>
</tr>
</tbody>
</table>

### [services:transaction-monitor]

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
<th>Example</th>
<th>Default</th>
<th>Environment Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>bt-max-duration-ms</td>
<td>Maximum duration of a business transaction in milliseconds</td>
<td>60000</td>
<td>120000</td>
<td>BT_MAX_DURATIONS_MS</td>
</tr>
</tbody>
</table>
Python Agent Debugging and Logging

On this page:

- Logging Settings
- Logging Levels
- Location and Names of Agent Log Files
- Location of Proxy Logs

Related pages:

- Python Agent
- Python Agent Settings

Logging Settings

Logging is configured in the logging section of the AppDynamics configuration file. See Python Agent Settings.

Logging Levels

Set the logging level for the amount of detail that you want to see: INFO, WARNING or DEBUG. The default is level is WARNING.

For maximum debugging information, set the logging level to DEBUG. Debug-level logs show the information that the Python Agent is sending to the proxy, which can be useful for debugging the agent.

For quick debugging, set the debugging configuration option to on. This causes debug-level logs to be written to both stderr and log files.

This is the behavior you can expect with various combinations of logging level and debug settings:

The various combinations of logging level and debug settings produce behavior as follows:

- **level=info, debugging=off:** Info-level logs are written to the log file
- **level=warning, debugging=off:** Warning-level logs are written to the log file
- **level=debug, debugging=off:** Debug-level logs are written to the log file
- **level=info, debugging=on:** Debug-level logs are written to the log file and to stderr
- **level=warning, debugging=on:** Debug-level logs are written to the log file and to stderr
- **level=debug, debugging=on:** Debug-level logs are written to the log file and to stderr

The Proxy logging level is separate and is not affected by the configuration settings of the Agent logging levels. Proxy logging level is "info" by default. Typically, you can redirect that output to a file if desired. In case if you wish to increase the logging level output, you must do so by starting the Proxy manually. First assign the new logging level within the Proxy's configuration file (proxy/conf/logging/log4j.xml) and then invoke the Proxy task as described above.

Note

There is no way to automatically adjust the Proxy logging level from the Agent configuration file settings.

Location and Names of Agent Log Files

Agent logs are written in the directory specified by the dir directive, APPD_LOGS_DIR in the logging section of the configuration file.

For each node, the log is output to a file named APP_NAME-<NODE_NAME>.log. These files live in the log dir directory, APPD_LOGS_DIR.

The agent creates and rotates a maximum of five log files. Maximum log size is 20MB, which gives you a maximum of the most recent 100MB of log data at one time.

For example:

1. myApp-myNode.log
2. myApp-myNode.log.1
3. myApp-myNode.log.2
4. myApp-myNode.log.3
5. myApp-myNode.log.4

Location of Proxy Logs

The proxy logs are always written to dir APPD_LOGS_DIR, which defaults to /tmp/appd/logs, APPD_DIR/logs.
Python Agent API

Related pages:
- Python Agent API Guide
- Python Agent API Reference

The Python Agent APIs provide facilities to support the following:

- Custom business transactions
- Custom exit points
- Custom data collectors
The Python Agent APIs:

- Let you programmatically define custom business transactions that would not automatically be detected by the agent.
- Let you create custom exit calls to discover and monitor backends and that are not automatically detected by the agent.

If your instrumented application starts up and you do not see all the business transactions or backends that you expect to see, first check Python Supported Environments to see if your framework or exit point is supported. If not, you can use the APIs to instrument your code manually.

**Custom Business Transactions**

You can use the `start_bt()` and `end_bt()` methods to surround the code that you want to monitor as a custom business transaction.

Or you can use the "bt" context manager. Consider using the `bt` context manager where you start and end the business transaction in the same code. For example, where you can wrap the whole business transaction in a `with` statement.

For example, given the code:

```python
setup()
do_work()
teardown()
```

you want to report `do_work()` as a business transaction.

**Use `start_bt()` and `end_bt()`**

This example uses `start_bt()` and `end_bt()` to create a business transaction named `do_work`.

```python
from appdynamics.agent import api as appd
setup()

bt_handle = appd.start_bt('do work')
try:
    do_work()
except Exception as exc:
    raise
finally:
    appd.end_bt(bt_handle, exc)
teardown()
```
Use bt context manager

If the business transaction starts and ends in the same context, you can use the `bt` context manager instead. This is simpler:

```python
setup()
with bt('do work'):
    do_work()
tear Down()
```

**Custom Exit Calls**

You can use the `start_exit_call()` and `end_exit_call()` methods to create a custom exit call from a specific business transaction to a backend that the Python Agent does not automatically detect.

The business transaction must be a custom business transaction.

If you want to make a custom exit call from a business transaction that is normally automatically detected, you can exclude that business transaction to prevent it from being automatically detected and then create it as a custom business transaction. This enables you to get the BHandle that you need to create the custom exit call. See Configure Python Web Custom Match and Exclude Rules for information on excluding a business transaction.

Given the code:

```python
try:
    db = custom_db.connect(host='financials-lb', port=3456)
    all_employees = db.query_path('/financials/employees')
    individual_contributors = all_employees.filter(lambda r:
        r.level < 3)
    salaries_by_dept = individual_contributors.sum(value='salary',
        group='dept', as='total')
    for dept, total in salaries_by_dept.extract('dept', 'total'):
        report_salary_data(dept, total)
```

You want to send the query via an exit call to a proprietary database.

You want the database to be labeled Financials Database in the Controller UI.

You want the backend properties that appear in the backend dashboard to appear as:

- **Host**: financials-lb
- **Port**: 3456
- **Vendor**: custom db

The following examples assume you are wrapping the exit call in a custom business transaction named department rollup, created in another part of your code.

**Use start_exit_call() and end_exit_call()**
This example uses `start_exit_call()` and `end_exit_call()`.

```python
from appdynamics.agent import api as appd
appd.init()

# Set the identifying properties
FINANCIALS_ID_PROPS = {'Host': 'financials-lb', 'Port': 3456, 'Vendor': 'custom db'}

with appd.bt('department rollup') as bt_handle:
    # Start the exit call
    exit_call = appd.start_exit_call(bt_handle, appd.EXIT_DB, 'Financials Database', FINANCIALS_ID_PROPS)
    exc = None

    try:
        db = custom_db.connect(host='financials-lb', port=3456)
        all_employees = db.query_path('/financials/employees')
        individual_contributors = all_employees.filter(lambda r: r.level < 3)
        salaries_by_dept = individual_contributors.sum(value='salary', group='dept', as='total')

        for dept, total in salaries_by_dept.extract('dept', 'total'):
            report_salary_data(dept, total)
            except Exception as exc:
                raise  # Assuming something above handles exceptions for you

    finally:
        # End the exit call
        end_exit_call(exit_call, exc)
```

**Use exit_call context manager**

If the business transaction starts and ends in the same context, you can use the simpler `exit_call` context manager instead.
from appdynamics.agent import api as appd
appd.init()

with appd.bt('department rollup') as bt_handle:
    with appd.exit_call(bt_handle, appd.EXIT_DB, 'Financials Database', FINANCIALS_ID_PROPS):
        db = custom_db.connect(host='financials-lb', port=3456)
        all_employees = db.query_path('/financials/employees')
        individual_contributors = all_employees.filter(lambda r:
            r.level < 3)
        salaries_by_dept =
        individual_contributors.sum(value='salary', group='dept',
        as='total')

        for dept, total in salaries_by_dept.extract('dept',
        'total'):
            report_salary_data(dept, total)

The next example starts a custom exit call to a Cassandra backend from a business transaction that was auto-detected by the Python Agent default Flask instrumentation. It uses the Flask import feature to get the request object which it passes to appd_get_active_bt_handle().

Get bt handle using the flask request context

from flask import request
from appdynamics.agent import api as appd

@app.route('/metrics/recent')
def metrics_recent():
    bt = appd.get_active_bt_handle(request)  # Get the active BT from the Flask request object
    with appd.exit_call(bt, appd.EXIT_DB, 'cassandra time-series',
        {'VENDOR': 'Cassandra', 'SERVER POOL': '10.0.0.1'}):
        load_recent_data_from_cassandra()

Other supported frameworks have different mechanisms for getting the request object.
The Python Agent APIs provide facilities to support:

- custom business transactions
- custom exit points
- custom data collectors

List of the APIs:

- `init(environ=None, timeout_ms=NO_TIMEOUT)`
- `shutdown(timeout_ms=None)`
- `start_bt(name, correlation_header=None)`
- `end_bt(bt_handle, exc=None)`
- `add_snapshot_data(bt_handle, key, value)`
- `get_active_bt_handle(request)`
- `start_exit_call(bt_handle, exit_type, display_name, identifying_properties, optional_properties=None)`
- `end_exit_call(exit_call_handle, exc=None)`
- `make_correlation_header(bt_handle, exit_call_handle)`
- `bt(name, correlation_header=None)`
- `exit_call(bt_handle, exit_type, display_name, identifying_properties, optional_properties=None)`

**Access to the APIs**

Before your app can access the Python Agent APIs, install the Python Agent using:

```
pip install appdynamics
```

Then at the top of the instrumented application add:

```
from appdynamics.agent import api as appd
```

**Start and Stop Agent**

`init(environ=None, timeout_ms=NO_TIMEOUT)`

Initializes the Python Agent.

Call this function at the very beginning of your instrumented application, preferably before any other imports and before creating any other threads.

Agent initialization occurs asynchronously. `init()` returns immediately if no timeout is specified but it is possible that the agent will not immediately be ready to report business transactions. In this case, the `start_bt()` method will return `None`.

This function will never raise an exception.
Returns True if the agent is properly configured, False if the agent is not properly configured.

To be properly configured the agent configuration file must contain the minimal settings and possibly some additional settings if they are required for your environment. See Configure the Agent in Install the Python Agent for the minimal settings and Python Agent Settings for the complete list.

environ: dict, optional

If specified, a dictionary of environment variables to use to override the agent's configuration, derived from the actual OS environment variables.

timeout_ms: int or None, optional

Timeout interval in milliseconds.

By default, init() returns immediately, even if the Python agent has not received its configuration from the controller. If timeout_ms is None, init() blocks until the agent is properly configured. Otherwise, init() waits for up to timeout_ms for agent to become properly configured.

shutdown(timeout_ms=None)

Shuts down the Python Agent. The agent stops reporting metrics to the Controller.

This function ends all active business transactions and waits for them to finish reporting to the controller before returning.

timeout_ms: int, optional

By default, this function waits until all pending business transactions have been reported before returning. If timeout_ms is set, this function will return after timeout_ms, regardless of whether all business transactions have been reported.

Business Transaction Management

start_bt(name, correlation_header=None)

Starts a business transaction of the specified name.

There can only be one active business transaction per thread. Attempts to start subsequent business transactions in the same thread will return None.

Returns a BtHandle on starting the transaction or None if no transaction was started.

name: str

Name of the business transaction being started. The following characters cannot be used in business transaction names: { } [ ] | & ;

correlation_header: str, optional

If specified, a correlation header that has been generated by another AppDynamics agent and passed to this agent as a string. A correlation header provides information to enable this transaction to correlate with a transaction that is upstream to this one.

end_bt(bt_handle, exc=None)

Ends the business transaction identified by bt_handle.

bt_handle: BtHandle

Identifies the business transaction being ended. The handle is returned by start_bt().

exc: Exception, optional

If an exception occurred during processing this business transaction that you have caught with a try-except block, and you wish to report the exception as part of the business transaction, pass the exception as the exc.

add_snapshot_data(bt_handle, key, value)

Attaches custom data to transaction snapshots generated for the business transaction identified by bt_handle.
The custom data is exposed in the **USER DATA** tab of the transaction snapshot details in the Controller UI.

The custom data is added to the business transaction but reported only when a transaction snapshot is generated. See Transaction Snapshots for information on when transaction snapshots are generated.

**bt_handle: BtHandle**

Identifies the business transaction to which custom data is added. The handle is returned by `start_bt()`.

**key: bytes or unicode**

Name of the data item to be reported in the snapshot.

If passed as bytes, the buffer must represent a UTF-8 encoded string.

**value: any**

Value of the data.

If passed as bytes, the data is treated as a UTF-8 encoded string. If passed as unicode, it is directly reported.

All other objects are converted to `str(value)`.

**get_active_bt_handle(request)**

Returns a BtHandle to the business transaction associated with the request object or None if no such business transaction was found.

This is useful for passing a business transaction handle to another API function, such as `start_exit_call()` when the business transaction was created by the default Python Agent instrumentation rather than through the APIs.

**request object**

The request object associated with an active business transaction. Varies depending on the framework in which the business transaction was started.

See the documentation for the applicable framework to find out how to get the request object.

<table>
<thead>
<tr>
<th>Framework</th>
<th>Documentation for the Request Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Django</td>
<td><a href="https://docs.djangoproject.com/en/1.8/ref/request-response/">https://docs.djangoproject.com/en/1.8/ref/request-response/</a></td>
</tr>
<tr>
<td>WSGI environ</td>
<td><a href="https://www.python.org/dev/peps/pep-0333/">https://www.python.org/dev/peps/pep-0333/</a></td>
</tr>
</tbody>
</table>

**Exit Call Management**

**start_exit_call(bt_handle, exit_type, display_name, identifying_properties, optional_properties=None)**

Starts a custom exit call from within the specified business transaction.

There can be only one active exit call per business transaction. Attempts to start subsequent exit calls will return None.

Returns an ExitCallHandle that identifies this exit call.

**bt_handle: BtHandle**

Identifies the business transaction making the exit call. Handle is returned from `start_bt()`.

**exit_type: int**

The type of exit call. Valid values are:

- `appd.EXIT_HTTP`
- `appd.EXIT_DB`
**display_name:** `str`
Name used to identify this exit call in the controller. This is the label that will be used to display the exit call in the AppDynamics UI.

**identifying_properties:** `dict`
A dictionary of name/value pairs that uniquely identify the downstream component being called.
In the Controller UI, these properties are visible in the upper right panel of the backend dashboards.
The key of the dict is the name of the property; the value is the value of the property.

**operation:** `str`, optional
A string describing the operation that is being performed, such as the URL of an HTTP request or an SQL query.

**optional_properties:** `dict`, optional
A dictionary of name/value pairs that identify additional properties for this exit call. In the Controller UI, these properties are visible in the Exit Calls and Async Activities modal in the snapshot drill-down pane.

`end_exit_call(exit_call_handle, exc=None)`
Ends the exit call identified by `exit_call_handle`.

**exit_call_handle:** `ExitCallHandle`
Identifies the exit call being ended. The handle is returned from `start_exit_call()`.

**exc:** Exception
If an exception occurred during the exit call that you have caught with a `try-except` block, and you wish to report the exception as part of the exit call, pass the exception as the `exc` argument.

`make_correlation_header(bt_handle, exit_call_handle)`
Make a correlation header for a custom exit call.

If you are performing custom exit calls to other instrumented tiers, adding a correlation header allows continuing business transactions on the downstream tier.

It is up to you to send the header, as well as parse it at the other end and pass it to `start_bt()`.

Returns a tuple of header name and header value, or `None` if correlation is disabled.

**bt_handle:** `BtHandle`
A handle identifying the business transaction.

**exit_call_handle:** `ExitCallHandle`
A handle identifying the exit call.

**Context Managers**

`bt(name, correlation_header=None)`
Context manager for reporting some work as a business transaction. Yields a `BtHandle`.

If you need to start and end the transaction in different places in the code, use `start_bt()` and `end_bt()`.

`exit_call(bt_handle, exit_type, display_name, identifying_properties, optional_properties=None)`
Context manager for adding exit calls to a business transaction. Yields an `ExitCallHandle`. 
If you need to start and end exit calls in different places in your code, you can use `start_exit_call()` and `end_exit_call()`.

**HTTP Status Codes**

HTTP status codes will not cause error transactions when you use the using the Python Agent API to instrument your application, regardless of the error detection configuration.
Serverless APM for AWS Lambda

AppDynamics Serverless Application Performance Monitoring (Serverless APM) for AWS Lambda gives you visibility into the performance of your application's components that run as functions on serverless compute environments.

Serverless APM gives you an end-to-end view of applications, including microservices that are implemented as serverless functions, through business transaction correlation. Serverless APM correlates business transactions between AWS Lambda functions and:

- components instrumented with other AppDynamics app agents
- end-user devices instrumented with AppDynamics End User Monitoring (EUM) agents

Additionally, Serverless APM correlates business transactions through serverless functions, such as an AWS Lambda that invokes another AWS Lambda function.

Serverless APM in the Controller

When business applications have instrumented serverless functions, the Controller experience differs slightly in flow maps, dashboards and metric browser pages, and health rules.

Flow Maps

Flow maps are a dynamic visual representation of the components and activities of your monitored application environment. When business applications have serverless functions, you cannot view the nodes within flow map tiers because of the opaque nature of serverless platforms. A serverless icon replaces the node count in the application flow map.

The screenshot depicts an application composed of AWS Lambda tiers:

Dashboards and Metric Browser Pages

You can view AWS Lambda functions on your application dashboards and metrics pages. All functionality is identical to that which you would get with any other tier type, with the exception of node-level granularity. AWS Lambda tiers do not offer node-level dashboards or metrics because serverless platform runtime instances spin up and down on demand.

Health Rules

When you configure a health rule for an application comprised of serverless functions, you can choose to monitor the serverless tiers or business transactions that originate in or flow through the serverless functions. See Configure Health Rules.
Get Started

Serverless APM requires a subscription through AWS Marketplace. See Subscribe to Serverless APM for AWS Lambda to get started. If you have already subscribed to Serverless APM, see Instrument Serverless APM for AWS Lambda.

Amazon Web Services, the AWS logo, AWS, and any other AWS Marks used in these materials are trademarks of Amazon.com, Inc. or its affiliates in the United States and/or other countries.
## Subscribe to Serverless APM for AWS Lambda

This topic explains how to subscribe to Serverless APM for AWS Lambda. AppDynamics provides Serverless APM functionality in the form of a tracer library. Learn more about Serverless APM.

### Requirements

Serverless APM for AWS Lambda supports AWS Lambda functions implemented in Java. A subscription to Serverless APM requires:

- AppDynamics SaaS Controller, version 4.5.11 or later.
- An active AppDynamics Pro account.
- Your Controller access key. This requires you to be an AppDynamics Account Owner or have Administrator privileges.
- AWS Identity and Access Management (IAM) role with permission to one of the following policies:
  - AWSMarketplaceManageSubscriptions
  - AWSMarketplaceFullAccess

### Subscribe to Serverless APM

Subscribe to AppDynamics Serverless APM for AWS Lambda through AWS Marketplace.

1. Navigate to AWS Marketplace.
2. Sign in to your AWS account.
4. Click AppDynamics Serverless APM for Lambda.
5. Click Continue to Subscribe.
6. Click Subscribe. A message, Congratulations! You are now subscribed!, appears confirming your subscription.
   - If the Subscribe button is unavailable:
     a. Navigate to Having issues signing up for your product?
     b. Select click here.
     c. Proceed to step eight.
7. Click Set Up Your Account.
8. Enter your SaaS Controller URL and Controller Access Key (Default), then click Continue.
   - The SaaS Controller URL must be a full path URL with no whitespace or trailing slashes.
   - Registration may take up to 30 seconds.
   - You can only access the registration page through AWS Marketplace.
   - Do not close the registration page while it attempts to connect to the Controller. If you close the registration page, you will have to subscribe again through AWS Marketplace.
9. Click Getting Started to learn how to instrument Serverless APM with your AWS Lambda functions.

### Subscribe Multiple Controllers

You can subscribe multiple Controllers to Serverless APM, as long as each Controller meets the requirements described on this page. Repeat the steps listed in Subscribe to Serverless APM for AWS Lambda for each Controller.

---

**Need help?** Technical support is available through AppDynamics Support. All billing related questions are handled by AWS Marketplace.
Amazon Web Services, the AWS logo, AWS, and any other AWS Marks used in these materials are trademarks of Amazon.com, Inc. or its affiliates in the United States and/or other countries.
View the Serverless APM Subscription

On this page:
- Verify Your Serverless APM Subscription
- Check Your Serverless APM Subscription Status

After you have subscribed to Serverless APM for AWS Lambda, you can view your subscription and its status through your AppDynamics account.

Verify Your Serverless APM Subscription

You can verify your subscription to Serverless APM for AWS Lambda in your AppDynamics account.

2. Sign in to your account.
3. Select the Controller you subscribed to Serverless APM.
4. Confirm that Serverless APM for AWS Lambda subscription is activated in the License usage tab.

Under the Usage column, you can view the date your subscription began, and access billing information on the AWS Console. Serverless APM for AWS Lambda matches the Usage Period of your APM license, regardless of the Serverless APM subscription status.

Check Your Serverless APM Subscription Status

You can check your Serverless APM for AWS Lambda subscription status in your AppDynamics account. To access your account see the instructions in Verify Your Serverless APM Subscription. Click the License details tab and navigate to AWS Lambda Status.

Active status indicates you are currently subscribed to Serverless APM for AWS Lambda. You can see the date your subscription began and the encrypted AWS Customer ID of the user who subscribed.

Inactive status indicates you have canceled your subscription to Serverless APM for AWS Lambda for all Controllers. You can see the cancellation date and the encrypted AWS Customer ID of the user who unsubscribed.
Amazon Web Services, the AWS logo, AWS, and any other AWS Marks used in these materials are trademarks of Amazon.com, Inc. or its affiliates in the United States and/or other countries.
Troubleshoot the Serverless APM Subscription

On this page:
- Subscribe Button Unavailable
- Error Messages
  - Upgrade to Enable Serverless APM for AWS Lambda
  - Incorrect Credentials
  - Controller Already Registered

This document contains links to AWS documentation. AppDynamics makes no representation as to the accuracy of Amazon documentation because Amazon controls its own documentation.

This page provides troubleshooting information for issues that may arise when subscribing to Serverless APM for AWS Lambda.

**Subscribe Button Unavailable**

After an AWS user clicks **Subscribe** on the AWS product listing, the button is unavailable for future visits to the AWS Marketplace subscription page. To complete the subscription process, see step six in **Subscribe to Serverless APM for AWS Lambda**.

**Error Messages**

You may encounter error messages while subscribing through the AWS Marketplace.

**Upgrade to Enable Serverless APM for AWS Lambda**

This error message refers to your current Controller version and AppDynamics edition. To subscribe to serverless monitoring, you need a SaaS Controller version 4.5.11 and an active AppDynamics Pro edition.

- **AppDynamics SaaS Controller, version 4.5.11 or later**
  
  Serverless APM for AWS Lambda is available for AppDynamics Pro SaaS Controllers. Your Controller must be version 4.5.11 or later. You can request an upgrade if you have a previous version of the Controller. To view your current

  AppDynamics build version in the Controller UI, go to Gear icon ( ) > About AppDynamics.

  Serverless monitoring is not available for on-premises Controllers at this time.

- **Active AppDynamics Pro Edition**

  You need an active AppDynamics Pro edition to enroll in serverless monitoring. If you are currently enrolled in a trial or unpaid edition, or if your account is inactive, purchase a license.

  To view your current AppDynamics edition and expiration date in the Controller UI, go to Gear icon ( ) > License > Account.
Incorrect Credentials

You need your SaaS tenant URL and the access key for the Controller you want to enable. You must be an AppDynamics Account Owner or Administrator to view the Controller access key. To find your Controller access key, see Agent-to-Controller Connections.

Controller Already Registered

You cannot register a single Controller for serverless monitoring multiple times. You can view your current Serverless APM for AWS Lambda subscriptions in your AppDynamics account.

2. Sign in to your account.
3. Select the desired Controller.
4. In the License Usage tab, find Serverless APM for AWS Lambda.

Need help? Technical support is available through AppDynamics Support. All billing related questions are handled by AWS Marketplace.

Amazon Web Services, the AWS logo, AWS, and any other AWS Marks used in these materials are trademarks of Amazon.com, Inc. or its affiliates in the United States and/or other countries.
Serverless APM Business Transaction Correlation

AppDynamics provides Serverless Application Performance Monitoring (Serverless APM) functionality in the form of a tracer library. This topic discusses how the tracer correlates business transactions.

Business Transaction Correlation Architecture

Serverless APM for AWS Lambda correlates business transactions with upstream and downstream calls. APM agents connect directly with your serverless functions to generate business transactions. End User Monitoring (EUM) agents connect outside the serverless functions to the business transaction itself.

This diagram illustrates how the tracer correlates business transactions through a serverless function to APM and EUM agents:

Business Transaction Correlation Process

Serverless APM correlates calls flowing through your AWS Lambda function with activity in upstream or downstream components. Business transaction correlation occurs through an opaque correlation string that the application passes across the wire. First, the tracer generates the correlation string at an exit call, and then the application passes the correlation string from the exit call to its endpoint. The downstream component retrieves the correlation string to continue the business transaction. This process enables Serverless APM for AWS Lambda to construct the flow map.

Correlation String Transportation
Your application needs to pass the correlation string, generated by the tracer, to correlate a business transaction between multiple services. The correlation string must be serialized and communicated alongside the application payload.

**Inbound HTTP Calls**

If you pass the correlation string through an HTTP call, the string is passed as an HTTP header using the tracer's `getCorrelationHeader` method. This method searches your code's object schema for a key called `singularityheader` in your `InputStream` object. Correlation occurs when `getCorrelationHeader` finds `singularityheader`. If the header cannot be found, the tracer creates a new business transaction.

**Other Protocols**

For protocols other than HTTP, you need to define protocol-specific transportation of the correlation string to enable business transaction correlation.

The tracer creates a new business transaction if:

- You have not arranged for the correlation string to be passed, or
- The tracer cannot find the correlation string.

To get started, see Instrument Serverless APM for AWS Lambda.

Amazon Web Services, the AWS logo, AWS, and any other AWS Marks used in these materials are trademarks of Amazon.com, Inc. or its affiliates in the United States and/or other countries.
Instrument Serverless APM for AWS Lambda

On this page:
- Before You Begin
- Instrumentation Process

Related pages:
- Set Up the Serverless APM Environment
- Instrument the Function Code
- Deploy the Functions

Serverless APM gives you an end-to-end business transaction view through correlation.

After you have subscribed to Serverless APM for AWS Lambda, you can begin instrumentation. This topic provides requirements and an overview of the instrumentation process.

Before You Begin

Ensure that your setup has the following requirements:
- Existing AWS Lambda functions implemented in Java
- Active Serverless APM for AWS Lambda subscription
- AppDynamics SaaS Controller version 4.5.11 or later

Instrumentation Process

Instrumentation of Serverless APM for AWS Lambda consists of setting up the environment, instrumenting the code, and deploying the function.

Set up the environment

1. Get the Serverless Tracer
2. Configure Environment Variables

Instrument the function code

There are two ways to instrument the tracer.

Automatic Tracer Instrumentation

1. Instantiate the Tracer
2. (Optional) Override Correlation Header Method
3. Obtain a Transaction Object
4. Create External Exit Calls

Manual Tracer Instrumentation

1. Instantiate the Tracer
2. Create a Transaction
3. Start and Stop a Transaction
4. Create External Exit Calls

Deploy the function

1. Deploy to AWS Lambda
2. Verify the Instrumentation
Set Up the Serverless APM Environment

On this page:
- Get the Serverless Tracer
- Configure Environment Variables

To instrument Serverless APM for AWS Lambda, you first need to add the Serverless Tracer to your project and configure environment variables.

Get the Serverless Tracer

AppDynamics provides Serverless APM functionality in the form of a tracer library. Express your project’s dependency on the Serverless Tracer using Apache Maven or Gradle as follows:

Maven

```
<dependency>
  <groupId>com.appdynamics</groupId>
  <artifactId>lambda-tracer</artifactId>
  <version>1.1.1363</version>
</dependency>
```

Gradle

```
dependencies {
  compile group: 'com.appdynamics', name: 'lambda-tracer', version: '1.1.1363'
}
```

If you do not use Apache Maven or Gradle, please contact AppDynamics to obtain the Serverless Tracer JAR file.

Configure Environment Variables

The following table lists the required and optional environment variables. Enter all applicable information as key-value environment variables in the AWS Management Console.

When you change an environment variable, any existing instances finish execution using the previous value. All new requests execute with the updated value.

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Required</th>
<th>Description</th>
<th>Example Value</th>
</tr>
</thead>
</table>

Copyright © AppDynamics 2012-2019
<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Required</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPDYNAMICS_ACCOUNT_NAME</td>
<td>Yes</td>
<td>The account name associated with the Controller used by your AWS Lambda function.</td>
<td>customer1</td>
</tr>
<tr>
<td>APPDYNAMICS_AGENT_ACCOUNT_ACCESS_KEY</td>
<td>Yes</td>
<td>The access key for your Controller. See Agent-to-Controller Connections.</td>
<td>AB1a2b3c4$123</td>
</tr>
<tr>
<td>APPDYNAMICS_APPLICATION_NAME</td>
<td>Yes</td>
<td>The name of the application where the tracer is instrumented.</td>
<td>testApp</td>
</tr>
<tr>
<td>APPDYNAMICS_CONTROLLER_HOST</td>
<td>Yes</td>
<td>The host associated with the Controller used by your AWS Lambda function. Do not include http:// or https://</td>
<td>&lt;accountname&gt;.saas.appdynamics.com</td>
</tr>
</tbody>
</table>
| APPDYNAMICS_SERVERLESS_API_ENDPOINT | Yes | The AppDynamics endpoint for the tracer to report to. Serverless API endpoints are available for the following AWS regions: Sydney, Frankfurt, and Oregon. | • Asia Pacific (Sydney): https://syd-sls-agent-api.saas.appdynamics.com/  
• EU (Frankfurt): https://fra-sls-agent-api.saas.appdynamics.com/  
| APPDYNAMICS_CONTROLLER_PORT | No | The port associated with your Controller. If the port is left blank or is invalid, defaults to 443. | 8080 |
| APPDYNAMICS_LOG_LEVEL | No | The log level for the tracer. Accepted values are DEBUG, INFO, WARN, ERROR, FATAL. Defaults to INFO if not specified. | INFO |
| APPDYNAMICS_TIER_NAME | No | The tier name of your AWS Lambda function. By default, the tier name is the name of your AWS Lambda function. Use this variable to provide another name for your tier. | serverlessTestTier |

To add environmental variables through the AWS Management Console:

1. Sign in to the AWS Management Console.
2. Go to AWS Services and open Lambda.
3. Select your AWS Lambda function.
4. Go to Environment variables and enter your function's key-value pair environment variables.
5. Click Save.

Optionally, you can set or override the environment variables in your function code. See Customize the Serverless APM Instrumentation for instructions.

The following image shows an example of all required environment variables:
Environment variables

You can define environment variables as key-value pairs that are accessible from your function code. These are useful to store configuration settings without the need to change function code. Learn more

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPDYNAMICS_ACCOUNT_NAME</td>
<td>customer1</td>
</tr>
<tr>
<td>APPDYNAMICS_APPLICATION_NAME</td>
<td>testApp</td>
</tr>
<tr>
<td>APPDYNAMICS_AGENT_ACCOUNT_ACCESS_KEY</td>
<td>A81a2b3c45123</td>
</tr>
<tr>
<td>APPDYNAMICS_CONTROLLER_HOST</td>
<td>customer1.saas.appdynamics.com</td>
</tr>
<tr>
<td>APPDYNAMICS_SERVERLESS_API_ENDPOINT</td>
<td><a href="https://pdx-sls-agent.api.saas.appdynamics.com">https://pdx-sls-agent.api.saas.appdynamics.com</a></td>
</tr>
<tr>
<td>Encryption configuration</td>
<td></td>
</tr>
</tbody>
</table>

Amazon Web Services, the AWS logo, AWS, and any other AWS Marks used in these materials are trademarks of Amazon.com, Inc. or its affiliates in the United States and/or other countries.
Instrument the Function Code

This topic discusses Serverless APM instrumentation options.

AppDynamics provides Serverless APM functionality in the form of a tracer library. You can instrument the tracer:

- Automatically, using AppDynamics class `MonitoredRequestStreamHandler`, or
- Manually, using AppDynamics method `AppDynamics.getTracer(context)`.

**Automatic Tracer Instrumentation**

You can automatically instrument the tracer if your function uses the `RequestStreamHandler` interface.

In automatic tracer instrumentation, your function inherits the tracer's configurations. Your project's class becomes a subclass of `MonitoredRequestStreamHandler`, a wrapper class provided with the tracer. Automatic tracer instrumentation performs these actions:

- Creates, starts, and stops a transaction
- Locates any correlation header
- Reports transaction errors

`RequestStreamHandler` and `RequestHandler` are predefined handlers provided by AWS. See AWS documentation for more information.

**Manual Tracer Instrumentation**

In manual tracer instrumentation, the method `AppDynamics.getTracer(context)` instantiates the tracer at the beginning of your function's entry point method. Manually instrument the tracer if:

- Your AWS Lambda function does not implement the `RequestStreamHandler` interface.
- You do not want your function to inherit configurations from the `MonitoredRequestStreamHandler`.
- Your AWS Lambda functions are multi-threaded, to ensure that behavior is defined. See Synchronize the Tracer below for more details.

Manual tracer instrumentation requires you to perform these actions:

- Instantiate the tracer
- Create, start, and stop a transaction
- Locate a correlation header
- Report transaction errors

The AWS Lambda context object in Java, `context`, is provided by AWS. See AWS documentation for more information.

**Synchronize the Tracer**

For both instrumentation options, you should synchronize any access to the tracer.

In automatic instrumentation, you need to synchronize the transaction object returned by the `getTransaction()` method.

In manual instrumentation, you need to synchronize the tracer object returned by the `getTracer()` method. Manual tracer instrumentation is required for multi-threaded use of the tracer.

**Create Exit Calls**
Create an `exitCall` object from a transaction when your function makes a call to external resources, such as to another AWS Lambda function or an outbound HTTP call. The `exitCall` object records the time spent in external exit calls, which allows AppDynamics to register and display the calls on a flow map and highlight application performance issues. Additionally, the `exitCall` object allows you to tag and follow requests when your AWS Lambda function is upstream of other instrumented tiers.

**Backend Exit Calls**

A set of identifying properties identifies a backend of any given exit type. For example, if the exit call type is HTTP, the identifying properties are the HTTP host and port. For exit calls between AWS Lambda functions, the identifying property is the name of the downstream function. You can find additional backend naming details for AppDynamics agents in [Backend Detection Rules](https://appdynamics.com). You must map these backend identifying properties and pass them to the `createExitCall` method. The `createExitCall` method obtains an `ExitCall` object, which is used to time the exit. When correlation with a downstream tier occurs, you can use the `ExitCall` object to obtain a correlation header to write into the transmitted payload.

_Amazon Web Services, the AWS logo, AWS, and any other AWS Marks used in these materials are trademarks of Amazon.com, Inc. or its affiliates in the United States and/or other countries._
Automatic Tracer Instrumentation

On this page:

- Instantiate the Tracer
- (Optional) Override Correlation Header Method
- Obtain a Transaction Object
- Create External Exit Calls

Automatic tracer instrumentation consists of the following steps:

1. Instantiate the Tracer
2. (Optional) Override Correlation Header Method
3. Obtain a Transaction Object
4. Create External Exit Calls

Instantiate the Tracer

Automatic instantiation is compatible with RequestStreamHandler interfaces. All other interfaces require manual instantiation.

RequestStreamHandler is a predefined handler provided by AWS. See AWS documentation for more information.

To automatically instantiate the tracer:

1. Make your current handler class extend AppDynamics MonitoredRequestStreamHandler.
2. Rename your handler class’s existing method handleMonitoredRequest.

The code snippet demonstrates how to instantiate the tracer automatically:
package <MyTestPackage>;
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;

//AppDynamics tracer imports
import com.appdynamics.serverless.tracers.aws.api.AppDynamics;
import com.appdynamics.serverless.tracers.aws.api.MonitoredRequestStreamHandler;
import com.appdynamics.serverless.tracers.aws.api.ExitCall;
import com.appdynamics.serverless.tracers.aws.api.Tracer;
import com.appdynamics.serverless.tracers.aws.api.Transaction;

//Keyword extends makes MonitoredRequestStreamHandler your project's superclass
public class <MaryHadALittleLambda> extends MonitoredRequestStreamHandler {
    @Override
    //Change your method to handleMonitoredRequest
    public void handleMonitoredRequest(InputStream input, OutputStream output, Context context) throws IOException {
    
    // Your AWS Lambda function code starts here, for example int letter = 0;
    while((letter = input.read()) >= 0) {
        output.write(Character.toUpperCase(letter));
    }
    }

(Optional) Override Correlation Header Method

Automatic instrumentation uses the MonitoredRequestStreamHandler class, which provides default logic to find a correlation header.

If you need to use a custom transport mechanism for your correlation header, you must override the tracer's default logic in the getCorrelationHeader() method. Insert your custom logic to find the inbound correlation header so the tracer can return the inbound header.

The code snippet demonstrates how to override the getCorrelationHeader() method:
// (Optional) Override getCorrelationHeader().
// Only override if the tracer will not be able to find a
correlation header using its pre-configured logic.
@Override
public String getCorrelationHeader(InputStream input, Context context) {
    // Logic to find the inbound header goes here, so that the inbound
    // header can be returned
    return "correlation-header";
}

Obtain a Transaction Object

The MonitoredRequestStreamHandler class creates a transaction for you automatically. However, you need to obtain the transaction object for use with exit calls.

The code snippet below demonstrates how to obtain the transaction object:

```java
// Obtain your transaction for use with exit calls.
Transaction transaction = getTransaction();
```

Create External Exit Calls

The createExitCall method is used to obtain an exitCall object. Exit call objects record the time spent in external exit calls and allow for correlation of downstream activity.

Exit Call Types and Identifying Properties

In the Controller, each exit call has a distinct type, determined by a set of identifying properties.

The exitCall types and identifying properties are listed below:

<table>
<thead>
<tr>
<th>exitCall Type</th>
<th>Identifying Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP</td>
<td>HOST</td>
</tr>
<tr>
<td></td>
<td>PORT</td>
</tr>
<tr>
<td></td>
<td>URL</td>
</tr>
<tr>
<td></td>
<td>QUERY STRING</td>
</tr>
<tr>
<td>JDBC</td>
<td>URL</td>
</tr>
<tr>
<td></td>
<td>HOST</td>
</tr>
<tr>
<td></td>
<td>PORT</td>
</tr>
<tr>
<td></td>
<td>DATABASE</td>
</tr>
<tr>
<td></td>
<td>VERSION</td>
</tr>
<tr>
<td></td>
<td>VENDOR</td>
</tr>
</tbody>
</table>
This code sample demonstrates how you might define identifying properties for exit calls:

```java
String outgoingHeader = null;
String callType = null;
Map<String, String> identifyingProperties = new HashMap<>();

// Below properties are appropriate for an inter-AWS Lambda call
identifyingProperties.put("DESTINATION", functionName);
identifyingProperties.put("DESTINATION_TYPE", "LAMBDA");
callType="CUSTOM";

// Below properties are appropriate for an external HTTP call
identifyingProperties.put("HOST", url.getHost());
identifyingProperties.put("PORT", String.valueOf(url.getPort()));
callType="HTTP";
```

Create an Exit Call

Call `createExitCall()` on your `transaction` object.

The following code sample demonstrates how you might perform an external exit call:
public void makeExitCall(URL url){
HttpURLConnection conn = (HttpURLConnection)url.openConnection();
String outgoingHeader = null;
String callType = null;
Map<String, String> identifyingProperties = new HashMap<>();

//Below properties are appropriate for an inter-AWS Lambda call
identifyingProperties.put("DESTINATION", functionName);
identifyingProperties.put("DESTINATION_TYPE", "LAMBDA");
callType="CUSTOM";

//Below properties are appropriate for an external HTTP call
identifyingProperties.put("HOST", url.getHost());
identifyingProperties.put("PORT", String.valueOf(url.getPort()));
callType="HTTP";

//Define the createExitCall method to obtain an exitCall object.
ExitCall exitCall = transaction.createExitCall(callType,
identifyingProperties);
outgoingHeader =  exitCall.getCorrelationHeader();
exitCall.start();
try {
    // Tracer.APPDYNAMICS_TRANSACTION_CORRELATION_HEADER_KEY is the
    // name of the header that should be set
    if (outgoingHeader != null) {
        conn.setRequestProperty(Tracer.APPDYNAMICS_TRANSACTION_CORRELATION_ 
        HEADER_KEY, outgoingHeader); // set the correlation header on an
        HttpURLConnection
    }
    // Make the exit call here
} finally {
    exitCall.stop();
}
}

Making Multiple External Exit Calls

If one AWS Lambda function makes multiple exit calls, each function should be identified by a unique exitCall object. To make
multiple exit calls, create new exitCall objects for each distinct exit call. The exitCall objects are not reusable.

The following code sample demonstrates how you might perform multiple external exit calls:
public void makeExitCall(URL url)
{
    HttpURLConnection conn = (HttpURLConnection)url.openConnection();
    String outgoingHeader = null;
    String callType = null;
    Map<String, String> identifyingProperties = new HashMap<>();
    identifyingProperties.put("HOST", url.getHost());
    identifyingProperties.put("PORT", String.valueOf(url.getPort()));
    callType = "HTTP"; // This should be the callType for both HTTP and HTTPS

    // Define the createExitCall method to obtain an exitCall object for the first exitCall.
    // Give the exitCall a unique name.
    ExitCall exitCall1 = transaction.createExitCall(callType, identifyingProperties);
    outgoingHeader = exitCall1.getCorrelationHeader();
    exitCall1.start();

    try {
        // add the outgoing header to the payload and make the request,
        // Tracer.APPDYNAMICS_TRANSACTION_CORRELATION_HEADER_KEY is the name of the header that should be set
        // For example for HTTP, conn is of type HttpURLConnection
        if (outgoingHeader != null) {
            conn.setRequestProperty(Tracer.APPDYNAMICS_TRANSACTION_CORRELATION_HEADER_KEY, outgoingHeader);
        }
        // Make the first exit call
    } finally {
        exitCall1.stop();
    }

    // Define the createExitCall method to obtain an exitCall object for the second exitCall.
    // Give the exitCall a unique name.
    ExitCall exitCall2 = transaction.createExitCall(callType, identifyingProperties);
    outgoingHeader = exitCall2.getCorrelationHeader();
    exitCall2.start();

    try {
        // add the outgoing header to the payload and make the request,
        // Tracer.APPDYNAMICS_TRANSACTION_CORRELATION_HEADER_KEY is the name of the header that should be set
        // For example for HTTP, conn is of type HttpURLConnection
        if (outgoingHeader != null) {
            conn.setRequestProperty(Tracer.APPDYNAMICS_TRANSACTION_CORRELATION_HEADER_KEY, outgoingHeader);
        }
        // Make the second exit call
    } finally {
        exitCall2.stop();
    }
}
conn.setRequestProperty(Tracer.APPDYNAMICS_TRANSACTION_CORRELATION_HEADER_KEY, outgoingHeader);
}
    //Make the second exit call
} finally {
    exitCall2.stop();

Manual Tracer Instrumentation

On this page:

- Instantiate the Tracer
- Create a Transaction
- Start and Stop a Transaction
- Create External Exit Calls

Manual tracer instrumentation consists of the following steps:

1. Instantiate the Tracer
2. Create a Transaction
3. Start and Stop a Transaction
4. Create External Exit Calls

Instantiate the Tracer

To instantiate the tracer, call the `AppDynamics.getTracer(context)` method. Place the method at the beginning of your function's entry point method.

The code snippet demonstrates how to instantiate the tracer:

```java
@override
public void handleRequest(InputStream input, OutputStream output, Context context) throws IOException {
    // Instantiate the tracer.
    Tracer tracer = AppDynamics.getTracer(context);

    // Your AWS Lambda function code begins here.
    int letter = 0;
    while((letter = input.read()) >= 0) {
        output.write(Character.toUpperCase(letter));
    }
}
```

Create a Transaction

Next, you need to create transactions. A transaction is a monitored segment in a larger business transaction. Creating a transaction involves locating the correlation header, which the serverless tracer uses to pass contextual information downstream.

Transactions either:

- Start a new business transaction, or
- Continue an existing business transaction

Start a New Business Transaction

A transaction that originates in the current function does not have an inbound correlation header. To create a business transaction, use the `createTransaction()` method and provide an empty correlation header.

Continue an Existing Business Transaction

To continue an existing business transaction, you need to locate an inbound correlation header. You can locate a correlation header automatically or manually.

If the tracer cannot find the correlation header, the tracer creates a new business transaction.
Continue a Business Transaction by Locating the Correlation Header Automatically

Use the `createTransaction(inputStream, context)` method to create a transaction that locates the correlation header automatically. This method searches your code's object schema for a key called `singularityheader` in the `inputStream` object. The correlation header finds the `singularityheader` key, then continues the business transaction.

The tracer reads the `inputStream` object, which is configured to be read once. If your application needs to read the `inputStream` object, you must use a converter method to allow both the tracer and your application to read the stream.

The code snippet shows an example of how to create a transaction and automatically locate the correlation header:

```java
public class <LambdaChops> implements RequestStreamHandler {
    @Override
    public void handleRequest(InputStream input, OutputStream output, Context context) throws IOException {
        Tracer tracer = AppDynamics.getTracer(context);

        // Use a converter method if you need to read your inputStream more than once. The tracer reads your inputStream once.
        inputStream = InputStreamConverter.convertToMarkSupportedInputStream(input);

        // Create a transaction.
        Transaction transaction = tracer.createTransaction(input, context);

        // Your AWS Lambda function code begins here.
        int letter = 0;
        while((letter = input.read()) >= 0) {
            output.write(Character.toUpperCase(letter));
        }
    }
}
```

Create a Transaction by Locating the Correlation Header Manually

To create a transaction that locates the correlation header using custom logic, you need to manually parse a `correlationHeader` string. Next, call the `createTransaction()` method and provide the `correlationHeader` object. This process gives you full control over the parsing of your function's input payload.

The code snippet shows an example of how to obtain the correlation header string and create a transaction:
public class <LambdaChops> implements RequestHandler {
    @Override
    public O handleRequest(InputStream input, OutputStream output, Context context) {
        Tracer tracer = AppDynamics.getTracer(context);

        // Manually parse the correlation string.
        String inputAsString = IOUtils.toString(inputStream, Charset.forName("UTF-8"));
        JsonParser parser = new JsonParser();
        JsonObject inputObject = parser.parse(inputAsString).getAsJsonObject();

        String correlationHeader = "";
        if (inputObject.has(Tracer.APPDYNAMICS_TRANSACTION_CORRELATION_HEADER_KEY)) {
            correlationHeader = inputObject.get(Tracer.APPDYNAMICS_TRANSACTION_CORRELATION_HEADER_KEY).getAsString();
        } else {
            // Try reading from HTTP headers
            if (inputObject.has("headers")) {
                JsonObject httpHeaders = inputObject.getAsJsonObject("headers");
                if (httpHeaders.has(Tracer.APPDYNAMICS_TRANSACTION_CORRELATION_HEADER_KEY)) {
                    correlationHeader = httpHeaders.get(Tracer.APPDYNAMICS_TRANSACTION_CORRELATION_HEADER_KEY).getAsString();
                }
            }
        }

        // Create transaction object using the correlation header. If the correlationHeader string is empty, the transaction you create uses the default transaction name.
        Transaction transaction = tracer.createTransaction(correlationHeader);

        // Your AWS Lambda function code starts here
        ...
    }
}

Start and Stop a Transaction
After creating a transaction, you need to set transaction boundaries. Use the `transaction.start()` and `transaction.stop()` methods to place boundaries around any section of code that you want the tracer to monitor.

As a best practice, you can report all events, even if your function crashes, by running your entire function within a `try` block, and then stopping the transaction within a `finally` block.

This code sample demonstrates how to start and stop a transaction that monitors an entire function:

```java
@override
public void handleRequest(InputStream input, OutputStream output, Context context) throws IOException {
    Tracer tracer = AppDynamics.getTracer(context);
    Transaction transaction = tracer.createTransaction(input, context);

    //Start the transaction monitoring.
    transaction.start();
    try {
        int letter = 0;
        while((letter = input.read()) >= 0) {
            output.write(Character.toUpperCase(letter));
        }
    } finally {
        transaction.stop();
        AppDynamics.cleanup();
    }
}
```

Create External Exit Calls

The `createExitCall` method is used to obtain an `exitCall` object. Exit call objects record the time spent in external exit calls and allow for correlation of downstream activity.

**Exit Call Types and Identifying Properties**

In the Controller, each exit call has a distinct type, determined by a set of identifying properties.

The `exitCall` types and identifying properties are listed below:

<table>
<thead>
<tr>
<th>exitCall Type</th>
<th>Identifying Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP</td>
<td>HOST</td>
</tr>
<tr>
<td></td>
<td>PORT</td>
</tr>
<tr>
<td></td>
<td>URL</td>
</tr>
<tr>
<td></td>
<td>QUERY STRING</td>
</tr>
</tbody>
</table>
This code sample demonstrates how you might define identifying properties for exit calls:

```java
String outgoingHeader = null;
String callType = null;
Map<String, String> identifyingProperties = new HashMap<>();

// Below properties are appropriate for an inter-AWS Lambda call
identifyingProperties.put("DESTINATION", functionName);
identifyingProperties.put("DESTINATION_TYPE", "LAMBDA");
callType="CUSTOM";

// Below properties are appropriate for an external HTTP call
identifyingProperties.put("HOST", url.getHost());
identifyingProperties.put("PORT", String.valueOf(url.getPort()));
callType="HTTP";
```

Create an External Exit Call

Call `createExitCall()` on your transaction object.

The following code sample demonstrates how you might perform an external exit call:
public void makeExitCall(URL url){

HttpURLConnection conn = (HttpURLConnection)url.openConnection();

String outgoingHeader = null;
String callType = null;
Map<String, String> identifyingProperties = new HashMap<>();

//Below properties are appropriate for an inter-AWS Lambda call
identifyingProperties.put("DESTINATION", functionName);
identifyingProperties.put("DESTINATION_TYPE", "LAMBDA");
callType="CUSTOM";

//Below properties are appropriate for an external HTTP call
identifyingProperties.put("HOST", url.getHost());
identifyingProperties.put("PORT", String.valueOf(url.getPort()));
callType="HTTP";

//Define the createExitCall method to obtain an exitCall object.
ExitCall exitCall = transaction.createExitCall(callType,
identifyingProperties);
outgoingHeader = exitCall.getCorrelationHeader();
exitCall.start();

try {
// Tracer.APPDYNAMICS_TRANSACTION_CORRELATION_HEADER_KEY is the
// name of the header that should be set

    if (outgoingHeader != null) {
        conn.setRequestProperty(Tracer.APPDYNAMICS_TRANSACTION_CORRELATION_HEADER_KEY,
                              outgoingHeader); // set the correlation header on an
        HttpURLConnection
    }

// Make the exit call here

} finally {
    exitCall.stop();
}
}

Create Multiple External Exit Calls

If one AWS Lambda function makes multiple exit calls, each function should be identified by a unique exitCall object. To make multiple exit calls, create new exitCall objects for each distinct exit call. The exitCall objects are not reusable.

The following code sample demonstrates how you might perform multiple external exit calls.
public void makeExitCall(URL url) {

HttpURLConnection conn = (HttpURLConnection) url.openConnection();

String outgoingHeader = null;
String callType = null;
Map<String, String> identifyingProperties = new HashMap<>();

identifyingProperties.put("HOST", url.getHost());
identifyingProperties.put("PORT", String.valueOf(url.getPort()));
callType = "HTTP"; // This should be the callType for both HTTP and HTTPS

// Define the createExitCall method to obtain an exitCall object for the first exitCall.
// Give the exitCall a unique name.
ExitCall exitCall1 = transaction.createExitCall(callType, identifyingProperties);
outgoingHeader = exitCall1.getCorrelationHeader();
exitCall1.start();

try {
    // add the outgoing header to the payload and make the request,
    Tracer.APPDYNAMICS_TRANSACTION_CORRELATION_HEADER_KEY is the name of the header that should be set
    // For example for HTTP, conn is of type HttpURLConnection
    if (outgoingHeader != null) {

        conn.setRequestProperty(Tracer.APPDYNAMICS_TRANSACTION_CORRELATION_HEADER_KEY, outgoingHeader);
    }
    // Make the first exit call
    finally {
        exitCall1.stop();
    }
}

// Define the createExitCall method to obtain an exitCall object for the second exitCall.
// Give the exitCall a unique name.
ExitCall exitCall2 = transaction.createExitCall(callType, identifyingProperties);
outgoingHeader = exitCall2.getCorrelationHeader();
exitCall2.start();

try {
    // add the outgoing header to the payload and make the request,
    Tracer.APPDYNAMICS_TRANSACTION_CORRELATION_HEADER_KEY is the name of the header that should be set
    // For example for HTTP, conn is of type HttpURLConnection
    if (outgoingHeader != null) {


conn.setRequestProperty(Tracer.APPDYNAMICS_TRANSACTION_CORRELATION_HEADER_KEY, outgoingHeader);
}  //Make the second exit call
} finally {

exitCall2.stop();
}
}
Deploy the Functions

On this page:

- Deploy to AWS Lambda
- Verify the Instrumentation

After you have completed the required steps in Set Up the Serverless APM Environment and Instrument the Function Code, you can deploy the function to AWS and verify successful instrumentation.

**Deploy to AWS Lambda**

Deploy your function to AWS as follows:

1. Save changes to your code and environment variables.
2. Upload your function to AWS Lambda.
3. Test your function. You need to create a test load on your function to see activity within AppDynamics.

If your test log yields no errors, Serverless APM has successfully instrumented. Your AWS Lambda function appears on the flow map after a few minutes of processing application load.

To protect your production code, the tracer never fails your function. Your function may execute successfully yet fail to instrument the tracer. Check the logs of each AWS Lambda test in the AWS Management Console to confirm Serverless APM successfully instrumented.

**Verify the Instrumentation**

After your AWS Lambda function successfully uploads in the AWS Management Console:

1. Log in to your AppDynamics Controller.
2. Navigate to Applications and find your new application.

Amazon Web Services, the AWS logo, AWS, and any other AWS Marks used in these materials are trademarks of Amazon.com, Inc. or its affiliates in the United States and/or other countries.
Administer Serverless APM for AWS Lambda

The topics in this section describe how to maintain Serverless APM for AWS Lambda and tune its configuration for your requirements.

Customization Options

Serverless APM customization options include additional error reports, HTTP proxy support, and configuring environment variables in your function code. See Customize the Serverless APM Instrumentation.

Error Reports

When you instrument automatically, the tracer reports all internal thrown exceptions as errors, including runtime exceptions. If you instrument the tracer manually, you need to configure transaction error reports in your code. You may also need to customize error reports if your business logic causes your business transaction to be marked in error.

Additionally, you can create error reports for exit calls. Exit call error reports require manual coding, regardless of how you instrument the tracer.

HTTP Proxy Support

The tracer makes outbound calls over HTTPS to report metrics. If your deployment requires an HTTP proxy, you can forward requests to your proxy server.

Configure Environment Variables in Your Function Code

You need to configure environment variables for the tracer to know where to send your data. Input environment variables in your AWS Management Console or in your function code.

If you prefer to customize the function and business transaction names and keep the tracer settings in your code, you can override the environment variables using the AppDynamics.Config.Builder object. This option allows you to retrieve values at runtime from any desired source. See Customize Serverless APM Instrumentation.

End User Monitoring Integration

Correlation between AWS Lambda functions and End User Monitoring (EUM) agents is identical to the correlation between Browser RUM and business transactions. When an instrumented AWS Lambda function originates a business transaction, the tracer can return EUM metadata as response headers. The response headers link the end user device's response time with the business transaction execution time. Data flows from the headers to your EUM server, then to your Controller. See Integrate Serverless APM with End User Monitoring for more details.

Amazon Web Services, the AWS logo, AWS, and any other AWS Marks used in these materials are trademarks of Amazon.com, Inc. or its affiliates in the United States and/or other countries.
Customize Serverless APM Instrumentation

On this page:

- HTTP Proxy Support
- Configure Environment Variables in Your Function Code
- Transaction Error Reports
- Exit Call Error Reports

This topic describes optional features of Serverless APM for AWS Lambda:

- HTTP Proxy Support
- Configure Environment Variables in Your Function Code
- Transaction Error Reports
- Exit Call Error Reports

**HTTP Proxy Support**

To enable HTTP Proxy Support, define the following variables in the AWS Management Console:

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPDYNAMICS_HTTP_PROXY_HOST</td>
<td>Publicly-accessible hostname of the proxy.</td>
<td>myproxy.example.com</td>
</tr>
<tr>
<td>APPDYNAMICS_HTTP_PROXY_PORT</td>
<td>Port on which the proxy is running.</td>
<td>8080</td>
</tr>
<tr>
<td>APPDYNAMICS_HTTP_PROXY_SERVER_CERTIFICATE</td>
<td>Relative path to the proxy server's certificate. Required if your proxy server runs on a self-signed certificate for your AWS Lambda function to trust the proxy. If you have configured your runtime environment to trust your proxy server, you do not need this variable.</td>
<td>resources/proxy-cert.pem</td>
</tr>
</tbody>
</table>

Additionally, you can add credentials for your proxy server's basic authentication. Add the following variables in the AWS console:

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPDYNAMICS_HTTP_PROXY_USER</td>
<td>Username associated with your basic authentication.</td>
<td>user1</td>
</tr>
<tr>
<td>APPDYNAMICS_HTTP_PROXY_PASSWORD</td>
<td>Relative path to the proxy server's basic authentication password. Required for your proxy's password to trust your AWS Lambda function.</td>
<td>password123</td>
</tr>
<tr>
<td>APPDYNAMICS_HTTP_PROXY_PASSWORD_FILE</td>
<td>Plaintext file to the proxy server's basic authentication password file. Update your deployment package to contain the proxy server's password file. Required for your proxy's password to trust your AWS Lambda function.</td>
<td>resources/proxy-password.txt</td>
</tr>
</tbody>
</table>

When adding basic credentials, you are required to include either `APPDYNAMICS_HTTP_PROXY_PASSWORD` or `APPDYNAMICS_HTTP_PROXY_PASSWORD_FILE`. If you include both, the tracer only passes `APPDYNAMICS_HTTP_PROXY_PASSWORD_FILE`.

**Configure Environment Variables in Your Function Code**

Use the `AppDynamics.Config.Builder` method to override settings that the tracer reads from the process environment:
configBuilder.accountName(accountName)
  .applicationName(appName)
  .tierName(tierName)
  .controllerHost(controllerHost)
  .controllerPort(controllerPort)
  .defaultBtName(context.getFunctionName() + "_bt")
  .controllerAccessKey(controllerAccessKey)
  .lambdaContext(context);

Implementation of the `configBuilder` object differs slightly depending on which instrumentation method you choose, automatic or manual.

Automatic Instrumentation

Automatic instrumentation uses the `MonitoredRequestStreamHandler` class, which implements a factory method, `getConfigBuilder()`, to instrument the tracer.

You can override this factory method and use any builder method to configure environment variables.

The code snippet shows how to enter variables in your code for automatic instrumentation:

```java
@Override
public AppDynamics.Config.Builder getConfigBuilder(Context context) {
  String controllerHost = System.getenv("APPDYNAMICS_CONTROLLER_HOST");
  int controllerPort = Integer.parseInt(System.getenv("APPDYNAMICS_CONTROLLER_PORT"));
  String accountName = System.getenv("APPDYNAMICS_ACCOUNT_NAME");
  String appName = System.getenv("APPDYNAMICS_APPLICATION_NAME");
  String tierName = System.getenv("APPDYNAMICS_TIER_NAME");
  String controllerAccessKey = System.getenv("APPDYNAMICS_AGENT_ACCOUNT_ACCESS_KEY");

  configBuilder.accountName(accountName)
    .applicationName(appName)
    .tierName(tierName)
    .controllerHost(controllerHost)
    .controllerPort(controllerPort)
    .defaultBtName(context.getFunctionName() + "_bt")
    .controllerAccessKey(controllerAccessKey)
    .lambdaContext(context);
  return configBuilder;
}
```

Manual Instrumentation
In manual instrumentation, the overloaded version of the `AppDynamics.getTracer()` method configures the tracer’s settings. The overloaded method takes a `configBuilder` object as a parameter, instead of the `context` object, to instrument the tracer.

When you use this variant of `AppDynamics.getTracer()`, you must pass the `context` object via the `AppDynamics.Config.Builder.lambdaContext()` method.

The code snippet shows how to enter variables in your code for manual instrumentation:

```java
@override
public void handleRequest(InputStream input, OutputStream output, Context context) throws IOException {  
    String controllerHost = System.getenv("APPDYNAMICS_CONTROLLER_HOST");
    int controllerPort = Integer.parseInt(System.getenv("APPDYNAMICS_CONTROLLER_PORT"));
    String accountName = System.getenv("APPDYNAMICS_ACCOUNT_NAME");
    String appName = System.getenv("APPDYNAMICS_APPLICATION_NAME");
    String tierName = System.getenv("APPDYNAMICS_TIER_NAME");
    String controllerAccessKey = System.getenv("APPDYNAMICS_AGENT_ACCOUNT_ACCESS_KEY");
    configBuilder.accountName(accountName)
        .applicationName(appName)
        .tierName(tierName)
        .controllerHost(controllerHost)
        .controllerPort(controllerPort)
        .defaultBtName(context.getFunctionName() + "_bt")
        .controllerAccessKey(controllerAccessKey)
        .lambdaContext(context);

    //Replace the context object with config.Builder.build()
    Tracer tracer = AppDynamics.getTracer(configBuilder.build());
}
```

**Transaction Error Reports**

If you automatically instrument the tracer, transaction errors are reported if an exception is thrown. If you manually instrument the tracer, transaction error reports are optional and require manual configuration.

If your business logic defines the transaction as in error for any reason, you can customize the error report method to mark the transaction in error.

To create transaction error reports, use `transaction.reportError()`.

The `transaction.reportError()` method reports one error per invocation. You can also use `transaction.reportError()` for any error name or errorMessage:

```java
transaction.reportError(errorName, errorMessage);
```
Exit Call Error Reports

You can report errors and exceptions that occur during exit call execution using the `exitCall.reportError()` method. You must call the method before `transaction.stop()`.

Use `exitCall.reportError()` for any exception or subclass:

```java
exitCall.reportError(exception);
```

You can also use `exitCall.reportError()` for a specific `errorName` or `errorMessage`:

```java
exitCall.reportError(errorName, errorMessage);
```

Amazon Web Services, the AWS logo, AWS, and any other AWS Marks used in these materials are trademarks of Amazon.com, Inc. or its affiliates in the United States and/or other countries.
Integrate Serverless APM with End User Monitoring

On this page:

- Before You Begin
- Configure Response Header Mappings
- Update the Front End Application
- Enable CORS Configuration
- Troubleshoot the EUM Integration

This document contains links to AWS documentation. AppDynamics makes no representation as to the accuracy of Amazon documentation since Amazon controls its own documentation.

Serverless APM for AWS Lambda is designed to integrate with your existing End User Monitoring (EUM) configurations. EUM integration provides complete end-to-end visibility on the performance of your web and mobile applications, linking calls from an end-user device through your serverless functions to continue your business transactions.

AWS Lambda functions can correlate EUM and AWS Lambda-originated business transactions, in conjunction with the following EUM agents:

- Browser Real User Monitoring (Browser RUM)
- Mobile Real User Monitoring (Mobile RUM)
- IoT Monitoring

Integration of Serverless APM with End User Monitoring consists of these steps:

1. Add and Return EUM Metadata
2. Configure Response Header Mappings
   a. Use the Amazon API Gateway
   b. Use AWS Lambda Proxy Integration
3. Update the Front End Application
4. Enable CORS configuration (If applicable)
5. Troubleshoot the EUM Integration

This document assumes you are familiar with AppDynamics End User Monitoring. See EUM documentation for more information.

Before You Begin

Ensure your deployment has:

- An AWS Lambda function with an instrumented with Serverless APM
- Active EUM licenses and JavaScript agents

Add and Return EUM Metadata

To integrate EUM metadata with an AWS Lambda function, use the transaction.getEumMetadata() method after calling transaction.stop().

To gather EUM metadata, this method queries a transaction object and returns an ordered list of strings. Assign each string separately, in the order specified, to the headers ADRUM_0, ADRUM_1, ADRUM_2, and ADRUM_3 for consumption by the downstream JavaScript agent.

This code snippet shows how to call transaction.getEumMetadata():

```javascript
// Call the transaction.stop() method
transaction.stop();

// Call the transaction.getEumMetadata() method
var eumMetadata = transaction.getEumMetadata();
```
public void stopTransactionAndRecordEUM(OutputStream output) throws IOException {
    Transaction transaction = getTransaction();

    // Your AWS Lambda function code

    // You must call transaction.stop(); before calling transaction.getEumMetadata();
    transaction.stop();
    // call transaction.getEumMetadata(); function to query transaction object for EUM metadata
    List<String> eumMetadata = transaction.getEumMetadata();


Configure Response Header Mappings

Configure response header mappings to pass ADRUM_n headers back to the browser. You can configure response header mappings using the AWS API Gateway or pass them directly using AWS Lambda Proxy Integration.

Response Header Mappings in Amazon API Gateway

Add header mappings for each ADRUM_n header in the Amazon API Gateway. To configure response header mappings in the Amazon API Gateway, refer to the Amazon documentation.

The screenshot illustrates how to map these headers in the AWS API Gateway:

Amazon Web Services, the AWS logo, AWS, and any other AWS Marks used in these materials are trademarks of Amazon.com, Inc. or its affiliates in the United States and/or other countries.

Next, return the ADRUM_n headers as part of the AWS Lambda function output, which requires adding the field `eumMetadata` to your output object class. You can use another field name, as long your function code is consistent with the response header mappings.

This code snippet shows how to add the headers field to an output object class:
// return the EUM ADRUM_n headers in an eumMetadata field in the output object

String outputString = "success";
Output outputObj = new Output(outputString);
outputObj.eumMetadata = eumMetadata;
output.write(new Gson().toJson(outputObj).getBytes(StandardCharsets.UTF_8));

Response Header Mappings in AWS Lambda Proxy Integration

As an alternative to custom integration, AWS Lambda proxy integration allows the client to call each function in the backend and returns output in a JSON format. Map these responses by adding each single-value ADRUM_n header directly to the headers field in your output object class. Refer to the Amazon documentation for details.

Update the Front End Application

To see the correlation between EUM and AWS Lambda-originated business transactions, you must inject the JavaScript Agent into your browser's HTML, as described in the EUM documentation.

Enable CORS Configuration

Browsers require cross-origin resource sharing (CORS) for functions that access responses from browser requests to a domain other than the base page's domain. You must explicitly grant cross-domain access to all applicable ADRUM_n headers.

To enable CORS configuration, define the required AWS response headers in the Amazon API Gateway:

- **Access-Control-Expose-Headers** - maps the custom ADRUM headers. These headers must have the names ADRUM_0 through ADRUM_3 to be compatible with the JavaScript agent.
- **Access-Control-Allow-Methods** - check the header(s) with the appropriate method.
- **Access-Control-Allow-Origin** - set to the full URL of the web page that originated the request, including schema, hostname, and port.

See AWS documentation for advanced headers and additional details.

The screenshot illustrates how to enable CORS in the Amazon API Gateway:
Enable CORS

Gateway Responses for **-test-api** API

- DEFAULT 4XX
- DEFAULT 5XX

Methods

- POST
- OPTIONS

**Access-Control-Allow-Methods**

- POST, OPTIONS

**Access-Control-Allow-Headers**

- Content-Type, X-Amz-Date, Authorization

**Access-Control-Allow-Origin**

- http://my-saas-service.com:8000

**Advanced**

- **Access-Control-Expose-Headers**
  - ADRUM_0, ADRUM_1, ADRUM_2, AD

- **Access-Control-Max-Age**

- **Access-Control-Allow-Credentials**

---

After you have enabled CORS, you must define the response header mappings. To configure response header mappings in the Amazon API Gateway, refer to the Amazon documentation.

The screenshot illustrates response header mappings after CORS configuration:

<table>
<thead>
<tr>
<th>Response header</th>
<th>Mapping value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADRUM_3</td>
<td>integration.response.body.eumHeaders[3]</td>
</tr>
<tr>
<td>ADRUM_1</td>
<td>integration.response.body.eumHeaders[1]</td>
</tr>
<tr>
<td>Access-Control-Expose-Headers</td>
<td>'ADRUM_0,ADRUM_1,ADRUM_2,ADRUM_3'</td>
</tr>
<tr>
<td>ADRUM_2</td>
<td>integration.response.body.eumHeaders[2]</td>
</tr>
<tr>
<td>Access-Control-Allow-Credentials</td>
<td>'true'</td>
</tr>
<tr>
<td>Access-Control-Allow-Methods</td>
<td>'POST, OPTIONS'</td>
</tr>
<tr>
<td>Access-Control-Allow-Origin</td>
<td>'<a href="http://localhost:8000">http://localhost:8000</a>'</td>
</tr>
<tr>
<td>ADRUM_0</td>
<td>integration.response.body.eumHeaders[0]</td>
</tr>
</tbody>
</table>

Troubleshoot the EUM Integration

If you do not see a correlation between EUM data and business transactions, collect an HTTP Archive (HAR) file that captures the activity in your browser. Use the HAR file to confirm all necessary CORS headers are set and expected data populates in the ADRUM_\_n headers.

The EUM documentation provides additional troubleshooting advice.
Disable Serverless APM

You can disable Serverless APM for AWS Lambda at any time to stop all tracing by AppDynamics.

To disable Serverless APM for AWS Lambda:

1. Sign in to the AWS Management Console.
2. Navigate to the AWS Lambda function that you want to disable.
3. Add the key: APPDYNAMICS_DISABLE_AGENT to your environment variables.
4. Set the value to true.

Additionally, you need to unsubscribe from Serverless APM in the AWS Marketplace:

1. Sign in to the AWS Management Console.
2. Go to Your Account > Your Software Subscriptions > SaaS. Locate the AppDynamics Serverless APM for AWS Lambda subscription.
3. Select Cancel Subscription. Cancel Subscription unsubscribes all Controllers from Serverless APM.

When you disable Serverless APM, the tracer continues to monitor any instances currently executing until those instances finish. Requests executed by instances started after you disable the tracer are not monitored.

Amazon Web Services, the AWS logo, AWS, and any other AWS Marks used in these materials are trademarks of Amazon.com, Inc. or its affiliates in the United States and/or other countries.
Apache Web Server Agent

On this page:
- About the Apache Web Server Agent
- Agent Deployment Architecture
- Naming Apache Web Server Nodes

Related pages:
- Apache Web Server Backend Detection
- Transaction Detection for Apache Web Servers

The Apache Web Server Agent discovers, maps and tracks metrics for business transactions, app services, and backends in your web application by injecting instrumentation into the Apache server at runtime.

About the Apache Web Server Agent

The Apache Agent automatically discovers incoming HTTP requests to Apache Web Server nodes as business transaction entry points. AppDynamics uses URI-based naming for transactions originating at Apache Agent nodes. The agent discovers exit points to Apache modules and resolves them to the downstream tier or backend. See Apache Web Server Backend Detection.

The Apache Agent tracks the following metrics for web servers:
- Calls per minute and time spent in the web server tier
- Calls per minute and time spent in Apache module backends
- Number of errors

The agent identifies slow and stalled transactions according to dynamic baselines. When it detects an HTTP response code greater than 400, the agent identifies the transaction as an error transaction.

Transaction snapshots taken on Apache Web Server nodes include the following data:
- URL
- HTTP error codes
- HTTP request data:
  - Cookie, including JSESSIONID
  - Referer
  - X-Forwarded-For
  - SM_USER
- Exit calls to Apache modules

Agent Deployment Architecture

The Apache Web Server Agent consists of these components:
- Apache extension
- AppDynamics agent proxy

The agent proxy is a Java process that handles communication between the Apache Web Server Agent and Controller. The proxy transmits the raw performance data from the agent to the Controller, which stores, analyzes, and presents the data.
As a separate process, the agent proxy must be started separately from the monitored web server. You can start the proxy manually or configure it to start automatically by setting `AppDynamicsLaunchProxy` in `appdynamics_agent.conf`. Whether the proxy is started automatically or manually, the commands that start the agent check whether the proxy is already running and do not attempt to start it again if it is.

**Naming Apache Web Server Nodes**

Each web server instance maps to a node in the AppDynamics model. When naming Apache Web Server nodes, be sure to use a meaningful name. Some options are:

- `hostName-appName-nodeName`
- `hostName-tierName-nodeName`
- `appName-nodeName`
- `tierName-nodeName`
- `IP address`
- `fully qualified domain name`
Supported Apache Web Servers

On this page:

- Apache Web Servers
- Operating Systems
- Architectures
- Automatically Discovered Business Transactions
- Remote Service Detection

Related pages:

- Browser RUM Supported Environments

Apache Server Agent Support

**Apache Web Servers**

<table>
<thead>
<tr>
<th>Supported Apache Web Server Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Apache HTTP Server 2.2.x</td>
</tr>
<tr>
<td>• Apache HTTP Server 2.4.x</td>
</tr>
<tr>
<td>• IBM HTTP Server 7.0 +</td>
</tr>
<tr>
<td>• Oracle HTTP Server 11g+</td>
</tr>
</tbody>
</table>

**Operating Systems**

- Any Linux distribution based on glibc 2.5+

**Architectures**

- 32-bit
- 64-bit

**Automatically Discovered Business Transactions**

The Apache Agent automatically discovers the following business transactions:

<table>
<thead>
<tr>
<th>Type</th>
<th>Custom Configuration Options</th>
<th>Downstream Correlation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web (HTTP)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

By default the agent excludes requests for the following static file types:

- bmp
- cab
- class
- conf
- css
- doc
- gif
- ico
- jar
- jpeg
- jpg
- js
- mov
- mp3
- mp4
- pdf
- png
- pps
Remote Service Detection

Apache Modules

The Apache Agent automatically detects loaded Apache modules as remote services. The agent excludes a list of common modules from detection.

Show the list of excluded modules...

- core.c
- http_core.c
- mod_access_compat.c
- mod_actions.c
- mod_alias.c
- mod_allowmethods.c
- mod_appdynamics.cpp
- mod_auth_basic.c
- mod_auth_digest.c
- mod_authn_alias.c
- mod_authn_anon.c
- mod_authn_core.c
- mod_authn_dbd.c
- mod_authn_dbm.c
- mod_authn_default.c
- mod_authn_file.c
- mod_authn_socache.c
- mod_authnz_ldap.c
- mod_authz_core.c
- mod_authz_dbd.c
- mod_authz_dbm.c
- mod_authz_default.c
- mod_authz_groupfile.c
- mod_authz_host.c
- mod_authz_owner.c
- mod_authz_user.c
- mod_autoindex.c
- mod_cache.c
- mod_cache_disk.c
- mod/cgi.c
mod_data.c
mod_dbd.c
mod_deflate.c
mod_dir.c
mod_disk_cache.c
mod_dumpio.c
mod_echo.c
mod_env.c
mod_expires.c
mod_ext_filter.c
mod_file_cache.c
mod_filter.c
mod_headers.c
mod_include.c
mod_info.c
mod_lbmethod_bybusyness.c
mod_lbmethod_byrequests.c
mod_lbmethod_bytraffic.c
mod_lbmethod_heartbeat.c
mod_log_config.c
mod_logio.c
mod_lua.c
mod_mem_cache.c
mod_mime.c
mod_mime_magic.c
mod_negotiation.c
mod_perl.c
mod_python.c
mod_remoteip.c
mod_reqtimeout.c
mod_rewrite.c
mod_setenvif.c
mod_slotmem_plain.c
mod_slotmem_shm.c
mod_so.c
mod_socache_dbm.c
mod_socache_memcache.c
mod_socache_shmcb.c
mod_speling.c
mod_ssl.c
mod_status.c
mod_substitute.c
mod_suexec.c
mod_systemd.c
mod_unique_id.c
mod_unixd.c
mod_userdir.c
mod_usertrack.c
mod_version.c
mod_vhost_alias.c
prefork.c
util_ldap.c

For End User Monitoring, the Apache Agent does not support automatic injection of the Javascript adrum header and footer to instrument web pages.
Install the Apache Agent

To monitor the performance of your Apache HTTP Server, IBM HTTP Server (IHS) or Oracle HTTP Server, install the AppDynamics Apache Agent on the servers where you run Apache, IHS or OHS. The agent instruments the Apache server and sends performance data to a Java proxy, which in turn sends data to the AppDynamics Controller.

If you are using the Apache Monitoring Extension with the Standalone Machine Agent, you can continue to use it. You may need to restart the Standalone Machine agent after installing the Apache Agent.

Before Starting

Ensure that the web server version and operating system are supported. See Supported Apache Web Servers for more information.

Also verify the following requirements on the machine on which you are installing the agent:

- Perform the installation under the same user and group that Apache is using when spawning worker processes and threads
- Ensure that the ulimit setting for open file descriptors is set such that your process (or any others) does not exceed it. At a minimum you need two times more than the maximum number of simultaneously active threads created by the Apache Web Server during heavy loads. AppDynamics recommend setting the ulimit open file descriptors to 10000.
- Ensure that you do not configure more than 4032 concurrent threads
- Check for modules with dependencies on libstdc++.so.5. see Special Considerations for Apache Web Servers with libstdc++.so.5 Modules. This is most common with IHS and OHS.

Download and Install the Apache Agent

To guarantee performance and stability, you should install one agent for each Apache instance. Do not exceed two Apache instances per agent.

1. Download the Apache Agent from the Getting Started wizard or https://www.appdynamics.com/download.
2. Extract the agent to /opt.

```
  tar -xzvf ./appdynamics-sdk-native.<version>.tar.gz -C /opt
```

The agent installs to /opt/appdynamics-sdk-native. This is the <agent_install_directory>. If you are installing multiple agents, you must create separate install directories for each agent.

3. Grant the Apache, IHS or OHS owner read and write permissions to the logs directory: <agent_install_directory>/logs.
4. Run install.sh. This script installs the agent proxy.

```
<agent_install_directory>/install.sh
```

5. For an agent to monitor your Apache instance, you must launch the proxy. If you are monitoring multiple Apache instances, we recommend that you manually launch each proxy, and set AppDynamicsLaunchProxy (see 'Configure the Apache Agent') to
The next step is to create the agent configuration file and configure the agent.

Configure the Apache Agent

1. Create a configuration file for the AppDynamics Apache Web Server agent, such as appdynamics_agent.conf, in the Apache, IHS, or OHS configuration directory. For example:

   ```
touch /etc/<path_to_webserver_dir>/conf/appdynamics_agent.conf
   ```

2. In the appdynamics_agent.conf file, configure the following settings. If you have multiple agents installed, you must edit the appdynamics_agent.conf file for each agent, specifying which directory the agent load modules are located. Ensure that each Apache instance references a different appdynamics_agent.conf file, and that each file specifies a different agent install directory.

   - **LoadFile**: Loads the AppDynamics Agent SDK shared library. Assuming the agent is installed in /opt, the setting would be /opt/appdynamics-sdk-native/sdk_lib/lib/libappdynamics_native_sdk.so. Required. For example:

     ```
     LoadFile
     /opt/appdynamics-sdk-native/sdk_lib/lib/libappdynamics_native_sdk.so
     ```

   - **LoadModule**: Loads the AppDynamics Apache Agent shared library. The correct module is libmod_appdynamics.so for Apache 2.4 and libmod_appdynamics22.so for Apache 2.2. Required. IHS and OHS require the Apache 2.2 module. For example:

     ```
     LoadModule appdynamics_module
     /opt/appdynamics-sdk-native/WebServerAgent/Apache/libmod_appdynamics.so
     ```

     Apache 2.2 example:

     ```
     LoadModule appdynamics_module
     /opt/appdynamics-sdk-native/WebServerAgent/Apache/libmod_appdynamics22.so
     ```

   - **AppDynamicsEnabled**: Must be included and set to ON for web server monitoring to be enabled. Otherwise, monitoring is disabled by default. For example:

     ```
     AppDynamicsEnabled  ON
     ```

     To disable monitoring, set to OFF or remove the AppDynamics module include statement (see Configure the Apache Server for AppDynamics) from the Apache configuration file.

   - **AppDynamicsControllerHost**: Hostname or IP address of the AppDynamics controller to connect to. Required. For example
AppDynamicsControllerHost
mycontroller.saas.appdynamics.com

- **AppDynamicsControllerPort**: Controller HTTP(S) port. For SaaS, use 443 for HTTPS or 80 for HTTP. For on-premises, HTTP defaults to 8090 and HTTPS defaults to 8181. Optional. Configure if you do not want to use these defaults. If you are using HTTPS, the SSL setting (**AppDynamicsControllerSSL**) must also be set. For example:

```
AppDynamicsControllerPort 80
```

- **AppDynamicsControllerSSL**: Set to connect to the Controller over SSL. Default is OFF. Set to ON to enable SSL connection and also set the **AppDynamicsControllerPort** to an HTTPS port.

Example:

```
AppDynamicsControllerSSL OFF
```

- **AppDynamicsAccountName**: AppDynamics account name. For on-premises controllers, find your credentials under gear (⚙️) icon > License in the Controller. For SaaS controllers, AppDynamics provides your credentials in your Welcome email. Required.

Example:

```
AppDynamicsAccountName MyCompany
```

- **AppDynamicsAccessKey**: AppDynamics account access key. For on-premises Controllers, find your credentials under gear (⚙️) icon > License in the Controller. For SaaS controllers, AppDynamics provides your credentials in your Welcome email. Required.

Example:

```
AppDynamicsAccessKey zd8yjh5yuy5k
```

- **AppDynamicsProxyHost**: Proxy server hostname or IP address. Optional. Use if the agent connects to the Controller through a HTTP proxy server.

- **AppDynamicsProxyPort**: Proxy server port. Optional. Use if the agent connects to the Controller through a HTTP proxy server.

- **AppDynamicsApplication**: AppDynamics application name. See [Overview of Application Monitoring](#) for more information. Required.

Example:

```
AppDynamicsApplication MyWS2App
```

- **AppDynamicsTier**: AppDynamics tier name. See [Overview of Application Monitoring](#) for more information. Required.

Example:

```
AppDynamicsTier MyWS2
```
**AppDynamicsNode**: AppDynamics node name. See *Overview of Application Monitoring* for more information. The node name is required and each node name must be unique. For example:

```
AppDynamicsNode WS2_1
```

### Sample appdynamics_agent.conf

```bash
# Load the AppDynamics SDK.
LoadFile /opt/appdynamics-sdk-native/sdk_lib/lib/libappdynamics_native_sdk.so

# Load the Apache Agent. In this example for Apache 2.4
LoadModule appdynamics_module
/opt/appdynamics-sdk-native/WebServerAgent/Apache/libmod_appdynamics.so

AppDynamicsEnabled On

# AppDynamics Controller connection.
AppDynamicsControllerHost mycontroller.saas.appdynamics.com
AppDynamicsControllerPort 80
AppDynamicsControllerSSL OFF

# Account credentials
AppDynamicsAccountName MyCompany
AppDynamicsAccessKey zd8yjh5yy5k

# Configure Controller connection through an HTTP proxy server.
# AppDynamicsProxyHost <proxy host>
# AppDynamicsProxyPort <proxy port>

# Business application, tier, node
AppDynamicsApplication MyApache2App
AppDynamicsTier Apache2
AppDynamicsNode Apache2_1
```

### Launch the Proxy

To launch the proxy manually, run `runSDKProxy.sh` using the user ID of the Apache worker threads.

```
nohup <agent_install_directory>/runSDKProxy.sh >/dev/null 2<&agent_install_dir>/logs/proxy.out &
```

It is recommended that you create a Unix System Service that automatically starts the proxy at system startup.

The Proxy expects the appropriate (64- vs. 32-bit) `libstdc++6` library location. If you are installing the Apache Agent on a
Additional Directives

Under special circumstances, you may need to set the optional directives below. Do not set these directives unless instructed by AppDynamics support.

- **AppDynamicsResolveBackends**: Controls the way Apache module backends appear in the Controller UI. If ON, metrics for Apache modules are shown as part of downstream tiers or backends. Module backends do not appear on flow maps. If OFF, Apache module backends appear in the flow maps with their own metrics. See ‘Apache Modules’ on Install the Apache Agent. Defaults to ON. Optional.
  
  For example:

  ```
  AppDynamicsResolveBackends OFF
  ```

- **AppDynamicsTraceAsError**: If ON, tracepoints are written as errors to the Apache logs, by default error_log. Default is OFF. Optional.
  
  For example:

  ```
  AppDynamicsTraceAsError OFF
  ```

- **AppDynamicsReportAllInstrumentedModules**: If OFF, the agent reports only modules that occur during the HANDLER stage of Apache request processing. If ON, the agent reports modules during all stages of Apache request processing. The default is OFF.
  
  For example:

  ```
  AppDynamicsReportAllInstrumentedModules OFF
  ```

- **AppDynamicsLaunchProxy**: If ON, the agent launches the proxy automatically when on startup. If OFF, you need to manually launch the proxy. If your system uses any command to do a graceful restart of Apache, such as log rotate, or if your system encounters heavy loads, you should set this property to OFF. The default is OFF.
  
  For example:

  ```
  AppDynamicsLaunchProxy OFF
  ```

You may want to launch the proxy manually for debugging. For example, to set a different proxy root directory or runtime directory or to output additional debugging information.

- **AppDynamicsProxyCommDir**: Path to the directory containing the domain control socket, which the agent uses to start an AppDynamics node. Defaults to `<agent_install_directory>/logs/appd-sdk`. If you modify this setting, make sure that the directory path does not exceed 107 characters, which is the Linux limit to the socket file name size. If the directory path is too long, you will get an error message when the agent attempts to start.

  ```
  AppDynamicsProxyCommDir
  <agent_install_directory>/proxy/altcommdir
  ```

---

64-bit OS that supports running a 32-bit Apache HTTP Server, make sure that the downloaded Apache Agent matches (i.e., is 32-bit), and that the libstdc++6 library is 32-bit as well. If you have both 64-bit and 32-bit versions of that library, make sure you are pointing to the correct version using the command `export LD_PRELOAD=<path to 32bit library> before starting runSDKProxy`. 
Configure the Apache Server for AppDynamics

1. Add an include for the AppDynamics Apache Agent configuration to the server configuration. Ensure that the AppDynamics module is loaded after any modules you want to be instrumented. Any modules loaded after the AppDynamics agent are excluded from instrumentation and monitoring. For example, add the following as the last line in the httpd.conf file—on Ubuntu and Debian, this file is called apache2.conf:

```
#include AppDynamics Apache Agent Configuration
Include conf/appdynamics_agent.conf
```

If you are running multiple instances of the web server, modify the .conf Include path to match the configuration file name for that instance. For example:

```
#include AppDynamics Apache Agent Configuration.
Include conf/appdynamics_agent1.conf
```

If you are setting values for Apache thread limits—such as ServerLimit, ThreadsPerChild, and ThreadLimit—ensure that the concurrent thread limit does not exceed 4095.

2. Restart the web server. For example, for Apache on CentOS:

```
apachectl restart
```

3. Apply load to your web server to activate instrumentation.

The AppDynamics Apache Agent automatically detects the incoming HTTP calls as business transactions. It detects loaded modules as backends. Log in to the AppDynamics Controller to begin monitoring.

Map Virtual Hosts to AppDynamics Tiers

By configuring virtual hosts, Apache Web Server administrators can have a single Apache Web Server instance act as the entry point for what appears to be different websites from the end user's perspective.

In the AppDynamics application model, it will usually make sense to represent each virtual host configured in an Apache Web Server as its own tier. This tends to better represent the logical model of the environment and reduces the likelihood of quickly exhausting tier business transaction limits for an Apache Web Server that proxies a large application environment.

To associate different virtual hosts with different tiers, add the AppDynamicsApplicationContext directive to the virtual host configuration, supplying the application, tier and node name as arguments, as follows:

```
AppDynamicsApplicationContext <application> <tier> <node>
```

For example:
In the example above, the tier name is a combination of the virtual machine ServerName and port number. You are not required to include the ServerName and port number in the tier name, but they can be useful for identifying which virtual machines are creating business transactions. We recommend using a tier name that best represents the purpose of the virtual machine.

The Apache Web Server Agent associates incoming requests to the AppDynamics context based upon the virtual host server and port in the request.

Note that instead of using virtual hosts to multi-host from Apache, an administrator may run multiple instances of the Apache Web Server with a distinct httpd.conf file—on Ubuntu and Debian, this file is called apache2.conf—for each instance. In this case, you would instrument the Apache Web Server by creating an AppDynamics agent configuration file for each instance, and include each configuration file in the httpd.conf file of the corresponding instance. See Configure the Apache Agent.
Apache Agent Directories

**On this page:**
- Agent Install Directory
- WebServerAgent Directory
- Conf Directory
- Logs Directory
- Proxy Directory
- Proxy Control Directory

This page describes the directories for the Apache Agent.

**Agent Install Directory**

The agent install directory `<agent_install_directory>` is the `appdynamics-sdk-native` directory where you installed the agent.

For example, if you un-tarred the agent to `/opt`, the agent home would be: `/opt/appdynamics-sdk-native`.

It contains the following files:

- `install.sh` to install the Apache Agent and the agent proxy.
- `runSDKProxy.sh` to start the agent proxy.

**WebServerAgent Directory**

The `<agent_install_directory>/WebServerAgent` directory contains the Apache Agent libraries.

The Apache subdirectory contains the Apache modules for the Apache Agent:

- For Apache 2.2: `libmod_appdynamics22.so`
- For Apache 2.4: `libmod_appdynamics.so`

**Conf Directory**

The `<agent_install_directory>/conf` directory contains the logging configuration files:

- A logging configuration template: `appdynamics_sdk_log4cxx.xml.template`
- A log configuration file: `appdynamics_sdk_log4cxx.xml`

**Logs Directory**

The logs `<agent_install_directory>/logs` directory contains the logs for the Apache Agent and the agent proxy.

**Proxy Directory**

The proxy `<agent_install_directory>/proxy` directory contains the agent proxy files.

**Proxy Control Directory**

The proxy control directory contains the domain control socket, which the agent uses to start an AppDynamics node. Defaults to `<agent_install_directory>/logs/appd-sdk`. You need to pass this path to the `runProxy` script if you are starting the proxy manually.
Troubleshoot Apache Agent Installation

On this page:
- Undefined Symbol Error
- App Crash Caused by libstdc++ Mismatch

If you installed the Apache Agent and are experiencing problems, try these suggestions for investigating installation issues.

**Undefined Symbol Error**

An error message similar to the following typically indicates that you have loaded the incorrect agent library for the version of Apache you are using.

```
ERROR
httpd: Syntax error on line 953 of
/srv/jas/app/embcms/HTTPServer/cma1-dev2/conf/httpd.conf: Syntax error on line 7 of
/srv/jas/app/embcms/HTTPServer/cda0-dev2/conf/appdynamics.conf:
Cannot load
/srv/jas/data/AppDynamics/WebAgent/WebServerAgent/Apache/libmodappdynamics.so into server:
/srv/jas/data/AppDynamics/WebAgent/WebServerAgent/Apache/libmod_appdynamics.so: undefined symbol: ap_log_error
```

Check the description of `LoadModule` on Install the Apache Agent to make sure that you are loading the correct module. For example, for Apache 2.2, your LoadModule directive should be:

```
LoadModule appdynamics_module
/srv/jas/data/AppDynamics/WebAgent/ApacheAgent/Apache/libmod_appdynamics22.so
```

**App Crash Caused by libstdc++ Mismatch**

The AppDynamics agent uses libstdc++ v6. If your application uses a different version of libstdc++ and the modules were not loaded in the correct order, the wrong library routines could be called. If you get a dump with a long stack trace, this could be the cause. See Apache with libstdc++5 Considerations for information on how to prevent this.
Apache with libstdc++5 Considerations

On this page:

- Check for libstdc++5 Dependencies
- Resolution Overview
- Identify the Paths to libstdc++.so.6
- Determine the Correct libstdc++.so.6 Path for the Installation
- Add the Export LD_PRELOAD Command

Related pages:

- Install the Apache Agent

The Apache Agent depends on libstdc++6. Some servers, primarily IHS and OHS, have modules built with libstdc++5. This conflict can prevent Apache from starting successfully.

Check for libstdc++5 Dependencies

If you do not know if your web server has libstdc++5 dependencies, find out as follows:

1. Change directory to your web server’s modules directory.
2. Run this command:

   ```bash
   ldd *.so 2>&1 | grep -v 'execution permission' | grep 'libstdc++.so.5'
   ```

3. If this command returns any results—for example, `libstdc++.so.5 => /lib64/libstdc++.so.5`—your server has a dependency on libstdc++.so.5.

Resolution Overview

To avoid libstdc++5 conflicts, force libstdc++6 to load before libstdc++5 when Apache is started. The general steps for performing this configuration are:

1. Identify the paths to libstdc++56.
2. Determine the correct libstdc++56 path for your installation.
3. Use the LD_PRELOAD environment variable to have libstdc++6 preloaded.

These steps are detailed in the following sections.

Identify the Paths to libstdc++56

From a command prompt, run:

```bash
ldconfig -p | grep libstdc++56
```

This will return one or more paths. For example:

```
/lib64/libstdc++56
/lib/libstdc++56
```

Determine the Correct libstdc++56 Path for the Installation

If more than one path is returned, determine which is the correct path for your Apache Server installation.

If you are running a 64-bit operating system, but a 32-bit version of IHS or OHS, choose the 32-bit version, which in our example would be:
Add the Export LD_PRELOAD Command

Open the envvars** file, which is located in the /bin directory of your IHS or OHS installation. Add the following line, using the path to the libstdc++.so.6 path that you determined in the previous step.

```
export LD_PRELOAD=<path-to-libstdc++.so.6>
```

This will cause libstdc++.so.6 to be loaded before libstdc++.so.5, and the conflict will be averted.
Upgrade the Apache Agent

To upgrade the Apache Agent, back up the current agent install directory elsewhere. Then download and install the new agent into the same directory as the old one was.

Then follow these steps to upgrade the agent:

1. **Kill the proxy process.**
   
   If you do not know the PID of the `proxy` process you can get it by running this command:
   
   ```
   ps -aux | grep "proxy"
   ```
   
   Then kill the process as follows:
   
   ```
   kill <PID>
   ```

2. Shut down the web server.
3. Move/rename the agent install directory, `/opt/appdynamics-sdk-native` to make a backup.
4. Download the new version of the agent and extract it into `/opt/appdynamics-sdk-native`.
5. Change directory to the agent install directory:
   
   ```
   cd <agent_install_directory>
   ```

6. Run the installer:
   
   ```
   ./install.sh
   ```

   It is not necessary to change the configuration when you use the same directory location as the old agent installation.

7. Restart the web server.
Uninstall the Apache Agent

1. Remove the include statement for the AppDynamics Apache Agent from the HTTP server configuration. For example in httpd.conf—on Ubuntu and Debian, this file is called apache2.conf—remove this line:

```
Include conf/appdynamics_agent.conf
```

2. Restart the web server.

3. Optionally remove the following files:
   - `libmod_appdynamics.so`—for Apache 2.4.x—or `libmod_appdynamics22.so`—for Apache 2.2.x—from the web server modules directory, if your administrator put it there
   - `appdynamics_agent.conf` or similar from the web server configuration directory

4. Kill the proxy process.

5. Uninstall the agent:

```
<agent_install_directory>/install.sh -u
```

6. Optionally remove any remaining AppDynamics files:
   - `<agent_install_directory>`
   - `/tmp/appd-sdk`
Apache Agent Logging

On this page:
- Agent Log
- Proxy Log

Related pages:
- Dynamic Agent Proxy Logging
- Dynamic Language Agent Proxy

The Apache agent generates two types of logs: the agent log and the proxy log, as described here.

Agent Log

The log contains information about the transactions that the agent processes and sends to the proxy.

The log is located at $<apache_agent_install>/logs/agent.log. By default this is /opt/appdynamics-sdk-native/logs/agent.log.

The default pattern for agent log naming is the following:

- agent.log: the current log
- agent.log.1: most recent log
- agent.log.2: second most recent log
- agent.log.3: third most recent log
- agent.log.4: fourth most recent log
- agent.log.5: fifth recent log.

The agent creates and rotates a maximum of six log files. Maximum log size is 20MB, which gives you a maximum of the most recent 120MB of log data at one time.

Proxy Log

The proxy log contains information about the transactions between the proxy and the Controller.

The proxy log is located at $<apache_agent_install>/logs/proxy_$date.log.

See Dynamic Agent Proxy Logging for details.
The AppDynamics Agent SDK for C/C++ lets you monitor the performance of C and C++ applications with AppDynamics. The SDK provides transaction and backend reporting, automatic tier mapping, automatic dynamic baselining, health rules, data collectors, and transaction snapshots. Together, these tools give you visibility on application load and response times, and any custom metrics you define.

You monitor C and C++ applications by adding a few API calls to the source code.

Once running, the agent registers business transactions with the AppDynamics Controller. You can then see your application flow map monitor performance in the Controller.

**C/C++ Agent Supported Platforms**

**Operating Systems**

- Any Linux distribution based on glibc 2.5+
- Windows Server 2008 R2 and above

**Install the SDK on Linux**

1. Download the C/C++ SDK from https://appdynamics.com/download/
2. Extract the SDK to the desired SDK home directory location, such as in /opt:

   ```bash
   sudo tar xvzf appdynamics-sdk-native-64bit-linux-VERSION.tar.gz -C /opt
   ```

   Replace `VERSION` in the file name with the version of the downloaded agent SDK.
3. Update your build scripts to be able to access the AppDynamics SDK library files:
   a. Add `/opt/appdynamics-cpp-sdk/lib` to your library path. For example, `-L/opt/appdynamics-cpp-sdk/lib`.
   b. Add `-lappdynamics` to your link options.
4. Modify your program to use the AppDynamics SDK. See C/C++ Agent SDK Reference and Using the C/C++ Agent SDK.
5. Ensure that the ulimit setting for open file descriptors on the operating system is set to a sufficient number. At a minimum you will need two file descriptors for each active thread that calls the C++ SDK `appd_sdk_init()` function.

**Install the SDK on Windows**

1. Download C/C++ SDK from https://appdynamics.com/download/. Be sure to use the SDK that corresponds to the bit number for your operating system, whether 32-bit or 64-bit. Using the SDK version that doesn't match the bitness of the OS causes a linker error about missing symbols.
2. Extract the downloaded SDK zip file to the directory where you want to place the SDK, such as `C:\AppDynamics`
3. Add the SDK lib directory to the system PATH, for example, add `C:\AppDynamics\appdynamics-cpp-sdk\lib` to `%PATH%`.
4. Configure your Visual Studio project to have the include path and link the AppDynamics library. Open Project > Properties.
   a. From the sidebar, unfold **C/C++** then choose **General**. In the new pane, change Additional Include Directories to add:
C:\AppDynamics\appdynamics-cpp-sdk\include

b. From the sidebar, unfold **Linker** then choose **Input**. In the new pane, change **Additional Dependencies** to add: C:\AppDynamics\appdynamics-cpp-sdk\lib\appdynamics.lib

5. Modify your program to use the AppDynamics SDK. See C/C++ Agent SDK Reference and Using the C/C++ Agent SDK.
C/C++ Agent Supported Environments

C/C++ Agent Supported Platforms

*Operating Systems*

- Any Linux distribution based on glibc 2.5+
- Windows Server 2008 R2 and above
  - Visual Studio 2015, or later
Before you can start instrumenting your applications with the C/C++ SDK, you need to get the current version of the C/C++ SDK. The C/C++ Application Agent topic describes how to download and install the SDK.

Once you have the SDK, you can instrument your application as described here. The SDK provides routines for creating and managing business transactions, transaction snapshots, backends, exit points and collecting custom metrics. This topic provides an overview of these concepts.

Task Flow to Use the SDK

These are the tasks your application needs to perform to get instrumented:

- Add the AppDynamics Header File to the Application
- Initialize the Controller Configuration
- Initialize the Agent
- Create Business Transactions
- Manage Business Transaction Snapshots
- Create Backends
- Manage Exit Calls
- Create Custom Metrics
- APIs for Resource Management (C++ Only)
- Generating Call Graphs (New in 4.5.1)
- Generating Pre-populated Call Graphs (C++ only)
- Analytics Support for C++ SDK

Add the AppDynamics Header File to the Application

After downloading the SDK, you are ready to add AppDynamics instrumentation to your C/C++ application. The first step is to include the AppDynamics header file at the top of your `main` function file:

```
#include <path_to_SDK>/sdk_lib/appdynamics.h
```

Initialize the Controller Configuration

Controller information settings permit the agent to connect to the Controller. Some settings are required for all applications, while others are needed only for certain types of application environments. For example, if the agent needs to connect to the Controller via an internal proxy in your network, you need to set up the connection settings for the proxy.
See ‘Agent Configuration Settings’ in C/C++ Agent SDK Reference for a complete list of the settings and information about which ones are required.

In your application, assign values to the required settings. For example, to set the Controller connection information:

```c
const char APP_NAME[] = "SampleC";
const char TIER_NAME[] = "SampleCTier1";
const char NODE_NAME[] = "SampleCNode1";
const char CONTROLLER_HOST[] = "controller.somehost.com";
const int CONTROLLER_PORT = 8080;
const char CONTROLLER_ACCOUNT[] = "customer1";
const char CONTROLLER_ACCESS_KEY[] = "MyAccessKey";
const int CONTROLLER_USE_SSL = 0;
```

Declare an `appd_config` struct with the Controller settings. For example:

```c
struct appd_config* cfg = appd_config_init(); // appd_config_init()
resets the configuration object and pass back an handle/pointer
appd_config_set_app_name(cfg, APP_NAME);
appd_config_set_tier_name(cfg, TIER_NAME);
appd_config_set_node_name(cfg, NODE_NAME);
appd_config_set_controller_host(cfg, CONTROLLER_HOST);
appd_config_set_controller_port(cfg, CONTROLLER_PORT);
appd_config_set_controller_account(cfg, CONTROLLER_ACCOUNT);
appd_config_set_controller_access_key(cfg, CONTROLLER_ACCESS_KEY);
appd_config_set_controller_use_ssl(cfg, CONTROLLER_USE_SSL);
```

Initialize the Agent

In your `main` function, call these initialization functions:

1. Initialize the `config` structure as shown in the previous section.
2. Next initialize the agent by passing the configuration structure to `appd_sdk_init()`

If `appd_sdk_init()` returns zero, the agent is initialized successfully. Non-zero indicates failure, probably because the agent could not reach the Controller.

The following example illustrates how to initialize the SDK:

```c
int initRC = appd_sdk_init(cfg);
if (initRC) {
    std::cerr << "Error: sdk init: " << initRC <<;
    return -1;
}
```

When using `fork()`, the `appd_sdk_init()`, and all other SDK calls, should occur only in the forked process, never in the parent, and each child should behave as its own process, and never share handles between the process itself and other SDK instrumented child processes.
Create Business Transactions

Define a business transaction by enclosing the code that constitutes the request that you want to monitor between `appd_bt_begin()` and `appd_bt_end()` calls. `appd_bt_begin()` returns a handle to use in subsequent routines that affect that business transaction.

If you are creating a business transaction that correlates with an upstream business transaction, pass the correlation header of the upstream transaction so the new transaction that you are creating can correlate with it. See `appd_exitcall_get_correlation_header()` in the C/C++ Agent SDK Reference. If the transaction does not need to correlate with another transaction, pass `NULL` for the correlation header parameter.

You can optionally store the business transaction handle in the global handle registry with a guid for easy retrieval later using `appd_bt_store()`. Retrieve the handle from the global handle registry using `appd_bt_get()`.

The following shows an example of setting a business transaction:

```c
// start the "Checkout" transaction
appd_bt_handle btHandle = appd_bt_begin("Checkout", NULL);

// Optionally store the handle in the global registry
appd_bt_store(btHandle, my_bt_guid);
...

// Retrieve a stored handle from the global registry
appd_bt_handle myBtHandle = appd_bt_get(my_bt_guid);
...

// end the transaction
appd_bt_end(btHandle);
```

Between starting and ending the transaction, you can perform operations such as adding errors to the business transaction, defining the transaction snapshot attributes, adding backends and exit calls, and so on.

When the business transaction ends, via a call to `appd_bt_end()`, the agent stops reporting metrics for the business transaction.

Add Business Transaction Errors

Use `appd_bt_add_error()` to report business transaction errors. If you set this function parameter, `markBTAsError` the transaction is reported as an error transaction when the error occurs. For the function `{{appd_bt_add_error,}}`, the Controller database truncates log messages that exceed 5,000 characters.

The SDK provides an `enum` to classify the error level as `APPD_LEVEL_NOTICE`, `APPD_LEVEL_WARNING` or `APPD_LEVEL_ERROR`.

Manage Business Transaction Snapshots

When the agent is monitoring a business transaction, it automatically creates transaction snapshots, which describe instances of the business transaction at certain points in time. Transaction snapshots are extremely useful for troubleshooting poor performance because they contain a lot of detail.

You do not have to modify anything to create these snapshots, other than create the business transaction, but you can add calls to:

- Find out if a snapshot is being taken
- Provide additional data in a snapshot
- Set the URL for a snapshot

**Determine if the Agent is Taking a Snapshot Now**
The agent is not always collecting snapshots, because that would be expensive. By default, it collects a snapshot every ten minutes, but this schedule is configurable. See ‘Configure Snapshot Periodic Collection Frequency’ in Transaction Snapshots for information about the frequency.

You can determine if a snapshot is happening using `appd_bt_is_snapshotting()`, which returns non-zero if the agent is collecting a snapshot. The main reason to do this is to avoid the wasted overhead for collecting user data for a snapshot or setting the snapshot URL if no snapshot is currently being collected.

### Add Business Transaction User Data

You can optionally add data to transaction snapshots. For example, you might want to know which users are getting a lot of errors, from which regions users are experiencing slow response times or which methods are slow. In the AppDynamics UI, the data appears in the USER DATA tab of the transaction snapshot.

If a snapshot is occurring, use `appd_bt_add_user_data()` passing a key and value for the data that you want the snapshot to collect. For the function `{{appd_bt_add_user_data,}}`, the Controller database truncates log messages that exceed 5,000 characters.

### Add Snapshot URL

A snapshot URL lets Controller users share a snapshot with others. You can set a URL for the current snapshot using `appd_bt_set_url()`.

### Set Snapshot Example

```c
void setSnapshotAttributes(appd_bt_handle btHandle, int minute, int halfsec)
{
    // do this only if the agent is collecting a snapshot
    if (appd_bt_is_snapshotting(btHandle))
    {
        char nameBuf[30];
        char valueBuf[30];
        // add custom data to the snapshot
        snprintf(nameBuf, sizeof(nameBuf), "BT:%p
", btHandle);
        snprintf(valueBuf, sizeof(valueBuf), "Minute:%d
Second:%d
", minute, halfsec/2);
        appd_bt_add_user_data(btHandle, nameBuf, valueBuf);

        static int snapCount = 0;
        int switchVal = snapCount % 4;

        // report errors, but only ERROR_LEVEL errors are marked as error transactions
        if (switchVal)
        {
            appd_error_level errorLevel;
            bool markBtAsError;
            switch (switchVal)
            {
                case 1:
                    errorLevel = APPD_LEVEL_NOTICE;
                    markBtAsError = false;
                    snprintf(nameBuf, sizeof(nameBuf), "NOTICE BT:%p M:%d S:%d
", btHandle, minute, halfsec/2);
                    break;
```
case 2:
    errorLevel = APPD_LEVEL_WARNING;
    markBtAsError = false;
    snprintf(nameBuf, sizeof(nameBuf), "WARNING BT:%p M:%d S:%d\n", btHandle, minute, halfsec/2);
    break;
    case 3:
    errorLevel = APPD_LEVEL_ERROR;
    markBtAsError = true;
    snprintf(nameBuf, sizeof(nameBuf), "ERROR BT:%p M:%d S:%d\n", btHandle, minute, halfsec/2);
    break;
)
    appd_bt_add_error(btHandle, errorLevel, nameBuf,
    markBtAsError, markbtaserror);
}
    snapCount++;
    // set the snapshot url
    snprintf(nameBuf, sizeof(nameBuf), "http://bt-%p.com", nameBuf, btHandle);
Create Backends

A backend is a database or a remote service such as a message queue, HTTP service or cache service that your application uses. A backend component is not itself monitored by the application agent, but the agent monitors calls to it from instrumented servers. You need to create backends in the instrumented environment so that the agent can discover them. This involves:

- Declaring the backend
- Setting its identifying properties
- Optionally configuring how the backend is presented in the AppDynamics UI
- Adding the backend to the instrumented application

Declare the Backend

You must declare a backend using `appd_backend_declare()` before the agent can detect it. After you declare a backend, you don't need to declare it again if the backend is used by other business transactions in a single SDK instance. A backend must be of one of the supported types listed under 'Exit Call Types' in the Agent SDK for C and C++ Reference.

Identify the Backend

A backend also has identifying properties, which you set using `appd_backend_set_identifying_property()`. The properties vary depending on the type of the backend and the types of information that you want to display. The Controller displays identifying properties in backend dashboards. You must set at least one identifying property for the type of any backend that you plan to add.

The following shows the backend properties in the Controller UI for an ActiveMQ:

Resolve to a Tier

By default, the Controller doesn't display a detected backend as a separate entity in flowmaps, but the agent reports its metrics as part of the downstream tier. If you want to display the backend as a separate component and not resolved to the tier use `appd_backend_prevent_agent_resolution()`. See Resolve Remote Services to Tiers for more information.

Add to Application

After you declare and configure the backend, add it to the application so the agent can detect it.
**Backend Example**

The following listing shows an example of setting up a database backend.

```c
// declare a backend, only once for this SDK instance
const char backendOne[] = "first backend";
appd_backend_declare(APPD_BACKEND_HTTP, backendOne);

// set the host property
rc = appd_backend_set_identifying_property(backendOne, "HOST", "sqs-us-west-hostname");
if (rc) {
    std::cerr << "Error: appd_backend_set_identifying_property: " <<
    rc << ".";
    return -1;
}

// do not resolve the backend to the tier
rc = appd_backend_prevent_agent_resolution(backendOne);
if (rc) {
    std::cerr << "Error: appd_backend_prevent_agent_resolution: " <<
    rc << ".";
    return -1;
}

// add the backend
rc = appd_backend_add(backendOne);
if (rc) {
    std::cerr << "Error: appd_backend_add: " << rc << ".";
    return -1;
}
```

**Manage Exit Calls**

When an application makes a call to another component—a detected backend or another application server—the agent reports metrics on those calls.

Define an exit call by enclosing the code that constitutes the exit call between `appd_exitcall_begin()` and `appd_exitcall_end()` calls. `appd_exitcall_begin()` returns a handle to use in subsequent routines that affect that exit call. An exit call occurs in the context of a business transaction.

You can optionally store the exit call handle in the global handle registry with a guid for easy retrieval later using `appd_exitcall_store()`. Retrieve the handle from the global handle registry using `appd_exitcall_get()`.

Pass the business transaction handle and the destination backend to `appd_exitcall_end()`.

You can optionally add details to an exit call as any arbitrary string. The details are reported in the exit call details in the transaction snapshots in the Controller UI:
You can also add errors to an exit call using `appd_exitcall_add_error()`. Use the enum for the error levels. You can also add an error message.

**Simple Exit Call Example**

```c
// start the exit call to backendOne
appd_exitcall_handle ecHandle = appd_exitcall_begin(btHandle, backendOne);
...

// optionally store the handle in the global registry
appd_exitcall_store(ecHandle, my_ec_guid);
...

// retrieve a stored handle from the global registry
appd_exitcall_handle myEcHandle = appd_exitcall_get(my_ec_guid);

// set the exit call details
rc = appd_exitcall_set_details(ecHandle, "backend ONE");
if (rc) {
    std::cerr << "Error: exitcall details1"
    return -1;
}

// add an error to the exit call
appd_exitcall_add_error(ecHandle, APPD_LEVEL_ERROR, "exitcall1 error!", true);

// end the exit call
appd_exitcall_end(ecHandle)
```

**Correlate with Other Business Transactions**

A correlation header contains the information that enables the agents to continue the flow of a business transaction across multiple
tiers.

An SDK agent can correlate with other SDK agents as well as other AppDynamics agents—such as Java, .NET or PHP—that perform automatic correlation for certain types of entry and exit points.

Correlate with an Upstream Tier

When your instrumented process receives a continuing transaction from an upstream agent that supports automatic correlation:

1. Using a third-party `http_get_header()` function, extract the header named `APPD_CORRELATION_HEADER_NAME` from the incoming HTTP payload.
2. Pass the header to `appd_bt_begin()`. For example:

   ```c
   const char* hdr = http_get_header(req, APPD_CORRELATION_HEADER_NAME);
   appd_bt_handle bt = appd_bt_begin("fraud detection", hdr);
   ```

   If the header retrieved by the `http_get_header()` function is valid, the business transaction started by the `appd_bt_begin()` call will be a continuation of the business transaction started by the upstream service.

Correlate with a Downstream Tier

The downstream agent is watching for a correlation header named `singularityheader` in the HTTP payload.

If your SDK agent is making an exit call to a downstream agent that supports automatic correlation:

1. Set the name of the correlation header using `APPD_CORRELATION_HEADER_NAME`.
2. Begin the exit call.
3. Retrieve the correlation header from the exit call using the `appd_exitcall_get_correlation_header()` function.
5. Inject a header named `APPD_CORRELATION_HEADER_NAME` with the value of the correlation header retrieved in step 3 into the outgoing payload of the HTTP request.
6. Make the request. For example:

   ```c
   const char* const APPD_CORRELATION_HEADER_NAME = "singularityheader";
   appd_exitcall_handle inventory = appd_exitcall_begin(bt, "inventory");
   const char* hdr = appd_exitcall_get_correlation_header(inventory);
   http_request req;
   http_init(&req, HTTP_POST, "https://inventory/holds/%s", sku);
   http_set_header(&req, APPD_CORRELATION_HEADER_NAME, hdr);
   http_perform(&req);
   ...
   ```

cURL Downstream Correlation Example

The following example uses the cURL library to correlate with a downstream transaction.

```c
#include <curl/curl.h>
```
// get the correlation header into the response
class LibcurlURLGet
{
public:
LibcurlURLGet(const std::string& url) : url(url),
curlHandle(NULL) {}
std::string GetResponse(const char* correlationHeader)
{
CURLcode res;
Response response;
curlHandle = curl_easy_init();
if (curlHandle)
{
curl_easy_setopt(curlHandle, CURLOPT_URL, url.c_str());
curl_easy_setopt(curlHandle, CURLOPT_WRITEFUNCTION,
WriteDataCallback);
curl_easy_setopt(curlHandle, CURLOPT_WRITEDATA,
&response);
if(correlationHeader && strlen(correlationHeader))
{
// start new added code
std::string
singularityHeader(APPD_CORRELATION_HEADER_NAME);
singularityHeader.append(": ");
singularityHeader.append(correlationHeader);
// end new added code
curl_slist* slistNewHeaders = 0;
slistNewHeaders =
curl_slist_append(slistNewHeaders, correlationHeader);
curl_easy_setopt(curlHandle, CURLOPT_HTTPHEADER,
slistNewHeaders);
}
res = curl_easy_perform(curlHandle);
if(res != CURLE_OK)
{
}
}
curl_easy_cleanup(curlHandle);
return response.GetResponseString();
}
private:
class Response
{
public:
std::string GetResponseString()
{

Copyright © AppDynamics 2012-2019

Page 475


return sResponse;
}
void AppendResponse(void* ptr, size_t sizeBytes)
{
    sResponse.append((char*)ptr, sizeBytes);
}
private:
    std::string sResponse;
};
static size_t WriteDataCallback(void* ptr, size_t size, size_t nmemb, Response* response)
{
    response->AppendResponse(ptr, size * nmemb);
    return size * nmemb;
}
std::string url;
CURL* curlHandle;
};
...
// start an exit call from the current transaction
appd_exitcall_handle ecHandle1 = appd_exitcall_begin(btHandle1, tier2);
...
// before exiting get the correlation header
const char* corrHeader1 = appd_exitcall_get_correlation_header(ecHandle1);

// put the correlation in the HTTP response
std::string response = urlGet.GetResponse(corrHeader1);

...  
// start the next business transaction to correlate with the previous one
Create Custom Metrics

You can create custom metrics with the C/C++ SDK. Custom metrics supplement the built-in metrics you get with the C/C++ agent, such as the BT call count and response time.

To create custom metrics, you use two calls in the API:

- appd_custom_metric_add()
- appd_custom_metric_report()

You use the `appd_custom_metric_add()` to declare a metric. The method signature is:

```c
APPD_API void appd_custom_metric_add (const char * application_context,
                                      const char *  metric_path,
                                      enum appd_time_rollup_type
                                      time_rollup_type,
                                      enum appd_cluster_rollup_type
                                      cluster_rollup_type,
                                      enum appd_hole_handling_type
                                      hole_handling_type)
```

The following example illustrates a call that uses this method:

```c
appd_custom_metric_add("app_context", "Custom Metrics|Memory|Total Memory Usage",
APPD_TIMEROLLUP_TYPE_AVERAGE, APPD_CLUSTERROLLUP_TYPE_INDIVIDUAL,
APPD_HOLEHANDLING_TYPE_RATE_COUNTER);
```

As shown, the method passes an application context for the metric, the name of the metric—Total Memory Usage, in the example—and the path to the metric in the Metric Browser tree—Custom Metrics > Memory. The other parameters define how the metric is processed by the Controller.

With the metric declared, your code can then report data to the metric with the `appd_custom_metric_report()` method, as follows:

```c
appd_custom_metric_report("app_context", "Custom Metrics|Memory|Total Memory Usage", 1234);
```

The Controller applies some standard processing functions to custom metric values just as it does to built-in metrics. These functions define, for example, how the metric is rolled up over time. The manner in which the Controller should process a custom metric depends on the nature of that metric. For instance, for a metric representing the state of the machine, it may make sense to roll up the value to the last observed value for the metric. For a call rate metric, it may make sense to roll the metric up to an average value.

The following sections provide more information on the parameters you can use to define metric handling by the Controller.
**Time Rollup Type**

The `time_rollup_type` parameter tells the Controller how to roll up values for the metric over time. There are three ways in which the Controller can roll up metrics, as follows:

- **Average Response Time** may be calculated as the average of all reported values in the time period. An example of a built-in metric that uses this is the `Average Response Time` metric.
- **Sum of all reported values in that minute.** This operation behaves like a counter. An example metric is the `Calls per Minute` metric.
- **Current** is the last reported value in the minute. If no value is reported in that minute, the last reported value is used. An example of a metric that uses this would be a machine state metric, such as the `Max Available (MB)` metric.

**Cluster Rollup Type**

The `appd_cluster_rollup_type` parameter tells the Controller how to aggregate metric values for the tier—a cluster of nodes.

- **Individual**: Aggregates the metric value by averaging the metric values across each node in the tier. For example, `Hardware Resources|Memory|Used %` is a built-in metric that uses the individual rollup type.
- **Collective**: Aggregates the metric value by adding up the metric values for all the nodes in the tier. For example, `Agent|Metric Upload|Metrics uploaded` is a built-in metric that uses the collective rollup type.

**Hole Handling Type**

A particular metric may not report data for a given minute. The `appd_hole_handling_type` parameter tells the Controller how to set the metric count for that time slice. The count is set to zero if the hole handling type is `REGULAR_COUNTER`, and set to one if `RATE_COUNTER`. In effect, `REGULAR_COUNTER` does not affect aggregation while `RATE_COUNTER` does.

For example, consider four time slices with the data 9, 3, 0, 12. Notice that the third time slice is zero-filled, with no metric being reported for that time slice. The sum of the values is 9+3+0+12 = 24. If the metric is `REGULAR_COUNTER`, the count for the third time slice would be zero, and therefore the overall count would be 1+1+1+1 = 4. In the case of the `REGULAR_COUNTER`, the average would be 24/3 = 8, while for the `RATE_COUNTER` the average would be 24/4 = 6.

Built-in metrics that use the regular counter hole-handling type include `Average Response Time`, `Average CPU Used`, `Number of Stalls`. Built-in metrics that use the rate counter hole-handling type include `BT Calls per minute` and `Errors per minute`.

**APIs for Resource Management (C++ Only)**

For C++ applications, you can use the `BT` and `ExitCall` classes to simplify your instrumentation.

The advantage of using these classes is that you do not need to manually call `appd_bt_end()` and `appd_exitcall_end()`. When the `BT` or `ExitCall` object goes out of scope, the SDK calls its destructor and the business transaction or exit call automatically ends.

For example, the following code using these classes:

```cpp
{
    appd::sdk::BT bt("mybt");
    appd::sdk::ExitCall ec(bt, "auth");
    if (!make_auth_call())
    {
        bt.add_error(APPD_LEVEL_ERROR, "Authorization failed");
        return -1;
    }
    issue_api_call();
}
```

is the equivalent of this:
appd_bt_handle bt = appd_bt_begin("mybt", NULL);
appd_exitcall_handle ec = appd_exitcall_begin(bt, "auth");
if (!make_auth_call())
{
    appd_bt_add_error(bt, APPD_LEVEL_ERROR, "Authorization failed", 0);
    appd_exitcall_end(ec);
    appd_bt_end(bt);
    return -1;
}
issue_api_call();
appd_exitcall_end(ec);
appd_bt_end(bt);

When the BT lifetime depends on the nondeterministic lifetimes of other objects, you can use a shared pointer to a BT to keep the BT alive for the lifetimes of its dependencies. In this case, the BT ends when the last reference to it ends. For example:

```
//Initialize the BT with a shared pointer
auto bt = std::make_shared<appd::sdk::BT>("compute");
auto prod = createProducer(bt);
auto consumers = createConsumers(bt, NUM_WORKERS);
//Variable BT goes out of scope with no further references.
//BT ends automatically.
```

For managing exit calls with complex lifetimes, consider using smart pointers `std::unique_ptr<appd::sdk::ExitCall>` or `std::shared_ptr<appd::sdk::ExitCall>`.

**Generating Call Graphs (New in 4.5.1)**

You can use the C/C++ SDK to instrument methods so that they are reported and displayed in the call graph. You can instrument methods in one of the following ways:

- Using the `APPD_AUTO_FRAME` macro, C++ only
- Using `appd_frame_begin` and `appd_frame_end`

**Instrumenting methods using the `APPD_AUTO_FRAME` macro**

You can use the `APPD_AUTO_FRAME` macro to instrument applications built on C++ frameworks. The macro is equivalent to calling the `appd_frame_begin` and `appd_frame_end` methods.

1. Specify the business transaction that you want to instrument calls for, and configure the root of the call graph.
2. Instrument the methods as shown below.

```cpp
void method1(appd::sdk::BT& bt)
{
    APPD_AUTO_FRAME(bt);
    // Code for method1...
}
void method2(appd::sdk::BT& bt)
{
    APPD_AUTO_FRAME(bt);
    // Code for method2...
}
```

Note that the methods must have access to the business transaction so the business transaction can be passed into the method as an argument.

**Instrumenting methods using the appd_frame_begin and appd_frame_end functions**

You can instrument methods for applications that are built on non-C++ frameworks. To instrument a method, you call `appd_frame_begin` at the beginning of the method, and `appd_frame_end` at the end of the method. When you do this, the duration of the method call is automatically calculated and reported in the call graph.

If your method makes an exit call, you can instrument the exit call by wrapping the exit call code between `appd_exitcall_begin` and `appd_exitcall_end`.

The code samples below show three methods that are each instrumented using `appd_frame_begin` and `appd_frame_end`, and then called in a root method, `main()`. The second sample method, `method2()`, contains an exit call.

```cpp
void method1()
{
    appd_frame_handle frameHandle = appd_frame_begin(btHandle,
            APPD_FRAME_TYPE_CPP, nullptr, APPD_FUNCTION_NAME, __FILE__,
            __LINE__);
    // code for method1
    appd_frame_end(btHandle, frameHandle);
}
```
```c
void method2()
{
    appd_frame_handle frameHandle = appd_frame_begin(btHandle,
    APPD_FRAME_TYPE_CPP, nullptr, APPD_FUNCTION_NAME, __FILE__,
    __LINE__); // code for method2
    appd_exitcall_handle exitCallHandle = appd_exitcall_begin(btHandle,
    BACKEND_NAME); // code for exit call
    appd_exitcall_end(exitCallHandle);
    appd_frame_end(btHandle, frameHandle);
}
```

```c
void method3()
{
    // To illustrate instrumenting a method where this SDK cannot be used
    // method3 is a wrapper for the call of the actual method which will show up in the call graph
    appd_frame_handle frameHandle = appd_frame_begin(btHandle,
    APPD_FRAME_TYPE_CPP, "Test", "Compute",
    "C:\modules\source\Test.cs", 143); // call the wrapped method
    appd_frame_end(btHandle, frameHandle);
}
```

The `main()` method below is the root of the call graph. It specifies the business transaction that invokes the three methods above by passing the transaction into `appd_bt_begin` and `appd_bt_end`. 

```c
```
int main(int argc, char **argv)
{
    // initialize AppDynamics
    btHandle = appd_bt_begin(TRANSACTION_NAME, "");
    appd_frame_handle frameHandle = appd_frame_begin(btHandle, APPD_FRAME_TYPE_CPP, nullptr, APPD_FUNCTION_NAME, __FILE__, __LINE__);
    // code for main
    method1();
    method2();
    method3();
    appd_frame_end(btHandle, frameHandle);
    appd_bt_end(btHandle);
}

The sample methods above are represented by the call graph shown below:

<table>
<thead>
<tr>
<th>Name</th>
<th>Time (ms)</th>
<th>Percent %</th>
<th>Exit Calls / Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>c:\source\main.cpp:230</td>
<td>41 ms</td>
<td>10.2%</td>
<td></td>
</tr>
<tr>
<td>c:\source\main.cpp:173</td>
<td>120 ms</td>
<td>29.9%</td>
<td></td>
</tr>
<tr>
<td>c:\source\main.cpp:180</td>
<td>160 ms</td>
<td>39.8%</td>
<td>HTTP</td>
</tr>
<tr>
<td>C:\modules\source\Test.cs:143</td>
<td>81 ms</td>
<td>20.1%</td>
<td></td>
</tr>
</tbody>
</table>

Generating Pre-populated Call Graphs (C++ only)

The previous section describes instrumenting your C/C++ application by calling the instrumentation methods in your code. However, if you do not have the option of modifying your source code, you can generate call graphs that are populated with data from sources such as log files.

To generate a pre-populated call graph, follow these steps:

1. Create the root of the call graph by instantiating a CallGraph class. The CallGraph class takes the following parameters:
   - bt: The business transaction
   - class_name: The name of the class that contains the root of the call graph
   - method_name: The name of the method that represents the root of the call graph
   - file_path: The file path to the class
   - line_number: The line number of the method
   - time_msec: The time taken by this frame (method) in milliseconds.
   - frame_type: The type of the frame (the language for the method). Currently, APPD_FRAME_TYPE_CPP is the only option.

The sample below shows a CallGraph instantiation.

```cpp
appd::sdk::CallGraph callGraph(bt, "Class1", "main", "/src/file1.cc", 276, 100, APPD_FRAME_TYPE_CPP);
```
2. Create the call graph tree by calling `add_child` on the root and on any of the added children, as shown below.

```cpp
callGraph.root().add_child("Class2", "method1", "/src/file2.cc"), 101, 40, APPD_FRAME_TYPE_CPP)
  .add_child("Class2", "method2", "/src/file2.cc"), 523, 30, APPD_FRAME_TYPE_CPP); auto& cge1 = callGraph.root()
  .add_child("Class3", "method1", "/src/file3.cc"), 27, 30, APPD_FRAME_TYPE_CPP); cge1.add_child("Class3", "method2", "/src/file3.cc"), 430, 15, APPD_FRAME_TYPE_CPP);
```

3. Call `add_to_snapshot` on the call graph. For this to work, the business transaction must be snapshotting.

```cpp
callGraph.add_to_snapshot();
```

Analytics Support for C++ SDK

Two of the existing API methods, which were previously only used for adding data to a snapshot, will now send Analytics data if analytics is enabled for the current Business Transaction on the Controller (in the web interface):

Set URL for a snapshot – URL is set for a snapshot if one is occurring. Data should be either 7-bit ASCII or UTF-8. You can call this function when a snapshot does not occur. When the given Business Transaction is not snapshotting, this function returns immediately. However, if extracting the data to pass to this function is expensive, you can use `appd_bt_is_snapshotting` to check if the Business Transaction is snapshotting before extracting the data, and calling this function.

```cpp
APPD_API void appd_bt_set_url(appd_bt_handle bt, const char* url);
```

- param bt: The business transaction to add the user data to, if it is taking a snapshot
- param url: The value of the URL for the snapshot as 7-bit ASCII or UTF-8

Add user data to a snapshot – data should be either 7-bit ASCII or UTF-8. You can call this function when a snapshot does not occur or, analytics is not enabled. When the data is only for snapshotting, and if extracting the data to pass to this function is expensive, you can use `appd_bt_is_snapshotting` to check if the Business Transaction is snapshotting before extracting the data, and calling this function.

```cpp
APPD_API void appd_bt_add_user_data(appd_bt_handle bt, const char* key, const char* value);
```

- param bt: The business transaction to add the user data to, if it is taking a snapshot
- param key: The name of the user data to add to the snapshot as 7-bit ASCII or UTF-8
- param value: The value of the user data to add to the snapshot as 7-bit ASCII or UTF-8

Note that these API methods existed previously, but their behavior did not extend to reporting data to Analytics. To correct connect to Analytics, initialize using the following configuration APIs:

```cpp
APPD_API void appd_config_set_analytics_host(struct appd_config* cfg, const char* host);
```

- param host: The host for analytics agent - defaults to "localhost".

```cpp
APPD_API void appd_config_set_analytics_port(struct appd_config* cfg, const unsigned short port);
```

- param port: The port on which the analytics agent is listening - defaults to 9090.

Terminate the Agent

Just before the application exits, terminate the agent.
apd_sdk_term();
Enable SSL for C++ on Libagent

To enable SSL on the proxyless version of the C++ agent, do the following:

1. Open your application to the file where you configure the controller settings.
2. Set `const int CONTROLLER_USE_SSL = 1;`
3. In the `appd_config` struct, include the following method: `appd_config_set_controller_use_ssl(cfg, CONTROLLER_USE_SSL);

If you are using a self-signed certificate, you must additionally do the following:

1. Get the Controller certificate by running the following command:

   ```
   openssl s_client -connect <hostname>:<port> -showcerts < /dev/null | openssl x509 > cert.pem
   ```

2. Update the `certificate_file` Controller setting to point to absolute path to certificate.

   ```
   appd_config_set_controller_certificate_file(cfg, <Path to Cert File>);
   ```

Check the Certificate CommonName

The certificate CommonName must match the Controller hostname. If they do not match, you must reconfigure the certificate on the Controller.

You can check the CommonName by running the following command:

```
openssl s_client -connect <CONTROLLER_HOST>:<CONTROLLER_PORT> -showcerts
```

The CommonName appears above the BEGIN CERTIFICATE line.
C/C++ Agent SDK Reference

<table>
<thead>
<tr>
<th>On this page:</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Basic Types</td>
</tr>
<tr>
<td>- appd_config_init</td>
</tr>
<tr>
<td>- appd_config_set_app_name</td>
</tr>
<tr>
<td>- appd_config_set_tier_name</td>
</tr>
<tr>
<td>- appd_config_set_node_name</td>
</tr>
<tr>
<td>- appd_config_set_controller_host</td>
</tr>
<tr>
<td>- appd_config_set_controller_port</td>
</tr>
<tr>
<td>- appd_config_set_controller_account</td>
</tr>
<tr>
<td>- appd_config_set_controller_access_key</td>
</tr>
<tr>
<td>- appd_config_set_controller_use_ssl</td>
</tr>
<tr>
<td>- appd_config_set_controller_http_proxy_host</td>
</tr>
<tr>
<td>- appd_config_set_controller_http_proxy_port</td>
</tr>
<tr>
<td>- appd_config_set_controller_http_proxy_username</td>
</tr>
<tr>
<td>- appd_config_set_controller_http_proxy_password</td>
</tr>
<tr>
<td>- appd_config_set_controller_certificate_file</td>
</tr>
<tr>
<td>- appd_config_set_controller_certificate_dir</td>
</tr>
<tr>
<td>- appd_config_set_flush_metrics_on_shutdown</td>
</tr>
<tr>
<td>- appd_config_set_logging_min_level</td>
</tr>
<tr>
<td>- appd_config_set_logging_log_dir</td>
</tr>
<tr>
<td>- appd_config_set_logging_max_num_files</td>
</tr>
<tr>
<td>- appd_config_set_logging_max_file_size_bytes</td>
</tr>
<tr>
<td>- appd_config_set_init_timeout_ms</td>
</tr>
<tr>
<td>- appd_config_getenv</td>
</tr>
<tr>
<td>- appd_context_config_init</td>
</tr>
<tr>
<td>- appd_context_config_set_controller_host</td>
</tr>
<tr>
<td>- appd_context_config_set_controller_port</td>
</tr>
<tr>
<td>- appd_context_config_set_controller_account</td>
</tr>
<tr>
<td>- appd_context_config_set_controller_access_key</td>
</tr>
<tr>
<td>- appd_context_config_set_controller_use_ssl</td>
</tr>
<tr>
<td>- appd_context_config_set_controller_http_proxy_host</td>
</tr>
<tr>
<td>- appd_context_config_set_controller_http_proxy_port</td>
</tr>
<tr>
<td>- appd_context_config_set_controller_http_proxy_username</td>
</tr>
<tr>
<td>- appd_context_config_set_controller_http_proxy_password</td>
</tr>
<tr>
<td>- appd_context_config_set_controller_certificate_file</td>
</tr>
<tr>
<td>- appd_context_config_set_controller_certificate_dir</td>
</tr>
<tr>
<td>- appd_sdk_init</td>
</tr>
<tr>
<td>- appd_sdk_term</td>
</tr>
<tr>
<td>- appd_bt_begin</td>
</tr>
<tr>
<td>- appd_bt_begin_with_app_context</td>
</tr>
<tr>
<td>- appd_bt_end</td>
</tr>
<tr>
<td>- appd_bt_get</td>
</tr>
<tr>
<td>- appd_bt_is_snapshotting</td>
</tr>
<tr>
<td>- appd_bt_set_url</td>
</tr>
<tr>
<td>- appd_bt_override_start_time_ms</td>
</tr>
<tr>
<td>- appd_bt_override_time_ms</td>
</tr>
<tr>
<td>- appd_bt_store</td>
</tr>
<tr>
<td>- appd_bt_add_error</td>
</tr>
<tr>
<td>- appd_bt_add_user_data</td>
</tr>
<tr>
<td>- appd_backend_add</td>
</tr>
<tr>
<td>- appd_backend_declare</td>
</tr>
<tr>
<td>- appd_backend_prevent_agent_resolution</td>
</tr>
<tr>
<td>- appd_backend_set_identifying_property</td>
</tr>
<tr>
<td>- appd_custom_metric_add</td>
</tr>
<tr>
<td>- appd_custom_metric_report</td>
</tr>
<tr>
<td>- appd_exitcall_add_error</td>
</tr>
</tbody>
</table>
To instrument C and C++ applications, you use the C/C++ Agent API. The API includes functions for creating business transactions, transaction backends and exit points, and defining custom metrics.

This page describes the functions and structures defined in the API.

**Basic Types**

The C/C++ SDK defines the following opaque types:

- `appd_bt_handle`: A handle to an active business transaction
- `appd_exitcall_handle`: A handle to an active exit call
- `struct appd_config*`: A pointer to a configuration object
- `struct appd_context_config*`: A pointer to a context configuration object

**appd_config_init**

Initializes an empty configuration structure. Call this function before before calling `appd_sdk_init()`.

**Format**

```
APPD_API struct appd_config * appd_config_init ()
```

**Parameters**

- `cfg`: AppDynamics configuration object.

**appd_config_set_app_name**

Set business application name.

**Format**

```
appd_config_set_app_name(cfg, APP_NAME);
```

**Parameters**

- `cfg`: AppDynamics configuration object.
- `APP_NAME`: Name of the application.

**appd_config_set_tier_name**

Set tier name.

**Format**

```
appd_config_set_tier_name(cfg, APP_NAME);
```

**Parameters**

- `cfg`: AppDynamics configuration object.
- `APP_NAME`: Name of the application.
• cfg: AppDynamics configuration object.
• TIER_NAME: Name of the tier.

**appd_config_set_node_name**

Set node name.
Format

```c
appd_config_set_node_name(cfg, NODE_NAME);
```

Parameters

- `cfg` – AppDynamics configuration object.
- `NODE_NAME` – Name of the node.

**appd_config_set_controller_host**

Set hostname of the Controller.
Format

```c
appd_config_set_controller_host(cfg, CONTROLLER_HOST);
```

Parameters

- `cfg` – AppDynamics configuration object.
- `CONTROLLER_HOST` – Name of the Controller host.

**appd_config_set_controller_port**

Set the port number on which the Controller is listening.
Format

```c
appd_config_set_controller_port(cfg, CONTROLLER_PORT);
```

Parameters

- `cfg` – AppDynamics configuration object.
- `CONTROLLER_PORT` – Port number that the Controller is listening on.

**appd_config_set_controller_account**

Set the account name for connecting to the controller.
Format

```c
appd_config_set_controller_account(cfg, CONTROLLER_ACCOUNT);
```

Parameters

- `cfg` – AppDynamics configuration object.
- `CONTROLLER_ACCOUNT` – Account name that the Controller is connected to.

**appd_config_set_controller_access_key**

Set the access key for connecting to the controller.
Format

```c
appd_config_set_controller_access_key(cfg, CONTROLLER_ACCESS_KEY);
```

Parameters

- `cfg` – AppDynamics configuration object.
- `CONTROLLER_ACCESS_KEY` – Access key for connecting to the Controller.

**appd_config_set_controller_use_ssl**


Specify whether SSL should be used to communicate with the controller.

**Format**

```c
appd_config_set_controller_use_ssl(cfg, CONTROLLER_USE_SSL);
```

**Parameters**

- `cfg`: AppDynamics configuration object.
- `CONTROLLER_USE_SSL`: Whether to use SSL to connect with the Controller. Set to a non-zero integer for true. Set to the integer zero for false. Note that SaaS Controllers require this to be set to non-zero.

**appd_config_set_controller_http_proxy_host**

Optional. Set the hostname of the HTTP proxy if using an HTTP proxy to talk to Controller.

**Format**

```c
appd_config_set_controller_http_proxy_host(cfg, CONTROLLER_HTTP_PROXY_HOST);
```

**Parameters**

- `cfg`: AppDynamics configuration object.
- `CONTROLLER_HTTP_PROXY_HOST`: Host name of the HTTP proxy if using an HTTP proxy to talk to Controller.

**appd_config_set_controller_http_proxy_port**

Optional. Set the port number of the HTTP proxy. Default is 80.

**Format**

```c
appd_config_set_controller_http_proxy_port(cfg, CONTROLLER_HTTP_PROXY_PORT);
```

**Parameters**

- `cfg`: AppDynamics configuration object.
- `CONTROLLER_HTTP_PROXY_PORT`: Port name of the HTTP proxy. Default is 80.

**appd_config_set_controller_http_proxy_username**

Optional. Set the username to connect to the HTTP proxy with.

**Format**

```c
appd_config_set_controller_http_proxy_username(cfg, CONTROLLER_HTTP_PROXY_USERNAME);
```

**Parameters**

- `cfg`: AppDynamics configuration object.
- `CONTROLLER_HTTP_PROXY_USERNAME`: Username to connect to the HTTP proxy with.

**appd_config_set_controller_http_proxy_password**

Optional. Set the password to connect to the HTTP proxy with.

**Format**

```c
appd_config_set_controller_http_proxy_password(cfg, CONTROLLER_HTTP_PROXY_PASSWORD);
```

**Parameters**

- `cfg`: AppDynamics configuration object.
- `CONTROLLER_HTTP_PROXY_PASSWORD`: Password to connect to the HTTP proxy with.

**appd_config_set_controller_http_proxy_password_file**

Optional. Set the file containing password to connect to the HTTP proxy with.

**Format**

```c
appd_config_set_controller_http_proxy_password_file(cfg, CONTROLLER_HTTP_PROXY_PASSWORD_FILE);
```

**Parameters**

- `cfg`: AppDynamics configuration object.
• CONTROLLER_HTTP_PROXY_PASSWORD_FILE: File containing password to connect to the HTTP proxy with.

**apdp_config_set_controller_certificate_file**

Optional. Set the the CA certificate file name. Set this if you choose to use your own certificate file.

Format

```
apdp_config_set_controller_certificate_file(cfg, CONTROLLER_HTTP_CERTIFICATE_FILE);
```

Parameters

- `cfg`: AppDynamics configuration object.
- `CONTROLLER_HTTP_CERTIFICATE_FILE`: Name of the certificate file. Defaults to the included `ca-bundle.crt` file.

**apdp_config_set_controller_certificate_dir**

Optional. Set the full path to the CA certificate file. Set this if you have multiple certificate files.

Format

```
apdp_config_set_controller_certificate_dir(cfg, CONTROLLER_HTTP_CERTIFICATE_DIR);
```

Parameters

- `cfg`: AppDynamics configuration object.
- `CONTROLLER_HTTP_CERTIFICATE_DIR`: Full path to the certificate files.

**apdp_config_set_flush_metrics_on_shutdown**

Specify whether to collect metrics in the final minute before SDK shutdown—via `apdp_sdk_term`. By default, metrics reported in the minute before shutdown are lost. If you enable metric flushing, then you can retain the metrics reported in the final minute before shutdown.

Format

```
APPD_API void apdp_config_set_flush_metrics_on_shutdown(struct appd_config* cfg, int enable);
```

Parameters

- `cfg`: AppDynamics configuration object.
- `int`: Set to a non-zero integer to enable flushing metrics to the Controller. Default is 0.

Setting this flag means the `apdp_sdk_term()` blocks until the underpinning agent logic has completely uploaded the metric data. Not using this flag means that any metrics collected since the last Controller update will be discarded. Flushing the metrics can take anywhere between 60 and 120 seconds (on an average), and under certain circumstances longer, depending on the load on the Controller. Do not use this flag if you need the termination/tear-down logic to complete immediately.

**apdp_config_set_logging_min_level**

Set the minimum level of logging that is allowed. If `APPD_LOG_LEVEL_TRACE`, all log messages are allowed. If `APPD_LOG_LEVEL_FAT` or `APPD_LOG_LEVEL_FATAL`, only the most severe errors are logged. The default is `APPD_LOG_LEVEL_INFO`.

Format

```
apdp_config_set_logging_min_level(cfg, LOGGING_MIN_LEVEL);
```

Parameters

- `cfg`: AppDynamics configuration object.
- `LOGGING_MIN_LEVEL`: Minimum level of logging that's allowed.
- `APPD_LOG_LEVEL_TRACE`.
- `APPD_LOG_LEVEL_DEBUG`.
- `APPD_LOG_LEVEL_INFO`.
- `APPD_LOG_LEVEL_WARN`.
- `APPD_LOGLEVEL_ERROR`.
- `APPD_LOGLEVEL_FATAL`.
**appd_config_set_logging_log_dir**

Set the directory to log to. The process running the SDK must have permissions to create this directory, if it does not already exist, to list the files within it, and to write to the files within it.

Format

```c
appd_config_set_logging_log_dir(cfg, LOGGING_LOG_DIR);
```

Parameters

- `cfg`: AppDynamics configuration object.
- `LOGGING_LOG_DIR`: The directory where you want to store log files. Defaults to `/tmp/appd`.

**appd_config_set_logging_max_num_files**

Set the maximum number of log files allowed per tenant. Log files are rotated when they reach this number.

Format

```c
appd_config_set_logging_max_num_files(cfg, LOGGING_MAX_NUM_FILES);
```

Parameters

- `cfg`: AppDynamics configuration object.
- `LOGGING_MAX_NUM_FILES`: The maximum number of log files allowed per tenant. Default is 10.

**appd_config_set_logging_max_file_size_bytes**

Set the maximum size of an individual log file, in bytes. Log files are rotated when they reach this size.

Format

```c
appd_config_set_logging_max_file_size_bytes(cfg, LOGGING_MAX_FILE_SIZE_BYTES);
```

Parameters

- `cfg`: AppDynamics configuration object.
- `LOGGING_MAX_FILE_SIZE_BYTES`: The maximum size of an individual log file, in bytes. Default is 5 * 1024 * 1024.

**appd_config_set_init_timeout_ms**

Set the number of milliseconds you want `appd_sdk_init` (an asynchronous action) to wait until it has received controller configuration and is ready to capture business transactions. Set this if you want to capture short-running business transactions that occur at application startup and you don't mind the delay of waiting for the Controller to send the configuration.

Format

```c
appd_config_set_init_timeout_ms(cfg, INIT_TIMEOUT_MS);
```

Parameters

- `cfg`: AppDynamics configuration object.
- `INIT_TIMEOUT_MS`: The number of milliseconds you want `appd_sdk_init` (an asynchronous action) to wait until it has received controller configuration and is ready to capture business transactions.
  - `X`: Wait up to `X` milliseconds for Controller configuration.
  - `0`: Do not Wait for Controller configuration.
  - `-1`: Wait indefinitely until Controller configuration is received by agent
  - Default is 0.

**appd_config_getenv**

Set the environment variable for configuring the SDK. Environment variables are not read by default. You must call this function to configure the SDK via environment variables.

For a list of available environment variables, see Environment Variables.

Format

```c
APPD_API void appd_config_getenv(struct appd_config * cfg, const char * prefix);
```

Parameters
appd_context_config_init

Initializes and returns an empty context configuration structure.

Format

APPD_API struct appd_context_config* appd_context_config_init(const char* context_name)

Parameters

- context_name: Name for this context.

appd_context_config_set_controller_host

Add the Controller host to the application context.

Format

APPD_API void appd_context_config_set_controller_host (struct appd_context_config * context_cfg, const char * host)

Parameters

- context_cfg: An AppDynamics context configuration object indicating the business application, tier and node name for this context.
- host: Name of the Controller host

appd_context_config_set_controller_port

Add the Controller port to the application context.

Format

APPD_API void appd_context_config_set_controller_port (struct appd_context_config * context_cfg, const unsigned short port)

Parameters

- context_cfg: An AppDynamics context configuration object indicating the business application, tier and node name for this context.
- port: Port number that the Controller is listening on

appd_context_config_set_controller_account

Add the Controller port to the application context.

Format

APPD_API void appd_context_config_set_controller_port (struct appd_context_config * context_cfg, const char * acct)

Parameters

- context_cfg: An AppDynamics context configuration object indicating the business application, tier and node name for this context.
- account: Account name that the Controller is connected to.

appd_context_config_set_controller_access_key

Add the Controller access key to the application context.

Format

APPD_API void appd_context_config_set_controller_port (struct appd_context_config * context_cfg, const char * key)

Parameters

- context_cfg: An AppDynamics context configuration object indicating the business application, tier and node name for this context.
• **key:** Access key for connecting to the Controller.

**appd_context_config_set_controller_use_ssl**

Add the Controller access key to the application context.

Format

```c
APPD_API void appd_context_config_set_controller_port (struct appd_context_config * context_cfg, unsigned int * ssl)
```

Parameters

- `context_cfg`: An AppDynamics context configuration object indicating the business application, tier and node name for this context.
- `ssl`: Set to a non-zero integer for true. Set to the integer zero for false. Note that SaaS Controllers require this to be set to non-zero.

**appd_context_config_set_controller_http_proxy_host**

Optional. Set the host name of the HTTP proxy if using an HTTP proxy to talk to controller.

Format

```c
APPD_API void appd_context_config_set_controller_http_proxy_host(struct appd_context_config * context_cfg, const char * host)
```

Parameters

- `context_cfg`: An AppDynamics context configuration object indicating the business application, tier and node name for this context.
- `host`: Hostname of the HTTP proxy if using an HTTP proxy to talk to Controller.

**appd_context_config_set_controller_http_proxy_port**

Optional. Set the host name of the HTTP proxy if using an HTTP proxy to talk to Controller.

Format

```c
APPD_API void appd_context_config_set_controller_http_proxy_host(struct appd_context_config * context_cfg, const unsigned short * port)
```

Parameters

- `context_cfg`: An AppDynamics context configuration object indicating the business application, tier and node name for this context.
- `port`: Port name of the HTTP proxy. Default is 80.

**appd_context_config_set_controller_http_proxy_username**

Optional. Set the username to connect to the HTTP proxy with.

Format

```c
APPD_API void appd_context_config_set_controller_http_proxy_host(struct appd_context_config * context_cfg, const char * user)
```

Parameters

- `context_cfg`: An AppDynamics context configuration object indicating the business application, tier and node name for this context.
- `user`: Username to connect to the HTTP proxy with.

**appd_context_config_set_controller_http_proxy_password**

Optional. Set the password to connect to the HTTP proxy with.

Format

```c
APPD_API void appd_context_config_set_controller_http_proxy_host(struct appd_context_config * context_cfg, const char * pwd)
```

Parameters
• context_cfg: An AppDynamics context configuration object indicating the business application, tier and node name for this context.
• pwd: Password to connect to the HTTP proxy with.

**appd_context_config_set_controller_http_proxy_password_file**

Optional. Set the password to connect to the HTTP proxy with.
Format

```
APPD_API void appd_context_config_set_controller_http_proxy_password_file(struct appd_context_config * context_cfg, const char * file);
```
Parameters

- context_cfg: An AppDynamics context configuration object indicating the business application, tier and node name for this context.
- file: File containing password to connect to the HTTP proxy with.

**appd_context_config_set_controller_certificate_file**

Optional. Set the CA certificate file name. Set this if you choose to use your own certificate file.
Format

```
APPD_API void appd_context_config_set_controller_certificate_file(struct appd_context_config * context_cfg, const char * file);
```
Parameters

- context_cfg: An AppDynamics context configuration object indicating the business application, tier and node name for this context.
- file: Name of the certificate file. Defaults to the included ca-bundle.crt file.

**appd_context_config_set_controller_certificate_dir**

Optional. Set the full path to the CA certificate file. Set this if you have multiple certificate files.
Format

```
APPD_API void appd_context_config_set_controller_certificate_dir(struct appd_context_config * context_cfg, const char * dir);
```
Parameters

- context_cfg: An AppDynamics context configuration object indicating the business application, tier and node name for this context.
- dir: Full path to the certificate files.

**appd_sdk_init**

Initialize the AppDynamics C/C++ agent. An instrumented application must call this function once, preferably during application startup.

```
When using fork(), the appd_sdk_init(), and all other SDK calls, should occur only in the forked process, never in the parent, and each child should behave as its own process, and never share handles between the process itself and other SDK instrumented child processes.
```

Format

```
APPD_API int appd_sdk_init (const struct appd_config * config)
```
Parameters

- config: AppDynamics configuration settings that enable communication between the agent and the Controller.

Returns

Zero on success, otherwise a non-zero value. If the return value is not zero, a log message describes the error.
appd_sdk_term

Stop the AppDynamics C/C++ agent. Ends all active business transactions for this agent and stops agent metric reporting to the controller.

```
If you execute the agent for less than a few minutes, the agent may not have enough time to fully report to the Controller, as this action is done once every 60 seconds. To ensure that any remaining data is transmitted to the Controller, wait at least one minute before issuing the appd_sdk_term() call.
```

Format

```
APPD_API void appd_sdk_term()
```

appd_bt_begin

Start a business transaction or continue an existing transaction.

Keep in mind that each application is limited to 200 registered business transactions, and each agent is limited to 50 registered business transactions. Like transactions discovered by other types of agents, business transactions that exceed the limit that are reported by the C/C++ SDK are included in the “all other traffic” grouping. This means that it’s up to you to ensure that your agent does not create excessive business transactions. Use care to ensure that your implementation does not introduce the possibility of business transaction explosion.

Format

```
APPD_API appd_bt_handle appd_bt_begin (const char * name, const char * correlation_header)
```

Parameters

- `name`: The name for the business transaction. In the case of a continuing transaction in the current business application with a valid correlation header, the SDK uses the name from the header. Do not use the following characters in transaction names: `{ }[]|&;`
- `correlation_header`: A correlation header if this is a continuing transaction, else NULL. If specified, the correlation header has been generated by another AppDynamics agent and made available to this agent as a string. The correlation header provides information to enable this transaction to correlate with an upstream transaction.

Returns

An opaque handle for the business transaction that was started.

appd_bt_begin_with_app_context

Start a business transaction or continue an existing transaction in a multi-tenant Controller environment.

Format

```
APPD_API appd_bt_handle appd_bt_begin_with_app_context (const char * context, const char * name, const char * correlation_header)
```

Parameters

- `context`: The application context name that this business transaction belongs to.
- `name`: The name for the business transaction. In the case of a continuing transaction in the current business application with a valid correlation header, the SDK uses the name from the header. Do not use the following characters in transaction names: `{ }[]|&;`
- `correlation_header`: A correlation header if this is a continuing transaction, else NULL. If specified, the correlation header has been generated by another AppDynamics agent and made available to this agent as a string. The correlation header provides information to enable this transaction to correlate with an upstream transaction.

appd_bt_end

End the given business transaction.

Format

```
APPD_API void appd_bt_end (appd_bt_handle bt)
```

Parameters

- `bt`: The handle to the business transaction to end.
appd_bt_get

Get a BT handle associated with the given guid by appd_bt_store.

Format

`APPD_API appd_bt_handle appd_bt_get (const char * guid)`

Parameters

- `guid`: The globally unique identifier that was passed to appd_bt_store.

Returns

The handle to the business transaction associated with the given guid.

In the following cases the SDK logs a warning and returns a handle that you may safely use in other API functions but that will cause these functions to immediately return without doing anything:

- The SDK does not find a handle associated with the guid
- The call ended before the SDK could retrieve the handle

appd_bt_is_snapshotting

Reports whether the agent is currently taking a transaction snapshot. Useful before calling appd_add_user_data() or appd_bt_set_url(), since those potentially expensive functions would do nothing if called when a snapshot is not being taken.

Format

`APPD_API char appd_bt_is_snapshotting (appd_bt_handle bt)`

Parameters

- `bt`: The handle to the business transaction to check for snapshotting.

Returns

Non-zero if the given business transaction is taking a snapshot. Otherwise, zero.

appd_bt_set_url

Set URL for a snapshot (if one is being taken).

URL is set for a snapshot if one is occurring. Data should be either 7-bit ASCII or UTF-8.

It is safe to call this function when a snapshot is not occurring. When the given business transaction is NOT snapshotting, this function immediately returns. However, if extracting the data to pass to this function is expensive, you can use appd_bt_is_snapshotting to check if the business transaction is snapshotting before extracting the data and calling this function.

Format

`APPD_API void appd_bt_set_url (appd_bt_handle bt, const char * url)`

Parameters

- `bt`: The business transaction to add the user data to, if it's taking a snapshot.
- `url`: The value of the URL for the snapshot as 7-bit ASCII or UTF-8.

appd_bt_override_start_time_ms

Takes a time, in milliseconds, representing the number of milliseconds elapsed since Jan 1, 1970 UTC. The time specified for this function overrides the default start time, which is the time at which the appd_bt_begin() API is called according to the system clock.

By overriding the start time with a specific value, the BT internal start time is disabled, and the specified start time is reported to the Controller.

This function is located in the appdynamics_advanced.h file, and is only used in special cases.

Format

`APPD_API void appd_bt_override_start_time_ms(appd_bt_handle bt, unsigned int timeMS);`

Parameters

- `bt`: The business transaction to override the timing of
- `timeMS`: Time in milliseconds since start of epoch (midnight, Jan 1, 1970 UTC).
appd_bt_override_time_ms

Takes a time, in milliseconds, that overrides the business transaction response time as reported for this business transaction to the Controller.

The C/C++ Agent SDK maintains its own, internal timer for the response time for the business transaction. This timer reflects the elapsed time spent between the `bt_begin` and `bt_end` calls. In some cases, you may want to set the business transaction timer directly, overriding the default business transaction response timer. For instance, this may be useful for integrating external monitoring systems with AppDynamics.

It is important to note that when calling this function, the reported business transaction will be either the time you specify or the sum of all exit call timings for the transaction, whichever is greatest. This is because a business transaction cannot be reported to take less time than the total of the exit calls it contains.

This function is located in the appdynamics_advanced.h file, and is only used in special cases.

Format

```c
APPD_API void appd_bt_override_time_ms(appd_bt_handle bt, unsigned int timeMS);
```

Parameters

- `bt`: The business transaction for which to set the response time.
- `timeMS`: The time the business transaction took, in milliseconds.

appd_bt_store

Store a BT handle for retrieval with appd_bt_get.

This function allows you to store a BT in a global registry to retrieve later. This is convenient when you need to start and end a BT in separate places, and it is difficult to pass the handle to the BT through the parts of the code that need it.

When the BT is ended, the handle is removed from the global registry.

Format

```c
APPD_API void appd_bt_store (appd_bt_handle bt, const char * guid)
```

Parameters

- `bt`: The BT to store.
- `guid`: A globally unique identifier to associate with the given BT.

Example

```c
int begin_transaction(uint64_t txid, uint64_t sku, float price)
{
    appd_bt_handle bt = appd_bt_begin("payment-processing", NULL);
    appd_bt_store(bt, std::to_string(txid).c_str());
    // ...
}
```

appd_bt_add_error

Add an error to a business transaction.

Errors are reported as part of the business transaction. However, you can add an error without marking the business transaction as an error (e.g., for non-fatal errors).

Format

```c
APPD_API void appd_bt_add_error (appd_bt_handle bt, enum appd_error_level level, const char * message, int mark_bt_as_error)
```

Parameters

- `bt`: The handle to the business transaction to which the error is added.
- `level`: The error level, from the following:

```c
enum appd_error_level
```

```c
APPD_ERROR_LEVEL_ERROR
APPD_ERROR_LEVEL_WARN
APPD_ERROR_LEVEL_INFO
APPD_ERROR_LEVEL_NONE
```
- **APPD_LEVEL_NOTICE**
- **APPD_LEVEL_WARNING**
- **APPD_LEVEL_ERROR**
- **message:** Error message for this error.
- **mark_bt_as_error:** If true, the business transaction experiencing this error is marked as an error transaction. In this case, the business transaction is counted only as an error transaction. It is not also counted as a slow, very slow or stalled transaction, even if the transaction was also slow or stalled. If false, the business transaction is not marked as an error transaction.

**appd_bt_add_user_data**

Attaches user data to transaction snapshots generated for the specified business transaction.

The user data is added to the business transaction but reported only when a transaction snapshot is occurring. In the Controller UI, the user data appears in the Business Data tab of the transaction snapshot details.

It is safe to call this function when a snapshot is not occurring. When the specified business transaction is not taking a snapshot, this function immediately returns.

If extracting the data to pass to this function is expensive, you can use `appd_bt_is_snapshotting()` to check if the business transaction is actually taking a snapshot before extracting the data and calling this function.

Data should be either 7-bit ASCII or UTF-8.

**Format**

```c
APPD_API void appd_bt_add_user_data (appd_bt_handle bt, const char * key, const char * value)
```

**Parameters**

- **bt:** The business transaction to add the user data to, if it's taking a snapshot.
- **key:** The name of the user data to add to the snapshot as 7-bit ASCII or UTF-8.
- **value:** The value of the user data to add to the snapshot as 7-bit ASCII or UTF-8.

**appd_backend_add**

Add a declared backend to the current business application. Use `appd_backend_declare` to declare a backend before adding it.

This function fails if the type of the backend being added does not have at least one identifying property.

**Format**

```c
APPD_API int appd_backend_add (const char * backend)
```

**Parameters**

- **backend:** Pointer to the backend.

**Returns**

Zero on success, otherwise a non-zero value. If the return value is not zero, a log message describes the error.

**appd_backend_declare**

Declare the existence of a backend. Call this only if the backend is not already registered with the C++ SDK instance.

**Format**

```c
APPD_API void appd_backend_declare (const char * type, const char * unregistered_name)
```

**Parameters**

- **type:** The type of the backend, from these options:
  - **APPD_BACKEND_HTTP**
  - **APPD_BACKEND_DB**
  - **APPD_BACKEND_CACHE**
  - **APPD_BACKEND_RABBITMQ**
  - **APPD_BACKEND_WEBSERVICE**
  - **APPD_BACKEND_JMS**
- **unregistered_name:** The name of the backend. Must be unique to the C++ SDK instance.
**appd_backend_prevent_agent_resolution**

Call to prevent a downstream agent from resolving as this backend. If used, this must be called before `appd_backend_add()`.

Normally, if an agent picks up a correlation header for an unresolved backend, it will resolve itself as that backend. This is usually the desired behavior.

However, if the backend is actually an uninstrumented tier that is passing through the correlation header (for example, a message queue or proxy), then you may wish the backend to show up distinct from the tier that it routes to. If you call this function, correlation headers generated for exit calls to this backend in the SDK will instruct downstream agents to report as distinct from the backend.

For example: if you have Tier A talking to uninstrumented Backend B which routes to instrumented Tier C, if you DO NOT call this function, the flow map will be A > C. If you DO call this function, the flow map will be A > B > C.

**Format**

```c
APPD_API int appd_backend_prevent_agent_resolution (const char * backend)
```

**Parameters**

- `backend`: Pointer to the declared backend.

**Returns**

Zero on success, otherwise a non-zero value. If the return value is not zero, a log message describes the error.

---

**appd_backend_set_identifying_property**

Sets an identifying property of a backend. Call once for every identifying property that you want to set. Call this function after `appd_backendDeclare()` and before `appd_backend_add()`.

A backend's identifying properties uniquely identify the downstream component being called. In the controller UI, these properties are visible in the upper right panel of the backend dashboards.

You must set at least one identifying property when you add a backend of one of the built-in exit call types. The valid properties vary per exit call type as indicated below.

<table>
<thead>
<tr>
<th>Exit Call Type</th>
<th>Valid Identifying Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPD_BACKEND_HTTP</td>
<td>&quot;HOST&quot;, &quot;PORT&quot;, &quot;URL&quot;, &quot;QUERY STRING&quot;</td>
</tr>
<tr>
<td>APPD_BACKEND_DB</td>
<td>&quot;HOST&quot;, &quot;PORT&quot;, &quot;DATABASE&quot;, &quot;VENDOR&quot;, &quot;VERSION&quot;</td>
</tr>
<tr>
<td>APPD_BACKEND_CACHE</td>
<td>&quot;SERVER POOL&quot;, &quot;VENDOR&quot;</td>
</tr>
<tr>
<td>APPD_BACKEND_RABBITMQ</td>
<td>&quot;HOST&quot;, &quot;PORT&quot;, &quot;ROUTING KEY&quot;, &quot;EXCHANGE&quot;</td>
</tr>
<tr>
<td>APPD_BACKEND_WEBSERVICE</td>
<td>&quot;SERVICE&quot;, &quot;URL&quot;, &quot;OPERATION&quot;, &quot;SOAP ACTION&quot;, &quot;VENDOR&quot;</td>
</tr>
<tr>
<td>APPD_BACKEND_JMS</td>
<td>&quot;DESTINATION&quot;, &quot;DESTINATIONTYPE&quot;, &quot;MAJOR VERSION&quot;, &quot;VENDOR&quot;</td>
</tr>
<tr>
<td>APPD_BACKEND_WEBSPHEREMQ</td>
<td>&quot;HOST&quot;, &quot;PORT&quot;, &quot;DESTINATION&quot;, &quot;DESTINATIONTYPE&quot;, &quot;MAJOR VERSION&quot;, &quot;VENDOR&quot;</td>
</tr>
</tbody>
</table>

**Format**

```c
APPD_API int appd_backend_set_identifying_property (const char * backend, const char * key, const char * value)
```

**Parameters**

- `backend`: Pointer to the backend.
- `key`: Name of the property. For example, DESTINATION or PORT.
- `value`: Value of the property. For example, Order Queue or 3304

**Returns**

Zero on success, otherwise a non-zero value. If the return value is not zero, a log message describes the error.

---

**appd_custom_metric_add**
Define a custom metric.

Format

```c
APPD_API void appd_custom_metric_add (const char * application_context, const char * metric_path,
       enum appd_time_rollup_type time_rollup_type, enum appd_cluster_rollup_type cluster_rollup_type,
       enum appd_hole_handling_type hole_handling_type)
```

Parameters

- `application_context`: The application context for this custom metric.
- `metric_path`: The path of the custom metric. This path defines where the custom metric appears in the Metric Browser. Use a pipe character to separate branches in the path. For example, "Custom "Metrics|MyCustomMetric", defines MyCustomMetric as a top-level metric in the tree. For more about defining the custom metric path, see the 'Metric Path' section in [Build a Monitoring Extension Using Java](#).
- `time_rollup_type`: Specifies how to rollup metric values for this metric over time, from the following:
  - `APPD_TIMEROLLUP_TYPE_AVERAGE`: Compute the average value of the metric over time.
  - `APPD_TIMEROLLUP_TYPE_SUM`: Compute the sum of the value of the metric over time.
  - `APPD_TIMEROLLUP_TYPE_CURRENT`: Report the current value of the metric.
- `cluster_rollup_type`: Specifies how to rollup metric values for this metric across clusters, from the following:
  - `APPD_CLUSTERROLLUP_TYPE_INDIVIDUAL`: Roll-up the value individually for each member of the cluster.
  - `APPD_CLUSTERROLLUP_TYPE_COLLECTIVE`: Roll-up the value across all members of the cluster.
- `hole_handling_type`: Specifies how to handle holes—gaps where no value has been reported from this metric. Use one of these options:
  - `APPD_HOLEHANDLING_TYPE_RATE_COUNTER`: Considers the gap or 0 value to be meaningful in the aggregation of the metric.
  - `APPD_HOLEHANDLING_TYPE_REGULAR_COUNTER`: Does not consider the gap or 0 value to be meaningful in the aggregation of the metric.

For additional information on these parameters, see the topic on creating custom metrics in [Using the C/C++ Agent SDK](#).

**appd_custom_metric_report**

Report a value for a given metric.

Format

```c
APPD_API void appd_custom_metric_report (const char * application_context, const char * metric_path,
                                           long value)
```

Parameters

- `application_context`: The application context for this custom metric
- `metric_path`: The path of the metric to report, as defined by `appd_custom_metric_add`
- `value`: The value to report for the metric. The way the value is aggregated is specified by the roll-up parameters to `appd_custom_metric_add`

**appd_exitcall_add_error**

Add an error to the exit call.

Format

```c
APPD_API void appd_exitcall_add_error (appd_exitcall_handle exitcall, enum appd_error_level level,
                                         const char * message, int mark_bt_as_error)
```

Parameters

- `exitcall`: Handle to the exit call.
- `level`: The level of this error, from the following:
  - `APPD_LEVEL_NOTICE`
  - `APPD_LEVEL_WARNING`
  - `APPD_LEVEL_ERROR`
- `message`: Error message for this error.
- `mark_bt_as_error`: If true, the business transaction making the exit call that is experiencing this error is marked as an error transaction. In this case, the business transaction is counted only as an error transaction. It is not also counted as a slow, very slow or stalled transaction, even if the transaction was also slow or stalled. If false, the business transaction is not marked as an error transaction.
appd_exitcall_begin

Start an exit call to the specified backend as part of a business transaction.

Format

APPD_API appd_exitcall_handle appd_exitcall_begin (appd_bt_handle bt, const char * backend)

Parameters

- bt: Handle to the business transaction making the exit call.
- backend: The destination backend of the exit call. AppDynamics does not automatically detect backends for Go applications, so you must specify the destination backend.

Returns

An opaque handle to the exit call that was started.

appd_exitcall_end

Complete the exit call.

Format

APPD_API void appd_exitcall_end (appd_exitcall_handle exitcall)

Parameters

- exitcall: Handle to the exit call being ended.

appd_exitcall_get

Get a handle to an exit call associated with a guid via appd_exitcall_store().

Format

APPD_API appd_exitcall_handle appd_exitcall_get (const char * guid)

Parameters

- guid: The globally unique identifier that was passed to appd_exitcall_store().

Returns

The handle to the exit call associated with the given guid.

In the following cases the SDK logs a warning and returns a handle that you may safely use in other API functions but that will cause these functions to immediately return without doing anything:

- The SDK doesn't find a handle associated with the guid
- The call ended before the SDK could retrieve the handle

appd_exitcall_get_correlation_header

Get the header for correlating a business transaction.

If a business transaction makes exit calls that you wish to correlate across, you should retrieve the correlation header and inject it into your exit call's payload.

The returned string is freed when the exit call ends. Do not free it yourself.

Format

APPD_API const char* appd_exitcall_get_correlation_header (appd_exitcall_handle exitcall)

Parameters

- exitcall: Handle to the exit call.

Returns

On success returns a 7-bit ASCII string containing the correlation information. You can inject this string into the exit call's payload. A downstream agent can then extract the header from that payload and continue the business transaction.
On error, returns the default header that prevents downstream business transaction detection.

**appd_exitcall_override_start_time_ms**

Takes a time, in milliseconds, representing the number of milliseconds elapsed since Jan 1, 1970 UTC. The time specified for this function overrides the default start time, which is the time at which the appd_exitcall_begin() API is called according to the system clock. By overriding the start time with a specific value, the exit call's internal start time is disabled, and the specified start time is reported to the controller.

This function is located in the `appdynamics_advanced.h` file, and is only used in special cases.

**Format**

```c
APPD_API void appd_exitcall_override_start_time_ms(appd_exitcall_handle exitCall, int64_t timeMS);
```

**Parameters**

- `exitCall`: The exit call for which to override the start time
- `timeMS`: The time that you want to override the exit call start time

**appd_exitcall_override_time_ms**

Takes a time, in milliseconds, that overrides the exit call time reported to the Controller.

The C/C++ Agent SDK maintains its own internal timer for the time it takes to complete the exit call. In some cases, you may want to disable the exit call's internal timer. For instance, this can be useful for reporting exit calls that are recorded in external monitoring systems and read into an SDK program.

This function is located in the `appdynamics_advanced.h` file, and is only used in special cases.

**Format**

```c
APPD_API void appd_exitcall_override_time_ms(appd_exitcall_handle exitCall, int64_t timeMS);
```

**Parameters**

- `exitCall`: The exit call for which to set the completion time
- `timeMS`: The time that you want to override the exit call time.

**appd_exitcall_set_details**

Set the details string for an exit call. This can be used, for example, to add the SQL statement that a DB backend has executed as part of the exit call. This data is then visible in the exit calls details UI for the transaction snapshot.

**Format**

```c
APPD_API int appd_exitcall_set_details (appd_exitcall_handle exitcall, const char * details)
```

**Parameters**

- `exitcall`: Handle to the exit call.
- `details`: An arbitrary string to add to the exit call.

**Returns**

Zero on success, otherwise a non-zero value. If the return value is not zero, a log message describes the error.

**appd_exitcall_store**

Store an exit call handle for retrieval with `appd_exitcall_get()`.

This function allows you to store an exit call in a global registry to retrieve later. This is convenient when you need to start and end the call in separate places, and it is difficult to pass the handle through the parts of the code that need it.

The handle is removed when the exit call (or the BT containing it) ends.

**Format**

```c
APPD_API void appd_exitcall_store (appd_exitcall_handle exitcall, const char * guid)
```

**Parameters**

- `exitcall`: The exit call to store.
• guid: A globally unique identifier to associate with the given call.

Example

```c
appd_exitcall_handle ec = appd_exitcall_begin(bt, "authdb");
appd_exitcall_store(ec, "login-exit");
```

**appd_frame_begin** *(New in 4.5.1)*

Records the start of a method call in the call stack. To track the duration of your method call, you must call `appd_frame_begin` near the start of the method code, and `appd_frame_end` when the method returns.

Format

```c
APPD_API appd_frame_handle appd_frame_begin(appd_bt_handle bt, enum appd_frame_type frame_type,
const char* class_name, const char* method_name, const char* file, unsigned int line_number);
```

Parameters

- `bt`: The business transaction for the call graph.
- `frame_type`: The type of the frame. When used in C or C++ code, use `APPD_FRAME_TYPE_CPP`.
- `class_name`: The name of the class if this method is a member of the class. If not, then set to NULL.
- `method_name`: The name of the method.
- `file`: The path of the source file.
- `line_number`: The line number in the source file.

Returns

An opaque handle for the frame. NULL if an error occurred.

**appd_frame_end** *(New in 4.5.1)*

Records the end of a method call in the call stack. To track the duration of your method call, you must call `appd_frame_begin` near the start of the method code, and `appd_frame_end` when the method returns.

Format

```c
APPD_API void appd_frame_end(appd_bt_handle bt, appd_frame_handle frame);
```

Parameters

- `bt`: The business transaction for the call graph.
- `frame`: The handle of returned by the corresponding `appd_frame_begin` call.

**appd_sdk_add_app_context**

Initialize the AppDynamics SDK.

Format

```c
APPD_API int appd_sdk_add_app_context (struct appd_context_config * context_cfg)
```

Parameters

- `context_cfg`: An AppDynamics context configuration object indicating the business application, tier and node name for this context.

**appd::sdk::BT**

An RAII (Resource Acquisition Is Initialization) pattern constructor for C++ applications. For a new business transaction, this constructor creates the business transaction object and a handle to that object. Continues existing transactions. Adds the correlation header if one is passed.

When the BT object goes out of scope, the SDK automatically ends the business transaction. A BT object cannot be copied.

See 'APIs for Resource Management (C++)' in *Using the C/C++ Agent SDK* for more information.

Format
appd::sdk::BT(const std::string& name)
appd::sdk::BT(const std::string& name, const std::string& correlation_header)
appd::sdk::BT(const char* name, const char* correlation_header=NULL)

Parameters

- **name**: Business transaction name, passed as either a `std::string` or `char *`, depending on which constructor you choose. In the case of a continuing transaction in the current business application with a valid correlation header, the SDK uses the name from the header.
- **correlation_header**: Optional correlation header, passed as either a `std::string` or `char *`, depending on which constructor you choose.

**appd::sdk::ExitCall**

An RAI (Resource Acquisition Is Initialization) pattern constructors for C++ applications. Constructs an ExitCall object and creates a handle to that object. You must declare a backend and assign it at least one identifying property before you use the backend to instantiate the ExitCall object.

When the ExitCall object goes out of scope, the SDK automatically ends the ExitCall.

An ExitCall object cannot be copied.

See 'APIs for Resource Management (C++)' in Using the C/C++ Agent SDK for more information.

**Format**

appd::sdk::ExitCall(BT& bt, const char* backend)
appd::sdk::ExitCall(BT& bt, const std::string& backend)

Parameters

- **bt**: Handle to the business transaction making the exit call.
- **backend**: The destination backend of the exit call. Passed as either a `std::string` or `char *`, depending on which constructor you choose.

**appd::sdk::Frame (New in 4.5.1)**

Use the Frame class to enable call graphs for your C++ application. You can use the Frame class to instrument the methods that you want to track, as an alternative to calling `appd_frame_begin` and `appd_frame_end`. You must pass the following parameters into the Frame class:

**Format**

Frame(BT& bt, appd_frame_type frame_type, const char* class_name, const char* method_name, const char* file, unsigned int line_number)

Parameters

- **bt**: The BT object that owns this function call
- **frame_type**: The type of the frame. When used in C or C++ code, use APPD_FRAME_TYPE_CPP.
- **class_name**: The name of the class if this method is a member of the class, else NULL.
- **method_name**: The name of the method
- **file**: The path of the source file.
- **line_number**: The line number in the source file.

**Environment Variables**

You can configure the following environment variables for the C++ agent.

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>APP_NAME</td>
<td>Name of the application</td>
</tr>
<tr>
<td>CONTROLLER_ACCESS_KEY</td>
<td>Access key for connecting to the Controller</td>
</tr>
<tr>
<td>CONTROLLER_ACCOUNT</td>
<td>Account name for connecting to the Controller</td>
</tr>
<tr>
<td>CONTROLLER_HOST</td>
<td>Hostname of the Controller</td>
</tr>
<tr>
<td>Variable Name</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>CONTROLLER_PORT</td>
<td>Port number on which the controller is listening.</td>
</tr>
<tr>
<td>CONTROLLER_HTTP_PROXY_HOST</td>
<td>Hostname of the HTTP proxy if using an HTTP proxy to talk to Controller</td>
</tr>
<tr>
<td>CONTROLLER_HTTP_PROXY_PASSWORD</td>
<td>The password for connecting to the HTTP proxy</td>
</tr>
<tr>
<td>CONTROLLER_HTTP_PROXY_PASSWORD_FILE</td>
<td>File containing password to connect to the HTTP proxy with</td>
</tr>
<tr>
<td>CONTROLLER_HTTP_PROXY_PORT</td>
<td>Port number of the HTTP proxy</td>
</tr>
<tr>
<td>CONTROLLER_HTTP_PROXY_USERNAME</td>
<td>Username to connect to the HTTP proxy with</td>
</tr>
<tr>
<td>CONTROLLER_USE_SSL (optional)</td>
<td>To disable SSL, set to off, 0, f, or false. To enable SSL, set to on, 1, t, or true.</td>
</tr>
<tr>
<td>CONTROLLER_CERTIFICATE_FILE</td>
<td>The SSL certificate file name for connecting to the Controller</td>
</tr>
<tr>
<td>CONTROLLER_CERTIFICATE_DIR</td>
<td>The directory where the SSL certificate is located</td>
</tr>
<tr>
<td>FLUSH METRICS_ON_SHUTDOWN</td>
<td>To enable metric flushing before shutdown, set to a non-zero integer. To disable metric flushing before shutdown, set to 0 (default).</td>
</tr>
<tr>
<td>INIT_TIMEOUT_MS</td>
<td>Number of milliseconds that you want appd_sdk_init, an asynchronous action, to wait until it has received Controller configuration and is ready to capture business transactions.</td>
</tr>
<tr>
<td>LOGGING_LOG_DIR</td>
<td>The directory that the logs are saved to</td>
</tr>
<tr>
<td>LOGGING_LEVEL</td>
<td>The level of severity for which activity is logged. Options: all, trace, debug, info, warn, error</td>
</tr>
<tr>
<td>LOGGING_MAX_NUM_FILES</td>
<td>The maximum number of log files that can be saved</td>
</tr>
<tr>
<td>LOGGING_MAX_FILE_SIZE_BYTES</td>
<td>The maximum log file size that can be saved</td>
</tr>
<tr>
<td>NODE_NAME</td>
<td>Name of the node</td>
</tr>
<tr>
<td>TIER_NAME</td>
<td>Name of the tier</td>
</tr>
<tr>
<td>UNIQUE_HOST_ID (optional)</td>
<td>The identifier of the physical host or virtual machine that the agent is installed on.</td>
</tr>
</tbody>
</table>
Go Language Agent

On this page:
- About Go Support
- Go SDK Supported Platforms
- Install the AppDynamics Go SDK
- Limitations

This topic describes how to install and start using the app server agent for Go applications.

About Go Support

AppDynamics can monitor applications that have been created with Go SDK.

The AppDynamics Go SDK can:
- Trace transactions through Go tiers
- Surface Go runtime errors
- Report backend calls made by Go services

Once the agent connects to the Controller, you can see flow maps, key performance indicators, errors and other information for the Go tier.

To instrument, use the AppDynamics Go SDK, as described in the following topics.

Go SDK Supported Platforms

Operating Systems
• Any Linux distribution based on glibc 2.5+
• Mac OS X 10.8+

Install the AppDynamics Go SDK

To install the SDK follow these steps:

1. Download the Go SDK distribution. See Download AppDynamics Software.
2. Extract the Go SDK ZIP into the Go workspace.

When finished installing the Go SDK, you are ready to instrument your Go application using the API.

Limitations

The Go SDK uses cGo in the background. In Go, when you are calling out to C/C++, the code converts the current goroutine to an OS thread. This is a limitation of the runtime itself, not of the Go SDK agent that we provide. However, it means that if you are running the agent with many Go routines and make Go SDK calls within the Go routines in your application code, there may be a risk of reaching the OS thread limit. Contact support if you encounter this limitation.
Go Language Supported Environments

Go SDK Support

*Operating Systems*

- Any Linux distribution based on glibc 2.5+.
Using Go SDK

On this page:

- About Go Application Instrumentation
- Import the AppDynamics Package to the Application
- Initialize the Controller Configuration
- Initialize the Agent
- Create Business Transactions
- Add Business Transaction Errors
- Manage Business Transaction Snapshots
- Create Backends
- Manage Exit Calls
- Correlate with Other Business Transactions
- Terminate the Agent

Before you can start instrumenting your applications with the Go SDK, you need to get the current version of the Go SDK. The Go Language Agent and Download AppDynamics Software topic describes how to get and install the SDK.

Once you have the SDK, you can instrument your application. The SDK provides routines for creating and managing Business Transactions, transaction snapshots, backends, and exit calls. This topic provides an overview of these concepts.

About Go Application Instrumentation

The following topics provide a general outline of how to implement instrumentation in your Go application:

- Import the AppDynamics API File to the Application
- Initialize the Controller Configuration
- Initialize the Agent
- Create Business Transactions
- Manage Business Transaction Snapshots
- Create Backends
- Manage Exit Calls
- Correlate with Other Business Transactions
- Terminate the Agent

Import the AppDynamics Package to the Application

After downloading the SDK, you are ready to add AppDynamics instrumentation to your Go application. The first step is to import the AppDynamics package:

```go
import appd "appdynamics"
```

Initialize the Controller Configuration

Controller information settings permit the SDK to connect to the Controller. Some settings are required for all applications, while others are required only for certain types of application environments. For example, if the SDK needs to connect to the Controller via an internal proxy in your network, set up the connection settings for the proxy.

See ‘Agent Configuration Settings’ in Go SDK Reference for a complete list of the settings and information about which ones are required.

In your application, assign values to the required settings. For example, to set the Controller connection information:
cfg := appd.Config{
   AppName = "exampleapp",
   TierName = "orderproc",
   NodeName = "orderproc01",
   Controller: Controller{
        Host = "my-appd-controller.example.org",
        Port = 8080,
        UseSSL = true,
        Account = "customer1",
        AccessKey = "secret",
        InitTimeoutMs = 1000  // Wait up to 1s for initialization to finish
    }
}

Notice the InitTimeoutMs field. Once you initialize the configuration, you pass the configuration object to the call to initialize the agent via InitSDK(). The InitTimeoutMs field can have these values:

- When set to 0, the default, the InitSDK() function operates as an asynchronous action, so that the initialization call does not block your program.
- Setting the value to -1 instructs the program to wait indefinitely until the controller configuration is received by the agent, that is, the InitSDK() method returns control. This is useful when you want to capture short-running Business Transactions that occur at application startup and you do not mind the delay of waiting for the Controller to send the configuration.
- Alternatively, set it to a specific number of milliseconds to wait.

If you use a multi-tenant Controller (SaaS or on-premises multi-tenant), you need to create the context for the multi-tenant environments using the AddAppContextToConfig() method. You can then pass the context as a parameter to methods for performing particular operations, such as starting a BT or adding custom metrics. See Go SDK Reference for more about AddAppContextToConfig() and related methods.

Initialize the Agent

In your main function, initialize the agent by passing the configuration structure to InitSDK().

If InitSDK() returns nil, the agent is initialized successfully. If an error returns, it is likely because the agent could not reach the Controller.

The following example illustrates how to initialize the SDK:

```go
if err := appd.InitSDK(&cfg); err != nil {
    fmt.Printf("Error initializing the AppDynamics SDK\n")
} else {
    fmt.Printf("Initialized AppDynamics SDK successfully\n")
}
```

Create Business Transactions

Define a Business Transaction by enclosing the code that constitutes the request that you want to monitor between StartBT() and EndBT() calls. StartBT() returns a handle to use in subsequent routines that affect that Business Transaction.

If you are creating a Business Transaction that correlates with an upstream Business Transaction, pass the correlation header of the upstream transaction so the new transaction that you are creating can correlate with it. See GetExitcallCorrelationHeader() in the Go SDK Reference. If the transaction does not need to correlate with another transaction, pass an empty string for the correlation header parameter.
You can optionally store the Business Transaction handle in the global handle registry with a guid for easy retrieval later using `StoreBT()`. Retrieve the handle from the global handle registry using `GetBT()`.

The following shows an example of setting a Business Transaction:

```go
// start the "Checkout" transaction
btHandle := appd.StartBT("Checkout", "")

// Optionally store the handle in the global registry
appd.StoreBT(btHandle, my_bt_guid)
...

// Retrieve a stored handle from the global registry
myBtHandle = appd.GetBT(my_bt_guid)

// end the transaction
appd.EndBT(btHandle)
```

Between starting and ending the transaction, you can perform operations such as adding errors to the Business Transaction, defining the transaction snapshot attributes, adding backends and exit calls, and so on.

When the Business Transaction ends, via a call to `EndBT()`, the agent stops reporting metrics for the Business Transaction.

**Add Business Transaction Errors**

Use `AddBTError()` to report Business Transaction errors. If you set this function's `markBTAsError` parameter, the transaction is reported as an error when the error occurs.

The SDK provides constants classifying the error level as `APPD_LEVEL_NOTICE`, `APPD_LEVEL_WARNING` or `APPD_LEVEL_ERROR`.

**Manage Business Transaction Snapshots**

When the agent is monitoring a Business Transaction, it automatically creates Transaction Snapshots, which describe instances of the Business Transaction at certain points in time. Transaction snapshots are extremely useful for troubleshooting poor performance because they contain a lot of detail.

You do not have to modify anything to create these snapshots, other than creating the Business Transaction, but you can add calls to:

- Find out if a snapshot is being taken
- Provide additional data in a snapshot
- Set the URL for a snapshot

**Determine if the Agent is Taking a Snapshot Now**

The agent is not always collecting snapshots, because that would be expensive. By default, it collects a snapshot every ten minutes, but this schedule is configurable. See 'Configure Snapshot Periodic Collection Frequency' in Transaction Snapshots for information about the frequency.

You can determine if a snapshot is happening using `IsBTSnapshotting()`, which returns `true` if the agent is collecting a snapshot. The main reason to do this is to avoid the wasted overhead for collecting user data for a snapshot or setting the snapshot URL if no snapshot is currently being collected.

**Add Business Transaction User Data**

You can optionally add data to transaction snapshots. For example, you might want to know which users are getting a lot of errors, or from which regions users are experiencing slow response times, or which methods are slow. In the AppDynamics UI, the data appears in the 'USER DATA' tab of the transaction snapshot.
If a snapshot is occurring, use `AddUserDataToBT()` passing a key and value for the data that you want the snapshot to collect.

**Add Snapshot URL**

If a snapshot is occurring, you can set a URL for the current snapshot using `SetBTURL()`.

**Set Snapshot Example**

```go
func setSnapshotAttributes(bt appd.BtHandle, key, value string) {
    if appd.IsBTSnapshotting(bt) {
        appd.AddUserDataToBT(bt, key, value)
        appd.SetBTURL(bt, "user/login")
    }
}
```

Create Backends

A backend is a database or a remote service such as a message queue, HTTP service or cache service that your application uses. A backend component is not itself monitored by the application agent, but the agent monitors calls to it from instrumented servers. You need to create backends in the instrumented environment so that the agent registers them. Creating and adding the backend to the instrumented application involves:

- Naming the backend
- Setting its identifying properties
- Optionally configuring how the backend is presented in the AppDynamics UI

**Identifying Properties**

A backend has identifying properties. These vary depending on the type of the backend and the types of information that you want to display. The Controller displays identifying properties in backend dashboards. You must set at least one identifying property for the type of any backend that you plan to add.

The following shows the backend properties in the Controller UI for an Oracle database:

**Resolve to a Tier**

By default the Controller doesn’t display a detected backend as a separate entity in flow maps, but the agent reports its metrics as part of the downstream tier. If you want to display the backend as a separate component and not resolved to the tier set `resolve` to false. See Resolve Remote Services to Tiers for more information.
**Backend Example**

The following listing shows an example of setting up a database backend.

```go
backendName := "Cart Product Database"
backendType := "DB"
backendProperties := map[string]string {
  "DATABASE": "sqlite3",
}
resolveBackend := false

appd.AddBackend(backendName, backendType, backendProperties, resolveBackend)
```

**Manage Exit Calls**

When an application makes a call to another component, such as a detected backend or another application server, the agent reports metrics on those calls.

Define an exit call by enclosing the code that constitutes the exit call between `StartExitCall()` and `EndExitcall()` calls. `StartExitCall()` returns a handle to use in subsequent routines that affect that exit call. An exit call occurs in the context of a Business Transaction.

You can optionally store the exit call handle in the global handle registry with a guid for easy retrieval later using `StoreExitcall()`. Retrieve the handle from the global handle registry using `GetExitcall()`.

You can optionally add details to an exit call as any arbitrary string with `SetExitcallDetails()`. The details are reported in the exit call details in the transaction snapshots in the Controller UI:

```
executeQuery
```  

You can also add errors to an exit call using `AddExitcallError()`. Use the enum for the error levels. You can also add an error message.

**Simple Exit Call Example**
// start the exit call to backendName
ecHandle := appd.StartExitcall(btHandle, backendName)
...

// optionally store the handle in the global registry
appd.StoreExitcall(ecHandle, my_ec_guid)
...

// retrieve a stored handle from the global registry
myEcHandle := appd.GetExitcall(my_ec_guid)

// set the exit call details
if err := appd.SetExitcallDetails(myEcHandle, "Exitcall Detail String"); err != nil {
    log.Print(err)
}

// add an error to the exit call
appd.AddExitcallError(myEcHandle, appd.APPD_LEVEL_ERROR, "exitcall error!", true)

// end the exit call
appd.EndExitcall(myEcHandle)

Correlate with Other Business Transactions

A correlation header contains the information that enables the agents to continue the flow of a Business Transaction across multiple tiers.

An SDK agent can correlate with other SDK agents as well as other AppDynamics agents, such as Java, .NET, or PHP, that perform automatic correlation for certain types of entry and exit points.

**Correlate with an Upstream Tier**

When your instrumented process receives a continuing transaction from an upstream agent that supports automatic correlation:

1. Using a third-party `Header.Get()` function, extract the header named `APPD_CORRELATION_HEADER_NAME` from the incoming HTTP payload.
2. Pass the header to `StartBT()`. For example:

   
   ```
   hdr := req.Header.Get(appd.APPD_CORRELATION_HEADER_NAME)
   bt := appd.StartBT("Fraud Detection", hdr)
   ```

If the header retrieved by the `Header.Get()` function is valid, the business transaction started by the `StartBT()` call will be a continuation of the Business Transaction started by the upstream service.

**Correlate with a Downstream Tier**

The downstream agent is watching for a correlation header named `singularityheader` in the HTTP payload.
If your SDK agent is making an exit call to a downstream agent that supports automatic correlation:

1. Set the name of the correlation header using `APPD_CORRELATION_HEADER_NAME`.
2. Begin the exit call.
3. Retrieve the correlation header from the exit call using the `GetExitcallCorrelationHeader()` function.
5. Inject a header named `APPD_CORRELATION_HEADER_NAME` with the value of the correlation header retrieved in step 3 into the outgoing payload of the HTTP request.
6. Make the request. For example:

```go
inventoryEcHandle := appd.StartExitcall(btHandle, "Inventory DB")
hdr := appd.GetExitcallCorrelationHeader(inventoryEcHandle)

client := &http.Client{
    CheckRedirect: redirectPolicyFunc,
}

// ...
req.Header.Add(appd.APPD_CORRELATION_HEADER_NAME, hdr)
resp, err := client.Do(req)
// ...
```

**Terminate the Agent**

Just before the application exits, terminate the agent.

```go
appd.TerminateSDK()
```
Go SDK Reference

On this page:

- Basic Types
- Config Struct
- ContextConfig Struct
- AddAppContextToConfig
- InitSDK
- TerminateSDK
- StartBT
- StartBTWithAppContext
- EndBT
- GetBT
- IsBTSnapshotting
- SetBTURL
- StoreBT
- AddBTError
- AddUserDataToBT
- AddBackend
- AddExitcallError
- StartExitcall
- EndExitcall
- GetCRowupType
- GetClusterRollupType
- GetExitcall
- GetExitcallCorrelationHeader
- SetExitcallDetails
- StoreExitcall

Related pages:

- Using Go SDK

Use the AppDynamics Go SDK to instrument Google Go applications. The API includes functions for creating business transactions, transaction backends, exit points, and more, as described here.

Basic Types

The AppDynamics Go SDK defines the following opaque types:

- **BtHandle**: A handle to an active business transaction
- **ExitcallHandle**: A handle to an active exit call

Config Struct

The struct `Config` contains settings used by the agent to connect to the AppDynamics Controller. The structure serves an equivalent function of the agent configuration files described in Agent-to-Controller Connections.

The `Config` struct is defined as follows:
type Config struct {
    AppName, TierName, NodeName string
    Controller Controller
    InitTimeoutMs int
    Initialized uint // a special field used by the underlying AppDynamics libraries. Do not use.
    Logging LoggingConfig
    UseConfigFromEnv bool
    EnvVarPrefix string
}

The structure is made up of these fields:

- **AppName**: Name of the business application to which this node belongs.
- **TierName**: Name of the tier to which this node belongs. A tier is a group of identical or similar nodes. An originating tier is the tier that receives the first request of a business transaction. A downstream tier is a tier that is called from another tier, thus continuing a business transaction.
- **NodeName**: The monitored application server to be monitored by this agent.
- **Controller**: Connection settings for the Controller. This struct is defined as:

  type Controller struct {
      Host string
      Port uint16
      Account, AccessKey string
      UseSSL bool
      HTTPProxy HTTPProxy
      CertificateFile
      CertificateDir
  }

  The Controller struct contains these fields:
  - **Host**: Controller host.
  - **Port**: Controller port; defaults to 8080.
  - **Account**: Name of your AppDynamics account.
  - **Access_key**: Key to your AppDynamics account. To find your access key, click the settings (gear) icon in the upper right corner of the AppDynamics UI, then click License.
  - **UseSSL**: Enable SSL communication with the controller; the default is false. Set to true to enable. If you set UseSSL to true, the default certificate, ca-bundle.crt, is used. To use your own certificate, specify your certificate file and directory in the CertificateFile and CertificateDir parameters.
  - **HTTPProxy**: If the agent needs to connect to the Controller via a local HTTP proxy, use this struct to specify connection settings for the proxy.
  - **CertificateFile**: The file name of the SSL certificate.
  - **CertificateDir** - The directory where the certificate is located.

- **InitTimeoutMs**: The initialization function, InitSDK relies on the Controller configuration to start business transactions. To prevent the initialization function from blocking your application, you can use this setting. It lets you instruct InitSDK whether to wait for the Controller configuration and if so how long to wait before starting business transactions. Valid values are:
  - **N**: Wait for up to N milliseconds for the Controller configuration.
  - **0**: Do not wait for the Controller configuration. This is the default.
  - **-1**: Wait indefinitely until the Controller configuration is received by the agent.
- **Logging**: Logging settings for the controller. This struct is defined as:
const (
    APPD_LOG_LEVEL_DEFAULT LogLevel = iota
    APPD_LOG_LEVEL_TRACE
    APPD_LOG_LEVEL_DEBUG
    APPD_LOG_LEVEL_INFO
    APPD_LOG_LEVEL_WARN
    APPD_LOG_LEVEL_ERROR
    APPD_LOG_LEVEL_FATAL
)

type LoggingConfig struct {
    BaseDir             string
    MinimumLevel        LogLevel
    MaxNumFiles         uint
    MaxFileSizeBytes    uint
}

The **Logging** struct contains these fields:

- **BaseDir**: The absolute path to the directory where log files are written. If left empty, the default is /tmp/appd.
- **MinimumLevel**: One of the APPD_LOG_LEVEL_xxx constants that are shown above. The default is APPD_LOG_LEVEL_INFO.
- **MaxNumFiles**: The maximum number of logging files to store on disk. Once this maximum is hit, older logs are rotated out.
- **MaxFileSizeBytes**: The maximum size of the log files before they are rotated.

- **UseConfigFromEnv**: Set to true if you want the SDK to check for any configuration environment variables and use those configuration values for initialization. Note that because this happens on initialization, the environment variable settings override the configuration you set in your program.
- **EnvVarPrefix**: If **UseConfigFromEnv** is set to true, use this property to specify the prefix to use for environment variable names. The default prefix is APPD_SDK.

If the agent needs to connect to the Controller via a local HTTP proxy server, you need to configure the settings for the proxy server. You can do so using the **HTTPProxy** struct, defined as follows:

```
type HTTPProxy struct {
    Host string
    Port  uint16
    Username, PasswordFile string
}
```

The **HTTPProxy** struct has the following fields:

- **Host**: Hostname or IP address of the HTTP proxy server
- **Port**: Port of the proxy server
- **Username**: Proxy server user name
- **PasswordFile**: Proxy server password file

**ContextConfig Struct**

You use the **ContextConfig** struct for calls that apply to multi-tenant Controller environments.
```go
type ContextConfig struct {
   AppName string
   TierName string
   NodeName string
}
```

The `ContextConfig` struct has the following fields:

- **AppName**: Name of the business application to which this node belongs.
- **TierName**: Name of the tier to which this node belongs. A tier is a group of identical or similar nodes. An *originating tier* is the tier that receives the first request of a business transaction. A *downstream tier* is a tier that is called from another tier, thus continuing a business transaction.
- **NodeName**: The monitored application server to be monitored by this agent.

**AddAppContextToConfig**

Add application context to the AppDynamics configuration for a multi-tenant Controller.

**Format**

```go
func AddAppContextToConfig(cfg *Config, context string, contextCfg *ContextConfig) error
```

**Parameters**

- `cfg`: AppDynamics configuration object.
- `context`: A unique identifier used to refer to this context.
- `contextCfg`: An AppDynamics context configuration object indicating the business application, tier, and node name for this context.

**InitSDK**

Initialize the AppDynamics Go SDK. An instrumented application must call this function once, preferably during application startup.

**Format**

```go
func InitSDK(cfg *Config) error
```

**Parameters**

- `cfg`: AppDynamics configuration settings that enable communication between the agent and the Controller.

**Returns**

- `nil` on success, otherwise an error value

**TerminateSDK**

Stop the AppDynamics Go SDK. Ends all active business transactions for this agent and stops agent metric reporting to the Controller.

**Format**

```go
func TerminateSDK()
```

**StartBT**

Start a business transaction or continue an existing transaction.

Keep in mind that each application is limited to 200 registered business transactions, and each agent is limited to 50 registered business transactions. Unlike transactions discovered by other types of agents, business transactions that exceed the limit that are reported by the Go SDK are not included in the all other traffic grouping. This means that it's up to you to ensure that your agent does not create excessive business transactions. Use care to ensure that your implementation does not introduce the possibility of business transaction explosion.

**Format**

```go
```
func StartBT(name, correlation_header string) BtHandle

Parameters

- name: The name for the business transaction. In the case of a continuing transaction in the current business application with a valid correlation header, the SDK uses the name from the header. Do not use the following characters in transaction names: `{ }[]|&;`
- correlation_header: A correlation header if this is a continuing transaction, else NULL. If specified, the correlation header has been generated by another AppDynamics agent and made available to this agent as a string. The correlation header provides information to enable this transaction to correlate with an upstream transaction.

Returns

An opaque handle for the business transaction that was started.

**StartBTWithAppContext**

Start a business transaction or continue an existing transaction in a multi-tenant Controller environment.

Format

func StartBTWithAppContext(context, name, correlation_header string) BtHandle

Parameters

- context: The application context name that this business transaction belongs to.
- name: The name for the business transaction. In the case of a continuing transaction in the current business application with a valid correlation header, the SDK uses the name from the header. Do not use the following characters in transaction names: `{ }[]|&;`
- correlation_header: A correlation header if this is a continuing transaction, else NULL. If specified, the correlation header has been generated by another AppDynamics agent and made available to this agent as a string. The correlation header provides information to enable this transaction to correlate with an upstream transaction.

**EndBT**

End the given business transaction.

Format

func EndBT(bt BtHandle)

Parameters

- bt: The handle to the business transaction to end.

**GetBT**

Get a BT handle associated with the given guid by `appd_bt_store`.

Format

func GetBT(guid string) BtHandle

Parameters

- guid: The globally unique identifier that was passed to `appd_bt_store`.

Returns

The handle to the business transaction associated with the given guid.

In the following cases the SDK logs a warning and returns a handle that you may safely use in other API functions but that will cause these functions to immediately return without doing anything:

- The SDK doesn't find a handle associated with the guid
- The call ended before the SDK could retrieve the handle

**IsBTSnapshotting**

Reports whether the agent is currently taking a transaction snapshot. Useful before calling `AddUserDataToBT` or `SetBTURL`, since those potentially expensive functions would do nothing if called when a snapshot is not being taken.

Format
func IsBTSnapshotting(bt BtHandle) bool
Parameters
- `bt`: The handle to the business transaction to check for snapshotting.

Returns
True if the given business transaction is taking a snapshot. Otherwise, false.

**SetBTURL**
Set the URL for a snapshot, if one is being taken. It is safe to call this function when a snapshot is not occurring. When the given business transaction is not snapshotting, this function immediately returns. However, if extracting the data to pass to this function is expensive, you can use `IsBTSnapshotting` to check if the business transaction is snapshotting before extracting the data and calling this function. The `url` argument data should be either 7-bit ASCII or UTF-8.

Format

```go
func SetBTURL(bt BtHandle, url string)
```
Parameters
- `bt`: The business transaction to add the user data to, if it is taking a snapshot.
- `url`: The value of the URL for the snapshot as 7-bit ASCII or UTF-8.

**StoreBT**
Store a BT handle in a global registry to retrieve later with `GetBT`. This is convenient when you need to start and end a business transaction in separate places, and it is difficult to pass the handle to the business transaction through the parts of the code that need it.

When the business transaction is ended, the handle is removed from the global registry.

Format

```go
func StoreBT(bt BtHandle, guid string)
```
Parameters
- `bt`: The BT to store.
- `guid`: A globally unique identifier to associate with the given BT.

**AddBTError**
Add an error to a business transaction. Errors are reported as part of the business transaction. However, you can add an error without marking the business transaction as an error (e.g., for non-fatal errors).

Format

```go
func AddBTError(bt BtHandle, level ErrorLevel, message string, mark_bt_as_error bool)
```
Parameters
- `bt`: The handle to the business transaction to which the error is added.
- `level`: The error level, from the following:
  - `APPD_LEVEL_NOTICE`
  - `APPD_LEVEL_WARNING`
  - `APPD_LEVEL_ERROR`
- `message`: Error message for this error.
- `mark_bt_as_error`: If true, the business transaction experiencing this error is marked as an error transaction. In this case, the business transaction is counted only as an error transaction. It is not also counted as a slow, very slow or stalled transaction, even if the transaction was also slow or stalled. If false, the business transaction is not marked as an error transaction.

**AddUserDataToBT**
Attaches user data to transaction snapshots generated for the specified business transaction.

The user data is added to the business transaction but reported only when a transaction snapshot is occurring. In the Controller UI, the user data appears in the USER DATA tab of the transaction snapshot details.

It is safe to call this function when a snapshot is not occurring. When the specified business transaction is not taking a snapshot, this
function immediately returns.

If extracting the data to pass to this function is expensive, you can use `IsBTSnapshotting` to check if the business transaction is actually taking a snapshot before extracting the data and calling this function.

The data in the `value` argument should be either 7-bit ASCII or UTF-8.

Format

```go
func AddUserDataToBT(bt BtHandle, key, value string)
```

Parameters

- `bt`: The business transaction to add the user data to, if it's taking a snapshot.
- `key`: The name of the user data to add to the snapshot as 7-bit ASCII or UTF-8.
- `value`: The value of the user data to add to the snapshot as 7-bit ASCII or UTF-8.

**AddBackend**

Add a backend to the business application.

This function fails if the type of the backend being added does not have at least one identifying property.

Format

```go
func AddBackend(name, backendType string, identifyingProperties map[string]string, resolve bool) error
```

Parameters

- `name`: The name of the backend. Must be unique to the Go SDK instance.
- `backendType`: The type of the backend, from these options:
  - `APPD_BACKEND_HTTP`
  - `APPD_BACKEND_DB`
  - `APPD_BACKEND_CACHE`
  - `APPD_BACKEND_RABBITMQ`
  - `APPD_BACKEND_WEBSERVICE`
  - `APPD_BACKEND_JMS`
- `identifyingProperties`: A map of key/value pairs that contain the backend's identifying properties. The properties uniquely identify the backend. In the Controller, these properties appear in the upper right panel of the backend dashboards. You must set at least one identifying property when you add a backend of one of the built-in exit call types. The valid properties vary per exit call type as indicated below.

<table>
<thead>
<tr>
<th>Exit Call Type</th>
<th>Valid Identifying Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>APPD_BACKEND_HTTP</code></td>
<td>&quot;HOST&quot;, &quot;PORT&quot;, &quot;URL&quot;, &quot;QUERY STRING&quot;</td>
</tr>
<tr>
<td><code>APPD_BACKEND_DB</code></td>
<td>&quot;HOST&quot;, &quot;PORT&quot;, &quot;DATABASE&quot;, &quot;VENDOR&quot;, &quot;VERSION&quot;</td>
</tr>
<tr>
<td><code>APPD_BACKEND_CACHE</code></td>
<td>&quot;SERVER POOL&quot;, &quot;VENDOR&quot;</td>
</tr>
<tr>
<td><code>APPD_BACKEND_RABBITMQ</code></td>
<td>&quot;HOST&quot;, &quot;PORT&quot;, &quot;ROUTING KEY&quot;, &quot;EXCHANGE&quot;</td>
</tr>
<tr>
<td><code>APPD_BACKEND_WEBSERVICE</code></td>
<td>&quot;SERVICE&quot;, &quot;URL&quot;, &quot;OPERATION&quot;, &quot;SOAP ACTION&quot;, &quot;VENDOR&quot;</td>
</tr>
<tr>
<td><code>APPD_BACKEND_JMS</code></td>
<td>&quot;DESTINATION&quot;, &quot;DESTINATIONTYPE&quot;, and &quot;VENDOR&quot;</td>
</tr>
</tbody>
</table>

For example, a key PORT would have a value of 3304.

- `resolve`: Set the property to false to prevent a downstream agent from resolving as this backend. If true, the default, an agent that detects a correlation header for an unresolved backend resolves itself as that backend. However, if the backend is actually an uninstrumented tier that is passing through the correlation header—for example, a message queue or proxy—you may wish the backend to show up distinct from the tier that it routes to. If false, correlation headers generated for exit calls to this backend will instruct downstream agents to report as distinct from the backend.

For example, if you have Tier A connecting to uninstrumented Backend B, which routes to instrumented Tier C, the setting affects the flow map as follows:

- True, the flow map will be A > C.
- False, the flow map will be A > B > C.

Returns
AddExitcallError
Add an error to the exit call.
Format

```go
func AddExitcallError(exitcall ExitcallHandle, level ErrorLevel, message string, mark_bt_as_error bool)
```

Parameters

- `exitcall`: Handle to the exit call.
- `level`: The level of this error, from the following:
  - `APPD_LEVEL_NOTICE`
  - `APPD_LEVEL_WARNING`
  - `APPD_LEVEL_ERROR`
- `message`: Error message for this error.
- `mark_bt_as_error`: If true, the business transaction making the exit call that is experiencing this error is marked as an error transaction. In this case, the business transaction is counted only as an error transaction. It is not also counted as a slow, very slow or stalled transaction, even if the transaction was also slow or stalled. If false, the business transaction is not marked as an error transaction.

StartExitcall
Start an exit call to the specified backend as part of a business transaction.
Format

```go
func StartExitcall(bt BtHandle, backend string) ExitcallHandle
```

Parameters

- `bt`: Handle to the business transaction making the exit call.
- `backend`: The destination backend of the exit call. AppDynamics does not automatically detect backends for Go applications, so you must specify the destination backend.

Returns
An opaque handle to the exit call that was started.

EndExitcall
Complete the exit call.
Format

```go
func EndExitcall(exitcall ExitcallHandle)
```

Parameters

- `exitcall`: Handle to the exit call being ended.

GetCRollupType
Specify how to roll up values for the metric over time.
Format

```go
func GetCRollupType(rollUp RollupType)
```

Parameters

- `rollUp`: The approach for rolling up metrics. The following roll-up options are valid:
  - `APPD_TIMEROLLUP_TYPE_AVERAGE`: The average of all metric values across each node in the tier.
  - `APPD_TIMEROLLUP_TYPE_SUM`: Sum of all reported values in the minute.
  - `APPD_TIMEROLLUP_TYPE_CURRENT`: The last reported metric in the minute.

GetCClusterRollupType
Specify how to aggregate metric values for the tier (a cluster of nodes).

Format

func GetCClusterRollupType(rollUp ClusterRollupType)

Parameters

- rollUp: The approach for rolling up metrics. The following roll-up options are valid:
  - APPD_CLUSTERROLLUP_TYPE_INDIVIDUAL: The average of all metric values across each node in the tier.
  - APPD_CLUSTERROLLUP_TYPE_COLLECTIVE: The sum of all metric values for all the nodes in the tier.

GetExitcall

Get a handle to an exit call associated with a guid via StoreExitcall.

Format

func GetExitcall(guid string) ExitcallHandle

Parameters

- guid: The globally unique identifier that was passed to appd_exitcall_store().

Returns

The handle to the exit call associated with the given guid.

In the following cases the SDK logs a warning and returns a handle that you may safely use in other API functions but that will cause these functions to immediately return without doing anything:

- The SDK doesn't find a handle associated with the guid
- The call ended before the SDK could retrieve the handle

GetExitcallCorrelationHeader

Get the header for correlating a business transaction.

If a business transaction makes exit calls that you want to correlate across, retrieve the correlation header using this function and inject it into your exit call's payload.

The returned string is freed when the exit call ends. Do not free it yourself.

Format

func GetExitcallCorrelationHeader(exitcall ExitcallHandle) string

Parameters

- exitcall: Handle to the exit call.

Returns

On success, returns a 7-bit ASCII string containing the correlation information. You can inject this string into the payload of the exit call. A downstream agent can then extract the header from that payload and continue the business transaction.

On error, returns the default header that prevents downstream business transaction detection.

SetExitcallDetails

Set the details string for an exit call. This can be used, for example, to add the SQL statement that a DB backend has executed as part of the exit call. This data is then visible in the exit calls details UI for the transaction snapshot.

Format

func SetExitcallDetails(exitcall ExitcallHandle, details string) error

Parameters

- exitcall: Handle to the exit call.
- details: An arbitrary string to add to the exit call.

Returns

Nil on success, otherwise an error value.
StoreExitcall

Store an exit call handle in a global registry for later retrieval with GetExitcall. This is useful when you need to start and end a call in separate places, and it is difficult to pass the handle through the parts of the code that need it.

The handle is removed when the exit call, or the BT containing it, ends.

Format

```go
func StoreExitcall(exitcall ExitcallHandle, guid string)
```

Parameters

- `exitcall`: The exit call to store.
- `guid`: A globally unique identifier to associate with the given call.

Example

```go
ExitcallHandle ec = StartExitcall(bt, "authdb");
StoreExitcall(ec, "login-exit");
```
IBM Integration Bus Agent

On this page:
- How It Works

The IBM Integration Bus (IIB) agent allows you to see the activity flow in your IIB instance. You can use the AppDynamics IIB agent to do the following:

- Correlate business transactions that flow through IIB to identify where time is spent processing transactions in the larger end-to-end architecture.
- Measure where business transaction processing time is spent within IIB flows, allowing operations staff and flow developers to identify any bottlenecks within the flows.

How It Works

The agent supports inbound and outbound correlation for HTTP, JMS, and IBM MQ messaging nodes. You can identify business transactions that originate in the broker itself by the transaction name, which contains the message flow name and the message flow node name where the BT originated.

To measure where the time is spent within the flows, the agent models each message flow node as a thread (with thread names corresponding to the message flow node name). You can view the per-node timings in the tree view of the Business Transactions dashboard for any business transaction that passed through IIB, as shown below:

The IIB agent also reports message flow node timing data, represented as thread segments in transaction snapshots, for business transactions that flow through IIB. Message flow node timing data is displayed in the Waterfall View.

Each broker is represented in the AppDynamics flow map as a tier, which is named as you specified in the configuration. Individual broker processes register with AppDynamics as nodes. The node names are generated from the Broker name and the Execution Group name.

This page describes IBM IIB in the context of it being monitored by AppDynamics. For a full description of IBM IIB and its capabilities, refer to the IBM product documentation.
IIB Supported Environments

On this page:

- IIB Agent Support

IIB Agent Support

**IIB Versions**

The IIB agent supports WMB v8, IIB v9, v10.

**Operating Systems**

The IIB agent supports the following operating systems:

- Linux x86-64 bit

**IIB Node Types**

The agent can continue business transactions detected upstream at the following node types:

- SOAPInput
- HTTPInput
- JMSInput
- MQInput

The agent can detect and tag exit calls for downstream correlation at the following node types:

- SOAPRequest
- HTTPRequest
- JMSOutput, JMSReply
- MQOutput, MQReply

For MQ, we use the MQRFH2 message header to provide correlation. Any applications consuming MQ messages from IIB with IIB agent must support the MQRFH2 header.

The agent can detect database backend calls for the following node types:

- DatabaseRetrieve
- DatabaseRoute

All nodes are represented within AppDynamics Business Transactions as Threads. The per node timings can be seen in the tree view of the Business Transaction dashboard, and in transaction snapshots.
Install the IIB Agent

On this page:

- Install the Agent
- Configure the IIB Agent
- Troubleshooting

Related pages:

- mqsichangeflowuserexits command
- cciRegisterUserExit

To begin monitoring IIB, you must install the IIB agent.

Install the Agent

1. Download the agent from https://download.appdynamics.com/download/.
2. Extract the IIB agent zip file into your chosen installation directory.

Configure the IIB Agent

1. Specify the settings for agent-Controller communication. In the controller-info.xml file, set the following properties:
   - account-name: Account name
   - account-access-key: Access key
   - application-name: Name of the application that the IIB server belongs to
   - controller-host: Hostname
   - controller-port: Port
   - controller-proxy-host: host name or IP address of any proxy needed for the agent to connect to the Controller
   - controller-proxy-port: port number of any proxy needed for the agent to connect to the Controller
   - controller-proxy-username: username used to authenticate with the proxy
   - controller-proxy-password: password used to authenticate the proxy user
   - controller-proxy-passwordfile: full path name of a file containing the password used to authenticate the proxy user
   - controller-ssl-enabled: Set 1 to enable, 0 to disable
   - controller-cert-file: Full path to PEM format X509 certificate for SSL
     See, Enable SSL for C++ on Libagent for more information on obtaining the file.
   - log-dir: Full path to the directory containing the IIB agent log files. Defaults to /tmp/appd. Any logs written prior to this configuration element being processed will be written there.
   - log-level: Set this property to trace|debug|info|warning|error
   - tier-name: Name of the tier representing the broker
   - user-exit: Strictly alphanumeric user exit name, as provided to the mqsichangebroker command
2. Stop the broker by running the command below. In order to configure user exits, your broker must be stopped.

   mqsistop <broker_name>

3. Run the following command to install the IIB agent user exit and enable it for all flows by default:

   mqsichangebroker <broker_name> -x <install-directory> -e <user_exit_name>

   If the agent is installed in the broker, it starts and stops with the broker.

4. Start the broker by running the following command:
You may wish to monitor a subset of your message flows. You can exclude less important flows from being monitored by running the following command:

```
mqsichangeflowuserexits <broker_name> -e <integrationServerName> -f <MessageFlow> -i <user_exit_name>
```

**Instrument Flows on Opt-in Basis**

If you have a high volume of activity flow in your IIB instance, you may want to select only the critical flows to instrument.

To instrument flows on an opt-in basis, install the agent with:

```
mqsichangebroker <broker_name> -x <install-directory>
```

and then add the agent to individual flows with:

```
mqsichangeflowuserexits <broker_name> -a <user_exit_name> -e <integrationServerName> -f <MessageFlow> -k <application_name>
```

An application node will register for each broker process that has one or more instrumented flows deployed to it. Business transactions are only created (or continued) for flows where the agent's user exit is active.

**Troubleshooting**

If you experience any functional issues with the agent, review the following information, and supply it to AppDynamics Support.

- The agent log. The IIB agent writes status information into the configured log directory, or /tmp/appd if the configuration has not been read at the time the message is logged. The status information can help diagnose issues with the agent status.
- The system log. The IIB broker writes status messages into the system log. The messages include information about the loading and operation of user exits, such as the AppDynamics IIB agent.
- Abend files in the IIB errors directory. The default location is /var/mqsi/common/errors.
- Core files, if they exist.

If the user-exit name provided in the controller-info.xml file is not strictly alphanumeric, the broker is unable to load the agent, and fails to start.

**Disabling the IIB Agent**

If you have an issue with the agent or if you want to disable it, complete the following:

1. Stop the broker `mqsistop <broker-name>`
2. Remove the AppDynamics user exit `mqsichangebroker <broker-name> -x "" -e ""`
3. Start the broker `mqsistart <broker-name>`

These steps disables the agent globally.

If you have enabled the agent on a flow-by-flow basis, do the following for each flow the agent is enabled on:
1. Stop the broker `mqsistop <broker-name>`
2. Disable the user exit for each flow `mqsichangebroker <broker-name> -a "" -e <integrationServerName> -f <messageFlow> -k <applicationName>`
3. Start the broker `mqsistart <broker-name>`
Deploy AppDynamics for Azure

AppDynamics integrates seamlessly with Microsoft Azure. AppDynamics offers visibility into .NET code execution and Microsoft Azure services out of the box, allowing you to troubleshoot performance bottlenecks and optimize the performance of your Microsoft Azure applications.

AppDynamics for Azure App Service

To install and deploy AppDynamics for Azure App Service:

- For Ops teams: Install the AppDynamics Azure Site Extension for .NET or Java:
  - See Install the AppDynamics Azure Site Extension for .NET
  - See Install the AppDynamics Site Extension for Java

AppDynamics for Azure Cloud Services and Azure Service Fabric

To install and deploy AppDynamics for Azure Service Fabric, see Install AppDynamics for Azure Service Fabric.

If you are instrumenting Cloud Services, see Install AppDynamics for Azure Cloud Services.

Upgrade the AppDynamics for Windows Azure NuGet Package

If you are currently using the AppDynamics.WindowsAzure NuGet Package, you will need to upgrade to a new NuGet package, depending on your Azure platform. See Upgrade AppDynamics.WindowsAzure NuGet Package.
Install AppDynamics for Azure App Service

There are two ways to install the .NET microservices agent into Azure App Service:

- Use the AppDynamics Azure Site Extension for .NET or Java:
  - See Install the AppDynamics Azure Site Extension for .NET
  - See Install the AppDynamics Site Extension for Java
- Use the AppDynamics for Windows Azure NuGet Package: See Install the AppDynamics .NET Microservices Agent

Azure Site Extension is used by Ops teams that may not have access to source files, or would not like to modify or recompile them, yet still want to monitor their Azure projects and solutions.

The NuGet Package is used by Dev teams to deploy the .NET agent so they can include the agent binaries, scripts, and configurations in their project and configure it using Visual Studio or other IDE.
Install the AppDynamics Azure Site Extension for .NET

You can use the Windows Azure Portal to add the AppDynamics Azure Site Extension to your Azure App Service web app. Azure Site Extension is used by Ops teams that may not have access to source files, or would not like to modify or recompile them, yet still want to monitor their Azure projects and solutions.

Prepare to Install

To install the AppDynamics for Windows Azure Site Extension, you need the following:

- Connection information for your AppDynamics Controller. See also Agent and Controller Compatibility.
- A Windows Azure account.
- An Azure web app to monitor.

If you are upgrading from a previous version, see Upgrade the AppDynamics Azure Site Extension.

Add the AppDynamics Azure Site Extension

Add the AppDynamics Azure Site extension as you would any site extension for any Azure web app.

1. Log in to the Windows Azure Portal.
2. Browse to your web app.

If you want to configure the .NET Agent using environment variables, add the environment variables before you install the AppDynamics Azure Site Extension. See configure the agent using environment variables

3. From the DEVELOPMENT TOOLS list, click Extensions.
4. Click +Add to install the version of the AppDynamics Azure Site Extension you want to add to your web app. After you install the AppDynamics Azure Site Extension, it appears in the installed extensions list.

Add the AppDynamics Azure Site Extension Using an ARM Template

You can deploy the AppDynamics Azure Site Extension to Azure App Services using an Azure Resource Manager (ARM) template.

The following procedure uses Visual Studio Community 2017.

To create and deploy an ARM template:

1. From your web application in Visual Studio, choose File > New Project.
2. Click Cloud, then click Azure Resource Group, then click OK.
3. From the Select Azure Template dialog box, click Web App, then click OK.
4. Under your newly-created Resource Group, click the Website.json file.
5. Under JSON outline in the left pane, right-click resources, then choose Add New Resource.
6. In the Add Resource dialog box, select Application Settings for Web Apps, enter a name, then click OK.
7. Under the properties section, enter your AppDynamics Controller information. The following example shows the Website.json file with the application properties for AppDynamics.
8. Add a new apiVersion section with your Azure Site Extension details, as shown in the following example.
Here is the sample text for copying or pasting:

```json
{
  "apiVersion": "2015-08-01",
  "name": "[variables('webSiteName')]",
  "type": "Microsoft.Web/sites",
  "location": "[resourceGroup().location]",
  "tags": {
    "[concat('hidden-related:', resourceGroup().id, '/providers/Microsoft.Web/serverfarms/', parameters('hostingPlanName'))]": "Resource",
    "displayName": "Website"
  },
  "dependsOn": ["[resourceId('Microsoft.Web/serverfarms/', parameters('hostingPlanName'))]"],
  "properties": {
    "apiVersion": "2016-08-01",
    "name": "appsettings",
    "type": "config",
    "dependsOn": ["[resourceId('Microsoft.Web/Sites', variables('webSiteName'))]"]
  },
  "resources": [
    {
      "apiVersion": "2015-08-01",
      "name": "appsettings",
      "type": "appsettings",
      "dependsOn": ["[resourceId('Microsoft.Web/Sites', variables('webSiteName'))]"]
    }
  ]
}
```
"name": "appsettings",
"type": "config",
"dependsOn": [
  "{resourceId('Microsoft.Web/Sites',
  variables('webSiteName'))}"
],
"properties": {
  "appdynamics.controller.hostName":
  "leargasstage.saas.appdynamics.com",
  "appdynamics.controller.port": "443",
  "appdynamics.controller.ssl.enabled": "true",
  "appdynamics.agent.accountName": "leargasstage",
  "appdynamics.agent.accountAccessKey":
  "[parameters('danielAppDAccessKeyInKeyVault')]",
  "appdynamics.agent.applicationName": "HelloWorldSecureAppDKey",
  "appdynamics.agent.tierName": "TestTier",
  "appdynamics.agent.nodeName": "TestNode"  
}]
},
Configure the Controller Connection

You have the following options to configure the .NET Agent to connect to the AppDynamics Controller:

- Configure the Controller connection using the Kudu console.
- Configure the Controller connection settings using environment variables.

For more detail, see Agent-to-Controller Connections.

Configure the Agent with the Kudu Console

When you add the AppDynamics Azure Site Extension to your web app, you can interactively configure the .NET Agent using the Kudu console.

1. Navigate to the AppDynamics Controller Configuration page in the Kudu Console:
   http://[web app].scm.azurewebsites.net/appdynamics/
   For example: https://myazureexample.scm.azurewebsites.net/appdynamics/
2. On the AppDynamics Controller Configuration page, enter your Controller connection information. For example:

   ![AppDynamics Controller Configuration](image)

3. Click Validate to test the connection to the AppDynamics Controller and save your settings.
4. Restart your web app.
   After you apply some load to your web app, you can view it on flow maps in the AppDynamics Controller UI.

Configure the Agent Using Environment Variables

Configuring the .NET Agent using environment variables allows for unattended configuration. To configure agents in this manner, add the environment variables before you install the AppDynamics Azure Site Extension, as follows:
1. Navigate to SETTINGS > Application Settings for your web app.
2. Add the .NET Agent environment variables under App settings:
   - appdynamics.controller.hostName: the address for the AppDynamics Controller
   - appdynamics.controller.port: the Controller port
   - appdynamics.agent.accountName: the account name you use to log on to the Controller
   - appdynamics.agent.accountAccessKey: the account key you use to log on to the Controller
   - appdynamics.agent.applicationName: the business application name in the Controller
   - appdynamics.controller.ssl.enabled: set to "True" to enable SSL connection to the Controller. Otherwise set to "False". For example:

   ![App settings table]

3. Restart your web app. After you apply some load to your web app, you can view it on flow maps in the AppDynamics Controller UI.

**Specify Which WebJobs to Monitor**

Use the APPDYNAMICS.PROCESSLIST environment variable to specify which WebJobs the .NET Agent monitors.

Environment Variable: APPDYNAMICS.PROCESSLIST
Type: Strings separated by "|
Default: None. When undefined, the agent instruments all WebJob running in the app service. When defined, the .NET Agent instruments only the specified processes.
Required: No
Usage Cases:

- Environment 1: An Azure app service with no WebJobs, where the Appdynamics .NET Agent site extension is installed and configured. After defining the environment variable AppDynamics.ProcessList with the value w3wp.exe; the Daasrunner.exe WebJob, which is a default WebJob that is added to all app services, is not instrumented.
- Environment 2: An Azure app service, with two WebJobs, webjob1.exe and webjob2.exe, where the Appdynamics .NET Agent site extension is installed and configured. After defining the environment variable AppDynamics.ProcessList with the value w3wp.exe|webjob1.exe|webjob2.exe, the worker process serving the web app and the webjob1.exe are instrumented, but the webjob2.exe is not instrumented. With the AppDynamics.ProcessList removed from the App settings of the Azure app service, all the WebJobs, including Daasrunner.exe, are instrumented, including the worker process serving the web app.

**Upgrade the AppDynamics Azure Site Extension**

When you click on the Extensions tab for your web app, the Microsoft Azure Portal displays the currently installed version of the AppDynamics Azure Site Extension. The Update Available column of the installed extensions list indicates if there is a more recent minor release of the .NET Agent available. If so, you can click to update the extension from the list.

**Upgrade a Major Version of the .NET Agent**

AppDynamics maintains major release versions of the .NET Agent as separate site extensions. Therefore you need to uninstall the installed version of the AppDynamics Azure Site Extension before you upgrade to a new major release:

1. Log in to the Windows Azure Portal.
2. Stop your web app.
3. Click on the AppDynamics Azure Site Extension from the list of installed extensions and click **Delete** to uninstall it.
4. Install the new version of the AppDynamics Azure Site Extension as normal.
If you are upgrading from version 4.2 of the .NET agent and you used environment variable configuration, you must update your web app environment variables. See configure the agent using environment variables. Note that the APPD_UNATTENDED variable is no longer required.
Install the AppDynamics .NET Microservices Agent


Prepare to Install the NuGet Package

To install the AppDynamics for Windows Azure NuGet package you need the following:

- Connection information for your AppDynamics Controller version 4.4
- Visual Studio 2012 or later
- A Visual Studio solution to monitor
- Windows Azure SDK
- Windows Azure account

These instructions assume you are familiar with NuGet package management in Visual Studio and with the Microsoft Azure Portal. You can also try the step-through tutorial on how to instrument an App Service on the community knowledgebase.

Add the .NET Microservices Agent to Your Azure Solution


For details about managing NuGet packages, see the documentation for your version of Visual Studio.

The `AppDynamics.Agent.Azure.AppService.Windows` package installation prompts you as follows:

1. The installer may ask you to verify your changes. If so, click **OK**.
2. Click **I accept** to agree to the terms of the license.
3. In the Solution Explorer, click the `AppDynamicsConfig.json` file.
4. Enter the connection information for your AppDynamics Controller in the area on the left.

For information about the `AppDynamicsConfig.json` file, see `AppDynamicsConfig.json File`.

Configure the Controller Connection

You have the following options to configure the .NET Agent to connect to the AppDynamics Controller:

- During development: As shown in the previous screenshot, you can enter your environment variables in the `AppDynamicsConfig.json` file, and save it in source control.
- During build: Define your msbuild parameters or environment variables that are passed to the `AppDynamicsConfig.json` file at build time. `AppDynamicsConfig.json` does exist at build time. So, if you're defining your msbuild parameters/env variables at build time, you will need to ignore it in source control so that the new `AppDynamicsConfig.json` file will be created.
- During runtime: Enter your environment variables in Azure.

By default, the .NET microservices agent names Azure tiers as site name. The .NET microservices agent names Azure nodes by...
machine name appended with the website hostname.
Monitor Virtual Applications of Azure Web Apps

This topic applies to the .NET Agent in the Azure App services environment. It describes how the .NET Agent by default names your virtual application tiers and nodes and also how you can customize tier names.

There are two ways to automatically name the tiers of virtual applications:

- Default: AppDynamics creates new tiers for each virtual directory. Each tier contains a node representing the virtual directory.
- Custom: When you specify a tier name in the `AppDynamicsConfig.json` file, all virtual directory applications will report as nodes under this tier.

**Default Behavior**

The following shows the Azure Web Apps configuration of the `demomultiapp` application with three virtual apps, the main application and two sub-applications.

Once you instrument the application with the .NET Agent, restart the application and apply load to it, the .NET Agent reports metrics for the application and virtual applications. In the Controller UI, each virtual application node appears under a separate tier. The name of the tier consists of the name of the application appended with the virtual path of the sub-application. For example, `demomultiapp/billing` as shown below. The name of the node is the same as the tier name prepended with the name of the virtual machine on which it runs.
You can manually edit the tier name in the Controller UI.

**Customize Tier Names**

Rather than creating a tier for each virtual application, all virtual applications can report as nodes under a single tier. The tier name specified in the AppDynamicsConfig.json file sets the tier name.

Notes:

- When you instrument your application using the .NET Agent site extension, the .NET Agent automatically instruments all virtual directories and applications. However, all AppSettings set at the root app are inherited by all virtual applications.
- For NuGet-based deployments, the agent uses the AppDynamicsConfig.json file deployed in the same path as the Profiler DLL and ignores the AppDynamicsConfig.json file settings for individual virtual apps in the deployment.

For example, in Azure Web Apps we have the following virtual app and sub-applications.
By default, the tier name of the application in the AppDynamicsConfig.json file is null. The following edited version of the AppDynamicsConfig.json file shows a tier name of Business App 15. Note that the node name cannot be changed as its value is automatically updated as the application scales vertically and horizontally.

```json
{
    "controller": {
        "host": "10.10.10.10", "appdynamics", "port": "8080",
        "account": "admin", "password": "admin",
        "ssl": "true", "enable_tls12": "true",
        "proxy": "null"
    },
    "application": {
        "name": "Demo Multi App",
        "tier": "Business App 15",
        "node": "null"
    },
    "instrumentors": {
        "customCorrelationConfig": "null",
        "enable": "false",
        "disable": "false"
    }
}
```

After the application is restarted, the virtual app tier name appears as Business App 15. When you expand the tier on the Tiers & Nodes Dashboard, you see the nodes of the virtual applications and the root application.

The Nodes Dashboard displays the nodes of the tier Business App 15 in a similar manner.
Nodes of the virtual application are displayed in the Controller UI using the name of the virtual machine the app is running on, followed by the name of the virtual application as it appears in Azure Web Apps.

The current agent configuration file does not support targeting different virtual applications for customized tier naming, however, you can manually create .NET tiers and move the nodes to different tiers in the Controller UI.

Prevent Daas Instrumentation

You can prevent the agent from instrumenting the Daasrunner process by creating the environment variable `AppDynamics.ProcessInstrumentationList` and setting it to `w3wp.exe`. This environment variable acts as a whitelist, where only the processes specified will be instrumented. Then delete the Daas tier and the Daasrunner node on the Tiers & Nodes Dashboard. You might also want to include WebJob processes in this list. See ‘Specify Which WebJobs to Monitor’ on Install the AppDynamics Azure Site Extension for .NET.

Limitations

In this version, you cannot customize the tier names of virtual sub-applications in the `AppDynamicsConfig.json` file.

Also, this version does not support monitoring virtual apps together with deployment slots.
Install AppDynamics for Azure Service Fabric

The following instructions explain how to deploy the .NET microservices agent for AppDynamics for Azure Service Fabric.

Before you Install


Choose a procedure below to match your environment:

- Deploy the Package using Visual Studio
- Deploy the Package using another IDE

Deploy the Package using Visual Studio

1. From your editor, open ServiceManifest.xml in each folder in the published application root.
2. Add the following environment variables in each CodePackage:

   ```xml
   <EnvironmentVariables>
       <EnvironmentVariable Name="COR_ENABLE_PROFILING" Value="1" />  
       <EnvironmentVariable Name="COR_PROFILER" Value="{39AEABC1-56A5-405F-B8E7-C3668490DB4A}" />
       <EnvironmentVariable Name="COR_PROFILER_PATH" Value="AppDynamics/AppDynamics.Profiler_x64.dll" />
   </EnvironmentVariables>
   ```

3. Make a copy of the AppDynamicsConfig.json file and rename it to <executable_name>.AppDynamicsConfig.json. For example, <<servicefabricapplicationname>>.AppDynamicsConfig.json.
4. Copy AppDynamics.Agent.dll, AppDynamics.Profiler_x64.dll, AppDynamicsAgentLog.config, AppDynamicsConfig.json from <nuget_package>\content\AppDynamics and add these files in the Visual Studio solution of each service project at the top level, not under any subfolders.
5. Make a copy of AppDynamicsConfig.json and rename it to:<<executable_name>>.AppDynamicsConfig.json, for example, <<servicefabricapplicationname>>.AppDynamicsConfig.json. Put it into the root of each service project.

   When you rename the .json file to the application name, do not include .exe at the end of your application name.

6. Right-click on the files in the AppDynamics sub-folder and select Copy Always.
7. Update <<executable_name>>.AppDynamicsConfig.json with your configuration information:

   ```json
   {  
       "controller":  
       {  
           "host": "",  
           "port": 0,  
           "account": "",  
           "password": ""  
       },  
       "application":  
       {  
           "name": ""  
       }  
   }  
   ```

   Do not specify node; it will be assigned automatically; Specifying tier is optional, it could be assigned automatically.

   For information about the AppDynamicsConfig.json file, see AppDynamicsConfig.json File.

   Next steps are optional and should only be used if you need per service-instance control of the controller/application/tier.
1. Add the following environment variables to each CodePackage:
   <EnvironmentVariable Name="appdynamics.controller.hostName" Value="<<  >>" />
   <EnvironmentVariable Name="appdynamics.controller.port" Value="<<  >>" />
   <EnvironmentVariable Name="appdynamics.agent.accountName" Value="<<  >>" />
   <EnvironmentVariable Name="appdynamics.agent.accountAccessKey" Value="<<  >>" />
   <EnvironmentVariable Name="appdynamics.agent.applicationName" Value="<<  >>" />
   <EnvironmentVariable Name="appdynamics_agent_tier_name" Value="<<  >>" />

2. Modify ApplicationManifest.xml. In each ServiceManifestImport and CodePackage folder add:
   - In ServiceManifestImport:
     <EnvironmentOverrides CodePackageRef="Code">
     <EnvironmentVariable Name="appdynamics.controller.hostName" Value="<<AppD_ControllerHostName>>" />
     <EnvironmentVariable Name="appdynamics.controller.port" Value="<<AppD_ControllerPort>>" />
     <EnvironmentVariable Name="appdynamics.agent.accountName" Value="<<AppD_AccountName>>" />
     <EnvironmentVariable Name="appdynamics.agent.accountAccessKey" Value="<<AppD_AccountAccessKey>>" />
     <EnvironmentVariable Name="appdynamics.agent.applicationName" Value="<<AppD_ApplicationName>>" />
     <EnvironmentVariable Name="appdynamics_agent_tier_name" Value="<<AppD_Service_Name_TierName>>" />
     </EnvironmentOverrides>
   - In Parameters:
     <Parameter Name="AppD_ControllerHostName" DefaultValue="<<Your_Controller_Name>>" />
     <Parameter Name="AppD_ControllerPort" DefaultValue="<<Your_Controller_Port>>" />
     <Parameter Name="AppD_AccountName" DefaultValue="<<Your_Controller_Account_Name>>" />
     <Parameter Name="AppD_AccountAccessKey" DefaultValue="<<Your_Controller_Access_Key>>" />
     <Parameter Name="AppD_ApplicationName" DefaultValue="<<Your_Controller_App_Name>>" />
   - For each service:
     <Parameter Name="AppD_%SERVICE_NAME%_TierName" DefaultValue="<<Tier_Name_For_Service>>" />

3. If you need to provide the AppDynamics configuration during application package deployment, you can now do it using the -ApplicationParameter switch of the New-ServiceFabricApplication cmdlet.

Deploy the Package using another IDE

1. From your editor, open ServiceManifest.xml in each folder in the published application root.
2. Add the following environment variables in each CodePackage:
   <CodePackage ...>
   <EntryPoint>...
   </EntryPoint>
   <EnvironmentVariables>
     <EnvironmentVariable Name="COR_ENABLE_PROFILING" Value="1" />
     <EnvironmentVariable Name="COR_PROFILE" Value="{39AEABC1-56A5-405F-B8E7-C3668490DB4A}" />
     <EnvironmentVariable Name="COR_PROFILER_PATH" Value="AppDynamics/AppDynamics.Profiler_x64.dll" />
   </EnvironmentVariables>
   </CodePackage>
3. Using PowerShell, unpack AppDynamics.Agent.dll, AppDynamics.Profiler_x64.dll, AppDynamicsAgentLog.config from <nuget_package>\content\AppDynamics to the code package folder that contains the executable you want to run.
4. Make a copy of the of the AppDynamicsConfig.json file from <nuget_package>\content\AppDynamics to the code package folder that contains the executable you want to run.

When you rename the .json file to the application name, do not include .exe at the end of your application name.

5. Make a copy of AppDynamicsConfig.json and rename it to <<executable_name>>.AppDynamicsConfig.json. For example, <<servicefabricapplicationname>>.AppDynamicsConfig.json. Put it into the root of each service project.
6. Update `<<executable_name>>.AppDynamicsConfig.json` with your configuration information:

```
{
  "controller":
  {
    "host":"",
    "port":0,
    "account":"",
    "password":"
  },
  "application":
  {
    "name":""
  }
}
```

Do not specify the node; it will be assigned automatically. Specifying the tier is optional; it could be assigned automatically.

For information about the `AppDynamicsConfig.json` file, see [AppDynamicsConfig.json File](#).

Next steps are optional and should only be used if you need per service-instance control of the controller/application/tier.

1. Add the following environment variables to each CodePackage:

   ```
   <EnvironmentVariable Name="appdynamics.controller.hostName" Value="" />
   <EnvironmentVariable Name="appdynamics.controller.port" Value="" />
   <EnvironmentVariable Name="appdynamics.agent.accountName" Value="" />
   <EnvironmentVariable Name="appdynamics.agent.accountAccessKey" Value="" />
   <EnvironmentVariable Name="appdynamics.agent.applicationName" Value="" />
   <EnvironmentVariable Name="appdynamics_agent_tier_name" Value="" />
   ```

2. Modify `ApplicationManifest.xml`. In each `ServiceManifestImport` and `CodePackage` folder add:

   - In `ServiceManifestImport`:
     ```
     <EnvironmentOverrides CodePackageRef="Code">
       <EnvironmentVariable Name="appdynamics.controller.hostName" Value="<<AppD_ControllerHostName>>" />
       <EnvironmentVariable Name="appdynamics.controller.port" Value="<<AppD_ControllerPort>>" />
       <EnvironmentVariable Name="appdynamics.agent.accountName" Value="<<AppD_AccountName>>" />
       <EnvironmentVariable Name="appdynamics.agent.accountAccessKey" Value="<<AppD_AccountAccessKey>>" />
       <EnvironmentVariable Name="appdynamics.agent.applicationName" Value="<<AppD_ApplicationName>>" />
       <EnvironmentVariable Name="appdynamics_agent_tier_name" Value="<<AppD_Service_Name_TierName>>" />
     </EnvironmentOverrides>
     ```

   - In Parameters:
     ```
     <Parameter Name="AppD_ControllerHostName" DefaultValue="<<Your_ControllerHostName>>" />
     <Parameter Name="AppD_ControllerPort" DefaultValue="<<Your_ControllerPort>>" />
     <Parameter Name="AppD_AccountName" DefaultValue="<<Your_Controller_AccountName>>" />
     <Parameter Name="AppD_ApplicationName" DefaultValue="<<Your_Controller_ApplicationName>>" />
     <Parameter Name="AppD_Service_Name_TierName" DefaultValue="<<AppD_Service_Name_TierName>>" />
     ```

   - For each service:
     ```
     <Parameter Name="<<AppD_Service_Name_TierName>>" DefaultValue="<<Tier_Name_For_Service>>" />
     ```

3. If you need to provide the AppDynamics configuration during application package deployment, you can now do it using the `-ApplicationParameter` switch of the `New-ServiceFabricApplication` cmdlet.
Install AppDynamics for Azure Cloud Services
You can use Microsoft Visual Studio with the NuGet package `AppDynamics.Agent.Azure.CloudServices` to add the .NET Agent directly to Azure Cloud Services projects: Web Roles and Worker Roles.

Prepare to Install the NuGet Package
To install the AppDynamics for Windows Azure NuGet package you need the following:

- Connection information for your Agent-to-Controller Connections version 4.4
- Visual Studio 2012 or later
- A Visual Studio solution to monitor
  The user account running Visual Studio must have the following permissions to the solution:
  - Read and Write permissions to each project directory
  - Read and Write permissions to each Visual Studio .NET C# Project (*.csproj) file
  - Read and Write permissions to the Service Definition (ServiceDefinition.cadl) file
- Windows Azure SDK
- Windows Azure account

These instructions assume you are familiar with NuGet package management in Visual Studio and with the Microsoft Azure Portal. You can also try the step-through tutorial on how to instrument an App Service on the community knowledgebase.

If you are upgrading to a new version of the AppDynamics for Windows Azure NuGet Package, see Upgrade `AppDynamics.WindowsAzure NuGet Package`.

Add the .NET Agent to Your Azure Solution

For detail about managing NuGet packages, see the documentation for your version of Visual Studio.

The `AppDynamics.Agent.Azure.CloudServices` package installation prompts you as follows:

- The installer may ask you to verify your changes. If so, click OK.
- Click I accept to agree to the terms of the license.

The package manager installs the .NET agent into your project.

The configuration file installs to `<project_root>/App_Data/AppDynamics/Config/config.xml`. Enter the connection information for your AppDynamics Controller in the config.xml file. For details about Controller connection settings, see Agent-to-Controller Connections.

```xml
<app-agents azure="true" azure-role-name="" azure-role-instance-Id=""

<app-agents>
  <standalone-applications>
    <standalone-application executable="WebWorkerHost.exe">
      <tier name="/" />
    </standalone-application>
  </standalone-applications>
</app-agents>
</app-agents>
```
By default the .NET Agent names Azure tiers as role name for Cloud Services. If you want to customize the tier name, edit config.xml under `<project_root>/App_Data/AppDynamics/Config.xml`. For more information and examples, see “Name IIS Tiers Manually” on Name .NET Tiers.

After you finish any configuration changes, publish your solution to Windows Azure. For Cloud Services solutions, you can put load on your published project and log on to the AppDynamics Controller and begin monitoring your solution.

Update .NET Agent Configuration for Cloud Services

If you have already published your solution, you can update the .NET Agent configuration for the currently installed version without upgrading the agent.

1. Edit the config.xml file to make configuration changes.
2. Edit the startup.cmd file under `<solution_root>/AppDynamics`.
3. By default, the APPD_AGENT_CONFIGUPDATE variable is set to true. If you want to modify the configuration on your Azure Cloud Services machine, and you want to prevent the AppDynamics script from overwriting it, set APPD_AGENT_CONFIGUPDATE to false.
Upgrade AppDynamics.WindowsAzure NuGet Package

If you are currently using the AppDynamics.WindowsAzure NuGet package to monitor your Azure solutions, you must upgrade to one of the newest packages on nuget.org, depending on your platform and the applications that you want to monitor. For a list of packages, see .NET Microservices Agent.

To upgrade your Azure NuGet package:

1. In Visual Studio or other IDE, uninstall the AppDynamics.WindowsAzure NuGet package.

   For Azure App Service only: After you uninstall the NuGet package, you must delete the AppDynamics folder from the Azure portal and your Visual Studio solution. You may need to stop your application first before deleting this folder. During uninstallation, the webconfig modification that is responsible for filtering user requests against the AppDynamics folder is removed, but this does not remove the AppDynamics folder. You must manually delete this folder. You must also remove the AppDynamics-specific information from apphostconfig.xdt.

2. Install one of the new NuGet packages. See .NET Microservices Agent for a list of packages.
## Release Notes and PDFs

<table>
<thead>
<tr>
<th>Release Notes</th>
<th>Associated PDF</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4.3 Azure Enhancements and Resolved Issues</td>
<td>4.4.3_AppDynamics_2018-04-02.pdf</td>
</tr>
</tbody>
</table>
### 4.4.3 Azure Enhancements and Resolved Issues

**Enhancements:**

- Microsoft Orleans is supported on the .NET and .NET microservices agents.
- .NET Core 2.0 for Windows is supported on the .NET microservices agent.
- Support for Azure Service Fabric Remoting v2.

The following NuGet packages are now available. If you have used a previous version of an AppDynamics NuGet package, you will need to upgrade to the new version. See [Upgrade AppDynamics Windows Azure NuGet Package](#).

<table>
<thead>
<tr>
<th>Package Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AppDynamics.Agent.Distrib.Micro.Windows</td>
<td>AppDynamics NuGet package for .NET. This package <em>should not</em> be installed directly and is intended for download and file distribution. This package is used for Azure Service Fabric deployments. See <a href="#">Install AppDynamics for Azure Service Fabric</a> for instructions.</td>
</tr>
<tr>
<td>AppDynamics.Agent.Windows</td>
<td>AppDynamics .NET microservices agent for Windows. Recommended for standalone installations. See <a href="#">Install the .NET Core Microservices Agent for Windows</a> for deployment instructions. NOTE: This package does not support .NET Framework, only .NET Core for Windows.</td>
</tr>
</tbody>
</table>
Standalone Machine Agents

Standalone Machine Agents and Network Agents give you end-to-end visibility into the hardware and networks on which your applications run. These agents can help you identify and troubleshoot problems that can affect application performance such as server failures, JVM crashes, and network packet loss.

You use the Standalone Machine Agent to collect basic hardware metrics. The functionality provided by the Machine Agent includes:

- Reporting basic hardware metrics from the server OS, for example, %CPU and memory utilization, disk and network I/O
- Reporting metrics passed to the Controller by extensions
- Running remediation scripts to automate your runbook procedures. You can optionally configure the remediation action to require human approval before the script is started.
- Running JVM Crash Guard to monitor JVM crashes and optionally run remediation scripts

If you have a Server Visibility license, the Standalone Machine Agent provides the following additional functionality:

- Extended hardware metrics such as machine availability, disk/CPU/virtual-memory utilization, and process page faults
- Monitor internal or external HTTP and HTTPS services
- Group servers together so that health rules can be applied to specific server groups
- Define alerts that trigger when certain conditions are met or exceeded based on monitored server hardware metrics
Administer App Server Agents

On this page:
- Planning Agent Deployment

Related page:
- Install App Server Agents

The following pages provide information on installing app agent, along with general information on installing AppDynamics app agents.

For Agent-to-Controller version compatibility information, see Agent and Controller Compatibility.

Planning Agent Deployment

When planning your deployment, you should consider whether to install agents manually or using some form of automation. If deploying a relatively small number of agents, you may choose to install the agent manually. For larger environments involving hundreds of agents, it is likely that you would want to develop an automated deployment strategy.

The pages in this section contain information applicable across agent types. For specific types of agents, see the following links:

- Java: Java Agent or Install the Java Agent.
- .NET: .NET Agent or Install the .NET Agent for Windows.
- Node.js: Node.js Agent or Install the Node.js Agent.
- PHP: PHP Agent or Install the PHP Agent.
- Python: Python Agent or Install the Python Agent.
- Database Agent: Administer the Database Agent
- Standalone Machine Agent: Install the Standalone Machine Agent

For automated deployment guidelines, see Controller Deployment.
Encrypt Agent Credentials

AppDynamics agents store various types of credential information on disk, including:

- Controller account access key
- Controller keystore/agent truststore password
- Proxy server password

For environments where security policies require you to secure credentials stored on disk, you can use the Secure Credential Store to encrypt credentials for use in agent configuration.

Two components comprise the Secure Credential Store:

- `scs-tool.jar`: A utility to create the secure credential store, encrypt credentials, and obfuscate the credential store password
- Secure credential keystore: A keystore for the secret encryption key

The secure credential store utility encrypts plain text using the strongest encryption available according to the system's encryption jurisdiction policy.

For the .NET Agent, see Encrypt Credentials in .NET Agent Configuration.

**Required: Update Agent Properties**

After you set up the Credential Keystore, you must specify the following settings:

**Analytics Agent**

In Analytics Agent Properties:

- `ad.secure.credential.store.filename`
- `ad.secure.credential.store.password`

**Java, Machine, and Database Agents**

In each agent's respective properties (Java Agent Properties, Machine Agent Properties, Database Agent Properties):

- `<controller-ssl-enabled>`
- `<controller-keystore-filename>`
- `<controller-keystore-password>`
Initialize the Secure Credential Store

Before you can encrypt or obfuscate passwords, you must run the secure credential store utility to create the keystore for your secret encryption key. The agent distribution includes the secure credential store utility in the following locations:

- **Java Agent:** `<javaagent_home>/verX.X.X.X/utils/scs/scs-tool.jar`
- **Machine Agent:** `<machine_agent_home>/lib/secure-credential-store-tool-1.3.0.0.jar`
- **Database Agent:** `<database_agent_home>/lib/scs-tool.jar`
- **Analytics Agent:** `<analytics_agent_home>/bin/tool/scs-tool.jar`

Run the secure credential store utility `generate_ks` command with the following parameters:

- `filename`: Absolute path where the utility will create the secure credential keystore. Use this path for `<credential-store-filename>` in agent configuration.
- `storepass`: The secure credential keystore password. Use the obfuscated version of this password as the value for `<credential-store-password>` in agent configuration.

For example:

```bash
<full path to application JRE>/bin/java -jar foo ./scs-tool.jar generate_ks -filename '/opt/appdynamics/secretKeyStore' -storepass 'MyCredentialStorePassword'
```

The secure credential store utility confirms it created and initialized the keystore:

```
Successfully created and initialized new KeyStore file: /opt/appdynamics/secretKeyStore
Verification - New KeyStore file: /opt/appdynamics/secretKeyStore is properly initialized.
```

Encrypt Passwords

To encrypt passwords using the secure credential store utility, run the `encrypt` command with the following parameters:

- `filename`: Absolute path to the secure credential keystore file.
- `storepass`: Password for the secure credential keystore. You can use either a plain-text password or a password that has been obfuscated as described in the following section.
- `plaintext`: Any plain text to encrypt. For instance, account access key or password.

Here is an example using a plain-text —passwordthe `storepass` argument—for the secure credential keystore:

```bash
java -jar ./scs-tool.jar encrypt -filename '/opt/appdynamics/secretKeyStore' -storepass 'MyCredentialStorePassword' -plaintext 'MyAccessKeyOrPassword'
```

Here is the same example using an obfuscated password:

```bash
java -jar ./scs-tool.jar encrypt -filename '/opt/appdynamics/secretKeyStore' -storepass 's_gsnwR6+iDch8JBf1RamiBoWfMvjipkrtJMZXAYEkw8=' -plaintext 'MyAccessKeyOrPassword'
```
The secure credential store utility writes out an encrypted password for use in agent configuration files:

```
r9iDWPzHRCNDM1B6KTag4A/cA5B4pouVPkv48ovRm6c=
```

**Obfuscate the Secure Credential Store Password**

In order to access the secret key in the secure credential keystore, the agent needs the obfuscated credential store password. Run the secure credential store utility `obfuscate` command with the following parameter:

- `plaintext`: The plain text secure credential keystore password.

For example:

```
java -jar /opt/appdynamics/scs-tool.jar obfuscate -plaintext 'MyCredentialStorePassword'
```

The secure credential store utility writes out an obfuscated password for use in the `<credential-store-password>` configuration. For example:

```
s_gsnwR6+LDch8JBf1RamiBoWfMvjipkrtJMZXAYEkw8=
```

**Encrypt a plain text property**

After you obfuscate the secure credential store password, you can encrypt plain text properties. For example, the following demonstrates how to encrypt properties in the analytics agent:

```
$ java -jar scs-tool.jar encrypt -filename /opt/appdynamics/secretKeyStore -storepass 'Welcome' -plaintext 'MyAccountAccessKey'
```

This generates an encrypted credential, such as:

```
-001-24-pFoSE/xdPcinkBj9iiKvpQ==Rznx8Kt3sPZHqKfYubVuhorrBEbYFTdTPm8c/1kFO+Z2eR2WEHtBRg4vy1GyvJ
```

**Sample Agent Configuration**

The following example demonstrates the agent configuration properties for the Secure Credential Store. For more information see the agent-specific configuration property documentation.

**Java Agent Configuration**
<controller-info>
  ...
  <!-- Encrypted account access key -->
  <account-access-key>r9iDWPzHRCNDM1B6KTag4A/cA5B4pouVPkv48ovRm6c=/account-access-key>

  <!-- Encrypted Controller keystore / agent trust store password -->
  <controller-keystore-password>Tw49bd0hdCMBoQ5pfMMuYA/cA5B4pouVPkv48ovRm6c=/controller-keystore-password>

  <!-- Enable the Secure Credential Store -->
  <use-encrypted-credentials>true</use-encrypted-credentials>

  <!-- Path to they secure credential keystore -->
  <credential-store-filename>/opt/appdynamics/secretKeyStore</credential-store-filename>

  <!-- Obfuscated secure credential keystore password -->
  <credential-store-password>n/8GvAZsKk4gM3Z6g+XQlw==</credential-store-password>

  ...
</controller-info>

Analytics Agent Configuration

```
ad.credential.store.filename=/opt/appdynamics/secretKeyStore
ad.credential.store.password=s_gsnwR6+LDch8JBfiRamiBoWfMvjipkrtJMJXAYEkw8=
```

Encrypt Data on the Analytics Agent

You can encrypt any data on the analytics agent using `secure://<your-encrypted-credentials>`. You can encrypt data in the Analytics Agent Properties file or System Properties. The following example demonstrates how to encrypt `http.event.accessKey` in the Analytics Agent Properties file.
http.event.accessKey=secure://-001-24-Dr9FQGC179o4vPnuljnxF88A==ZGVw/P4OONvpUidIhJ2u78FpRVVW8fbgr8J1HBHXwnE=
ad.secure.credential.store.filename=/opt/appdynamics/secretKeyStore
ad.secure.credential.store.password=s_gsnwR6+LDch8JBf1RamiBoWfMvjji
pkrtJMZXAYEkw8=
Manage App Agents

You can manage app agents connected to the Controller from the AppDynamics Agents page. On the page, you can view agent status and information, reset agents, disable agents, and more.

Permissions

To manage app agents, you must belong to a role with the Account level permission:

- Administer users, groups, roles, authentication, and so on
- View license
- View AppDynamics agents
- Use Agent Download Wizard

The predefined Account Owner role includes the required permission.

To access app agent administration settings, click gear ( ) icon > AppDynamics Agents > App Server Agents.

Associate App Agents with a Business Application

If you start an application server with an app agent without specifying the business application, the agent appears in the App Server Agents tab as an unassociated agent. You can associate an agent with an application manually:

1. From the agents list in the App Server Agents tab, select the unassociated app agent from the list.

   In the View menu, click Agents not associated with an Application to filter the list of App Server Agents.

2. Click Associate with Application. This button is enabled only for app agents that are connected to the Controller but not associated with a business application.

Reset App Agents

To reset app agents, users need the Agent Advanced Operation permission.

Resetting the app agent causes the agent to register itself with the Controller in the same way it does at JVM or application startup, but without restarting the JVM or application. Resetting the app agent purges in-memory data for the agent. It also applies certain configuration changes to the app agent, such as app agent node property changes.

Resetting an app agent causes the Controller to:

- Purge data such as in-memory business transactions and metrics, exit calls, and registration information for the agent
- Reset the business transaction limit counter to zero for the selected agent

Resetting an agent does not re-instrument or remove existing instrumentation. After a reset, the agent re-registers business transactions and backends and it creates new metrics. You may lose a few minutes worth of data between the reset and re-registration operations.
You may need to reset agents after you have reconfigured business transaction or backend detection and the agent is not applying the updated rules:

2. Choose a reset option:
   - To reset all the app agents for a business application, click Reset Agents.
   - To disable a individual app agents, select one or more agents and click Reset Selected App Agent(s).

Enable and Disable App Agents

Disabling an app agent stops the agent from reporting metrics to the Controller. Disabling an agent can help you diagnose agent installation or application issues. It also lets you compare the difference in overhead between when the agent when capturing data versus when it is not, for example, without removing the agent. Disabling the agent does not require you to shut down or restart the application server.

Disabling an app agent does not stop the agent from operating or remove the bytecode instrumentation added by the agent. To fully remove the agent including remove bytecode instrumentation, follow the steps to uninstall the app agent or fully disable agent instrumentation under the Install App Server Agents topic specific for the agent type. Uninstalling an app agent or fully disabling an app agent in this manner typically requires an application restart.

All agents connected to the Controller count against the agent license limits for that Controller. Even though it only reports minimal app server-related data, a disabled app agent is still connected to the Controller and consumes an app agent license.

To enable or disable all the app agents for a business application, the App Server Agents tab, click Enable Agents or Disable Agents.

You can enable or reenable individual agents by selecting them and clicking Enable Selected App Agent or Disable Selected App Agent. Alternatively, enable or disable app agents individually from the Agents tab on the Node Dashboard:

- To disable the agent, click Agent is Off.
- To enable the agent, click Agent is On.
  It takes about a minute for the operation to take effect.

By default turning the agent Off completely disables monitoring. For Java agents, un-check Disable all monitoring including JVM and JMX metrics on the Disable This App Agent window to keep collecting JVM and JMX metrics such as heap memory, memory pools, garbage collection, and thread count.

Delete App Agents

Deleting an app agent removes the agent and any associated data from the Controller database. This does not affect the instrumentation of the application server. If the application server is still running with an app agent, or if it is subsequently restarted with an app agent, the agent re-registers and appears again in the Controller.

You can delete an app agent by clicking Delete Agent from System.

To completely remove an app agent completely so that it does not register again, uninstall it. See the uninstall instructions for your agent platform under Install App Server Agents.
Agent Log Files

On this page:

- What is Logged
- Logging Levels
- Log Structure
- Agent-Specific Information

Related pages:

- Java Agent Logging
- .NET Agent for Windows Logging
- PHP Agent Logging
- Node.js Agent Logging
- Python Agent Debugging and Logging
- Apache Agent Logging
- Dynamic Agent Proxy Logging
- Business Transactions Logging
- Bytecode Transformer Logging
- REST Logging
- Analytics Agent Logging

Agent logs can help you resolve agent configuration and application instrumentation issues.

The Controller can generate and archive agent log files that you can submit to AppDynamics Support for troubleshooting assistance. See Request Agent Log Files for more information.

What is Logged

The beginning of the log file shows the startup of the agent monitoring services and the configuration settings. The agent log also contains the sequence of agent runtime activity and exceptions that are encountered. Much of this information is useful, especially for troubleshooting deployment issues.

Examples of information in the log:

- Agent version and build date
- JVM runtime version (Java only)
- Configuration changes
- Backend detection
- Exceptions
- Output from logging session requests

Logging Levels

The logging levels, listed in order from collecting the most information to the least, are the following:

- ALL: Logs all events.
- TRACE: Reports finer-grained informational events than the debug level that may be useful to debug an application.
- DEBUG: Reports fine-grained informational events that may be useful to debug an application.
- INFO: Default log level. Reports informational messages that highlight the progress of the application at coarse-grained level.
- WARN: Reports on potentially harmful situations.
- ERROR: Reports on error events that may allow the application to continue running.

Not all agents types log at all levels.

Log Structure

This section describes how the agent logs are rolled over for the following agents:
When the maximum file size is reached, a new log file is created. The first file is named agent.0.log, second file is agent.1.log, and so on. There is a maximum of five files per set and a maximum of five sets for each instrumented node. If the maximum number of sets is reached, when a new set is created, the oldest set is deleted.

While the agent rotates away old log files, it retains the initial log file. The first log file contains information that reflects the specific context in which the agent was started, along with other information that can be useful for troubleshooting and record keeping. On agent restart, a new set is created and the oldest set is deleted.

Each set includes not only the application agent log but also, depending on which logs exist, the ByteCodeTransformer log, the BusinessTransaction log and the REST log. A single set might consist of the following:

- agent.<timestamp>.#.log
- ByteCodeTransformer.<timestamp>.#.log
- REST.<timestamp>.#.log
- BusinessTransaction.<timestamp>.#.log

where # is the number of the set.

For example, the following logs in the logs directory were created on April 3 or the first set, set 0. The agent is a Java Agent; this is indicated by the prefix agent. Other app agents use different naming conventions. See the agent-specific information for details.

```plaintext
agent.2015_04_03__14_49_38.0.log
agent.2015_04_03__14_49_46.0.log
agent.2015_04_03__14_51_04.0.log
BusinessTransactions.2015_04_03__14_49_38.0.log
BusinessTransactions.2015_04_03__14_49_46.0.log
BusinessTransactions.2015_04_03__14_51_04.0.log
ByteCodeTransformer.2015_04_03__14_49_40.0.log
ByteCodeTransformer.2015_04_03__14_49_47.0.log
ByteCodeTransformer.2015_04_03__14_51_04.0.log
REST.2015_04_03__14_49_38.0.log
REST.2015_04_03__14_49_46.0.log
REST.2015_04_03__14_51_04.0.log
```

On the next agent restart a new set is created. The logs in this set, set 1, will have names such as:

```plaintext
agent.2015_04_03__14_53_05.1.log
...
```

and the next set, set 2,

```plaintext
agent.2015_04_04__15_12_06.2.log
...
```

and so on through the five potential sets.
Agent-Specific Information

AppDynamics app server agents have different logging locations and different maximum log sizes. See the related pages for logging documentation for details about the agent you are interested in.
Bytecode Transformer Logging

Related pages:
- Agent Log Files
- Java Agent Logging
- .NET Agent for Windows Logging
- Dynamics Agent Proxy Logging

The Bytecode Transformer log is generated by the Java Agent, the .NET Agent and the proxy. It contains the information associated with the AppDynamics platform bytecode instrumentation (BCI) engine. The BCI engine is used to inject interceptors. The BCI engine logs the classes being loaded in the application.

This log is useful for troubleshooting issues that depend on bytecode instrumentation including entry points, exit points, missing metrics, loggers, errors, exceptions, and asynchronous threads.

This log includes the following:
- agent version and build date
- each Java class that was examined and whether the methods were considered for bytecode instrumentation
- each method that was instrumented.

For the Java Agent and the proxy, the log is named ByteCodeTransformer Year_mon_day_hr_min.#.log, where # is the logging set number.

For the .NET Agent, the log is named ByteCode.txt.

See Agent Log Files for details about how the sets are organized that roll over.
REST Logging

The REST log contains information about the AppDynamics agent and Controller communication. The log contains request and response payload data between the agent and the Controller, and the raw details of communications between the agent and the Controller. By default, the REST log contains INFO level logging for all registration requests up to a limit of one MB every five minutes. You can use this log to troubleshoot issues with detecting and reporting business transactions, backends, events, and metric reporting.

The communication between the agent and controller consists of the following:

- Configuration data
- Node identification
- Machine registration
- Business transaction registration
- Metric registration
- Event registration
- Snapshot capture/upload to controller database
- Metric data upload to controller database
- Event data upload to the Controller database
- Output from Logging Session requests

The naming convention for the REST log is REST.<timestamp>.#.log where # is the set.

See Agent Log Files for information about the structure of the log files into sets that roll over.

Within a set, a REST log file can reach a maximum of 5MB.
Metrics Limits

On this page:
- Agent Metric Limits
- Controller Metric Limits

To ensure that you see the metric information that is most relevant to your application, agents have limits for the number of metrics they can store. Different limits apply to app agents and machine agents, as follows:

- For an app agent, the default maximum number of registered metrics is 5000.
- For the Standalone Machine Agent, the default maximum number of registered metrics is 450. This includes both basic hardware metrics and the expanded metrics available with Server Visibility.

If the limit is reached, an error event is generated of type `AGENT_METRIC_REG_LIMIT_REACHED` with a summary of *Metric registration limit of \( n \) reached*. No new metrics are created until the agent restarts. You can increase the default limit if necessary, as described next.

Agent Metric Limits

You can increase or decrease the default metric registration limits for machine agents or app agents.

Changing these limits can affect the resource consumption of your deployment. Before you change this setting verify your application environment and Controller can handle the increased resource requirements.

For the Java agent, modify the limit using the `agent.maxMetrics` system property. For example, to increase the machine agent metric limit, specify the maximum number of metrics as an argument when starting the machine agent in the following format:

```
-Dappdynamics.agent.maxMetrics=<max-number-of-metrics>
```

For example, when starting the machine agent, increase the maximum number of metrics that can be registered to 2000 as follows:

```
nohup java -Dappdynamics.agent.maxMetrics=2000 -jar machineagent.jar &
```

For the .NET Agent, set the `maxMetrics` property as an environment variable. This setting only affects the app agent. For example:

```
appdynamics.agent.maxMetrics=5500
```

For the .NET Machine Agent, specify the maximum number of metrics using the Metrics element in the config.xml. See 'Machine Agent Element' on .NET Agent Configuration Properties. See also Administer the .NET Agent.

```
<metrics max-metrics="300"/>
```

Controller Metric Limits

The Controller applies its own limits on metric registration. For an on-premises Controller, you can view and modify the properties that
control the metric registration limit in the Controller settings page in the Controller Administration Console.

The relevant Controller settings are:

- `metric.registration.limit`: The maximum number of metrics that can be registered for an account in the Controller. The default is 2 million.
- `application.metric.registration.limit`: The maximum number of metrics that can be registered for a business application in the Controller. The default is 1 million.
Historical and Disconnected Nodes

On this page:
- Node Activity and Agent Licensing
- Configuring Node Activity Settings
- Agent Behavior WhenDisconnected from the Controller

Related pages:
- Access the Administration Console

If a node has been out of contact with the Controller for a certain amount of time, the Controller marks the node as a historical node. The Controller suspends certain types of processing activities for the node, such as rule evaluation.

If the node resumes contact with the Controller before the node deletion period expires, the Controller restores it to an active state. Otherwise, it is permanently removed from the Controller and the node level data is no longer accessible in the UI. Tier and application level historical metric data for the node remain available however.

By default, the Controller considers a node historical after about 20 days of inactivity and deletes the node after 30 days. For a highly dynamic application environment in which nodes are created and destroyed frequently, it usually makes sense to shorten the node activity timeout period. This allows recycled nodes to be treated as such in the Controller.

The node activity timeout period is determined by the node retention period or activity settings.

The names of historical nodes can be assigned to new nodes. Node name reuse is a Java Agent option that, when enabled, directs the Controller to reuse node names, so that data generated by multiple, short-lived nodes in a given tier is associated with a single logical node.

Node Activity and Agent Licensing

For licensing purposes, the Controller releases the license for the agent if the Controller has not received data from the agent in the previous 5 minutes. This license availability behavior is not affected by the historical node status or node deletion timeout settings.

Configuring Node Activity Settings

The node activity settings are account level settings that the root AppDynamics administrator can modify from the administration console:

- `node.permanent.deletion.period`: Time (in hours) after which a node that has lost contact with the Controller is deleted permanently from the system. The data is removed. If the agent starts reporting again after this period, it will start like a new node. Therefore, no historical data will be available at the node level. You will see historical data at the tier and app level, and cluster roll up will take place as normal.
  
  The default is 720 hours, the minimum value is 6 hours, and the maximum value for this setting is unlimited.

- `node.retention.period`: Time (in hours) after which a node that has lost contact with the Controller is deleted. In this case, the AppDynamics UI will not display the node, however, the system will continue to retain it. If the agent starts reporting again within these hours, it will reappear in the UI and the counter will reset. The data is persisted.

  The default is 500 hours, the minimum value is 1 hour, and the maximum value for this setting is unlimited.

  Additional notes about the node retention period:
  - A node will not be impacted by the node retention period if the Machine Agent is associated with that node.
  - If you need a node to be considered for the node retention period, it should be marked as historical on shutdown:

```plaintext
Dappdynamics.jvm.shutdown.mark.node.as.historical=true
```

Agent Behavior WhenDisconnected from the Controller

The Controller may become unreachable when there are network problems, agent errors or when the Controller server is down for a variety of reasons.
If the Controller is unreachable for one minute:

- The agent goes into standby mode during which it does not detect any transactions.
- Any collected snapshots and events are dropped and lost. Snapshots and events are dropped because they consume too much memory to cache.
- All metrics that have not been posted to the Controller are stored in memory. The memory impact of retaining metrics is minimal.
- New business transaction registrations that have not been posted to the Controller are stored in memory.
- The agent attempts to connect to the Controller every minute and resumes normal activity when it can download its full configuration.

If the Controller becomes reachable in the following minute or two:

- All metrics that have been stored in memory are posted to the Controller.
- New business transaction registrations that have been stored in memory are posted to the Controller.
- Snapshots and events collected in the 20 seconds prior to the reconnection are posted to the Controller.

If the Controller is not reachable after three failed attempts that are one minute apart:

- The agent is muted and all business transaction interceptors are disabled. The interceptors are still called when monitored application entry point methods are executed, but they are unproductive. No new business transactions are discovered or registered. Correlation exit points will set a header such as “notxdetect=true”, which tells downstream tiers to also ignore the transaction.
- JMX metrics are stored in the application server memory and transmitted to Controller after reconnection; so, there are no gaps in the metric history.
- Periodic metrics for the last three minutes are stored in memory. Metrics older than three minutes are purged from memory.
- The agent configuration channel and the metric channel continue to attempt to connect to the Controller once each minute.

If the Controller is not reachable after five minutes, the license is freed for another agent to use.

If the connection is later successful and the agent is able to download its full configuration and a license:

- All periodic metrics, such as JMX metrics and Windows performance counters for the last three minutes, are posted to the Controller. The Controller drops metrics that were collected too long ago in the past, such as when rollups are already completed.
- The agent is reactivated, business transaction interceptors are re-enabled, business transactions are monitored and possibly snapshotted, new business transactions will be discovered and registered, and downstream correlation is re-enabled.
Request Agent Log Files

On this page:
- Permissions
- Get Agent Log Files and Thread Dump Samples
- Start an Agent Logging Session
- Submitting Log Files to Support

Related pages:
- Agent Log Files

Agent logging information can help you troubleshoot issues that are local to an AppDynamics app agent.

Note: The Controller cannot access the log files generated by the .NET Microservices Agent.

This topic describes how to work with agent logs.

To access agent logging operations in the Controller UI, navigate to the following location: Node Dashboard > Agents tab > App Server Agent tab > Agent Operations section.

Permissions

To request agent thread dumps or agent debug logs, users need the Agent Advanced Operation permission.

Get Agent Log Files and Thread Dump Samples

Requesting log files causes the agent to send the contents of its logs directory as a zip file to the Controller. From there, you can download the zip to your local machine.

The following agent-specific considerations apply to logging sessions:

- For the Java Agent or PHP Agent, you can get all logs or choose from these alternatives:
  - Output from a specific logger at a set level, for a fixed duration. For a list of supported loggers, see Java Supported Environments.
  - Thread dump samples. The agent takes a thread dump according to the specified collection interval and specified number to collect. After the interval elapses the thread dumps are uploaded by the agent as a zip file.
- For the Node.js agent, the proxy logs are available.
- For the Python Agent, if you choose All logs in the logs directory, you get both the proxy logs and the agent logs. The agent log contains the log messages that it is configured to log in the agent configuration file. See Python Agent Settings.

While the status is PENDING, you can click Refresh to update the status field. Once the request succeeds, as indicated by the status, right-click the agent operation name for your request and choose Download Data. Enter a name for your request to prepend the generated zip file with your name. For example, a name of CurrentTime produces a zip file named similar to CurrentTime_Node_800_1390503141046.zip.

If SUCCESSFUL doesn't eventually appear in the status for the request, the request may have timed out. A request can timeout due to incorrect request parameters or if the server becomes unavailable. In this case, right-click the request and choose Delete Data and retry the request.

Start an Agent Logging Session

For Java, you can generate log information with the specific type of logging information. For the Node.js and Python agents, the log information generated is from the proxy only.

1. Confirm that you have enough disk space to create the logs before starting a logging session. Log file requests fail if there is not enough disk space available. The disk space required depends on application activity. If you are uncertain about the amount of space required, try conducting an agent logging session with the shortest duration and closely monitor the disk space on the monitored machine.
2. Click **Start Agent Logging Session** in the Agent Operations panel.
3. Choose the duration for the logging session and the debug logger types to initiate. The logger type determines the type of information captured with these options:

<table>
<thead>
<tr>
<th>Logging Session Type</th>
<th>Type of Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application Wide Configuration</td>
<td>Application configuration changes</td>
</tr>
<tr>
<td>Business Transaction Registration</td>
<td>Transaction, exit identification, self-resolution</td>
</tr>
<tr>
<td>Current Time</td>
<td>Requests to sync up the clock skew between the controller and the agent</td>
</tr>
<tr>
<td>Events</td>
<td>Changes in application state that are of potential interest</td>
</tr>
<tr>
<td>Metric Data</td>
<td>Metric data</td>
</tr>
<tr>
<td>Metric Registration</td>
<td>Metric registration</td>
</tr>
<tr>
<td>One Way Agent</td>
<td>For system (machine) agent only, information from the capture feature</td>
</tr>
<tr>
<td>Request Segment data</td>
<td>Snapshot upload of request segment</td>
</tr>
<tr>
<td>System Agent Registration</td>
<td>System (machine) agent registration</td>
</tr>
<tr>
<td>System Agent Reregistration</td>
<td>System (machine) agent reregistration</td>
</tr>
<tr>
<td>System Agent Polling Handler</td>
<td>System (machine) agent configuration request</td>
</tr>
<tr>
<td>Task Execution</td>
<td>Relates to the machine agent where you can create a task to execute on a machine, such as to restart the application server and run a script</td>
</tr>
<tr>
<td>Top Summary Stats</td>
<td>Request to upload the 'most expensive backend' summary</td>
</tr>
<tr>
<td>Application Configuration</td>
<td>Configuration change requests</td>
</tr>
<tr>
<td>Transient Channel</td>
<td>The activity of transient channel that is used to upload and download instructions for JMX console live data</td>
</tr>
</tbody>
</table>

4. Click **Start Agent Logging Session**
   Each selected session type appears in the session panel with a start time and end time based on the selected Duration of logging session.

**Submitting Log Files to Support**

When submitting log files to AppDynamics Support, ensure you send the entire contents of the agent logs directory available as a ZIP file for download when you request the agent logs.
App Agent Node Properties

On this page:

- About Node Properties
- Permissions
- Edit a Registered Node Property
- Add a Registered Node Property

Works with:

You can register app agent node properties to customize app agent behavior at the application, tier or node level.

About Node Properties

App agent node properties control the features and preferences for the Java Agent and .NET Agent. Such agent-specific settings include limits on the number of business transactions, the minimum number of requests to evaluate before triggering a diagnostic session, and so on.

Node properties follow an inheritance model similar to instrumentation detection, so you can set an individual property globally for an application or at the tier or node levels.

Even though it is possible to configure node properties in the app-agent-config.xml file in the agent home directory, AppDynamics recommends that you use the Controller UI to configure node properties. The Controller UI displays only those node properties that are registered to the agent.

The App Agent Node Properties reference includes additional properties that do not appear in the UI by default. You can register these properties yourself, but unregistered properties are intended for specific application or troubleshooting scenarios and can impact the performance of your deployment. You should register properties or configure properties directly in app-agent-config.xml only under the guidance of AppDynamics Support or as specifically instructed by the documentation.

Permissions

Users need the Configure Agent Properties permission to create, edit, or delete agent configuration.

Edit a Registered Node Property

In the Controller UI, you can access node properties for a particular node or for all nodes from the dashboard of any node in the application, as follows:

1. Access the node dashboard by going to the App Servers page. See Tiers and Nodes
2. Expand the tier that contains the node on which you want to configure a node property and double-click the node.
3. In the node dashboard, click Actions > Configure App Server Agent.
4. Select the Use Custom Configuration button, and then find and double click on the property you want to modify. See Hierarchical Configuration Inheritance for more information on how node settings work.

After customizing a configuration, you can copy the configuration to other nodes, to the tier, or apply it to the entire application.
Add a Registered Node Property

You can register and configure unregistered App Agent Node Properties as instructed by AppDynamics Support or as documented.

To register a node property, create a custom configuration for the node, as described in Edit Registered Node Property. Add properties by clicking the + plus icon at the top of the list of current node properties.

In the Create Agent Property window, use the values from App Agent Node Properties to provide values for the name, description, type, and value of the property.
App Agent Node Properties Reference

This reference describes the app agent node properties by type to help you decide which properties to use. In addition, the reference includes the following pages containing information about the app agent node properties:

- App Agent Node Properties (A)
- App Agent Node Properties (B-C)
- App Agent Node Properties (D-E)
- App Agent Node Properties (F-I)
- App Agent Node Properties (J-L)
- App Agent Node Properties (M)
- App Agent Node Properties (N-R)
- App Agent Node Properties (S)
- App Agent Node Properties (T-Z)

In general use caution when modifying the agent default settings. If increasing limits specified for an agent, you need to carefully assess and monitor memory consumption by the agent after the change.

App Agent Node Properties by Type

The following table groups the app agent node properties by type, so that you can browse the properties by functionality and feature area.

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automatic Leak Detection</td>
<td>minimum-age-for-evaluation-in-minutes</td>
</tr>
<tr>
<td></td>
<td>minimum-number-of-elements-in-collection-to-deep-size</td>
</tr>
<tr>
<td></td>
<td>minimum-size-for-evaluation-in-mb</td>
</tr>
<tr>
<td>Bytecode injection (BCI)</td>
<td>bci-log-config</td>
</tr>
<tr>
<td></td>
<td>bciengine-disable-retransformation</td>
</tr>
<tr>
<td></td>
<td>enable-interceptors-for-security</td>
</tr>
<tr>
<td></td>
<td>enable-json-bci-rules</td>
</tr>
<tr>
<td></td>
<td>enable-xml-bci-rules</td>
</tr>
<tr>
<td>Backend Detection</td>
<td>enable-kafka-consumer</td>
</tr>
<tr>
<td></td>
<td>msmq-correlation-field</td>
</tr>
<tr>
<td></td>
<td>msmq-single-threaded</td>
</tr>
<tr>
<td></td>
<td>nservicebus-single-threaded</td>
</tr>
<tr>
<td>Object Instance Tracking (OIT)</td>
<td>collection-capture-period-in-minutes</td>
</tr>
<tr>
<td></td>
<td>enable-instance-monitoring</td>
</tr>
<tr>
<td></td>
<td>enable-object-size-monitoring</td>
</tr>
<tr>
<td></td>
<td>leak-diagnostic-interval-in-minutes</td>
</tr>
<tr>
<td>JMX Property</td>
<td>heap-storage-monitor-devmode-disable-trigger-pct</td>
</tr>
<tr>
<td>Transaction Monitoring</td>
<td>ado-new-resolvers</td>
</tr>
<tr>
<td></td>
<td>aspdotnet-core-naming-controllerarea</td>
</tr>
<tr>
<td></td>
<td>aspdotnet-mvc-naming-controlleraction</td>
</tr>
<tr>
<td><strong>api-thread-activity-timeout-in-seconds</strong></td>
<td></td>
</tr>
<tr>
<td><strong>api-transaction-timeout-in-seconds</strong></td>
<td></td>
</tr>
<tr>
<td><strong>async-tracking</strong></td>
<td></td>
</tr>
<tr>
<td><strong>async-transaction-demarcator</strong></td>
<td></td>
</tr>
<tr>
<td><strong>capture-404-urls</strong></td>
<td></td>
</tr>
<tr>
<td><strong>capture-error-urls</strong></td>
<td></td>
</tr>
<tr>
<td><strong>capture-raw-sql</strong></td>
<td></td>
</tr>
<tr>
<td><strong>capture-set-status</strong></td>
<td></td>
</tr>
<tr>
<td><strong>capture-spring-bean-names</strong></td>
<td></td>
</tr>
<tr>
<td><strong>disable-custom-exit-points-for</strong></td>
<td></td>
</tr>
<tr>
<td><strong>disable-exit-call-correlation-for</strong></td>
<td></td>
</tr>
<tr>
<td><strong>disable-exit-call-metrics-for</strong></td>
<td></td>
</tr>
<tr>
<td><strong>disable-percentile-metrics</strong></td>
<td></td>
</tr>
<tr>
<td><strong>downstream-tx-detection-enabled</strong></td>
<td></td>
</tr>
<tr>
<td><strong>enable-all-rsd-error-propagation</strong></td>
<td></td>
</tr>
<tr>
<td><strong>enable-default-http-error-code-reporter</strong></td>
<td></td>
</tr>
<tr>
<td><strong>enable-soap-header-correlation</strong></td>
<td></td>
</tr>
<tr>
<td><strong>enable-spring-integration-entry-points</strong></td>
<td></td>
</tr>
<tr>
<td><strong>enable-transaction-correlation</strong></td>
<td></td>
</tr>
<tr>
<td><strong>end-to-end-message-latency-threshold-millis</strong></td>
<td></td>
</tr>
<tr>
<td><strong>find-entry-points</strong></td>
<td></td>
</tr>
<tr>
<td><strong>jdbc-callable-statements</strong></td>
<td></td>
</tr>
<tr>
<td><strong>jdbc-connections</strong></td>
<td></td>
</tr>
<tr>
<td><strong>jdbc-dbcam-integration-enabled</strong></td>
<td></td>
</tr>
<tr>
<td><strong>jdbc-prepared-statements</strong></td>
<td></td>
</tr>
<tr>
<td><strong>jdbc-resultsets</strong></td>
<td></td>
</tr>
<tr>
<td><strong>jdbc-statements</strong></td>
<td></td>
</tr>
<tr>
<td><strong>jmx-appserver-mbean-finder-delay-in-seconds</strong></td>
<td></td>
</tr>
<tr>
<td><strong>jmx-operation-timeout-in-milliseconds</strong></td>
<td></td>
</tr>
<tr>
<td><strong>jmx-rediscover-mbean-servers</strong></td>
<td></td>
</tr>
<tr>
<td><strong>jrmp-enable</strong></td>
<td></td>
</tr>
<tr>
<td><strong>log-request-payload</strong></td>
<td></td>
</tr>
<tr>
<td><strong>max-async-task-registration-requests-allowed</strong></td>
<td></td>
</tr>
<tr>
<td><strong>max-async-task-registrations-allowed</strong></td>
<td></td>
</tr>
<tr>
<td><strong>max-business-transactions</strong></td>
<td></td>
</tr>
<tr>
<td>max-service-end-points-per-async-type</td>
<td></td>
</tr>
<tr>
<td>max-service-end-points-per-entry-point-type</td>
<td></td>
</tr>
<tr>
<td>max-service-end-points-per-node</td>
<td></td>
</tr>
<tr>
<td>max-service-end-points-per-thread</td>
<td></td>
</tr>
<tr>
<td>max-urls-per-error-code</td>
<td></td>
</tr>
<tr>
<td>min-load-per-minute-diagnostic-session-trigger</td>
<td></td>
</tr>
<tr>
<td>rmqsegments</td>
<td></td>
</tr>
<tr>
<td>percentile-method-option</td>
<td></td>
</tr>
<tr>
<td>rest-num-segments</td>
<td></td>
</tr>
<tr>
<td>rest-transaction-naming</td>
<td></td>
</tr>
<tr>
<td>rest-uri-segment-scheme</td>
<td></td>
</tr>
<tr>
<td>slow-request-deviation</td>
<td></td>
</tr>
<tr>
<td>slow-request-monitor-interval</td>
<td></td>
</tr>
<tr>
<td>socket-collection-bci-enable</td>
<td></td>
</tr>
<tr>
<td>spring-batch-enabled</td>
<td></td>
</tr>
<tr>
<td>spring-integration-receive-marker-classes</td>
<td></td>
</tr>
<tr>
<td>spring-mvc-naming-scheme</td>
<td></td>
</tr>
<tr>
<td>thread-correlation-classes</td>
<td></td>
</tr>
<tr>
<td>thread-correlation-classes-exclude</td>
<td></td>
</tr>
<tr>
<td>max-service-end-points-per-node</td>
<td></td>
</tr>
<tr>
<td>max-service-end-points-per-thread</td>
<td></td>
</tr>
<tr>
<td>max-urls-per-error-code</td>
<td></td>
</tr>
<tr>
<td>websocket-entry-calls-enabled</td>
<td></td>
</tr>
</tbody>
</table>

**Transaction Snapshots**

<p>| adaptive-callgraph-granularity                       |
| callgraph-granularity-in-ms                          |
| dev-mode-suspend-cpm                                 |
| dont-show-packages                                    |
| enable-startup-snapshot-policy                       |
| max-call-elements-per-snapshot                       |
| max-concurrent-snapshots                             |
| max-error-snapshots-per-minute                       |
| max-jdbc-calls-per-callgraph                         |
| max-jdbc-calls-per-snapshot                          |
| min-duration-for-jdbc-call-in-ms                     |
| on-demand-snapshots                                  |</p>
<table>
<thead>
<tr>
<th>Category</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>show-packages</td>
</tr>
<tr>
<td></td>
<td>slow-request-threshold</td>
</tr>
<tr>
<td></td>
<td>thread-cpu-capture-overhead-threshold-in-ms</td>
</tr>
<tr>
<td>Miscellaneous</td>
<td>collect-user-data-sync</td>
</tr>
</tbody>
</table>
App Agent Node Properties (A)

On this page:
- adaptive-callgraph-granularity
- ado-new-resolvers
- always-add-eum-metadata-in-http-headers
- analytics-sql-cpm-limit
- api-thread-activity-timeout-in-seconds
- api-transaction-timeout-in-seconds
- apply-reactive-rules
- apply-additional-reactive-rules
- aspnetcore-naming-controllerarea
- aspnet-mvc-naming-controlleraction
- aspnet-mvc-naming-controllerarea
- async-tracking
- async-transaction-demarcator

This reference page contains information about app agent node properties. The properties are listed in alphabetical order.

### adaptive-callgraph-granularity

This property enables adaptive snapshots. The call graph granularity for adaptive snapshots is based on the average response time for the business transaction during the last one minute and is thus adaptive. The following distribution is used:

- Granularity of 10 ms for average response time of <= 10 seconds
- 50 ms for 10 to 60 seconds
- 100 ms for 60 to 600 seconds
- 200 ms for > 600 seconds

<table>
<thead>
<tr>
<th>Type:</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>false</td>
</tr>
<tr>
<td>Platform(s):</td>
<td>Java, .NET</td>
</tr>
</tbody>
</table>

### ado-new-resolvers

Enable database detection and naming for ODP.NET backends labeled Unknown.

<table>
<thead>
<tr>
<th>Type:</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>true</td>
</tr>
<tr>
<td>Platform(s):</td>
<td>.NET</td>
</tr>
</tbody>
</table>
## always-add-eum-metadata-in-http-headers

By default, the Java Agent, .NET Agent, and Node.js Agent set business transaction correlation data in a cookie for HTTP responses, except when the JavaScript Agent has already set an `isAjax: true` header in the request. When it finds the `isAjax: true` header, the agent sets the correlation metadata in the XHR header.

For cross-origin AJAX requests, the JavaScript Agent does not set `isAjax: true` so that the app agent doesn't write correlation data to the header of those responses.

Set `always-add-eum-metadata-in-http-headers` to `true` to configure the app agent to write business transaction metadata to the XHR header and in a cookie even if the request is considered cross-origin.

<table>
<thead>
<tr>
<th>Type</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>false</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java, .NET, Node.js</td>
</tr>
</tbody>
</table>

## analytics-sql-cpm-limit

This property specifies the per minute upper limit on the number of SQL queries that collect parameter data for analytics. The number is a cumulative total. It is not the number of distinct SQL queries, but the overall number of invocations of the SQL queries that have been configured to collect analytics data.

<table>
<thead>
<tr>
<th>Type</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>10000</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java, .NET</td>
</tr>
</tbody>
</table>

## api-thread-activity-timeout-in-seconds

This property provides a time-out value that comes into play when you have added global transactions to your application using APIs from the AppDynamics SDK. In the event that the added transaction spawns additional threads that do not return or complete, this property provides a safety valve time-out value. The value is in seconds. The `removeCurrentThread` method is invoked after the specified timeout period.

<table>
<thead>
<tr>
<th>Type</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>300 (seconds)</td>
</tr>
<tr>
<td>Range</td>
<td>Minimum =1; Maximum=3600</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java</td>
</tr>
</tbody>
</table>
### api-transaction-timeout-in-seconds

This property provides a time-out value that comes into play when you have added global transactions to your application using APIs from the AppDynamics SDK. In the event that the added transaction does not return or complete, this property provides a safety valve time-out value. The time-out value is in seconds. The `endTransaction` method is invoked after the specified time-out period.

<table>
<thead>
<tr>
<th>Type</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>300 (seconds)</td>
</tr>
<tr>
<td>Range:</td>
<td>Minimum =1; Maximum=3600</td>
</tr>
<tr>
<td>Platform(s):</td>
<td>Java</td>
</tr>
</tbody>
</table>

### apply-reactive-rules

This property provides a switch to flip the entire reactor instrumentation. If you set it to false, all rules pertaining to reactor thread correlation are unapplied.

<table>
<thead>
<tr>
<th>Type</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>true</td>
</tr>
<tr>
<td>Platform(s):</td>
<td>Java</td>
</tr>
</tbody>
</table>

### apply-additional-reactive-rules

This property provides a switch to apply in process correlation rules for prospective thread hand-offs. By default these rules are disabled. These rules can only be applied if `apply-reactive-rules` node property is set to true. If `apply-reactive-rules` property is set to false, setting this property to `true` will not apply any rules.

<table>
<thead>
<tr>
<th>Type</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>False</td>
</tr>
<tr>
<td>Platform(s):</td>
<td>Java</td>
</tr>
</tbody>
</table>

### aspdotnet-core-naming-controllerarea

If true, causes the agent to identify ASP.NET Core on the full framework business transactions as Area/Controller/Action. See Name MVC Transactions by Area, Controller, and Action.

<table>
<thead>
<tr>
<th>Type</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>true</td>
</tr>
<tr>
<td>Platform(s):</td>
<td>.NET</td>
</tr>
</tbody>
</table>
### aspdotnet-mvc-naming-controlleraction
If true, causes the agent to identify ASP.NET MVC business transactions as Controller/Action. See Name MVC Transactions by Area, Controller, and Action.

<table>
<thead>
<tr>
<th>Type</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>false</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>.NET</td>
</tr>
</tbody>
</table>

### aspdotnet-mvc-naming-controllerarea
If true, causes the agent to identify ASP.NET MVC business transactions as Area/Controller. See Name MVC Transactions by Area, Controller, and Action.

<table>
<thead>
<tr>
<th>Type</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>false</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>.NET</td>
</tr>
</tbody>
</table>

### async-tracking
Enable or disable detection of asynchronous exit points. See Asynchronous Exit Points for .NET.

<table>
<thead>
<tr>
<th>Type</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>true</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>.NET</td>
</tr>
</tbody>
</table>

### async-transaction-demarcator
This class name and method name combination marks the end of an asynchronous distributed transaction. Use the format ClassName/MethodName. For example, foo/bar where foo is the class name and bar is the method name.

<table>
<thead>
<tr>
<th>Type</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>none</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java</td>
</tr>
</tbody>
</table>
## App Agent Node Properties (B-C)

On this page:
- `bci-log-config`
- `bciengine-disable-retransformation`
- `boot-amx`
- `callgraph-granularity-in-ms`
- `capture-404-urls`
- `capture-error-urls`
- `capture-raw-sql`
- `capture-set-status`
- `capture-spring-bean-names`
- `check-bt-excludes-early`
- `collect-user-data-sync`
- `collection-capture-period-in-minutes`

This reference page contains information about app agent node properties. The properties are listed in alphabetical order.

### bci-log-config

Use this property to configure the bytecode transformer log (BCT log). This log shows what AppDynamics instruments and what classes are loaded in the JVM. The initial file (the `.0.log`) is saved to preserve the context of the server startup and is not rolled over. The subsequent files rotate. The format of the file name is `ByteCodeTransformer.<timestamp>.<N>.log`. The time stamp is represented as `YYYY_MM_DD_HH_mm_ss.N` and increments starting from zero.

<table>
<thead>
<tr>
<th>Type</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>20,5,4</td>
</tr>
<tr>
<td>Platform(s):</td>
<td>Java</td>
</tr>
</tbody>
</table>

**Examples**

```
ByteCodeTransformer.2012_09_12_20_17_57.0.log
```

Because the file size is checked every 15 seconds, the files may be a bit larger than the specified threshold value before they are rolled over.

The first log file generated is named as follows: `ByteCodeTransformer.<timestamp>.0.log`

The format for this property value is illustrated by the default value, 20,5,4. The numbered segments have the following meaning:

- 20 is the size, in MB, of the first log file, the .0 version
- 5 is the size in MB for each subsequent rolling file
- 4 is the number of ByteCodeTransformer log files to keep
### bciengine-disable-retransformation

Disable or enable bytecode retransformation. By default, the Sun and JRockit variant of the Java Agent applies configuration changes requiring bytecode retransformation, such as new POJO rules, without a restart to the JVM. Set `bciengine-disable-retransformation` to `true` to prevent the agent from performing automatic retransformation to apply such rules.

<table>
<thead>
<tr>
<th>Type</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>false</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java</td>
</tr>
</tbody>
</table>

### boot-amx

If set to true, enables support for Glassfish AMX MBeans. See GlassFish Startup Settings for more information about Glassfish server support.

<table>
<thead>
<tr>
<th>Type</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>false</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java</td>
</tr>
</tbody>
</table>

### callgraph-granularity-in-ms

Specifies the granularity for call graphs for this node. The global configuration is ignored if this property is used. This value is ignored if the adaptive-callgraph-granularity property is set to true. A default value of zero means the global configuration, from Configuration > Instrumentation > Call Graph Settings is used. Does not need a restart.

<table>
<thead>
<tr>
<th>Type</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>0</td>
</tr>
<tr>
<td>Range:</td>
<td>Minimum=0; Maximum=5000</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java, .NET</td>
</tr>
</tbody>
</table>
### capture-404-urls

This property disables or enables the capture of the URLs causing 404 errors. The URLs are reported as ERROR events every 15 minutes and are viewable through the Event Viewer. The JVM needs a restart if retransformation is not supported for the JVM version.

404 errors usually mean that no application code is executed, resulting in nothing to be captured in a snapshot. You can get insight into the 404 error by setting this property to true. It reports all the URLs which caused 404 error.

The `capture-404-urls` node property is deprecated in AppDynamics v. 3.6 and replaced with `capture-error-urls`.

<table>
<thead>
<tr>
<th>Type</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>false</td>
</tr>
<tr>
<td>Platform(s):</td>
<td>Java</td>
</tr>
</tbody>
</table>

### capture-error-urls

This property enables or disables the capture of the following HTTP errors:

- 401 - Unauthorized
- 500 - Internal Server Error
- 404 - Page Not Found
- All other error codes are put in a generic HTTP error code bucket.

For these four categories, the agent collects URLs, limited to 25 per category per minute, and sends an event out every 5 minutes.

You can see these URLs when you drill down on an error code by clicking Troubleshoot > Errors > Exceptions tab > HTTP Error Codes.

<table>
<thead>
<tr>
<th>Type</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>true</td>
</tr>
<tr>
<td>Platform(s):</td>
<td>Java</td>
</tr>
</tbody>
</table>
**capture-raw-sql**

If `capture-raw-sql` is enabled, SQL calls with dynamic parameters—such as question mark parameters—are captured by the agent and shown in the Controller UI with the dynamic parameters bound to their runtime values.

<table>
<thead>
<tr>
<th>Type</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>false</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java, .NET</td>
</tr>
</tbody>
</table>

**Examples**

For example, consider Java code that constructs a SQL call as follows:

```java
stmt = new PreparedStatement("select * from user where ssn = ?")
stmt.bind(1, "123-123-1234")
stmt.execute()
```

With `capture-raw-sql` enabled, AppDynamics captures the SQL call in the following form:

```sql
select * from user where ssn = '123-123-1234'
```

If `capture-raw-sql` is disabled, the SQL call appears with question mark parameters not bound to values.

Disabling `capture-raw-sql` and using question mark parameters in SQL prepared statements gives you a mechanism for preventing sensitive data from appearing in the Controller UI.

It is important to note that the sensitive values must be parameterized in the original, prepared statement form of the SQL statement, as shown above. The following statement would result in the potentially sensitive information—the ssn value—appearing in the UI whether `capture-raw-sql` is enabled or disabled.

```java
stmt = new PreparedStatement("select * from user where ssn = '123-123-1234'")
```

If you change this node property in an environment that is using the IBM JVM, you need to restart the JVM. This is because the feature requires retransformation of certain JDBC classes, which is not possible with the IBM agent.

Setting the option as an agent property affects the SQL capture mode for the node. You can configure the behavior for all nodes using the Capture Raw SQL option described in Call Graph Settings.

**capture-set-status**

Directs the agent to capture errors where the webservice is using `setStatus()` to send back an error. By default, only `sendError()` is instrumented by the agent.

<table>
<thead>
<tr>
<th>Type</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>false</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java</td>
</tr>
</tbody>
</table>
### capture-spring-bean-names

When a class is mapped to multiple Spring bean names, by default only the name of the first Spring bean found displays. This can be misleading. For example, when you see a call graph for web service A that has Spring beans from web service B. Setting this property to false shows only the class name when these conflicts occur and does not show the Spring bean name.

<table>
<thead>
<tr>
<th>Type</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>true</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java</td>
</tr>
</tbody>
</table>

### check-bt-excludes-early

Reverses the default order in which Java and .NET agents evaluate rules. If true, exclude rules are evaluated before match rules.

<table>
<thead>
<tr>
<th>Type</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>false</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java, .NET</td>
</tr>
</tbody>
</table>

### collect-user-data-sync

Collect user data from diagnostic POJO data collectors synchronously. Does not require a restart.

<table>
<thead>
<tr>
<th>Type</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>true</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java</td>
</tr>
</tbody>
</table>

### collection-capture-period-in-minutes

Total interval in minutes since server restart for which collections are captured for leak evaluation. The property takes effect only after the node restart.

<table>
<thead>
<tr>
<th>Type</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>30</td>
</tr>
<tr>
<td>Range</td>
<td>Minimum=5; Maximum=N/A</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java</td>
</tr>
</tbody>
</table>
This reference page contains information about app agent node properties. The properties are listed in alphabetical order.

### disable-agent

This property enables/disables the agent. When this property is set to true, TransactionEntryPoints will not be monitored. No new BTs or metrics will be registered, metrics, and snapshots will not be reported. Agent background threads will not be stopped. When set to false, the agent becomes active immediately. It does not need a restart.

<table>
<thead>
<tr>
<th>Type</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>False</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java</td>
</tr>
</tbody>
</table>
### disable-agent-api

This property disables all calls to the agent-api library. The calls function as no-ops when disabled.

<table>
<thead>
<tr>
<th>Type</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>False</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java</td>
</tr>
</tbody>
</table>

### disable-custom-exit-points-for

Disables certain types of automatically detected exit points: SAP, Mail, LDAP, and so on. Type names are case sensitive. Use commas to separate multiple types.

<table>
<thead>
<tr>
<th>Type</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>none</td>
</tr>
<tr>
<td>Supported values</td>
<td>CASSANDRA, Coherence, DangaMemcache, EHCache, LDAP, Memcache, MongoDB, RABBIT_MQ, REDIS, RMI, SAP, THRIFT</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java</td>
</tr>
</tbody>
</table>

### disable-exit-call-correlation-for

Disable exit call correlation for a specific type of call. For example, KAFKA, WCF, WEB_SERVICE, HTTP, JMS, and RMI. By default, all exit call correlations are enabled.

<table>
<thead>
<tr>
<th>Type</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>none</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java, .NET</td>
</tr>
</tbody>
</table>
### disable-exit-call-metrics-for

Disables exit call monitoring for a specific type of exit call; for example, HTTP, JMS, WEB_SERVICE. If this property is set, the average data—calls/min, avg response time—for the specific exit call type is not collected. However, for a snapshot, all details are collected. Set this property if the application makes a large number of exit calls per transaction and the avg metrics are not important.

<table>
<thead>
<tr>
<th>Type</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>By default, all exit call metrics are enabled.</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java, .NET</td>
</tr>
</tbody>
</table>

### disable-percentile-metrics

App agents that support percentile metrics enable collection by default. Disable percentile metrics on the Configuration > Slow Transaction Thresholds window or set this node property manually to true to disable percentiles. Changes to this property do not require an agent restart.

<table>
<thead>
<tr>
<th>Type</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>false</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java, .NET</td>
</tr>
</tbody>
</table>

### disable-service-monitoring-metrics

This property stops the BT-to-BT incoming cross-application metrics to be reported (but normal cross-application metrics continue to be reported, as expected).

<table>
<thead>
<tr>
<th>Type</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>True</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java</td>
</tr>
</tbody>
</table>

### disable-soap-header-correlation-non-http

This property controls correlation with web service transactions. When enabled, it prevents injection of the correlation header into a SOAP message if the WCF transport is not over HTTP or HTTPS.

<table>
<thead>
<tr>
<th>Type</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>false</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java, .NET</td>
</tr>
</tbody>
</table>

For more information, see [WCF Entry Points](#)
### disabled-features

Specifies types of data for which agent reporting to suppress. Use this property to provide to disable data collection mechanisms at the agent to limit data reported by the agent for security or privacy reasons. This agent configuration overrides any Controller configuration that affects the data.

You can disable

- **LOG_PAYLOAD**: Log payload, such as the node property `log-request -payload`
- **RAW_SQL**: Raw SQL statements
- **CUSTOM_EXIT_SNAP_DATA**: Snapshot data in custom exits
- **METHOD_INV_DATA_COLLECTOR**: Diagnostic data collectors, method invocation
- **HTTP_DATA_COLLECTOR**: Diagnostic data collectors, HTTP requests
- **INFO_POINT**: Information points
- **ALL**: All of the above
- **NONE**: None of the above. This is equivalent to the default agent behavior.

<table>
<thead>
<tr>
<th>Type</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>none</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java</td>
</tr>
</tbody>
</table>

**Examples**

Configure disabled features in the app-agent-config.xml in the versioned conf directory for the agent. Specify the data category as the value attribute of the disabled-features property. You can have multiple data categories excluded by listing each separated by commas. For example:

With `capture-raw-sql` enabled, AppDynamics captures the SQL call in the following form:

```xml
<app-agent-configuration>
  <configuration-properties>
    ....
    <property name="disabled-features"
      value="RAW_SQL,LOG_PAYLOAD"/>
  </configuration-properties>
  ....
</app-agent-configuration>
```

---

### disable-ibmbpm-data-collectors

Sets whether data-collectors for IBM-BPM task business transactions should be disabled (value=true) or enabled (value=false).

<table>
<thead>
<tr>
<th>Type</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>false</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java</td>
</tr>
</tbody>
</table>
### disable-ibmbpm-usertask-bt-in-process-correlation
Sets whether business transactions in-process correlation for IBM-BPM UserTask business transactions should be disabled (value=true) or enabled (value=false).

<table>
<thead>
<tr>
<th>Type</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>false</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java</td>
</tr>
</tbody>
</table>

### disable-ibmbpm-usertask-bt-naming
Sets whether business transactions naming scheme for IBM-BPM UserTask Business Transactions should be disabled (value=true) or enabled (value=false). When it is set to true, the Business Transactions would be named as per the default URL and not the meaningful names.

<table>
<thead>
<tr>
<th>Type</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>false</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java</td>
</tr>
</tbody>
</table>

### dev-mode-suspend-cpm
The maximum number of transactions monitored per minute during development mode before the system switches out of development mode into normal operation mode.

<table>
<thead>
<tr>
<th>Type</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>500</td>
</tr>
<tr>
<td>Range</td>
<td>Minimum=0; Maximum=N/A</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java</td>
</tr>
</tbody>
</table>

### dont-show-packages
Do not show these packages / class names in addition to the ones configured in the global call graph configuration, for the call graphs captured on this node. Does not need a restart.

<table>
<thead>
<tr>
<th>Type</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>none</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java,.NET</td>
</tr>
</tbody>
</table>
**downstream-tx-detection-enabled**

If the agent cannot reach the controller for a prolonged period, it turns off most services and notifies the continuing tiers that upstream transaction was detected and is not being monitored. Set this property to `true` to enable the continuing tiers to detect their own transactions in the event of network failure on the upstream tiers.

<table>
<thead>
<tr>
<th>Type:</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>false</td>
</tr>
<tr>
<td>Platform(s):</td>
<td>Java, .NET</td>
</tr>
</tbody>
</table>

**enable-all-rsd-error-propagation**

This property enables the agent to recognize errors that occur in calls outside of or tangential to business transactions.

- `false`: All business transaction errors are reported, but errors outside of the business transaction are not reported.
- `true`: All business transaction errors are reported. In addition, errors outside of the business transaction are reported. For example, errors generated from tracking analytics for a business transaction are reported.
- `full-disable`: No asynchronous errors are recognized.

<table>
<thead>
<tr>
<th>Type:</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>false</td>
</tr>
<tr>
<td>Platform(s):</td>
<td>Java</td>
</tr>
</tbody>
</table>

**enable-async-service-endpoints**

By default the Java Agent automatically detects service endpoints for worker threads. Set this property to "false" to disable service endpoint detection for worker threads. This has the same effect as the Automatic Service Endpoint Detection checkbox for other types of service endpoints on Configuration > Instrumentation > Service Endpoints.

<table>
<thead>
<tr>
<th>Type:</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>true</td>
</tr>
<tr>
<td>Platform(s):</td>
<td>Java</td>
</tr>
</tbody>
</table>
### enable-axon-entry

This node property is used to disable axon entry. By default, it is enabled.

<table>
<thead>
<tr>
<th>Type:</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>true (By default, axon entry is enabled.)</td>
</tr>
<tr>
<td>Platform(s):</td>
<td>Java</td>
</tr>
</tbody>
</table>

### enable-axon-exit

This node property is used to disable axon exit. By default, it is enabled.

<table>
<thead>
<tr>
<th>Type:</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>true (By default, axon entry is enabled.)</td>
</tr>
<tr>
<td>Platform(s):</td>
<td>Java</td>
</tr>
</tbody>
</table>

### enable-bt-block-wait-time-monitoring

This property controls capture of per BT block and wait time metrics. It is disabled by default.

<table>
<thead>
<tr>
<th>Type:</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>false - (Disabled by default)</td>
</tr>
<tr>
<td>Platform(s):</td>
<td>Java</td>
</tr>
</tbody>
</table>

### enable-bt-cpu-time-monitoring

This property controls whether the agent captures the CPU time taken by a business transaction.

<table>
<thead>
<tr>
<th>Type:</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>true</td>
</tr>
<tr>
<td>Platform(s):</td>
<td>Java</td>
</tr>
</tbody>
</table>
### enable-default-http-error-code-reporter

This property disables or enables automatic HTTP error code reporting for error codes between 400 to 505.

<table>
<thead>
<tr>
<th>Type</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>true</td>
</tr>
<tr>
<td>Platform(s):</td>
<td>Java, .NET</td>
</tr>
</tbody>
</table>

### enable-info-point-data-in-snapshots

This property disables or enables the capture of information point calls in snapshots. When this property is set to true, information point calls appear in the User Data section of the snapshot.

<table>
<thead>
<tr>
<th>Type</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>false</td>
</tr>
<tr>
<td>Platform(s):</td>
<td>Java, .NET</td>
</tr>
</tbody>
</table>

### enable-instance-monitoring

This property enables or disables Instance tracking on this node. Does not need a JVM restart.

<table>
<thead>
<tr>
<th>Type</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>false</td>
</tr>
<tr>
<td>Platform(s):</td>
<td>Java</td>
</tr>
</tbody>
</table>

### enable-interceptors-for-security

This property enables or disables security interceptors on this node. Set this property to true in environments where the Java 2 Security Manager is enabled. If the Java 2 Security Manager is enabled, and this property is not set to true, then the agent will encounter SecurityExceptions, and will not be able to collect the data that it should. Does not need a JVM restart.

<table>
<thead>
<tr>
<th>Type</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>false</td>
</tr>
<tr>
<td>Platform(s):</td>
<td>Java</td>
</tr>
</tbody>
</table>
### enable-json-bci-rules

Set this property to true to enable JSON bytecode instrumentation rules. AppDynamics instruments the `get` and `getString` methods within the `org.json.JSONObject` package/class when you set this value to true. Needs a JVM restart.

<table>
<thead>
<tr>
<th>Type</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Default value</strong>:</td>
<td>true (This only affects new applications; applications created with a 3.7.x controller will still have this property set by default to false.)</td>
</tr>
<tr>
<td><strong>Platform(s)</strong>:</td>
<td>Java</td>
</tr>
</tbody>
</table>

### enable-kafka-consumer

Set the `enable-kafka-consumer` to `true` to enable Apache Kafka consumer entry points. For more information see 'Apache Kafka Backends' on Java Backend Detection.

<table>
<thead>
<tr>
<th>Type</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Default value</strong>:</td>
<td>false</td>
</tr>
<tr>
<td><strong>Platform(s)</strong>:</td>
<td>Java</td>
</tr>
</tbody>
</table>

### enable-object-size-monitoring

This property is related to Automatic Leak Detection (ALD) and enables or disables Object Size monitoring on this node. Changing this property does not need a JVM restart. ALD is supported for JVM version 1.6 and up.

<table>
<thead>
<tr>
<th>Type</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Default value</strong>:</td>
<td>false</td>
</tr>
<tr>
<td><strong>Platform(s)</strong>:</td>
<td>Java</td>
</tr>
</tbody>
</table>
### enable-soap-header-correlation

This property controls correlation with web service transactions. When enabled, a node which receives a web service transaction may correlate that transaction with any downstream transactions. The ability to correlate depends on the particular web services framework. Currently, correlation is supported only by Apache Synapse and CXF frameworks. When disabled, the agent will not perform correlation through any web service tiers.

<table>
<thead>
<tr>
<th>Type</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>false</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java, .NET</td>
</tr>
</tbody>
</table>

### enable-spring-integration-entry-points

This property disables or enables the default detection of Spring Integration entry points. Set to `false` to disable.

Default detection of Spring Integration entry points is based on `MessageHandler`. In cases where a lot of application flow happens before the first `MessageHandler` is executed:

- Set this property to `false`
- Configure suitable **POJO** entry points
- Specify the property `spring-integration-receive-marker-classes`

See also [Spring Integration Support](#).

<table>
<thead>
<tr>
<th>Type</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>true</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java</td>
</tr>
</tbody>
</table>

### enable-startup-snapshot-policy

This property disables or enables the policy for start-up transaction snapshot. This means snapshots are collected for all BTs for all invocations for the first 15 minutes of application server startup.

<table>
<thead>
<tr>
<th>Type</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>false</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java, .NET</td>
</tr>
</tbody>
</table>
### enable-transaction-correlation

This property disables or enables transaction correlation. It does not require a restart.

<table>
<thead>
<tr>
<th>Type</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>true</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java, .NET</td>
</tr>
</tbody>
</table>

### enable-vertx-http

Enable or disable servlet HTTP entry points and exit points for Vert.x.

<table>
<thead>
<tr>
<th>Type</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>true</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java</td>
</tr>
</tbody>
</table>

### enable-vertx-message-entry

Enable or disable Vert.x verticle message entry points for continuing transactions.

<table>
<thead>
<tr>
<th>Type</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>true</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java</td>
</tr>
</tbody>
</table>

### enable-xml-bci-rules

This property enables Java XML Binding and DOM Parser bytecode instrumentation rules. Set to true to enable. The change takes effect after a JVM restart.

<table>
<thead>
<tr>
<th>Type</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>true (This only affects new applications; applications created with a 3.7.x Controller will still have this property set by default to false.)</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java</td>
</tr>
</tbody>
</table>
### end-to-end-message-latency-threshold-millis

Enables end-to-end message latency monitoring for distributed asynchronous systems by setting up a threshold. Any message taking more time than the threshold is viewable through the Event Viewer.

<table>
<thead>
<tr>
<th>Type</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>0</td>
</tr>
<tr>
<td>Range</td>
<td>Minimum=0; Maximum=36000</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java</td>
</tr>
</tbody>
</table>
App Agent Node Properties (F-I)

This reference page contains information about app agent node properties. The properties are listed in alphabetical order.

**find-entry-points**

Set this property to `true` to log all potential entry points that are hitting instrumented exit points or loggers to the Business Transactions log file.

Use this property when you suspect that some traffic is not being detected as business transactions. You should only enable this property for debugging purposes. You should disable this property in a production setup.

**Tip:** For new applications, you can use the interactive Live Preview tools to discover entry points.

<table>
<thead>
<tr>
<th>Type:</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>false</td>
</tr>
<tr>
<td>Platform(s):</td>
<td>Java, .NET</td>
</tr>
</tbody>
</table>

**heap-storage-monitor-devmode-disable-trigger-pct**

The maximum Java heap utilization percentage for development mode. If the heap utilization exceeds this value, development mode is automatically disabled.

<table>
<thead>
<tr>
<th>Type:</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>90</td>
</tr>
<tr>
<td>Range:</td>
<td>Minimum=0, Maximum=100</td>
</tr>
<tr>
<td>Platform(s):</td>
<td>Java</td>
</tr>
</tbody>
</table>
### ibmbpm-systemtask-bt-naming

Decides BT naming scheme for IBM-BPM System Task POJO business transactions.

The property value is comma-separated identifiers chosen from `project`, `bpd`, `task`, and `implementation`, and is order-sensitive.

<table>
<thead>
<tr>
<th>Type</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>default</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java</td>
</tr>
</tbody>
</table>

**Example**

For example:

- `value = project, task` means that the POJO business transactions is named as: `<project-name> / <task-name>`
- `value = task, project` means that the POJO business transactions is named as: `<task-name> / <project-name>`
- `value = none` means that the POJO business transactions is not detected, in other words, disabled.
- `value = default` would mean that all the identifiers are used for BT naming are in default order, that is, they are named as: `<project-name> / <bpd-name> / <task-name> / <implementation-name>`
This reference page contains information about app agent node properties. The properties are listed in alphabetical order.

### jdbc-callable-statements

Use this property to indicate the implementation classes of the `java.sql.CallableStatement` interface that should be instrumented.

<table>
<thead>
<tr>
<th>Type:</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>none</td>
</tr>
<tr>
<td>Platform(s):</td>
<td>Java</td>
</tr>
</tbody>
</table>

**Examples**

For example, to instrument calls to Times Ten (an unsupported database), you could set this property to the following:

```java
com.timesten.jdbc.JdbcOdbcCallableStatement
```
### jdbc-connections

Use this property to indicate the implementation classes of the `java.sql.Connection` interface that should be instrumented.

<table>
<thead>
<tr>
<th>Type</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>none</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java</td>
</tr>
</tbody>
</table>

**Examples**

For example, to instrument calls to Times Ten (an unsupported database), you could set this property to the following:

```
com.timesten.jdbc.JdbcOdbcConnection
```

### jdbc-dbcam-integration-enabled

Use this property to integrate the Java Agent with AppDynamics for Databases and Database Monitoring. Changes to this property do not require a JVM restart for JDK 1.6 and higher. Older 1.5 JVMs do not support class reloading, so for those environments, a restart is required.

<table>
<thead>
<tr>
<th>Type</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>false</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java</td>
</tr>
</tbody>
</table>

**More Info**

**AppDynamics for Databases**: When this property is enabled, you can link to AppDynamics for Databases from a Transaction Snapshot Flow Map where an exit call is to an Oracle database, and analyze the SQL statements that were running at the time of the snapshot. This property works in conjunction with the AppDynamics for Databases license and database collector that has been previously set up. Integration must also be set up from the Admin pages of the Controller UI. For more information, see Integrate and Use AppDynamics for Databases with AppDynamics Pro.

**Database Monitoring**: Set this property to enable snapshot correlation between Java applications and Oracle databases. This property configures Globally Unique Identifier (GUID) session tagging between business transactions monitored by the Java Agent and Oracle databases monitored by Database Monitoring. Each snapshot is identified by a GUID which Database Monitoring instrumentation injects into the Oracle session using a standard JDBC API. This enables AppDynamics to collect the session properties, including the GUID. When the queries are correlated with the GUID, AppDynamics can correlate the backend database activity with the business transaction snapshot.
jdbc-prepared-statements

Use this property to indicate the implementation classes of the `java.sql.PreparedStatement` interface that should be instrumented.

<table>
<thead>
<tr>
<th>Type</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>none</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java</td>
</tr>
</tbody>
</table>

Examples

For example, to instrument calls to Times Ten (an unsupported database), you could set this property to the following:

```
com.timesten.jdbc.JdbcOdbcPreparedStatement
```

jdbc-resultsets

Use this property to indicate the implementation classes of the `java.sql.ResultSet` interface that should be instrumented.

<table>
<thead>
<tr>
<th>Type</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>none</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java</td>
</tr>
</tbody>
</table>

Examples

For example, to instrument calls to Times Ten (an unsupported database), you could set this property to the following:

```
com.timesten.jdbc.JdbcStatement
```

jdbc-statements

Use this property to indicate the implementation classes of the `java.sql.Statement` interface that should be instrumented.

<table>
<thead>
<tr>
<th>Type</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>none</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java</td>
</tr>
</tbody>
</table>

Examples

For example, to instrument calls to Times Ten (an unsupported database), you could set this property to the following:

```
com.timesten.jdbc.JdbcOdbcStatement
```
### jmx-appserver-mbean-finder-delay-in-seconds

When an app server starts up, the associated MBean server starts and the MBeans are discovered. The timing of these activities varies by app server and by configuration. If this activity is not completed in the time that the AppDynamics agent is expecting to discover the MBeans, then the MBean Browser will not show them. Using this node property, you can delay the discovery of MBeans to make sure that agent discovers all the domains after complete startup of the app server. For example, you can set the delay to a time which is 1.5 times of the server startup time. The default delay for the AppDynamics agent is two minutes.

<table>
<thead>
<tr>
<th>Type</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>120</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java</td>
</tr>
</tbody>
</table>

### jmx-operation-timeout-in-milliseconds

Controls the length of time the Java Agent waits before timing out an MBean operation. If you have MBean operations that run longer than the default of 10 seconds, you can increase the timeout value up to 5 minutes.

<table>
<thead>
<tr>
<th>Type</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>10000 (10 seconds)</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java</td>
</tr>
</tbody>
</table>

### jmx-rediscover-mbean-servers

When an app server starts up, the associated MBean server starts and the MBeans are discovered. The timing of these activities varies by app server and by configuration. If this activity is not completed in the time that the AppDynamics agent is expecting to discover the MBeans, then the MBean Browser will not show them. Using this node property, you can trigger the rediscovery of MBeans to make sure that the agent discovers all the domains after complete startup of the app server. Set this property to true and reset the app agent, as described in Manage App Agents.

<table>
<thead>
<tr>
<th>Type</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>false</td>
</tr>
</tbody>
</table>
| Platform(s)   | Java/td>
### jmx-query-timeout-limit

This property sets the timeout limit. Setting this property to negative will disable timeout. For example, if you set the property to -1, timeout is disabled.

<table>
<thead>
<tr>
<th>Type</th>
<th>Long</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>5</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java</td>
</tr>
</tbody>
</table>

### jrmp-enable

This property enables or disables AppDynamics support for Sun RMI over Java Remote Protocol (JRMP). You should test Sun RMI JRMP support in a staging environment before using it on production systems. Enable Sun JRMP support by setting this property to true.

<table>
<thead>
<tr>
<th>Type</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>false</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java</td>
</tr>
</tbody>
</table>

### leak-diagnostic-interval-in-minutes

The interval at which diagnostic data, content summary and activity trace, is captured for leaking collections.

<table>
<thead>
<tr>
<th>Type</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>30</td>
</tr>
<tr>
<td>Range</td>
<td>Minimum=2; Maximum=N/A</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java, .NET</td>
</tr>
</tbody>
</table>

### log-request-payload

Set this property to true to log the request payload—HTTP parameters, cookies, session keys, and so on—as part of a transaction snapshot. The log-request-payload property includes logging of WCF HTTP parameters for .NET.

<table>
<thead>
<tr>
<th>Type</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>false</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java, .NET</td>
</tr>
</tbody>
</table>
### logbased-visibility-log-check-interval-in-millis

How often the agent checks application log files for information related to garbage collection performance.

<table>
<thead>
<tr>
<th>Type</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>1000 ms</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java, .NET</td>
</tr>
</tbody>
</table>
App Agent Node Properties (M)

On this page:

- maximum-activity-trace-stack-depth
- max-async-task-registration-requests-allowed
- max-async-task-registrations-allowed
- max-business-transactions
- max-call-elements-per-snapshot
- max-concurrent-snapshots
- max-correlation-header-size
- max-error-snapshots-per-minute
- max-jdbc-calls-per-callgraph
- max-jdbc-calls-per-snapshot
- max-metrics-allowed (New in 4.5.7)
- max-service-end-points-per-async-type
- max-service-end-points-per-entry-point-type
- max-service-end-points-per-node
- max-service-end-points-per-thread
- max-urls-per-error-code
- min-duration-for-jdbc-call-in-ms
- min-load-per-minute-diagnostic-session-trigger
- minimum-age-for-evaluation-in-minutes
- minimum-number-of-elements-in-collection-to-deep-size
- minimum-size-for-evaluation-in-mb
- min-transaction-stall-threshold-in-seconds
- msmq-correlation-field
- msmq-single-threaded

This reference page contains information about app agent node properties. The properties are listed in alphabetical order.

In general use caution when modifying the agent default settings. If increasing limits specified for an agent, you need to carefully assess and monitor memory consumption by the agent after the change.

### maximum-activity-trace-stack-depth

This determines the depth of the stack trace to capture as part of an activity trace session. By default, the size of the code paths for OIT (Object Instance Tracking), ALD (Automatic Leak Detection) and MIDS (Memory Intensive Data Structures) are set to 10. To increase this limit, use this property.

**Warning:** A larger depth has higher overhead on the system. AppDynamics recommends that you increase the default value of this property only temporarily, and remove it or set it back to 10 once you get the desired output.

<table>
<thead>
<tr>
<th>Type</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>10</td>
</tr>
<tr>
<td>Platform(s):</td>
<td>Java</td>
</tr>
</tbody>
</table>
### max-async-task-registration-requests-allowed
Adjust this property to increase or decrease the number of asynchronous task registration requests allowed.

<table>
<thead>
<tr>
<th>Type</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>500</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java</td>
</tr>
</tbody>
</table>

### max-async-task-registrations-allowed
Adjust this property to increase or decrease the number of asynchronous task registrations allowed.

<table>
<thead>
<tr>
<th>Type</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>500</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java, .NET</td>
</tr>
</tbody>
</table>

### max-business-transactions
Sets a limit on the number of business transactions discovered once an agent is started. The limit helps to ensure that the Controller I/O processing capability and agent memory requirements are appropriate for a production environment. See [Business Transactions](#).

**Warning:** Changing this setting can affect the resource consumption of your deployment. Before you change this setting verify your application environment and Controller can handle any increased resource requirements.

<table>
<thead>
<tr>
<th>Type</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>50</td>
</tr>
<tr>
<td>Range</td>
<td>Minimum=N/A; Maximum=200</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java, .NET</td>
</tr>
</tbody>
</table>

### max-call-elements-per-snapshot
This property represents the maximum number of elements that are collected for any call graph for a snapshot. When the limit is reached, the agent stops collecting more data for this call graph, reports what has been collected to that point, and marks the call graph with a warning that the limit was reached. It is not recommended to dramatically increase this number as it may have overhead implications.

<table>
<thead>
<tr>
<th>Type</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>5000</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java, .NET</td>
</tr>
</tbody>
</table>
### max-concurrent-snapshots

The maximum number of total snapshots that are allowed, including continuing transactions. When the queue goes over the value set, additional snapshots are dropped. This property is ignored in Development mode.

<table>
<thead>
<tr>
<th>Type:</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>20</td>
</tr>
<tr>
<td>Range: Values must be positive integers. No other constraints.</td>
<td></td>
</tr>
<tr>
<td>Platform(s):</td>
<td>Java, .NET</td>
</tr>
</tbody>
</table>

### max-correlation-header-size

The maximum size of the correlation header.

<table>
<thead>
<tr>
<th>Type:</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>4096</td>
</tr>
<tr>
<td>Range: Values must be positive integers. No other constraints.</td>
<td></td>
</tr>
<tr>
<td>Platform(s):</td>
<td>Java</td>
</tr>
</tbody>
</table>

### max-error-snapshots-per-minute

A limit for the number of snapshots per minute due to errors. For example, if too many error snapshots are being seen, then tweak this value to reduce noise in the snapshots.

<table>
<thead>
<tr>
<th>Type:</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>5</td>
</tr>
<tr>
<td>Platform(s):</td>
<td>Java, .NET</td>
</tr>
</tbody>
</table>
max-jdbc-calls-per-callgraph

The maximum number of JDBC/ADO.NET exit-call stack samples per call graph. Only queries taking more time than the value of `min-duration-for-jdbc-call-in-ms` are reported. Changing the value does not require a restart.

<table>
<thead>
<tr>
<th>Type</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>100</td>
</tr>
<tr>
<td>Range</td>
<td>Minimum=1; Maximum=1000</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java, .NET</td>
</tr>
</tbody>
</table>

max-jdbc-calls-per-snapshot

The maximum number of JDBC/ADO.NET exit calls allowed in a snapshot. Calls after the limit are not recorded. Changing the value does not require a restart.

<table>
<thead>
<tr>
<th>Type</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>500</td>
</tr>
<tr>
<td>Range</td>
<td>Minimum=1; Maximum=5000</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java, .NET</td>
</tr>
</tbody>
</table>

max-metrics-allowed (New in 4.5.7)

This property sets the upper limit to the number of metrics that can be registered by the agent.

<table>
<thead>
<tr>
<th>Type</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>5000 or as set with -Dappdynamics.agent.maxMetrics=</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java</td>
</tr>
</tbody>
</table>

Note: If the number of metrics—registered + unregistered—is greater than the new node property value, then the new metrics will not be created.
### max-service-end-points-per-async-type

Maximum total number of service endpoints that can be registered for each asynchronous entry point type, such as worker thread. Because a single transaction may spawn many threads, you may expect more asynchronous service endpoint types than for synchronous service endpoint types.

<table>
<thead>
<tr>
<th>Type</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>40</td>
</tr>
<tr>
<td>Range</td>
<td>Minimum=0; Maximum=N/A</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java</td>
</tr>
</tbody>
</table>

### max-service-end-points-per-entry-point-type

Maximum total number of service endpoints that can be registered for each entry point type, such as servlet, struts action, web service, and so on.

<table>
<thead>
<tr>
<th>Type</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>25</td>
</tr>
<tr>
<td>Range</td>
<td>Minimum=0; Maximum=N/A</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java, .NET</td>
</tr>
</tbody>
</table>

### max-service-end-points-per-node

Maximum total number of service endpoints that can be detected on a single node. Increasing the value of this property enables more service endpoints to be detected on a particular node.

<table>
<thead>
<tr>
<th>Type</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>100</td>
</tr>
<tr>
<td>Range</td>
<td>Minimum=0; Maximum=N/A</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java, .NET</td>
</tr>
</tbody>
</table>
### max-service-end-points-per-thread

Maximum number of service endpoints detected on a single thread of transaction execution.

If this property is set to the default value of one and two service endpoints are detected that impact one specific transaction, only one service endpoint will be evaluated at any time. If a second service endpoint is detected in the context of the first one, the second is ignored. But, if the second service endpoint starts after the first one ends, the second service endpoint will be evaluated.

Increase this property to monitor additional service endpoints on a thread. This number ensures a maximum limit on overhead and number of metrics due to service endpoints on each thread execution.

**Type:** Integer  
**Default value:** 1  
**Range:** Minimum=0; Maximum=N/A  
**Platform(s):** Java, .NET

### max-urls-per-error-code

Increases the number of URLs the agent can track that produced a certain error. Once the maximum has been reached, all remaining errors are classified as unknown.

**Type:** Integer  
**Default value:** 50  
**Platform(s):** Java

### min-duration-for-jdbc-call-in-ms

A JDBC/ADO.NET call taking more time than the specified time (in milliseconds) is captured in the call graph. The query continues to show up in a transaction snapshot. Setting this value too low (< 10ms) may affect application response times. Changing the value does not require a restart.

**Type:** Integer  
**Default value:** 10  
**Range:** Minimum=0; Maximum=N/A  
**Platform(s):** Java, .NET
### min-load-per-minute-diagnostic-session-trigger

Indicates the number of requests per Business Transaction to evaluate before triggering a diagnostic session. This is useful to prevent diagnostic sessions when there is not enough load.

<table>
<thead>
<tr>
<th>Type</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>10</td>
</tr>
<tr>
<td>Range</td>
<td>Minimum=0; Maximum=N/A</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java, .NET</td>
</tr>
</tbody>
</table>

### minimum-age-for-evaluation-in-minutes

Automatic Leak Detection (ALD) tracks all frequently used Collections. For a Collection object to be identified and monitored it must meet the conditions defined by the ALD properties. This property is the first criteria that needs to be met. The value is the minimum age of the Collection in minutes. The property takes effect after node restart.

From the point the collection is captured, it is monitored if it is still available for the specified period without getting garbage collected. If it survives then it is evaluated for size checks and if it meets the criteria then it is monitored for long term growth in size.

**Warning:** If you reduce the default there may be a performance hit on the CPU and memory because AD needs to process more collections.

<table>
<thead>
<tr>
<th>Type</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>30</td>
</tr>
<tr>
<td>Range</td>
<td>Minimum=5; Maximum=N/A</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java</td>
</tr>
</tbody>
</table>

### minimum-number-of-elements-in-collection-to-deep-size

Automatic Leak Detection (ALD) tracks all frequently used Collections. For a Collection object to be identified and monitored for it must meet the conditions defined by the ALD properties. This property sets the number of elements threshold.

**Warning:** If you reduce the default there may be a performance hit on the CPU and memory because AD is processing more collections.

<table>
<thead>
<tr>
<th>Type</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>1000</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java</td>
</tr>
</tbody>
</table>
minimum-size-for-evaluation-in-mb

Automatic Leak Detection (ALD) tracks all frequently used Collections. For a Collection object to be identified and monitored it must meet the conditions defined by the ALD properties. This property sets the minimum initial size in megabytes for a collection to qualify for monitoring. The collection must also survive for the period specified in the minimum-age-for-evaluation-in-minutes property.

**Warning:** If you reduce the default there may be a performance hit on the CPU and memory because AD is processing more collections.

<table>
<thead>
<tr>
<th>Type</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>5</td>
</tr>
<tr>
<td>Range</td>
<td>Minimum=1; Maximum=N/A</td>
</tr>
<tr>
<td>Platform(s):</td>
<td>Java</td>
</tr>
</tbody>
</table>

min-transaction-stall-threshold-in-seconds

For Executor mode only, the asynchronous transactions do not get checked for stall unless they run at least the specified number of seconds.

<table>
<thead>
<tr>
<th>Type</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>60</td>
</tr>
<tr>
<td>Platform(s):</td>
<td>Java</td>
</tr>
</tbody>
</table>
**msmq-correlation-field**

By default, the .NET agent disables downstream correlation for MSMQ message queues. Register this node property on both the publishing and receiving tiers to enable downstream correlation for MSMQ and to specify the field where the agent writes correlation data.

The agent supports *Extension* or *Label* fields. By default, the agent writes correlation data to the *Extension* field, however, some frameworks built on MSMQ write data to the *Extension* field. Only use *Label* when the *Extension* field is not available because it is already in use by the framework. The NServiceBus implementation of MSMQ uses the *Extension* field, so for NServiceBus use *Label*. See MSMQ Backends for .NET.

<table>
<thead>
<tr>
<th>Type</th>
<th>String</th>
</tr>
</thead>
</table>
| Supported values: | • None: Disable downstream correlation for MSMQ  
|               | • Label: Store correlation information in the label field  
|               | • Extension: Store correlation information in the extension field |
| Default value:    | None |
| Platform(s):       | .NET |

**msmq-single-threaded**

Specify the threading architecture for the MSMQ message queue. The default value is false. For multithreaded queue implementations, change the value to false. See MSMQ Backends for .NET.

<table>
<thead>
<tr>
<th>Type</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>false</td>
</tr>
<tr>
<td>Platform(s):</td>
<td>.NET</td>
</tr>
</tbody>
</table>
This reference page contains information about app agent node properties. The properties are listed in alphabetical order.

In general use caution when modifying the agent default settings. If increasing limits specified for an agent, you need to carefully assess and monitor memory consumption by the agent after the change.

### normalize-prepared-statements

When this flag is set to `true`, any variables in prepared statement SQL would be substituted with `'?` before query text is added to any snapshots.

<table>
<thead>
<tr>
<th>Type</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td><code>true</code></td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java</td>
</tr>
</tbody>
</table>

### nservicebus-single-threaded

Specify the threading architecture for the NServiceBus message queue. The value defaults to `true`. For multithreaded queue implementations, change the value to `false`. See NServiceBus Backends for .NET.

<table>
<thead>
<tr>
<th>Type</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td><code>true</code></td>
</tr>
<tr>
<td>Platform(s)</td>
<td>.NET</td>
</tr>
</tbody>
</table>
### osb-ignore-exit-types

This comma-separated property defines which exits protocols (as configured in the Transport Configuration pane) are excluded from detection as uncorrelated custom backends.

Set the property to ‘all’, if you do not want to detect any exit.

<table>
<thead>
<tr>
<th>Type</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>http,jms</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java</td>
</tr>
</tbody>
</table>

### on-demand-snapshots

Collect snapshots for all Business Transactions executed in this node. Does not need a restart. This property is ignored in the Development mode.

<table>
<thead>
<tr>
<th>Type</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>false</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java, .NET</td>
</tr>
</tbody>
</table>

### percentile-method-option

You can choose one of two different algorithms to calculate percentiles in AppDynamics:

- **P Square algorithm (default):** This option consumes the least amount of storage and incurs the least amount of CPU overhead. The accuracy of the percentile calculated varies depending on the nature of the distribution of the response times. You should use this option unless you doubt the accuracy of the percentiles presented.

- **Quantile Digest algorithm:** This option consumes slightly more storage and CPU overhead but may offer better percentiles depending on how the response times are distributed.

Changes to this property do not require that you restart the agent.

<table>
<thead>
<tr>
<th>Type</th>
<th>Numeric</th>
</tr>
</thead>
</table>
| Supported Values | 1: P Square  
                      2: Quantile Digest |
| Default value | 1 |
| Platform(s)  | Java, .NET |
### queue-single-threaded

Specify the threading architecture for the IBM MQ message queue. The value defaults to `false`.

When the IBM MQ `Get()` method is called, the transaction/snapshot is ended prematurely. To prevent this, set the agent node property: `queue-single-threaded=true`.

<table>
<thead>
<tr>
<th>Type</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td><code>false</code></td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java, .NET</td>
</tr>
</tbody>
</table>

### reportingFrequencyInMillis

The agent property specifies the frequency of Request Segment Data (RSD) uploads from the agent.

<table>
<thead>
<tr>
<th>Type</th>
<th>Numeric</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td><code>10000 ms</code></td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java</td>
</tr>
</tbody>
</table>

### rest-num-segments

The property, `rest-num-segments` specifies the \( n \) in the first-\( n \)-segments parameter in `rest-uri-segment-scheme`. If this property is 0 or less, then the value of this property is ignored. The value of this property is also ignored if `rest-uri-segment-scheme=full`.

<table>
<thead>
<tr>
<th>Type</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td><code>2</code></td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java</td>
</tr>
</tbody>
</table>
rest-transaction-naming

This node property determines the format in which REST-based business transactions are named. You can use variables to populate the name with values bound at runtime. Any characters in the property value that do not match a variable are treated as literal text in the business transaction name, so you can, for example, separate variables with a colon, slash, or another character.

The agent takes each parameter and fills in the proper value based on the annotations and properties of the Java class:

- `{class-name}`: The app agent will fill in the name of the Java class mapped to the REST resource.
- `{method-name}`: The method being called.
- `{class-annotation}`: Class annotation values.
- `{method-annotation}`: Method annotation applied to the method (not always present).
- `{rest-uri}`: URI of the REST resource. The REST URI is further configured using the following properties:
  - `rest-uri-segment-scheme`
  - `rest-num-segments`
- `{http-method}`: HTTP method of the request, GET, POST, and so on.
- `{param-%d}`: A parameter to the method identified by position. Replace `%d` with the position of the parameter (ZERO-based).

<table>
<thead>
<tr>
<th>Type</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td><code>{class-annotation}/{method-annotation}.{http-method}</code></td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java</td>
</tr>
</tbody>
</table>

Examples

See Using Default Settings and Using rest-transaction-naming Properties.

rest-uri-segment-scheme

The property, rest-uri-segment-scheme has three valid values: `first-n-segments`, `last-n-segments`, and `full`. This property indicates how many segments of the URI to use for the URI in `{rest-uri}`. This option is case-sensitive. If the value of this property is `full`, then the value of rest-num-segments is ignored.

<table>
<thead>
<tr>
<th>Type</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td><code>first-n-segments</code></td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java</td>
</tr>
</tbody>
</table>
The App Agent for .NET (agent) automatically discovers RabbitMQ remote services. Use the rmqsegments node property to refine the queue backend name to include some or all segments of the routing key. You must be familiar with your implementation RabbitMQ exchanges and routing keys. See RabbitMQ Exchanges and Exchange Types.

The RabbitMQ routing key is a string. The agent treats dot-separated (".") substrings of the routing key as segments. Set the value for rmqsegments to an integer that represents the number of routing key segments to include in the name. For more details, see information on refining backend naming in RabbitMQ Backends for .NET.

<table>
<thead>
<tr>
<th>Type:</th>
<th>Numeric</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value:</td>
<td>Integer representing the number of routing key segments to include in the name.</td>
</tr>
<tr>
<td>Default value:</td>
<td>0</td>
</tr>
<tr>
<td>Platform(s):</td>
<td>.NET</td>
</tr>
</tbody>
</table>
**App Agent Node Properties (S)**

This reference page contains information about app agent node properties. The properties are listed in alphabetical order.

### show-packages

For the call graphs captured on this node, show the specified packages or class names in addition to the ones configured in the global call graph configuration. Does not need a restart.

<table>
<thead>
<tr>
<th>Type:</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>none</td>
</tr>
<tr>
<td>Platform(s):</td>
<td>Java, .NET</td>
</tr>
</tbody>
</table>

### slow-request-deviation

The value in milliseconds for the deviation from the current average response time. This setting is used for evaluation of slow in-flight transactions. Also, see the `slow-request-threshold` property for more details.

<table>
<thead>
<tr>
<th>Type:</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>200</td>
</tr>
<tr>
<td>Range:</td>
<td>Minimum=10; Maximum=3600</td>
</tr>
<tr>
<td>Platform(s):</td>
<td>Java, .NET</td>
</tr>
</tbody>
</table>
### slow-request-monitor-interval

In-flight requests are checked for slowness in the interval specified by this property. The value is specified in milliseconds.

<table>
<thead>
<tr>
<th>Type</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>100</td>
</tr>
<tr>
<td>Range</td>
<td>Minimum=0; Maximum=3600</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java, .NET</td>
</tr>
</tbody>
</table>

### slow-request-threshold

In-flight requests taking more time than this threshold (in ms) with a deviation greater than the slow-request-deviation property from the current average response time are monitored to capture hot spots.

<table>
<thead>
<tr>
<th>Type</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>500</td>
</tr>
<tr>
<td>Range</td>
<td>Minimum=0; Maximum=3600</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java, .NET</td>
</tr>
</tbody>
</table>

### spring-batch-enabled

Use this property to enable or disable OOTB BT Detection for Spring Batch.

<table>
<thead>
<tr>
<th>Type</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>true</td>
</tr>
<tr>
<td>Platform(s)</td>
<td>Java</td>
</tr>
</tbody>
</table>
spring-integration-receive-marker-classes

Use this property to specify the class and method you have identified as suitable POJO entry points for Spring Integration.

Based on the MessageHandler interface, the App Agent for Java by default automatically discovers exits for all channels except DirectChannel. In cases where a lot of application flow happens before the first MessageHandler is executed,

- Set enable-spring-integration-entry-points=false
- Configure suitable POJO entry points,
- Declare each suitable POJO entry point class/method in this property spring-integration-receive-marker-classes

If the application code polls for messages in a loop, the span of each loop iteration is tracked as a transaction. Tracking begins when the loop begins and end it when the iteration ends. To safeguard against cases where pollableChannel.receive() is not called inside a loop, specify this property for each class/method combination that polls messages in a loop.

After setting this property, restart the application server for changes to this property to take effect.

<table>
<thead>
<tr>
<th>Type:</th>
<th>Comma-separated string of fully-qualified class/method name, such as spring-integration-receive-marker-classes = ,&lt;&gt; ....</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>none</td>
</tr>
<tr>
<td>Platform(s):</td>
<td>Java</td>
</tr>
</tbody>
</table>

Examples

For example, to enable tracking for the following:

```java
class MessageProcessor
{
  void process()
  {
    while(true)
    {
      Message message = pollableChannel.receive()
    }
  }
}
```

set this property as follows:

```
spring-integration-receive-marker-classes = MessageProcessor/process
```

See also Spring Integration Support.
Register this node property to modify the naming scheme for Spring MVC transactions. Bean ID cannot be used as a global naming type. Use the bean ID and method name for global.

<table>
<thead>
<tr>
<th>Type:</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allowed values:</td>
<td>bean-id, simple-class-name, fully-qualified-class-name, business-interface-name, bean-method-name, ben-id-and-method-name, class-and-method-name</td>
</tr>
<tr>
<td>Default value:</td>
<td>none</td>
</tr>
<tr>
<td>Platform(s):</td>
<td>Java</td>
</tr>
</tbody>
</table>
App Agent Node Properties (T-Z)

On this page:
- thread-correlation-classes
- thread-correlation-classes-exclude
- thread-cpu-capture-overhead-threshold-in-ms
- wcf-enable-eum
- websocket-entry-calls-enabled

This reference page contains information about app agent node properties. The properties are listed in alphabetical order.

### thread-correlation-classes

For multi-threaded applications, use this property to configure classes to be included in Java thread correlation when simple prefix-matching (matching on STARTSWITH) is sufficient to identify the classes.

The `thread-correlation-classes` property specifies the classes to include for thread correlation. This property can be used together with the `thread-correlation-classes-exclude` property.

The configured correlation takes effect without requiring a restart of the managed application.

Also see, Threading and the Java Agent.

<table>
<thead>
<tr>
<th>Type</th>
<th>Comma-separated string of fully-qualified class names or package names</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>none</td>
</tr>
<tr>
<td>Platform(s):</td>
<td>Java</td>
</tr>
</tbody>
</table>

This property is not recommended for use in the Executor Mode.
thread-correlation-classes-exclude

For multithreaded applications, use this property to configure classes to be excluded in Java thread correlation when simple prefix-matching (matching on `STARTWITH`) is sufficient to identify the classes. This property specifies the classes to exclude from thread correlation. This property can be used in conjunction with the `thread-correlation-classes` node property.

The configured correlation takes effect without requiring a restart of the managed application. See Configure Multithreaded Transactions (Java only)

<table>
<thead>
<tr>
<th>Type:</th>
<th>Comma-separated string of fully-qualified class names or package names</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>none</td>
</tr>
<tr>
<td>Platform(s):</td>
<td>Java</td>
</tr>
</tbody>
</table>

This property is not recommended for use in the Executor Mode.
### thread-cpu-capture-overhead-threshold-in-ms

Determines the timeout allotted for collecting and calculating Thread CPU Time for 1000 iterations. If the timeout is exceeded, the Java Agent automatically disables CPU time collection for threads. CPU usage information will then be absent from the BT overview or snapshots in the Controller UI. In addition, an INFO-level log similar to the following appears in the logs:

```
[Thread-0] 22 Oct 2013 14:19:26,346 INFO JVMThreadCPUTimeCalculator - Disabling BT CPU Monitoring. Time taken to calculate Thread CPU Time for [1000] iterations is [15 ms] which is greater than the allowed budget of [10 ms].
```

This issue may particularly affect JDK 1.6 on Linux due to the issue `getCurrentThreadCpuTime` is drastically slower than Windows Linux.

**Type:** Integer  
**Default value:** 10 ms  
**Platform(s):** Java

**Examples**

You can increase the `thread-cpu-capture-overhead-threshold-in-ms` property, but it is important to note that this may result in increased overhead on your application. We recommend you use this Java HotSpot VM option instead to speed up the API call itself:

```
-XX:+UseLinuxPosixThreadCPUClocks
```

Restart is needed after changing this value.

### wcf-enable-eum

Enable and disable EUM correlation from a WCF node.

**Type:** Boolean  
**Default value:** false  
**Platform(s):** .NET

**Examples**

The enable EUM correlation, the WCF application must also have the following entry in the Web.Config:

```
<serviceHostingEnvironment aspNetCompatibilityEnabled="true" />
```

See serviceHostingEnvironment.
<table>
<thead>
<tr>
<th>websocket-entry-calls-enabled</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>When set to false, no WebSocket entry calls are detected.</strong></td>
</tr>
<tr>
<td><strong>Type:</strong></td>
</tr>
<tr>
<td><strong>Default value:</strong></td>
</tr>
<tr>
<td><strong>Platform(s):</strong></td>
</tr>
</tbody>
</table>
The AppDynamics proxy is a Java daemon process that handles communication between the following agents and the Controller:

- Node.js
- PHP
- Python
- Web Server

The proxy reports data collected by the agent to the Controller, which stores, baselines, and analyzes it.

This topic presents basic information to help you examine and resolve proxy issues that may prevent an application agent from connecting to the Controller or reporting data correctly. For detailed information about how the proxy works with a specific agent, see the appropriate Related pages.

### Proxy Basics

#### Single or Multi-Tenant

For the Node.js and PHP agents, the proxy can be single-tenant (one proxy per agent) or multi-tenant (multiple agents communicating through a single proxy), depending on the way the agents are configured. The default setup for these agents is single-tenant, but there are circumstances for which a multi-tenant proxy is required or desirable. The Apache Web Server and Python agents always communicate through a multi-tenant proxy when there are multiple agents on a machine.

#### Automatic or Manual Startup and Shutdown

Typically the proxy is automatically started when the application agent starts up, but in some cases, you need to launch it manually. These cases vary depending on both the particular agent and the application environment.

After the proxy is started (automatically or manually) it registers with the Controller and requests the agent configuration. The agent must receive the configuration from the Controller via the proxy before it can report metrics.

If the proxy was automatically started, the agent is supposed to shut down the proxy as part of its own cleanup procedures. If the proxy was manually started, it must be shut down manually.

#### Proxy Logs

If the proxy is running, you can check the proxy log to examine connection issues.

The proxy log files are named `proxy.<timestamp>.log`.

If the proxy is not running, examine the agent logs for the typical Java startup signature to see if the proxy started.

See the agent-specific documentation for the location of these logs:

- Node.js: Default is `/tmp/appd/logs`. See Install the Node.js Agent for details
Proxy Not Started Issues

The agent will not work if the proxy did not start.

Determine whether the proxy started by running this command:

```
ps aux|grep java
```

If the proxy is running, you should see `java` and `proxy` in the output, something like this:

```
/usr/lib/appdynamics-php5/proxy/jre/bin/java -server -Xmx120m
-serverpath
/usr/lib/appdynamics-php5/proxy/conf/logging/*:/usr/lib/appdynamics
/appdynamics-php5/proxy/*
-Djava.library.path=/usr/lib/appdynamics-php5/proxy/lib/tp
-Dappdynamics.agent.logs.dir=/usr/lib/appdynamics-php5/proxy/logs
-Dcomm=/tmp/ad-siJ4rp -DagentType=PHP_APP_AGENT
-Dappdynamics.agent.runtime.dir=/usr/lib/appdynamics-php5/proxy
com.appdynamics.ee.agent.proxy.kernel.Prox
```

If the proxy did not start, the most common reason is insufficient permissions.

- The agent installation directory, and its proxy control subdirectory, must be readable and executable by all and writable by the directory owner:

```
chmod -R 755 <agent_install_dir>
```

- The agent installation directory must be owned by the instrumented application user. Who this user is depends on the platform. It could be the Apache user, the python container user, the nginx user, the node.js process, etc.

```
chown -R <appuser>:<appsuser> <agent_install_dir>
```

If the proxy did not start, set these permissions and try again.

Proxy Connection Issues

If the proxy started but you don't have a connection to the Controller, you may be using the wrong controller information.

Examine the proxy log and verify that there are no typos in the Controller domain name or port and that SSL setting is enabled for an SSL connection. If there are mistakes, edit the Controller/port/SSL values in the configuration. These settings are on the first page of the Agent Download and Install Wizard or the agent settings (require statement for Node.js) if you installed the agent manually.

If you have verified that the Controller settings are correct and the proxy still does not connect to the Controller, telnet the SaaS Controller:
If you cannot access the Controller via telnet, examine the agent-registered messages in the logs that indicate why the proxy is unable to connect, such as the existence of a firewall or other obstacle at your site. Work with your administrator to see if you can resolve the issue.

If you can reach the Controller via telnet, there may be a problem on the Controller side.

**Node.js proxyless option**

To use the Node.js agent without the Java proxy, set the property `libagent: true` in your `require('appdynamics')` statement.

The following set of features will not be available when using the agent without a Java proxy:

- Backend configuration (only default backends are supported)
- Log file upload
- Object instance tracking
Dynamic Agent Proxy Logging

Related pages:
- Dynamic Language Agent Proxy

Proxy logs are created for the application agents that use the Dynamic Agent Proxy to communicate with the Controller. The PHP Agent, Node.js Agent (in debug mode), Python Agent, and the Web Server Agent generate logs for the dynamic agent proxy. The logs can be useful for determining why snapshots are not being taken and for diagnosing other communication issues with the Controller.

The proxy logs consist of a top-level file named `proxyCore Year_mon_day_hr_min.#.log` and files named `proxy Year_mon_day_hr_min.#.log`, where # is the log set. See Agent Log Files for information about how the logs are organized into sets that roll over. See the agent-specific logging documentation for information about the location of the proxy log associated with a specific agent.

Other logs associated with each proxy log are the following:
- bytecode Transformer log
- REST log
- Dynamic Services (used internally for debugging) log.

Each proxy log has a maximum size of five MB. Allowing for the possibility of several restarts and all the possible logs that could be generated (Proxy, BytecodeTransformer, REST and Dynamic Services), the maximum number of logs is 20 files (four log files times five sets). Per set, the maximum size is 100MB (five times 20).
Business Applications

As described in Overview of Application Monitoring, a business application serves as the top-level container in the AppDynamics APM model. Business applications contain a set of related business transactions and service endpoints, along with the infrastructure components and artifacts that interact to provide the services.

Organizing Business Applications

You can model your environment into one or more business applications. When you have multiple business applications within the same AppDynamics account, you can use the cross application flow map to view relationships between the different applications. If you have a more complex environment with multiple AppDynamics accounts, you can establish federated friendship between the accounts to view the federated cross application flow.

Since access permissions in the Controller UI can be assigned by business application, organizing business application by teams in your organization may make the most sense. Keep in mind that configuration settings such as Health Rules are scoped to a business application, avoid having more business applications than needed to ease configuration.

Permissions

Creating applications requires the Can Create Applications permission. Permission to view, edit, or delete applications can be set as part of the default application permissions for a custom role or for specific applications. See Application Permissions for more details.

Viewing Applications

The Applications page, accessible from the menu bar in the Controller UI, gives you a high-level view of the business applications in your environment. At a glance, you can see performance for the applications.

There are several views and sorting options in the page. To view relationships between business application (i.e., when a business transaction in one invokes a service in another), choose the Flow View icon.

Modifying Applications

You can create new business applications from the Applications page, but it's not necessary to create the business application manually. The Controller creates a new business application automatically the first time an app agent registers itself using a new business application name.
## Renaming Applications

You can rename applications by choosing the **Edit Application Properties** from the drop-down menu of an application.

If you change the name of an application, you also need to change it in the agent configuration at `controller-info.xml`. If you do not change the `controller-info.xml` application name field, agents will re-create the old application name and report to the old application.

## Deleting Applications

To delete an application:

Select the application in the Applications drop-down menu and choose **Actions > Delete**.

The Controller prompts you to acknowledge a list of related items that will be deleted including the following:

- Number of tiers and nodes
- Number of business transactions
- Browser RUM Apps
- Mobile Apps

## Application Performance Details

From the Applications page, click a listed application to configure and monitor the business application. In the business application instance Application Dashboard page, you can view:

- **Top Business Transactions**: The key performance indicators for the most expensive business transactions sorted by the various criteria, such as load, response time, errors and so on. Click View All in any of the panels in this tab to see a list displaying all the key performance indicators for the top business transactions in one panel.
- **Transaction Snapshots**: The transaction snapshots for the selected time range. From a transaction snapshot, you can drill down to the root cause of a performance problem.
- **Transaction Analysis**: Application performance over the selected time range as a graph. Use the graph to analyze the impact of different events on the application response time.
- **Machine Snapshots**: When an application has at least one .NET tier, this appears as a list of the machine snapshots for the selected time range. From a machine snapshot, you can drill down to view environmental conditions, such as running processes and memory usage for a particular machine. See [Machine Snapshots for .NET](#).
Flow Maps

Flow map overview

Flow maps present a dynamic visual representation of the components and activities of your monitored application environment. This topic describes how you can view and use flow maps.

Flow Map Overview

Flow maps show the tiers, nodes, message queues, and databases in the environment, and the business transactions that flow through them. The following figure shows a basic flow map for an e-commerce application. In the sample, three server tiers interact with databases and an Apache ActiveMQ message broker.

Types of Connections

Flow lines represent connections between components in the flow map. Solid lines indicate synchronous connections, while dashed lines indicate asynchronous connections.

Many modern frameworks use asynchronous patterns even if you do not explicitly call an asynchronous function or method. For example, your application code may employ an asynchronous call to a framework or an Object-relational mapping style API, but the framework itself invokes an asynchronous executor to handle the call. These types of asynchronous segments show up as a dotted line on the flow map. For more about asynchronous exit calls, see Trace Multithreaded Transactions for Java or Asynchronous Exit Points for .NET.
Request Times

The numbers above the flow lines indicate the calls made per minute to the tier and the average time taken for the request to be serviced; that is, the round-trip time for the request. The round-trip time includes time spent on the network, if applicable to your topology, and the time that the backend server or other process spends processing the request. The calls per minute for a given context, such as a tier, must be one or more for the flow map to display.

Performance Baselines

If performance baselines are set for transactions represented in the flow map, the flow lines use color to indicate the performance of the service represented by the flow line relative to the baseline. For example, a green flow line indicates that that response times in the time range do not differ significantly from the baseline. A yellow line indicates that response times are slower than the baseline. It takes some time for the Controller to establish baselines for a new installation. If there are no baselines for comparison, the flow lines are blue.

Live Data

By default, the flow map only shows the nodes receiving performance data to optimize the rendering of the flow map and to enable you to quickly view the active nodes. You can set a filter to view the nodes not receiving performance data or all nodes. Choosing to view the nodes not receiving performance data can help you troubleshoot node issues.

If the Controller detects that a flow map is taking a long time to load, it does not load the flow map automatically. In this case, you can click Show Flow Map to display the flow map if you choose.

Types of Flow Maps

Flow maps appear in several of the built-in dashboards in the UI, and show different information depending upon the context in which they appear:

- **All Applications flow map** shows all cross-application flows. When one application makes exit calls to another, the flow map shows cross-application flow within the monitored environment. See Cross Application Flow
- **Application flow maps** show the topology and activities within an application. It displays metric values across all business transactions in the application for the selected time range. For example, the application flow map displays calls per minute; average response time for calls made to databases and remote services; and business transaction errors per minute. These metrics are based on all calls made from a specific tier to a database or remote service across all business transactions.
- **Tier and node flow maps** display these metric values across all business transactions for the subset of the application flow related to the selected tier or node.
- **Business transaction flow maps** show the activity for a business transaction. The START label indicates the tier where the transaction starts (the originating tier). The business transaction flow map shows metrics that are calculated based on all executions of the business transaction during the selected time range.
- **Snapshot flow maps** illustrate the metrics associated with a single snapshot. The metrics values shown in the map are specific to a particular execution of the transaction.

AppDynamics shows cross-application flow on all flow maps where appropriate. For example, a tier flow map shows correlation when there are exit calls from the tier to another instrumented application. For more information, see 'Business Applications' on Overview of Application Monitoring.
What an item of information in a flow map represents can differ based on the context of the flow map. For example, for an application flow map, the average response time for calls to a database indicates the average for all calls to the database in the time range.

On the other hand, for a business transaction flow map, the average response time for the database represents the average time for calls to that database for each execution of the business transaction. If a business transaction makes multiple calls to a database, this number is the average response time for all calls to the database per transaction execution. If a business transaction makes two calls to a database, therefore, the value shown for the database connection in the business transaction flow map is likely to be twice the value shown for calls to the same database in the application flow map.

Interacting with Flow Maps

On flow maps you can:

- Click on items in the flow map to see key performance indicators in an informational popup. For some components of the flow map, like tiers, the popup displays additional details.
- Change the Time Range setting to have the flow map represent the activity of the system within the selected time frame.
- For SaaS Controllers, note that the flow maps show a maximum of the last 60 minutes of data, even if the time range in the UI is set to a greater range. This only applies to flow maps; the data in other graphs on the dashboard represent the selected time range.
- Click Legend to learn more about how flow maps represent data.
- Use a filter to view tiers and nodes receiving performance data, not receiving performance data, or all nodes.

- Drag and drop items to rearrange the flow map layout or use automatic arrangement options using the controls at the top right of the flow map:

  Use the controls to view the mapped components as a list, auto arrange the flow map—in which components are arranged for the fewest crossing flows—maximize the view, and more.

Managing Flow Maps

In a large scale deployment, the flow map may show hundreds of monitored nodes, not all of which may be of interest to specific users or teams. You can create flow maps that are targeted to specific areas of interest by creating custom flow maps.

The custom flow map can be configured to show only certain tiers or those based on performance thresholds, for example:

- Only tiers from where the load exceeds fifty calls per minute and the average response time exceed 10,000 ms
- Only backends receiving at least 400 calls per minute and generating more than 10 errors per minute

To create, copy, or delete a flow map, click the flow map menu and click **Manage My Flow Maps**.
When you create a flow map, the new flow map inherits the context of the flow map in which it was created, whether created from an application, business transaction, tier, or node flow map. Customize your new flow map as described next.

**Customizing Flow Maps**

See Customize Flow Maps.
You can customize flow maps to show information based on performance criteria or to have specific tiers or databases and remote services visible.

You can customize a built-in flow map, but in most cases, you will likely want to customize a flow map you have created. To customize a flow map, choose Edit Current Flow from the flow map menu. You can make the flow map shared or private. When shared, your custom flow map appears in the flow map menu for all users.

Add ELBs to the Flow Map

AppDynamics users can now view Amazon-hosted entities and data in several AppDynamics products by connecting their Amazon account to AppDynamics. This topic explains how to connect and view Amazon CloudWatch Elastic Load Balancing (ELB) data on a Network Visibility flow map, and describes what metric data is reported. AppDynamics uses the Amazon CloudWatch API to obtain metrics in near real-time from your Amazon Web Services (AWS) resources and applications. This combination of AppDynamics and AWS cloud-native monitoring helps identify and diagnose a variety of issues from a single Controller.

Requirements

- AppDynamics SaaS Controller, version 4.5.13 or later.
- A Network Visibility license.
- An AWS Classic Load Balancer. See Getting Started with Elastic Load Balancing in the Amazon documentation. Classic Load Balancer is intended for applications that were built within the EC2-Classic network.
- An AWS Identity and Access Management (IAM) role with permission to create a user and assign a role to that user.

Create an Amazon IAM User and Service Role

To integrate AppDynamics with Amazon CloudWatch, create an AWS Identity and Access Management (IAM) user within your AWS account. Once you create the IAM user, attach read-only policies to limit the permissions that this policy grants to the user or role. See Creating IAM Users (Console).

1. Navigate and sign in to the AWS Management Console.
2. Open the IAM console.
3. In the Identity and Access Management (IAM) left navigation pane, select Users.
4. Click Add user. If you are unable to add a user, see Access Management in the Amazon documentation.
5. Enter the User name: appD_monitoring_user or a user name of your choice.
6. Under Select AWS access type, check Programmatic access.
6. Click Next: Permissions.
7. Under Set permissions, select Attach existing policies directly
8. Choose an appropriate policy for the **appD_monitoring_user** monitoring account. We recommend `ReadOnlyAccess` policies. You can also use custom policies specific to the active resource that you want to ingest Amazon CloudWatch metrics. See Access Management and Example IAM Identity-Based Policies in the Amazon documentation. In our example we chose these policies:
   - AmazonEC2ReadOnlyAccess
   - CloudWatchReadOnlyAccess
   - ElasticLoadBalancingReadOnly
9. Click Next: Tags (optional).
10. Click Next: Review.
12. Click **Create user**.

Add user

- **Success**
  
  You successfully created the users shown below. You can view and download user security credentials. You can also email users instructions for signing in to the AWS Management Console. This is the last time these credentials will be available to download. However, you can create new credentials at any time.
  
  Users with AWS Management Console access can sign-in at: https://appdynamics-npm.signin.aws.amazon.com/console

- **Success**

13. Note the **Access Key ID** and the **Secret access key** for the user. You’ll need them to add Amazon CloudWatch to your AppDynamics account.

If you navigate away from this window without capturing these keys, you won’t be able to access them again. You will have to create a new **appD_monitoring_user** monitoring account.

### Connect AppDynamics to Amazon CloudWatch

To connect your Amazon account to AppDynamics:

2. Sign in to your account.
3. Select the Controller that you want to connect to Amazon CloudWatch.
4. Navigate to **Home > Cloud Platform** tab.
5. Under **Connect a Cloud Platform**, select **Amazon CloudWatch**.
6. Enter your Amazon Credentials.
   - a. Enter a Connection Name.
   - b. Enter the **appD_monitoring_user** Access Key ID.
   - c. Enter the **appD_monitoring_user** Secret Access Key.
7. Click **Connect**. This initiates the AppDynamics data connection with Amazon CloudWatch.
The **Home > Cloud Platform > Integration Status** tab displays the connection status of your Amazon CloudWatch integration. After five minutes, refresh your browser to verify the connection. Confirm that the Integration Status of your AWS account shows success before navigating to the Network Dashboard.
View and Monitor Elastic Load Balancing Data

To view and analyze Amazon CloudWatch metrics in the Controller UI:

2. Sign in to your account.
3. Select the Controller that you want to connect to Amazon CloudWatch.
4. Go to Applications and select the desired application.
5. Click Network Dashboard. A flow map appears.
6. Click on an Amazon Elastic Load Balancer (ELB) icon. A dialog appears displaying Connections and Amazon CloudWatch tabs.

![Diagram](image.png)

7. Click the Amazon CloudWatch tab. The following metrics appear:
   - **Backend Connection Errors**: Count of backend connection errors per minute.
   - **Latency**: Average request latency per minute.
   - **Requests**: Average requests per minute.
8. Click See More Detail.
10. To modify the time interval, in the top right corner of the page, click Last 1 hour drop-down list, and choose your desired time range.
Edit an Elastic Load Balancing Connection

As an account administrator, to edit an existing connection:

2. Sign in to your account.
3. Select the Controller that you want to connect to Amazon CloudWatch.
4. Navigate to the Home > Cloud Platform tab. A list of connections is displayed.
5. Select the connection name that you want to edit.
6. Click the Edit (pencil) icon.
7. Connect.

It can take several minutes for your edits to be synced. Refresh your browser after five minutes to verify connection. Confirm that the Integration Status of your Amazon account shows success before navigating to the Network Dashboard.

Delete an Elastic Load Balancing Connection

As an account administrator, to delete an existing connection:

2. Sign in to your account.
3. Select the Controller that you want to connect to Amazon CloudWatch.
4. Navigate to the Home > Cloud Platform tab. A list of connections is displayed.
5. Select the connection name that you want to delete.
6. Click the Delete (trash can) icon. A dialog asks you to confirm that you understand the consequences of deleting the connection.
7. Click **Yes**. If the connection has any metrics, deleting the connection also deletes these metrics.

*Amazon Web Services, the AWS logo, AWS, and any other AWS Marks used in these materials are trademarks of Amazon.com, Inc. or its affiliates in the United States and/or other countries.*
In the AppDynamics model, business applications usually represent a complete application environment. However, it is possible that different business applications share services or infrastructure components.

The cross application flow map lets you see those dependencies and points of contact between business applications in the Controller. This topic describes the cross application flow map.

Permissions

To view the cross-application flow map, the user must have the account-level `View Business Flow` permission.

Accessing the Cross Application Flow Map

The Applications tab in the AppDynamics Controller UI provides several views of the business applications in your account: a card view, list view, and flow view. To access the cross application flow map, click the flow view link in the tab.

Using the Cross Application Flow Map

To see the cross application flow click the `View All Applications` link on the Controller UI Home page. The page appears as follows:

In the context of a given business application, the call to another application is considered an exit call. If the other application is an AppDynamics-instrumented business application, the call is shown as cross application flow. For an example, see the All Applications
Flow Map on Flow Maps. All users of one application can see correlated applications on the flow map. However, to drill into metrics and snapshots for a correlated application, a user must be a member of a role with view permissions to the correlated application. See Roles and Permissions

If your environment includes business applications from multiple accounts on one or more Controllers, you can view correlation between those applications with the federated cross application flow.

**Hiding Components or Business Application on the Flow Map**

You can hide backend components, applications, or queues in the right-clicking menu for the component on the cross-application flow map.

For applications monitored with the Java Agent, you can exclude a node from being resolved to a cross-application flow by setting the following app agent node property to false:

```plaintext
support-cross-app-correlation=false
```

This property must be set in the downstream application and prevents calls from external apps registering as cross-application flows.

You may choose to do this, for example, if you wanted the cross application call to be represented as a backend call in the calling applications flow map—and metric scheme—rather than as a call to another application. This ensures that metric data reflecting interactions with the tier in the external business application are retained as backend call metrics.

**Note**
The node property `support-cross-app-correlation=false` works only for the Java Agent.
This topic introduces federated cross application flow, a form of transaction correlation that works across business applications in different Controller accounts or even on different Controllers.

**About Federated Cross Application Flow**

Cross application flow shows the performance impact between business applications within a Controller account. However, some application environments require a larger field of view; you may need to see performance information across business applications in different Controller accounts.

These types of environments may include, for instance:

- Very large organizations that have deployed multiple controllers to accommodate many business applications
- Organizations that have set up different controllers based upon geographical location
- Organizations that are typically isolated from one another, such as a client and vendor with separate Controller accounts, but for which the business applications for one affects the business applications of the other

Federated cross application flow, or federation, describes the ability of AppDynamics to correlate business transactions across business applications in different accounts which may be on different Controllers. Accounts that trust one another for correlation between their business applications are called federated friends.

The diagram below illustrates federated cross application flow between the applications in Account A and Account B and between the applications in Account B and Account C:

The two applications in Account A participate in a standard cross application flow.

Federated cross application flow works with Java and .NET agents that are version 4.4 or later.
After you establish a federated friendship between two accounts, any user in either account can view business applications from the friend account in flow maps. An application from a friend account shows up in the flow map when:

- A tier from the current application makes an exit call to an application in the friend account and the friend account handles the incoming call as an entry point.
- A tier in from a friend account makes an exit call to the current application and the current application handles the incoming call as an entry point.

AppDynamics appends the account name to the application name when the application resides in a friend account. For example App\_ACME-Downstream-Services@ACMEEnterprise in the example below:

The color of the outer circle of the friend application on the flow map indicates transaction health while the inner circle represents the node health. These indicators only represent the transactions and nodes that participate in the cross application flow. Similarly, the tier count and node count inside the circles only account for tiers and nodes that participate in upstream or downstream events related to entry or exit calls from the current application, tier or node.

In the example below, App B shows up in the flow map for App A as 2 tiers and 5 nodes because only those tiers and node participate in the Checkout business transaction:

The flow map does not include Tier 3 from App B and its three nodes because the transaction from App A does not flow through them.
Access a Federated Friend Application in the Controller

You must have credentials to an account in order to view its applications in the Controller. To navigate to the Application Dashboard for an application in a friend account, you can double-click the application in a flow map.

AppDynamics prompts you to supply your credentials to the friend account unless you have enabled SAML single sign-on and you use the same SAML system for both accounts.
Configure Federated Friendship

About Federated Friendship

To enable applications in different Controller accounts to participate in federated cross application flow, you must first establish federated friendship between the accounts. Federated friendship establishes trust between different accounts on one or more controllers. This trust enables AppDynamics to correlate transaction data between the accounts.

Establishing federated friendship relies on two REST APIs, which are used in the following sample workflow:

- The first call, apikeyforfederation, creates an API key for federation on an account, ACMEOnline in this example:

  ![Diagram of creating API key for federation](image)

- The second call, establishmutualfriendship, provides the API key and other identifying information about the first account to the other account in the federated friendship, such as from ACMEOnline to ACMEEnterprise:
After you run the `establishmutualfriendship` API, the Controller creates a second API key for the friend account and passes it back to the original account using the connection information provided via the API. This establishes mutual trust between the accounts, as illustrated below between ACMEOnline and ACMEEnterprise:

The following instructions use cURL to illustrate the workflow, but cURL is not required to establish federated friendship. You can use scripts or your preferred client tool to invoke the API calls.

**Set up Federated Friendship**

**Before Starting**

Review the following items before you begin to set up a federated friendship:

- The tier making the outbound call in the upstream application and the tier receiving the inbound call in the downstream application must be either Java or .NET.
- The Java and .NET Agent versions must be 4.4 or later.
- You must have access to two accounts on one or more Controllers. For each account you need:
  - Account name
  - Account owner user name
  - Account owner password
  - Controller hostname
  - Controller port
For each account, you must have app agents installed and reporting to the Controller.
You need the URL and port for the Controller for each account.
If your accounts are on different Controllers, the two Controllers must be able to communicate with one another over HTTP or HTTPS. In the case of HTTPS, this means that each Controller must trust the SSL certificate of the other.
- You may need to use keytool to import the root certificate for the CA that signed one Controller certificate into the other trust store. The trust store is located at <Controller Home>/appserver/glassfish/domains/domain1/config/cacerts.jks. See Controller SSL and Certificates.
- Once that is complete, you must restart your Controller Appserver so that the trust stores can be reread.

Create an API Key for Federation

1. Choose one of the accounts to create an API key to use for federated friendship. You can do this on either one of the accounts. These instructions refer to this account as account1.
2. Call the apikeyforfederation REST API to create the API key for account1:

```bash
curl -v -u <account1-admin-user>@<account1-name>:<password> -H "Content-Type: application/json" -d '{}' -X POST http://<account1-controller-url>:<controller-port>/controller/rest/federation/apikeyforfederation
```

The account1-admin-user must be an administrator for the account where you are making the API call. See ‘Authentication’ on Using the Controller APIs.

For <account1-controller-url> and <controller-port>, provide the Controller connection information for account1.

Calling the apikeyforfederation REST API twice with the same name results in a 500 internal server error.

The example below creates a key in an account named ACMEOnline:

```bash
curl -v -u admin@ACMEOnline:mypassword -H "Content-Type: application/json" -d '{}" -X POST http://ACMEEnterprise.example.com:80/controller/rest/federation/apikeyforfederation
```

The API returns the API key and additional identifying information. For example:

```json
{
  "id": "6e60b0eb-778c-4211-8d46-b557addc44c2",
  "accountId": 2,
  "name": "fed_key_b24ed0e7-a9c6-41ad-9732-82afc097f936",
  "description": "Federation Key for account {ACMEOnline}'",
  "key": "NmU2MGIwZWItNzc4Yy00MjExLThkNDYtYjU1N2FkZWQ0NGMyOmU4ODVhY2JkMjNhThiYjk4NWUyZGQ0MGU3N2E4ODVlOGJiMjhmYWU=",
  "expiryDate": -1,
  "canExpire": false,
  "state": "enabled"
}
```

Use the value for "key" in the step to establish federated friendship.
Establish Federated Friendship

To complete the federated friendship, provide identifying information about the first account (account1), including the API key for federation, to the friend account. These instructions refer to the friend account as account2.

1. Create a text file named friend.json.
2. Edit friend.json and paste the following JSON template:

```json
{
    "controllerUrl": "http://<account2-controller-url>:<controller-port>",
    "friendAccountName": "<account1-account-name>",
    "friendAccountApiKey": "<account1-key>"
}
```

3. Edit the values as in friend.json as follows:
   - account2-controller-url and controller-port: Controller connection information for account2. This address must be accessible from the account1 Controller.
   - account1-controller-url and controller-port: The Controller connection information for the friend account where you created the API key. The address must be accessible from the account2 Controller. If both accounts are on the same controller, this may be the same value as account1-controller-url and controller-port.
   - account1-account-name: Enter the friend account name the same way you would enter it for the API or on the Controller login screen.
   - account1-key: Enter the value of key for account1 returned from the apikeyforfederation API call.

   For instance, to pass the friend information about ACMEOnline to the account ACMEEEnterprise:

```json
{
    "controllerUrl": "http://ACMEEnterprise.example.com:80",
    "friendAccountControllerUrl": "http://ACMEOnline.example.com:80",
    "friendAccountName": "ACMEOnline",
    "friendAccountApiKey": "NmU2MGIwZWItNzc4Yy00MjExLThkNDYtYjU1N2FkZGM0NGMyOmU4ODhY2JKMDVjNThlYjk4NWUyZGQ0MGU3N2E4ODV1OGJiMjhjYWU="
}
```

4. Call the establishmutualfriendship API against the account2 and pass the information about the friend account that you saved in the friend.json file:

```
curl -v -u <account2-admin-user@<account2-name>:<password> -H "Content-Type: application/json" -d @friend.json -X POST http://<account2-controller-host>:<controller-port>/controller/rest/federation/establishmutualfriendship
```

The account2-admin-user must be an administrator for the account where you are making the API call. See ‘Authentication’ on Using the Controller APIs.

For <account2-controller-url> and <controller-port>, provide the Controller connection information for account2.

If friend.json is not in the same directory where you execute the cURL command, you need to provide the path to the
4. JSON file. For example, to finalize the friendship between ACMEOnline and ACMEEnterprises:

```bash
curl -v -u admin@ACMEOnline:mypassword -H "Content-Type:application/json" -d @friend.json -X POST http://ACMEOnline.example.com:80/controller/rest/federation/establishmutualfriendship
```

When the API runs successfully, the Controller returns a 200 status. After a few minutes, the Controller begins to correlate the transaction data between the two accounts and the accounts show up in flow maps using Federated Cross Application Flow.

**Remove Federated Friendship**

Just as you can set up a federated friendship between business applications in different Controller accounts, you can unfriend and delete the federated friendship information. Removing a federated friendship disables federation between the two Controllers and clears all associated metadata and metrics.

To remove the federated friendship, you must provide identifying information about the local and friend accounts, including the API keys for federation. These instructions refer to the local account as account1 and the friend account as account2.

1. Choose the account you would like to initiate the removal of federated friendship. You can do this on either one of the accounts. These instructions refer to this account as account1.

2. Retrieve the account API keys for both accounts. If you do not already know them, you can use the showfriends REST API to retrieve the keys:

```bash
curl -v -u <account1-admin-user>@<account1-name>:<password> -H "Content-Type: application/json" -d '{}' -X GET http://<account1-controller-url>:<controller-port>/controller/rest/federation/showFriends/account/account1
```

The account1-admin-user must be an administrator for the account where you are making the API call. See "Authentication" on Using the Controller APIs. For <account1-controller-url> and <controller-port>, provide the Controller connection information for account1. The account1 is the AccountKey. It should be the GUID portion of the global account name on the Controller.

3. Use the mutual unfriend REST API to complete your request. The mutual unfriend request checks if the friendship configuration is consistent and provisioned in the local and friend Controllers before going through with the deletion process. If an error is found during the check, it will return an error without impacting the existing settings.

```bash
curl -v -u <account2-admin-user>@<account2-name>:<password> -X DELETE http://<account2-controller-host>:<controller-port>/controller/rest/federation/mutualfriendship?accountKey={account1}&friendAccountKey={account2}
```

For <account2-controller-url> and <controller-port>, provide the Controller connection information for account2.

For example, to remove the friendship between ACMEOnline and ACMEEnterprises using the mutual unfriend REST API:
```
curl -v -u admin@ACMEOnline:mypassword -X DELETE  
http://ACMEOnline.example.com:80/controller/rest/federation/mutualfriendship?accountKey=fab3653a-75a4-4eaa-alac-900d017118ea  
&friendAccountKey=b2af3573-2aa2-406c-9daf-7948fedad1fe'
```

When the API runs successfully, the Controller returns a 202 status. It may take up to 15 minutes for your request to completely remove the friendship between the two accounts, and the associated metadata and metrics.
Federated Friendship API Reference

The REST APIs used for establishing and removing federated friendship follow the pattern documented in Using the Controller APIs. This page lists and describes the APIs. For an overview of configuration federated friendships, see Configure Federated Friendship.

Create API Key for Federation

Creates an API key for an account. A second account (friend account) must use the API key to establish a level of trust called federated friendship between itself and the account.

Format

POST /controller/rest/federation/apikeyforfederation

POST Data

```
{
  "apiKeyName": "<key name>
}
```

<table>
<thead>
<tr>
<th>Property</th>
<th>Value Description</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>apiKeyName</td>
<td>Arbitrary name for the API key. If you post an empty JSON object, the Controller automatically assigns a name for the API key.</td>
<td>No</td>
</tr>
</tbody>
</table>

Response Format

```
{
  "id": string,
  "accountId": int,
  "name": string,
  "description": string,
  "key": string,
  "expiryDate": int,
  "canExpire": bool,
  "state": string
}
```

<table>
<thead>
<tr>
<th>Property</th>
<th>Value Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>Unique ID for the API key.</td>
</tr>
<tr>
<td>accountId</td>
<td>The account to which the API key belongs.</td>
</tr>
<tr>
<td>name</td>
<td>Name for the key.</td>
</tr>
<tr>
<td>description</td>
<td>Human readable description of the key.</td>
</tr>
<tr>
<td>key</td>
<td>The key itself. Required to establish federated friendship.</td>
</tr>
<tr>
<td>expiryDate</td>
<td>Date the key expires, if any. A value of -1 means the key does not expire. Will always show -1 by default.</td>
</tr>
<tr>
<td>canExpire</td>
<td>Indicates whether the key can expire. Will always show &quot;false&quot; by default.</td>
</tr>
</tbody>
</table>
Establish Mutual Friendship

Provides the Controller with data it needs to establish federated friendship between two accounts. Create an API key for federation on the current account.

Format

POST /controller/rest/federation/establishmutualfriendship

POST Data

```
{
  "accountName": string,
  "controllerUrl": string,
  "friendAccountName": string,
  "friendAccountApiKey": string,
  "friendAccountControllerUrl": string
}
```

<table>
<thead>
<tr>
<th>Property</th>
<th>Value Description</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>accountName</td>
<td>Name of the account where you are posting the API. If empty, assumes the currently logged in user.</td>
<td>NO</td>
</tr>
<tr>
<td>controllerUrl</td>
<td>Controller URL for the account. If empty assumes the default Controller URL for the account. The friend account must be able to access the account at this URL. If you have enabled SSL, the certificate must be valid and trusted for this URL.</td>
<td>NO</td>
</tr>
<tr>
<td>friendAccountName</td>
<td>The name of the friend account with which you are establishing trust.</td>
<td>YES</td>
</tr>
<tr>
<td>friendAccountApiKey</td>
<td>API key for the friend account with which you are establishing trust. The value of key from the apikeyforfederation API.</td>
<td>YES</td>
</tr>
<tr>
<td>friendAccountControllerUrl</td>
<td>Controller URL where the current account can access the friend account. If you have enabled SSL, the certificate must be valid and trusted for this URL.</td>
<td>YES</td>
</tr>
</tbody>
</table>

Response Format

Returns standard HTTP codes. 200 indicates the call executed successfully.

Retrieve API Key for Federation

Gets the API keys for friendship accounts. The API keys can then be used to remove mutual friendships.

Format

GET /controller/rest/federation/showFriends/account/<AccountKey>

Response Format
{  
"id": string,  
"version": string,  
"configKey": string,  
"friendAccountKey": string,  
"friendApiKey": string,  
"friendControllerHost": string,  
"friendControllerPort": string,  
"friendControllerProtocol": string,  
"accountKey": string,  
"lastUpdateTimeStamp": int,  
"markedForDeletion": bool  
}

<table>
<thead>
<tr>
<th>Property</th>
<th>Value description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>Unique ID for the API key.</td>
</tr>
<tr>
<td>version</td>
<td>Version of the federated friendship.</td>
</tr>
<tr>
<td>configKey</td>
<td>The friendship configuration key.</td>
</tr>
<tr>
<td>friendAccountKey</td>
<td>Account key for the friend account with which you have established trust.</td>
</tr>
<tr>
<td>friendApiKey</td>
<td>API key for the friend account with which you have established trust.</td>
</tr>
<tr>
<td>friendControllerHost</td>
<td>Controller host for the friend account with which you have established trust.</td>
</tr>
<tr>
<td>friendControllerPort</td>
<td>Controller port for the friend account with which you have established trust.</td>
</tr>
<tr>
<td>friendControllerProtocol</td>
<td>Controller protocol for the friend account with which you have established trust.</td>
</tr>
<tr>
<td>accountKey</td>
<td>Account key for the current account.</td>
</tr>
<tr>
<td>lastUpdateTimeStamp</td>
<td>The last time the friendship was updated.</td>
</tr>
<tr>
<td>markedForDeletion</td>
<td>Indicates whether the friendship is scheduled for deletion.</td>
</tr>
</tbody>
</table>

**Remove Mutual Friendship**

Provides the Controller with data it needs to remove federated friendship between two accounts. You can use the mutual unfriend API to complete your request. It requires account keys for both the local and friend Controller accounts.

**Format**

DELETE  
/controller/rest/federation/mutualfriendship?accountKey={LocalAccount}&friendAccountKey={FriendAccount}

**Response Format**

Returns standard HTTP codes. 204 indicates the call executed successfully. 404 indicates the friendship pair is invalid.
Export and Import Business Application Settings

On this page:
- What Settings Are Exported?
- Export a Business Application
- Import a Business Application

This topic describes how to export and import the application configuration in the Controller. Application export/import allows you to back up just a single application configuration in the Controller, as opposed to the entire Controller instance, as described in Controller Data Backup and Restore.

For information on exporting other settings from the Controller, see Configuration Import and Export API. For JMX configurations, see Configure JMX Metrics from MBeans.

What Settings Are Exported?

You can export a business application configuration and import it as a new business application. The two applications can be on the same or different Controllers. If on different Controllers they must be using the same major version of AppDynamics.

The export is in the form of an XML file that includes:

- Snapshot collection settings
- Call graph settings
- Error configuration
- Stall configuration and Business Transaction thresholds
- HTTP and SQL Data gatherer settings
- Tier and Node definitions
- Custom entry point configurations for Business Transactions
- Metric baselines
- Information point configurations

The configuration information does not include data such as Events, Health Rule Violations, and metrics. Nor does it include configuration artifacts not related to the business application, such as users, dashboards, policies, database, and remote services, or schedules.

Export a Business Application

To export the application, from the Applications home page, choose Export Application from the action menu for the business application you want to export.
Your browser downloads the XML-formatted configuration file for the application.

**Import a Business Application**

To import an application configuration, you must be logged in as a user with Administrator user role privileges. You also need access to a previously exported configuration file.

To import the configuration, follow these steps:

1. From the Applications page, click Actions > Import.
2. Select the XML file that contains the configuration you want to import. You can choose to import into an existing business application or create a new one.
3. Choose the configuration settings you want to import. Options include configuration settings for app agents, data collectors, health rules, MDS (scoped instrumentation settings), and more.
4. Click OK to start importing. The import process first validates the file. As it proceeds, it shows the status for each type of artifact in the Import Application dialog.
5. When done, close the Import Application dialog.

Application import does not work over HTTPS if the Controller uses a self-signed SSL certificate. Use a trusted, CA-signed certificate instead. See Controller SSL and Certificates
Business Transactions

On this page:
- View Business Transactions
- Transaction Entry Points and Exit Points
- Sample Business Transaction
- Refine Business Transactions
- Business Transaction Limits
- Correlate Business Transaction Logs

Related pages:
- Organize Business Transactions
- Transaction Detection Rules
- Service Endpoints

In the AppDynamics application model, a business transaction represents the end-to-end, cross-tier processing path used to fulfill a request for a service provided by the application. This topic introduces and describes business transactions.

View Business Transactions

To view business transactions for a business application, click Business Transactions in the application navigation tree. The business transaction list shows key metrics for business transactions for the selected time range.

Only business transactions that have performance data in the selected time range appear in the list by default. You can show inactive business transactions for the time range by modifying the filter view options.

Other ways to modify the default view include showing transactions that belong to business transaction groups or transactions that exceed a configurable average response time. You can also choose which performance metrics appear for business transactions in the list from the View Options menu.

To see the actions you can perform on business transactions, open the More Actions menu. Actions include viewing health rule violations, configuring thresholds, renaming business transactions, grouping business transactions, starting a diagnostic session for the transaction, and classifying a business transaction as a background task.

Transaction Entry Points and Exit Points

When you install an app agent, the agent detects incoming calls and registers transactions based on the default transaction detection rules. Automatic detection rules describe entry points for transactions based on supported frameworks.

Usually, more than one tier participates in the processing of a transaction. A request to the originating tier may invoke services on:
- Another instrumented tier, called a downstream tier.
- Remote services that are not instrumented.

Outbound requests from an instrumented application tier are called exit points. Downstream tiers may, in turn, have exit points that invoke other services or backend requests.

App agents tag exit point calls with metadata describing the existing transaction. When an agent on a downstream tier detects an entry point that includes transaction metadata from another AppDynamics app agent, it treats the entry point as a continuation of the transaction initiated on the upstream tier. This linking of upstream exit points to downstream entry points is called correlation. Correlation maintains the client request context as it is processed by various tiers in your business application.

Sample Business Transaction

Consider, for example, the fictional ACME Online application. The application exposes a checkout service at http://acmeonline.example.com/checkout. A user request to the service triggers the following distributed processing flow and actions:
1. The business transaction entry point at the originating tier is `/checkout` URI, which is mapped to a Servlet called CheckoutServlet.

2. The request results in the originating tier invoking the createOrder method on a downstream tier, the ECommerce-Services server.

3. The inventory tier application calls a backend database, which is an exit point for the business transaction. The request context is maintained across tiers, including calls to backend tiers.

4. Any user request on the entry point is similarly categorized as this business transaction, the Checkout business transaction.

To enable detection of all the components in a distributed business transaction, downstream agents must be at the same AppDynamics release or newer than upstream agents. This is true whether the tiers are all built on the same platform—for example all Java—or multiple platforms—a Node.js tier calling a Java tier, calling a .NET tier and so on.

**Refine Business Transactions**

While the default rules can go a long way towards getting you a useful list of business transactions to track, an important part of implementing AppDynamics is verifying and refining the business transactions used to monitor your application. The business transaction you are monitoring should reflect those operations that are important to your application and business. It is important to consider the **limits on business transactions**, and apply your refinements accordingly.

Refining your business transaction list requires a solid understanding of the important business processes in your environment. Identify the 5 to 20 most important operations in the application. These are the key operations that must work well for the application to be successful.

Important services can be indicated by the number of calls or calls per minute received by the business transactions generated for the services. You can refine the list of transactions you want to monitor by locking down critical transactions and enabling automatic cleanup of stale transactions. For the Java and .NET environments, you can use interactive Live Preview tools to help identify important transactions.

You can add business transactions manually from a virtual business transaction called All Other Traffic, which is populated with transactions once the business transaction registration limits are reached, as described below.

To customize the business transaction list, you can use either of these approaches:

- You can modify existing business transactions by grouping, renaming, or removing the business transactions. Most of these operations are available from the business transaction list. Use this approach to apply relatively minor, small scale changes to the current business transaction list. For more information, see Organize Business Transactions.
- You can affect how business transactions are created by modifying the automatic discovery rules. You can modify rules to similarly achieve business transaction grouping and naming, and to exclude transactions. Discovery rules also enable you to define new entry points for business transactions. Discovery rule modification is a powerful mechanism for changing transaction discovery on a larger scale. For more information, see Transaction Detection Rules.

**Business Transaction Limits**

When reviewing and refining your business transaction limits, it is important to consider the business transaction limits for the Controller and app server agents. Business transaction limits prevent boundless growth of the business transaction list.

The default limits are:
• Business Application Limits: Each application is limited to 200 registered business transactions.
• App Server Agent Limits: Each agent is limited to 50 registered business transactions.

There is no limit at the tier level.

Also note that the app agent limit applies to each app agent, not to the machine. If you have multiple app agents on a single machine, the machine the business transactions originating from the machine could be up to the number of agents times 50.

**Correlate Business Transaction Logs**

For those times when tracing application code doesn’t provide enough clues to track down the cause of a problem, AppDynamics provides visibility into the transaction logs that can be correlated to specific business transaction requests. Log correlation visibility requires a license for both Transaction Analytics and Log Analytics. See *Business Transaction and Log Correlation*.
Organize Business Transactions

On this page:
- Rename Business Transactions
- Group Business Transactions
- Delete or Exclude Business Transactions
- Manage Business Transaction Detection
- All Other Traffic Business Transactions
- Lock Down Business Transactions

Related pages:
- Transaction Detection Rules
- Business Transactions

Organize and manage registered business transactions on the business transaction list to ensure that you are monitoring the most critical transactions. You can rename transactions, delete transactions, exclude transactions, or create custom transaction detection rules.

Especially for busy applications with lots of transactions, identify the best strategy for maximizing visibility into business-critical operations. Consider the following strategies:

- Set transaction detection priorities and delete old transactions to ensure that AppDynamics instruments your most important transactions.
- Combine multiple transactions into rules and exclude unimportant transactions to ensure that you stay within the business transaction limits.
- Rename transactions to informative names and group together similar transactions so that your metrics are easier to read.

Rename Business Transactions

AppDynamics names the business transactions it discovers based on the default naming scheme for the application type of the entry point for the transaction.

You can change the name for the business transaction by selecting the business transaction and choosing Rename from the More Actions menu. This change only affects the label for the business transaction in the user interface. You can also modify the default naming scheme for subsequently discovered business transactions, as described in the discussion on naming in Transaction Detection Rules.

By default, AppDynamics uniquely names business transactions according to the tier name and entry point type, along with an internal name. When you rename a business transaction, ensure its new name is unique; otherwise, there will be multiple business transactions reporting metrics under the same name.

Group Business Transactions

A transaction group lets you collect multiple related transactions into a single organizational unit in the AppDynamics model of your environment. For example, you may wish to group business transactions all of which have entry points in a specific WAR file. Groups could also be used to collect transactions of a specific organization, region, or category.

After grouping transactions, you can see aggregate metrics for the group. Metrics for individual transactions that make up the group remain available.

Grouping transactions does not affect the overall count of registered business transactions. To manage your transaction count, see Custom Match Rules.

To create a business transaction group:

1. Select the transactions you want to group from the Business Transactions list. Use the control or shift keys to multi-select business transactions.
2. Right-click the selected transactions and choose Create Group.
3. Give the group a name, verify the transactions you selected, and click Create Group.
3. Delete or Exclude Business Transactions

Excluding and deleting business transactions reduce the business transaction registration count. You can choose to exclude a short-lived business transaction temporarily or delete a stale business transaction permanently.

**Exclude a Business Transaction**

In effect, excluding the transaction disables the business transaction for metric processing purposes; it works even if the processing path that the business transaction represents remains active in the monitored application. Both serve to reduce the business transaction count considered against transaction registration limits.

Excluding a business transaction in the business transaction list is not to be confused with using custom exclude rules to control business transaction detection. While the two approaches achieve similar goals, custom exclude rules operate at the transactions detection point and are most useful for preventing discovery and registration of a range or multiple ranges of business transactions at a time. See Custom Match Rules for more information.

To exclude a business transaction:

1. Select your transaction from the list of business transactions.
2. Right-click the selection and click Exclude Transactions.
3. On the Exclude Business Transactions page, select the transaction and click Exclude Transactions.

**Restore an Excluded Business Transaction**

When you exclude a business transaction, the accumulated metrics for the transaction along with its underlying configuration is retained. You can restore the excluded transaction, if required.

2. From the Excluded Business Transactions page, select your transaction, and click Unexclude Selected.

**Delete a Business Transaction**
When you delete a business transaction from the list, the accumulated transaction metrics are removed. However, the deleted transaction is rediscovered if the corresponding function in the application is used. To delete a business transaction permanently, modify the discovery rules after deleting a transaction.

A business transaction can be considered stale in the following scenarios:

- You have decommissioned an application, and you want to delete the historical data associated with business transactions that made up the application.
- You have changed business transaction discovery rules to either eliminate or reorganize business transactions and need to remove unwanted or otherwise obsolete business transactions.

To delete a business transaction:

1. Select your transaction from the list of business transactions.
2. Right-click the selection and click **Delete Transactions**.
3. On the Delete Business Transactions page, select the transaction and click **Delete Business Transactions**.

**Automatic Cleanup of Stale Business Transactions**

Automatic cleanup keeps less active transactions from cluttering your business transaction list. Enabling automatic cleanup deletes business transactions that are flagged as stale. You can set the activity threshold for a business transaction after which the transaction is considered stale.

To configure automatic cleanup threshold:

1. From the left pane, click **Configuration > Instrumentation > Transaction Detection > More**.
2. Under Business Transaction Automatic Cleanup, check **Enable Business Transaction Automatic Cleanup**.
3. Update the automatic cleanup threshold that qualifies a business transaction as stale:
   a. Specify your threshold time in, **Monitor Business transactions for ___ minutes since creation**.
   b. Specify the number of calls in, **Remove Business transactions that have less than or equal to ___ calls**.

Since creation, a business transaction is monitored for 15 minutes when automatic cleanup is enabled with default settings. If there are no calls reported within the first 15 minutes since the business transaction is registered, the business transaction is qualified as short-lived and the automatic cleanup deletes that business transaction. Automatic cleanup keeps a business transaction intact if it is reporting data for days.

However, automatic cleanup ignores the business transactions that are:

- Marked as Permanent
- Older than three days

**Business Transaction Purger**

There are Controller configurations for the business transaction purger:

- **max.bt.to.purge.per.execution**: the number of business transactions that can be purged when the purger executes once. The default is 1000, min is 0, and the max is 100000.
- **max.bt.to.purge.per.transaction**: in order to reduce db lock, the purger tries to purge business transactions in multiple transactions if there are too many business transactions to purge in each execution. This configuration specifies the number of business transactions that can be purged in each transaction. The default is 10, min is 0, and the max is 1000.

**Manage Business Transaction Detection**

If your system has hundreds of business transactions, you can leverage the rule system to fine-tune which transactions you want to detect and monitor. Creating rules gives you the exact transaction visibility that you want. For example, you can create rules to:

- Detect transactions that are not automatically detected
- Prioritize monitoring some transactions over others
- Stay under the transaction limit. When you combine multiple transactions in a rule, the individual transactions no longer count towards the transaction limit; the rule is counted as a single transaction.
- Exclude transactions that you are not interested in monitoring.

Suppose John is creating transaction detection rules for an ECommerce application. John discovers that the most critical transaction, `user-checkout`, is grouped into All Other Traffic, which obscures its visibility. To ensure that AppDynamics collects detailed metrics for `user-checkout`, John creates a rule that detects `user-checkout`, and sets the rule priority to a high value, 50. For less critical
transactions, John creates a rule that detects multiple transactions: createaccount, billing, and delivery-options. He sets the rule priority to a lower value, 15. Combining multiple transactions into one rule helps John stay within the transaction limit.

For more information, see Transaction Detection Rules.

All Other Traffic Business Transactions

As an organizational unit in the AppDynamics model, business transactions are meant to be used for the most important transactions in your environment. The Controller limits the number of business transactions (50 for a node and 200 for a business application). This limit helps control resource utilization on the Controller, and also helps you focus on the most important services performed by your application environment.

Once the business transaction registration limit is reached or transaction lockdown is enabled, newly detected transactions are not registered as business transactions, but they are monitored and reported by AppDynamics. The transactions are grouped to a virtual business transaction named All Other Traffic - tier_name. An All Other Traffic group exists for each tier on which the limit is reached.

All Other Traffic appears alongside other business transactions in the Business Transactions list. Like you can for any business transaction, you can view the dashboard and key performance metrics for All Other Traffic in the Metric Browser.

When refining business transactions, it’s likely that you’ll want to move certain transactions from the All Other Traffic bucket to make them first-class business transactions. You can keep within the limits by deleting or excluding other business transactions, such as those that have little or no load, by using a custom exclude rule. If the number of business transactions is under the limit and lockdown is enabled, you can promote a transaction from the list of All Other Traffic by registering it from the Traffic Details window.

To view incoming calls categorized in the All Other Traffic transaction:

1. Open the dashboard for the All Other Traffic business transaction.
2. Click the View Traffic Details link.

The Traffic Details window lists transaction entry points that were hit by incoming requests after the registration limits were exceeded or
after business transaction detection was locked down. The Business Transaction Name column contains auto-generated names for the transactions. The Call column shows the number of instances of the transaction and the Type column shows the entry point type.

If the Fetch more link appears, click it to see more calls. You retrieve up to 600 calls each time you click the link. If the Fetch more link does not appear, there are no more calls to retrieve for the selected time range.

The name used for All Other Traffic in the REST API is APPDYNAMICS_DEFAULT_TX. For an example, see Use the AppDynamics REST API.

Lock Down Business Transactions

A production implementation of AppDynamics usually involves an initial period of assessing, organizing and refining business transactions by customizing your default business transaction discovery rules.

Once you arrive at the set business transactions to monitor, you may want to lock down business transactions. Locking down business transactions prevents application changes, upgrades to the agent software, or other changes from affecting the number or selection of business transactions you monitor as the primary, first-class business transactions in the AppDynamics model of your environment.

With business transaction lockdown enabled, the Controller puts newly discovered transactions into the All Other Traffic transaction collection for the tier. You can promote a business transaction from the All Other Transactions list by registering it as a first-class business transaction manually.

Business transaction lockdown gives you a way to register business transactions individually, by manual selection. In certain scenarios, it may make sense to enable business transaction lockdown before discovery occurs and registering transactions manually. For example, this may be useful if your environment would otherwise generate a large number of business transactions and you only want to monitor a relative few as first class business transactions.

To enable business transaction lock down, select the Enable Business Transaction Lock down option in the application instrumentation settings page.

After locking down business transactions, you can promote a transaction manually by accessing the Traffic Details dialog from the All Other Transactions dashboard. In the dialog, select the transaction and click Register to promote the transaction. If the agent is at the business transaction registration limit, you will need to delete a registered business transaction before registering one from the All Other Traffic list.

If you enable automatic cleanup, business transactions that you lock down but that do not meet the activity threshold are still deleted. To override automatic cleanup for a business transaction, right-click the transaction and select Mark As Permanent.
After verifying and refining your business transaction scheme, you can focus on monitoring business transaction performance. The monitoring and troubleshooting tools within the AppDynamics Controller UI provide a business transaction oriented view of performance. For example, transaction snapshots tie together all the code call graphs across the monitored tiers that participated in processing a particular instance of a business transaction.

The business transaction list contains a high-level look at the performance of the business application by transaction. You can click a particular business transaction to access additional information on the business transaction, such as information on slow performing instances of this transaction, errors, and the transaction analysis view.

### Performance at a Glance

The transaction scorecard summarizes the performance of a business transaction at the application, tier, or node level within a specified time range. It covers the number and percentage of business transaction instances (calls) that are normal, slow, very slow, stalled or errors based on the configured thresholds for these criteria.

<table>
<thead>
<tr>
<th>Transaction Type</th>
<th>Percentage</th>
<th>Call Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>95.2%</td>
<td>3.9k</td>
</tr>
<tr>
<td>Slow</td>
<td>0.1%</td>
<td>6</td>
</tr>
<tr>
<td>Very Slow</td>
<td>1.7%</td>
<td>70</td>
</tr>
<tr>
<td>Stall</td>
<td>0.0%</td>
<td>1</td>
</tr>
<tr>
<td>Errors</td>
<td>2.9%</td>
<td>120</td>
</tr>
</tbody>
</table>

The call counts in the scorecards in different dashboards have different meanings:

- In the application dashboard, the business transaction health scorecard aggregates the metrics for all the completed business transactions in the application, including the default business transactions. The number of calls is based on the number of business transactions that are completed. The percentages for those calls are based on the average for all the completed business transactions in the application. For example, if the scorecard displays 3% for very slow transactions, on average 3% of the calls for completed business transactions in the application were slow.
- In a tier dashboard, the transaction scorecard aggregates the metrics for all the nodes in the tier. The number of calls is based on the number of requests that the tier handled, including continuing transactions, threads spawned, and so forth.
- A node dashboard is similar to the tier dashboard in that it is based on the number of requests the node handled. The transaction scorecard displays the metrics for the single node.
- In a business transaction dashboard, the transaction scorecard displays the metrics for the entry point of a business transaction.

### Transaction Score View

For a more detailed view of the performance of a transaction over time, navigate to the business transaction dashboard and click the Transaction Score tab. The graph in the tab shows the user experience for the selected business transaction, showing its performance relative to the performance thresholds as a bar chart.

For each time slot, the bar reflects the total calls, while the color segments—green, yellow and so on—indicate the relative number of those calls that were normal, slow, very slow, stall or errors.
The solid blue line in the graph indicates the average response time for the transaction over time, while the dotted line shows the baseline, if available.
A threshold is a boundary of acceptable or normal business transaction performance. AppDynamics provides default thresholds against which it compares the performance of every business transaction. Each transaction is classified into what are called a user experience: normal, slow, very slow, stall or error transaction.

The user experience reflects the performance of a transaction instance relative to the usual performance of the business transaction. AppDynamics makes some intelligent guesses and calculations based on existing traffic flow to establish the default user experience profiles.

You can refine the criteria by which the classification is applied for your own requirements, as described here.

How is a transaction marked slow, very slow, stall or error?

As soon as a transaction starts, the particular thread representing that transaction is monitored for a stall. Every five seconds, the in-flight transactions are evaluated to determine if they have met or exceeded the stall threshold.

If a transaction execution finishes before the stall threshold value, the execution time is first compared against the very slow threshold, then the slow threshold, and so on, and marked accordingly.

If a transaction hits the stall threshold (takes more than 300 deviations above the average for the last two hours or the set stall threshold, a stall transaction event is generated. Whether the transaction eventually finishes successfully or times out, it is considered a stall for performance monitoring purposes.

Static and Dynamic Thresholds

Thresholds can be based on a static value or dynamic value. A dynamic threshold is based on performance for a most recent period of time, the previous two hours, by default. A dynamic threshold can be specified using either a percentage deviation or a standard deviation measure.

To understand how a transaction is compared to this moving window, consider a two-hour moving average window from 11:00 AM to 1:00 PM. All transactions that come in during 1:00 PM and 1:01 PM are measured against the response time and standard deviation of this window. For the next minute, between 1:01 and 1:02 PM, the window moves to 11:01 AM to 1:01 PM. The moving average itself is calculated based on the exponential moving average formula.

Note that:

- Data for minutes where there is no load is not counted.
- The data is maintained locally on each node for the starting tier for a business transaction. The Controller does not hold any of this data.

Percentage Deviation Threshold

Percentage deviation defines a threshold based on the moving average of the requests over a certain interval. The default time interval is 2 hours. If the average response time for the last two hours is X milliseconds, and if the request takes the percentage deviation over X ms, the transaction violates threshold.

For example, a transaction occurs at 10:00:01 AM. The average of the requests between 8:00:00 and 10:00:00 is 100 ms and the slow threshold is defined as 20% over the threshold. If the transaction takes 120 ms, it will be considered a slow transaction.
**Standard Deviation Threshold**

Standard deviation defines a threshold based on the moving average over a certain interval. The default time interval is two hours. This means if the average response time for the last two hours is X milliseconds, and if the request takes the standard deviation over X ms, it violates the threshold.

To understand how a standard deviation threshold works, consider a moving average that is 1500 ms with a standard deviation of 100 ms. If you set the threshold to 3, as in the following example, it means that the threshold will be three times the standard deviation. In other words, a transaction that takes more than 1500 + (3*100) or 1800 ms will violate the threshold.

Alternatively, you can set the threshold to a static value or use a percentage of the average.

You can also disable stall detection. If you disable stall detection, a transaction that might qualify as a stall is reported as very slow if it completes. If it does not complete, nothing is reported for the incomplete transaction.

**Configure Thresholds**

You can configure thresholds by going to Configuration > Slow Transaction Thresholds. To do so, you will need Configure Business Transaction application permissions.

Thresholds apply to various levels, by application or business transaction, including background task transactions. You can also configure thresholds for diagnostic sessions.

You can specify thresholds for User Transactions—regular business transactions—or background transactions tab, depending on the type of entity for which you want to configure thresholds.

Thresholds for Browser Real User Monitoring (RUM) are configured separately. For information about these thresholds see Configure Browser RUM Performance Thresholds.

You can configure thresholds for slow, very slow, and stalled transactions. When a transaction exceeds a threshold, AppDynamics starts capturing snapshots of it. Because snapshots are not normally captured while performance is within normal range, the snapshots may not contain the full call graph for the transaction.
Dynamic Baselines

On this page:

- How Baselines Work
- Identify Trends
- Baseline Deviations in Health Rules
- View Baselines
- Configure the Time Period

Related pages:

- Health Rules

AppDynamics automatically calculates the baseline performance for your applications, that is, the prevailing performance characteristics of those applications. Once it establishes a baseline, it can detect anomalous conditions for your application.

Baselines appear in these contexts in AppDynamics:

- Flow map colorization
- Transaction score dashboards
- Metric graphs
- Health rules

How Baselines Work

AppDynamics calculates baselines on the fly by using the underlying hour data. A baseline can be calculated for every metric in the controller.

Baselines have two main variables:

- The time over which accumulated data should be considered in calculating the baseline.
- The hourly segment for which the baseline should be calculated. It can be:
  - Unsegmented, in which case all performance data over the time period is considered in calculating the baseline; or
  - Segmented, in which the hour of the day, day of the week, and so on, is considered.

Identify Trends

Performance expectations can differ between hours of the day, days of the week, or even months of the year. For example:

- A retail application may experience heavier traffic on the weekend than the rest of the week.
- A payroll application may experience higher load at the beginning and end of the month compared to the rest of the month.
A Customer Relationship Management (CRM) application may experience heavy load during business hours Monday through Friday, but relatively light traffic over the weekend.

To account for this variation, you can use a rolling time period as the baseline context. A rolling time period, or dynamic baseline, establishes the baseline against data from the current hour taken at a daily, weekly or monthly interval over the course of the time period.

For example, say you have a baseline that uses the weekly trend and with the time period configured for 90 days, one of the baselines in the default Controller configuration.

At a given day and time, say Monday 10:30 AM, the baseline in effect is one that considers only the data accumulated on the same hour and day of the week over the last 90 days. This is illustrated by the following diagram.

A monthly trend calculates the baseline from data accumulated at the same hour but only on the same day of the month. So, for example, on January 5th at 10:30 AM, the baseline is established based on data accumulated at the same hour on the 5th of each month for the prior year, 365 days:
Baselines are not available immediately upon startup. It takes time and application load for the AppDynamics platform to collect data and create its initial baselines. The time depends on the type of baseline being used, whether daily, weekly, monthly, or none. None requires several hours before a baseline is available, daily takes several days, weekly takes several weeks, and so on.

Baseline Deviations in Health Rules

A baseline deviation is the standard deviation from a baseline at a point in time, represented as an integer value. You can set health rule conditions based on baseline deviations. For example, you can configure a warning condition as 2 standard deviations from the baseline and a critical condition as four standard deviations from the baseline.

How is the baseline deviation calculated?

The Controller uses the following standard formula to calculate the standard deviation:

\[
\text{std dev} = \sqrt{\frac{(B - A^2/N)/N}{}},
\]

where
- \(A\) is the sum of the data values.
- \(B\) is the sum of the squared data values.
- \(N\) is the number of data values.

Statistics are not compared against baselines in the following scenarios:

- There are fewer than 20 calls per minute. You can configure the calls per minute threshold by setting the \text{cpm.baseline.comparison} property.
- The time range is greater than 2 hours.

View Baselines
You can view built-in baselines and add new ones from the Configuration > Baselines page.

The daily trend is the default baseline. This is the baseline used by health rules if another baseline is not specified during health rule creation.

You can choose another as the default by selecting the baseline in the baseline list and choosing Set as Default. However, note that changing the default baseline changes the actual baseline for all existing health rules that rely on the default—that is, that do not specify another baseline. Be aware of your existing baselines and health rule definitions before you select this option.

Configure the Time Period

When configuring a baseline, you set the trend time period and the trend. The base time period comes in two forms:

- **Fixed** time range: from some specific date and time to a second specific date and time. For example, if you have a release cycle at a specific time you might limit your data collection to that specific time.
- **Rolling** time range: in which the most recent X number of days is used. This is the more common choice.
Transaction Snapshots

AppDynamics monitors every execution of a business transaction in the instrumented environment, and the metrics reflect all such executions. However, for troubleshooting purposes, AppDynamics takes snapshots of specific instances of a transaction. A transaction snapshot gives you a cross-tier view of the processing flow for a single invocation of a transaction.

Call drill downs, where available, detail key information including slowest methods, errors, and remote service calls for the transaction execution on a tier. A drill down may include a partial or complete call graph. Call graphs reflect the code-level view of the processing of the business transaction on a particular tier.

This topic covers how to view transaction snapshot data to monitor and troubleshoot business transaction performance. To learn more about when AppDynamics takes transaction snapshots or how to configure transaction snapshot settings, see Transaction Snapshot Collection.

View Transaction Snapshots

You can access business transaction snapshots from the following locations in the AppDynamics Controller:

- Troubleshooting > Slow Response Times or Troubleshooting > Errors left navigation tree for a business application
- Transaction Snapshots tab on the Business Transaction Dashboard

Double-click a business transaction snapshot to display the snapshot viewer. The following screen capture and its accompanying table identify the metrics available in the transaction flow map:

<table>
<thead>
<tr>
<th>Callout</th>
<th>Metric Name</th>
<th>Explanation</th>
</tr>
</thead>
</table>

Copyright © AppDynamics 2012-2019
1. **Tier Response Time (ms)**
   - The total response time for the call as measured at the calling tier. This includes the processing time on the called tier as well as on any tiers and backends it calls in turn.

2. **Percentage of Time Spent (%)**
   - The time spent downstream processing at all downstream tiers and backends as measured by the calling tier and represented as a percentage of the entire execution lifespan of a business transaction. This metric does not include the processing time of asynchronous activities, if any.

3. **Asynchronous Activity Processing Time (ms)**
   - The processing time of all asynchronous activities at this tier. This metric does not contribute to the overall tier response time because the activity is asynchronous by nature. This metric is calculated by adding the execution times of all asynchronous activities at a tier and the time spent in communication between other tiers and backends as follows:

   \[
   \text{Asynchronous Activity Processing Time} = \text{Asynchronous-activity-1-processing-time} + \text{Asynchronous-activity-2-processing-time} + \ldots
   \]

   - In the Metric Browser, you can view the Average Async Processing Time metric, which shows the average of the async activity processing time over the selected time range.

4. **Execution Time (ms)**
   - Total time spent processing by the business transaction in all affected tiers and communication with other tiers and backends. This metric does not include the processing time of the asynchronous activities. However, in the case of Wait-for-Completion, the originating business transaction will take longer to process the request due to blocking and waiting for all the activities to complete before proceeding.

   - The formula for this metric is calculated by summing up the processing times of a Business Transaction at a particular Tier/communication between Tiers/Backends as follows:

   \[
   \text{Execution Time} = \text{Time-spent-processing-in-Tier-1} + \text{Time-spent-processing-in-Tier-2} + \text{Time-spent-communicating-with-Tier-2} + \ldots
   \]

The Potential Issues panel highlights slow methods and slow remote service help you investigate the root causes for performance issues. Click an item in the Potential Issues list to view the call in the call graph.

In the flow map for a business transaction snapshot, a tier with a **Drill Down** link indicates AppDynamics has taken a call graph for that tier.

🔒 To drill into a call graph for a correlated application in a transaction with cross application flow, you need view permissions to the destination business application.

If you use the Java Agent and monitor Oracle databases with Database Monitoring and you have configured snapshot correlation between the Java Agent and the Database Agent, the flow map may also include database drill downs.

The flow map or Overview is one of several views of the business transaction in the snapshot viewer. Other views include:

- **Slow Calls and Errors**, which presents information on the slowest database and remote service calls, slowest methods, and errors. You can gain further insight into these slow calls and errors either by viewing their details on by drilling down into their call graphs.
- **Waterfall View**, which shows the execution of an individual business transaction broken into execution segments. Each segment (shown as a blue bar) represents the time spent executing code on a particular thread within an instrumented application runtime, or the time spent waiting for responses from uninstrumented backends. Handoffs between segments are shown as solid lines for synchronous requests, and dotted for asynchronous requests. The waterfall view allows you to quickly determine which calls consumed the transaction time for a given snapshot. You can click on a segment to view the resource wait time for its business transaction.
• Segment List, which shows the various legs of the transaction in descending order of duration and give access to their snapshots and allows you to drill down into their details.

Call Drill Downs

A call drill down contains details for that business transaction execution on a particular tier. It takes you to the code-level information for the transaction.

The contents of a transaction snapshot containing asynchronous segments look slightly different if you access the snapshot via the Business Transaction view or via the App/Tier/Node views:

• Initially the Business Transaction view only displays the originating segments for the transaction. You have the option drill down into the asynchronous segments as desired.
• The App/Tier/Node views surface all the segments that are relative to that specific entity. Therefore you access one of these views, you can see all segments, originating and asynchronous.

Node Drill Down

The node drill down organizes diagnostic data among the following tabs:

• The Overview tab includes a problem summary, execution times, CPU time stamp, tier, node, process ID, thread name, etc.
• The Call Graphs tab lists call graphs showing the execution flow for the transaction on a given tier. For details, see Call Graphs.
• The Slow Calls and Errors tab lists all the slow method calls and calls that resulted in an error. You can use the Hot Spots slider to sort calls by execution time with the most expensive calls in the snapshot at the top.
• The Error Details tab exposes exception stack traces and HTTP error codes.
• The DB & Remote Service Calls tab shows all SQL query exit calls to databases and exit calls to other remote services such as web services, message queues or caching servers. See Database queries and batching for more information about how AppDynamics handles SQL exit calls.
• The Server tab displays graphs for hardware—CPU Memory, Disk IO, Network IO—Memory—Heap, Garbage Collection, Memory Pools—JMX, and more. If you have Server Visibility, you will have access to full performance details for the server hardware and operating system.
• For customers using Network Visibility, the Network tab shows charts related to the impact of the network on the transaction and other pertinent data. For information on network KPIs and troubleshooting see KPI Metrics in Network Dashboard and
Application Flow Map and KPI Metrics in Right-Click Dashboards.

- The Data Collectors tab shows pertinent application data for the transaction snapshot. For configuration options, see Data Collectors.

  HTTP Data: HTTP payloads contain basic data such as the URL and session ID, and additional data for Servlet entry points, Struts, JSF, Web Services, etc. You can use HTTP data collectors to specify which query parameter or cookie values should be captured in the transaction snapshot.

  Cookies: The snapshot can use cookie values to help identify the user who initiated the slow or error transaction.

  User Data: User data from any method executed during a transaction, including parameter values and return values, to add context to the transaction. You can use method invocation data collectors to specify the method and parameter index.

  In cases where an exit call is made just before a business transaction starts, exit call information can show up in this field, particularly if the transaction is marked as slow or having errors. Please note that sensitive information on the exit call may be shown in this situation.

- The More tab shows hows metrics for the node that deviate the most from the established baselines as Node Problems. It also shows all the Service Endpoints invoked during the snapshot and the Servlet URI and Process ID of the transaction.

Database queries and batching

AppDynamics normalizes SQL queries and by default does not display raw/bind values. You can configure SQL capture settings to monitor raw SQL data in the queries. Individual calls that take less than 1 second are not reported.

When returning data to a JDBC client, database management systems often return the results as a batched response. Each batch contains a subset of the total result set, with typically 10 records in each batch. The JDBC client retrieves a batch and iterates through the results. If the query is not satisfied, the JDBC client gets the next batch, and so on.

In the SQL query window, a number followed by an X in the Query column means that the query ran the number of times indicated within a batch. The value in the Count column indicates the number of times that the batch job executed.

Database Drilldown

Database drill-downs tell you the following about the transaction:

- Queries lists the queries consuming the most time in the database as top SQL statements and Stored Procedures. Comparing the query weights to other metrics such as SQL wait times may point you to SQL that requires tuning.
- Clients displays the hostname or IP addresses of the Top N clients using the database. A database client is any host that accesses the database instance.
- Sessions displays the Session ID of the Top N sessions using the database sorted by time spent.
- Schemas shows the names of the Top N busiest schemas on the database server.

Using the Transaction Snapshot List

You can view transaction snapshots generated in the UI time range from the Transaction Snapshots tab of the application, tier, node, or business transaction dashboards. From there you can:

- Compare Snapshots shows the performance of calls in two snapshots as a side-by-side comparison.
- Identify the most expensive calls / SQL statements in a group of Snapshots shows the calls that take the most time across the snapshots you have selected. You can select up to 30 snapshots.
- Find snapshots using the filter options.

The Controller purges transaction snapshots after two weeks by default, but you can configure the snapshot retention period. You can archive a snapshot to preserve it beyond the normal snapshot lifespan. For example, if you want to retain a snapshot associated with a particular problem for future analysis. To archive a snapshot, select it from the list and choose Actions > Archive.

To archive a snapshot, you need the Application level - Can create applications permission.

The file cabinet icon in the far right column indicates that the snapshot is an archive snapshot (🔒).

To display only archived snapshots in the snapshot list, filter the snapshot list and check Only Archived.
Call Graphs

On this page:
- Types of Call Graphs
- View Call Graphs
- SQL Query Exit Calls in Call Graphs
- Request Root Node
- Node.js Call Graphs
- Display Excluded Classes in Call Graphs
- Configure Instrumentation

Related pages:
- Call Graph Settings
- Data Collectors

A call graph in a transaction snapshot shows you business transaction processing information on a particular tier that participated on the business transaction. A call graph lists the methods in a call stack and provides information about each call.

A call graph can tell you general information such as the total execution time, the node name, the time stamp for the start of execution, and the unique identifier for the business transaction instance. Call graphs help you diagnose performance issues and optimize the flow of a complex business transaction.

Call graphs are also captured by the Node.js agent in process snapshots, which describe an instance of a CPU process on an instrumented Node.js node. See Event Loop Blocking in Node.js.

Types of Call Graphs

A call graph can be one of the following types:

- Full call graphs capture the entire call sequence for the business transaction invocation. In this case, call graphs exist for each monitored node involved in the processing of the business transaction. Periodically collected and diagnostic snapshots are always full call graphs.
- Partial call graphs represent the subset of the call sequence for processing a business transaction, typically from the point at which the transaction has been determined to be slow or have errors. Partial call graphs are identified as such in the transaction snapshot.

View Call Graphs

To view a call graph:

1. Open the dashboard for the business transaction for which you want to see a call graph and click the Transaction Snapshots tab.
2. In the transaction snapshot list, double click on a snapshot.
3. Click the Drill Down link for the tier on which you want to see a call graph. An asterisk next to the link means that multiple calls were made at this tier for the transaction, which a dialog offers as choices to drill down into.

In the window, the method execution sequence shows the names of the classes and methods that participated in processing the business transaction on this node, in the order in which the flow of control proceeded.

For each method, you can see the time spent processing and the line number in the source code, enabling you to pinpoint the location in code that could be affecting the performance of the transaction.

The call graph window includes UI controls for navigating large call graphs. For instance, you can use the filter field to have only a particular method or types of methods displayed. When you find a method of interest, you can set it as the root method in the tree to view only the portion of the call graph that occurs from that point forward.
The call graph displays exit call links for methods that make calls to outside applications and services such as database queries and web service calls. Exit calls that do not exceed a minimum time threshold may not be represented.

**SQL Query Exit Calls in Call Graphs**

Often times application code executes database calls in multiple phases. For example:

- Construct and execute a SQL query
- Iterate over the result set

For example:

```java
stmt = con.createStatement();
# Execute the query
ResultSet rs = stmt.executeQuery(query);
# Iterate through the results
while (rs.next()) {
    String PizzaType = rs.getString("PIZZA_TYPE");
    float price = rs.getFloat("PRICE");
}
```

The Java Agent represents JDBC SQL query executions and the iterations over the result set separately in the call graph:

- For the execute call, the call graph shows the SQL query and the execution time. You can access the exit call details via a database link in the External Call column.
- For the iteration over the shows a count of the number of iterations and the total time. You can access the exit call details via a result set iteration link in the External Call column.

The .NET Agent does not separate ADO.NET SQL query execution exit calls from iteration over the result set in the call graph.

The Python Agent and the Node.js Agent sometimes separate out iterations as separate exit calls based upon the database driver used to make the call.

AppDynamics does not always represent every database call as a separate exit call. Therefore you may see a greater number of database calls than the number of exit calls in the call graph.

**Minimum Capture Threshold Times for SQL Calls**

The query and result set iteration are both subject to the minimum capture threshold times for SQL calls, 10 ms by default. If the execution time for a query or result set iteration does not exceed the threshold it does not appear in the call graph. This means a given database interaction may be represented in a call graph in one of three ways depending on which parts of the operation took longer than 10 ms, if any:

- **Exit calls to remote services**
- **Execution time per method**
- **Method details: type, class name, method name and line number**
- **Transaction globally Unique Identifier (GUID)**
1. Only the query appears (callout 1 in the image below)
2. Only the ResultSet iteration appears (callout 2)
3. Both appear (callout 3)

<table>
<thead>
<tr>
<th>Time (ms)</th>
<th>Percent %</th>
<th>Exit Calls / Threads...</th>
</tr>
</thead>
<tbody>
<tr>
<td>12 ms (self)</td>
<td>42.9%</td>
<td>JDBC ResultSet Iteration JDBC</td>
</tr>
<tr>
<td>0 ms (self)</td>
<td>0%</td>
<td>JDBC</td>
</tr>
<tr>
<td>5 ms (self)</td>
<td>17.9%</td>
<td>JDBC ResultSet Iteration</td>
</tr>
<tr>
<td>0 ms (self)</td>
<td>0%</td>
<td>JDBC ResultSet Iteration</td>
</tr>
<tr>
<td>0 ms (self)</td>
<td>0%</td>
<td>JDBC ResultSet Iteration</td>
</tr>
</tbody>
</table>

Clicking the link takes you to more details for that SQL exit call or result set iteration.

**Maximum Number of SQL Queries in Snapshots**

The maximum number of SQL queries in snapshots is set to 500 by default. For Java Agents, you can increase the value of the `max-jdbc-calls-per-snapshot` node property if you do not see expected SQL queries in a call graph.

**Request Root Node**

The call graphs in transaction snapshots of the Node.js, PHP, and Python agents have an artificial root node named `{request}`.

For Python and PHP, the purpose of this node is to reconcile time differences between the total business transaction time and the sum of the observed call graph nodes, to act as a single entry point for the call graph, and to contain any exit calls which could not be attributed to any other nodes.

For Node.js, the `{request}` node in a business transaction call graph signals the availability of a process snapshot call graph that intersects with the transaction snapshot. If there is no `{request}` node, no process snapshot is available. For more information, see [Process Snapshots and Business Transaction Snapshots](#).

**Node.js Call Graphs**

When the Node.js agent captures a complete call graph, the call graph tab of the transaction snapshot displays the total execution total time of the transaction, including wait times, at the top of the call graph.

The execution time shown in the call graph pane is the sum of the times of the individual method calls inside the call graph. This value may be less than the total execution time because it does not include wait times between calls.

**Display Excluded Classes in Call Graphs**

The sequence of method invocations in a call graph may involve hundreds of classes. These classes include user application classes, framework classes (such as Tomcat or Websphere runtime classes), Java/J2EE core libraries, .NET Framework classes, and more.

Not all of those classes may be relevant to the type of troubleshooting you normally do. In most cases, your primary interests are the custom application classes you have built on top of the framework or language platform. Other classes may be needed, but only on rare occasions.

To make the call graph more readable and to avoid the overhead of processing the timing information for non-user-application classes, other classes are not shown in the call graph.

If packages (Java) or namespaces (.NET) have been excluded, you can click the `packages/namespaces have been excluded` link to view the packages, and optionally configure the classes to be included in call graphs.
Configure Instrumentation

You can right-click any item in a call graph and select Configure Instrumentation for this Class/Method.

The Configure Instrumentation window presents a drop-down menu from which you can select the type of configuration that you want to create for the method.
By default, AppDynamics retains transaction snapshots that it captures at regular intervals and when a transaction is detected to be performing abnormally. The snapshots may have full, partial, or no call graphs.

Diagnostic sessions let you trigger a timed session in which AppDynamics captures transaction snapshots for a business transaction at a more frequent interval that you set. In the session, AppDynamics captures full call graphs for the transactions.

Diagnostics sessions can also be triggered automatically by a health rule violation or in response to an abnormal amount of slow or error transactions. By default, if more than 10% of the requests in a minute are slow, AppDynamics starts a diagnostic session.

If the diagnostic session is triggered manually, the diagnostic session collects snapshots on all the nodes that the selected business transaction passes through. If the diagnostic session is triggered to start automatically, the diagnostics session collects snapshots on the triggering node.

### Triggering a Diagnostic Session Manually

To start a diagnostic session manually, you need the Start Diagnostic Sessions permission.

You can start a diagnostic session from the business transaction list. In the list, select the transactions and start the session from the More Actions menu.

To view the transaction snapshots gathered in the session, in the business transaction dashboard, choose Transaction Snapshots > Diagnostic Sessions.

### Configure Automatic Diagnostic Sessions for Slow and Error Transactions

You configure settings that trigger automatic diagnostic sessions from the transaction threshold configuration page. There you can configure the threshold settings that trigger a diagnostic session as well as the snapshot collection frequency and duration once the session is triggered.

For performance reasons, you want to limit the duration and frequency of diagnostic sessions to the minimum required time to obtain the maximum amount of information for troubleshooting purposes. To avoid continuous collection when there are ongoing performance problems, configure the wait period between sessions and increase the time as needed.

### Using Diagnostic Sessions for Asynchronous Activity

Diagnostic sessions are usually triggered based on the overall performance of a business transaction. However, the overall performance of the business transaction may not reflect the execution time of asynchronous activity. It’s possible that the originating transaction executes within normal bounds, while the asynchronous activity takes much longer.

To initiate diagnostic sessions for these scenarios, you can use health rules based on a performance metric (such as response time) of the asynchronous activity. Then create a policy that triggers a diagnostic session when the health rule violates. The general steps to do this are:

1. Create a custom health rule based on the asynchronous metric, such as average response time. The metrics for thread tasks are visible in the metric browser under the Thread Tasks node for transactions with asynchronous activity. Each thread task
has an individual node (usually its simple class name). Remember to select Custom as the type for the health rule.

2. Create a policy that is based on the baseline of the asynchronous metric of interest, for example, the average response time.
3. Configure the policy to trigger a diagnostic session on the affected business transaction.
KPI Analyzer

The AppDynamics KPI analyzer helps you to identify the root causes of poor application performance. By applying machine learning algorithms, the KPI analyzer recognizes performance anomalies based on the average response time key performance indicator (KPI) for a node. It isolates the metrics that are the most likely contributors to the poor performance and identifies the likely degree of impact on the KPI for each metric.

In this way, the analyzer makes troubleshooting quicker and easier by surfacing those metrics that are most likely to be related to the root cause for poor response times.

The KPI analyzer correlates metrics to performance issues using metric data from the observed node from the last 4 hours. It ranks the contributing metrics based upon likely impact using a contribution score. The contribution score indicates the degree of impact of the underlying metric on the KPI.

Enabling the KPI Analyzer

The KPI Analyzer is available on select SaaS Controllers only. If you are interested in using the feature, contact your AppDynamics representative.

Using the KPI Analyzer

Users can view KPI Analyzer information by drilling down into node information for a transaction snapshot.

The general workflow you follow to perform root cause analysis with the KPI Analyzer is as follows:

1. From a transaction snapshot list, find the transaction snapshot for the transaction to view:

2. In the snapshot viewer, drill down into the specific node that is presenting performance problems:
3. Click the KPI Analyzer tab.

The tab shows the KPI and contributing metrics as time series charts, as follows:

The tab is organized into two general areas:

- The top of the tab shows the key KPI, average response time, values over time.
- The metrics that are identified as most likely contributing to the performance are shown in ranked order below, allowing you to identify and resolve the root causes of the performance problems.
Transaction Snapshot Collection

AppDynamics takes transaction snapshots of select instances of a transaction to give you a cross-tier view of the processing flow for a single invocation of a transaction. This topic covers the scenarios that trigger AppDynamics to take a transaction snapshot and how to configure transaction snapshot settings. For information about the content of a transaction snapshot, see Transaction Snapshots.

Subject to the guidelines and limits described in Snapshot Retention Rules and Limits, AppDynamics takes transaction snapshots in these cases:

- The app agent determines the user experience for the business transaction to be slow or the transaction incurred an error.
- The app agent collects snapshots during periodic snapshot collection.
- The app agent collects snapshots during a diagnostic session.

Snapshot and Call Graph Retention Rules

For a given transaction instance, a call graph may be available for some tiers but not all. The following guidelines describe the rules for when the app agent captures transaction snapshots for the originating and downstream tiers in a transaction. They also describe the type of call graphs a snapshot might include depending on the cause for the snapshot. The guidelines apply to business transaction correlation as well as cross-application flow.

- Any tier (originating or continuing) takes a snapshot when it recognizes that it is experiencing slow, very slow, or stalled response times or has errors.
- An originating tier takes a transaction snapshot:
  - When the agent starts a diagnostic session on the originating tier because it has detected a pattern of performance problems. You can also manually start a diagnostic session from the Business Transaction Dashboard. For details see Diagnostic Sessions.
  - When the agent identifies slow, very slow, or stalled response times, or errors on the originating tier. These snapshots may have partial call graph information because they start at the time when the transaction slowed or experienced an error.
  - Based on the periodic collection schedule. By default, the agent captures one snapshot every 10 minutes.
- The downstream tier captures snapshots when the tier immediately upstream to it tells it to take a snapshot. An upstream tier might direct its downstream tier to take a snapshot under these circumstances:
  - The upstream tier is taking a snapshot for a diagnostic session.
  - The upstream tier is taking a snapshot based on the periodic collection schedule.

Within the guidelines, snapshot retention is also subject to snapshot retention limits, as described in the following section.

Transaction Snapshot Limits

Snapshot retention limits prevent excessive resource consumption at the node level:

- Originating transaction snapshots are limited to a maximum of 20 originating—5 concurrent—snapshots per node per minute.
- Continuing transaction snapshots are limited to a maximum of 200—100 concurrent—snapshots per node per minute.

AppDynamics applies snapshot retention limits to error transactions as well. As a result, not every error occurrence that is represented in an error count metric, for example, will have a corresponding snapshot. For error transactions, the following limits apply:

- For a single transaction, there is a maximum of two snapshots per minute.
- Across transactions, the maximum is limited to five snapshots per minute. (Specified by the node property max-error-snapshots-per-minute.)

Configure Snapshot Periodic Collection Frequency

By default, AppDynamics collects a snapshot every 10 minutes. You can modify this default in the Slow Transaction Thresholds configuration page. The value will apply to subsequently created business transactions, but if you check Apply to all Existing Business Transactions, all existing business transactions are affected by the change as well.

If you have a high load production environment, it is important that you do not use low values for snapshot collection, in other words, configure collection on a very frequent basis. When there are thousands or millions of requests per minute, collecting snapshots too frequently may result in many extra snapshots that are not highly useful. Either turn OFF the periodic snapshots and apply to all Business Transactions, or choose a very conservative (high) rate depending on the expected load. For example, if you have a high load on the application, choose every 1000th executions or every 20 minutes, depending on the load pattern. See Overview of Transaction Snapshots.
In the left navigation pane click **Configuration > Slow Transaction Thresholds.**
End-to-End Latency Performance

On this page:
- Before You Begin
- Monitor End-to-End Performance
- End-to-End Performance Metrics

Related pages:
- Asynchronous Transaction Demarcators
- Trace Multithreaded Transactions for Java

Works with:

Java .net

The response time metric for a business transaction can tell you a lot about the performance of the transaction. In asynchronous applications, however, it may not tell you everything.

Consider a method that spawns other threads, possibly on other tiers, and then immediately returns control to the calling thread. The return of control stops the clock on the transaction in terms of measuring response time, but meanwhile, the logical processing for the transaction continues.

End-to-end latency metrics can reflect the response times for such asynchronous transactions.

Before You Begin

In order to monitor end-to-end metrics in AppDynamics, you must first configure the endpoints for your asynchronous transactions. See Asynchronous Transaction Demarcators.

Monitor End-to-End Performance

After you enable end-to-end latency transaction monitoring, end to end metrics appear in the metric browser for individual nodes and the overall application. The business transaction flow map also shows the End to End Latency Transaction Scorecard.

The End to End Latency Transaction Scorecard shows health based on the end-to-end asynchronous transaction time. The scorecard criteria—normal, slow, very slow and stall—for asynchronous transactions are the same as for the standard transaction scorecard. See Transaction Thresholds for more information. Note that the Transaction Scorecard reflects the data for the originating business transaction only, not end-to-end transactions.
End-to-End Performance Metrics

The performance metrics for end-to-end message transactions, which can be found in the overall application metric tree in the metric browser, are:

- **Average End to End Latency**: The average time in milliseconds spent processing end to end message transactions over the selected time frame.
- **Number of Slow End to End Messages**: The number of end to end transactions that exceeded the slow threshold for the business transaction over the selected time frame. The slow user experience is based on the slow transactions thresholds applicable to the general business transaction. That is, separate thresholds do not apply to end-to-end latency.
- **Number of Very Slow End to End Latency Times**: The number of end to end transactions that exceeded the very slow threshold for the business transaction over the selected time frame. The very slow user experience is based on the very slow transactions thresholds applicable to the general business transaction. (That is, separate thresholds do not apply to end to end latency.)

If you enable *End to End Latency Time (ms)* view option for the Business Transactions list, the waterfall view shows the end-to-end transaction performance for asynchronous transactions.
Monitor Background Tasks

A background task is a processing activity such as an administrative task or internal process that you want to monitor separately from primary business transactions, such as a cron job that backs up a server.

While the process may be important to your environment, it is likely that you do not want to monitor them in the same way as primary business transactions. Background task performance metrics do not count toward the average response time of the tier or node.

During business transaction discovery, AppDynamics attempts to identify processes in the environment that are background tasks and classify them accordingly. You can modify the default configuration based on your requirements and manually change business transactions to background tasks, as described here.

Viewing Background Tasks

Although background tasks are monitored separately, they are displayed along with the other business transactions in the Business Transactions list. Background tasks are identified in the list by the background task icon ( )

The following information is available on background tasks:

- The Business Transactions list shows currently executing background tasks and their statistics.
- The Metric Browser shows the response time for each background task.
- Transaction snapshots are generated for each execution of a background task. However, if a particular job runs frequently, the snapshot may not capture all details for each execution.

The tier and application-level metrics do not reflect background task activity. In addition, background tasks are not reflected on the application dashboard.

Reclassify a Business Transaction as a Background Task

You can reclassify any existing business transaction as a background task by right-clicking on it in the business transaction list and choosing the Set as Background Task option. Choosing Set as User Transaction turns a background task into a primary business transaction.

Enabling Automatic Discovery for Background Tasks

At transaction discovery, a transaction is categorized as a background task based on a setting in a custom match rule that matches the transaction.

AppDynamics includes preconfigured match rules for Java frameworks that are likely to be background tasks, including for JChronTab, JavaTimer, Cron4J and Quartz frameworks. The match rules are disabled by default.

You can enable the match rules, modify them, or create your own custom match rules to have matched transactions categorized as background transactions.

To enable background task detection configuration, from the application or custom tier level configuration, edit an existing match rule or create a new one. Enable the rule, and check the Background Task option to business transactions discovered by this rule categorized as background tasks.

Configuring Thresholds for Background Tasks
You can configure performance thresholds for background tasks separately from primary business transactions. From the Configuration > Slow Transaction Thresholds page, click the Background Tasks Thresholds tab to access settings specific for background tasks.

AppDynamics recommends that you configure a threshold that is suitable for the background task in your environment. Use static thresholds for slow and very slow background tasks, if they have infrequent load patterns such as once every night. This is because the dynamic moving average-based thresholds are more suitable for production load scenarios and will automatically classify a background process as slow or very slow.
Business Transactions Logging

The Business Transactions log contains the following information:

- Discovered business transactions
- Discovery stack trace for each entry point
- Dropped business transactions, which show up in the "All Other Traffic" business transaction when there is business transaction overflow. Only executed methods show up in the business transaction log. If the business functionality is not exercised by the application, the code is not executed, and the methods are not examined as potential entry points.

For the Java Agent and the proxy, the name of the Business Transactions log is `BusinessTransactionsYear_mon_day_hr_min.#.log` where # is the log set.

For the .NET Agent the name of the Business Transactions log is `BusinessTransactionsLog.#.txt` where # is the log set.

See Agent Log Files for information about how the logs are organized into sets that roll over. Within a set, the Business Transactions log file can reach a maximum of five MB.
Service Endpoints

A business transaction offers a view of application performance that cuts across the environment, reflecting all services that participate in fulfilling the transaction. Another way of viewing application performance, however, focuses on the performance of a particular service independently of the business transactions that use it.

Like business transactions, service endpoints give you key performance indicators, metrics, and snapshots. However, they provide that information exclusively in the context of that service, omitting business transaction context or downstream performance data.

Monitor Service Endpoints

From the Application navigation menu in the Controller, click Service Endpoints to view a list of service endpoints with key performance metrics. Service endpoints that have performance data during the selected time range appear in the list. Use the filter view options to show inactive service endpoints for the time range. Other view options include showing transactions of certain types or ones that exceed a configurable average response time, among others.

The Actions menu lets you view, delete, and exclude selected service endpoints. Excluding service endpoints from this menu hides the service endpoint from the list. To completely disable metric collection for the service endpoint, create an exclude rule. See Service Endpoint Detection.

Diagnostic sessions are not intended to run in the context of a service endpoint alone. Therefore you cannot directly start a diagnostic session from a service endpoint. If you run a diagnostic session on the business transactions that include calls to the service endpoint, the transaction snapshots from these diagnostic sessions appear in the service endpoint dashboard.

Service Endpoint Limits

A service endpoint adds a small amount of system overhead. Agents capture approximately three metrics per service endpoint, so monitoring each service endpoint results in additional metric traffic.

To prevent the possibility of a boundless expansion of service endpoints, the Controller and agent configurations apply the following limits:

<table>
<thead>
<tr>
<th>Element</th>
<th>Limit</th>
<th>For more information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>25 service endpoints per entry point type</td>
<td>See max-service-end-points-per-entry-point-type on App Agent Node Properties Reference.</td>
</tr>
<tr>
<td>Java Agent</td>
<td>40 asynchronous worker thread service endpoints</td>
<td>See max-service-end-points-per-async-type on App Agent Node Properties Reference.</td>
</tr>
</tbody>
</table>
### Service Endpoint Metrics

Service endpoint metrics appear under *Service Endpoints* in the metric browser. Service endpoint metrics are subject to the same operations as other metrics, including those around metric registration, metric rollups for tiers, and limits on the number of metrics. Custom metrics are not available for service endpoints.

### Customize Service Endpoints

For Java applications, AppDynamics automatically detects and registers service endpoints. For either Java or .NET, you can configure custom service endpoint detection rules.

For information on customizing service endpoints, see *Service Endpoint Detection*.

<table>
<thead>
<tr>
<th>Node</th>
<th>100 service endpoints</th>
<th>See max-service-end-points-per-node on App Agent Node Properties Reference.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Controller account</td>
<td>4000 service endpoints</td>
<td>For an on-prem Controller, the limit is configurable using the sep.ADD.registration.limit configuration property accessible in the Administration Console.</td>
</tr>
<tr>
<td>Execution thread</td>
<td>1 endpoint</td>
<td>See max-service-end-points-per-thread on App Agent Node Properties Reference.</td>
</tr>
</tbody>
</table>
The Tier list in the Controller UI lets you view and organize the tiers in your business application and the monitored nodes within each tier.

View the Tier List

The tier list gives you a consolidated view of the tiers and nodes in the business application.

To view the list, click Tiers and Nodes in the application tree. The page shows all tiers in the business application, along with health data for each tier.

In the tiers list, the Machine Agent status column indicates whether the Standalone Machine Agent is reporting to the Controller. If the machine agent is not reporting, the column is empty. In this case, either an agent is not installed or it is not reporting for some other reason.

You can change the selection of data displayed in the list by choosing Health, Hardware, or Memory from the pulldown list. If the machine agent is reporting to the Controller, Hardware shows additional metrics for the machine on which the app agent is installed.

Memory shows memory information relevant to your application environment. Note that the specific information shown for each environment can vary, depending on the nature of the environment. PHP memory management is different than other languages. The relevant memory metrics supplied by the App Agent for PHP are in the Metric Browser.

View the Tier or Node Dashboard

The tier and node dashboards show a flow map view at a tier or node level. Select the tier or node (by first expanding the tier that contains the node) from the Tiers & Nodes page, as follows:

- For a tier dashboard, select the tier in the Tiers & Nodes page and click Details. The dashboard shows the tier flow map, the nodes in the tier (if viewing a tier dashboard), events, and slow response snapshots.
- For a node dashboard, select expand the tier in the Tiers & Nodes page and double-click on the node.

Create a New Tier in an Application

You can create a tier in the AppDynamics model of your environment by specifying the new tier in the agent configuration of your instrumented application. Most tiers are created in this manner. However, you can also create tiers manually by clicking Actions > Create App Server Tier button on the tier list.

After creating a tier, you can add nodes to the tier via the app agent configuration or by reassigning an existing node to the new tier, as described in the following section.

Naming Node, Tiers and Business Applications in the UI
To rename a business application, tier, or node, see Interacting with Flow Maps.

The UI prevents you from using certain special characters for tier, nodes and business applications names. Besides numbers and upper and lower case characters, the special characters you can use in tier or node names include:

```
: ~ ! @ # $ % ^ * ( ) _ = / \ , . [ ] { } | ? .
```
Moving and Renaming Nodes

In AppDynamics, you can rename nodes or move them between business applications or tiers to reflect changes in your environment model, as described here.

Moving Nodes

When you move a node to another tier or application using the AppDynamics UI, the following happens:

- All historical data is removed for that node, with no possibility for data recovery. Moving nodes in the UI is primarily intended for refining the AppDynamics application model of your environment, or other non-production scenarios. If retaining historical data for the node is important to you, you should change the tier, application or node name in the agent’s controller-info.xml file.
- The tier or application association in the agent’s configuration file no longer reflects the actual tier or application association. The UI association prevails unless the force-agent-registration flag in the agent’s controller-info.xml is set to true.

You can move a node to a new application or tier in one of the following ways:

- Using AppDynamics UI: in the node dashboard, choose Actions > Move Node. You do not need to restart the JVM using this method.
- Using the configuration file: update the tier or application setting in the agent’s controller-info.xml configuration file and restart the JVM.

Force Node Registration

If you moved a node in the UI and you want to move it again elsewhere using controller-info.xml, set the force-agent-registration property in the configuration file to true and restart the JVM. See Force Agent Registration Property.

Data Retention

As mentioned above, when you move a node to another tier or application all node-level historical data (metrics, events, and snapshots) is lost. The historical data is not recoverable. If retaining historical data for the node is important to you, you should change the tier, application or node name in the agent’s controller-info.xml file.
Any tier-level data that was generated by the moved node is retained in the former tier. Similarly, when moving nodes between tiers in the same application, the tier-level data generated by the node is retained in the former tier.

**Moving a Java Node and Machine Agent**

If your JVM machine has both a Java Agent and a Machine Agent, you cannot change the associations in the Machine Agent controller-info.xml file. You can only change these associations either through the UI or by modifying the Java Agent controller-info.xml file, as described in Java Agent Configuration Properties.

**Renaming Nodes**

To rename a node, change the name in the AppDynamics UI and then in the app agent configuration file. Changing the node name in the configuration file ensures that the new name is retained through the next agent restart.

If you rename a node from the app agent configuration file only, the effect is that a new node is created with the restarted agent reporting data as that node. AppDynamics retains the historical data previously collected for the node, but the data is associated with the old node name. For most purposes, therefore, renaming the node involves changing it in the UI first, and then in the configuration file.

To change the name in the UI, follow the instructions above for moving the node, but choose **Edit Properties** from the **Actions** menu, and enter the new name in the Node Properties dialog.

For information on changing the node name in the configuration files, see the agent installation or configuration documentation for your agent under **Install App Server Agents**.

---

For information on renaming a business application, see **Business Applications**.
Troubleshoot Node Problems

On this page:
- Using the Node Problem Viewer

AppDynamics categorizes the ten items that deviate the most from the baseline performance as node problems. You can analyze node problems from the Node Problems viewer.

The Machine Agent must be installed on the machine hosting the node that you are troubleshooting.

Using the Node Problem Viewer

If accessed from the Snapshot viewer, the Node Problem viewer displays node problems for the time range of the snapshot. If accessed from the Node dashboard, it uses the time range set in the Node dashboard. You can edit the time range in the Node Problem viewer and then apply the new time range. You can also define and save a custom time range.

The right panel of the Node Problem viewer displays the metrics that deviate the most from their baselines for the specified time range.

On the left, you can set filters on what data appears. Options include:

- The baseline to use to define node problems.
- The type of metrics, whether custom, hardware metrics, JVM, and so on.
- Specify whether to display problems with values that are:
  - higher than baseline
  - lower than the baseline
  - higher and lower than baseline

For example, if you are only interested in CPU that is too high, set the Analysis Type to Higher for Hardware Resources. On the other hand, if you want to monitor the load on your machine continuously because low CPU usage would also be a problem, set the Analysis Type to Higher and Lower. The right panel shows the ten items that deviate the most from the baseline.

After changing an option, click the Analyze Node Problems button to apply the change.
Monitor JVMs

This topic provides an overview of some of the tools AppDynamics provides for monitoring Java applications and troubleshooting common issues.

JVM Key Performance Indicators

A typical JVM may have thousands of attributes that reflect various aspects of the JVM's activities and state. The key performance indicators that AppDynamics focuses on as most useful for evaluating performance include:

- Total classes loaded and how many are currently loaded
- Thread usage
- Percent CPU process usage

On a per-node basis, AppDynamics reports:

- Heap usage
- Garbage collection
- Memory pools and caching
- Java object instances

You can configure additional monitoring for

- Automatic leak detection
- Custom memory structures

Monitoring JVM performance

You can view JVM performance information from the Tiers & Nodes dashboard or from the Metric Browser.

In the Tiers & Nodes dashboard, see the following tabs for JVM-specific information:

- The Memory subtab of the Tiers & Nodes dashboard allows you to view various types of JVM performance information: Heap and Garbage Collection, Automatic Leak Detection, Object Instance Tracking, and Custom Memory Structures.
- The JMX subtab of the Tiers & Nodes dashboard allows you to view information about JVM classes, garbage collection, memory, threads, and process CPU. In the JMX Metrics subtab metric tree, click an item and drag it to the line graph to plot current metric data.

In the Metric Browser, click Application Infrastructure Performance and expand the JVM folder for a given node to access information about Garbage Collection, Classes, Process CPU, Memory, and Thread use.

Alert for JVM Health

You can set up health rules based on JVM or JMX metrics. Once you have a health rule, you can create specific policies based on health rule violations. One type of response to a health rule violation is an alert. See Alert and Respond for a discussion of how health rules, alerts, and policies can be used.

You can also create additional persistent JMX metrics from MBean attributes. See Configure JMX Metrics from MBeans.

JVM Crash Guard

Using the Standalone Machine Agent, when a JVM crash occurs on a machine or node, you can be notified almost immediately and take remediation actions. Learning of a JVM crash is because it may be a sign of a severe runtime problem in an application.
Implemented as part of JVM Crash Guard, JVM Crash is an event type that you can activate to provide the critical information you need to expeditiously handle JVM crashes.

**Memory Management**

Memory management includes managing the heap, certain memory pools, and garbage collection. This section focuses on managing the heap. You can view heap information in the metric browser or in the Memory tab for a given node as directed in “Monitoring JVM Information.”

The size of the JVM heap can affect performance and should be adjusted if needed:

- A heap that is too small will cause excess garbage collections and increases the chances of OutOfMemory exceptions.
- A heap that is too big will delay garbage collection and stress the operating system when needing to page the JVM process to cope with large amounts of live data

For more detail on garbage collection, see Garbage Collection.

**Detect Memory Leaks**

By monitoring the JVM heap and memory pool, you can identify potential memory leaks. Consistently increasing heap valleys might indicate either an improper heap configuration or a memory leak. You can identify potential memory leaks by analyzing the usage pattern of either the survivor space or the old generation. To troubleshoot memory leaks see Java Memory Leaks.

**Detect Memory Thrash**

Memory thrash is caused when a large number of temporary objects are created in very short intervals. Although these objects are temporary and are eventually cleaned up, the garbage collection mechanism might struggle to keep up with the rate of object creation. This might cause application performance problems. Monitoring the time spent in garbage collection can provide insight into performance issues, including memory thrash. For example, an increase in the number of spikes for major collections affects the JVM’s ability to serve Business Transaction traffic and might indicate potential memory thrash.

The Tiers & Nodes > Memory > Object Instance Tracking subtab helps you isolate the root cause of possible memory thrash. To troubleshoot memory thrash, see Java Memory Thrash.

**Monitor Long-lived Collections**

AppDynamics automatically tracks long-lived Java collections (HashMap, ArrayList, and so on) with Automatic Leak Detection. Custom Memory Structures you have configured are displayed in Tiers & Nodes > Memory > Custom Memory Structures.

AppDynamics provides visibility into

- Cache access for slow, very slow, and stalled business transactions
- Usage statistics (rolled up to Business Transaction level)
- Keys being accessed
- Deep size of the internal cache structure
Automatic Leak Detection for Java

On this page:
- Permissions
- Enable Automatic Leak Detection
- Troubleshooting Leak Detection

Related Pages:
- Java Memory Leaks

This topic describes how to enable automatic leak detection, as described in Java Memory Leaks. Automatic leak detection works only with specific JVMs, Oracle JVM version 1.5 or higher, JRockit JVM 1.5 or higher, and IBM JVM 1.6 or higher.

The Automatic Memory Leak dashboard shows:
- Collection Size: The number of elements in a collection.
- Potentially Leaking: Potentially leaking collections are marked as red. You should start diagnostic sessions on potentially leaking objects.
- Status: Indicates if a diagnostic session has been started on an object.
- Collection Size Trend: A positive and steep growth slope indicates a potential memory leak.

To identify long-lived collections, compare the JVM start time and Object Creation Time.

Permissions

To enable or disable automatic leak detection, you need the Configure Agent Properties permission. For information on AppDynamics Role Based Access Control, see Roles and Permissions.

Enable Automatic Leak Detection

1. In the left navigation pane, click Tiers & Nodes.
2. In the right pane, expand the tier of the node you want to configure.
3. Select the node and click View Dashboard.
4. Click Memory > Automatic Leak Detection.
5. Click On.
6. Click Start On Demand Capture Session to detect leaking collections.

Test before implementing
Leak Detection may affect performance, so test this functionality in a preproduction environment or on a single node first.

AppDynamics begins to automatically track the top 20 application classes and the top 20 system (core Java) classes in the heap.

Troubleshooting Leak Detection

If you cannot see any captured collections, ensure that you have the correct configuration for detecting potential memory leaks. As described in the on-screen instructions you may need to lower the minimum collection size from the default, 5 MB:

To configure the agent of a specific node to capture smaller collections:
1. On the Tiers & Nodes list, expand a tier and double-click the node you want to configure.
2. Ensure the App Server Agent tab is active and click Configure.
3. Select the Application, Tier, and Node and then click Use Custom Configuration.
4. Search for minimum-size-for-evaluation-in-mb in the node properties list. Set its value to a smaller size.
5. Choose to overwrite the node configuration.
Object Instance Tracking for Java

This topic helps you understand how to configure and use object instance tracking. For more information about why you may need to configure this, see Java Memory Thrash.

When object instance tracking is enabled, AppDynamics tracks the top 20 application and top 20 system (core Java) classes in the heap by default. You can configure specific classes to track as described below.

Enabling object instance tracking increases the amount of information captured by the agents, resulting in additional overhead. AppDynamics recommends using object instance tracking only while troubleshooting potential memory leaks. It does not normally need to be enabled during normal operation.

Permissions

To enable or disable object instance tracking, you need the Configure Agent Properties permission.

To configure the custom classes to track, you need the Configure Memory Monitoring permission.

For information on AppDynamics Role Based Access Control, see Roles and Permissions.

Prerequisites for Object Instance Tracking

- Review Java Supported Environments for information about platform support for object instance tracking.
- For JVMs prior to Java 9, object instance tracking uses tools.jar. If your application runs with the JDK, tools.jar should be already available. If you are running with the JRE, you must add tools.jar to <JRE_HOME>/lib/ext and restart the JVM. You can find tools.jar in <JAVA_HOME>/lib/tools.jar. For Java 9 onwards JEP220 moved the capabilities needed by OIT to the Core Java runtime, meaning tools.jar is no longer necessary.
- In some cases, you might also need to copy libattach.so (Linux) or attach.dll (Windows) from your JDK to your JRE.
- Depending on the JDK version, you may also need to specify the classpath as shown below (along with other -jar options).

Specifying the Classpath

When using the JDK runtime environment, set the classpath using the -classpath option for the application. For example:

- On Windows:
  ```
  java -classpath <complete-path-to-tools.jar>;%CLASSPATH% -jar myApp.jar
  ```
- On Unix:
  ```
  java -Xbootclasspath/a:<complete-path-to-tools.jar> -jar myApp.jar
  ```

Starting Object Instance Tracking

To start an object instance tracking session, follow these steps:

1. In the left navigation pane, click Tiers & Nodes.
2. In the right pane, expand the tier node and open the Node Dashboard for the node on which you want to enable object instance tracking.
3. Click the **Memory** tab.
4. Click the **Object Instance Tracking** subtab.
5. Click **ON**.

Tracked classes now appear in the Object Instance Tracking table. You can drill down to the tracked classes to see details.

**Tracking Specific Classes**

For performance reasons, only the top 20 application classes and the top 20 system (core Java) classes in the heap are tracked automatically.

Use the **Configure Custom Classes to Track** option on the Object Instance Tracking subtab to specify instances of specific classes to track. Note that the classes you configure here are tracked only if their instance count is among the top 1000 instance counts in the JVM.

To track instances of custom classes:

1. On the Object Instance Tracking tab, click **Configure Custom Classes to Track**.
2. On the Instrumentation page that displays, click the tier you want to customize.
3. In the Object Instance Tracking section for the tier, click **Add**.

4. Enter the fully-qualified class name of the class to track and click **Save**.

The class you added is now tracked during object instance tracking sessions.
Custom Memory Structures for Java

On this page:
- Permissions
- Custom Memory Structures and Memory Leaks
- Using Automatic Leak Detection vs Monitoring Custom Memory Structures
- Identifying Potential Memory Leaks

AppDynamics automatically tracks long-lived Java collections (HashMap, ArrayList, and so on) with Automatic Leak Detection. You can add tracking for specific classes using the Custom Memory Structures capability.

You can use this capability to monitor a custom cache or other structure that is not a Java collection. For example, you may have a custom cache or a third party cache such as Ehcache. In a distributed environment, caching can easily become a prime source of memory leaks. In addition, custom memory structures may or may not contain collections of objects that would be tracked using automatic leak detection.

To configure custom memory structures, ensure custom memory structures are supported in your JVM environment. See JVM Support.

Permissions

To enable or disable object instance tracking, you need the Configure Agent Properties permission.

To configure the custom classes to track, you need the Configure Memory Monitoring permission.

For information on AppDynamics Role Based Access Control, see Roles and Permissions.

Custom Memory Structures and Memory Leaks

Typically custom memory structures are used as caching solutions. In a distributed environment, caching can easily become a source of memory leaks. AppDynamics helps you to manage and track memory statistics for these memory structures.

AppDynamics provide visibility into:

- Cache access for slow, very slow, and stalled business transactions
- Usage statistics rolled up to the Business Transaction level
- Keys being accessed
- Deep size of internal cache structures

Using Automatic Leak Detection vs Monitoring Custom Memory Structures

Automatic leak detection captures memory usage data for all map and collection libraries in a JVM session. However, custom memory structures may not contain all collections objects. For example, you may have a custom cache or a third-party cache such as Ehcache for which you want to collect memory usage statistics.

Using custom memory structures, you can monitor any custom object created by the app and the size data can be traced across JVM restarts. Automatic leak detection is typically used to identify leaks, while custom memory structures are used to monitor large coarse-grained custom cache objects.

The following provides the workflow for configuring, monitoring, and troubleshooting custom memory structures. You must configure custom memory structures manually.

1. On the Tiers & Nodes dashboard, use the Automatic Leak Detection, On Demand Capture Session feature to determine which classes aren’t being monitored, for example, custom or third-party caches such as EhCache.
2. Configure Custom Memory Structures and then restart the JVM if necessary.
3. Enter the fully-qualified classname on the Create New Instance Tracker window and click Save.

AppDynamics automatically tracks long-lived Java collections (HashMap, ArrayList, and so on) with Automatic Leak Detection.
4. Turn on Custom Memory Structures monitoring to detect potential memory leaks in the custom memory structures you have configured. See Custom Memory Structures for Java.
5. Drill down into leaking memory structures for details that will help you determine where the leak is.

To identify custom memory structures:

1. On the Automatic Leak Detection subtab of the Memory tab, click **On**.
2. Click **Start On Demand Capture Session** to capture information on which classes are accessing which collections objects. Use this information to identify custom memory structures.

AppDynamics captures the top 1000 classes, by instance count.

**Identifying Potential Memory Leaks**

Start monitoring memory usage patterns for custom memory structures. An object is automatically marked as a potentially leaking object when it shows a positive and steep growth slope. The Memory Leak Dashboard provides the following information:

<table>
<thead>
<tr>
<th>Dashboard</th>
<th>My Dashboards</th>
<th>Container</th>
<th>Server</th>
<th>Memory</th>
<th>JNM</th>
<th>JMX</th>
<th>Events</th>
<th>Slow Response Times</th>
<th>&gt;&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heap &amp; Garbage Collection</td>
<td>Automatic Leak Detection</td>
<td>Object Instance Tracking</td>
<td>Custom Memory Structures</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Custom Memory Structures**

- Class: The name of the class or collection being monitored.
- Deep Size (bytes): The upper boundary of memory available to the structure. The deep size is traced across JVM restarts.
- % of Current Used Heap: The percentage of memory available for dynamic allocation.
- Potentially Leaking: Potentially leaking collections are marked as red. We recommend that you start a diagnostic session on potentially leaking objects.
- JVM Start Time: Custom Memory Structures are tracked across JVM restarts.
- Status: Indicates if a diagnostic session has been started on an object.
- Deep Size: A positive and steep growth slope indicates a potential memory leak.

After the potentially leaking collections are identified, start the diagnostic session.

**Diagnosing memory leaks**

On the Custom Memory Structures Dashboard, select the class name to monitor and click **Drill Down** or right-click the class name and select **Drill Down**.
Isolating a leaking collection

Use Content Inspection to identify to which part of the application the collection belongs. It allows monitoring histograms of all the elements in a particular memory structure. Start a diagnostic session on the object and then follow these steps:

1. Select the Content Inspection tab.
2. Click **Start Content Summary Capture Session**.
3. Enter the session duration. Allow at least 1-2 minutes for the data to generate.
4. Click **Refresh** to retrieve the session data.
5. Click a snapshot to view the details about that specific content summary capture session.
**Access Tracking**

Use Access Tracking to view the actual code paths and business transactions accessing the memory structure. Start a diagnostic session on the object and follow these steps:

1. Select the Access Tracking tab.
2. Select **Start Access Tracking Session**.
3. Enter the session duration. Allow at least 1-2 minutes for data generation.
4. Click **Refresh** to retrieve the session data.
5. Click a snapshot to view the details about that specific content summary capture session.
### Code Paths and Business Transactions accessing this Collection (get(), put(), etc)

#### Code Paths

<table>
<thead>
<tr>
<th>Code Path</th>
<th>Occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>memorymonitoring.CustomMemoryStructure.addNewCollector(CustomMemoryStructure.java)</td>
<td>109</td>
</tr>
<tr>
<td>at memorymonitoring.App.main(App.java):34</td>
<td></td>
</tr>
<tr>
<td>at sun.reflect.NativeMethodAccessorImpl.invoke(Native Method)</td>
<td></td>
</tr>
<tr>
<td>at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)</td>
<td></td>
</tr>
<tr>
<td>at java.lang.reflect.Method.invoke(Method.java:577)</td>
<td></td>
</tr>
<tr>
<td>at com.intellij.execution.application.AppMain.main(AppMain.java:120)</td>
<td></td>
</tr>
</tbody>
</table>

#### Business Transactions accessing this Collection

<table>
<thead>
<tr>
<th>Transaction Name</th>
<th>Occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>PG0</td>
<td>100</td>
</tr>
</tbody>
</table>
When a JVM crash occurs, you need to know as soon as possible. A JVM crash may be a sign of a severe runtime problem in an application and often call for immediate remediation steps.

This topic describes how to use AppDynamics JVM Crash Guard, which you can use to monitor and remediate JVM crashes.

Viewing JVM Crash Information

When a JVM crash occurs, a corresponding event type is generated in AppDynamics. To analyze and troubleshoot the crash:

1. In the Events window, double-click the JVM Crash of interest.
2. Examine any logs associated with the JVM Crash event. (The local log folder is determined by the type of JVM and how it is configured. For more information, refer to the documentation for the specific JVM.)

The JVM Crash window also displays information about actions executed as a result of the crash. These are actions that you specify when creating a policy that is triggered by a JVM crash event. See Policies for more information.

The JVM Crash event details include the following information:

- Timestamp
- Crash reason
- Hostname IP address
- Process ID
- Application name
- Node name
- Tier name

In the Crash Reason details field of the JVM Crash Details tab, the JVM Crash details indicate the root cause of the crash if available. For example, a java.lang.OutOfMemoryError or Segmentation Fault.

To facilitate the discovery and display of the reason for the JVM crash, JVM Crash Guard supports:

- Hotspot JVM error log analysis
- IBM JVM System Dump log analysis
- Jrockit JVM error log analysis

Start Monitoring for JVM Crashes

Before you can monitor JVM crashes, you must do the following:

- Install and enable a Machine Agent on the machine that you want to monitor for JVM crashes. JVM Crash Guard works with the Standalone Machine Agent to trigger an AppDynamics policy when a JVM Crash event occurs.
- Ensure that the Machine Agent is running with the required privileges.
  - On Windows, the Machine Agent must run in Administrator root mode.
  - On Linux, JVM Crash Guard requires that the Machine Agent user be able to read all the processes in /proc/*. This may be the ‘root’ user or another user with this privilege.
- Enable the JVM Crash Guard (it is disabled by default). Navigate to extensions/CrashGuard/conf/crashGuardConfig.yml and open the crashGuardConfig.yml file. Set enabled to true.

Once you have verified the requirements, follow these steps to create a policy for JVM crash events:
1. From the left-hand navigation menu, click **Alert & Respond > Policies** and then click **Create a Policy**.

   ![Create Policy](image)

   - **Trigger**
   - **Health Scope**
   - **Object Scope**
   - **Actions**

   - **Name**
   - **Enabled**
   - **Execute actions in batch**

   **This Policy will fire when any of these events occur**

   - **Health Rule Violation Events**
     - Health Rule Violation Started - Warning
     - Health Rule Violation Started - Critical
     - Health Rule Violation Continues - Warning
     - Health Rule Violation Continues - Critical
     - Health Rule Violation Upgraded - Warning to Critical
     - Health Rule Violation Downgraded - Critical to Warning
     - Health Rule Violation Ended - Warning
     - Health Rule Violation Ended - Critical
     - Health Rule Violation Cancelled - Warning
     - Health Rule Violation Cancelled - Critical

   - **Other Events**
     - Slow Transactions
     - Code Problems
     - Application Changes
     - Server Crashes
       - JVM Crash
       - CLR Crash
     - AppDynamics Config Warnings
     - Discovery
     - Synthetic Availability
     - Synthetic Performance

   - **Custom Events**

   **Enable this option to activate JVM Crash Guard**

   ![Enable Custom Events](image)

   ![No Custom Events Selected](image)

2. In the Other Events section, expand the **Server Crashes** option and click **JVM Crash**.

3. Proceed as usual to create the Policy. For more information on creating Policies, see **Policies**.

**Note:** If an uninstrumented JVM crash happens within less than a minute of a previous crash then it will not be reported by the Standalone Machine Agent. In some circumstances, the JVM may crash and then be restarted only to crash again within one minute. For this repetitive cycle crash and restart scenario, only the first JVM crash is reported by the agent.
Garbage Collection

AppDynamics gathers Garbage Collection metrics and lets you analyze how periodic garbage collections affect the performance of your application. It is important to identify the impact of excessive garbage collection or memory-caused instability on the application. A typical JAVA application which runs on the Java Virtual Machine (JVM) creates objects such as strings, files, and arrays of primitives on the heap. The JAVA Garbage collection is an automatic memory management process which finds and gets rid of the objects which are no longer used by the application. The JVM periodically performs Garbage Collection to maximize available memory and the programmer need not explicitly mark the objects to be deleted. Garbage collection requires a stop-the-world suspension of all application threads. This process affects the performance, especially for applications with large amounts of data, multiple threads, and high transaction rates.

See How to Master Your Java Memory for a review of how generational garbage collection works.

Before You Begin

If your application runs under JDK version 1.6.0_26-b03 and you have configured the monitored application to use G1GC, the Java Agent cannot capture memory statistics. To capture memory statistics, you can do any one of the following:

- Remove G1GC (-XX:+UseG1GC) from the application startup options, or
- Upgrade the JDK to version 1.6.0_32 or higher.

In JVM version 1.7 or later, the agent attaches listeners to the Java event notification service to generate garbage collection metrics. In JVM versions prior to 1.7, the agent parses certain JVM log files to generate metrics, so you must verify that your JVM is generating the required logs. See 'Enable Log-based Garbage Collection.'

Monitor Garbage Collection

The Java Agent reports certain garbage collection metrics at one-minute intervals. You can view these metrics on the Heap & Garbage Collection sub-tab of the Memory tab on the Node dashboard:

- Heap utilization: free, used, committed, and available. This is the most coarse-grained view of memory use.
- Garbage collection: minor, major, and total on one timeline. This should give you some notion of the ratio of minor to major collections.
- Minor garbage collection on a timeline.
- Major garbage collection on a timeline.
- Memory pool use, including the use of all memory spaces: Young Generation, Old Generation, and PermGen.

For a finer-grained view of the impact of garbage collection on application performance, you can view garbage collection metrics in the Metric Browser. Navigate to Application Infrastructure Performance > Tier > JVM > Garbage Collection.

In addition to the periodically-collected metrics, the Metric Browser shows the following metrics triggered by minor or major garbage collection events:

<table>
<thead>
<tr>
<th>Metric</th>
<th>Triggering Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocated-Objects</td>
<td>Minor collection</td>
<td>The amount of memory allocated to the Young Generation. A high growth rate might indicate memory thrash. The allocation rate affects the frequency of minor collection events, which can impact application performance over time. The value for Count indicates how many collections were made.</td>
</tr>
</tbody>
</table>
### Tune Garbage Collection in the JVM

After reviewing Garbage Collection diagnostic metrics, you can use the following JVM arguments to tune space allocation in the JVM garbage collection memory pool. For example, you might want to increase the size of tenured space if your application needs to store many objects long term.

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Xms</td>
<td>The amount of total memory allocated to young and old generation space.</td>
</tr>
<tr>
<td>-XX:NewSize</td>
<td>The amount of memory allocated to the young generation space.</td>
</tr>
<tr>
<td>-XX:PermSize</td>
<td>The amount of memory allocated to the permanent generation</td>
</tr>
</tbody>
</table>

Consult the documentation for your JVM for additional arguments that relate to Garbage Collection space allocation.

### Enable Log-based Garbage Collection

For applications running in JVMs prior to version 1.7, garbage collection monitoring is based on periodically parsing certain garbage collection logs.

To set up garbage collection logging for your application, use the following arguments:

- `Xloggc:log-file-path`
- `XX:+UseGCPolicyType`
- `XX:+PrintGCDetails`

The `log-file-path` option specifies where the log file is located. The table below describes the possible values for collector types for the `XX:+UseGCPolicyType` option:

<table>
<thead>
<tr>
<th>Collector Type</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMS Collector</td>
<td>Good for applications that require low pause times and that can share resources with the garbage collector.</td>
</tr>
<tr>
<td><code>XX:+UseConcMarkSweepGC</code></td>
<td>Use the <code>XX:ParallelCMSThreads=&lt;n&gt;</code> to set the number of threads to use.</td>
</tr>
<tr>
<td>Throughput or Parallel Collector</td>
<td>Can use multiple CPUs to speed up application throughput; good to use for work-intensive apps that can accept long pauses.</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-XX:+UseParallelGC</td>
<td></td>
</tr>
<tr>
<td>-XX:+UseParallelOldGC</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>G1 Collector</th>
<th>Available in Java 7 and designed to be a long term replacement for the CMS collector. This is a parallel, concurrent, and incrementally compacting low-pause collector.</th>
</tr>
</thead>
<tbody>
<tr>
<td>-XX:+UseG1GC</td>
<td></td>
</tr>
</tbody>
</table>

To direct the Controller to display logged information, register the following node properties in the Controller UI:

```markdown
enable-jmx-visibility=true  
enable-log-based-gc=visibility=true
```

To change the default log check interval, register the following node property:

```markdown
logbased-visibility-log-check-interval-in-mills=1000
```

**Specify Regular Expressions for Parsing Garbage Collection Logs**

When you enable log-based garbage collection metrics, the agent uses built-in regular expressions to accommodate different JDK and garbage collection type combinations. If the built-in regular expressions don't return the garbage collection metrics for your system, you can register the following node properties to specify custom regular expressions for parsing the logs:

- young-gc-custom-regex-1
- young-gc-custom-regex-2
- young-gc-custom-regex-3
- full-gc-custom-regex-1
- full-gc-custom-regex-2
- full-gc-custom-regex-3

After you set a custom regular expression using a node property, the agent no longer uses any of the built-in regular expressions to parse the logs.
Monitor JMX

On this page:

- About JMX Monitoring Support
- Requirements for JMX Monitoring
- MBean-Based Metrics
- View JMX Metric Data
- View Trending MBeans with Live Graphs
- Working with MBean Values
- Invoke MBean Operations
- Available JMX Metrics
- Adding JMX Metrics
- Extending Monitoring with the Agent API

Java application environments usually support JMX (Java Management Extensions) or IBM Performance Monitoring Infrastructure (PMI). AppDynamics automatically discovers JMX and PMI attributes.

About JMX Monitoring Support

JMX uses objects called MBeans (Managed Beans) to expose data and resources from your application. In a typical application environment, there are three main layers that use JMX:

- JVMs provide built-in JMX instrumentation or platform-level MBeans that supply important metrics about the JVM.
- Application servers provide server or container-level MBeans that reveal metrics about the server.
- Applications often define custom MBeans that monitor application-level activity.

MBeans are typically grouped into domains to indicate where resources belong. Usually, in a JVM there are multiple domains. For example, for an application running on Apache Tomcat, there are Catalina and Java.lang domains. Catalina represents resources and MBeans relating to the Tomcat container, and Java.lang represents the same for the JVM Hotspot runtime. The application may have its own custom domains.

You can use MBean attributes to create persistent JMX metrics in AppDynamics as described here. In addition, you can import and export JMX metric configurations from one version or instance of AppDynamics to another.

To learn about PMI see Writing PMI Applications Using the JMX Interface.

Requirements for JMX Monitoring

AppDynamics can capture MBean data if:

- The monitored system runs on Java 1.5 or later.
- Each monitored Java process has JMX enabled. See JMX documentation.

MBean-Based Metrics

AppDynamics creates long-term metrics of the key MBean attributes that represent the health of the Java container. Depending on your application configuration, metrics may include:

- Session information such as the number of active and expired sessions, maximum active sessions, processing time, average and maximum alive times, and a session counter.
- Web container runtime metrics that represent the thread pool that services user requests. The metrics include pending requests and the number of current threads servicing requests. These metrics are related to Business Transaction metrics such as response time.
- Messaging metrics related to JMS destinations, including the number of current consumers and the number of current messages.
- JDBC connection pool metrics including current pool size and maximum pool size.

To view the JMX metrics discovered in a node, see the JMX tab on the Node Dashboard. For additional metrics not discovered automatically, you can configure JMX Metrics from MBeans yourself.
You can mask sensitive JMX and MBean data by setting the sensitive-data-filters property in the app-agent-config.xml. For more information, see Filter Sensitive Data.

**View JMX Metric Data**

You can view MBean-based metrics using the Node Dashboard and the Metric Browser. In addition, the MBean Browser enables you to view all the MBeans defined in the system.

To view JMX metrics in the Metrics Browser, click the JMX tab in the Node Dashboard. The JMX Metrics browser opens and displays the MBeans in a Metric Tree.

You can perform all the operations that are provided by the Metric Browser. For information on using the Metric Browser, see Metric Browser.

**View Trending MBeans with Live Graphs**

You can monitor the trend of a particular MBean attribute over time using the Live Graph.

To view live graphs:

1. In the Node Dashboard, click the JMX tab and then the MBean Browser sub-tab.
2. Select the domain for which you want to monitor MBeans.
3. In the domain tree, expand the domains to find and then select the MBean that is of interest to you.
4. Expand the Attributes section and then choose an attribute of the MBean.
5. Click Start Live Graph for Attribute and then click Start Live Graph. The runtime values appear.
6. Select an attribute and click Live Graph for Attribute to see a larger view of a particular graph.

**Working with MBean Values**

When troubleshooting or monitoring a Java-based system, you may want to change the values of composite MBeans and execute MBean methods.

To change the value of an MBean attribute or invoke operation, you need the Set JMX MBean Attributes and Invoke Operations permission for the application. For information about configuring user permissions for applications, see Application Permissions.

To view and edit MBean attribute values:

1. From the JMX window, select MBean Browser.
2. In the Domain tree, search and find the MBean that interests you.
3. Select an editable attribute, one that has Yes in the Editable column, and then click View/Edit Attribute.
4. In the MBean Attribute window that displays, you see the current value of the MBean Attribute.
5. You can change the value of an editable MBean Attribute by entering a new value in the Value field.

**Invoke MBean Operations**

Using the JMX viewer, you can invoke an MBean operation with standard Java language strings for parameters, and view return values.

To perform this action, a user must have the Set JMX MBean Attributes and Invoke Operations permission for the application.

To invoke MBean operations:
1. Open the **MBean Browser**.
2. In the Domain tree, search and find the MBean that interests you.
3. Open the **Operations** pane, scroll to find the operation that interests you, and double-click the **Invoke Action** activator.
4. Enter the parameter values for the operation and then click **Invoke**. Scalar values for constructors of complex types, such as `getMBeanInfo(java.util.Locale)` allow you to enter “en-us”.

A message appears indicating that the operation is in progress and the number of seconds elapsed. When the operation completes, the results display. The method return result from an invocation can also be a complex attribute. In this case, the name, description, type, and editable attributes of the method are also displayed in the MBean Operation Result area.

**Available JMX Metrics**

Java Management Extensions (JMX) is a public specification for monitoring and managing Java applications. Through JMX, AppDynamics can access Java class properties that collect management data, such as the resources your application is consuming.

For information on the specific metrics available for your environment, see the documentation provided by your vendor:

- Apache ActiveMQ, see [ActiveMQ MBeans Reference](ActiveMQ Features documentation)
- Apache Kafka, see [Default JMX Metrics for Apache Kafka Backends](Default JMX Metrics for Apache Kafka Backends)
- Apache Solr, see [Quick Demo](Solr JMX documentation)
- Apache Tomcat, see the descriptions of [JMX MBeans for Catalina](mbeans-descriptor.xml file for each package)
- Cassandra, see [Cassandra Metrics](Cassandra Metrics)
- Coherence, see the Coherence MBeans Reference in Appendix A of the [Coherence Management Guide](Coherence Management Guide)
- GlassFish, see the Oracle Sun Glassfish Administration Guide where you can find a [Metrics Information Reference](Oracle Sun Glassfish Administration Guide)
- HornetQ, see [Using Management Via JMX](Using Management Via JMX)
- JBoss, see [An Introduction to JMX](An Introduction to JMX)
- Kafka Broker, see [Monitoring Kafka](Monitoring Kafka)
- Oracle WebLogic Server, see [Understanding JMX](Understanding JMX)
- WebSphere PMI, see [PMI data organization](PMI data organization) in the IBM Websphere Application documentation

**Adding JMX Metrics**

In addition to the preconfigured metrics, you can define a new persistent metric using a JMX Metric Rule that maps a set of attributes from one or more MBeans. For more information, see [Configure JMX Metrics from MBeans](Configure JMX Metrics from MBeans).

**Extending Monitoring with the Agent API**

The Application Server Agent API lets you access metrics that are not supported by default or by MBeans. You can use the API to:

- Inject custom events and report on them
- Create and report on new metrics
- Correlate distributed transactions when using protocols that AppDynamics does not support

To learn more about the Application Server Agent API, see the Javadoc included with the Java Agent software at the following location:

<agent_home>/sdk/docs/index.html
Default JMX Metrics for Apache Kafka Backends

The Java Agent includes rules for key metrics exposed by Apache Kafka producers and consumers. To monitor JMX metrics not collected by default, you can use the MBean browser to select the Kafka JMX metric and create a rule for it.

Kafka Producer JMX Metrics

- Response rate: the rate at which the producer receives responses from brokers
- Request rate: the rate at which producers send request data to brokers
- Request latency average: average time between the producer's execution of KafkaProducer.send() and when it receives a response from the broker
- Outgoing byte rate: producer network throughput
- IO wait time: percentage of time the CPU is idle and there is at least one I/O operation in progress
- Record error rate: average record sends per second that result in errors
- Waiting threads: number of user threads blocked waiting for buffer memory to enqueue their records
- Requests in flight: current number of outstanding requests awaiting a response
- Network IO rate: average number per second of network operations, reads or writes, on all connections

Kafka Consumer JMX Metrics

- Records lag max: maximum lag in terms of number of records for any partition within the timeframe
- Bytes consumed rate: average number of bytes consumed per second
- Fetch rate: number of fetch requests per second
- Records consumed rate: average number of records consumed per second
- Fetch latency max: maximum time this is taken for any fetch request

Kafka Server JMX Metrics

- broker-request-total-time-ms: Total end-to-end time in milliseconds.
- broker-request-send-response-ms: Responses dequeued are sent remotely through a non-blocking IO. The time between dequeuing the response and completing send is indicated by this metric.
- broker-request-response-queue-ms: Responses too are added to a queue. There is one queue per network processor. Network processor dequeues this response and sends it back. Time spent waiting in this queue is indicated by this metric.
- broker-request-remote-time-ms: If the request needs remote processing, the time spent in remote processing is indicated by this metric. For example, for the producer request, if acks is set to -1, the request is not completed until the acknowledgment is received from the followers. Time spent waiting for the followers is indicated by this metric. A fetch request can also be delayed if there is not enough data to fetch. That time too is accounted for by this metric.
- broker-request-processing-ms: The request is then processed by the KafkaAPI. The time spent in the processing is indicated by this metric. If this metric is high, debug the relevant request handler.
- broker-request-queue-time-ms: The request is added to a common queue. The items in the queue are processed by the request handler threads. The number of these handler threads can be configured through num.io.threads parameter. The average time, a request spends in this queue, is indicated by this metric. If this metric is high, increase the handler threads.
Monitor .NET Nodes

On this page:

- End-to-End APM for .NET

Search the .NET APM topics:

---

Once you instrument your .NET application environment with the AppDynamics .NET Agent, you can start monitoring .NET application performance. For general information on how to use AppDynamics, see the topics under AppDynamics Essentials.

This section describes considerations specific for monitoring .NET applications. This section also describes how to fine-tune the configuration for specific .NET frameworks and technologies. These include, for example, monitoring and configuration topics for your IIS applications, Windows services, and standalone applications.

End-to-End APM for .NET

- .NET Supported Environments
- Install a Controller or get a SaaS Controller
- .NET Agent
- Monitor CLRs
- Monitor Windows Hardware Resources
Monitor Windows Hardware Resources

- Related pages:
  - Monitor IIS
  - Monitor CLRs
  - Manage Windows Performance Metrics

The AppDynamics .NET Agent includes an embedded .NET Machine Agent that runs as part of the AppDynamics.Agent.Coordinator service. Among other things, the .NET Machine Agent regularly gathers system performance data metrics such as:

- CPU activity
- Memory usage
- Disk reads and writes
- Network traffic
- Percent free disk space and megabytes free

View Windows Hardware Resource metrics in the Metric Browser. When IIS is installed on the machine, the .NET Machine Agent also reports IIS, ASP.NET and ASP.NET Application metrics. See Monitor IIS. The .NET Machine Agent also reports CLR metrics, which are based on performance counters. See Monitor CLRs.

For applications running on the Windows Azure platform, AppDynamics disables some hardware resource monitoring and machine agent features:

- CLR crash reporting
- Machine snapshots
- Performance counter metrics

Manage the .NET Machine Agent

You can manage the .NET Machine Agent in the Controller under Settings > AppDynamics Agents on the Machine Agents tab.

- To enable or disable all the machine agents, including the Standalone Machine Agent, for a business application, click Enable Agents or Disable Agents.
- To enable or disable an individual .NET Machine Agent, right-click the agent and click Enable Selected Machine Agent or Disable Selected Machine Agent.

Enabled agents maintain a ‘heartbeat’ connection to the Controller so you can enable it again. The agent persists its enabled/disabled state even after a restarted.

- To reset a .NET Machine Agent, right-click the agent and click Reset Selected Machine Agent. The reset operation purges all existing metrics for the agent. The agent restarts and begins gathering metrics again.

Machine Agent Tier

Immediately after you install and configure the .NET Agent, the .NET Machine Agent registers with the Controller and starts reporting performance data.

Frequently the machine agent metrics reach the controller before the app agent has had time to instrument and register IIS applications, Windows services, or standalone applications. If there are no application tiers, the machine agent registers as the Machine Agent tier.

Once the app agent begins reporting metrics for configured application tiers, the .NET Machine Agent reports to the application tiers and stops sending data to the Machine Agent tier. If you don't instrument any IIS applications, Windows services, or standalone applications on the server, the .NET Machine Agent always reports to the Machine Agent tier.

If you install the Standalone Machine Agent, a Java application, in a .NET environment, then you can add custom monitors and extensions such as the HTTP listener. See Configure the .NET Machine Agent for details.
Machine Snapshots for .NET

On this page:
- Before Starting
- Working With Machine Snapshots

Related pages:
- Configure Machine Snapshots for .NET
- Monitor Infrastructure
- Transaction Snapshots

Sometimes environmental conditions cause trouble on the machines where your application runs. Issues that occur outside monitored application code don’t show up on transaction snapshots. Machine snapshots provide critical details about CPU usage, memory usage, and the IIS queue on a server at a specific moment in time. Use the data in machine snapshots to uncover and resolve environmental problems.

AppDynamics generates machine snapshots to capture the state of a server at a specific moment in time. The machine snapshots show the processes running on the machine, the IIS application pool activity, and any related transaction snapshots. By default the .NET Machine Agent takes machine snapshots under the following conditions:

- **Periodic collection**: The agent takes one snapshot every 10 minutes.
- **Breached thresholds**: The .NET Machine agent takes samples of machine statistics every 10 seconds within a 10-minute window. For each sample, the agent checks the CPU percent usage, the memory percent usage, and the oldest item in the IIS application pool queue. The agent flags a sample as a violation when the current usage meets or exceeds one of the following thresholds:
  - CPU at 80% or higher
  - Memory at 80% or higher
  - IIS application pool queue item older than 100 milliseconds

The agent takes a snapshot when it identifies 6 violations of a single type, such as CPU usage, within the window. The agent only takes one snapshot per window for breached thresholds.

To customize the periodic collection or threshold settings, see Configure Machine Snapshots for .NET.

**Before Starting**

The .NET Machine Agent requires IIS 7 or later to return machine snapshot data for IIS application pools. Additionally, you must enable the Request Monitor for the IIS Health Monitoring feature.
Working With Machine Snapshots

The Machine Snapshot tab on the application dashboard lists all the snapshots for the selected time range. Entries for individual snapshots show the following:

- Time the .NET Machine Agent took the snapshot
- Machine name for the snapshot
- Snapshot trigger, either periodic snapshot collection or threshold exceeded
- Percent CPU usage
- Percent memory usage

Use Filters to limit the snapshot list to a specific snapshot trigger:

- Periodic collection
- Memory events
- CPU events
- IIS events
- Machines where the agent took snapshots
- Archived snapshots

Double-click a snapshot to open the Machine Snapshot window:

- The Processes tab shows information similar to that from the Windows Task Manager, including the following:
  - Process ID
  - Process name
  - Process description
  - Percent CPU used by the process
  - Percent memory used by the process

Click a column heading to sort by the column. For example, click CPU % and sort descending to identify processes using the most CPU.

```
Install-WindowsFeature Web-Request-Monitor
```
• The IIS App Pools tab shows information on the active IIS application pools:
  - Application pool name
  - Arrivals to the queue per second
  - Queue requests processed per second
  - Age of the oldest item in the queue

• The Transaction Snapshots tab displays all transaction snapshots involving the current machine for five minutes prior to and five minutes after the machine snapshot. This demonstrates how current environmental factors are affecting your business transaction performance. For example, when CPU usage is high, you may discover slow and stalled transactions.

  Double-click a transaction snapshot to open it.
Monitor CLRs

On this page:

- CLR Events
- CLR Metrics
- Alerting for CLR Health

Related pages:

- Manage Windows Performance Metrics

The AppDynamics .NET Agent includes the .NET Machine Agent that runs as part of the AppDynamics.Agent.Coordinator service. Among other things, the .NET Machine Agent regularly gathers CLR performance data and reports it back to the Controller as metrics.

CLR Events

The .NET Machine Agent monitors for CLR shutdown and restart events:

- The agent reports an App Server Restart event and indicates if the restart was graceful or not.
- For non-graceful shutdown, the agent reports a CLR Crash if it detects one.

> The default exit code for a graceful shutdown is "0", if your Windows service or standalone application uses a different exit code, see "Profiler - Successful Exit Code Element" on .NET Agent Configuration Properties.

CLR Metrics

CLR metrics give insight into how the .NET runtime is performing. The AppDynamics preconfigured CLR metrics include:

- .NET CLR memory usage
- Total classes loaded and how many are currently loaded
- Garbage collection time spent, and detailed metrics about GC memory pools and caching
- Locks and thread usage
- Memory heap and non-heap usage, including the large object heap
- Percent CPU process usage

Alerting for CLR Health

You can set up health rules based on infrastructure metrics. Once you have a health rule, you can create specific policies based on health rule violations. One type of response to a health rule violation is an alert.

In addition to the default metrics, you may be interested in additional metrics. You can specify additional performance counters to be reported by the .NET Machine Agent. See Manage Windows Performance Metrics for details.
Monitor CLR Crashes

On this page:

- Monitor for CLR Crash Events
- Analyze and Respond to CLR Crashes
- Disable CLR Crash Event Reporting
- Crash Events the .NET Machine Agent Monitors

The .NET Machine Agent monitors Windows for non-graceful CLR shutdowns on IIS, Windows services, and standalone applications. The agent raises CLR crash events in the Controller when such crashes occur. Use a policy to alert responsible parties when the agent reports a CLR crash.

Monitor for CLR Crash Events

Create a policy that sends notifications when a CLR crash event occurs.

1. Verify your Email / SMS Configuration. See Enable an Email Server.
2. Create a notification action configured to alert the parties who respond to CLR crashes. See Notification Actions.
3. Create a policy to trigger on CLR Crash under Other Events on the Create Policy window. For the action, choose the notification action you created in step 2. See Configure Policies.

After you create the policy, the Controller sends a notification when the .NET Machine Agent raises CLR crash events.

Analyze and Respond to CLR Crashes

When a CLR crashes, the .NET Machine Agent raises an event in the Controller. The events page on the application dashboard displays the number of server crash events, including CLR crashes, during the selected time range.

1. Click the Server Crashes link in the application dashboard to display a list of all server crash events during the selected time range.
2. Optionally filter on the CLR Crash event type.

Entries for individual CLR crashes display information about the crash event, including the process id and the Windows event log id.

| After a CLR crash, IIS automatically tries to restart the CLR, so one issue frequently causes multiple CLR crash events. |

3. Double-click a CLR Crash type event to display more information in the CLR Crash window:
   - The Summary tab shows the Windows process ID of the w3wp process that crashed and the Windows event log id for the crash. The summary also includes the affected tier and node names.
   - The Details tab displays essential information about the crash.
   - The Actions Executed tab shows any actions the event triggers. See Monitor for CLR crash events.
   - Read and add new comments on the Comments tab. For example, add a comment to the CLR crash event to notify colleagues that you’re triaging the issue.

Disable CLR Crash Event Reporting

The .NET Machine Agent enables CLR crash event reporting by default. Disable it as follows:
1. Edit the agent config.xml as an administrator. See Where to Configure App Agent Properties.
2. Set enabled="false" for the CLR Crash Reporting element. See CLR Crash Reporting Element.

Crash Events the .NET Machine Agent Monitors

The .NET Machine Agent listens on the following Windows event logs for crash events:

- Application Log
- System Log

The agent listens for events logged at event-level Warning and higher on the following sources:

- Application Error
- .NET Runtime
- WAS (Windows Process Activation Service)

The agent listens for events logged at event-level Information and higher on the following source:

- Windows Error Reporting

The agent reports CLR crashes as events to the Controller as follows:

- All w3wp process crashes
- Instrumented Windows service crashes
- Instrumented standalone application crashes

The Controller treats all events as Warning severity, which is not directly related to the Windows event level.
Object Instance Tracking for .NET

On this page:
- Before You Begin
- Enable Object Instance Tracking on a Node
- Identify Memory Usage Problems
- Track Object Instances for Custom Classes

Related pages:
- CLR Memory Diagnostics

When you enable object instance tracking for a node, AppDynamics analyzes the heap to identify classes with the most instances on the heap. AppDynamics tracks the top 20 .NET framework classes and the top 20 application classes based upon the number of instances. Use object instance tracking to identify memory usage trends in classes with large numbers of instances.

Permissions

To enable object instance tracking, you need one of the following permissions:

- Configure Memory Monitoring
- OR
- Configure Agent Properties

For information on AppDynamics Role Based Access Control, see Roles and Permissions.

Before You Begin

- The .NET Agent only supports object instance tracking for .NET 4 and later.
- The .NET Agent Coordinator Service must be the same architecture as the monitored application. For example, a 64-bit coordinator with a 64-bit IIS application.

Enable Object Instance Tracking on a Node

1. From the node dashboard, click the Memory tab.
2. Click the Object Instance Tracking subtab.
3. Click ON.

Once the agent completes the heap analysis, AppDynamics begins to track the top 20 application classes and the top 20 system (core .NET) classes in the heap.

Identify Memory Usage Problems

Use the following guidelines to identify memory usage problems:

- The controller resolution for object instances is one minute, but the .NET Agent agent only sends data every 10 minutes, so .NET memory appears as a series of peaks over time.
- It is normal for the controller to display 0 for the Current Instance Count and Shallow Size in between instance count collection times.
- Hover over a peak to display information about the instance count.
- Look for trends where the peaks increase in size from left to right which may indicate a memory leak.
Track Object Instances for Custom Classes

If you want to track a class that doesn’t appear in the top 20 on the Object Instance Tracking tab, you can configure a specific class to track.

1. On the Object Instance Tracking tab, click **Configure Custom Classes to Track** to navigate to the Configure Instrumentation > Memory Monitoring tab.
2. Click **Add**.
   The Create New Instance Tracker window opens.
3. Leave **Enabled** checked.
4. Enter the fully qualified class name for the instance to track.
5. Click **Save**.
Monitor IIS

On this page:
- IIS Events
- Default IIS Metrics for .NET
- Monitor IIS Application Pools

Related Pages:
- Monitor CLR Crashes

The AppDynamics .NET Agent includes the .NET Machine Agent that runs as part of the AppDynamics.Agent.Coordinator service. Among other things, the .NET Machine Agent regularly gathers IIS performance data and reports it back to the Controller as metrics.

IIS Events

The .NET Machine Agent monitors IIS for shutdown and restart events:
- The agent reports an App Server Restart event and indicates if the restart was graceful or not.
- For non-graceful shutdown, the agent reports a CLR Crash if it detects one.

Default IIS Metrics for .NET

Internet Information Services (IIS) must be installed on the machine for you to view the metrics for IIS, ASP.NET, and ASP.NET applications. The .NET Machine Agent uses Microsoft Windows Performance Counters to gather IIS metrics. In the Controller, you can view preconfigured metrics for IIS in the Metric Browser.

IIS Metrics

From the Metric Browser:
- To view the IIS metrics for a tier, expand Application Infrastructure Performance > <Tier> > IIS.
- To view the IIS metrics for a node, expand Application Infrastructure Performance > <Tier> > Individual Nodes > <Node> > IIS.

AppDynamics reports each metric for the entire tier, each individual application pool, and each individual node as follows:
- Application Infrastructure Performance > <Tier> > IIS = combined for all IIS processes in all Application Pools for this tier
- Application Infrastructure Performance > <Tier> > Application Pools > <application pool name> = combined for all processes in this specific Application Pool
- Application Infrastructure Performance > <Tier> > Individual Nodes > <Node> = metrics for the specific node

ASP.NET Metrics

To view the ASP.NET metrics in the Metric Browser, expand Application Infrastructure Performance > <Node> > ASP.NET.

AppDynamics reports the following ASP.NET metrics:
- Application Restarts
- Applications Running
- Requests Disconnected
- Requests Queued
- Requests Rejected
- Request Wait Time
- Worker Process Restarts

ASP.NET Application Metrics
To view the ASP.NET Application metrics in the Metric Browser, expand **Application Infrastructure Performance > <Node> > ASP.NET Applications**.

AppDynamics reports the following ASP.NET Application metrics:

- Anonymous Requests
- Anonymous Requests/sec
- Cache Total Entries
- Cache Total Hit Ratio
- Cache Total Turnover Rate
- Cache API Entries
- Cache API Hit Ratio
- Cache API Turnover Rate
- Errors Unhandled During Execution/sec
- Errors Total/sec
- Errors During Preprocessing
- Errors During Compilation
- Errors During Execution
- Errors Unhandled During Execution
- Errors Unhandled During Execution/sec
- Errors Total
- Errors Total/sec
- Output Cache Entries
- Output Cache Hit Ratio
- Output Cache Turnover Rate
- Pipeline Instance Count
- Requests Executing
- Requests Failed
- Requests In Application Queue
- Requests Not Found
- Requests Not Authorized
- Requests Succeeded
- Requests Timed Out
- Requests Total
- Requests/sec
- Session State Server Connections Total
- Session SQL Server Connections Total
- Sessions Active
- Sessions Abandoned
- Sessions Timed Out
- Sessions Total
- Transactions Aborted
- Transactions Committed
- Transactions Pending
- Transactions Total
- Transactions/sec

**Monitor IIS Application Pools**

You can monitor the health of IIS application pools for the instrumented .NET nodes in a tier. You can view the information by application pool, machine, and process IDs.

This view enables you to visualize key performance indicators for your infrastructure, such as node health and last CLR restart time.

View the IIS application pools:

1. In the left navigation pane, click **Tiers & Nodes**, then select the **Nodes** tab.
2. In the Show Data drop-down menu, select **IIS App Pools**.

If a machine or application pool name is not available for a .NET node, the .NET Agent creates the "Unknown App Pool" / "Unknown Machine" grouping.
Monitor Node.js Processes

On this page:
- Process Snapshots
- Customizations

Related pages:
- Node.js Agent
- Manage Node.js Process Snapshots

The AppDynamics Node.js Agent helps you monitor Node.js applications in production to determine which applications are slower than normal or producing a lot of errors. It also provides tools for troubleshooting application problems so that you can take action before poor performance becomes apparent to your users.

To access the data, log in to the Controller from a Web browser using your AppDynamics credentials. You can also access the data programmatically using the AppDynamics REST API.

Process Snapshots

The node dashboard for Node.js is like the node dashboards for other app agents except that it also includes a Process Snapshots tab, which you use to access process snapshots for the Node.js process.

In a single-threaded model, such as Node.js, one slow function forces other functions to wait. You can monitor Node.js processes using lists of process snapshots to identify which functions have high CPU times and which consume a lot of memory. From the list, you can select and examine process snapshots to identify exactly which functions in your code are blocking the CPU or leaking memory.

A process snapshot describes an instance of a CPU process on an instrumented Node.js node. It generates a process-wide call graph for a CPU process over a configurable time range. Process snapshots are independent of any running business transactions.

You can monitor process snapshots at the tier level or the node level.

Customizations

The Node.js Agent collects a lot of metrics that do not appear in standard dashboards. You can see all the metrics in the Metric Browser, including special metrics related to Node.js processes under the Application Infrastructure Performance > tier > Node.js tree.

You can create custom dashboards that display any of these metrics using a variety of widgets to create a visual display of information customized for specific types of users in your organization: executives, ops, developers, QA and so forth.

You can also create health rules that stipulate certain levels of acceptable performance and then create policies that automatically send an alert or take an automated action when those rules are violated.
Object Instance Tracking for Node.js

On this page:
- Enable object instance tracking on a node
- Identify memory usage problems
- Track object instances for custom object types
- Examine OIT Metrics in the Metric Browser

The Node.js Agent provides object instance tracking (OIT) for monitoring memory usage.

When you enable object instance tracking for a node, the Node.js Agent attempts to identify the object types with the most instances on the heap. The agent tracks the top 20 object types based upon the number of instances. Use object instance tracking to identify memory usage trends in objects with large numbers of instances.

Enable object instance tracking on a node

1. In the left navigation pane, click Tiers & Nodes.
2. In the Tiers & Nodes window navigate to and select the node that you want to track.
3. Double click or click View Dashboard.
4. In the node dashboard, click the Memory tab.
5. Click the Object Instance Tracking subtab.
6. Click On to start tracking.
   After completing the heap analysis, the agent begins tracking the top 20 object types in the heap.
7. When you are finished tracking, click Off because tracking for a long time can slow down the Node.js process. OIT should be used only for brief periods.

   Although the Node.js OIT UI displays "Class" and "Classes" throughout, the Node.js agent tracks object types, not classes. Substitute "object" for "class" wherever you see "class" for this feature.

Identify memory usage problems

Use the following guidelines to identify memory usage problems:

- The Node.js Agent attempts to report once per minute, but on a heavily loaded, memory-saturated application it could be less often.
- It is normal for the controller to display 0 for the Current Instance Count and Shallow Size between instance count collection times.
- Hover over a peak in the graph to display information about the instance count.
- Look for trends where the peaks increase in size from left to right. These may indicate a memory leak.

Track object instances for custom object types

If you want to track an object that doesn't appear in the top 20 on the Object Instance Tracking tab, you can configure a specific object to track.

1. In the Object Instance Tracking tab, click Configure Custom Classes to Track.
2. In the Define Custom Classes to Track window, click Add.
   The Create New Instance Tracker window opens.
3. Leave Enabled checked.
4. Enter the object for the instance to track.
5. Click Save.
   The configured object is added to the objects to track when OIT is on.

Examine OIT Metrics in the Metric Browser

When enabled, OIT metrics are also reported under the Application Infrastructure Performance > <tier> Individual Nodes > <node>
> **Object Instance Tracking** branch in the Metric Browser.
Node.js Metrics

This topic defines per process metrics that are specific to the Node.js Agent.

You can view the metrics under the following paths in the metric browser navigation tree:

- By tier at Application Infrastructure Performance > <TierName> > Node.js
- By node at Application Infrastructure Performance > <TierName> > Individual Nodes > Nodejs_<NodeName> > Node.js

CPU Usage Metric

- CPU Usage: % busy

Memory Metrics

**Garbage Collection**

- Full GC Per Min: Number of full garbage collection cycles per minute that the V8 JavaScript runtime has performed in the selected time range.
- Incremental GC Per Min: Number of incremental garbage collection cycles per minute that the V8 JavaScript runtime has performed in the selected time range.

**Memory**

- Heap size changed: Total amount of memory reclaimed by the full and incremental garbage collection cycles (as a percentage) in the selected time range.
- Current usage (V8 heap used in MB): Total size of the heap at the current time point. This reports how much memory the node process is using for data storage.
- RSS: Resident Set Size of the Node process. This reports the amount of memory (heap and stack) allocated for the process and in RAM, not swapped out.

I/O Metrics

**Disk**

- KB read per second: KB read from disk per second for the selected time range.
- KB written per second: KB written to disk per second for the selected time range.

**Network**

- Incoming: KB/sec received for the selected time range.
- Outgoing: KB/sec sent for the selected time range.

Socket.io Metrics
- Number of Connections: Number of currently open Socket.IO connections.
- Total Number of Connections: Number of connections that have been opened since the application started.
- Number of Messages Sent/Received: Number of messages that have been exchanged between the application and all connected Socket.IO clients.
- Size of Messages Sent/Received: Average size, in characters, of the messages exchanged. The underlying transport is text only, so non-string messages are JSON serialized to determine their size.

**Event Loop Metrics**

- Average IO Time: Average number of milliseconds per event loop tick spent processing IO callbacks.
- Average Tick Length: Average amount of time between event loop ticks.
- Maximum Tick Length: Shortest amount of time between event loop ticks.
- Minimum Tick Length: Longest amount of time between event loop ticks.
- Tick Count: Number of times the event loop was ticked.
N|Solid Monitoring Data

On this page:
- Metrics
- Process Snapshots Data

N|Solid is an enterprise-grade Node.js runtime produced by Nodesource that provides additional performance metrics about Node.js processes.

This topic describes the additional data available in AppDynamics for N|Solid monitoring.

Metrics

When the Node.js agent is installed on an N|Solid runtime, it collects these additional metrics and makes them available in the AppDynamics controller so they can be used to diagnose latency issues and memory leaks.

You can view the metrics under the following paths in the metric browser navigation tree, by node at Application Infrastructure > Performance > Individual Nodes > Nodejs_<NodeName> > Node.js.

- N|Solid -> 1-minute, 5-minute, and 15-minute load average
- N|Solid -> System Uptime & Process uptime (ms)
- N|Solid -> Memory -> JS heap -> Total Usage (MB)
  - The amount of the heap that is being used by JavaScript.
- N|Solid -> Memory -> V8 heap -> Total Available (MB)
  - Total memory available to the V8 heap.
- N|Solid -> Memory -> Total Available (MB)
  - Total memory available to the process.
- N|Solid -> Memory -> Total Size (MB)
  - Total memory used by the process.
- N|Solid -> Event Loop -> Active Handles:
  - The number of active handles the event loop will process. Handles tend to be longer-lived larger-scale asynchronous operations, such as open sockets or timers. An uptick in this metric could indicate a possible memory leak.
- N|Solid -> Event Loop -> Active Requests:
  - The number of active requests the event loop will process. Requests tend to be shorter-lived smaller-scale operations such as writing to file handles and other file-related operations. This metric provides additional insight into load characteristics.
- N|Solid -> Idle Percent:
  - Time spent waiting for I/O and not running Javascript.
- N|Solid -> Estimated Lag
  - Average amount of time a I/O response may have to wait before being processed.
- N|Solid -> Loops Per Second
  - Number of executions per second of the libuv event loop.
- N|Solid -> Average Tasks:
  - Number of asynchronous Javascript tasks per turn of the loop.
- N|Solid -> Total Count:
  - Total number of event loop turns.

Process Snapshots Data

You can view details on asynchronous activities in the Node.js event loop for N|Solid processes in Process Snapshots > Async Activity. The following categories of event loop activity are displayed in Async Activity:

- Active Handles: active handles that the event loop will process
- Active Requests: the number of active requests the event loop will process
- Pending: lower level asynchronous activity

The following information is displayed for each property:

- Type: The property type (e.g., TCP socket connection, setTimeout)
- Location: The source location associated with the activity, when available
Remote Services

What is a Remote Service?

A remote service is a process that resides outside of the application server that provides a service to the application. An example of remote service is a web service, message queue, or caching server.

AppDynamics automatically detects many common types of remote services are automatically when instrumented nodes make outbound requests. For more details on backend support by app agent type, refer to the supported environments page for your agent type under Install App Server Agents, such as Java Supported Environments.

To monitor call performance to a service, first make sure it shows up in the remote services list, which you can access by clicking the Remote Services link in the Applications menu. If a service you expect to see listed does not appear in the list, make sure the backend detection configuration is configured appropriately for your environment, as described in Backend Detection Rules.

Monitoring Information for Remote Services

AppDynamics monitors the overall performance of calls to a remote service, as well as the performance of those calls from within specific business transactions.

Metrics for remote services are presented in the Controller UI in the following locations:
- Business transaction metrics: The Transaction flow map shows the metrics for a specific business transaction for a specific service
- Tier metrics: The Tier flow map shows the metrics for all calls from a tier to the specified service
- Remote service metrics: The Application flow map and the Remote Services Dashboard show the overall remote service metrics across the application (all business transactions)

View Remote Service Performance on Flow Maps

Remote services detected during the specified time window appear on the Application Dashboard flow map. You can view the detected services in the context of the entire application’s transaction flow. The application flow map displays calls per minute and average response time for calls made to remote services. These metrics include all calls made from a specific tier to a service across all business transactions. The tier and node flow maps display the same metric in their respective contexts.

The detected remote services show up on the Tier dashboard flow map. You can view the detected services in the context of the traffic on this specific tier.

For business transactions involving calls to remote services, the services appear on the Business Transaction dashboard flow map. You can view the detected services in the context of the traffic for this specific business transaction. The transaction flow map shows the average time spent in remote service calls for the business transaction.
**View Discovered Remote Services**

The Remote Services list shows all detected remote services along with key performance indicators. Services that are not active are removed after a configurable time period. See [Stale Remote Service Removal](#).

From the Remote Services list, you can select a service and click **View Dashboard** to see the Remote Service Dashboard. The dashboard displays a Database Flow Map, backend properties, and graphs of the key performance indicators (KPIs). The properties indicate how the service is identified and determine how it shows in the flow map and how the metrics are aggregated. For a discussion of baselines and how they are used and configured, see [Dynamic Baselines](#).

The Remote Services Dashboard has two tabs and an action option menu:

- **Dashboard**: Displays the flow map showing traffic from the calling tier to the remote service, the backend properties used for auto-detection and naming, and key performance indicators.
- **Slowest Remote Services Calls**: Lists up to ten calls to the service with the longest execution time, by tier and for all tiers.
- From the Action menu, you can also rename a backend, delete a backend, or resolve a backend to a tier.

**Slow Remote Service Calls**

AppDynamics displays a list of the slowest remote service calls with call details. Click **Troubleshoot > Slow Response Times > Slowest DB & Remote Service Calls** tab to view specific call details and related business transaction snapshots that can help you to troubleshoot.

The **Slowest DB & Remote Service Calls** tab lists up to ten calls to a remote service with the longest execution time over the selected time frame, by tier and for all tiers, and relevant metrics.

**Max Time** determines which calls are displayed in the **Slowest DB & Remote Service Calls** list. **Max Time** must exceed 50 ms before AppDynamics tracks the call as a potential candidate for this list. App agents aggregate and report call data to the Controller every 15 minutes.

If Query, Queue or URL are not specified in the custom exit call, the details for the slowest service call will not be displayed.
Stale Remote Service Removal

A stale remote service is a service for which AppDynamics has previously captured metrics, but which has experienced no metric activity for a certain time period. If a backend is not called for 30 days (in other words, the call per minute metric is zero for 30 days) AppDynamics considers it a stale backend and removes it as a monitored artifact.

After a backend is removed, its metrics no longer appear in the Metric Browser and health rules that reference those metrics do not fire. You should remove any health rules conditions that reference metrics in stale backends.

For an on-premises Controller, you can configure the interval by setting the `backend.permanent.deletion.period` property in the Administration Console. Reducing the period may be advisable for large installations, since the maximum number of backends removed in one pass is 50. The minimum interval is one week. Alternatively, you can disable automatic removal by setting the interval to 0.
Group Remote Services on Flow Maps

You can group remote services—or backends—in the flow map configuration to have two or more services of the same type to appear as a single icon in the flow map.

Backend grouping can improve readability and the focus of the flow map, for example, if individual databases or remote service instances are not important to the flow map users.

The following screenshot shows grouped databases on a flow map:

If you want to ungroup backends on a flow map, right-click the backend group and choose **Ungroup Backends**.

Grouped backend services must meet the following conditions:

- All backends must be of the same type
- All backends are called by the same tiers
- No backend in the group calls other backends or calls into other tiers

The following example shows the scenarios in which a set of backends could be grouped.
You can configure backend grouping in the **Databases & Remote Services** tab of the **Configure Flow Map** dialog.

You can set a few parameters around grouping, such as the minimum number of backends must be present before they are grouped. For example, if you set the minimum to 4 and only three backends are detected, they are not grouped.

You can exclude a specific backend server from ever being grouped by the flow map, regardless of the grouping configuration, by checking the **Ungroupable** checkbox for that backend in the Visibility and Grouping list.

If there are specific database or remote services that you do not want to see or group in the flow map, check the **Hidden** or **Ungroupable** check box.
Monitor Databases

On this page:
- Measuring Database Performance
- View Database Performance on Flow Maps
- Resolving Unexpected Databases on the Flow Map
- View Discovered Databases
- Slow Database Calls

Related pages:
- Backend Detection Rules
- Call Graphs
- Access Database Visibility from Application Monitoring Views

AppDynamics can monitor the performance of database calls made by instrumented applications, as described here.

Measuring Database Performance

AppDynamics collects metrics for database calls and response times at these levels:

- Business transaction metrics: The metrics for a specific business transaction for a specific database are visible on the transaction flow map.
- Tier metrics: The metrics for all calls from a tier to the specified database are visible on the tier flow map.
- Database call metrics: The overall database access metrics across the application (all business transactions) are visible on the application flow map and the Database Calls dashboard.
- Integrated database metrics with Database Visibility:
  - When a database Collector has been set up in AppDynamics Database Visibility, you can link to that product from the Application and Database Calls dashboards. If you have already associated the database server with a Database Collector in Database Visibility, the Database Visibility UI opens. If not, you are prompted to link the database server to a Collector already configured in Database Visibility. Also, the user must have edit permission to the corresponding collector configured in Database Visibility, to map/unmap it with the database server.

- If a relational database backend of a Java application has the same hostname, port number, and database type as a database server already configured in a database Collector, the Oracle backend is automatically matched with the
corresponding Database Visibility Collector. The Snapshot correlation view, which is available for relational database backends, shows the details of queries, clients, sessions, and schemas when the snapshot was captured.

- When AppDynamics Pro has been integrated with AppDynamics for Databases, you can link to that product by right-clicking on a database from the database list or from the database icon on any flow map. See AppDynamics for Databases.

By default, many databases and data stores are automatically detected when calls are made from nodes instrumented with AppDynamics app agents.

To monitor call performance to a database, confirm that it appears in the Databases Calls list and has its own Database Calls dashboard. If a database is not appearing, check the configuration.

View Database Performance on Flow Maps

Databases detected during the specified time window show up on the Application Dashboard flow map, where you can view them in the context of the entire application's transaction flow. The application flow map displays calls per minute and average response time for calls made to databases. These metrics include all calls made from a specific tier to a database across all business transactions. The tier and node flow maps display a similar metric aggregating data from calls across all business transactions by tier or node respectively.

The detected databases appear on the Tier Flow Map, where you can view them in the context of the traffic on this specific tier.

For business transactions involving calls to databases, the databases appear on the Transaction Flow Map, where you can view them in the context of the traffic for this specific business transaction. The transaction flow map shows the average time spent in database calls for the business transaction.

Drill Down on Database Performance
In addition to seeing monitored databases in the flow map, you can view the following information:

- Business transactions that make the most database calls. To see this, navigate to Database Business Transactions Window.
- Queries executed the most number of times by business transactions. To see this, double click on the business transaction and open the DB Queries tab.

### Resolving Unexpected Databases on the Flow Map

AppDynamics can sometimes reveal unexpected connections from an application to a database on the flow map. If this occurs for you, try the following to determine why this database appears:

- From the left navigation menu, select Tiers & Nodes > Databases. Select the Slowest Database Calls tab and drill down into the snapshots to see the code that is calling the database. See To troubleshoot slow database and remote service calls.
- Run a diagnostic session to capture some transaction snapshots and look for calls to the database. See Diagnostic Sessions and Transaction Snapshots. If you have integrated AppDynamics for Databases, you can select a transaction snapshot that involves running SQL on an Oracle database and from the Transaction Flow Map you can link to AppDynamics for Database to see all queries executed in the SQL session that are associated with that transaction snapshot. See Use AppDynamics Pro with AppDynamics for Databases.
- Are any exceptions thrown when the database is seen? If so, look for error snapshots that point to the exception trace.

### View Discovered Databases

The database list shows all detected databases along with key performance indicators. Stale databases can be configured to be automatically removed.

From the database list, you can select a database and click View to see the Database Calls dashboard. The dashboard displays a Database Flow Map, database properties, and graphs of the key performance indicators (KPIs). The database properties indicate how the agent identifies the database and control how it shows in the display map and how the metrics are aggregated. For a discussion of baselines and how they are used and configured, see Dynamic Baselines.

The database dashboard has two tabs and an action options menu:

- Dashboard: Displays the flow map showing traffic from the calling tier to the database, the backend properties used for auto-detection and naming, and key performance indicators.
- Slowest Database Calls: Lists up to ten calls to the database with the longest execution time, by tier and for all tiers. See Slow Database Calls.

The Action menu provides additional actions:

- Rename Backend: Renames the database.
- Resolve Backend to Tier: Associates the database with the tier that you select so that the backend appears in the grid view of the tier and not as an independent component ("unresolved backend") on the application dashboard flow map. You can reverse this operation from the Configure Backends resolving to this Tier item in Actions menu in the Tier dashboard.
- Delete Backends: Removes instances of the database from the controller and all agents. An agent can re-discover the database and register it with the controller.

You can access the Database Server List by clicking Servers > Databases.

### Slow Database Calls

AppDynamics displays a list of the slowest database calls. For each call, you can view specific call details and related business transaction snapshots. The list shows up to ten database calls with the longest execution time over the selected time frame, by tier and for all tiers. Each call shows the following information:

- **Call**: SQL Query
- **Avg. Time per Call (ms)**: the average time per call in milliseconds
- **Number of Calls**: the number of calls executed during the time range
- **Max Time (ms)**: the maximum execution time in milliseconds
- **View snapshots**: a link to view existing transaction snapshots

App agents aggregate and report call data to the Controller every 15 minutes. **Max Time** determines which calls are displayed in the Slowest Database Calls list. For example for JDBC calls, **Max Time** must exceed 10 ms before AppDynamics tracks the call as a potential candidate for this list.

Slowest database calls are defined as:
- Max Time greater than 10 ms
- Top ten slowest
- Reported every 15 minutes

Viewing slowest database calls

To see slowest database calls, click **Troubleshoot > Slow Response Times > Slowest DB and Remote Service Calls.**

On this page, you can do the following:

- If transaction snapshots are available for a call, you can click the **View Snapshots** link in the **Snapshots** column to select a snapshot and drill down to the root cause of the slowness.
- You can view explain plans by selecting a call and clicking **View Details.** In the dialog box, click **Explain Plans.** If parameter values are filtered out from the captured SQL, the Explain Plan feature is disabled.

### NoSQL
AppDynamics displays NoSQL databases as Remote Services. See **Remote Services.**
Resolve Remote Services to Tiers

Resolving remote services to a tier can put the metrics for the services into an element of the AppDynamics application model that may provide a more useful view of the activities of the services in flow maps.

The tier may already exist in the business application model or you can create a new one. If it exists, after you resolve the backend to a tier it no longer appears as a backend icon in the flow maps.

AppDynamics automatically resolves some backends to tiers. In general, these are custom backends or HTTP, RMI, Thrift or web services, which are usually logically linked to a tier. If you want to see the backend on its own, you can unresolve it to display it on the flow map.

When to Resolve Backends to a Tier

You may want to resolve a backend to a tier if it represents a more logical view of your environment, for example, if a database runs on the same machine as an application server. Another reason to resolve backends to a tier is to combine similar backends.

For example, if there appear to be multiple databases that are actually the same database using different namespaces, the default flow map is likely to display the databases separately. You can resolve the database backends to a new tier of type "Database Server" so they display as one and the tier metrics are together. If a backend is taken out, the historical data for the backend is retained at the tier level.

Resolving Backends to Tiers

You can resolve a backend to a tier from the backend's dashboard page, either its Remote Service Dashboard or Database Dashboard. In the database dashboard, from the Actions menu, click Resolve Backend to Tier and select an existing tier or create a new one. Similarly, you can see backends that are associated with a tier and unresolve a backend from the action menu for the Tier dashboard.

Deleting the backend from the tier causes it to re-appear as an unresolved backend in the flow map or resolve it to a different tier.
Troubleshooting Applications

Accessing Troubleshooting

When starting to troubleshoot an application problem, a good place to start is in the Troubleshoot section of the UI. You can access from the left-hand navigation pane of the Controller UI in an application context.

The area includes pages for analyzing slow response times, errors and exceptions, and health rule violations. It also provides access to war rooms, an area of the UI dedicated to troubleshooting a specific problem.

Need more help?

If slow response time persists even after you’ve completed the steps outlined above, you may need to perform deeper diagnostics. If you can’t find the information you need on how to do so in the AppDynamics documentation, consider posting a note about your problem in the Community Discussion Boards. These discussions are monitored by customers, partners, and AppDynamics staff.
Slow Response Times

You may become aware of slow response times in a variety of ways. For example, you may receive a notification email or SMS based on a health rule violation or see performance indicators in flow maps or transaction scorecards.

The following guidelines describe a strategy for diagnosing and troubleshooting slow response times in AppDynamics.

Step 1: Determine if business transactions are slow or stalled

Slow, very slow, and stalled transactions are detected based on your transaction thresholds.

How do I know?

1. Make sure that the time frame selected in the Controller UI is likely to encompass the time when the performance issue occurred. If it's a continuing condition, you can keep the time frame relatively brief.
2. Click Troubleshoot > Slow Response Times.
3. Click the Slow Transactions tab if it is not selected.
   The upper pane of the Slow Transactions tab displays a graph of the slow, very slow, and stalled transactions for the time period specified in the Time Range drop-down menu. The lower pane lists transaction snapshots transaction instances.

   If you see one or more slow transaction snapshots on this page, the answer to this question is Yes. Otherwise, the answer is No.

No – Go to Step 2 and check for slow backends.

Yes – You have one or more slow or stalled transactions. Now you can drill down to find the root cause.

1. In the lower pane of the Slow Transactions tab, click the Exe Time column to sort the transactions from slowest to fastest.
2. Select a snapshot from the list and click View Transaction Snapshot.
3. Review the Potential Issues list to see the longest-running method and SQL calls in the transaction.
4. Click a potential issue and click Drill Down into Call Graph to go directly to that point in the call graph, or click Drill Down in the transaction flow map pane to see the complete set of call graph segments retained for this transaction.
4. View the Time (ms) column to see how long this method execution takes relative to the transaction execution time. In this example, the selected method is taking 96.3% of the transaction execution time.

5. View the Time (ms) column to see how long this method execution takes relative to the transaction execution time. In this example, the selected method is taking 96.3% of the transaction execution time.

6. In the information details pane, note the class, method, and if available line number represented by the execution segment. This information gives you or the application developer responsible for the code a starting point for troubleshooting this code issue.

If there are multiple slow or stalled transactions, repeat this step until you have resolved them all. However, there may be additional problems you haven’t resolved. Continue to Step 2.

**Step 2: Slow DB or remote service calls?**

AppDynamics collects metrics about the performance of business transaction calls to the databases and remote servers from the
instrumented app servers. You can drill down to the root cause of slow database and remote service calls.

Is there one or more slow DB or remote service call?

**How do I know?**

Click **Troubleshoot > Slow Response Times**, then click the **DB & Remote Service Calls** tab.

![DB & Remote Service Calls Tab](image)

If you see one or more slow calls on either the SQL Calls or Remote Service Calls tabs of this page, the answer to this question is Yes. Otherwise, the answer is No.

No – Go to Step 3 to determine if the problem is affecting all nodes in the slow tier.

Yes – You have one or more slow database or remote service calls, and need to drill down to find the root cause.

**Step 1:** On the SQL Calls tab of the transaction snapshot, sort the SQL Calls by Avg. Tims(ms). If you have a slow database call, you can click the database call to gain information about the call.

If you have Database Monitoring, you can right-click the database on the Application, Tier, Node or Backend Flow Map, and choose View in **Database Monitoring** to delve into any database problems.

If you have AppDynamics for Databases, you can right-click the database on the Application, Tier, Node or Backend Flow Map, and choose **Link to AppDynamics for Databases**. You can use AppDynamics for Databases to diagnose database issues.

If you have correlated snapshots between Java applications and Oracle databases, you can drill down into the Oracle database on the Transaction Snapshot to view database details captured during the snapshot.
On the Remote Service Calls tab, you can sort the queries by Avg. Tims/ms and then click the slow call. You can then click Drill Down into Downstream Call to gain further insight into the methods of the service call. Sort the methods by the Time (ms) column, and then investigate any slow methods by selecting the method and then clicking Details.

To see transaction snapshots for the business transaction that is correlated with a slow call:

a. Make sure that the time frame selected in the Controller UI is likely to encompass the time when the performance issue occurred. If it’s a continuing condition, you can keep the time frame relatively brief.

b. Click Troubleshoot > Slow Response Times.

c. Click the Slowest DB & Remote Service Calls.

d. From one of the slow calls in the Call column, click View Snapshots.

e. You can then view the transaction snapshots correlated with that slow call in the Correlated Snapshots dialog.
Step 3: Affects all nodes?

Is the problem affecting all nodes in the slow tier?

How do I know?

In the Application or Tier Flow Map, click the tier or node icon to see a quick overview of the health of each node in the tier.

If all the nodes are yellow or red, the answer to this question is Yes. Otherwise, the answer is No.

Yes – Go to Step 4.
No – The problem is either in the node's hardware or in the way the software is configured on the node. If only one node in a tier is affected, the problem is probably not related to the application code.

1. In the left navigation pane, click Tiers & Nodes.
2. Expand the Tier in the right pane and double-click the affected node to open its Node Dashboard.
3. Click the Memory tab. Explore each of the available tabs to determine if you need to add memory to the node, configure additional memory for the application, or take some other corrective action.
4. Click the Server tab. If the Hardware tab indicates a hardware related problem, contact your IT department.

You have isolated the problem.

Step 4: Most business transactions?

Is the problem affecting most of the business transactions?

How do I know?

On the Application Dashboard, look at the Business Transaction Health pane on the right side of the screen.
1. If the bar representing business transaction health is primarily yellow or red, the answer to this question is Yes. Otherwise, the answer is No.

   No –

   1. In the left navigation pane, click Business Transactions.
   2. Sort by Health, Transaction Score or other column headings to find the business transaction that is experiencing issues.
   3. Double-click the problematic business transaction to see its dashboard, then use the tabs to diagnose the problem.

   You have isolated the problem.

**Need More Help?**

If you've tried to diagnose the problem using the previous steps and haven't found the problem, see additional information for your specific agent:

- Troubleshooting Applications
- Slow Response Times for .NET
- Troubleshoot Slow Network Requests from Mobile Applications
- Monitor Databases
Errors and Exceptions

AppDynamics Application Intelligence Platform captures and presents information on business transaction errors in the monitored environment.

At a high-level, a business transaction is considered to have an error if its normal processing has been affected by a code-level exception or error event, including custom error events based on methods you specify.

Viewing Error and Exception Information

The Controller UI presents information on errors and exceptions in various places in the UI, including in transaction snapshots, metrics, and dashboards.

The informational popups for tiers in flow maps have an error tab that displays error rate metrics for the tier:

On the application and tier flow maps, the error rate is for all business transactions. On the business transaction flow map, errors apply only to the current business transaction.

The Metric Browser includes Error metrics:
The Troubleshoot > Errors page shows all error transactions. The page contains two tabs, one for transaction errors and one for Exceptions.

The tabs show information on the rate of errors or exceptions, and lets you drill down to the error or exception for more information, as shown:

The following sections provide more information on the contents of the tabs.

What is a Business Transaction Error?

All transaction errors that have been detected according to the configured error detection rules in the selected time frame of the Controller UI appear in the Error Transactions tabs of the Errors page.

By default, AppDynamics considers a business transaction to be in error if it detects one of the following types of events in the context of the transaction:
An unhandled error or a fatal exception. An exception that is thrown and never caught or caught after the business transaction terminates results in a transaction error, and the exception is presented in AppDynamics. An exception that is thrown and caught within the context of the business transaction is not considered a transaction error and the exception is not captured in AppDynamics.

An exception caught in an exit call, such as a web service or database call.

An HTTP error response, such as a status code 404 or 500 response.

A custom-configured error method and error message.

Error detection configuration is described in Error Detection.

Errors that occur on a downstream tier that are not propagated to the originating tier do not result in a business transaction error. If the originating client receives a 200 success response, for example, the business transaction is not considered an error. The error contained within the downstream tier does count against the Error Per Minute metric for the continuing segment.

When a business transaction experiences an error, it is counted only as an error transaction. It is not also counted as a slow, very slow or stalled transaction, even if the transaction was also slow or stalled.

What is an Exception in AppDynamics?

Code exceptions are a common cause of business transaction errors. The Exceptions tab in the Errors page shows an aggregated view of the exceptions across all transactions. For purposes of this view, AppDynamics considers the following types of incidents to be exceptions:

- Any exception logged with a severity of Error or Fatal (using Log4j, java.util.logging, Log4Net/NLog, or another supported logger). This applies even if the exception occurs outside the context of a business transaction, in which case the exception type is specified as Application Server.
- HTTP errors that do not occur in the context of a Business Transaction.
- Error page redirects.

Exceptions that are thrown and handled within a business transaction are not captured by AppDynamics and do not appear in the Exceptions tab.

When troubleshooting errors, notice that the number of business transaction errors does not necessarily correspond to the number of exceptions in a given time frame. A single transaction that counts as an error transaction can correspond to multiple exceptions. For example, as the transaction traverses tiers, it can generate an exception on each one. Troubleshooting an error typically involves finding the exception closest to the originating point of the error.

If a stack trace for the exception is available, you can access it from the Exception tab in the Controller UI. A stack trace is available for a given exception if the exception was passed in the log call. For example, a logging call in the form of logger.log(Level.ERROR, String msg, Throwable e) would include a stack trace, whereas a call in the form of logger.log(Level.ERROR, String msg) would not.

Agent Errors

The Java Agent differentiates between agent internal errors and application errors. By default, agent internal errors no longer set off health rule violations. You can view agent internal errors in the following Metric Browser path: Application Infrastructure Performance > <tier> > Agent > Internal Errors.

Error and Exception Limits

AppDynamics limits the number of registered error types (based on error-logging events, exception chains, and so on) to 4000. It maintains statistics only for registered error types.

Reaching the limit generates the CONTROLLER_ERROR_ADD_REG_LIMIT_REACHED event. While it is possible to increase the limit, we recommend refining the default error detection rules to reduce the number of error registrations to have the error you are not interested in capturing ignored.

For more information, see information on configuring log errors and exceptions to ignore in Error Detection.

Configuring Errors and Exceptions

AppDynamics automatically recognizes errors and exceptions for many common frameworks. You can customize the default error detection behavior as needed, for example, if you use your own custom error framework.
For information on configuring errors and exceptions, see Error Detection.
You can learn that your application's response time is slow in the following ways:

- You receive an alert: If you have received an email alert from AppDynamics that was configured through the use of health rules and policies, the email message provides a number of details about the problem that triggered the alert. See information about Email Notifications in Notification Actions. If the problem is related to slow response time, see Initial Troubleshooting Steps.
- You view the Application Dashboard for a business application and see slow response times.
- A user reported slow response time that relates to a particular business transaction, for example, an internal tester reports “Searching for a hotel is slow”.

**Initial Troubleshooting Steps**

In some cases, the source of your problem might be easily diagnosed by choosing Troubleshoot > Slow Response Times in the left navigation pane. See Slow Response Times.

**.NET Resource Troubleshooting**

If you've tried to diagnose the problem using those techniques and haven't found the problem, use the following troubleshooting approaches to find other ways to determine the root cause of the issue.

**Step 1 - CPU saturated?**

Is the CPU of the CLR saturated?

How do I know?

1. Display the Tier Flow Map.
2. Click the Nodes tab, and then click the Hardware tab.
3. Sort by CPU % (current).
If the CPU % is 90 or higher, the answer to the question in Step 4 is Yes. Otherwise, the answer is No.

Yes – Go to Step 2

No – Review various metrics in the Metric Browser to pinpoint the problem.

In the left navigation pane, click Servers > Tiers & Nodes > slow tier. Review these metrics in particular:

- ASP.NET -> Application Restarts
- ASP.NET -> Request Wait Time
- ASP.NET -> Requests Queued
- CLR -> Locks and Threads -> Current Logical Threads
- CLR -> Locks and Threads -> Current Physical Threads
- IIS -> Number of working processes
- IIS -> Application pools -> <Business application name> -> CPU%
- IIS -> Application pools -> <Business application name> -> Number of working processes
- IIS -> Application pools -> <Business application name> -> Working Set

You have isolated the problem and don’t need to continue with the rest of the steps below.

### Step 2 - Significant garbage collection activity?

**How do I know?**

1. Display the Tier Flow Map.
2. Click the Nodes tab, and then click the Memory tab.
3. Sort by Time Spent on Collections (%) to see what percentage of processing time is being taken up with garbage collection activity.
1. If Time Spent on Collections (%) is higher than acceptable (say, over 40%), the answer to the question in Step 5 is Yes. Otherwise, the answer is No.

Is there significant garbage collection activity?

Yes – Go to Step 3.

No – Use your standard tools to produce memory dumps; review these to locate the source of the problem.

You have isolated the problem and don’t need to continue with the rest of the steps below.

Step 3 - Memory leak?

Is there a memory leak?

1. From the list of nodes displayed in the previous step (when you were checking for garbage collecting activity), double-click a node that is experiencing significant GC activity.

2. Click the Memory tab, then review the committed bytes counter and the size of the Gen0, Gen1, Gen2 and large heaps.
If memory is not being released (one or more of the above indicators is trending upward), the answer to the question in Step 6 is Yes. Otherwise, the answer is No.

Yes – Use your standard tools for troubleshooting memory problems. You can also review ASP.NET metrics; click Tiers & Nodes > slo tier > ASP.NET.

No – Use your standard tools to produce memory dumps; review these to locate the source of the problem.

Whether you answered Yes or No, you have isolated the problem.
Java Resource Issues

On this page:

- Step 1. CPU saturated?
- Step 2. Significant garbage collection activity?
- Step 3. Memory leak?
- Step 4. Resource leak?

The following troubleshooting guidelines can help you to determine the root cause of many Java-related issues.

**Step 1. CPU saturated?**

Is the CPU of the JVM saturated?

**How do I know?**

1. Display the Tier Flow Map.
2. Click the **Nodes** tab, and then show Hardware data.
3. Sort by CPU % (current).

If the CPU % is 90 or higher, the answer to this question is Yes. Otherwise, the answer is No.

Yes – Go to **Step 2**.

No – The issue is probably related to a custom implementation your organization has developed. Take snapshots of the affected tier or node(s) and work with internal developers to resolve the issue.

**Step 2. Significant garbage collection activity?**

Is there significant garbage collection activity?

**How do I know?**

1. Display the Tier Flow Map.
2. Click the **Nodes** tab, and then click the Memory tab.
3. Sort by GC Time Spent to see how many milliseconds per minute is being spent on GC; 60,000 indicates 100%.
4. If GC Time Spent is higher than 500 ms, the answer to the question in Step 5 is Yes. Otherwise, the answer is No.
Result

Yes – Go to Step 3.
No – Go to Step 4.

Step 3. Memory leak?

Is there a memory leak?

How do I know?

1. From the list of nodes displayed in the previous step (when you were checking for Garbage Collecting activity), double-click a node that is experiencing significant GC activity.
2. Click the Memory tab, then scroll down to display the Memory Pool graphs at the bottom of the window.
3. Double-click the Old Gen memory pools chart.

If memory is not being released (use is trending upward), the answer to this question is Yes. Otherwise, the answer is No.

Result

Yes – Use various AppDynamics features to track down the leak. One useful tool for diagnosing a memory leak is object instance tracking, which lets you track objects you are creating and determine why they aren't being released as needed. Using object instance tracking, you can pinpoint exactly where in the code the leak is occurring. For instructions on configuring object instance tracking, as well as links to other tools for finding and fixing memory leaks, see Need more help? below.

No – Increase the size of the JVM. If there is significant GC activity but there isn't a memory leak, then you probably aren't configuring a large enough heap size for the activities the code is performing. Increasing the available memory should resolve your problem.

Whether you answered Yes or No, you have isolated the problem.

Step 4. Resource leak?

Is there a resource leak?

How do I know?

1. In the left Navigation pane, go to (for example) Metric Browser > Application Infrastructure Performance > TierName >
1. Individual Nodes > NodeName > JMX > JDBC Connection Pools > PoolName.
2. Add the Active Connections and Maximum Connections metrics to the graph.
3. Repeat as needed for various pools your application is using.

If connections are not being released (use is trending upward), the answer to the question in Step 7 is Yes. Otherwise, the answer is No.

Result

Yes – To determine where in your code resources are being created but not being released as needed, take a few thread dumps using standard commands on the problematic node. You can also create a diagnostic action within AppDynamics to create a thread dump; see Thread Dump Actions in Diagnostic Actions.

No – Restart the JVM. If none of the above diagnostic steps addressed your issue, it's possible you're simply seeing a one-time unusual circumstance, which restarting the JVM can resolve.
Java Memory Leaks

The garbage collection feature of the JVM greatly reduces the opportunities to introduce memory leaks into a codebase. However, because garbage collection does not eliminate memory leaks completely, AppDynamics includes Automatic Leak Detection for supported JVMs.

Permissions
To enable Automatic Leak Detection, you need the Configure Agent Properties permission.
To start an On Demand Capture Session, you need the Advanced Agent Operation permission.
For information on AppDynamics Role Based Access Control, see Roles and Permissions.

Overview of Automatic Leak Detection
You can access Automatic Leak Detection on the Memory tab of the Node Dashboard. Automatic Leak Detection is disabled by default because it increases overhead on the JVM. You should enable leak detection mode only when you suspect a memory leak problem. Turn off Automatic Leak Detection after you identify the cause for the leak.

Automatic Leak Detection uses On Demand Capture Sessions to capture actively used collections, any class that implements JDK Map or Collection interface during the capture period. The default capture period is 10 minutes.

AppDynamics tracks every Java collection that meets the following criteria:

- The collection has been alive for at least 30 minutes.
- The collection has at least 1000 elements.
- The collection Deep Size is at least 5 MB. The agent calculates Deep Size by traversing recursive object graphs of all the objects in the collection.

The following node properties define the defaults for leak detection criteria:

- minimum-age-for-evaluation-in-minutes
- minimum-number-of-elements-in-collection-to-deep-size
- minimum-size-for-evaluation-in-mb

See App Agent Node Properties for more information.

The Java Agent tracks the collection and identifies potential leaks using a linear regression model. You can identify the root cause of the leak by tracking frequent access to the collection over a period of time.

After it qualifies a collection, AppDynamics monitors the collection size for a long term growth trend. Positive growth indicates the collection is the potential source of a memory leak.

After AppDynamics identifies a leaking collection, the Java Agent automatically triggers diagnostics every 30 minutes. The diagnostics capture a shallow content dump and activity traces of the code path and business transactions that access the collection. You can drill down into any leaking collection monitored by the agent, to manually trigger Content Summary Capture and Access Tracking sessions.

You can also monitor memory leaks for custom memory structures. Typically custom memory structures are used as caching solutions. In a distributed environment, caching can easily become a prime source of memory leaks. It is therefore important to manage and track memory statistics for these memory structures. To do this, you must first configure custom memory structures. See Custom Memory Structures for Java.
Workflow to Troubleshoot Memory Leaks

Use the following workflow to troubleshoot memory leaks on JVMs that have been identified with a potential memory leak problem:

1. Monitor memory for potential JVM memory leaks
2. Enable automatic leak detection
3. Start on demand capture session
4. Detect and troubleshoot leaking condition

These steps are described in the following sections.

Monitor Memory for Potential JVM Leaks

Use the Node Dashboard to identify the memory leak. A possible memory leak is indicated by a growing trend in the heap as well as the old/tenured generation memory pool.

An object is automatically marked as a potentially leaking object when it shows a positive and steep growth slope.

The Automatic Memory Leak dashboard shows:

- Collection Size: The number of elements in a collection.
- Potentially Leaking: Potentially leaking collections are marked as red. You should start diagnostic sessions on potentially leaking objects.
- Status: Indicates if a diagnostic session has been started on an object.
- Collection Size Trend: A positive and steep growth slope indicates a potential memory leak.

**Tip:** To identify long-lived collections compare the JVM start time and Object Creation Time.

If you cannot see any captured collections, ensure that you have correct configuration for detecting potential memory leaks.

Enable Memory Leak Detection

Memory leak detection is available through the Automatic Leak Detection feature. Once the Automatic Leak Detection feature is turned on and a capture session has been started, AppDynamics tracks all frequently used collections. Therefore, using this mode results in higher overhead.

Turn on Automatic Leak Detection mode only when a memory leak problem is identified and then click **Start On Demand Capture Session** to start monitoring frequently used collections and detect leaking collections.

Turn the capture session and the leak detections modes off after you have identified and resolved the leak.

To achieve optimum performance, start diagnosis on one individual collection at a time.
Troubleshoot Memory Leaks

After detecting a potential memory leak, troubleshooting the leak involves performing the following three actions:

- Select the Collection Object that you want to monitor
- Use Content Inspection
- Use Access Tracking

Select the Collection Object to Monitor

On the Automatic Leak Detection dashboard, right-click the class name and click Drill Down.

For performance reasons, start the troubleshooting session on a single collection object at a time.

Use Content Inspection

Content Inspection identifies which part of the application the collection belongs to so that you can start troubleshooting. It allows monitoring histograms of all the elements in a particular collection.

Enable Automatic Leak Detection by starting an On Demand Capture Session, select the object you want to troubleshoot, and then follow the steps listed below:

1. Click the Content Inspection tab.
2. Click Start Content Summary Capture Session to start the content inspection session.
3. Enter the session duration. Allow at least 1 – 2 minutes for data generation.
4. Click Refresh to retrieve the session data.
5. Click on the snapshot to view details about an individual session.

Use Access Tracking

Use Access Tracking to view the actual code paths and business transactions accessing the collections object.

As described above in Workflow to Troubleshoot Memory Leaks, enable Automatic Leak Detection, start an On Demand Capture Session, select the object you want to troubleshoot, and then follow the steps listed below:

1. Select the Access Tracking tab
2. Click Start Access Tracking Session to start the tracking session.
3. Enter the session duration. Allow at least 1-2 minutes for data generation.
4. Click Refresh to retrieve session data.
5. Click on the snapshot to view details about an individual session.

The troubleshooting information pane shows the Java stack trace associated with the session. By default, the stack trace is shown to a depth of 10 lines. If you would like to temporarily increase the number of lines captures, you can use the maximum-activity-trace-stack-depth Java Agent property described on App Agent Node Properties Reference.

Increasing the stack trace depth can consume a significant amount of system resources. Be sure to remove the property or set it back to the default value of 10 after you have captured the desired information.
Java Memory Thrash

Memory thrash is caused when a large number of temporary objects are created in very short intervals. Although these objects are temporary and are eventually cleaned up, the garbage collection mechanism may struggle to keep up with the rate of object creation. This may cause application performance problems. Monitoring the time spent in garbage collection can provide insight into performance issues, including memory thrash.

For example, an increase in the number of spikes for major collections either slows down a JVM or indicates potential memory thrash. Use object instance tracking to isolate the root cause of the memory thrash. To configure and enable object instance tracking, see Object Instance Tracking for Java.

AppDynamics automatically tracks object instances for the top 20 core Java (system) classes and the top 20 application classes.

The Object Instance Tracking subtab provides the number of instances for a particular class and graphs the count trend of those object in the JVM. It provides the shallow memory size (the memory footprint of the object and the primitives it contains) used by all the instances.

Analyzing Memory Thrash

Once a memory thrash problem is identified in a particular collection, start the diagnostic session by drilling down into the suspected problematic class.

Select the class name to monitor and click Drill Down at the top of the Object Instance Tracking dashboard or right-click the class name and select the Drill Down option.

For optimal performance, trigger a drill-down action on a single instance or class name at a time.

After the drill down action is triggered, data collection for object instances is performed every minute. This data collection is considered to be a diagnostic session and the Object Instance Tracking dashboard for that class is updated with this icon , to indicate that a diagnostic session is in progress.

The Object Instance Tracking dashboard indicates possible cases of memory thrash. Prime indicators of memory thrash problems indicated on the Object Instance Tracking dashboard are:

- **Current Instance Count**: A high number indicates the possible allocation of a large number of temporary objects.
- **Shallow Size**: Is the approximate memory used by all instances in a class. A large number for shallow size signals potential memory thrash.
- **Instance Count Trend**: A saw wave is an instant indication of memory thrash.

If you suspect you have a memory thrash problem at this point, then you should verify that this is the case. See To verify memory thrash.
Verify Memory Thrash

Select the class name to monitor and click **Drill Down** at the top of the Object Instance Tracking dashboard. On the Object Instance Tracking window, click **Show Major Garbage Collections**.

The following Object Instance Tracking Overview provides further evidence of a memory thrash problem.

If the instance count doesn’t vary with the garbage collection cycle, it is an indication of a potential leak and not a memory thrash problem. See [Java Memory Leaks](#).

Troubleshooting Java Memory Thrash Using Allocation Tracking

Allocation Tracking tracks all the code paths and those business transactions that are allocating instances of a particular class. It detects those code path/business transactions that are creating and throwing away instances.

To use allocation tracking

1. Using the Drill Down option, trigger a diagnostic session.
2. Click the Allocation Tracking tab.
3. **Click Start Allocation Tracking Session** to start tracking code paths and business transactions.
4. Enter the session duration and allow at least 1 to 2 minutes for data generation.
5. **Click Refresh** to retrieve the session data.
6. Click on a session to view its details.
7. Use the Information presented in the Code Paths and Business Transaction panels to identify the origin of the memory thrash problem.

Monitor Java Object Instances
If the application uses a JRE (rather than a JDK), the following steps enabled object instance tracking:

1. Ensure the tools.jar file is in the jre/lib/ext directory.
2. On the Node Dashboard, click the Memory tab.
3. On the Memory tab, click the **Object Instance Tracking** subtab.
4. Click **On** and then **OK**.

See **Object Instance Tracking for Java**.
Code Deadlocks for Java

By default, the agent detects code deadlocks. You can find deadlocks and see their details using the Events list or the REST API.

Code Deadlocks and their Causes

In multi-threaded development environments, it is common to use more than a single lock. However, sometimes deadlocks will occur. Here are some possible causes:

- The order of the locks is not optimal
- The context in which they are being called (for example, from within a callback) is not correct
- Two threads may wait for each other to signal an event

Finding Deadlocks using the Events List

Select Code Problems (or just Code Deadlock) in the Filter By Event Type list to see code deadlocks in the Events list. The following list shows two deadlocks in the ECommerce tier.

To examine a code deadlock, double-click the deadlock event in the events list and then click the Code Deadlock Summary tab appears. Details about the deadlock are in the Details tab.

Finding Deadlocks Using the REST API

You can detect a DEADLOCK event-type using the AppDynamics REST API. For details see the example Retrieve event data.
Thread Contention

Performance Issues Resulting from Thread Contention

Thread Contention Detection

Exporting Contention Information

Related pages:

Works with:

Java

Thread contention arises when two or more threads attempt to access the same resource at the same time. This topic describes how AppDynamics helps you diagnose and resolve thread contention issues.

Performance Issues Resulting from Thread Contention

Multithreaded programming techniques are common in applications that require asynchronous processing. Although each thread has its own call stack in such applications, threads may need to access shared resources, such as a lock, cache, or counter.

While synchronization techniques can help to prevent interference between threads in such scenarios, they may nevertheless compete for access to shared resources. This can result in application performance degradation or even data integrity issues.

AppDynamics can help you identify and resolve problems relating to thread contention in business transactions and service endpoints.

Thread Contention Detection

AppDynamics detects thread contention based on the thread state of the instrumented application.

It identifies these block or waiting states in the JVM:

- Acquiring a lock (MONITOR_WAIT)
- Waiting for a condition (CONDOR_WAIT)
- Sleeping (OBJECT_WAIT)
- A blocking I/O operation

The OBJECT_WAIT state is triggered when the application makes one of the following calls:

- Thread.sleep
- Object.wait
- Thread.join
- LockSupport.parkNanos
- LockSupport.parkUntil
- LockSupport.park

The Controller alerts you to possible thread contention problems in the Potential Issues pane of the Business Transaction Flow Map. From there, you can use the browser to access additional information about blocked and waiting threads in business transactions or service endpoints, and determine the cause of the performance problem.

The following sections explain how you use the browser to surface contention information for business transaction and service endpoints.
Thread Contention in Transaction Snapshots

To view information about thread contention:

1. In the transaction snapshot navigation page, look for items labeled as Thread Contention issues in the Potential Issues pane. The time column indicates blocked or wait time:

   ![Potential Issues](image)

2. To display more information about the blocked method, click the thread contention item and select Drill Down into Call Graph:

   ![Call Graph](image)

   The call graph shows the following information relevant to thread contention:
   - In the Call Graph header, **Wait Time** and **Block Time** indicate aggregate measures for the thread in one segment of the business transaction.
   - In the Call Graph header, **Node** specifies the name of the node hosting the contending threads, **PojoNode** in the example above.
   - The **Time** column indicates the total self time for the method.
   - The **Percent** column shows the amount of time spent in the method as a percentage of overall time for the thread.
   - The **Thread State** column indicates the degree of thread contention issues for the method. Gray means no problems; yellow to red shading signals the severity of contention problems. (When you hover over the bar, a breakdown of the elements that make up the thread state is shown: This includes Block time and Wait time by default. To include Cpu Time in the Thread State detail, Dev mode must be enabled.)

3. Right-click on any method with a thread state that indicates block or wait times and select View Details. The Thread Contention details pane appears:
The Thread Contention details pane displays the name of the blocked method in the top left corner and adds the following information in the Thread Contention table:

<table>
<thead>
<tr>
<th>Element</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocking Thread</td>
<td>The thread holding a lock on the blocking object.</td>
</tr>
<tr>
<td>Blocking Object</td>
<td>The object that the blocked thread is waiting to access.</td>
</tr>
<tr>
<td>Block Time</td>
<td>The amount of time waiting to access the object.</td>
</tr>
<tr>
<td>Line Number</td>
<td>The line number in the blocked method where the blocking object is being accessed.</td>
</tr>
</tbody>
</table>

With respect to the example above, run is attempting to access a locked object at line 114.

The order in which blocking threads are shown in the table is not significant; it does not imply a call order or time sequence.

In development mode, AppDynamics reports explicit locks such as `java.util.concurrent.locks.ReentrantLock` as Wait Time instead of Block Time in the Thread Details Call Graph view. Take this into consideration when monitoring business transactions and analyzing performance related to lock contentions.

### Thread Contention in Service Endpoints

You can view thread contention information for service endpoint methods in AppDynamics. Call graphs identify service endpoint methods with the following icon: ![Service Icon](image)

Select More > Service Endpoints from the menu bar to view thread contention information by service endpoint.

### Exporting Contention Information

When you export the Call Graph for a Business Transaction, AppDynamics includes Transaction Contention information.

- The Summary pane includes Block Time data: the block time specified is the sum of all block times for the blocked methods shown in the CallGraph pane.
The Call Graph pane lists block time by method:

Call Graph

```
java.lang.Thread.run: 74% (execution time: 0 ms of 7026 ms total, block time: 0 ms of 3389 ms total)
Contention$3run: Unknown (execution time: 3445 ms of 7026 ms total, block time: 3384 ms of 3389 ms total)
java.util.concurrent.locks.ReentrantLock$ReentrantLockSupport$park: 28% (execution time: 0 ms of 5 ms total, block time: 0 ms of 5 ms total)
java.util.concurrent.locks.LockSupport$park: 17% (execution time: 5 ms of 5 ms total, block time: 5 ms of 5 ms total)
CPUBurnerUtil$findPrimes: Unknown (execution time: 2269 ms of 2269 ms total)
CPUBurnerUtil$findPrimes: Unknown (execution time: 1307 ms of 1307 ms total)
```
Event Loop Blocking in Node.js

On this page:
- Latency in Node.js Event Loops
- Process Snapshots In AppDynamics

Related pages:
- Manage Node.js Process Snapshots
- Process Snapshots and Business Transaction Snapshots

You can use process snapshots to examine Node.js event loop activity and identify functions with high CPU times that are blocking the event loop.

Latency in Node.js Event Loops

The event loop of a Node.js process is a single thread that polls for incoming connections and executes all application code. When a Node.js request makes a call to an external database, remote service or the filesystem, the event loop automatically directs the application's control flow to some other task, including other connections or callbacks.

CPU-intensive operations block the event loop, preventing it from handling incoming requests or finishing existing requests. This means that a CPU-intensive operation in one business transaction can cause slowness in other business transactions.

Process Snapshots In AppDynamics

A process snapshot describes an instance of a CPU process on an instrumented Node.js node. It generates a process-wide flame graph for a Node.js process over a configurable time range.

Process snapshots provide visibility into the Node.js event loop across all business transactions for the duration of the process snapshot. Process snapshots are useful when the main troubleshooting tools (e.g., business transaction snapshots) are inconclusive because the source of latency is a CPU-intensive operation in another business transaction. You can use lists of process snapshots to identify which functions have high CPU times. From the list, you can select and examine process snapshots to identify exactly which functions in your code are blocking the CPU.

For a given Node.js node or tier, you can access the list of process snapshots from the Process Snapshots tab of the node or tier dashboard. You can filter the process snapshot list to display only the snapshots that you are interested in. You can filter by execution time, whether the snapshot is archived, and the GUID of the request. If you access the list from the tier dashboard, you can also filter by node.

For more information on how process snapshots are generated and how to configure them, see Manage Node.js Process Snapshots.

To learn how process snapshots and business transaction snapshots are created, see Process Snapshots and Business Transaction Snapshots.

Process snapshots persist for 14 days, unless you archive them, in which case they are available forever.

A process snapshot contains the following tabs:
- Overview
- Flame graph
- Call Graph
- Allocation Call Graphs
- Hot Spots

Overview

Summarizes the snapshot. Contents vary depending on the available information.

Usually contains at least the total execution time, tier and node of the process, timestamp, slowest method and request GUID.
Flame Graph

Provides a visualization of each stack frame's frequency on the CPU over the duration of a process snapshot. The frame's position relative to the bottom-most stack depicts the call-stack depth. The flame graph contains the same information as the call graph, but allows you to quickly spot methods that are consuming more CPU resources relative to others.

The method corresponding to the stack frame on the top edge of the flame graph represents the method's CPU resource consumption frequency. To identify long-running CPU executions, look for long horizontal cells on the top edge of the flame graph. A healthy Node.js process has minimal CPU-blocking activity; correspondingly, a flame graph for a healthy Node.js process has minimal long, horizontal cells along the top edge of its flame graph. For more information on how to interpret flame graphs, see The Flame Graph.

Call Graph

Shows the total execution time and the percentage of the total execution time of each method on the process's call stack. The numbers at the ends of the methods are the line numbers in the source code. You can filter out methods below a certain time to simplify the graph and isolate the trouble spots.

The Time and Percentage columns identify which calls take the longest time to execute.

To see more information about a call, select the call and click Details.

Allocation Call Graph

Available only for process snapshots that are collected manually. See Collect Process Snapshots Manually in Manage Node.js Process Snapshots.

Shows the amount and percentage of the memory allocated and not freed by each method on the process’s call stack during the process snapshot. You can use the Method Size slider to configure how much memory a method must allocate to be displayed in the allocation call graph. You can also filter out methods that consume less than a certain amount of memory to simplify the graph and isolate the trouble spots.

The Size and Percentage columns identify which calls consume the most memory.

The agent cannot report allocations made prior to the beginning of the allocation snapshot.

The allocation reported in the snapshot is the memory that is still referenced when the snapshot ends: memory allocated during the snapshot period minus memory freed during the snapshot period.

To see more information about a call, select the call and click Details.

Hot Spots

This tab displays the calls by execution time, with the most expensive calls at the top. To see the invocation trace of a single call in the lower panel, select the call in the upper panel.

Use the Method Time slider in the upper right corner to configure how slow a call must be to be considered a hot spot.
Manage Node.js Process Snapshots

- **Automatic Process Snapshot Generation**
- **Collect Process Snapshots Manually**

**Related pages:**
- Event Loop Blocking in Node.js
- Node.js Metrics
- Process Snapshots and Business Transaction Snapshots

This topic explains how process snapshots are generated and viewed.

**Automatic Process Snapshot Generation**

When a business transaction snapshot is triggered by periodic collection or by a diagnostic session, a ten-second process snapshot is automatically started. By default, the agent starts no more than two process snapshots per minute automatically, but this behavior is configurable.

You can also start process snapshots manually on demand. See [Collect Process Snapshots Manually](#) below.

**Configure Automatic Collection**

You can configure automatic process snapshot collection using the following settings:

- `processSnapshotCountResetPeriodSeconds`: Frequency, in seconds, at which the automatic process snapshot count is reset to 0; default is 60 seconds
- `maxProcessSnapshotsPerPeriod`: Number of automatic process snapshots allowed in `processSnapshotCountResetPeriodSeconds` seconds; default is 2 snapshots
- `autoSnapshotDurationSeconds`: Duration of an automatically-generated process snapshot; default is 10 seconds

To configure these settings, add them to the require statement in your application source code as described in *Install the Node.js Agent*. Then stop and restart the application.

**Collect Process Snapshots Manually**

If you want to generate some process snapshots now, you can start them manually.

1. Navigate to the dashboard for the tier or node for which you want to collect process snapshots.
2. Click the Process Snapshots tab.
3. Click Collect Process Snapshots.
4. If you are in the Tier dashboard, select the node for which you want to collect snapshots from the Node dropdown list. If you are in the node dashboard, you can only set up snapshot collection for that node.
5. Enter how many seconds you want to collect process snapshots for this node. The maximum is 60 seconds.
6. Click Create.

The agent collects process snapshots for the configured duration. Process snapshots that are started manually include an allocation call graph that shows how much memory has been allocated and not freed during the period recorded by the snapshot.
Process Snapshots and Business Transaction Snapshots

On this page:
- V8 Sampler
- Call Graph Data in Snapshots
- Business Transaction Snapshots Trigger Process Snapshots
- Concurrent Business Transaction and Process Snapshots

Related pages:
- Transaction Snapshots
- Manage Node.js Process Snapshots

This topic explains in detail the relationship between transaction snapshots and process snapshots created by the Node.js Agent.

V8 Sampler

Node.js is built on the V8 JavaScript engine, which includes a code sampler.

The Node.js Agent uses the V8 sampler to create process-wide process snapshots, which contain call graphs of the methods on the Node.js process's call stack.

Call Graph Data in Snapshots

Call graph data is displayed in business transaction snapshots as well as process snapshots.

When you view a business transaction snapshot, the displayed call graph specific to the transaction instance is derived from the concurrent process snapshot call graph.

When you view a process snapshot, the complete call graph of all the business transactions executed while the process snapshot was captured is displayed.

The call graph in a business transaction snapshot displays a view of the data from a concurrent process snapshot that is filtered to display only time in methods attributable to the specific business transaction. It is a subset of the concurrent process snapshot call graph.

For this reason, you might see an execution time for a method in a business transaction call graph that is less than the execution time for the same method in the concurrent process snapshot call graph. This would indicate that some calls to that method were made outside the context of the business transaction instance captured by the transaction snapshot.

The summary tab of a transaction snapshot includes a link to the process snapshot that was taken during the time covered by the transaction snapshot.

Business Transaction Snapshots Trigger Process Snapshots

To provide call graph data associated with business transaction snapshots, the agent starts a ten-second process snapshot whenever it starts a business transaction snapshot that is triggered by periodic collection or a diagnostic session if there is no existing process snapshot in progress for the current process. Process snapshots do not overlap. Periodic collection means that a business transaction is collected at periodic intervals, by default every ten minutes, but configurable. Diagnostic session means that either the agent has detected a pattern of possible performance issues and automatically started capturing transaction snapshots or a human has manually started a diagnostic session for the same reason.

Concurrent Business Transaction and Process Snapshots

The result is that you will usually see a process snapshot that ran concurrently with a business transaction. How well the two snapshots line up depends on the relative durations and start times of the transaction and the process snapshots.

In the scenario sketched below, all of the five-second blue transaction's calls, and most of the 10-second green transaction's calls are...
captured by a 10-second process snapshot, but only about half of the 14-second orange transaction snapshot’s calls.

If you find that your business transactions are running longer than your process snapshots, you can increase the default length of a process snapshot in the `autoSnapshotDurationSeconds` setting in the `require` statement.
Information Points

On this page:
- Permissions
- Information Point Data: Code Metrics and Business Metrics
- Creating Information Points
- Viewing Information Point Data

Related pages:
- PHP Information Points
- Java and .NET Information Points

Works with:

You can use information points to define custom metrics based on methods you configure.

Information points are similar to data collectors. However, while data collectors show application data in the context of a business transaction, information points reflect data state across all invocations of a method, independently of business transactions. They also let you apply computations to the values, for example, representing the sum or average for a method return value or input parameter.

This topic describes how to configure and use information points. For information on data collectors, see Data Collectors.

Permissions

To create, edit, or delete information points you need the Configure Information Points permission.

Information Point Data: Code Metrics and Business Metrics

When you configure an information point, you automatically get the KPI metrics (called code metrics) for the information point method. The code metrics are:
- Total call count
- Calls per minute count
- Errors per minute
- Average response time

You can supplement the KPI metrics with custom business metrics for the information point.

Business metrics reflect the value of runtime data, such as the method parameter, return value, or a value captured by getter chain on the object on which the identified method was invoked. The business metric value represents either the sum or average of the values of the code point you identify as the information point.

Information points can give you significant insight into how the performance of an application corresponds to business performance. For example, depending on the nature of your application, you could use it to resolve business questions such as:
- What is the average value of the credit card total?
- How many credit cards did my application process in a certain time period, regardless of the business transaction?
- What was the average time spent processing a credit card transaction?

A example of a practical use of an information point is ignored exceptions. Exceptions, especially one that occurs frequently, can contribute to CPU spikes in a JVM. If you configure the exception to be ignored in AppDynamics, for example, if it is generated in the underlying application framework and does not have a direct bearing on your application performance, it may not be readily evident to you when the exception is affecting your application. An information point that counts the exception occurrence can help you identify the additional overhead.
Creating Information Points

For information on creating information points, see:

- Java and .NET Information Points
- PHP Information Points

Viewing Information Point Data

You can view information point values in various places in the Controller UI, including the following:

- Information Points page: The primary page for viewing and administering information points is the **Information Points** page. From there, click on an information point to view a histogram of the metrics for the information point for the selected time period.
- Metric Browser: The metrics also appear in the Metric Browser under the Information Points folder.
- REST API: Business metrics can be accessed from the **AppDynamics REST API**.
- Business Transaction Snapshots: Information point data does not appear in transaction snapshots by default, but you can configure it to appear by setting the `enable-info-point-data-in-snapshots` node property to true. When the `enable-info-point-data-in-snapshots` node property is set, information point calls appear in the User Data section of the snapshot. For more information, see **App Agent Node Properties**.
Java and .NET Information Points

On this page:
- Before Starting
- Create an Information Point from a Transaction Snapshot
- Manually Create and Manage Information Points

This topic describes how to create information points for Java and .NET applications. For PHP-based information points, see PHP Information Points.

Before Starting

To apply the configuration for a new information point for .NET and JVM 1.5 applications, you will need to restart the application server. Before starting, make sure that you have the ability to restart the application while minimizing user impact.

In the Controller UI, you need to be logged in as a user with Configure Information Points permissions. See Roles and Permissions for more information.

Create an Information Point from a Transaction Snapshot

An easy way to create an information point is from the call graph of a transaction snapshot that includes the method.

When you find an invocation of the method in the call graph, right click it and select the option for configuring instrumentation on this class/method.

An information point is created from your selection.

Manually Create and Manage Information Points

You can create, view, and modify information points in the More > Information Points page.

When creating an information point, you need to specify the class and method for which you want to collect information, along with other identifying information. For details on using the UI to configure code matching for an information point, see the equivalent
information on configuring data collectors on Data Collectors. Information points are evaluated following execution of the method on which they are placed. Any match conditions therefore apply to the state of the data at that time.

To collect business metrics from the method, add a custom metric for the information point. The custom metric configuration settings let you specify the data to collect as the business metric.
Information points reflect key performance metrics and custom metrics for methods and code data points that you configure. You can define information points for PHP applications as described in this topic.

About PHP Information Points

To create an information point for PHP, you first define the information point using JSON notation.

In the definition, you specify the method for the information point and optionally a point in the code that you want to capture as a custom metric, such as a parameter value or return value.

After creating the definition, add an information point in the More > Information Points page in the Controller UI. When adding the information point, choose PHP as the agent type and then paste the definition into the information point text box, as shown in the following screenshot:
Essentially, the elements of the definition correspond to configuration options available in the Add Information Point dialog box for Java or .NET applications.

However, for PHP, the information point is defined in JSON syntax, and there are a few limits on configuration options. For one, class matching is limited to classname-based matches only. (Superclass matching or annotation matching, for example, are not available.) Also, you can only define a single match condition on a matched method. (A match condition refines the match of a method by testing a parameter or return value.)

If you do not configure a custom metric, the information point captures the generic KPIs for a matched method (response time, calls and calls per minute, errors an errors per minute). Custom metrics extend the information point by capturing the return or parameter value of a method, or the value of an invoked object.

**Before Starting**

To create information points in the Controller UI, you need to be logged in as a user with Configure Information Points permissions. See Roles and Permissions for more information.

**Method Match Conditions**

A match definition specifies the method associated with the value you want to capture as an information point. The match definition must specify the method, but it may also specify the containing class and a refining match condition. The match condition can test the value of a method parameter, return value, or the return value of an invoked object.

Use the following template to create a match definition:
Edit the JSON objects as follows:

- **classMatch** is optional. If you do not want to specify a class to match, delete the `classMatch` object from the `phpDefinition` object.
- If you are using the class match, set the value for the `classNameCondition` match string to the class name.
  - The class must be defined in a separate file from the file in which it is instantiated and invoked.
  - If your class names are defined in a namespace, escape the backslashes in the namespace with an escape backslash.
- **methodMatch** is required. Set the value for the `methodNameCondition` match string to the method name.

The following values are required for PHP information points:

- classMatch **type**: MATCHES_CLASS
- classNameCondition **type**: EQUALS
- methodNameCondition **type**: EQUALS

For example, the following JSON creates an information point on a class for which the class name equals `CheckoutManager` and the method name `processPayment`:
If creating an information point primarily to capture KPI metrics for the method, it is likely you will define the class for the method. However, if you are creating an information point to implement a code metric, you may only need to specify method name matching.

The following example shows a method match that tracks how many times a method is called:

```json
{
  "probe": {
    "phpDefinition": {
      "methodMatch": {
        "methodNameCondition": {
          "type": "EQUALS",
          "matchStrings": ["deleteCartItems"]
        }
      }
    }
  }
}
```

**Match Conditions**

A match condition (defined by a `matchConditions` object) lets you refine the method matching criteria by specifying a match condition based on a parameter or return value or the value returned by an invoked object.
A match condition is optional, but there can be at most one for the information point. If you do not supply a match condition, the information point includes all invocations of the matched method.

The condition consists of a match type and the comparison operator that defines the data to be compared. The match types are:

- **EQUALS**
- **NOT_EQUALS**
- **LT**
- **GT**
- **LE**
- **GE**
- **NOT**
- **AND**
- **OR**

The comparison operator is an expression node that defines a left side \((lhs)\) and a right side \((rhs)\) of the operation. It contains:

- a type: **ENTITY**, **STRING**, or **INTEGER**
- a value, which is an \(\text{entityValue}\), \(\text{stringValue}\), or \(\text{integerValue}\)

If the type is **ENTITY**, the \(\text{entityValue}\) has one of the following types: **INVOKED_OBJECT**, **RETURN_VALUE**, or **PARAMETER**. If the type of the \(\text{entityValue}\) is **PARAMETER**, the \(\text{parameterIndex}\) indicates the 0-based numeric index of the parameter on which to match. So if the method takes two parameters, for example, the first being a string and second an integer, a \(\text{parameterIndex}\) of 0 matches against the string and a \(\text{parameterIndex}\) of 1 matches the integer value.

The information point in the following example only matches invocations of the **processPayment** method in which the second parameter (at parameter index 1) equals **VISA**.
Metric Definitions

You can define custom metrics for data captured by an information point using the `metricDefinitions` object in your information point definition. Custom metrics appear in the Metric Browser and in the dashboard for the information point. You can create health rules based on a custom metric and retrieve custom metric values using the AppDynamics REST API.

There can be multiple custom metrics for a single information point. For example, one metric might be based on the average and one on the accumulated (sum) of the information point’s values.

A `metricDefinitions` object consists of one or more definitions, each having the following structure:

- a name
- a rollup type (AVERAGE or SUM)
- data, which consists of a type, (ENTITY, STRING, or INTEGER) and a value (entityValue, stringValue, or integerValue)

If the type is ENTITY, the entityValue has a type, which is INVOKED_OBJECT, RETURN_VALUE, or PARAMETER. If the type of the entityValue is PARAMETER, the zero-based parameterIndex indicates the parameter on which to base the match. A return value or parameter metric cannot be an array.

For example, the following `metricDefinitions` object defines two custom metrics: VisaTotal, which reports the sum of the Visa payments processed and VisaAverage, which reports the average value of the Visa payments processed.
Sample JSON Information Point Configuration

This is an example that you can copy and paste into the JSON text field in the PHP information point window. It produces two metrics, named VisaTotal and VisaAverage.

```
"metricDefinitions": [
    {
      "name": "VisaTotal",
      "rollup": "SUM",
      "data": {
        "type": "ENTITY",
        "entityValue": {
          "type": "RETURN_VALUE"
        }
      }
    },
    {
      "name": "VisaAverage",
      "rollup": "AVERAGE",
      "data": {
        "type": "ENTITY",
        "entityValue": {
          "type": "RETURN_VALUE"
        }
      }
    }
]
```
"matchCondition": {
  "type": "EQUALS",
  "comparisonOp": {
    "lhs": {
      "type": "ENTITY",
      "entityValue": {
        "parameterIndex": 1,
        "type": "PARAMETER"
      },
      "rhs": {
        "type": "STRING",
        "stringValue": "VISA"
      }
    }
  }
},
"metricDefinitions": [
  {
    "name": "VisaTotal",
    "rollup": "SUM",
    "data": {
      "entityValue": {
        "type": "RETURN_VALUE"
      },
      "type": "ENTITY"
    }
  },
  {
    "name": "VisaAverage",
    "rollup": "AVERAGE",
    "data": {
      "type": "ENTITY",
      "entityValue": {
        "type": "RETURN_VALUE"
      }
    }
  }
]
Configure Instrumentation

On this page:
- Access Instrumentation Settings
- Other Configuration Topics

In AppDynamics, instrumentation refers to how app agents interact with your application software to gather performance data and report it back to the AppDynamics Controller. The first step to instrumenting your application is installing the app agents on the servers where the code runs.

App agents ship with default instrumentation settings that cover the most common types of application frameworks and programming patterns. If your requirements are more complex, you can customize the instrumentation behavior to suit your environment.

See Configure Instrumentation Overview for information on how AppDynamics applies instrumentation settings across your business application.

Access Instrumentation Settings

Navigate to Configuration > Instrumentation to customize the following:

- Transaction detection rules that define entry points for business transactions.
- Backend detection rules that define exit points for your business transactions.
- Error detection settings that define which transactions qualify as error transactions.
- Service endpoints to view metrics specific to services and transaction segments outside the context of a single business transaction.
- Diagnostic data collectors for an app agent to extract additional context about a specific transaction at transaction snapshot time. You can also configure data collectors for Application Analytics.
- Call Graph Settings to tune the amount of data the app agent collects for call graphs.
- JMX metrics that the Java agent collects for specific products and frameworks. See "Configure JMX Metric Rules" on Configure JMX Metrics from MBeans.
- Memory monitoring settings for agents that collect instance tracking data for specific classes or objects. See the following topics:
  - Object Instance Tracking for Java
  - Object Instance Tracking for .NET
  - Object Instance Tracking for Node.js
- Asynchronous transaction demarcators to set endpoints for asynchronous transactions.

Other Configuration Topics

In addition to the instrumentation configuration options covered in this topic, there are many other ways to customize AppDynamics for your environment. For more information on the items on the Configuration page, refer to the following topics:

- "Configure Thresholds" on Transaction Thresholds.
- "View Baselines" on Dynamic Baselines.
- Information Points
- Development Level Monitoring

If you have an End-User monitoring license, see the following topics for more information on User Experience App Integration:

- Correlate Business Transactions for Browser RUM
- Automatic Injection of the JavaScript Agent
- Assisted Injection
Configure Instrumentation Overview

On this page:
- Instrumentation Configuration
- Live Preview Mode
- Programmatic Expressions

Instrumentation Configuration

The scope configuration model reduces the number of steps it takes to apply transaction detection rules to multiple tiers.

Under the scope configuration model, you can apply settings to groups of tiers, in addition to individual tiers and nodes. Groups of tiers are called scopes. Only transaction detection settings for applications created in AppDynamics 4.3 or later adhere to the scope model.

Live Preview Mode

You can interactively experiment with transaction detection rule configurations using Live Preview. Live Preview streams data from an active node, providing real-time data for the following configurations:

- All transaction detection rules that apply to a node. See Business Transaction Discovery Sessions.
- A single custom match rule applied to a node. See Custom Match Rule Live Preview.

The following limits apply to Live Preview:

- 100 sessions on the Controller
- 20 sessions on an account
- 10 sessions on a business application

Programmatic Expressions

You can specify the following types of programmatic expressions when configuring detection rules:

- Regular expression: A string or character search pattern
- Getter chains: Enable you to access data returned by methods in your application
Scope Configuration Model

The scopes configuration model allows you to bundle multiple tiers into a scope, and apply transaction detection rules to that scope. You can apply a rule to select tiers all at once. The Default Scope consists of all the tiers in the application.

Creating a Custom Scope

You can create a custom scope by navigating to Configuration > Instrumentation > Scopes.

1. Click Add.
2. Enter a name for your scope.
3. In the Include the following Tiers dropdown menu, select one of the following options:
   a. All Tiers in the Application: creates a scope that includes all the tiers in the application, except tiers that you specify. You can select tiers to exclude from the scope in the Available Tiers box and clicking the left-pointing arrow.
   b. These Specific Tiers: creates a scope that includes only tiers that you specify. You can select tiers to include in the scope in the Available Tiers box and clicking the left-pointing arrow.
4. Click Save.

To view the rules that are applied to a scope, select the scope in the Scopes tab. The rules applied to that scope are displayed in the panel on the right.

Set the Scope for a Rule

When you create a new application, all rules for the application use the default scope. You can create additional scopes for that application, and choose which rules to apply to that scope. Each rule can only apply to one scope.
1. Click on the **Rules** sub-tab.
2. Select the rule that you want to apply to a scope. The tiers that the rule is applied to are displayed in the panel on the right.
3. Click **Edit**.
4. Click **Change Scope**.
5. Select the scope that you want to apply this rule to.
6. Click **OK**.
7. Click **Save**.

To apply a rule to a scope, you need permissions to all the tiers in the scope. To apply a rule to the default scope, you need application-level permission.
Using Regular Expressions

On this page:
- Matching Guidelines
- Regular Expression Engines
- Regular Expression Examples
- Performance Considerations

Related pages:
- Transaction Detection Rules
- Custom Match Rules

Regular expressions appear in various places in the AppDynamics configuration. These places include, for example, transaction detection rules, data collectors, EUM injection settings, health rules, and more. This topic describes the use of regular expressions in AppDynamics.

Matching Guidelines

A match condition consists of a named property to match (such as a method name, Servlet name, URI, parameter, or hostname), a comparison operator, and a matching value. For complex match conditions, you can use a regular expression (often abbreviated to just regex).

Match rules are case sensitive. Also, matching is based on subsequence pattern matching. To match a complete string instead, you need to include "^" to match the start of the string and "$" to match the end of the string in your regular expression.

In the context of business transaction matching, the pattern does not include the protocol (e.g., "http://"), host, port, or query parameters in the URL. So for example, a URI of http://www.mysite.com:8000/account/settings.html?action=update would be matched by a business transaction regular expression with only "/account/settings.html".

Regular Expression Engines

The Java Agent uses Java libraries for regular expressions. See:
- Javadoc: http://download.oracle.com/javase/1.5.0/docs/api/java/util/regex/Pattern.html

The .NET Agent uses the built in .NET regular expressions engine. See:
- MSDN: .NET Framework Regular Expressions

The Node.js Agent uses JavaScript regular expressions. See:

The PHP Agent uses PHP's built-in PCRE regular expression engine and requires the same syntax, including delimiters (for example: /^Foo$). See:
- PCRE Manual

The Python Agent uses Python's regular expression syntax. See:
- https://docs.python.org/2/library/re.html

The Web Server Agent uses Perl regular expression syntax. See:

Regular Expression Examples

The following sections illustrate examples of how to construct regular expressions to achieve different results.

Matching non-adjacent URL segments

...
A typical use of regular expressions in the AppDynamics configuration is for business transaction custom match rules in which the expression is matched to a requested URI. In this context, it's common for an application's URI pattern to put information that would be useful for business transaction identification in different segments of the URI.

For example, given an example URL of http://retailstore.example.com/store/jump/category/shoes/departments/view-all/cat630006p, a business transaction might need to match on /store/jump and all to group user requests to view all of an available category.

A regular expression to match this case could be:

```
/store/jump.*\b?all\b
```

**Matching any digit**

Say you want to ignore numbers contained within a pattern. For example, consider the following URL examples:

- /group/1/session/
- /group/1/session/
- /group/31/session/
- /group/2/session/

Examples of matching regular expressions would be:

- `^/group/d*/session/?$`
- `session`
- `/d*/session/?$

**Requiring a digit**

To group URLs that contain letters then numbers into one business transaction, such as the following:

- /aaa123.aspx
- /b1.aspx

You could use an expression such as the following:

```
/[a-z]+?[0-9]+?
```

Not matched would be a URL that does not have digits after the letters, for example: /z.aspx

**Handling letter casing**

Regular expression matching is performed in a case-sensitive manner. To match any case letters, you can use something similar to the following:

```
/[a-zA-Z]+?[0-9]+?
```

Or match in a case-insensitive manner using the / modifier. For example:

```
(?i)\w*cart\w*
```

Would match addToCart as well as addTocart.

**Backend Discovery Rules**

For an example of a JDBC backend regular expression, see the section JDBC with complex URLs in Example JDBC Backend.
Performance Considerations

Although regular expressions are a powerful way to set AppDynamics configurations, you should consider the following to avoid performance issues:

- Do not use wildcard expressions unless absolutely needed. You do not need to use wildcard expressions (.* ) before or after a match, and each wildcard regular expression results in approximately an order of magnitude performance impact.
- The number of configurations, the frequency the configuration is applied, and the content length of the target string: For example, if the configuration using a regular expression is applied in many places at high frequency for longer target strings, you might consider reducing the usage of the configuration or decrease the frequency of its application.
Using Getter Chains

On this page:

- About Getter Chains
- Special Characters in Getter Chains
- Escape Literal Characters
- .NET Notes and Examples
- Java Notes and Examples

Related pages:

- Transaction Detection Rules

Getter chains let you access method data in various contexts of the AppDynamics configuration. For example, getter chains enable you to name business transactions based on return values.

This topic describes how to use getter chains.

About Getter Chains

You can use getter chains to:

- Create a new JMX Metric Rule and define metrics from MBean attributes. See Configure JMX Metrics from MBeans.
- Configure method invocation data collectors. See the Configuration Notes in Data Collectors.
- Define a new business transaction custom match rule that uses a POJO object instance as the mechanism to name the transaction. See POJO Entry Points.
- Configure a custom match rule for servlet entry points and name the transaction by defining methods in a getter chain. See “Split by POJO Method Call” on Split Servlet Transaction by Payload Examples.

As a best practice, you should use getter chains with simple getter methods only. Getter chains on processing-intensive methods, such as one that make numerous SQL calls, can result in degraded performance for the application and agent.

For example, the following shows a simple get method that returns a property for a class, such as MyUser.Name:

```csharp
public class MyUser
{
    private String Name {get; set;}
    private String Url;
    public String GetUrl() {
        return this.Url;
    }
}
```

Special Characters in Getter Chains

Use the following special characters as indicated:

- Parentheses () to enclose parameters
- Commas , to separate parameters
- Forward slashes / to separate type declarations from a value in a parameter
- Backslashes \ to escape characters.
  - To search for a backslash \, escape the backslash with a backslash:

```csharp
GetString().Find(\)
```
To split a string by a backslash \\, escape the backslash in Java and within the Getter Chain:

```
GetString.split(\\\).[0]
```

- Dots . to separate methods and properties in the getter chain
- Dot . to represent “anything” must be escaped
- Curly braces { } to delineate getter chains from static elements in custom expressions on HttpServletRequest objects (including in the Java Servlet Transaction Naming Configuration window and in the Split Transactions Using Request Data tab of the servlet custom match and exclude rules)

Getter chains also treat spaces at the beginning or end of a string as special characters.

**Escape Literal Characters**

If the getter chain should treat a special character literally, escape it using a backslash. For parentheses, you only need to escape the closing parenthesis in a string.

- The following example shows how to escape the dot in the string parameter.

```
GetAddress().GetParam(a\.b\.c\.)
```

The agent executes GetParam("a.b.c.") on the result of GetAddress() and returns the value of the parameter.

- In the following example, the first dot is part of the string method parameter, which requires an escape character. The second and third dots don't require an escape character because they are used as method separators.

```
GetUser(suze\.smith).GetGroup().GetId()
```

- The following example shows how to escape the opening and closing parenthesis in a search for "()" within a string.

```
GetString.Find(\()\)
```

**.NET Notes and Examples**

The following sections apply to getter chains used in .NET Agent configurations.

```
Escape backslashes within a getter chain for .NET with the double backslash.
```

**Declare Parameter Types**

The .NET Agent identifies parameter types as follows:

- Dictionaries, anything with property accessors, use a normal parameter set, which defaults to string.
- Arrays use single integers as parameters.

Therefore 0 means string/0 if you have a hash, or anything that translates to get_Item (any kind of property). 0 means int/0 if you have an array.

When your getter chain uses a method with parameters other than the default type, you need to declare the parameter type.

- The following example demonstrates how to declare the parameter types to resolve the overloaded Substring() method. The forward slash serves as the type separator.
For instance, if `GetAddress(appdynamics, sf) returns "303 2nd St, San Francisco, CA 94107"`, the full getter chain expression returns "303 2nd St".

- The following example shows how to declare the parameter types for a user-defined method that takes a float parameter, a boolean parameter, and an integer parameter. The forward slash serves as the type separator.

    ```csharp
    GetAddress(appdynamics, sf).MyMethod(float/0\.2, boolean/true, boolean/false, int/5)
    ```

**Access Indexed Properties and Dictionary Values**

If your getter chain accesses a dictionary object, you can access the values using the following syntax:

- The following example returns the value for the key `suze\smith`.

    ```csharp
    UserDictionary\[string/suze\\smith\
    ```

- The following example returns the value for the key `suze\smith` using the implied getter.

    ```csharp
    UserDictionary.get_Item(suze\\smith)
    ```

**Split by Character or Regular Expression Match in .NET Getter Chains**

You can split values matched by character or as a string by matching a regular expression pattern. The result of a split operation is a string or character array that you can reference in your getter chain by an array index value.

The following examples illustrate how to use the character and string regular expression split operations in getter chains.

**Split by Character**

- The following chain splits a URL on the forward slash character. In this case, the first slash acts as a type separator. The getter chain returns the fourth item in the array.

    ```csharp
    GetUrl().Split(char\[//\]).[3]
    ```

The agent returns “Search” when it applies the getter chain to the following URL: http://howdyworld.example.com/Search/Airfare

- In the following example, the split occurs on the forward slash character, and the result is the length of the resulting array.

    ```csharp
    GetUrl().Split(char\[//\]).Length
    ```

- This example illustrates a transaction splitting rule for URIs that use a semicolon delimiter. The getter chain splits on the forward slash, then splits the fourth element on the semicolon.
The agent returns `create-user` when it applies the getter chain to the following URL:

http://HowdyWorld.example.com/create-user;sessionid=BE7F31CC0235C796BF8C6DF3766A1D00?act=Add&uid=c42ab7ad-48a7-4353-bb11-0df3ab798b5

**Split by Regular Expression**

For more control, you can split values by string-based pattern matching. Pattern matching is particularly useful for situations that require complex matching, such as matching content within a request body.

The following example shows a getter chain that splits the value returned by `GetAddress()` and selects the seventh element in the resultant array:

```
GetAddress().Split(string/\W+).[6]
```

Given an address such as `303 2nd St, San Francisco, CA 94107`, the example splits the value by word and references the sixth word in the array, `CA` in this case.

**Compose Getter Chains for HTTP Requests**

The .NET Agent requires special syntax for getter chains for HTTP Requests:

- For ASP.NET WebForms, MVC, and MVC WebAPI applications create getter chains based upon the `System.Web.HttpRequest` objects.
- For ASP.NET Core on the full framework, create getter chains based upon `Microsoft.AspNetCore.Http.Internal.DefaultHttpContext` objects.

If you have both ASP.NET and ASP.NET Core on the full framework apps in the same tier, you cannot use a getter chain to apply to both because the two frameworks use different objects to handle HTTP requests.

- Use the following syntax to delineate the boundaries of the getter chain:
  ```
  ${myobject.myproperty}
  ```

- The following example determines the user principal:
  ```
  ${Context.User.Identity.Name}
  ```

Places to use this syntax include:
- HTTP Request Data Collectors
- ASP.NET Transaction Detection custom expressions

**Java Notes and Examples**

The following sections apply to getter chains used in Java Agent configurations.

**Values Passed in a Getter Chain**

The value passed in a getter chain is always a string unless cast to another type.

The following cast types are supported:
- int
- float
- bool (the primitive Boolean value)
- boolean (a boxed boolean value; i.e. an object type that wraps a boolean)
- long
- object

The following section shows examples of how to refer to the types in parameters to getter chain methods. Notice that letter case is not important for the names of the types. Type casting is performed in a case-insensitive manner.

**Java Getter Chain Examples**

- Getter chain with integer parameters in the substring method using the forward slash as the type separator:
  
  ```java
  getAddress(appdynamics, sf).substring(int/0, int/10)
  ```

- Getter chain with various non-string parameter types:
  
  ```java
  getAddress(appdynamics, sf).myMethod(float/0.2, boolean/true, boolean/false, int/5)
  ```

- Getter chain with forward slash escaped; escape character needed here for the string parameter:
  
  ```java
  getUrl().split(/) # node slash is escaped by a backward slash
  ```

- Getter chain with an array element:
  
  ```java
  getUrl().split(/).[4]
  ```

- Getter chains that return Hashmap values:
  
  ```java
  get(object/myvalue).substring(int/0,int/10)
  get(object/ACTION)
  ```

- Getter chain with multiple array elements separated by commas:
  
  ```java
  getUrl().split(/).[1,3]
  ```

- Getter chain retrieves property values, such as the length of an array:
  
  ```java
  getUrl().split(\.).length
  ```

- Getter chain using backslash to escape the dot in the string parameter; the call is getParam (a.b.c):
- In the following getter chain, the first dot requires an escape character because it is in a string method parameter (inside the parentheses). The second dot does not require an escape character because it is not in a method parameter (it is outside the parentheses).

```java
getName(suze\.smith).getClass().getSimpleName()
```

- The following getter chain is from a transaction splitting rule on URIs that use a semicolon as a delimiter.

```java
getRequestURI().split(\/).\[2\].split(;).\[0]`
Transaction Detection Rules

On this page:
- Permissions
- Live Preview
- Manage Transaction Detection Rules
- Transaction Detection Rule Priorities
- Export Detection Rules

Related pages:
- .NET Business Transaction Detection
- PHP Transaction Detection

AppDynamics uses different types of transaction detection rules to define entry points and name business transactions:

- Automatic transaction discovery rules include the default entry point types and naming configuration for each app agent type.
- Custom match rules offer finer control over transaction discovery and naming for a single entry point type.

In many cases, the built-in rules yield a useful set of business transactions. If not, you can create new rules or edit the existing rules to optimize the transaction detection for your environment.

Permissions

The Configure Transaction Detection permission is needed to customize transaction detection rules.

The View Sensitive Data permission, in combination with the Configure Transaction Detection permission, enables the use of Live Preview and Business Transaction Discovery features to stream live data from your application.

Live Preview

For Java and .NET environments, you can use interactive live preview tools to find business transaction entry points:

- Business transaction discovery sessions stream live data from the node to the Controller UI to help you make instrumentation decisions. These sessions display transactions based on all transaction detection rules active for the node.
- For certain types of entry points, custom match rule live preview enables you to preview transactions based on applying a single transaction detection rule on a node.

Business transactions within the context of the Live Preview, called transient transactions, do not persist after the Live Preview ends. Business transaction discovery sessions let you apply or discard rule changes when you end the session. Custom match live preview rules apply after you save the rule.

Some transient transactions may invoke exit calls to the same node where the Live Preview session is running. In certain cases, the agent may discover additional transient transactions and uninstrumented code on the downstream segment:

- The exit call and corresponding entry point are automatically discovered exit and entry points like HTTP or Web Service.
- Rare cases of custom activity in-process calls.

Manage Transaction Detection Rules

To customize transaction detection rules in the Controller UI, go to Configuration > Instrumentation. From the Business Transactions list, you can also click Configure to quickly access Transaction Detection settings.

You apply transaction detection rules to tiers using the flexible scope configuration model. From the Transaction Detection tab, you can manage rules as follows:

- Manage rules for individual scopes on the Transaction Detection tab.
- Manage rules application-wide on the Rules sub-tab. The Rules sub-tab also shows you the tiers where rules are applied and according to which scope.
- View or modify the rules for specific tiers on the Tiers sub-tab.
To modify an existing rule, you can double-click it to display the Rule Editor. Click +/-Add to define a new rule.

Use Live Preview button to start a Business Transaction Discovery session for nodes running the Java and .NET Agent.

Transaction Detection Rule Priorities

AppDynamics app agents apply transaction detection rules of the same type in the following order of precedence:

1. Rule priority from highest to lowest. A priority of 0 is the lowest priority possible.
2. Creation date from oldest to newest.

For example, consider an HTTP request that matches a priority 2 custom match rule and a priority 6 custom match rule. The agent applies the priority 6 rule.

When an incoming request matches more than one type of detection rule, AppDynamics applies the detection rules in the following order of precedence:

1. Custom match include rules according to the include rule priority. If the request matches an include rule, the agent names the business transaction based on the rule.
2. Custom match exclude rules according to the exclude rule priority. If the request matches an exclude rule, the app agent excludes the transaction from discovery.
3. Default automatic detection rules. If the request matches a default automatic detection rule, it names the transaction according to the auto-detection naming scheme.

To view the default order of transaction detection rules for a tier

1. Navigate to Configuration > Instrumentation > Transaction Detection > Tiers.
2. Click a tier to see the rules ordered by precedence and priority.

For Java and .NET, you can set the check-bt-excludes-early node property to “true” to configure the app agent to evaluate custom match exclude rules before custom match include rules. This is a node-based configuration and does not affect the display order of rules in the Controller UI.

Export Detection Rules

You can migrate the rules between applications and tiers using the Controller API. For more information, see Configuration Import and Export API.
Automatic Transaction Discovery Rules

Each AppDynamics app agent type has a default automatic transaction discovery rule that applies to all the entry points for that agent. When an agent identifies a business transaction that matches an automatic transaction discovery rule, it names the business transaction based on naming rules for the entry point type.

**Automatic Transaction Naming**

The naming logic varies depending on the application technology. For example:

- For web-oriented technologies that have URI entry points such as Java Servlets, detection rules name the transaction using the first two segments of the URI by default. See URI Based Entry Points.
- For message-oriented technologies such as an asynchronous message listener or message-driven bean (JMS), the agent names the business transaction for the destination name (the queue name) or the listener class name (if the destination name is not available.)
- For Web services, the agent names the business transaction for the Web service name plus the operation name.

**Customize Automatic Transaction Discovery Rules**

You can customize automatic transaction discovery rules as follows:

- Add an automatic transaction discovery rule for an agent type.
- Enable or disable different types of automatically discovered entry points.
- Customize transaction naming for some entry points, such as most URI-based entry points.

To access the Rule Editor, double-click an Automatic Transaction Discovery rule for an app agent. Alternatively, you can create a new automatic transaction discovery rule for the agent type by clicking Add. Clicking the Rule Configuration tab displays the entry points for the automatic discovery rule.

When you edit, delete, or change the scope for an automatic transaction discovery rule, the Controller UI performs validation to ensure each tier has the correct automatic discovery rule available for its agent type.

If you are using the Java Agent you can preview the effects of your customizations before you apply them in production. See "Edit and Preview Transaction Discovery Rules" on Business Transaction Discovery Sessions.

**Disable Transaction Monitoring for Entry Points**

Automatic transaction detection rules have a Transaction Monitoring Enabled checkbox for each entry point type to control automatic discovery. By default, all entry point types are enabled. When you disable discovery for an entry point type, the agent stops counting, measuring, and recording all activity for that entry point type for the scope where the rule is applied.

The Transaction Monitoring Enabled configuration affects custom match rules in the same scope for the entry point type. For example, if you disable POJO monitoring in the Java Default Automatic Transaction Discovery Rule for a scope, the Java Agent will not discover any POJO transactions for the scope.
Following are the cases when you might want to disable monitoring:

- You know that you don’t want to monitor any business transactions of a specific entry point type.
- You want to monitor transactions from a downstream call.

For example, consider an application where Servlets implement Spring Beans and you are interested in monitoring the transaction starting at the Spring Bean level. Disable Servlet monitoring and the auto-detection rule for Spring Beans will be the entry point for the transaction.

**Disable Automatic Transaction Detection for an Entry Point Type**

If you want to continue monitoring custom match rules, you can disable Discover transactions automatically for <entry point type>. When you disable automatic transaction discovery, the app agent stops reporting metrics for previously-detected transactions for the now disabled entry point type. The agent only detects transactions based on custom match rules. However, disabled transactions are not deleted or excluded. See Organize Business Transactions for information on deleting transactions.

- The agent does not discover any new transactions based on the Default Automatic Transaction Discovery Rule for the disabled type.
- The agent no longer auto-detects calls to methods and operations of the disabled entry point type.
- Custom match rules for the entry point type remain active and the agent reports metrics for transactions resulting from custom matches.
- Exit point detection remains enabled.

The Discover Transactions automatically configuration does not apply to POJO entry points or POCO entry points.

**Customize Transaction Naming for an Entry Point**

To view and edit the business transaction automatic discovery rules, go to Configuration > Instrumentation > Transaction Detection.

To customize transaction naming for an entry point type, double-click a Default Automatic Transaction Discovery rule for specific agent type. Click the Rule Configuration tab to see entry points for the rule. Entry point types that have customizable naming options have the Configure Naming expander, as shown below.
For more information specific to customizing naming for URI-based transactions, see **URI Based Entry Points**.

You can also customize transaction naming using a **Custom Match Rule**.
Transaction Detection for Apache Web Servers

On this page:

- Default Automatic Naming for Web Transactions
- Identify Transactions Using URI Segments
- Identify Transactions Using Headers, Cookies, and Other Parts of HTTP Requests
- Custom Match Rules and Exclude Rules

Related pages:

- Business Transactions

The AppDynamics Apache Agent monitors web server entry points and names business transactions originating on Apache web server tiers based on the request URI. You can modify the default business transaction naming scheme if needed.

**Default Automatic Naming for Web Transactions**

By default, the AppDynamics auto-detection naming scheme identifies all web server transactions using the full URI before the query string. For example, the following URI represents a funds transfer operation for an online bank:

http://bank.example.com/Account/Transferfunds/California

Based on the default scheme, the business transaction would be named /Account/Transferfunds/California.

You can customize the auto-detected naming scheme by configuring identification based on:

- URI segments
- Headers, cookies, and other parts of HTTP requests

To customize auto-detected naming:

1. On the Transaction Detection page, select the application or tier to configure and click the Web Server tab.
2. Verify that Transaction Monitoring is enabled and click Discover Transactions automatically for http web requests.
   If you disable Discover Transactions automatically for WEB requests, the agent doesn't discover WEB transactions even if you configure custom naming.
3. Click Configure Naming for the Web type in the Entry Points panel.

**Identify Transactions Using URI Segments**

AppDynamics offers several options to automatically name web transactions based on the URI. Consider the following URL representing a checkout operation in an online store:

http://onlinestore.example.com/Web/Store/Checkou

You can configure AppDynamics to identify a more meaningful name using one of the following options:

- Click **Use the first** or **Use the last** to use the selected number of contiguous segments at the beginning or end of the URI. For example, to identify the checkout transaction using the last two segments of the URI: /Store/Checkout.
- If you need more flexibility, such as using non-contiguous segments in the name, click **Name Transactions dynamically using part of the requests** and specify the segments with the **Use URI segments in Transaction names** option.
- To name the transaction for specific URI segments, click **Use URI segment(s) in Transaction names**. This enables you to skip URI segments or use non-contiguous segments in the naming scheme.

Enter the segment numbers separated by commas: 1, 3. For example, the following URL represents the checkout transaction requested by a customer with ID 1234:

http://onlinestore.example.com/Store/cust1234/Checkout

The checkout transaction is the same regardless of the customer, so it makes sense to omit the customer ID and name the transaction based on the first and third segments of the URI: /Store/Checkout.

**Identify Transactions Using Headers, Cookies, and Other Parts of HTTP Requests**

To identify business transactions using particular parts of the HTTP request, click **Name Transactions dynamically using part of the request** and configure the option that makes sense for your application.

Carefully consider your naming configuration choices. If you use a value such as the request originating address and you have many
clients accessing your application, it's likely that you would quickly reach the maximum number of registered business transactions. See "About the "All Other Traffic" Business Transaction" in Business Transactions for information about this event.

The following provides sample results based on the configuration options:

- **To name transactions based upon the parameter name**, click **Use a parameter value in Transaction names** and enter the **Parameter Name**.
  For example, when you name the following transaction using the parameter name "category": http://example.com/Store/Inventory?category=electronics, AppDynamics names the transaction to include the category parameter value: /Store/Inventory.electronics.

- **To use a header value in transaction names**, click **Use header value in transaction names** and enter a **Header Name**.
  For example, if you name the transaction using a header such as "Version", AppDynamics names transactions with the header value as follows: /Store/Inventory.v2.5.

- **To use a cookie value in transaction names**, click **Use a cookie value in Transaction names** and enter the **Cookie Name**.
  For example, for a website that tracks a user's loyalty status in a cookie, set the Cookie Name to "loyalty". AppDynamics names transactions for the loyalty cookie value: /Store/Inventory.Status=Gold.

- **To use a session attribute value in transaction names**, click **Use a session attribute in Transaction names** and enter the **Session Attribute Key**.
  For example, a website stores a customer's region in the session property. Set the Session Attribute name to "region". AppDynamics names transactions for the region session attribute value: /Store/Inventory.NorthAmerica.

- **To use the request method in Transaction names**, click **Use the request method (GET/POST/PUT) in Transaction names**. For example: /Store/Inventory.GET.

- **To use the request host in Transaction names**, click **Use the request host in Transaction names**. For example: /Store/Inventory.192.0.2.0

- **To use the request originating address in Transaction names**, click **Use the request originating address in Transaction names**.
  AppDynamics names transactions for the IP address of the request client. For example: /Store/Inventory.192.0.2.10.

- **To use a regular expression on the URI to name the transaction**, click **Apply a custom regular expression on the transaction name**. AppDynamics uses the following rules to name the transaction:
  - The Apache Agent uses **Perl style regular expressions**.
  - AppDynamics tests the regular expression against the segments specified in the configuration.
  - AppDynamics names the business transaction for the substring match.
  - If the regular expression pattern isn't found, the business transaction name follows the URI rules.
  - If you use groups in the regular expression, AppDynamics names the business transaction for the first matching group. If no matching groups are found in the pattern match, AppDynamics names the transaction for the fully matched substring.
  
  For example consider the following URL: http://mywebapp.example.com/abc/jsessionId=12345008;mykey=mytransaction;anotherkey=foo.
  Use the first two segments of the URI with the following regular expression to name the transaction for the value of my key: .*mykey.\(\w+\).*
  In this instance AppDynamics names the transaction mytransaction.

**Custom Match Rules and Exclude Rules**

You can create custom match rules and exclude rules for Web type entry points for Apache Agent. The configuration parameters work the same as those for custom naming in this topic.

If an excluded request on the Web Server tier passes to another instrumented tier such as Java, PHP, or Python, the downstream agent will detect the transaction.

For more information, see Custom Match Rules.
Validation for Automatic Transaction Discovery Rules

When you first create a business application in AppDynamics, the Default Scope ensures that all tiers for the application have a default automatic discovery rule for every agent type. Because the scope configuration model is highly configurable, the Controller UI enforces validations in the following cases:

- When you change the automatic discovery rules for a scope.
- When you assign or reassign tiers to a scope.

When you attempt to make an invalid change, the Controller prevents the change and displays a message in the Controller UI.

These validations ensure that after configuration updates every tier in your business application has the default automatic discovery rule for its agent type. They also ensure that there is an automatic discovery rule for all agent types to handle any prospective tiers you add to your application.

**Existing Tiers**

Every existing tier must belong to a scope that includes an automatic transaction detection rule for the tier's agent type. For example, a Java tier must belong to a scope with a Java automatic transaction detection rule.

The following rules apply:

- You can remove a tier from scope only when the tier has an auto-discovery rule from another scope.
- You can remove an automatic transaction detection rule from a scope only when tiers have auto-discovery rule from another scope.

**Prospective Tiers**

To ensure that any new tier you add to your application has an automatic transaction detection rule available, there must be at least one automatic transaction detection rule for each agent type in a scope of the following types:

- A scope that contains all tiers.
  OR
- A scope that contains all tiers excluding specific existing tiers.
# Custom Match Rules

A custom match rule in AppDynamics is a type of **transaction detection rule** that lets you define criteria for business transaction discovery and naming for an entry point type. There are two types of custom match rules:

- A custom include rule defines an entry point to use in a new transaction.
- A custom exclude rule lets you prevent AppDynamics from detecting business transactions you are not interested in tracking. For example, application framework services, administrative console requests, or heartbeat pings, etc.

You can use Live Preview to experiment with certain custom match rule configurations and identify potential business transaction entry points:

- During a **Business Transaction Discovery Session**, you can use the Classes/Methods browser or Uninstrumented Code to create POJO and POCO custom match rules.
- When you create or edit a POJO or Servlet entry point, you can use **Live Preview** for the individual rule you are creating.

## Transaction Naming

When an app agent detects a request that matches a custom include rule, it names the business transaction based on the rule name. You can further refine the transaction name using transaction splitting for URI based entry points, for POJOs, or for POCOs.

If a request matches more than one custom include rule, it applies the rule with the highest number priority. For more information, see "Transaction Detection Rule Priorities" on **Transaction Detection Rules**.

## Create a Custom Match Rule

1. Navigate to **Configuration > Instrumentation** and click the **Transaction Detection subtab**.
2. Click the Rules sub-tab.
3. Click **Add**.
   The Add Rule page opens with **Custom Match Rule selected**.
4. Select the **Agent Type** and the **Entry Point Type** and click **Next**.
5. On the **Summary tab**, make general rule configurations:
   - If you are creating a custom match exclude rule, click **Exclude Transactions discovered by this rule**.
   - Enter the rule **Name**. The app agent names requests that match the rule for the rule name.
   - Optionally disable the rule, set the rule **Priority** and assign the rule to a **Scope**.
     For information on rule priorities, see “Transaction Detection Rule Priorities” on **Transaction Detection Rules**.
     For information on scopes, see **Scope Configuration Model**.
6. On the **Rule Configuration tab**, configure the match criteria based upon the entry point type. The PHP web choices, for example, let you specify HTTP-oriented match criteria, such as those for HTTP method, URI, and query parameters.

   **Servlet and POJO entry point types feature a Live Preview button to launch custom match rule live preview session.**
   A live preview streams data from an active node so you can interactively experiment with rule configuration in the
Sample Custom Match Include Rule

The following example demonstrates a custom include rule named cart.GET in a scope named MyCustomScope.

The "cart.GET" rule generates business transactions for Web Service GET requests where the URI begins with /cart/ followed by a number. AppDynamics names the resulting business transaction cart.GET.
You can add conditions to the match rule to match based on specific HTTP parameters or hostnames.

For certain HTTP-based request types, such as Servlets or ASP.NET, a match rule can have more than one HTTP Parameter match condition. A request must match all HTTP Parameter conditions to match this rule. There is an implicit AND operator between the parameters rather than an OR operator.

You can use more complex regular expression matching. For example, for the following request URLs:

- `example.com/aaa123.aspx`
- `example.com/bl.aspx`

To match any incoming request URL to this business transaction, use the following:

```
example\..com/[a-zA-Z]+\?[0-9]+?
```

Any request to a URL that includes `example.com` followed by uppercase or lowercase letters and numbers would match this business transaction detection rule. A URL that has no digits after the letters, such as `example.com/z.aspx` does not match.

See [Using Regular Expressions](#) for more information on using regular expressions in the UI.

**Transaction Splitting for URI Based Entry Points**

You can configure business transaction splitting for the following types of URI based entry points:

- Java Servlet
- ASP.NET
- PHP Web
- Node.js Web
- Python Web
- Web Server.

When you enable transaction splitting, the agent resolves a subset of requests that match the custom include rule into a separate business transaction based on a dynamic part of a request. The app agent names the transaction as follows:

<custom match rule name>.<name derived from the split configuration>.

Consider an application with incoming requests:


The example below illustrates how you can configure transaction splitting to resolve the two requests into two different business transactions, named:

- Checkout.electronics
- Checkout.clothing
You can split servlet transactions based upon the payload. See Split Servlet Transaction by Payload Examples.

Default Custom Exclude Rules

AppDynamics includes default exclude rules for the entry points for frameworks that are not usually of interest. Navigate to Configuration > Instrumentation > Transaction Detection > Rules and filter on Rule Type: Custom Exclude to view or modify the default custom match exclude rules. For more information, see Custom Exclude Rule Examples.

To configure exclude rules for Service Endpoints, see ‘Configure Custom Service Endpoints and Exclude Rules’ on Service Endpoints.
.NET Business Transaction Detection

The topics in this section cover the different types of ASP.NET entry points for business transactions:

- POCO Entry Points
- ASP.NET Entry Points
- WCF Entry Points
- ASP.NET Web Service Entry Points
- Name MVC Transactions by Area, Controller, and Action
POCO Entry Points

On this page:
- POCO Custom Match Rules Overview
- Define a POCO Entry Point
- Task-Based POCO Entry Point Methods

Related pages:
- Monitor Background Tasks
- Transaction Detection Rules
- Custom Match Rules
- .NET Agent for Linux Business Transaction Configuration

Certain types of applications, such as Windows Services or standalone applications, may not have entry points that are automatically detected by the .NET Agent.

For these cases, you can create custom match rules for Plain Old CLR Objects (POCOs), as described here.

POCO Custom Match Rules Overview

To create a POCO custom match rule, define the custom match rule on the .NET class/method that is the most appropriate entry point for the business transaction. Someone who is familiar with your application code can help make this determination. You can also refer to the Custom Match Rules page.

You can optionally enable the Background Task option to have the transaction monitored as a background task. AppDynamics reports only Business Transaction metrics for background task transactions. It does not aggregate response time and call metrics at the tier and application levels for background tasks. See Monitor Background Tasks for more information.

After you create the rule, AppDynamics detects traffic on the matched entry point method and registers a business transaction for it, naming the transaction with the name of the custom match rule.

By default, you have to wait one minute and then restart the application to apply instrumentation changes required for new POCO entry points. If you create an application-level POCO that applies to a tier that has not yet registered with the Controller, you may need to restart the application after the tier registers in order to see the business transaction.

You can enable Runtime Reinstrumentation for the .NET Agent so that you don’t need to restart your application after instrumentation changes.

As an alternative to defining POCO entry points as described here, you can use the transaction discovery tool to create match rules based on discovered transactions. For details, see Business Transaction Discovery Sessions.

Define a POCO Entry Point

On an originating tier, a POCO entry point is the method that starts the business transaction. If the POCO entry point is on a downstream tier, it may correlate to an upstream exit point. When defining a POCO entry point, it is important to choose a method that begins and ends every time the business transaction executes. For more on entry points, see Business Transactions.

Good candidates for POCO entry points include the following:

- A method in a socket application that executes every time a client connects
- A loop in a standalone application that batch processes records via a web service call. For example, an expense reporting system that loops through approved expenses to submit them for reimbursement.
- A Windows service that regularly executes a database call to check for new jobs to process. For example, consider the sample application logic below:
using System.Threading;
using System;
using System.Threading.Tasks;
using System.Configuration;
using System.Collections.Generic;
namespace JobProcessor {
    class JobProcessorCore {
        private void ProcessJob()
        {
            var logic = new JobManagement();
            while (running)
            {
                var jobs = logic.GetJobs(); // Query in-memory Database for a list of jobs to process
                foreach (var job in jobs)
                {
                    logic.GetJobDetails(job, out type, out parameters); // Obtain the details on the specific job at hand
                    ProcessJob(type, parameters); // Execute the job in a separate thread
                }
                if (jobs.Count == 0) // if there was no job to process, wait 1 minute
                    Thread.Sleep(60000);
            }
        }
    }
}

In the example, the POCO entry point would be defined on the namespace JobProcessor, class JobProcessorCore and method ProcessJob.

Task-Based POCO Entry Point Methods

AppDynamics automatically tracks POCO transactions for asynchronous, Task-based method entry points as end-to-end transactions. End-to-end latency metrics reflect the time it takes to complete the asynchronous actions associated with such methods.

To be tracked as an end-to-end transaction, the method you define as the POCO entry point must return a Task object and use the async modifier. In other words, the method signature should look similar to the following:

```csharp
public async Task<int> SearchBestPrice()
{
    ...
}
```

AppDynamics records the response times for such methods as it does for other types of methods—the response time reflects the time from when the method is called to when it returns control to the calling thread. However, for asynchronous task-based methods, AppDynamics records the time it takes for the asynchronous task to be fulfilled as an end-to-end transaction latency metric. For more
information on end-to-end transactions, see End-to-End Latency Performance.
ASP.NET Entry Points

On this page:
- Custom Match Rules for ASP.NET Transactions
- Split Custom ASP.NET transactions

Related pages:
- .NET Business Transaction Detection
- .NET Agent for Linux Business Transaction Configuration
- Transaction Detection Rules
- Business Transactions

AppDynamics automatically detects entry points for client requests to ASP.NET applications. If the request occurs on an originating tier, the method or operation marks the beginning of a business transaction and defines the transaction name. In most cases, this type of entry point maps to a user request or action such as “Cart/Checkout”. AppDynamics allows you to configure transaction naming based upon the ASP.NET request.

For information on how to configure default ASP.NET transaction detection, see URI Based Entry Points.

Custom Match Rules for ASP.NET Transactions

Custom match rules provide greater flexibility for transaction naming. When you define a match rule, AppDynamics uses the rule name for the business transaction name.

See Custom Match Rules for general information on how to create custom match rule.

When AppDynamics detects a request matching your specified criteria, it identifies the request using your custom name. You can use the following criteria to match transactions:

- Method: Match on the HTTP request method, GET, POST, PUT or DELETE.
- URI: Set the conditions to match for the URI.
- HTTP Parameter: Match on HTTP parameter existence or a specific HTTP parameter value.
- Header: Match on a specific HTTP header’s (parameter’s) existence or a specific HTTP header value.
- Hostname: Match on the server hostname.
- Port: Match on the server port number.
- Class Name: Match on the ASP.NET class name.
- Cookie: Match on cookie existence or specific cookie value.

With automatic discovery for ASP.NET transactions enabled, configuring the match on GET or POST causes the agent to discover both GET and POST requests. If you only want either GET or POST requests for the transaction, consider the following options:

- Disable automatic discovery for ASP.NET transactions.
- Create an exclude rule for the method you don’t want: GET or POST.

URI: Set the conditions to match for the URI.

- For rules on regular expressions for .NET, see .NET Framework Regular Expressions.
- Optionally click the gear icon to set a NOT condition.
- You must set a URI match condition in order to use transaction splitting.

HTTP Parameter: Match on HTTP parameter existence or a specific HTTP parameter value.

Header: Match on a specific HTTP header’s (parameter’s) existence or a specific HTTP header value.

- Hostname: Match on the server hostname. Optionally click the gear icon to set a NOT condition.
- Port: Match on the server port number. Optionally click the gear icon to set a NOT condition.
- Class Name: Match on the ASP.NET class name. Optionally click the gear icon to set a NOT condition.
- Cookie: Match on cookie existence or specific cookie value.
- Hostname, Port, and Class Name options are non-functional in the 4.5.9 version of .NET Agent for Linux.

Split Custom ASP.NET transactions

AppDynamics lets you further refine ASP.NET custom transaction names using transaction splitting.

- To use transaction splitting, you must specify URI match criteria for the custom match rule.
- The Split Transactions Using Request Data options work the like the automatic transaction detection configuration options.
described in URI Based Entry Points.

For example, you have a custom match rule named **MyTransaction** that matches the following URL: `http://example.com/Store/Inventory?category=electronics`. You can split the transaction on the parameter value as follows:

<table>
<thead>
<tr>
<th>Summary</th>
<th>Rule Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>▼ HTTP Request Match Criteria</td>
<td></td>
</tr>
<tr>
<td>Match requests which meet ALL of the following criteria: + Add</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>URI</th>
<th>Contains</th>
<th>Inventory</th>
</tr>
</thead>
</table>

▼ Split Transactions Using Request Data

- Split Transactions Using Request Data
- Use **a parameter value** in Transaction names
- Parameter Name **category**

The .NET Agent names the resulting transaction `MyTransaction.electronics`. 
WCF Entry Points

On this page:

- Controlling Correlation Header Location
- Automatic Naming for WCF Transactions
- Custom Match Rules for WCF Transactions

Related pages:

- Configure Business Transaction Detection for .NET
- Windows Communication Foundation (Microsoft Windows Reference)

AppDynamics automatically detects entry points for client requests to Windows Communication Foundation (WCF) services. If the request occurs on an originating tier, the method or operation marks the beginning of a business transaction and defines the transaction name. For information on originating tiers, see Business Transactions.

The .NET Agent detects async entry points for the following patterns:

- Task-based asynchronous operations
- IAsyncResult Begin-End Asynchronous pattern

Controlling Correlation Header Location

In 4.4 and earlier, when WCF calls execute on a downstream tier, AppDynamics includes them as part of the business transaction from the originating tier. If the downstream WCF service uses the NetTcpBinding in WCF, you must register the app agent node property enable-soap-header-correlation with a value of true. See 'enable-soap-header-correlation' on App Agent Node Properties Reference. Otherwise, the .NET Agent will detect the WCF call as the entry point for a new business transaction.

In the following table, when the node property is set to the default (false), the correlation message with header will only successfully deliver when sent over HTTP, not over SOAP with an HTTP wrapper or other protocol. When the node property is set to true, the correlation message with header will succeed in all scenarios.

<table>
<thead>
<tr>
<th></th>
<th>enable-soap-header-correlation</th>
<th>HTTP</th>
<th>SOAP (HTTP)</th>
<th>SOAP (other)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4 default</td>
<td>false</td>
<td>X</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4.4</td>
<td>true</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

In 4.4.1 and later, the enable-soap-header-correlation node property and the default behavior for SOAP calls over non-HTTP protocols in WCF have changed. In addition, a new property, disable-soap-header-correlation-non-http, has been added. See disable-soap-header-correlation-non-http on App Agent Node Properties Reference. With the new default AppDynamics will, by default, include downstream WCF services using NetTcpBinding as part of business transactions.

The following table captures the changes and additions of the correlation data for the following use cases:

- When the node properties for both enable-soap-header-correlation and disable-soap-header-correlation-non-http are set to the default (false), the correlation message with header will only successfully deliver when sent over HTTP or SOAP wrapped in a protocol other than HTTP.
- When the node property for enable-soap-header-correlation is set to true, and the disable-soap-header-correlation-non-http node property is set to the default (false), the correlation message with header will succeed in all scenarios.
- When the node property for enable-soap-header-correlation is set to the default (false), and the disable-soap-header-correlation-non-http node property is set to true, the correlation message with header will only successfully deliver when sent over HTTP, not over SOAP with an HTTP wrapper or other protocol.
- When the node properties for both enable-soap-header-correlation and disable-soap-header-correlation-non-http are set to true, the correlation message with header will only successfully deliver when sent over HTTP or SOAP with an HTTP wrapper, not over SOAP with another protocol.

When upgrading from 4.4.0 to 4.4.1 or later, the enable-soap-header-correlation property retains its previous value and disable-soap-header-correlation-non-http defaults to false.
By default, the .NET Agent does not correlate EUM transactions for WCF entry points. You can enable EUM correlation for WCF using the `wcf-enable-eum` node property. This feature is useful when an instrumented mobile application calls directly into a WCF entry point. See 'wcf-enable-eum' on App Agent Node Properties Reference.

### Automatic Naming for WCF Transactions

By default, the AppDynamics auto-detection naming scheme identifies all WCF transactions using the service name and operation name:

```
ServiceName.OperationName
```

For example, a web service for a travel website books reservations from client front ends: `TravelService.BookTravel`. You can rename or exclude automatically discovered transactions from the Business Transactions List.

### Custom Match Rules for WCF Transactions

Custom match rules provide flexibility for WCF transaction naming. When you define a match rule, AppDynamics uses the rule name for the business transaction name. See Custom Match Rules for more information.

When AppDynamics detects a request matching your specified criteria, it identifies the request using your custom name. You can match based upon:

**Web Service Name**

- For rules on regular expressions for .NET, see .NET Framework Regular Expressions.
- Optionally use the gear dropdown to set a NOT condition.

**Operation Name**

- For rules on regular expressions for .NET, see .NET Framework Regular Expressions.
- Optionally use the gear dropdown to set a NOT condition.

For example, you could report all operations for a TravelService, such as `SearchTravel` and `BookTravel`, as one business transaction.
AppDynamics automatically detects entry points for client requests to ASP.NET web services. If the request occurs on an originating tier, the method or operation marks the beginning of a business transaction and defines the transaction name. For more information on originating tiers, see Configure Business Transaction Detection for .NET.

When web service calls execute on a downstream tier, AppDynamics includes them as part of the business transaction from the originating tier.

**Automatic Naming for Web Service Transactions**

By default, the AppDynamics auto-detection naming scheme identifies all web service transactions using the service name and operation name:

```
ServiceName.OperationName
```

For example, a web service for a travel website books reservations from client front ends.

You can rename or exclude automatically discovered transactions from the business transaction list.

**Custom Match Rules for ASP.NET Web Service Transactions**

Custom match rules provide flexibility for ASP.NET web service transaction naming. When you define a match rule, AppDynamics uses the rule name for the business transaction name.

For steps to access the Custom Match Rules pane, see 'To create custom match rules for .NET entry points' on .NET Business Transaction Detection.

To create an ASP.NET web service custom match rule

1. In the Custom Match Rules pane, click the plus symbol (+) to add an entry point.
2. Click Web Service in the dropdown list. Click Next.
3. Name the New Business Transaction Match Rule.
   - AppDynamics uses the rule Name to name the BT.
   - The Controller enables the rule by default. Disable it later if needed.
   - Set the Priority for the match rule. AppDynamics applies higher priority rules first.
4. Set one or more of the following match criteria. When AppDynamics detects a request matching your specified criteria, it identifies the request using your custom name.

**Web Service Name**

- For rules on regular expressions for .NET, see .NET Framework Regular Expressions.
- Optionally use the gear dropdown to set a NOT condition.

**Operation Name**

- For rules on regular expressions for .NET, see .NET Framework Regular Expressions.
- Optionally use the gear dropdown to set a NOT condition.

For example, to report all operations for a TravelService, such as SearchTravel and BookTravel, as one business transaction:

5. Click **Create Custom Match Rule.**
   The rule appears in the **Custom Match Rule** list. The business application or tier you customized displays a green check in the **Select Application or Tier** pane.

After the agent receives the updated configuration, it discovers the new business transaction and displays it in the Business Transactions list.
By default, the AppDynamics .NET Agent names ASP.NET transactions based upon the URI.

For ASP.NET Core on the full framework, the agent uses MVC area, controller, and action naming by default, and can also be configured to use the URI.

This topic describes the MVC transaction naming configuration options for both ASP.NET and ASP.NET Core on the full framework.

MVC Transaction Naming Overview

For some MVC applications, URI-based naming may not produce meaningful business transaction names. Depending on the URI scheme, it might create many business transactions, pushing you quickly over your business transaction limit and causing most traffic to fall into the All other traffic business transaction.

For example, consider an MVC application that indicates store locations in the URL as the result of a city location search, such as the following sample URL:

http://myapp.mycompany.com/Bellevue

Based on this URL, AppDynamics registers a business transaction named /Bellevue. Each search result with a unique city would similarly generate a unique business transaction.

Alternatively, you can configure the agent to identify transactions by the area, controller, or action name. After you configure the agent to name transactions by controller and action, the agent identifies the business transaction based on the controller and action name, in
In this case, Search/Results.

Similarly, if your application uses areas to logically group controllers, you can configure the use of area name in the business transaction.

For background information on ASP.NET Routing, see the following Microsoft Developer Network article: msdn.microsoft.com/en-us/library/cc668201.aspx

**Configure Area, Controller, Action Naming and URI for ASP.NET**

Configure custom MVC transaction naming using node properties, as follows. The node property works for MVC 3, MVC 4 and WebAPI transactions.

- To name business transactions using the Controller/Action values, register the asp.net-mvc-naming-controlleraction node property with a value of true.
- To name business transactions using the Area/Controller values, register the asp.net-mvc-naming-controllerarea node property with a value of true.
- To have all three component values used in business transaction names, register both properties and configure the ASP.NET transaction naming configuration to use three URI segments instead of the default, which is two. For general information on configuration transaction naming, see [Automatic Transaction Discovery Rules](#).
- To name business transactions using the URI, register the asp.net-core-naming-controllerarea and asp.net-core-naming-controlleraction node properties with a value of false.

For instructions on how to register a new node property, see [App Agent Node Properties](#).

After changing any transaction naming configuration, and once traffic to the previously registered business transactions terminates, you can delete the old business transactions.

**Configure Transaction Naming for ASP.NET Core on the Full Framework**

If you do not want to use the area value in your transaction names, you can omit it. Register the asp.net-core-naming-controllerarea node property with a value of false.

**Configure Transaction Naming for .NET Agent for Linux**

The .NET Agent for Linux can only use MVC area, controller, and action naming. It cannot be configured to use the URI. Register the asp.net-mvc-naming-controlleraction node property with a value of true.
Java Business Transaction Detection

The topics in this section cover the different types of Java entry points for business transactions.

A tier can have multiple entry points. For example, a Java framework implementation may have a combination of pure Servlets or JSPs, Struts, Web services, and Servlet filters, all co-existing as monitored entry points on the same JVM.

The middle-tier components like EJBs and Spring beans are usually not considered entry points because they are normally accessed using either the front-end layers such as Servlets or from classes that invoke background processes.

For details on types of Java entry points and how to set up custom match rules, see:

- Servlet Entry Points
- Struts Entry Points
- Spring Bean Entry Points
- EJB Entry Points
- POJO Entry Points
- Web Service Entry Points
- Binary Remoting Entry Points for Apache Thrift
- CometD Support
- JAX-RS Support
- Spring Integration Support
Servlet Entry Points

On this page:
- Frameworks Supported as Servlets or Servlets Filter
- Custom Match Rules for Servlet Transactions
- Exclude Rules for Servlets

AppDynamics automatically detects requests to servlet-based entry points and generates business transactions based on the requests. For general information on how to customize automatic transaction detection, see URI Based Entry Points.

Frameworks Supported as Servlets or Servlets Filter

AppDynamics supports many web frameworks based on servlets or servlet filters. The servlet configuration settings in AppDynamics apply to these frameworks as well as to plain servlets. Frameworks include:
- Spring MVC
- Wicket
- Java Server Faces (JSF)
- JRuby
- Grails
- Groovy
- Tapestry
- ColdFusion

Custom Match Rules for Servlet Transactions

Custom match rule let you control how business transactions are generated for Servlet-based requests.

**Match Conditions**

Match conditions can be based upon the URI, HTTP method, hostname, servlet name, or other characteristics of the request.

For HTTP Parameter conditions, you can add more than one. If you configure more than one HTTP Parameter match criteria, they must all be met by a request to be subject to this business transaction identification and naming rule, as must all conditions you configure for the rule.

Some of the options have NOT conditions that you can choose to negate the configured condition. Choose this option in the gear icon next to the condition.

**Splitting Transactions using Request Data or Payload**

If you match by URI, you can have the transaction identified (or split) based on values from the request, such as URI patterns, request data, or payload. See URI Based Entry Points for information on naming or splitting transactions by the request elements.

When you create the rule, it appears in the Custom Match Rule list, where you can enable, disable, edit or remove the rule.

See Split Servlet Transaction by Payload Examples for use case examples.

**POST Request Body Parameter Matching Considerations**

You can configure Servlet-based custom match that match POST requests based on HTTP body parameter values. To avoid interfering with the operation of the application, the Java Agent applies some special processing measures and limits on how it accesses the body parameters. This behavior can affect whether requests can be matched by POST body parameters for your particular application. If you are having trouble configuring matching by HTTP parameter in POST requests, you should understand this behavior.

**Transaction Naming Match Criteria**

For transaction match criteria, match conditions for HTTP parameters work under these conditions:

- If the parameter is a query string in the URL of the POST request, as opposed to the request body.
- If the Servlet Request body parameters have already been read and parsed by the application prior to the invocation of the servlet. A Servlet Filter, for example, may do this.

Otherwise, custom servlet match rules do not match HTTP parameter values in POST requests.

**Transaction Splitting**

For transaction split rules based on parameter matching in POST requests, the Java Agent defers transaction naming for incoming...
requests until the application servlet has accessed the request parameters.

The Java Agent considers the parameters “accessed” once the application invokes `getParameterMap()`, `getParameter()`, `getParameterNames()`, or `getParameterValues()` on the ServletRequest object. If the servlet does not call one of these methods, the agent will not apply the split rule and the transaction will not be named.

**Exclude Rules for Servlets**

To prevent specific Servlet methods from being monitored, add a custom `exclude rule`. The controls for selecting Servlets to exclude are the same as those for custom match rules.
Split Servlet Transaction by Payload Examples

### On this page:
- Split by POJO Method Call
- Split Transaction by JSP Name
- Split Transactions on XPath Expression
- Split Transactions on Java XML Binding
- Split Transactions on JSON Payload

AppDynamics lets you split servlet custom match rules according to the request payload. For each of these examples, you must create a servlet custom match rule and enable the URI option on the Rule Configuration tab.

#### Split by POJO Method Call

Using a Java method to name a transaction is useful when:

- You might not have a clear URI pattern, or
- You are using an XML/JSON framework not otherwise supported

For example, consider the `processOrder()` method in the `doPost()` method of the servlet at the following URL: `http://acmeonline.com/store`.

```java
public void doPost(HttpServletRequest req, HttpServletResponse resp) {
    // process the data from the servlet request and get the orderType and the items
    processOrder(orderType, item)
    ...
}

public void processOrder(String orderType, String item) {
    // process order
}
```

You want to derive transaction naming from the first parameter to the `processOrder()` method, `orderType`.

1. On the Rule Configuration tab for the custom match rule, enable the Split Transactions using the XML/JSON payload or Java method invocation.
2. Select POJO Method Call as the splitting mechanism.
3. For the method, choose the `processOrder`, and specify the parameter to use by numeric position in the parameter (0 index). The following screenshot displays the configuration of a custom match rule which will name all the qualifying requests into a `Store.order.creditcard` transaction:
In addition to the parameter, you can also specify either the return type or a recursive getter chain on the object to name the transaction. For example, if the method parameter points to a complex object like PurchaseOrder, you can use something like getOrderDetails().getType() to correctly name the transaction.

### Split Transaction by JSP Name

You can identify transactions by JSP name, as follows:

1. On the Rule Configuration tab, enable the **Split Transactions using the XML/JSON payload or Java method invocation**.
2. Select **POJO Method Call** as the splitting mechanism.
3. Set the name of the class to `com.sun.faces.application.ViewHandlerImpl`.
4. Set the name of the method to `renderView()`.
5. Set the argument index to 1.
6. Define the Method Call Chain or getter as `getViewId()`. The agent appends the value to the name of the transaction as follows: `<Name of the Custom Rule>.<path to jsp>`.

You can later rename these business transactions to a more user-friendly name if you like.

### Split Transactions on XPath Expression

You can access values in an XML payload for business transaction naming or splitting using an XPath expression. Consider the following example from an Ecommerce order transaction where the XML represents an order for three items. The order uses credit card processing, which is the distinguishing element for this body:
The URL is:

http://acmeonline.com/store

The `doPost()` method of the Servlet is:

```java
public void doPost(HttpServletRequest req, HttpServletResponse resp) {
    DocumentBuilderFactory docFactory = DocumentBuilderFactory.newInstance();
    DocumentBuilder docBuilder = docFactory.newDocumentBuilder();
    Document doc = docBuilder.parse(req.getInputStream());
    Element element = doc.getDocumentElement();
    //read the type of order
    //read all the items
    processOrder(orderType, items)
    ...
}
```

Imagine you want to differentiate "Order" transactions based upon the type of order. You can use the XPath expression `//order/type` on this XML payload, which in this example evaluates to `creditcard`.

1. On the Rule Configuration tab, click Split transactions using XML/JSON Payload or a Java method invocation.
2. Choose XPath Expressions for the Split Mechanism.
3. Enter the XPath expression that points to the value of the XML element to use for naming. In this example, `//order/type`.

The agent appends the value of the XPath expression to the name of the business transaction, for example, `Store.order.creditcard`. Even though the agent doesn't name the transaction until after XML parsing, AppDynamics measures the duration of the business transaction to include the execution of the `doPost()` method.

You can use one or more XPath expressions to chain the names generated for the Business Transaction.

You can specify whether the request results in transaction splitting when the expression does not evaluate to a value.

**Split Transactions on Java XML Binding**

You can identify transactions for Java XML data binding frameworks for these types of frameworks:

- Castor
- JAXB
- JibX
In the following example, the posted XML is unmarshalled to the `PurchaseOrderDocument` object, and the `getOrderType()` method should be used to identify the type of the order:

```xml
<acme>
  <order>
    <type>creditcard</type>
    <item>Item1</item>
    <item>Item2</item>
    <item>Item3</item>
  </order>
<acme>
```

The following snippet shows the `doPost()` method for the Servlet:

```java
public void doPost(HttpServletRequest req, HttpServletResponse resp) {
    PurchaseOrderDocument poDoc = PurchaseOrderDocument.Factory.parse(po);
    PurchaseOrder po = poDoc.getPurchaseOrder();
    String orderType = po.getOrderType();

    //read all the items
    processOrder(orderType, items)
    ...
}
```

To split the transaction based upon the XML Binding:

1. On the Rule Configuration tab, check **Split transactions using XML/JSON Payload or a Java method invocation.**
2. Select **Java XML Binding** as the split mechanism.
3. Enter these values for the match criteria:
   - **Unmarshaled Class Name:** `PurchaseOrderDocument`
   - **Method name:** `getOrderType()`

The agent identifies the business transaction for this example as `Store.order.creditcard`.

This custom rule ensures that the Java agent intercepts the method in XMLBeans (which unmarshalls XML to Java objects). If the name of the transaction is not on a first level getter on the unmarshalled object, you can also use a recursive getter chain such as `getOrderType().getOrder()` to get the name.

Although the transaction name is not obtained until the XML is unmarshalled, the response time for the transaction is calculated from the `doGet()` method invocation.

**Split Transactions on JSON Payload**

You can access JSON payload for transaction identification purposes using the method where the Servlet unmarshalls the payload.

For example, the following JSON payload posts an order for an item `car` and uses `creditcard` for processing the order. The URL is `http://acmeonline.com/store`:
order :{
    type:creditcard,
    id:123,
    name:Car,
    price:23
}

The following code snippet shows the `doPost` method of the Servlet:

```java
public void doPost(HttpServletRequest req, HttpServletResponse resp) {
    String orderType = jsonObject.get("type");
    processOrder(orderType,item)
    ...
}
```

After the application unmarshalls the posted JSON payload to the JSON object, the `type` key is available to identify the type of the order. In this case, this key uniquely identifies the business transaction.

To use the JSON payload for transaction identification you must set the `enable-json-bci-rules` node property to true on each node to enable this rule. To configure the rule:

1. On the Rule Configuration tab, check `Split transactions using XML/JSON Payload or a Java method invocation.`
2. For the JSON object key, enter the name of the JSON object. For example, `type`.

The agent automatically intercepts the `JSONObject.get("$JSON_Object_Key")` method to name the transaction. Although the agent doesn't obtain the transaction name until the JSON object is unmarshalled, the response time for the transaction will be calculated from the `doGet()` method.
Struts Entry Points

When your application uses Struts to service user requests, AppDynamics intercepts individual Struts Action invocations and names the user requests based on the Struts action names. A Struts entry point is a Struts Action that is being invoked.

AppDynamics supports the following versions of Struts:

- Struts 1.x
- Struts 2.x

Struts Action invocations are typically preceded by a dispatcher Servlet, but identification is deferred to the Struts Action. This ensures that the user requests are identified based on the Struts Action and not from the generic URL for Dispatcher Servlet.

The response time for the Struts-based transaction is measured when the Struts entry point is invoked.

Struts Request Names

When a Struts Action is invoked, by default AppDynamics identifies the request using the name of Struts Action and the name of the method. All automatically discovered Struts-based transactions are thus named using the convention `<Action Name>.<Method Name>`.

For example, if an action called ViewCart is invoked with the SendItems(), the transaction is named `ViewCart.SendItems`.

For Struts 1.x the method name is always `execute`.

You can rename or exclude auto-discovered transactions. See Organize Business Transactions.

Custom Match Rules for Struts Transactions

For finer control over the naming of Struts-based transactions, use custom match rules.

A custom match rule lets you specify customized names for your Struts-based requests. You can also group multiple Struts invocations into a single business transaction using custom match rules. See Custom Match Rules for information about accessing the configuration screens.

The matching criteria for Struts transactions are: Struts Action class names, Struts Action names, and Struts Action method names.

Exclude Rules for Struts Actions or Methods

To prevent specific Struts Actions and methods from being monitored, add a custom exclude rule. The criteria for Struts exclude rules are the same as those for custom match rules. In particular, you may need to exclude custom-built dispatch servlets, as described next.

Exclude Custom-built Dispatch Servlet

When a Struts action is called, it can demarcate a transaction as an entry point. AppDynamics instruments the Struts invocation handler to get the action name because the Struts action is not an interface. The invocation handler provides the Java Agent with the name of the action being invoked. If the dispatcher Servlet is custom-built and has not been excluded from instrumentation, the wrong entry point could be instrumented and the business transaction could be misidentified.

To address this issue, add a custom exclude rule for the dispatcher servlet or add a BCI exclude for it.
Spring Bean Entry Points

On this page:

- Spring Bean-based Transactions
- Default Naming for Spring Bean Requests
- Custom Match Rules for Spring Bean Requests
- Exclude Rules Spring Bean Transactions

This topic describes how to configure transaction entry points for Spring Bean requests.

Spring Bean-based Transactions

AppDynamics allows you to configure a transaction entry point for a particular method for a particular bean in your environment. The response time is measured from when the Spring Bean entry point is invoked (after receipt at a dispatcher servlet).

Default Naming for Spring Bean Requests

When you enable automatic discovery for Spring Bean based requests, AppDynamics automatically identifies all the Spring Beans based transactions and names these transactions using the following format:

```
BeanName.MethodName
```

By default, the transaction discovery for Spring Bean-based requests is turned off. You can enable it in any Automatic Transaction Discovery rule for Java. Check Discover Transactions automatically for all Spring Bean invocations on the Rule Configuration tab.

When a class is mapped to multiple Spring bean names, by default only the name of the first Spring bean found is used. This may not be ideal for your application such as cases where a call graph for web service A that has Spring beans from web service B. To contend with this scenario, you can remove the Bean name from the transaction name using the capture-spring-bean-names node property, as described on App Agent Node Properties (B-C).

Custom Match Rules for Spring Bean Requests

If you are not getting the required visibility with the auto-discovered transactions, you can create a custom match rule for a Spring Bean based transaction. You can match based on one or more of the following criteria:

- Bean ID
- Method Name
- Class Name
- Extends
- Implements

The following example creates a custom match rule for the `placeOrder` method in the `orderManager` bean.
Exclude Rules Spring Bean Transactions

To exclude specific Spring Bean transactions from detection, add a custom exclude rule.

The criteria for Spring Bean exclude rules are the same as those for custom match rules.
EJB Entry Points

On this page:
- Default Naming for EJB Entry Points
- Enabling EJB Transaction Detection
- Custom Match Rules for EJB based Transactions
- Exclude Rules for EJB Transactions

AppDynamics allows you to configure an EJB-based transaction entry point on either the bean name or method name. The response time for the EJB transaction is measured when the EJB entry point is invoked.

Default Naming for EJB Entry Points

AppDynamics automatically names all the EJB transactions `<EJBNname>`.`<MethodName>`.

Enabling EJB Transaction Detection

By default, automatic transaction discovery for EJB transactions is turned off. To get visibility into these transactions, enable the auto-discovery for EJB based transactions explicitly.

Before you enable auto-discovery for EJB based transactions, note that:

- If the EJBs use Spring Beans on the front-end, the transaction is discovered at the Spring layer and the response time is measured from the Spring Bean entry point.
- AppDynamics groups all the participating EJB-based transactions (with remote calls) in the same business transaction. However, if your EJBs are invoked from a remote client where the App Server Agent is not deployed, these EJBs are discovered as new business transactions.

You can enable auto-discovery for EJB transactions in any Automatic Transaction Discovery rule for Java. Check Discover Transactions automatically for all Spring Bean invocations on the Rule Configuration tab.

Custom Match Rules for EJB based Transactions

If you are not getting the required visibility with auto-discovered transactions, create a custom match rule for an EJB based transaction.

The following example creates a custom match rule for the receiveOrder method in the TrackOrder bean.

In addition to the bean and method names, other match criteria that could be used to define the transaction are the EJB type, class name, superclass name, and interface name.
Exclude Rules for EJB Transactions

To exclude specific EJB transactions from detection add a custom exclude rule. The criteria for EJB exclude rules are the same as those for custom match rules.
POJO Entry Points

On this page:
- About POJO Custom Match Rules
- Creating POJO Custom Match Rules
- Matching by Class and Method
- Matching on Inner Classes and Inner Interfaces
- Splitting Matches into Separate Business Transactions
- Excluding Matches from Transaction Splitting

Related pages:
- Custom Match Rules

This topic describes how to create custom match rules for POJO (Plain Old Java Object) applications. See Custom Match Rule Live Preview for instructions about interactively working with live data to create a POJO.

About POJO Custom Match Rules

Custom match rules for POJOs let you configure business transaction detection in application environments that run pure Java applications or that use frameworks with entry points that are not automatically detected.

Unlike common frameworks, which are characterized by well-known entry points for applications, the logical entry point for a business transaction for a POJO entry point could be any method in the application.

To configure a custom POJO entry point, therefore, you need to identify the method that AppDynamics should consider the business transaction entry point. Keep in mind that the start and end of the execution of the method will correspond to the start and end of the business transaction, so the method should encapsulate the complete execution of the business transaction.

For example, consider the method execution sequence:

```java
com.foo.threadpool.WorkerThread.run()
calls com.foo.threadpool.WorkerThread.runInternal()
calls com.foo.Job.run()
```

The first two calls to `run()` method are the blocking methods that accept a job and invoke it. The `Job.run()` method is the actual unit of work, because Job is executed every time the business transaction is invoked and finishes when the business transaction finishes.

Methods like these are the best candidates for POJO entry points. The response time for POJO transactions is measured from this entry point, and remote calls are tracked the same way as are remote calls for a Servlet's Service method.

Creating POJO Custom Match Rules

You can create a POJO entry point by adding a custom match rule with POJO as the Entry Point Type. The agent names business transactions for the custom match rule.

In order for the agent to discover POJO transactions, your custom match rule must belong to a scope that also includes a Default Java Automatic Transaction Discovery rule with POJO transaction monitoring enabled.

You can classify matching transactions as background tasks by enabling the Background Task check box. When a request runs as a background task, AppDynamics reports only Business Transaction metrics for the request. It does not aggregate response time and calls metrics at the tier and application levels for background tasks. This ensures that background tasks do not distort the baselines for the business application. Also, you can set a separate set of thresholds for background tasks. For more information, see Monitor Background Tasks.

The custom match rule configuration offers several options for matching method invocations and splitting matched calls into separate business transactions, as described below.

If you are running on IBM JVM v1.5 or v1.6, you must restart the JVM after defining the custom match rules.
Matching by Class and Method

You can specify matching criteria for the custom rule based on various forms of a method or class to which the method belongs.

When specifying the method name matching criteria, use parameter matching to match against a particular method signature. For example, say you want to instrument one of more methods in the following class:

```java
class A {
    public void m1();
    public void m1(String a);
    public void m1(String a, com.mycompany.MyObject b);
}
```

Configure instrumentation for each method based on its signature as follows:
- To instrument the first method signature (with no parameters), create a POJO-based business transaction match rule as follows:
  - Match Classes: with a Class Name that Equals A
  - Method Name: Equals m1()
- To instrument the second method, create a match rule as follows:
  - Match Classes: with a Class Name that Equals A
  - Method Name: Equals m1(java.lang.String)
- To instrument the third method, create a match rule as follows:
  - Match Classes: with a Class Name that Equals A
  - Method Name: Equals m1(java.lang.String, com.mycompany.MyObject)

You can also match methods that belong to classes with certain annotations. For example, say you want to match all classes that are annotated with `@com.acme.Processor`. The custom rule configuration can define the annotation as follows:

<table>
<thead>
<tr>
<th>Summary</th>
<th>Rule Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Match Class &amp; Method</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Match Classes</strong></td>
<td></td>
</tr>
<tr>
<td>that has an Annotation which</td>
<td></td>
</tr>
<tr>
<td><strong>Equals</strong></td>
<td>com.acme.Processor</td>
</tr>
<tr>
<td><strong>Method Name</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Equals</strong></td>
<td>process</td>
</tr>
</tbody>
</table>

When calls to the `process()` method match, the result is a business transaction named for the custom match rule name.

Matching on Inner Classes and Inner Interfaces

You can match on inner classes and inner interfaces by adding a $ sign after the class name. For example, you could specify the
following for the class name:

com.my.package.OuterClass$InnerClass

Splitting Matches into Separate Business Transactions

With only transaction matching configured, all requests matched by the match rule belong to the custom business transaction. Alternatively, you can create a match rule that generates multiple named business transaction based on criteria in the request.

For example, the following configuration defines a custom match rule that matches by superclass. In the example, the entry point matches the `process()` method defined in classes that extend the `com.acme.AbstractProcessor` superclass. In our example, the superclass is extended by the subclasses `SalesProcessor`, `InventoryProcessor`, and `BacklogProcessor`.

1. Create a custom rule that matches the superclass matches all of those subclasses:
2. Split the transaction on class name, you can have separate business transactions created for the respective subclasses. AppDynamics names the transactions with the class name, prepending the match rule name to the class name. For example, `Process.SalesProcessor`, `Process.InventoryProcessor`, and `Process.BacklogProcessor`.

It's possible that parameter values carry what would be meaningful identifiers for business transactions in your application. For example, say the `jobType` parameter in the following method may have the values of `Sales`, `Inventory`, or `Backlog`.

```java
public void process(String jobType, String otherParameters...)
```
You can split the transaction based on parameter value by indicating the zero-based position of the parameter in the method signature, 0 in the example:

- **Transaction Splitting**
  - Split POJO Transactions
  
  Name Business Transactions using the following...
  
  Parameter index 0  Getter chain toString()

The `toString()` method indicates how the value of the parameter should be rendered. See Using Getter Chains for more information. As indicated in the dialog, you can use other transaction splitting criteria as well, including thread ID, method name, class name, and so on.

The name of the rule is prepended to the dynamically-generated name to form the business transaction name.

**Excluding Matches from Transaction Splitting**

An exclude rule defines criteria which, when matched by a transaction instance, prevents transaction splitting from occurring as otherwise configured. In effect, this nullifies the effects of transaction splitting for the subset of transactions matched by the exclude condition.
Monitor Java Interface Static and Default Methods

As of Java 8, Java interfaces can include static and default methods. This topic describes how to monitor these types of interface methods.

Note that another Java language feature introduced in Java 8, lambda method interfaces, are not supported by the AppDynamics Java Agent.

Interface Static Methods

The following code snippet shows an example static interface method:

```java
package jdk8;

interface InterfaceForStaticMethod {
    static int sum(int acc, int x) {
        return acc + x;
    }
}
```

The actual bytecode for the method would reside in the compiled class file, so a class match against the interface class in the AppDynamics configuration would result in the instrumentation of that class for a business transaction.

To instrument such methods, create a POJO business transaction match rule for the `InterfaceForStaticMethod`, such as:

```
Summary   Rule Configuration

- Match Class & Method
  Match Classes
    with a Class Name that
    Equals jdk8.InterfaceForStaticMethod

  Method Name
    Equals sum

Browse classes and methods
```
In this case, the rule would match classes with a Class Name of `jdk8.InterfaceForStaticMethod`.

**Interface Default Methods**

Interfaces can also define default methods. As with interface static methods, the bytecode for default methods resides in the compiled class files.

When configuring instrumentation for a default method, note these points:

1. If the targets are classes that do *not* override the default method, target the interface via a class match.
2. If the targets are classes that do override the default method, target the interface via an interface match.

**Instrumenting an Interface Default Method Directly**

Given the following interface:

```java
package jdk8;

interface InterfaceForConcreteClassDefaultMethod {
    default int sum(int acc, int x) {
        return acc + x;
    }
}
```

And its implementing class:

```java
package jdk8;

class ClassForDefaultMethod implements InterfaceForConcreteClassDefaultMethod {
}
```

In this example, the `ClassForDefaultMethod` does *not* override the default method. Therefore, you can use a class match against the interface such as:
Instrumenting an Overridden Default Method

Given the following interface:

```java
class jdk8;

interface InterfaceForAnonymousClassDefaultMethod
{
    default int sum(int acc, int x)
    {
        return acc + x;
    }
}
```

And:

```java
InterfaceForAnonymousClassDefaultMethod i = new InterfaceForAnonymousClassDefaultMethod()
{
    @Override
    public int sum(int acc, int x)
    {
        return acc + x;
    }
};
```

Because the method is overridden, the target bytecode resides in the anonymous class, `InterfaceForAnonymousClassDefaultMethod`. So you need to create an interface match POJO business transaction match rule such as:
<table>
<thead>
<tr>
<th>Match Class &amp; Method</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Match Classes</strong></td>
<td></td>
</tr>
<tr>
<td>with a Class Name that</td>
<td></td>
</tr>
<tr>
<td>Equals</td>
<td>jdk8.InterfaceForAnonymousClass</td>
</tr>
<tr>
<td>Method Name</td>
<td></td>
</tr>
<tr>
<td>Equals</td>
<td>sum</td>
</tr>
</tbody>
</table>

*Browse classes and methods*
Instrument Java Lambda Expressions

On this page:
- About Java Lambda
- Instrumenting Java Lambda

This topic discusses Lambda expressions, a Java language feature introduced in Java 8 that enables a functional style of programming in Java.

About Java Lambda

Here's a short example:

```java
LambdaMethodInterface i = (acc, x) -> acc + x;
```

In the JVM, this creates an anonymous instance of the associated `FunctionalInterface`, as shown below:

```java
package jdk8;

@java.lang.FunctionalInterface
interface LambdaMethodInterface
{
    int sum(int acc, int x);
}
```

Instrumenting Java Lambda

To instrument lambda instances, create a POJO business transaction match rule for the `LambdaMethodInterface`, for example:
This sample rule matches classes that implement the interface name `jdk8.LambdaMethodInterface`. 
Web Service Entry Points

When your application uses Web Services to service user requests, AppDynamics intercepts the Web Service invocations and names requests based on the Web Service action names and operation name. A Web Service entry point is a Web Service end point that is being invoked.

This is relevant only when the Web Service invocation is part of the entry point tier and not in a downstream tier.

Web Service invocations are usually preceded by a dispatcher Servlet, but identification is deferred to the Web Service endpoints. This configuration ensures that the requests are identified based on the Web Service and not based on the generic URL for the dispatcher Servlet.

**Default Naming**

When the Web Service endpoint is invoked, the request is named after the Web Service name and the operation name. For example, if a service called CartService is invoked with the Checkout operation, the is named CartService.Checkout.

You can rename or exclude these automatically discovered transactions. See Organize Business Transactions.

**Custom Match Rules for Web Services**

You can aggregate different Web Service requests into a single business transaction using the web service name or the operation name. You do this by creating custom match rules for Web Services. See Custom Match Rules for information about accessing the configuration screens.

The following example names all operations for the Web Service named CartService:

<table>
<thead>
<tr>
<th>Summary</th>
<th>Rule Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Business Transaction Match Criteria</td>
<td></td>
</tr>
<tr>
<td>Match requests which meet ALL of the following criteria:</td>
<td>Add</td>
</tr>
<tr>
<td>Web Service Name</td>
<td>Equals</td>
</tr>
</tbody>
</table>

**Exclude Rules**

To exclude specific Web Services or operation names from detection, add an exclude rule. See Custom Match Rules. The criteria for Web Service exclude rules are the same as those for custom match rules.

**Transaction Splitting for Web Services Based on Payload**

1. Disable the Web Service automatic transaction discovery.
2. Disable the following default exclude rules:
   - Apache Axis Servlet
   - Apache Axis2 Servlet
   - Apache Axis2 Admin Servlet
3. Add the custom match rule for Axis or Axis2 Servlet (based on the version being used) and split the transaction using payload
or request depending on the pattern in your scenario.
Apache Thrift is a binary remoting protocol. Cassandra uses the Thrift protocol to achieve portability across programming languages. Applications written in many different languages can make calls to the Cassandra database using the Thrift protocol.

AppDynamics is preconfigured to detect business transaction entry points for Cassandra with Thrift framework applications. Note that Binary Remoting entry points are not available as Service Endpoints.

AppDynamics measures performance data for Thrift transactions as for any other transaction. Thrift entry points are POJO-based. The response time for the transaction is measured from the POJO entry point, and the remote calls are tracked the same way as remote calls for a Servlet's Service method.

**Default Naming for Binary Remoting (Thrift) Entry Points**

When the automatic discovery for a request using Binary Remoting (Thrift) protocol is enabled, AppDynamics automatically identifies all the transactions and names them using the following format:

```
RemoteInterfaceClassName:MethodName
```

**Enabling Detection for Binary Remoting (Thrift) Entry Points**

Binary Remoting (Thrift) entry points are enabled by default, but if you are not seeing them in the Tier Flow Map, verify that they have been enabled in the Java automatic transaction detection rule. For more information, see Transaction Detection Rules.

When enabled, you can see Thrift calls from calling tiers to the Cassandra database in the List View of the Tier Dashboard. Since the transactions using the Thrift protocol are POJO-based, they appear as POJO in the Type column.

**Creating Custom Match Rules for Binary Remoting (Thrift) Requests**

If you are not getting the required visibility with the auto-discovered transactions, you can configure custom match rules and transaction splitting for specific classes and methods.

When creating the custom match rule, choose POJO as the rule type. The rule should be defined on the class and method you want to use as the entry point. Someone who is familiar with your application code may need to help you make this determination.

For information about and examples of the various POJO-based business transaction match rules you can create for Binary Remoting (Thrift), see POJO Entry Points. The rules for Binary Remoting (Thrift) entry points are the same as those for POJO entry points.
CometD Supports

On this page:
- About the Built-in Support
- Create Container Transport Servlet Exclude Rules

Related pages:
- Jetty Startup Settings
- Exclude Rules

CometD is an HTTP-based event routing bus for transporting asynchronous messages over HTTP. CometD provides both a JavaScript API and a Java API and is often used for real-time interactive applications such as multi-player games and chat rooms.

CometD is distributed as a WAR file. When you instrument the CometD application container that runs the WAR (typically a Jetty server) with the Java agent, AppDynamics automatically detects and tracks CometD activity, as described in the following section.

About the Built-in Support

AppDynamics supports both the WebSocket and the HTTP long-polling client transports in CometD. It tracks CometD messages as they traverse channels using POJO rules that split business transactions based on channel name.

Accordingly, business transactions in AppDynamics correspond to CometD channels. The following screenshot shows a business transaction generated for the `/market/nyse` channel:

![Business transactions are identified by channel](image)

AppDynamics provides built-in exclude rules for the servlets that implement CometD transport functions, `org.cometd.annotation.AnnotationComet` and `org.cometd.server.CometdServlet`. For best results, you should exclude tracking for the container servlet as well, as described in the next section.

Create Container Transport Servlet Exclude Rules

CometD is implemented as a servlet. Since the servlet implements transport activity, you should create exclude rules for the servlets that perform the CometD transport function. This ensures that the results of business transaction discovery do not obscure the POJO-based BTs corresponding to CometD channels.

CometD is generally contained within a Jetty container, so the rules should exclude the Jetty container tracking in most cases. For Jetty, you should configure exclude rules for these Jetty servlets:

- `EQUALS`, `org.eclipse.jetty.server.handler.ContextHandler`
- `EQUALS`, `org.eclipse.jetty.servlet.ServletContextHandler`
- `ENDSWITH`, `jetty.plugin.JettyWebAppContext`
If using another type of container, you should create similar exclude rules for the equivalent supporting classes.
JAX-RS Support

The Java Agent supports Jersey 1.x and 2.x by default.

Business transaction entry points are named using the following app agent node properties:

- rest-num-segments
- rest-transaction-naming
- rest-uri-segment-scheme

Business Transaction Naming Scheme

By default, the Java Agent makes a best-effort basis to name JAX-RS-based transactions using the class annotation, method annotation, and HTTP method of the detected entry point, in the following form:

```{class-annotation}/{method-annotation}.(http-method)```

In some situations where the annotation values are not available, the agent may name the transaction as follows:

```{fully qualified class name}.{method}```

For example, when the JAX RS service endpoints don't have the class and method names in them explicitly, but the annotations are inferred in some other way.

For annotation inheritance cases, the agent attempts to fetch the annotations from the superclasses and interfaces as described by the JAX RS specification. The specification makes the following recommendation:

> For consistency with other Java EE specifications, it is recommended to always repeat annotations instead of relying on annotation inheritance.

Repeating annotations on the endpoint methods enables the Java Agent to name business transaction according to the default scheme.

rest-transaction-naming Variables

The `rest-transaction-naming` node agent property enables you to control naming for JAX-RS-based business transactions. You can use variables to have the transactions named with runtime values, for example, based on the method name, class name, parameter values, and more. For a full list of variables, see App Agent Node Properties Reference.

You can add a getter chain after any of the parameters to operate on the parameter values. To do so, add `:` followed by the getter chain. Standard getter chain escape requirements and rules apply.

Naming Examples

The following example shows naming based on a part of the value of the sixth parameter:

```rest-transaction-naming={param-5.toString().toLowerCase().substring(5,20)}```

The characters '{' and '}' are reserved. If used in getter chains or as text in the name the characters must be escaped with '\'. E.g. `{class-name}` would display as `com.singularity.RestObject`.

You can use characters in addition to parameters. For example, in the following example, the class name and method name are separated by a slash:
rest-transaction-naming={class-name}/{method-name}

Based on this example, when the Java Agent sees a REST resource with a class name of `com.company.rest.resources.Employees` with a `CreateNewEmployee` method, it names the business transaction `com.company.rest.resources.Employees/CreateNewEmployee`. 
Spring Integration Support

On this page:
- **Entry points**
- **Exit points**
- **Spring Integration Support Customizations**

In Spring-based applications, **Spring Integration** enables lightweight messaging and supports integration with external systems via declarative adapters.

The Java Agent automatically discovers entry points based on the **MessageHandler** interface. The Java Agent automatically discovers exit points for all Spring Integration Release 2.2.0 channels except **DirectChannel**.

The Java Agent supports tracking application flow through Spring Integration messaging channels. For pollable channels:

- A continuing transaction is tracked if the Spring Integration framework is polling for messages.
- If the application code polls for messages in a loop, the span of each loop iteration is tracked as a transaction. Tracking begins when the loop begins and end it when the iteration ends.

**Entry points**

Originating transactions begin with **MessageHandler.handleMessage()** implementations. If the incoming message is already recognized by the Java Agent then a continuing transaction is started.

**Exit points**

Exit points are based on **MessageChannel.send()**.

Most message channels exist inside the JVM. To represent this arrangement in a flow map, the Controller UI shows a link from the tier to the message channel component name (bean name) and then back to the tier.

**Spring Integration Support Customizations**

The following sections describe typical scenarios for tuning the default Spring Integration monitoring.

**Track Application Flow Before Message Handler is Executed**

In cases where a lot of application flow happens before the first MessageHandler gets executed, you should enable tracking the application flow as follows:

- Find a suitable POJO entry point and configure it.
- Set the **enable-spring-integration-entry-points** node property to **false**. This property is set to **true** by default.
- Restart the application server

**Limit Tracking of Looping Pollable Channels**

To safeguard against cases where **pollableChannel.receive()** is not called inside a loop, you can ensure that the Java Agent tracks a pollable channel loop only if it happens inside a class/method combination similar to that defined in the following example.

You can configure the **spring-integration-receive-marker-classes** node property for each class/method combination that polls messages in a loop, in which case only those class/methods identified in this node property are tracked.

```java
class MessageProcessor {
    void process() {
        while(true) {
            Message message = pollableChannel.receive()
        }
    }
}
```

For example, for the loop above, set the **spring-integration-receive-marker-classes** node property as follows and restart
the application server:

```
spring-integration-receive-marker-classes=MessageProcessor/process
```

**Note:** The `spring-integration-receive-marker-classes` node property must be configured before the method `process()` gets executed for any changes to take effect. Restart the application server after setting this property.
PHP Transaction Detection

On this page:

- PHP Entry Points
- Enable Monitoring by Entry Point
- Transaction Naming
- Customizing Transaction Detection and Naming

Related pages:

- Configuration Import and Export API
- Configure PHP Custom Match and Exclude Rules

This topic describes transaction detection and configuration procedures that related specifically to the PHP Agent. For general information about configuring business detection, see Transaction Detection Rules.

**PHP Entry Points**

The entry point is where the business transaction begins, typically a URI or a page callback name, MVC controller action, page template name, or page template. The PHP Agent automatically detects and monitors the entry point types listed under PHP Frameworks and Protocols in PHP Supported Environments.

If the agent cannot detect the framework, the entry point type defaults to PHP Web. PHP Web automatically detects all HTTP requests to the application.

**Enable Monitoring by Entry Point**

You can enable and disable transaction monitoring for each entry point type.

When monitoring is disabled, the agent stops counting, measuring, and recording all activity on servers of that type.

Enable or disable monitoring by checking or clearing the Transaction Monitoring checkbox in the Transaction Detection window for the PHP entry point type.

**Transaction Naming**

By default, PHP Agent names transactions are named after the first two segments in the service request URL. You can modify the default naming scheme in the same manner as for any web-based business transaction entry point.

For PHP-based entry point tiers that use a multi-layered framework architecture, the last framework that processes the transaction determines the name of the PHP business transaction. For example, for an incoming request processing path of PHP Web > MVC > Drupal 8, a new business transaction would be named based on the Drupal 8 invocation rather than the preceding layers in the request processing flow.

**Virtual Host Naming by Entry Point**

If you have multiple virtual hosts configured on a single web server, you can differentiate among the business transactions by adding the virtual host name as a prefix to the default business transaction name. Enable use of the prefix by checking the Use Virtual Host in Business Transaction names checkbox in the Transaction Detection window for the PHP entry point type. Transactions discovered and named subsequent to this configuration will use the virtual host name as a prefix to the default name.

For example, an application has two exit.php actions running on the same physical host. One action runs on a virtual host named phpagent1 and the other runs on a virtual host named phpagent2. Transaction naming will occur as follows:

- virtual host name is not used: agent identifies all requests to exit.php as a single business transaction named exit.php
- virtual host name is used: agent identifies requests as two different business transactions named phpagent1 : exit.php and phpagent2 : exit.php

**Customizing Transaction Detection and Naming**

If the default contentions do not produce transactions to fit your needs, you can change the transaction configuration as follows:

- Create exclude rules to eliminate entry points that you do not need to monitor.
- Create custom match rules that specify matching rules for identifying and naming transactions.
• For PHP Web entry points, you can also configure transaction naming.

To access the transaction detection screens, click Configuration > Instrumentation > Transaction Detection > PHP-Transaction Detection.
Configure PHP Custom Match and Exclude Rules

This topic summarizes the criteria that you can use for creating custom match and exclude rules for each entry point type.

The criteria are the same for custom match and exclude rules.

If you disable automatic detection for an entry point type and then apply custom match rules, the agent discovers only requests defined by those match rules. For example, if you have a PHP MVC custom match rule that matches on "Action contains search", and another rule that matches on "Action contains view", requests that contain "search" or "view" are the only MVC transactions that the agent detects.

On the other hand, if you do not disable automatic detection and you apply a custom match rule "Action contains search", all the requests that match that criterion are grouped together as a single "search" transaction and then any other requests (those that do not contain "search") are discovered using automatic detection.

**Accessing Custom Rule Configuration**

1. Click Configuration Instrumentation > Transaction Detection > PHP-Transaction Detection.
2. Then scroll down to either the Custom Match Rule section or the Exclude Rule section.
3. Click the + to create a new rule, then select from the dropdown menu the entry point type to which the rule applies.

**PHP Web**

Matching criteria can include one or more of the following:

- HTTP method
- URI segments
- HTTP parameter (value or existence)
- HTTP header (value or existence)
- Port
- Cookie (value or existence)

See also Set Priority When Multiple Rules Apply in Custom Match Rules.

**Transaction Splitting**

For PHP Web, if a transaction is configured to identify the entry point based on the URI, you can optionally split the transaction into multiple transactions. For example, a login request may be automatically detected as a single transaction, but you want to split it into two transactions based on whether the request branches to a new-user or existing-user operation.

You can split a PHP Web transaction using request data from the transaction names such as URI segments, parameter values, header values, and cookie values.

**PHP Web Splitting Example**

For example, to split a custom match rule transaction "mobile checkout" by the mobile carrier parameter, so the agent detects separate transactions as follows:

- products/mobile/checkout?carrier=verizon
- products/mobile/checkout?carrier=sprint
- products/mobile/checkout?carrier=att
To split transactions based upon the carrier parameter:

1. On the Rule Configuration tab, set a URI match option.
2. Check **Split Transactions Using Request Data**.
3. Select **Use a parameter value in Transaction names**.
4. Enter "carrier" for the **Parameter Name**.

The agent creates the following transactions: "mobile checkout.verizon", "mobile checkout.sprint", and "mobile checkout.att".

**PHP Web Custom Match Rules vs Auto-Detection Rules of Other Types**

If a custom match rule configured for PHP Web matches the same criteria as an auto-detection rule for a business transaction of a different transaction type in the same application, the PHP Web custom match rule takes precedence and the business transaction appears as a PHP Web transaction type.

However, if a custom match rule for PHP Web and a custom match rule for another transaction type match the same criteria, precedence depends on the values of the rules’ priorities. See Setting Match Rule Priority for information on rule priorities.

**PHP MVC**

By default, for most MVC frameworks, the business transaction is named using the controller:action. For modular MVC frameworks, the business transaction is named using the module:controller:action.

You can modify detection to match only a portion of the module, controller or view name (such as "Begins with" or "Contains") or on a regular expression, rather than the default which matches the entire name using "Equals" for the match condition.

You cannot configure a single match rule that matches on both the controller and the action. You can create MVC custom match rules using any of the following:

- controller name only
- action name only
- module name and action
- module and controller name

**PHP Web Service**

By default, for Web Services frameworks, the business transaction is named for the entire web service name and the entire operation name using "Equals" for the match condition.

You can modify detection with a custom rule that matches on only the web service name or only the operation name. You can match on a portion of either name or use a regular expression.

**PHP Drupal**

By default, AppDynamics automatically names Drupal transactions based on the entire page callback name of the Drupal module using "Equals" for the match condition.

You can modify detection with a custom match rule that uses just a portion of the page callback name or use a regular expression.

**PHP Wordpress**

By default, AppDynamics automatically names Wordpress transactions based on the entire Wordpress template name using "Equals" for the match condition.

You can modify detection with a custom match rule that uses just a portion of the template name, or use a regular expression.

**PHP CLI**

By default, AppDynamics automatically names PHP CLI transactions based on the last two segments of the script’s directory path plus the name of the script.

The agent creates a business transaction instance every time the script is run.

You can modify detection with a custom match rule that matches on a different portion of the script path and name or uses a regular expression.
Custom Exclude Rule Examples

On this page:

- Change the Default Exclude Rule Settings
- Define Transaction Names Using Downstream Application Logic
- Use an Exclude Rule as a Filter

Related pages:

- Custom Match Rules

Custom exclude rules prevent detection of business transactions that match certain criteria. You might want to use exclude rules in the following situations:

- AppDynamics detects business transactions that you do not want to monitor.
- To stay within agent and controller limits, you want to reduce the total number of business transactions.
- You need to suppress a default entry point to allow for the detection of a more precise entry point defined by a custom match rule.

You can customize existing rules or set up new rules, as described in the following section. See Custom Match Rules for general instructions to configure custom exclude rules.

Change the Default Exclude Rule Settings

AppDynamics has a default set of custom exclude rules for various agent types. Sometimes you need to disable a default rule or add a new custom exclude rule. Different entry point types include different match criteria that you can use to exclude transactions.

You can view the default rules on the Transaction Detection > Rules Tab:
Define Transaction Names Using Downstream Application Logic

Sometimes an incoming request invokes control logic in your code that uses payload data to execute different business logic. In such cases, it makes sense to name your business transactions for the business logic class and method. You can use a custom exclude rule to prevent AppDynamics from naming the business transaction for control logic. Then you can identify the transaction by the business logic instead.

Consider an example where you have implemented both the control logic and business logic as EJBs. With EJB discovery enabled, AppDynamics discovers and names business transactions based on the control class and method. You can create an EJB exclude rule for the control class name using the Class Name Equals match criteria. After you create the rule AppDynamics discovers and names the business transactions for your business logic APIs:
Use an Exclude Rule as a Filter

Custom exclude rules work well to filter out unwanted requests from transaction discovery. You can specify match criteria that allow eligible requests but ignore everything else.

For example, you want to use default discovery rules. Your application receives URI ranges that start with :/a, /b ... /z. However, you only want to monitor URIs only when they start with /a. You can create a custom exclude rule that does a “Match : Doesn't Start With /a” as shown here:

In another example, imagine a specific application that implements Spring Beans of the classes java.lang.String and java.util.ArrayList. This means that AppDynamics identifies all instances of these classes as Spring Beans with the same IDs. To fix the problem, you can define a custom exclude rule specific Spring Bean IDs:
Custom Match Rule Live Preview

When you create or edit servlet or POJO custom match rules in AppDynamics, you can click **Live Preview** to inspect pertinent transaction data from a node.

Custom transaction detection rule Live Preview session lets you interactively preview the effect of rule updates on your business transactions. Live Preview also provides tools for the different rule types to let you inspect the live data streaming from your application.

When you launch Live Preview, select the node where you want to preview transactions. Nodes are organized by tier. You can see which nodes have active agents and if there is a live preview session already running on a node.

**Permissions**

🔒 You need the following permissions to use Live Preview:

- Configure Transaction Detection
- View Sensitive Data

For information on AppDynamics Role Based Access Control, see [Roles and Permissions](#).

**Preview Business Transactions**

When you initiate a custom match rule live preview session, the Controller shows the rule configuration in the left pane. Click **Preview Business Transactions** in the right pane to see resulting transactions. You can edit the rule and click **Apply** to update the changes within the session.

When the agent discovers an entry point for a rule but the transaction payload contains correlation data from an existing transaction, Live Preview displays a **masked transaction**.

**POJO Match Rule Tools**

You can use the following tools to inspect data and configure the entry point for a POJO custom match rule:

- The Classes/Methods browser lets you search for classes. You can right-click a class or method in the result to set the class and method match criteria for the POJO. You can also view method invocations for a selected method.

  In live preview mode, the number of methods and classes that can be returned by an agent are now limit-bounded. The limits are:

  - Maximum number of classes: 1000
  - Maximum number of methods: 300

  You can change limits by modifying the node properties `classmethodservice.max.no.of.classes.reported` and `classmethodservice.max.no.of.methods.reported`, respectively

- Method Invocations lets you inspect live data from the server for the specified method. Expand a method invocation to see detailed information including invoked objects, parameters, and return values. Right-click on a data point to use it in transaction splitting. AppDynamics automatically creates a splitting rule following the `get<field>()` syntax. If your code follows another convention, you need to manually specify the getter method for the data point. You can also copy a **getter chain** to the clipboard.
The example below shows the data for a method, `Order.setQuantity()`:

Servlet Match Rule Tools

When you use Live Preview to inspect data for a servlet custom match rule, AppDynamics shows live HTTP Requests that meet the rule criteria. You can click an individual Request URL to view detailed request data: headers, cookies, and parameters.

Right-click an individual data point, such as a header, to add the value to the match criteria or to set it as a transaction splitting option. The following example, shows how to add a specific header value to the match criteria:
You can group the incoming HTTP requests by various parts of the request and its payload, such as the URI, an HTTP parameter, a header, etc. You can right-click a result from the Values column and add the value to the HTTP Request Match criteria.

The following example shows Live Preview of HTTP requests grouped by URI:

**Masked Transactions**

Live Preview flags transactions that are downstream from another existing business transaction as masked transactions. To view the originating business transactions masking the current transaction, click the link indicating the number of transactions.
To view the business transaction dashboard for the masking transaction, you can click the link for a business transaction name.

If you want to view metrics for the current entry point outside of a masking transaction or transactions, you can either:

- Disable the business transaction detection rules that created the upstream transactions, or
- Create a service endpoint for the entry point.

Otherwise, you can continue to track the performance of the masked entry point as a segment of the business transaction it currently belongs to.
For almost every web-based framework or platform, there is a URI based business transaction entry point for user requests. AppDynamics App Agents automatically discover the following URI-based entry points:

- Java Agent: Servlet entry points
- .NET Agent: ASP.NET entry points
- Node.js Agent: Node.js web entry points
- Python Agent: Python web entry points

The Web server Agent can detect Web request entry points, but automatic discovery is turned off by default so an agent on a downstream node can name the transaction.

URI based entry points all have similar configuration options for Automatic Transaction Discovery Rules and Custom Match Rules.

For additional functionality that applies to servlets, see Servlet Entry Points.

**Default Naming**

By default, when an app agent detects a URI request, it names the business transactions using the first two segments of the URI. For example, consider the following URI for a checkout operation in an online store: http://acmeonline.com/store/checkout.

The agent names the business transaction "/store/checkout" and assigns all requests that match that URI to the "store/checkout" business transaction.

If the first two segments of the URI do not contain enough information to effectively identify the business transaction in your environment, you can edit the automatic discovery rule to name the transaction based upon the URI and its context. Alternatively, you can create a custom match rule to specify a name.

For ASP.NET Core on the full framework, the default naming convention uses the Controller, action, and area. See Name MVC Transactions by Area, Controller, and Action for more information and configuration options.

**Customize Automatic Naming Rules**

When you edit or create a new Default Automatic Discovery Rule for URI based entry points, you can adjust naming to:

- Use the full URI to name transactions.
- Use part of the URI to dynamically name transactions.

Expand the Configure Naming option on the Rule Configuration tab to display the configuration options.

When you configure the rule to Use the full URI, the agent creates a business transaction name for the full request URI. There are no dynamic naming options. For example, for the URI http://acmeonline.com/store/coats/men, the agent names the transaction "/store/coats/men".

Use the full URI works well for Grails servlet transactions because the default naming scheme often doesn't differentiate between multiple transactions.
When you use parts of the URI to name transactions, you have several options to configure transaction naming:

- Use a part of the URI
- Use Headers, Cookies, and Other Parts of HTTP Requests
- Use a Custom Expression

**Use a part of the URI**

AppDynamics offers the following options to automatically name URI based transactions based upon the URI:

- Use the first or last URI segments
- Use specific URI segments

For the following URL, the first two segments of the URI don't provide a significant name for the business transaction:

http://example.com/Web/Store/Checkout yields a default transaction name of "Web/Store".

You can use one of the following options to identify more meaningful transaction names:

- Click **Use the first** or **Use the last n** segments to use two contiguous segments at the beginning or end of the URI, where \( n \) is the number of segments. For example, if you identify the checkout transaction using the last two segments of the URI, the agent names the transaction "/Store/Checkout".

- If you need more flexibility, such as using non-contiguous segments in the name, click **Name Transactions dynamically using part of the request**. Select **Use URI segments in Transaction names** option. To specify multiple segments, enter a comma-separated list of segment numbers.

For example, the following URL represents the checkout transaction requested by a customer with ID 1234: http://example.com/Store/cust1234/Checkout. The checkout transaction is the same regardless of the customer, so it makes sense to name the transaction based upon the first and third segments of the URI. AppDynamics names the transaction: "StoreCheckout".

**Use Headers, Cookies, and Other Parts of HTTP Requests**

When you use the **Name Transactions dynamically using part of the request** option, you can choose to identify your URI based transactions using particular parts of the HTTP request. Note that not all agents offer all the naming options below.

Carefully consider your naming configuration choices. If you use a value such as the request originating address and you have many clients accessing your application, you may see the all other traffic business transaction.
To use HTTP parameter values in transaction names, select **Use a parameter value in Transaction names** and enter the **Parameter Name**. For example, consider the following URL: http://example.com/Store/Inventory?category=electronics

### What part of the URI should be used in the Transaction Name?

- [ ] Use the full URI
- [x] Use a part of the URI (for example, if you have dynamic URIs)

Use the first 2 segments of the URL

- [x] Name Transactions dynamically using part of the request

Use **a parameter value** in Transaction names

**Parameter Name**: category

AppDynamics names the transaction to include the category parameter value “/Store/Inventory.electronics”.

- To use a header value in transaction names, select **Use header value in Transaction names** and enter a **Header Name**. For example, consider a site that uses the custom header "Version", AppDynamics names transactions with the header value "/Store/Inventory.v2.5".
- To use a cookie value in transaction names, select **Use a cookie value in Transaction names** and enter the **Cookie Name**. For example, a website tracks a user’s loyalty status in a cookie. Set the Cookie Name to “loyalty”. AppDynamics names transactions for the loyalty cookie value “/Store/Inventory.Status=Gold”
- To use a session attribute value in transaction names, click **Use a session attribute in Transaction names** and enter the **Session Attribute Key**. For example, a website stores a customer’s region in the session property. Set the Session Attribute name to "region". AppDynamics names transactions for the region session attribute value “/Store/Inventory.NorthAmerica”.
- To use the request method in Transaction names, click **Use the request method (GET/POST/PUT) in Transaction names**. For example AppDynamics names the transaction “/Store/Inventory.GET”.
- To use the request host in Transaction names, click **Use the request host in Transaction names**. For example AppDynamics includes the request host id in the transaction name “/Store/Inventory.192.0.2.0”.
- To use the request originating address in Transaction names, click **Use the request originating address in Transaction names**. AppDynamics names transactions for the IP address of the request client “/Store/Inventory.192.0.2.10”.

### Use a Custom Expression

You can use a custom expression on the URI to name:

- Java servlet transactions
- .NET ASP.NET transactions

Select **Name Transactions dynamically using part of the request** and choose **Use a custom expression in Transaction names**. Enter your custom expression getter chain as follows:

For Java, you can use getter chains to access values from the HttpServletRequest object. For example, suppose you want to use a combination of the authenticated user and the value of an HttpServletRequest:

```java
${getParameter(myParam)}-${getUserPrincipal().getName()}
```

The equivalent code expression would evaluate to:
You can create a custom expression on the HttpServletRequest to identify all Servlet based requests (modify global discovery at the transaction naming level) or for a specific set of requests (custom rule).

A custom expression for a servlet transaction can have a combination of any of the following getter methods on request attributes:

<table>
<thead>
<tr>
<th>Getters on Request Attributes</th>
<th>Transaction Identification</th>
</tr>
</thead>
<tbody>
<tr>
<td>getAuthType()</td>
<td>Use this option to monitor secure (or insecure) communications.</td>
</tr>
<tr>
<td>getContextPath()</td>
<td>Identify the user requests based on the portion of the URL.</td>
</tr>
<tr>
<td>getHeader()</td>
<td>Identify the requests based on request headers.</td>
</tr>
<tr>
<td>getMethod()</td>
<td>Identify user requests invoked by a particular method.</td>
</tr>
<tr>
<td>getPathInfo()</td>
<td>Identify user requests based on the extra path information associated with the URL (sent by the client when the request was made).</td>
</tr>
<tr>
<td>getQueryInfo()</td>
<td>Identify the requests based on the query string contained in the request URL after the path.</td>
</tr>
<tr>
<td>getRemoteUser()</td>
<td>Identify the user requests based on the login of the user making this request.</td>
</tr>
<tr>
<td>getRequestedSessionId()</td>
<td>Identify user requests based on the session id specified by the client.</td>
</tr>
<tr>
<td>getUserPrincipal()</td>
<td>Identify user requests based on the current authenticated user.</td>
</tr>
</tbody>
</table>

For .NET, you can use getter chains to access properties and methods for the HttpRequest object for ASP.NET or Microsoft.AspNet.NetCore.Http.Internal.DefaultHttpRequest for ASP.NET Core on the full framework.

- Enclose getter chains inside braces: `{{}}`. The custom expression may contain multiple getter chains.
- Use getter chain syntax.
- Use any HttpRequest request attributes or methods.

For example, consider the URL `http://mystore.example.com/Store/Inventory-Furniture`. The following custom expression uses two getter chains to access properties of the HttpRequest object:

```
${Url.ToString().Split(Char[]/-).[2]}-${UserAgent}
```

- The first getter chain fetches the URL, splits it on the dash character ("-"), and uses the second string in the array.
- The second getter chain fetches the HttpRequest.UserAgent property.
- The literal dash character "-" separates the two getter chains.

The result is the following business transaction name:

Furniture-Mozilla/5.0 (Windows NT 6.3; Trident/7.0; rv:11.0) like Gecko
Message Queue Entry Points

Most frequently an instrumented application calls to a message queue as a remote service or backend. However, in cases when an application uses asynchronous message listeners or message-driven beans as the primary trigger for business processing on an originating tier, AppDynamics can intercept the message listener invocations and track them as business transactions.

AppDynamics discovers the following types of business transaction entry points:

- Java JMS entry points.
- .NET Message Queue entry points

Message queue entry points all have similar naming configuration options for Automatic Transaction Discovery and Custom Match Rules.

Default Naming

By default, the AppDynamics auto-detection naming scheme identifies all message queue transactions using the destination name. Otherwise, if the destination name is not available, AppDynamics uses the listener class name.

Custom Match Rule Options

When you create a JMS or Message Queue custom match rule, the agent names matching transactions for the custom match rule name. See Custom Match Rules.

You have the option to specify the following match criteria as a combination of the following entry point properties:

- The Message Destination as Queue, Topic or Unknown.
- Message Property. You can match against a property’s existence or against a property’s value.
- Message Content.
Business Transaction Discovery Sessions

On this page:

- Launch a Business Transaction Discovery Session
- Edit and Preview Transaction Discovery Rules
- Inspect Uninstrumented Code to Discover Entry Points
- Search for Classes and Methods to Instrument

Works with:

Java .net

Business Transaction Discovery Sessions provide interactive tools to help you create the optimal set of transaction detection rules and discover crucial business transactions.

During a discovery session, you can preview the effects of all the transaction detection rules on a node. This way you can create rules that capture the precise business transactions that impact your business and avoid rules that generate too many transactions.

Within the context of the discovery session, you can:

- Edit Java Agent transaction detection rules and Preview Business Transactions. (Java only)
- Identify entry points from Uninstrumented Code.
- Browse Classes/Methods to identify entry points.

During the business transaction discovery session, you can test out various transaction detection rule configurations without impacting your production environment.

Any business transactions discovered within the context of the discovery session, called transient transactions, do not persist after the session ends. When you end the session you have the opportunity to discard the rule changes or to apply them to your production configuration.

Permissions

You need the following permissions to launch a Business Transaction Discovery Session:

- Configure Transaction Detection
- View Sensitive Data

For information on AppDynamics Role Based Access Control, see Roles and Permissions.

Launch a Business Transaction Discovery Session

1. Navigate to Configuration > Instrumentation and click the Transaction Detection tab.
2. Click Live Preview.
3. Click Start Discovery Session to start a Business Transaction Discovery session.
4. Select the node to act as the source of the live data stream.

Nodes are organized by tier. You can see which nodes have active agents and if there is a discovery session already running on a node.

The Transaction Discovery page displays information about the discovery session lets you navigate among the available interactive transaction discovery tools:
When you initiate a transaction discovery session on a Java node, the Rules pane displays the active transaction discovery rules that apply to the node. The right-hand pane lets you preview business transactions or use the available interactive tools: Uninstrumented Code detection and the Classes/Methods Browser.

When you launch a discovery session on a .NET node, you see the available tools: Uninstrumented Code detection and the Classes/Methods Browser.

During a discovery session, you can use other features of AppDynamics while the discovery process continues in the background. This is useful for busy environments with many active transactions. You can give AppDynamics a few minutes to complete the discovery. When you are ready to return to the Transaction Discovery page click Configuration > Instrumentation and click the Live Preview button to resume your discovery session.

**Edit and Preview Transaction Discovery Rules**

You can add, edit, or delete any type of transaction detection rule for the Java Agent in the Rules pane. Click Apply to update the rule within the context of the discovery session.

To see the effects of your changes, click Preview Business Transactions. As the app agent discovers business transactions based on the rule configuration, the transactions show up in a list in the right-hand pane:

You can select a transaction and click Capture Snapshots to collect transaction snapshots.

The Capture Snapshots feature is only available for Servlets.

Click a specific snapshot to display the payload data for the business transaction in the Snapshot Details:
When you create custom match rules for servlets or for POJOs you can start a Live Preview session limited to the specific rule.

Inspect Uninstrumented Code to Discover Entry Points

When the app agent doesn’t automatically discover an entry point for a framework in your application, you can use Uninstrumented Code detection to find entry points for the framework. After you identify an entry point you want to track as a business transaction, you can create a custom match rule.

Uninstrumented Code detection identifies the supported exit calls for an app agent on the node even when the entry point is undetected. In the following example the Java agent discovers some uninstrumented JDBC exit points:

You can select an exit point and click View Stack Trace to display the call stack for the exit point. Select any call in the stack trace and click Monitor to show interactive data for that specific call. When the agent returns the monitoring result, you can inspect the values for a specific method invocation including the Invoked Object, Return Value, Parameters, and more.

The example below shows the upstream stack trace for the com.mysql.jdbc.PreparedStatement:executeQuery JDBC exit call:
When you identify the method you want to instrument, you can click to add a POJO or POCO. The Controller UI automatically populates the Rule Configuration in the Add Rule editor with the class and method you want to instrument. Complete the custom match rule for POJO or POCO as normal.

Search for Classes and Methods to Instrument

If you are familiar with your application code, you can use the Classes/Methods browser to search the running code for specific classes and methods to instrument. The example below illustrates how a search for jetty.server in a sample application returns both instrumented and uninstrumented classes and methods:
When you identify a method you want to instrument, right-click the method to open the Add Rule editor. AppDynamics automatically populates the class and method configuration based upon your search criteria from the Class/Method browser. You can right-click a class to instrument it, but you must add the method before you can save the POJO or POCO. Complete the custom match rule for the POJO or POCO as normal.
Backend Detection Rules

On this page:

- Default Automatic Backend Discovery
- Modify Automatic Discovery
- Configurable Properties andNaming
- Aggregate or Split Backends
- Add Custom Backend Discovery Rules
- All Other Traffic Backends
- Backend Registration Limits
- Manage Backend Discovery
- Delete Unnecessary Backends
- Organize Backends in Flow Maps

Related pages:

- Java Backend Detection
- .NET Backend Detection

In AppDynamics, databases and remote services (and in fact, any detected out of process components that are involved in Business Transaction processing) are collectively known as backends.

AppDynamics discovers backends from exit point instrumentation placed in the application code. An exit point is the precise location in the code where an outbound call is made from an instrumented node. AppDynamics automatically discovers a wide range of backends.

Default Automatic Backend Discovery

A discoverable backend is identified by its type and associated properties. The precise set of properties used to identify any given backend is dependent on its type. The type itself is defined in the built-in backend discovery rules for backend types supported out of the box. For exit calls to backends that are not instrumented with APM agents, AppDynamics uses the backend properties to identify and name the backend. Where a backend call is actually processed by a downstream tier also instrumented with AppDynamics, ensure that any given backend that is identified will always result in calls that will be processed by just one downstream tier. This means, for example, that if tier A makes an HTTP call to an http router on localhost:4040 which will forward the call to tier B or tier C depending on the URL, custom backend naming rules will be required to include enough segments of the request URL - as well as the host and port - such that requests destined for tier B will be associated with a backend with a different set of backend properties than those associated with tier C.

The Automatic Backend Discovery list in the Configuration > Instrumentation > Backend Detection tab shows the configurable backend discovery rules.

The automatic discovery rules vary according to the type of backend being identified, but generally include settings for enabling automatic discovery of the backend type, enabling correlation, and properties used to identify and name the backend. AppDynamics uses transaction correlation to track request processing across distributed tiers. Certain types of backends are automatically discovered as high volume exit points (sometimes called turbo exit points). These backend types include cache servers, EhCache, Danga Memcached, Memcached, and Oracle Coherence. For more about high volume exit points, see Exit Point Detection Rules.

When an exit point is identified for the first time, the exit call results in a backend discovery event.

Permissions

To modify backend detection, you need the Configure Backend Detection permission.

Modify Automatic Discovery

If you know that the application flow map should be showing a specific backend and you are not seeing it, you can modify the configuration to detect backends. You can:

- Edit the default backend discovery rules.
- Create new custom backend discovery rules.
- Create new custom exit points to provide visibility for backend types that are not automatically discovered.
- Disable automatic discovery of backends that are not of interest.
• Disable correlation for the backend.

In many cases, you can achieve the level of customization that you need by editing the default rules rather than creating new rules.

To edit a rule, in the Backend Detection tab, select the application and click the **Edit Automatic Discovery** button.

<table>
<thead>
<tr>
<th>Edit Backend Discovery Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Summary</td>
</tr>
</tbody>
</table>

**Backend Naming Configurations**

Configure how Backends will be named, what properties to use in the name, and how to use those properties. All of the properties used will be concatenated together in the Backend name.

- **Server Pool**
  - Use it as is

- **Vendor**
  - Use it as is

For certain backend types—particularly those named by URL—the name of the discovered backend instances include a port number. If the app agent is unable to discover the port number for some reason, the port number is displayed in the UI as ”-1”. This can happen, for example, when a port is not a public port or the API for discovering the port is not available on the application environment (often the case for an RMI backend type). Despite having an unknown port number, the backends are nonetheless uniquely identified.

**Configurable Properties and Naming**

For many of the automatically discovered backend types, AppDynamics uses a set of configurable properties to identify and name the backend. For example, the properties for a JMS messaging queue are something like this:

- **DestinationType** (for example, QUEUE)
- **Destination** (for example, OrderQueue)
- **Vendor** (for example, Active MQ)

The following informational popup shows how the values are used to identify a discovered backend. In the following example, callout 1 shows the value that corresponds to the destination type, callout 2 to the vendor name, and callout 3 to the destination queue name:
Aggregate or Split Backends

You can change the default configuration for the backend types shown in the UI to aggregate or split the backends. For example, you could disable the Vendor property for JMS backend detection to aggregate (join) all JMS backends with the same destination and destination type. The metrics for all JMS backends with the same destination and destination type would be aggregated as one backend.

For properties that you want to use, you can also configure how AppDynamics uses the property. For example, if the URL property is used to identify a backend and your URLs include a file name, the default configuration results in a separate backend for every file. To reduce the number of backends identified using the URL property, configure which parts of the URL to use. For example, run a regular expression on the URL to discard the file name or any other parts of the URL that you do not want used for identification.

Add Custom Backend Discovery Rules

AppDynamics provides the additional flexibility to create new custom discovery rules for the automatically discovered backend types.

To create a custom discovery rule for an automatically discovered backend type, select the backend type in the Automatic Backend Discovery list. Click Add (the + icon) to create a new rule. Specify the following:

- **Name** for the custom rule.
- **Enabled** to enable or disable the rule.
- **Correlation Enabled** to specify whether or not to send business transaction correlation data to the backend.
- **Priority** for this custom rule compared to other custom rules for this backend type. The higher the number, the higher the priority. A value of 0 (zero) indicates that the default rule should be used.
- **Match Conditions** used to identify which backends are subject to the custom naming rules. AppDynamics applies the default rules to backends that do not meet all the defined match conditions.
- **Backend Naming Configuration** used to name the backends matching the match conditions. The naming configuration must include the property used by the match condition.

See specific exit points for examples.

All Other Traffic Backends

AppDynamics limits the number of backend systems that can be registered and tracked. Once the limit is reached (databases and remote services combined), additional backend are put in the "All other traffic for <type>" category.
The "type" is the backend type such as HTTP or Web Service. HTTP calls and JMS queues are the most common types that may reach the limit with the default configuration rules.

On a flow map, the category appears as follows:

![HTTP All other traffic for HTTP](image)

Metrics for the backend calls grouped in the "All other traffic" category are aggregated. If the group includes backend instances that you want to be individually tracked, you can manage backend detection, as described in Manage Backend Discovery.

### Backend Registration Limits

The limits on the number of backends that can be registered:

- The Controller enforces a limit of 1000 backend registrations in the business application for each type of backend (that is, 1000 HTTP backends, 1000 JMS backends, and so on). This setting is controlled by the `backend.registration.limit` Controller Setting. This limit applies whether the backends are resolved to a tier or unresolved.
- Agent apply a limit of 300 unresolved backends per application.

When the backend count exceeds the limit, the "All Other Traffic" category is automatically created. The logs indicate that a limit has been reached by a log entry such as the following:

```
```

### Manage Backend Discovery

After reaching the backend registration limit, you should make sure that the systems being tracked as first-class registered backends are the ones that are important to you relative to those in the "All Other Traffic" category.

Also make sure that application environmental conditions aren't causing an excess number of backends to be registered. These conditions may include:

- JMS queues that use the session ID in the destination. This causes each session to be identified as a separate backend.
- Calls to message queues hosted on the same server. In this case, you might not be interested in seeing each queue separately, and instead, want to aggregate everything for the same host and port to the same backend.
- Dynamic generation of IP addresses, such as when you host a service on Amazon Elastic Compute Cloud (Amazon EC2).
- Different JDBC drivers connecting to the same database may cause many backends. This can happen when there are slight variations in the driver metadata that AppDynamics uses to identify the database. For example, if two different JDBC drivers are used to access the same database, something as simple as a different spelling or format of the database name (ORACLE, oracle, or ORACL) can generate multiple database names when they are the actually the same physical database.

To manage backend registrations, you can change the configuration of the backend detection rules. For example, if a property causes excessive unique identification, consider removing the property or modifying its use in the detection and naming rule. If you created custom exit points, consider refining or removing them.

### Delete Unnecessary Backends
Some backends may no longer have traffic. You can delete unused backends from the Remote Services List, any Remote Service dashboard, the Databases List, and any Database dashboard. If the backend has new traffic in the future, the app agent rediscovers it and reregisters it with the Controller.

You can also configure the Controller to automatically remove stale backends. See Stale Remote Service Removal.

Organize Backends in Flow Maps

Once you have configured detection for all the backends you require, you may need to organize the way they appear in the UI. See Group Remote Services on Flow Maps.
Exit Point Detection Rules

On this page:
- Configuration Notes
- Adding a Custom Exit Point
- High Volume Exit Points
- Splitting and Grouping Exit Points

Related pages:
- Hierarchical Configuration Inheritance
- Custom Exit Points for Java

Custom exit points enable you to identify backend types that are not automatically detected. For example, you can define a custom exit point to monitor code calls to a file system read method.

After you have defined a custom exit point, the backend appears on the flow map with the type-associated icon you selected when you configured the custom exit point.

Configuration Notes

You configure exit point detection in the Configuration > Instrumentation page. Select the Backend Detection tab and choose the application or tier on which to configure an exit point and the application type of the exit point, Java, .NET, and so on. From there you can add new exit points or modify existing ones.

The specific configuration options available differ based on the application type, but in general, you identify the exit point and specify the mechanism for it to be detected, such as a class and method name. These additional notes apply to exit point configuration:

- If the method you are using to identify the exit point is overloaded, you need to add the parameters that identify the signature for this form of the method.
- Match conditions let you restrict the method invocations for which you want AppDynamics to collect metrics based on runtime values tested by the condition. Match conditions can test parameter values or any value accessible by getter chain from the invoked object.
- Optionally, you can split an exit point based on a method parameter, the return value, or the invoked object.
- You can configure custom metrics and transaction snapshot data to collect for the backend.
- If you configure a custom backend for a method that encapsulates an automatically detected backend, such as an HTTP client call, you may lose correlation from the automatically detected exit call.
- Splitting a backend on return value may break correlation.

You are limited to 50 characters for the exit point value when adding a custom exit point or splitting/grouping exit points.

Adding a Custom Exit Point

When adding a custom exit point, you specify the class and method that identify an external call to the exit point. You can refine the call by specifying a return value, parameter, or getter chain match value.

The type you choose determines the icon that appears in the flow map for the custom exit point. If the type is not listed, select Use Custom and enter a string in the Type field to identify the exit point calls in the UI.

Note that you can also split the exit point based on these values. If you add a split condition for an exit point, it means that any exit point that matches the overall match condition for the custom exit point is further evaluated against the split condition. If its call matches the split condition, it is given the more specific name you configure in the split exit point configuration.

High Volume Exit Points

The Is High Volume option for custom exit points lets you create an optimized exit point configuration for backend systems with high-performance requirements, such as caching servers or in-memory databases.

A "high volume" exit point keeps the overhead associated with the exit point to a negligible amount by bypassing certain processing
operations on calls to this exit point. As a result, transaction correlation and error detail reporting are not available for the exit point, nor are match conditions in the exit point configuration.

With a high volume exit point, you do get the number of calls, number of errors, and average response time for calls.

Splitting and Grouping Exit Points

By splitting an exit point, you can use dynamic application values to identify an exit point. Specifically, you can use the value of either a method parameter, the method return value, or a value returned by a specified getter chain on the invoked object of the identified method. The configuration settings you use to configure exit point splitting are the same as used for Data Collectors. See Data Collectors for information on how to use the configuration UI.

A simple example of how you would use exit point splitting is for a cache exit point. In this example, say you configure splitting for an exit point that identifies a method for writing to the cache. The object on which the method for writing to the cache offers a method to get the current cache name, the `getCacheName()` method. By configuring a split on this method (as a getter chain on the invoked object), you can have the exit point named with the name of the cache node, as dynamically determined:

Splitting also lets you group exit points. Given an exit point for `NamedCache.entrySet()`, as shown in the example above, suppose we created another exit point. This one uses the `getAll()` method on the same class, `NamedCache`. It also has a split configuration that uses the `getCacheName()` method of the invoked object.

If the `getCacheName()` getter chain points to the same cache name when the `getAll()` and the `entrySet()` methods are invoked, they will be shown as calls to the same backend.
Java Backend Detection

The AppDynamics Java Agent automatically discovers common types of remote services as backend systems based on the exit point associated with the backend. If AppDynamics does not automatically discover a backend system used in your environment, check whether it’s an automatically discovered type (based on the list below) and compare the default exit points for the type to that used in your application environment.

Each automatically discovered backend type has a default discovery rule. Many backend types have a set of backend naming properties. Where individual properties are configurable, you can configure the property as follows:

- Use the property as is.
- Use segments of the property.
- Run a regular expression on the property. See Using Regular Expressions.
- Execute methods on the property. See Using Getter Chains.

For instructions to revise backend discovery rules, see Backend Detection Rules.

Amazon Simple Notification Service Backends

AppDynamics detects exit calls to the Amazon SNS messaging service using the Amazon SNS Client: AmazonSNSClient.publish.

Amazon SNS Naming Properties

You can enable or disable the use of the following properties for Amazon S3 backend identification:

<table>
<thead>
<tr>
<th>Configurable Properties</th>
<th>Default Automatic Amazon SNS Backend Discovery/Naming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vendor</td>
<td>Yes</td>
</tr>
<tr>
<td>Topic ARN</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Amazon Simple Queue Service Backends

AppDynamics can detect the following types of Amazon Web Services SQS message queue actions:

- Basic send/receive
Batched send/receive
Asynchronous send/receive

Correlating SQS traffic requires you to configure the continuation entry point for the SQS message.

Amazon Simple Storage Service Backends

AppDynamics automatically detects the following exit points for Amazon Web Services S3 backends:

- `AmazonS3Client.deleteObject`
- `AmazonS3Client.getObject`
- `AmazonS3Client.getObjectMetadata`
- `AmazonS3Client.listObjects`
- `AmazonS3Client.putObject`

Amazon S3 Naming Properties

You can enable or disable the use of the following properties for Amazon S3 backend identification:

<table>
<thead>
<tr>
<th>Configurable Properties</th>
<th>Default Automatic Amazon S3 Discovery/Naming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bucket Name</td>
<td>Yes</td>
</tr>
<tr>
<td>Vendor</td>
<td>Yes</td>
</tr>
<tr>
<td>Object Key</td>
<td>No</td>
</tr>
</tbody>
</table>

Amazon Web Services Backends

AppDynamics detects the following exit calls to the following Amazon Web Services:

- DynamoDB requests using the `AmazonDynamoDBClient` create, read, update, and delete (CRUD) methods using the Amazon low-level DynamoDB API. The agent discovers the Document API calls as the underlying low-level API call.

Amazon Web Service Naming Properties

For Amazon Web Services, you can enable or disable the use of the following naming properties for backend identification:

<table>
<thead>
<tr>
<th>Configurable Properties</th>
<th>Default Automatic Amazon Web Service Backend Discovery/Naming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vendor</td>
<td>Yes</td>
</tr>
<tr>
<td>Service</td>
<td>Yes</td>
</tr>
<tr>
<td>Endpoint</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Apache Cassandra CQL Backends

By default, AppDynamics automatically detects and identifies exit calls to Apache Cassandra using Thrift and DataStax drivers.

Exceptions returned from Cassandra backends show up as separate exit calls. For the DataStax 1.0 driver, the Java Agent detects an "unknown" Cassandra backend in order to capture the error details.

Cassandra Backend Naming Properties

You can enable or disable the use of the properties below for Cassandra backend identification:

- For DataStax clients, AppDynamics does not recommend using the keyspace property. Due to driver limitations, the keyspace property is only available when `Statement.setKeyspace` is called explicitly in Cassandra. This can lead
Apache Kafka Backends

By default, the Java Agent detects publish activity to Apache Kafka as an exit point from a Java node. You can configure the Java Agent to detect Kafka consumer activity as an entry point, see Apache Kafka Consumer Backends.

Kafka Backend Naming Properties

You can enable or disable the use of the following properties for Kafka backend identification:

<table>
<thead>
<tr>
<th>Configurable Properties</th>
<th>Default Automatic IBM MQ Backend Discovery/Naming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vendor</td>
<td>Yes</td>
</tr>
<tr>
<td>Topic Name</td>
<td>Yes</td>
</tr>
<tr>
<td>Broker URL</td>
<td>No</td>
</tr>
</tbody>
</table>

Default Kafka JMX Metrics

There are lots of JMX metrics that are exposed by Kafka Producer and Consumer. We have selected a few of the metrics that we think could be useful for customers and created OOTB rules for them. PFB the list of metrics supported by Producer and Consumer respectively. In case you are interested in any of the other metric for which OOTB rule is not available you can go to MBean browser, select the metric and create a rule out of it.

HTTP Backends

HTTP exit point activity includes all HTTP calls done outside of a web service call. Web service calls are not considered an HTTP exit point. By default, the Java Agent names the HTTP backend for the Host and Port properties. For instance, "myHTTPHost:5000".

HTTP Backend Naming Properties

You can enable or disable the use of the following properties for HTTP backend identification:

<table>
<thead>
<tr>
<th>Configurable Properties</th>
<th>Default Automatic HTTP Backend Discovery/Naming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host</td>
<td>Yes</td>
</tr>
<tr>
<td>Port</td>
<td>Yes</td>
</tr>
<tr>
<td>URL</td>
<td>No</td>
</tr>
<tr>
<td>Query String</td>
<td>No</td>
</tr>
</tbody>
</table>

For examples of common HTTP backend configurations, see HTTP Backend Detection.

For sample JDBC backend configurations, see Example JDBC Backend Configuration.
IBM Websphere Message Queue Backends

IBM MQ, also known as IBM WebSphere MQ and IBM MQSeries, is IBM’s message-oriented middleware similar to JMS. Several additional properties are configurable, such as host and port. This is useful where you have lots of queues and you want to monitor them based on a subset of the properties.

IBM MQ Backend Naming Properties

You can enable or disable the use of the following properties for IBM MQ backend identification:

<table>
<thead>
<tr>
<th>Configurable Properties</th>
<th>Default Automatic IBM MQ Backend Discovery/Naming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Destination</td>
<td>Yes</td>
</tr>
<tr>
<td>Destination Type</td>
<td>Yes</td>
</tr>
<tr>
<td>Host</td>
<td>Yes</td>
</tr>
<tr>
<td>Port</td>
<td>Yes</td>
</tr>
<tr>
<td>Major Version</td>
<td>Yes</td>
</tr>
<tr>
<td>Vendor</td>
<td>Yes</td>
</tr>
</tbody>
</table>

For examples of common message queue configuration scenarios, see Example Message Queue Backend Configuration.

Java Message Service Backends

JMS backend activity includes all JMS message send and publish activity. By default, AppDynamics identifies JMS back ends using the properties of the JMS server such as: vendor, destination name, and destination type. The default configuration uses all three properties of the JMS queue.

JMS Backend Naming Properties

You can enable or disable the use of the following properties for JMS backend identification:

<table>
<thead>
<tr>
<th>Configurable Properties</th>
<th>Default Automatic JMS Backend Discovery/Naming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Destination</td>
<td>Yes</td>
</tr>
<tr>
<td>Destination Type</td>
<td>Yes</td>
</tr>
<tr>
<td>Vendor</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Transaction names are derived from the enabled properties, for example, “ActiveMQ-OrderQueue”.

For examples of common message queue configuration scenarios, see Example Message Queue Backend Configuration.

JDBC Backends

JDBC backend activity consists of all JDBC calls including inserts, queries, updates, getting connections from connection pools, and so on.

Expand to show exit calls the agent captures for the Oracle JDBC driver as an example...

<table>
<thead>
<tr>
<th>Class</th>
<th>Method</th>
</tr>
</thead>
</table>
| oracle.jdbc.driver.PhysicalConnection | * prepareStatement  
|                           | * prepareCall  
|                           | * commit  
|                           | * rollback |
The Java Agent measures the response time for a database call as the round trip time it takes for a JDBC call to return. The response time for a database call includes network round trip time + database connection time + SQL query time or any other time spent in the database.

By default, the agent identifies JDBC backends using the following logical properties of the database:

- URL
- Host name
- Port number
- Database schema
- Version
- Vendor

If a database backend has the same hostname, port number, and database type as a database server already configured in a database Collector, the database backend is automatically matched with the Collector, and drill-downs from the Application Flow Map, Tier Flow Map or Node Flow Map to Database Monitoring are enabled.

**JDBC Backend Naming Properties**

You can enable or disable the use of the following properties for JDBC backend identification:

<table>
<thead>
<tr>
<th>Configurable Properties</th>
<th>Default Automatic JDBC Backend Discovery/Naming</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>URL</td>
<td>Yes</td>
<td>JDBC URL provided to the driver</td>
</tr>
<tr>
<td>Host</td>
<td>Yes</td>
<td>Database host</td>
</tr>
<tr>
<td>Port</td>
<td>Yes</td>
<td>Database port</td>
</tr>
<tr>
<td>Database</td>
<td>Yes</td>
<td>Database schema</td>
</tr>
<tr>
<td>Version</td>
<td>Yes</td>
<td>Database version as reported by JDBC driver</td>
</tr>
<tr>
<td>Vendor</td>
<td>Yes</td>
<td>Database vendor as reported by JDBC driver</td>
</tr>
</tbody>
</table>

**Jolt Backends**

AppDynamics detects exit calls to Tuxedo services from the JoltRemoteService class in the Jolt Class Library.

**Jolt Naming Properties**

You can enable or disable the use of the following naming properties for Jolt backend identification:

<table>
<thead>
<tr>
<th>Configurable Properties</th>
<th>Default Automatic Jolt Backend Discovery/Naming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host</td>
<td>Yes</td>
</tr>
<tr>
<td>Port</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Service | Yes

### MongoDB Backends

By default, the Java Agent detects exit calls to MongoDB using the Mongo Java Driver 3.1 over the MongoDB Wire Protocol. The agent detects the exit point on the Wire Protocol methods and excludes the methods between the external API and the Wire Protocol.

#### MongoDB Backend Naming Properties

<table>
<thead>
<tr>
<th>Configurable Properties</th>
<th>Default Automatic MongoDB Backend Discovery/Naming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host</td>
<td>Yes</td>
</tr>
<tr>
<td>Port</td>
<td>Yes</td>
</tr>
<tr>
<td>Database</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### RabbitMQ Backends

RabbitMQ is open source, commercially supported, messaging middleware that runs on many different operating systems. The Java Agent discovers exit points using the RabbitMQ Java API, amqp-client.jar in most distributions. By default, RabbitMQ backends are identified by Host, Port, Routing Key, and Exchange. For instance: "amqp://guest@127.0.0.1:5672/exchange/task_queue".

#### RabbitMQ Backend Naming Properties

You can enable or disable the use of the following properties for RabbitMQ backend identification:

<table>
<thead>
<tr>
<th>Configurable Properties</th>
<th>Default Automatic RabbitMQ Backend Discovery/Naming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host</td>
<td>Yes</td>
</tr>
<tr>
<td>Port</td>
<td>Yes</td>
</tr>
<tr>
<td>Routing Key</td>
<td>Yes</td>
</tr>
<tr>
<td>Exchange</td>
<td>Yes</td>
</tr>
</tbody>
</table>

For examples of common message queue configuration scenarios, see [Example Message Queue Backend Configuration](#).

### RMI Backends

The Java Agent automatically discovers backends called using the standard Java RMI API. For a list of supported RMI frameworks, see 'RPC/Web Services API/HTTP Client Support' on [Java Supported Environments](#).

#### RMI Backend Naming Property

The JAVA Agent names the RMI backend for the URL. You can configure how the agent uses the URL to name the RMI backend.

### Thrift Backends

By default, AppDynamics automatically detects and identifies Apache Thrift exit points (backends). See [Apache Thrift](#) for details.

#### Thrift Backend Naming Properties

You can enable or disable the use of the following properties for Thrift exit backend identification:

<table>
<thead>
<tr>
<th>Configurable Properties</th>
<th>Default Automatic Thrift Backend Discovery/Naming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Web Services Backends

Web service backend activity includes all web service invocations. Web service backends are identified using the web service name.

**Web Services Backend Naming Properties**

<table>
<thead>
<tr>
<th>Configurable Properties</th>
<th>Default Automatic Web Service Backend Discovery/Naming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service</td>
<td>Yes</td>
</tr>
<tr>
<td>URL</td>
<td>No</td>
</tr>
<tr>
<td>Operation</td>
<td>No</td>
</tr>
<tr>
<td>Soap Action</td>
<td>No</td>
</tr>
<tr>
<td>Vendor</td>
<td>No</td>
</tr>
</tbody>
</table>
Custom Exit Points for Java

On this page:

- Coherence Exit Points
- EhCache Exit Points
- LDAP Exit Points
- Mail Exit Points
- Memcached Exit Points
- MongoDB Exit Points
- SAP Exit Points

Related pages:

- Exit Point Detection Rules

This topic gives you settings you can use to create custom exit point configurations for specific backends in Java environments.

In general, custom exit rules match calls to custom backends by class and method name. In some cases, you need to configure splitting as well. The following tables give you the class and method names to use for the various types.

Enter the configuration settings in the corresponding fields in the custom exit point UI. The following screenshot illustrates a sample configuration for an EhCache custom match rule:
For more about creating custom exit points, see create a custom exit point.

Oracle Coherence, EhCache, and Memcached backends are detected automatically as high volume exit points. You can create custom exit points for the backends as described here for particular backend splitting and naming results.

### Coherence Exit Points

<table>
<thead>
<tr>
<th>Name of the Exit Point</th>
<th>Type</th>
<th>Method Name</th>
<th>Class Match Criteria</th>
<th>Class/Interface/Superclass/Annotation</th>
<th>Method Name</th>
<th>Splitting Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coherence.Put</td>
<td>Cache</td>
<td></td>
<td>that implements an interface which</td>
<td>com.tangosol.net.NamedCache</td>
<td>put</td>
<td>getCacheName()</td>
</tr>
<tr>
<td>Coherence.PutAll</td>
<td>Cache</td>
<td></td>
<td>that implements an interface which</td>
<td>com.tangosol.net.NamedCache</td>
<td>putAll</td>
<td>getCacheName()</td>
</tr>
<tr>
<td>Coherence.EntrySet</td>
<td>Cache</td>
<td></td>
<td>that implements an interface which</td>
<td>com.tangosol.net.NamedCache</td>
<td>entrySet</td>
<td>getCacheName()</td>
</tr>
</tbody>
</table>
Coherence.KeySet | Cache | that implements an interface which | com.tangosol.net.NamedCache | keySet | getCacheName()
Coherence.Get | Cache | that implements an interface which | com.tangosol.net.NamedCache | get | getCacheName()
Coherence.Remove | Cache | that implements an interface which | com.tangosol.net.NamedCache | remove | getCacheName()

**EhCache Exit Points**

<table>
<thead>
<tr>
<th>Name of the Exit Point</th>
<th>Type</th>
<th>Class Match Criteria</th>
<th>Class/Interface/Superclass/Annotation Name</th>
<th>Method Name</th>
<th>Splitting Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>EhCache.Get</td>
<td>Cache</td>
<td>With a class name that</td>
<td>net.sf.ehcache.Cache</td>
<td>get</td>
<td>getname()</td>
</tr>
<tr>
<td>EhCache.Put</td>
<td>Cache</td>
<td>With a class name that</td>
<td>net.sf.ehcache.Cache</td>
<td>put</td>
<td>getname()</td>
</tr>
<tr>
<td>EhCache.PutIfAbsent</td>
<td>Cache</td>
<td>With a class name that</td>
<td>net.sf.ehcache.Cache</td>
<td>putIfAbsent</td>
<td>getname()</td>
</tr>
<tr>
<td>EhCache.PutQuiet</td>
<td>Cache</td>
<td>With a class name that</td>
<td>net.sf.ehcache.Cache</td>
<td>putQuiet</td>
<td>getname()</td>
</tr>
<tr>
<td>EhCache.Remove</td>
<td>Cache</td>
<td>With a class name that</td>
<td>net.sf.ehcache.Cache</td>
<td>remove</td>
<td>getname()</td>
</tr>
<tr>
<td>EhCache.RemoveAll</td>
<td>Cache</td>
<td>With a class name that</td>
<td>net.sf.ehcache.Cache</td>
<td>removeAll</td>
<td>getname()</td>
</tr>
<tr>
<td>EhCache.RemoveQuiet</td>
<td>Cache</td>
<td>With a class name that</td>
<td>net.sf.ehcache.Cache</td>
<td>removeQuiet</td>
<td>getname()</td>
</tr>
<tr>
<td>EhCache.Replace</td>
<td>Cache</td>
<td>With a class name that</td>
<td>net.sf.ehcache.Cache</td>
<td>replace</td>
<td>getname()</td>
</tr>
</tbody>
</table>

**LDAP Exit Points**

<table>
<thead>
<tr>
<th>Name of the Exit Point</th>
<th>Type</th>
<th>Class Match Criteria</th>
<th>Class/Interface/Superclass/Annotation Name</th>
<th>Method Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDAPExitPoint.Bind</td>
<td>LDAP</td>
<td>With a class name that</td>
<td>javax.naming.directory.InitialDirContext</td>
<td>bind</td>
</tr>
<tr>
<td>LDAPExitPoint.Rebind</td>
<td>LDAP</td>
<td>With a class name that</td>
<td>javax.naming.directory.InitialDirContext</td>
<td>rebind</td>
</tr>
<tr>
<td>LDAPExitPoint.Search</td>
<td>LDAP</td>
<td>With a class name that</td>
<td>javax.naming.directory.InitialDirContext</td>
<td>search</td>
</tr>
<tr>
<td>LDAPExitPoint.ModifyAttributes</td>
<td>LDAP</td>
<td>With a class name that</td>
<td>javax.naming.directory.InitialDirContext</td>
<td>modifyAttributes</td>
</tr>
<tr>
<td>LDAPExitPoint.GetNextBatch</td>
<td>LDAP</td>
<td>With a class name that</td>
<td>com.sun.jndi.ldap.LdapNamingEnumeration</td>
<td>getNextBatch</td>
</tr>
<tr>
<td>LDAPExitPoint.NextAux</td>
<td>LDAP</td>
<td>With a class name that</td>
<td>com.sun.jndi.ldap.LdapNamingEnumeration</td>
<td>nextAux</td>
</tr>
<tr>
<td>LDAPExitPoint.CreatePooledConnection</td>
<td>LDAP</td>
<td>With a class name that</td>
<td>com.sun.jndi.ldap.LdapClientFactory</td>
<td>createPooledConnection</td>
</tr>
<tr>
<td>LDAPExitPoint.Search</td>
<td>LDAP</td>
<td>With a class name that</td>
<td>com.sun.jndi.ldap.LdapClientFactory</td>
<td>search</td>
</tr>
<tr>
<td>LDAPExitPoint.Modify</td>
<td>LDAP</td>
<td>With a class name that</td>
<td>com.sun.jndi.ldap.LdapClientFactory</td>
<td>modify</td>
</tr>
</tbody>
</table>

**Mail Exit Points**

<table>
<thead>
<tr>
<th>Name of the Exit Point</th>
<th>Type</th>
<th>Class Match Criteria</th>
<th>Class/Interface/Superclass/Annotation Name</th>
<th>Method Name</th>
</tr>
</thead>
</table>
### Memcached Exit Points

<table>
<thead>
<tr>
<th>Name of the Exit Point</th>
<th>Type</th>
<th>Class Match Criteria</th>
<th>Class/Interface/Superclass/Annotation Name</th>
<th>Method Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memcached.Add</td>
<td>Cache</td>
<td>With a class name that</td>
<td>net.spy.memcached.MemcachedClient</td>
<td>add</td>
</tr>
<tr>
<td>Memcached.Set</td>
<td>Cache</td>
<td>With a class name that</td>
<td>net.spy.memcached.MemcachedClient</td>
<td>set</td>
</tr>
<tr>
<td>Memcached.Replace</td>
<td>Cache</td>
<td>With a class name that</td>
<td>net.spy.memcached.MemcachedClient</td>
<td>replace</td>
</tr>
<tr>
<td>Memcached.CompareAndSwap</td>
<td>Cache</td>
<td>With a class name that</td>
<td>net.spy.memcached.MemcachedClient</td>
<td>cas</td>
</tr>
<tr>
<td>Memcached.Get</td>
<td>Cache</td>
<td>With a class name that</td>
<td>net.spy.memcached.MemcachedClient</td>
<td>get</td>
</tr>
<tr>
<td>Memcached.Remove</td>
<td>Cache</td>
<td>With a class name that</td>
<td>net.spy.memcached.MemcachedClient</td>
<td>remove</td>
</tr>
</tbody>
</table>

### MongoDB Exit Points

<table>
<thead>
<tr>
<th>Name of Exit Point</th>
<th>Type</th>
<th>Class Match Criteria</th>
<th>Class/Interface/Superclass/Annotation Name</th>
<th>Method Name</th>
<th>Collect Data From</th>
<th>Getter Chain Operation on Invoked Object</th>
<th>Snapshot Data Operation on Invoked Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>MongoDB.Insert</td>
<td>JDBC</td>
<td>With a class name that</td>
<td>com.mongodb.DBCollection</td>
<td>insert</td>
<td>Invoked Object</td>
<td>getDB().getName()</td>
<td>Parameter_0.toString()</td>
</tr>
<tr>
<td>MongoDB.Find</td>
<td>JDBC</td>
<td>With a class name that</td>
<td>com.mongodb.DBCollection</td>
<td>find</td>
<td>Invoked Object</td>
<td>getDB().getName()</td>
<td>Parameter_0.toString()</td>
</tr>
<tr>
<td>MongoDB.Update</td>
<td>JDBC</td>
<td>With a class name that</td>
<td>com.mongodb.DBCollection</td>
<td>update</td>
<td>Invoked Object</td>
<td>getDB().getName()</td>
<td>Parameter_0.toString()</td>
</tr>
<tr>
<td>MongoDB.Remove</td>
<td>JDBC</td>
<td>With a class name that</td>
<td>com.mongodb.DBCollection</td>
<td>remove</td>
<td>Invoked Object</td>
<td>getDB().getName()</td>
<td>Parameter_0.toString()</td>
</tr>
<tr>
<td>MongoDB.Apply</td>
<td>JDBC</td>
<td>With a class name that</td>
<td>com.mongodb.DBCollection</td>
<td>apply</td>
<td>Invoked Object</td>
<td>getDB().getName()</td>
<td>Parameter_0.toString()</td>
</tr>
</tbody>
</table>

**Avoiding high overhead**  
AppDynamics recommends that you avoid instrumenting highly-used methods, such as the find() method, because the activity around these methods can cause an undesired amount of overhead on the system.

**Sample MongoDB Exit Point Configuration**
## SAP Exit Points

<table>
<thead>
<tr>
<th>Name of the Exit Point</th>
<th>Type</th>
<th>Class Match Criteria</th>
<th>Class/Interface/Superclass/Annotation Name</th>
<th>Method Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAP.Execute</td>
<td>SAP</td>
<td>With a class name that com.sap.mw.jco.rfc.MiddlewareRFC$Client</td>
<td>execute</td>
<td></td>
</tr>
<tr>
<td>SAP.Connect</td>
<td>SAP</td>
<td>With a class name that com.sap.mw.jco.rfc.MiddlewareRFC$Client</td>
<td>connect</td>
<td></td>
</tr>
<tr>
<td>SAP.Disconnect</td>
<td>SAP</td>
<td>With a class name that com.sap.mw.jco.rfc.MiddlewareRFC$Client</td>
<td>disconnect</td>
<td></td>
</tr>
<tr>
<td>SAP.Reset</td>
<td>SAP</td>
<td>With a class name that com.sap.mw.jco.rfc.MiddlewareRFC$Client</td>
<td>reset</td>
<td></td>
</tr>
<tr>
<td>SAP.CreateTID</td>
<td>SAP</td>
<td>With a class name that com.sap.mw.jco.rfc.MiddlewareRFC$Client</td>
<td>createTID</td>
<td></td>
</tr>
<tr>
<td>SAP.ConfirmTID</td>
<td>SAP</td>
<td>With a class name that com.sap.mw.jco.rfc.MiddlewareRFC$Client</td>
<td>confirmTID</td>
<td></td>
</tr>
</tbody>
</table>
Example Message Queue Backend Configuration

On this page:
- Monitor the Server by Ignoring the Queue Name
- Temporary Queues
- Session ID in the Queue Name

Monitor the Server by Ignoring the Queue Name

In JMS example, the application is making calls to a message server that handles several queues. The sample destination names look like this:

- AccountQ
- AccountReplyQ
- AccountRecQ
- AccountDebitQ

The default automatic discovery rule detects one backend for each unique destination and so the flow map shows one queue backend for each different queue name. In this example, each of the above would show as a separate backend on the application flow map. If you are interested in monitoring the performance of the server and not each queue name, you can modify the configuration to ignore the Destination property and use just the Type and Vendor.

To achieve this result, you create a new custom JMS Discovery Rule.

Taking IBM MQ for another example, the application is making calls to a message server that handles several queues. The sample destination names look like this:

- MQhostwest-US:1521
- MQhosteast-US:1521
- MQhostsouth-US:1521

The default automatic discovery rule detects one backend for each unique destination and so the flow map shows one queue backend for each different queue name. In this example, each of the above would show as a separate backend on the application flow map. If you are interested in monitoring the performance of the server and not each queue name, you can create a discovery rule that just uses the Host and Port, as follows:

Temporary Queues
In this example an application creates many temporary JMS response queues that are deleted after the message is received. By default, AppDynamics discovers these queues separately and lists each one as a unique remote service. This default behavior probably does not enable effective monitoring. Instead, you can create a custom JMS discovery rule stating that if the destination name contains "TemporaryQueue", list it as "WeblogicTempQueue", or whatever name makes sense in your monitoring environment. In this way, you can monitor the performance that matters. The configuration to accomplish this is shown in the following screen shot:

**Create Custom JMS Backend Discovery Rule**

<table>
<thead>
<tr>
<th>Name</th>
<th>WebLogicTempQueue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enabled</td>
<td>✓</td>
</tr>
<tr>
<td>Correlation Enabled</td>
<td>✓</td>
</tr>
<tr>
<td>Priority</td>
<td>1</td>
</tr>
</tbody>
</table>

**Match Conditions**

Backends that match ALL of the enabled match conditions below will be discovered and named according to the 'Backend Naming Configuration' below. You must configure at least one condition.

- **Destination**: Equals
- **DestinationType**: Equals
- **Vendor**: Equals

**Backend Naming Configuration**

Configure how Backends will be named, what properties to use in the name, and how to use these properties. All of the properties used will be concatenated together in the Backend name.

**Destination**

- **Use Destination in the Backend Name**

**How will Destination be used in the Backend Name?**

- **Regular Expression**
- **Regular Expression Groups** (Comma Separated List of list)

---

**Session ID in the Queue Name**

If your JMS queues use the session ID in the destination, this causes each session to be identified as a separate backend. In this case, you might not be interested in seeing each queue separately, and instead want to aggregate everything for the same host and port to the same backend. You can generate the right name and the correct number of backends to monitor by editing the automatic discovery rule. Doing this enables you to monitor the key performance indicators (KPIs) that are of most interest to you.
Example JDBC Backend Configuration

On this page:
- Multiple Databases from Same Vendor
- JDBC with Complex URLs
- EC2 Hosted Databases

Related pages:
- Java Backend Detection
- Using Regular Expressions

Depending on exactly what you need to monitor, you may want to change the default JDBC configuration. When you see the same physical database represented as multiple JDBC databases, you may need to revise the automatic discovery rule. Doing this enables you to more effectively monitor the key performance indicators (KPIs) that are of most value to you.

**Multiple Databases from Same Vendor**

JDBC connections to the same physical Oracle database (with the same URI) may appear as multiple backends. In some circumstances, the Vendor property captured for the database is different. This can happen when different drivers are used to access the database. For example, you might see JDBC backends with the following vendor names:

- Oracle DB
- Oracle

If the database driver package name matches the standard Oracle database driver, then the vendor name used is “Oracle DB”. If it doesn’t match, then the product name from the database metadata (using the java.sql.DatabaseMetaData class) is used as a vendor name. So database calls that use different drivers to reach the same physical database may be detected as separate databases. You can fix this by disabling the use of the Vendor property in the discovery rule.

**JDBC with Complex URLs**

In this example, the database URL is configured for high availability, so it is quite long and complex. Choosing the Run a regular expression on it URL option is the way to go. Disable the use of the Host and Port properties for JDBC automatic discovery. Instead enable the use of the URL that appears in the JDBC call, along with a regular expression to get the correct database naming and discovery.

For example, to extract all the hosts and all the ports from the following URL:

```
jdbc:oracle:thin:@(DESCRIPTION_LIST=(LOAD_BALANCE=OFF)(FAILOVER=ON)(DESCRIPTION= (ADDRESS_LIST=(LOAD_BALANCE=ON)(ADDRESS=(PROTOCOL=TCP)(HOST=titanpfmc01)(PORT=1521))) (CONNECT_DATA=(SERVICE_NAME=trafigura.com)))
```

This sample is for a string that contains the host and service name twice. You can also use port in your regular expression if needed by your requirements.

The following regular expression applied to the previous high availability URL results in a backend name similar to this: titanpfmc01-trafigura.com-titanpfmc02-trafigura.com.

```
.*HOST=({[^\\]*}).*SERVICE_NAME=({[^\\]*}).*HOST=({[^\\]*}).*SERVICE_NAME=({[^\\]*}).*
```

Note: the expression starts and end with ".". Set *Regular expression groups* to "1,2,3,4".
Set the **Merge Delimiter** to ".".
This configuration looks like this in the UI:

**EC2 Hosted Databases**

AppDynamics automatically discovers JDBC backends based on host, port, URL, database, version and vendor values. To monitor all JDBC databases that contain "EC2" in their host names as a single database, create a JDBC custom discovery rule and use the following match condition: Host Contains "EC2" as shown in the following screen shot.

Assuming host names of the format "EC2-segment2-segment3", use the following naming configuration:
This configuration results in a single database icon on the flow map named “EC2”.
Apache Kafka Consumer Backends

On This Page:

- KafkaConsumer.poll and Kafka Stream Entry Points
- Kafka SimpleConsumer Entry Points

You can configure the AppDynamics Java Agent to detect entry points for Apache Kafka consumer activity using KafkaConsumer.poll()(introduced in Kafka v0.10) or Kafka Streams and Kafka SimpleConsumer.fetch()(both introduced in Kafka v0.9). Prior to version 0.11, the Kafka payload did not include a location to store correlation data, so end-to-end Business Transaction correlation is only possible with Kafka client and broker versions 0.11 or greater.

KafkaConsumer.poll and Kafka Stream Entry Points

To instrument Kafka consumer entry points using KafkaConsumer.poll() or Kafka Streams, identify the method where the consumer reads messages in a loop in a custom interceptor definition. We instrument the iterator’s next method to start and end the BT for each message. There could be many iterators used for iterating messages but we only support iterators that are of type:

- kafka.consumer.ConsumerIterator
- org.apache.kafka.clients.consumer.ConsumerRecords$ConcatenatedIterable$1

1. Identify the class and method of the loop that processes messages from Kafka. Consider for example a class MyConsumer that employs the following loop to poll and process messages from Kafka:

```java
private void pollMessages() throws Exception {
    ConsumerRecords<String, String> records = kafkaConsumer.poll(1000);
    for (ConsumerRecord<String, String> record : records) {
        // Processing of the records
        System.out.println(record.value());
    }
}
```

For this case, you want to intercept:

- Class: MyConsumer
- Method: pollMessages

2. Use your preferred text editor to create and edit a file named custom-interceptors.xml at the following path:

   `<agent_home>/<version_number>/conf`

   For example:

   `/usr/home/appdynamics/appagent/ver4.3.1.0/conf/custom-interceptors.xml`

3. Copy the following XML to custom-interceptors.xml:
<custom-interceptors>
    <custom-interceptor>
        <interceptor-class-name>com.singularity.KafkaMarkerMethodInterceptor</interceptor-class-name>
        <match-class type="matches-class">
            <name filter-type="equals">my-fully-qualified-class-name</name>
        </match-class>
        <match-method>
            <name>my-method-name</name>
        </match-method>
    </custom-interceptor>
</custom-interceptors>

4. Set the value of the class name to the name of your consumer class. For instance, to specify the *MyConsumer* class:

```xml
<match-class type="matches-class">
    <name filter-type="equals">com.mycompany.mypackage.MyConsumer</name>
</match-class>
```

5. Set the value of the method name to the name of your message processing loop method. For instance, to specify the *pollMessages* method:

```xml
<match-method>
    <name>pollMessages</name>
</match-method>
```

After the Java Agent reads the updated configuration, it detects consumer activity and upstream Kafka queue. The application flow map shows the tier receiving data from the Kafka queue.

**Kafka SimpleConsumer Entry Points**

To enable consumer entry points for Kafka clients that retrieve messages using `SimpleConsumer.fetch()`, register the `enable-kafka-consumer` node property with a value of "true".

Kafka consumer activity shows up as an exit call in this case.
.NET Backend Detection

On this page:
- ADO.NET Backends
- Directory Service Backends
- HTTP Backends
- Message Queue Backends
- MongoDB Backends
- .NET Remoting
- WCF Backends
- .NET Web Services Backends
- All Other Traffic
- Custom Exit Points

Related Pages:
- .NET Supported Environments
- Backend Detection Rules
- Exit Point Detection Rules
- Asynchronous Exit Points for .NET
- HTTP Backend Detection

The .NET Agent automatically discovers many common backend types. Most backend types have a default discovery rule and a set of configurable properties.

For instructions to revise backend discovery rules, see Backend Detection Rules.

ADO.NET Backends

The .NET Agent automatically discovers ADO.NET data providers implementing standard Microsoft interfaces as database backends. For a complete list, see "Supported ADO.NET Clients" on .NET Supported Environments.

Because the ADO.NET API is interface-based, by default AppDynamics instruments all ADO.NET database providers that implement these interfaces.

For database identification, AppDynamics uses information from the ADO.NET connection string. The connection string specifies the server address and schema or the local file name. Most connection strings are formatted according to well-known rules that can be parsed and distilled to a database name. However, because there is no standard on the connection string, it is up to the ADO.NET provider implementer to choose the format.

For some providers, AppDynamics may fail to parse the connection string. In these cases, the .NET Agent uses the complete connection string minus any user password. The property is labeled ADO.NET connection string and the value shows the connection string minus any user password.

For example, the agent names following database backend using the connection string pattern <datasource name>-<database name>:

:\SQLEXPRESS-HowdyWorldDB

ADO.NET configurable properties

You can enable or disable the use of the following properties for ADO.NET exit points:

<table>
<thead>
<tr>
<th>Configurable Properties</th>
<th>Default Detection and Naming Property?</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host</td>
<td>Yes</td>
<td>data source or database server</td>
</tr>
<tr>
<td>Database</td>
<td>Yes</td>
<td>database name</td>
</tr>
<tr>
<td>Vendor</td>
<td>No</td>
<td>type of the client-side ADO.NET library</td>
</tr>
<tr>
<td>Connection String</td>
<td>No</td>
<td>full connection string with password filtered out</td>
</tr>
<tr>
<td>Port</td>
<td>No</td>
<td>port number</td>
</tr>
</tbody>
</table>
Directory Service Backends

The .NET Agent automatically discovers exit calls to directory services that use the System.DirectoryServices.Protocols (S.DS.P) libraries.

The agent names the backend for the server name. If the agent is unable to derive the server name from the request, it constructs a name using Domain Component (DC) values.

For example, activedirectory.example.com

HTTP Backends

AppDynamics automatically detects HTTP exit points (backends).

.NET Agent for Windows

For the .NET Agent for Windows, the default HTTP automatic discovery rule uses the URL property. From the enabled properties AppDynamics derives a display name using the URL.

For example:

http://api.example.com:8989/searchfares

.NET Agent for Linux

For the .NET Agent for Linux, the default HTTP automatic discovery rule for backends uses the host name and port number.

For example:

http://api.example.com:8989

HTTP configurable properties

For both the Windows and Linux versions of the .NET Agent, you can enable or disable the use of the following properties for HTTP exit points:

<table>
<thead>
<tr>
<th>Configurable Properties</th>
<th>Default Detection and Naming Property?</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>.NET Agent for Windows</td>
<td>.NET Agent for Linux</td>
</tr>
<tr>
<td>Host</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Port</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>URL</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Query String</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

New in 4.5.13 You can now customize HTTP backend detection and naming for the .NET Agent for Linux through the Controller UI. Customizing HTTP backend detection and naming is a feature preview in this agent version, recommended for pre-production systems. For information on enabling preview features, see Enable Preview Features.

Message Queue Backends

By default, AppDynamics automatically detects and identifies many message queue exit points. For a list of the supported message-oriented middleware products, see Supported Remote Service Detection for .NET.

The default queue automatic discovery rule uses the destination property to name the message queue backend.

For example:

HowdyWorldQueue$\HWT_MQ_Server1_MsgQ

Message Queue Configurable Properties
In general, the properties listed below are used for queue exit points. However, each message-oriented product is different and there may be variations in the properties or their names.

<table>
<thead>
<tr>
<th>Configurable Properties</th>
<th>Default Detection and Naming Property?</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host</td>
<td>No</td>
<td>queue server name</td>
</tr>
<tr>
<td>Destination</td>
<td>Yes</td>
<td>name of topic or queue</td>
</tr>
<tr>
<td>Destination Type</td>
<td>No</td>
<td>queue or topic</td>
</tr>
<tr>
<td>Vendor</td>
<td>No</td>
<td>vendor from the client library</td>
</tr>
</tbody>
</table>

For more information on specific queue types, see:
- MSMQ Backends for .NET
- NServiceBus Backends for .NET
- RabbitMQ Backends for .NET

**MongoDB Backends**

By default the .NET Agent detects MongoDB exit calls for create, read, update, and delete (CRUD) operations using the C# and .NET MongoDB Driver versions 1.10, 2.0, 2.2, and 2.4.

The default MongoDB automatic discovery rule uses the host, port and database name to name the MongoDB backend. For example, mymongohost.27017.mymongodb.

**MongoDB configurable properties**

You can configure the use of the following properties for MongoDB exit points:

<table>
<thead>
<tr>
<th>Configurable Properties</th>
<th>Default Detection and Naming Property?</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host</td>
<td>Yes</td>
<td>MongoDB host</td>
</tr>
<tr>
<td>Port</td>
<td>Yes</td>
<td>MongoDB port</td>
</tr>
<tr>
<td>Database</td>
<td>Yes</td>
<td>MongoDB database name</td>
</tr>
</tbody>
</table>

**.NET Remoting**

AppDynamics automatically detects and identifies remoting exit points (backends) when an application uses .NET remoting.

The default remoting automatic discovery rule uses the URL property. For example, tcp://remoting.example.com:8648/MovieTicketBooking

See [Enable Correlation for .NET Remoting](#) to configure downstream correlation.

**.NET Remoting configurable properties**

You can configure the use of the following property for .NET Remoting exit points:

<table>
<thead>
<tr>
<th>Configurable Properties</th>
<th>Default Detection and Naming Property?</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>URL</td>
<td>Yes</td>
<td>Full URL</td>
</tr>
</tbody>
</table>

**WCF Backends**

AppDynamics automatically detects and identifies WCF exit points (backends) when an application uses the WCF client library. The default WCF automatic discovery rule uses the remote address property. The agent uses the enabled properties to derive a display name using the remote address.

For example, http://wcf.example.com:8205/Services/Service1.svc
### WCF configurable properties

You can enable or disable the use of the following properties for WCF exit points:

<table>
<thead>
<tr>
<th>Configurable Properties</th>
<th>Default Detection and Naming Property?</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remote Address</td>
<td>Yes</td>
<td>URL minus the query, fragment and user information (name and password)</td>
</tr>
<tr>
<td>Operation Contract</td>
<td>No</td>
<td>WCF operation name</td>
</tr>
<tr>
<td>URL</td>
<td>No</td>
<td>full URL</td>
</tr>
<tr>
<td>Host</td>
<td>No</td>
<td>host portion of URL</td>
</tr>
<tr>
<td>Port</td>
<td>No</td>
<td>port number if present in the URL, otherwise protocol default</td>
</tr>
<tr>
<td>SOAP Action</td>
<td>No</td>
<td>for web service calls, the SOAP action</td>
</tr>
</tbody>
</table>

### .NET Web Services Backends

By default, AppDynamics automatically detects and identifies web services exit points (backends) when an application uses the Microsoft Web Services client library. The default web services automatic discovery rule uses the URL property. From the enabled properties AppDynamics derives a display name using the URL.

For example, http://webservice.example.com:8105/Services/Service1.asmx

### Web Services configurable properties

You can enable or disable the use of the following properties for Web Services exit points:

<table>
<thead>
<tr>
<th>Configurable Properties</th>
<th>Default Detection and Naming Property?</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service</td>
<td>No</td>
<td>web service name</td>
</tr>
<tr>
<td>URL</td>
<td>Yes</td>
<td>full URL</td>
</tr>
<tr>
<td>Operation</td>
<td>No</td>
<td>web service operation name</td>
</tr>
<tr>
<td>Soap Action</td>
<td>No</td>
<td>SOAP action</td>
</tr>
</tbody>
</table>

### All Other Traffic

For information on All Other Traffic transactions, see Business Transactions.

### Custom Exit Points

If you want to monitor a backend that isn't listed under ‘Remote Service Detection’ on .NET Supported Environments, then configure a custom exit point.

By default, you have to restart the instrumented application for instrumentation changes to take effect. You can enable Runtime Reinstrumentation for the .NET Agent so that you don’t need to restart your application and the AppDynamics.Agent.Coordinator after instrumentation changes.
Asynchronous Exit Points for .NET

Asynchronous programming patterns can enhance the scalability and performance of applications. Microsoft .NET lets you designate methods as asynchronous tasks.

The .NET runtime releases resources for asynchronous methods while tasks complete. When task processing finishes, the runtime calls back to the originating asynchronous method so the method may continue processing.

The .NET Agent detects certain asynchronous programming patterns as exit points. Because tasks may execute in parallel, AppDynamics sometimes represents asynchronous activity differently from synchronous activity in the Controller.

AppDynamics differentiates .NET async backend tracking from thread correlation. To configure thread correlation, see Thread Correlation for .NET.

**Supported Asynchronous Exit Point Patterns**

The agent discovers exit points for Microsoft .NET 4.5 `async` and `await` keywords. See Asynchronous Programming with Async and Await.

The agent detects asynchronous exit points for the following backend types:

- ADO.NET
- Azure Blob Storage, Azure File Storage, and Azure Table Storage
- HTTP
- MongoDB
- WCF
- Web Service

Flow maps in the Controller UI represent asynchronous exit points as dotted lines labeled “async”. Because calls in an asynchronous transaction may execute simultaneously, the Controller doesn’t display end-to-end percentage times for individual asynchronous calls.

**Troubleshoot Asynchronous Calls in Transaction Snapshots**

Transaction snapshots include several features to help you discover problem areas in business transactions that use asynchronous methods. For an overview of transaction snapshots, see Transaction Snapshots.

**Transaction Snapshot Flow Map**

The Transaction Snapshot Flow Map graphically represents the business transaction. It displays the user experience, execution time, and timestamp of the transaction. The flow map also provides details of the overall time that is spent in a particular tier and in database and remote service calls. The `async` label indicates asynchronous calls.

**Snapshot Execution Waterfall View**

The transaction Snapshot Execution Waterfall View shows a timeline representation of the end-to-end execution of the business transaction. Synchronous and asynchronous processes appear on a bar diagram illustrating their relative execution time arranged in chronological order. Synchronous calls are represented by solid line arrows, and asynchronous calls are represented by dotted line arrows.

The waterfall view enables you to visually identify which processes are running the longest. Double-click a bar to drill down to a call graph and investigate problematic code.
Analyze Asynchronous Activity in the Metric Browser

The Metric Browser displays asynchronous activity in the following places:

- Business Transaction Performance > Business Transactions > tier > business transaction > Thread Tasks > Asynchronous Operation > External Calls
- Overall Application Performance > tier > Thread Tasks > Asynchronous Operation > External Calls
- Overall Application Performance > tier > Individual Nodes > node name > Thread Tasks > Asynchronous Operation > External Calls
The AppDynamics .NET Agent (agent) automatically detects MSMQ backends when an instrumented tier makes a call to MSMQ. MSMQ exit points are methods that publish or push messages to a queue. If you are using NServiceBus over the MSMQ transport, see NServiceBus Backends for .NET.

MSMQ entry points are methods that listen or poll for new messages in the queue. Before the agent can detect entry points or perform downstream correlation for MSMQ you must define the correlation field and, for multithreaded architectures, specify threading architecture.

**Exit Points and Backend Naming**

The agent names the queue for the queue name returned from the system.

**Entry Points**

To enable downstream correlation for MSMQ, you must configure the agent as follows:

**Define the MSMQ correlation field**

Register the `msmq-correlation-field` app agent node property on both the publishing tier and on the receiving tier. Specify the field where the agent will write and read correlation information.

The agent supports the following fields:

- Extension, the default field
- Label

By default, the .NET Agent writes to the Extension field. However, some frameworks built on MSMQ write data to the Extension field. In this case, the Extension field does not work because it is already in use, so you must configure the .NET Agent to write correlation data to the Label field.

Choose a field not in use by your queue implementation. If your implementation uses both the "Label" and "Extension" fields, downstream correlation is not currently possible.

See "msmq-correlation-field" on App Agent Node Properties Reference. For instructions to register a node property, see App Agent Node Properties.

**Specify the threading architecture**

If your queue implementation uses a multithreaded architecture, you must register the `msmq-single-threaded` app agent node property to specify the threading architecture. Set `msmq-single-threaded` to "False" for multithreaded implementations of MSMQ. See "msmq-single-threaded" on App Agent Node Properties Reference. For instructions to register a node property, see App Agent Node Properties.

The threading architecture dictates how the agent calculates the call timing for the message queue:

- For single-threaded queues, the agent calculates the call time between receive requests. The call time begins at the end of the first receive method and ends when the next receive method starts.

In the following example, the length of the `ProcessMessage()` method:
MessageQueue messageQueue;
for(;;)
{
    var message = messageQueue.Receive();
    ProcessMessage(message);
}

- For multithreaded queues, the agent does not capture timing.
NServiceBus Backends for .NET

On this page:

- Exit Points and Backend Naming
- Entry Points in a Single-threaded Architecture
- Entry Points in a Multithreaded Architecture
- Resolve Issues with NServiceBus Backends

Related pages:

- .NET Backend Detection

The AppDynamics .NET Agent automatically discovers exit points to NServiceBus backends when an instrumented tier makes a call to NServiceBus over MSMQ or RabbitMQ transports. The agent discovers NServiceBus entry points for downstream correlation from NServiceBus version 5.

Exit Points and Backend Naming

NServiceBus exit points are methods that publish or push messages to a queue from an upstream tier. The agent detects exit points regardless of the NServiceBus version or threading architecture. The agent names the queue for the queue name returned from the system.

Entry Points in a Single-threaded Architecture

NServiceBus entry points are methods that listen or poll for new messages in the queue. When the message receiver uses a single-threaded architecture, the .NET Agent automatically discovers entry points and correlates downstream activity without additional configuration.

The agent measures the call time from the end of the first receive method to the start of the subsequent receive method. In the following example loop, the length of the ProcessMessage() method:

```csharp
MessageQueue messageQueue;
for(;;)
{
    // Call timing for previous iteration ends.
    var message = messageQueue.Receive();
    // Call timing begins.
    ProcessMessage(message);
}
```

Entry Points in a Multithreaded Architecture
For the .NET Agent to automatically detect NServiceBus entry points when the receiver is running a multithreaded architecture, set the `n-servicebus-single-threaded` app agent node property to "false" on the receiver. See "n-servicebus-single-threaded" on App Agent Node Properties Reference. For instructions to register a node property, see App Agent Node Properties.

For multi-threaded message receivers, the agent does not capture timing.

**Resolve Issues with NServiceBus Backends**

- If you have NServiceBus version 5 and are not seeing downstream correlation on the receiver, verify the threading architecture on the receiver. Set the `n-servicebus-single-threaded` app agent node property to "false" on the multithreaded receivers.
- Only use the node properties for MSMQ, `msmq-single-threaded` and `msmq-correlation-field`, if you are using MSMQ independently of NServiceBus. The .NET Agent does not use MSMQ node properties when discovering and instrumenting NServiceBus over MSMQ transport.
- If you are using NServiceBus version 4, you can disable NServiceBus instrumentation on both the publisher and the receiver:
  - Copy the following Instrumentation element to a child of the Machine Agent element:

    ```xml
    <!--Disable NServiceBus instrumentation-->  
    <instrumentation>
      <instrumentor name="NServiceBusReceiveContextInstrumentor" enabled="false" />
      <instrumentor name="NServiceBusPublishExitInstrumentor" enabled="false" />
      <instrumentor name="NServiceBusEntryInstrumentor" enabled="false" />
      <instrumentor name="NServiceBusExitInstrumentor" enabled="false" />
    </instrumentation>
    ```

    For more information, see "Instrumentation Element" on .NET Agent Configuration Properties.
  - Configure the publisher and receiver as you would for MSMQ or RabbitMQ.
RabbitMQ Backends for .NET

On this page:
- Exit Points and Backend Naming
- Entry Points

Related pages:
- .NET Backend Detection
- RabbitMQ Exchanges and Exchange Types
- RabbitMQ Monitoring Extension
- App Agent Node Properties

The AppDynamics .NET Agent automatically detects RabbitMQ backends based upon calls from instrumented tiers. RabbitMQ exit points are methods that publish or push messages to a queue. RabbitMQ entry points are methods that listen or poll for new messages in the queue.

If you are using NServiceBus over the RabbitMQ transport, see NServiceBus Backends for .NET.

Exit Points and Backend Naming

The agent discovers a RabbitMQ backend exit point when your application sends a message to the queue using the `BasicPublish()` method.

By default, the agent names the RabbitMQ backend for the exchange parameter of the `BasicPublish()` method.

For example:

```csharp
model.BasicPublish("MyExchange", "", false, false,
                 basicProperties, Encoding.UTF8.GetBytes(message));
```

In this case the agent names the queue `MyExchange`.

You can refine the backend name to include some or all segments of the routing key. To configure RabbitMQ naming you must be familiar with your implementation RabbitMQ exchanges and routing keys. See RabbitMQ Exchanges and Exchange Types.

Refine backend naming

Register the `rmqsegments` node property. For instructions on how to set a node property, see App Agent Node Properties.

Name: rmqsegments
Description: "Configure RabbitMQ naming to include routing key segments."
Type: Integer
Value: <integer>
The routing key is a string. The agent treats dot-separated ("." ) substrings of the routing key as segments. Set the value to an integer that represents the number of routing key segments to include in the name.

In the following example the routing key is "abc. def. ghi". Set the rmqsegments value to "2" to name the queue "MyExchange. abc. def".

```csharp
model.BasicPublish("MyExchange", "abc. def. ghi", false, false,
basicProperties, Encoding.UTF8.GetBytes(message));
```

After you save the node property, the Controller sends the configuration to the agent. After some time the RabbitMQ backend shows up with the new name.

**Entry Points**

The agent discovers RabbitMQ backend entry point when your application polls the queue. AppDynamics auto-detects RabbitMQ based upon the following patterns:

- **BasicGet**
- **HandleBasicDeliver**

**BasicGet Method**

The agent detects the pattern below where the application periodically polls the message queue using the `BasicGet()` method. The call timing is limited to the time spent inside the `while` loop. The timer only starts when the BasicGet result returns a value at line 4. The timer ends when the next BasicGet executes. In this example, the application polls every five seconds, so the execution time equals the time in the `if` loop plus five seconds.

```csharp
while (true)
{
    var result = channel.BasicGet("MyExchange", true);
    if (result != null)
    {
        var body = result.Body;
        var message = Encoding.UTF8.GetString(body);
        Console.WriteLine("Received: {0}.", message);
    }
    Thread.Sleep(5000);
}
```

**HandleBasicDeliver Method**

The agent detects the `HandleBasicDeliver()` method for custom implementations of the `IBasicConsumer` interface. In this case, the call timing reflects the execution time for the `HandleBasicDeliver` method.
Correlation Over Microsoft BizTalk

On this page:
- Requirements
- Entry Points
- Instrument the BizTalk Service

Related pages:
- Configure the .NET Agent for Windows Services and Standalone Applications
- Manage Windows Performance Metrics

When you install the .NET Agent on a Microsoft BizTalk Server, the agent automatically discovers BizTalk entry points and exit points so you can monitor your BizTalk integration.

Requirements

- Microsoft BizTalk 2010, 2013
- BizTalk must be a downstream tier from an originating tier for a business transaction. The agent doesn't discover BizTalk as the originating tier of a transaction.

Entry Points

The .NET Agent detects incoming BizTalk requests using the WCF Send Adapter or the SOAP Send Adapter.

Instrument the BizTalk Service

1. Install the .NET Agent on the BizTalk server. See Install the .NET Agent for Windows.
2. Register the "enable-soap-header-correlation" app agent node property with a value of "true" for the nodes upstream from the BizTalk tier. See App Agent Node Properties.
3. Instrument the BizTalk service executables: BTSNTSvc.exe and BTSNTSvc64.exe. See Configure the .NET Agent for Windows Services and Standalone Applications. For instance:

```xml
<standalone-applications>
  <standalone-application executable="BTSNTSvc">
    <tier name="BizTalk Service"/>
  </standalone-application>
</standalone-applications>
```

If you run multiple instances of BizTalk on the same Windows server, you can assign the different instances to unique nodes using command line options. For example, to differentiate nodes within the same tier, specify the BizTalk Service command "name" parameter in the standalone application command-line attribute:
3. <standalone-applications>
   <standalone-application executable="BTSNTSvc"
   command-line="-name "BizTalk1"">
     <tier name="BizTalk Service"/>
     <node name="BizTalk1"/>
   </standalone-application>
   <standalone-application executable="BTSNTSvc"
   command-line="-name "BizTalk2"">
     <tier name="BizTalk Service"/>
     <node name="BizTalk2"/>
   </standalone-application>
</standalone-applications>

Notice the use of quotes to enclose the name command-line parameter.

4. Optionally enable BizTalk performance counters. For more information, see Manage Windows Performance Metrics. Refer to the Microsoft BizTalk Server documentation for descriptions of individual performance counters.

<machine-agent>
 <!-- BizTalk Performance Counters -->
 <perf-counters>
   <perf-counter cat="BizTalk:Message Agent" name="High
database session" instance="*"/>
   <perf-counter cat="BizTalk:Message Agent" name="High
database size" instance="*"/>
   <perf-counter cat="BizTalk:Message Agent" name="High
in-process message count" instance="*"/>
   <perf-counter cat="BizTalk:Message Agent" name="High
message delivery rate" instance="*"/>
   <perf-counter cat="BizTalk:Message Agent" name="High
message publishing rate" instance="*"/>
   <perf-counter cat="BizTalk:Message Agent" name="High
process memory" instance="*"/>
   <perf-counter cat="BizTalk:Message Agent" name="High
system memory" instance="*"/>
   <perf-counter cat="BizTalk:Message Agent" name="High
thread count" instance="*"/>
   <perf-counter cat="BizTalk:Message Agent" name="Message
delivery delay (ms)" instance="*"/>
   <perf-counter cat="BizTalk:Message Agent" name="Message
delivery throttling state" instance="*"/>  
   <perf-counter cat="BizTalk:Message Agent" name="Message
publishing delay (ms)" instance="*"/>
   <perf-counter cat="BizTalk:Message Agent" name="Message
publishing throttling state" instance="*"/>
   <perf-counter cat="BizTalk:Message Agent" name="Message
delivery incoming rate" instance="*"/>
   <perf-counter cat="BizTalk:Message Agent" name="Message
delivery outgoing rate" instance="*"/>
   <perf-counter cat="BizTalk:Message Agent" name="Message
delivery incoming rate" instance="*"/>
   <perf-counter cat="BizTalk:Message Agent" name="Message
delivery outgoing rate" instance="*"/>
</perf-counters>
</machine-agent>
<perf-counter cat="BizTalk:Message Agent" name="Message publishing incoming rate" instance="*"/>
<perf-counter cat="BizTalk:Message Agent" name="Message publishing outgoing rate" instance="*"/>
<perf-counter cat="BizTalk:Message Agent" name="Message delivery throttling state duration" instance="*"/>
<perf-counter cat="BizTalk:Message Agent" name="Message delivery throttling user override" instance="*"/>
<perf-counter cat="BizTalk:FILE Receive Adapter" name="Bytes received/Sec" instance="*"/>
<perf-counter cat="BizTalk:FILE Receive Adapter" name="Lock failures/sec" instance="*"/>
<perf-counter cat="BizTalk:FILE Receive Adapter" name="Messages received/Sec" instance="*"/>
<perf-counter cat="BizTalk:FILE Send Adapter" name="Bytes sent/Sec" instance="*"/>
<perf-counter cat="BizTalk:FILE Send Adapter" name="Messages sent/Sec" instance="*"/>
<perf-counter cat="BizTalk:SOAP Receive Adapter" name="Messages received/Sec" instance="*"/>
<perf-counter cat="BizTalk:SOAP Send Adapter" name="Messages
Once you've completed instrumentation, the .NET Agent correlates traffic through your BizTalk server:

Set app agent node property `enable-soap-header-correlation` to "true" nodes upstream from BizTalk.

BizTalk server is a downstream tier.

Incoming request on WCF Send Adapter

Outgoing WCF request

If you enabled Performance Counter metrics, they appear under the Custom Metrics tree in the Metric Browser.
Apache Web Server Backend Detection

On this page:
- Apache Modules

Related pages:
- Backend Detection Rules

The Apache Agent discovers Apache Modules as backends. To review general information about monitoring backends see Backend Detection Rules.

Apache Modules

The Apache Agent automatically detects loaded Apache modules as remote services:

- Modules must be in the handler stage.
- The agent excludes a list of common modules from discovery, see "Remote Service Detection" on Supported Apache Web Servers.

By default, the Apache Agent includes metrics for module backends with the downstream backend or tier. The Controller doesn't display the module on flow maps. For example, if Apache calls mod_jk.c, the module backend doesn't show on flow maps and metrics are included with the downstream Tomcat backend or tier.

To view Apache modules on the flow map and to see separate metrics for Apache modules, set AppDynamicsResolveBackends to "OFF" in the appdynamics_agent.conf file. See Install the Apache Agent.

The agent names the Apache modules backend for the module name and the server: <module>-<host>:<port> . For example, mod_jk.c-myapache.example.com:80.
HTTP Backend Detection

On this page:
- About HTTP Backend Naming
- Using the URL Path in a Detection Rule
- Custom HTTP Backend Detection and Naming Configuration

Related pages:
- Java Backend Detection
- .NET Backend Detection

AppDynamics automatically detects HTTP backends when outbound calls are made through supported HTTP clients, and by default names them according to their host and port. If the default configuration results in names that are not meaningful for your application (such as EC2 host names, file paths, and ports in the name) you can change the default discovery rule, as described here.

Additionally, if an HTTP call is destined for an http router which could route the request to one of several instrumented downstream tiers then the default naming rules will need to be modified such that the name includes enough segments of the target URL such that each destination tier is associated with a differently named backend.

About HTTP Backend Naming

To ensure that the HTTP backends detected in your application have meaningful names, and that Business Transaction correlation functions correctly, your custom configuration will need to take into account the specific format used in your environment.

The format can vary even within a single environment. For example, some backend systems may have hostnames prefixed with ec2storage, which may not be meaningful in naming, while others may use hostnames such as salesforce.com, which may be meaningful.

To account for different formats, you should create a custom rule rather than changing the automatic discovery rule. This lets you apply different rules for different URL formats.

You can then apply a specialized approach for each case, as in the following examples:

- For format "ec2storage/servicename", you would use the URL
- For format "salesforce.com", use the host name
- For the other backends, you may use a query string
- For calls that could hit one of several downstream tiers via an HTTP router, you should use sufficient segments of the target URL in the name such that each downstream tier that could process the request is associated with a differently named backend

In some cases, your HTTP backend discovery configuration might consist of a combination of the default rule and custom rules. The following section walks you through a specific example:

Using the URL Path in a Detection Rule

For example, when all the HTTP backends for a tier or application have a similar format, such as a prefix like "ec2storage", you can generate the right name and the correct number of backends to monitor by editing the automatic discovery rule. This enables you to monitor the KPIs that are interested in.

Consider an application with the following HTTP URLs:

http://ec2-17:5400/service1
http://ec2-17:5450/service2
http://ec2-18:5400/service1
http://ec2-18:5450/service2

In this case, measuring performance based on host name would not be useful, since the IP addresses are transient and all performance numbers would be irrelevant after the IP addresses recycle. Instead, you can monitor by service name by avoiding the use of Host and Port properties and using only the URL property, as follows:

1. Edit the Automatic Backend Discovery rule for HTTP for your agent type. See Backend Detection Rules for details on accessing this screen.
2. First, select and disable the use of Host and Port.
3. Then select and enable the property you want to use to uniquely identify the backend. In this case, select URL and check Use URL in the Backend Name.
4. For the field How will URL be used in the Backend name?, select Use a segment of it.
5. From the segment options drop-down list, select Use the first N Segments.
6. Enter “/” slash for the Split Delimeter.
   Use a similar technique to strip out some segment of the URL, such as a user name as in the following URLs:
   [http://host:34/create/username1]
   [http://host:34/create/username2]
7. Once you change the configuration, delete all HTTP backends. When the agent rediscovers the backends with the new configuration, the flow map shows only the service backends.

Custom HTTP Backend Detection and Naming Configuration

By default, AppDynamics uses the host name and port number to detect and name HTTP backends. This may be suitable in many cases, however, when your microservices, containers or serverlets are managed by an API gateway or portal, backend services communicating with a client or with each other cannot be correctly mapped to the proper tiers unless you edit automatic discovery to use a portion of the query string or use HTTP custom discovery rules to uniquely name them. AppDynamics recommends you create a custom HTTP backend discovery rule in this case.

The following screenshots provide an example configuration.

Here we choose Match Conditions to describe the API gateway host, in this case, api.appdynamics.com.
This screenshot shows where we choose to use the First 3 segments, using ‘/’ as both the split and merge delimiters.

Configuring the HTTP Backend Discovery Rule as above detects backend metrics and reports them under a business transaction name that begins with `api.appdynamics.com` followed by the first three segments of the URL of the transaction, such as `api.appdynamics.com` or `api.appdynamics/api/payment`. Transactions matching the configured HTTP discovery rule are recognized as individual `/api/catalog` backends even when they share the same host name and port number. These transactions can be mapped to different tiers so you can analyze the metrics of the transactions that occur behind the API gateway.

There is a known issue in HTTP backend detection configuration in the current preview. When defining HTTP backend using URL segments, segment enumeration starts with number 2 instead of number 1, and segment 1 always returns empty. For example, to define unique backends using the first two segments of the URL, you would need to configure HTTP backend detection to use the first 3 segments or segment 2 and 3.
Error Detection

You can customize how AppDynamics associates code events with business transaction errors by adding or removing the methods, log messages, HTTP codes, or redirect pages that are configured to indicate business transaction errors.

Permissions

To configure error detection, you need the "Configure Error Detection" permission.

Error Detection Configuration

You can view or modify error detection configuration in the Error Detection tab under Configuration > Instrumentation. Tabs take you to the configurable options for each programming language or platform. Note that some settings are platform specific, such as custom loggers or error detection using redirect pages, which are available for Java and .NET only.

To prevent a transaction from being marked as an error transaction in AppDynamics, clear the Mark Business Transaction as error check box. If clear, the transaction does not add to the error count metric and is considered in other metrics, such as response time, despite the occurrence of the error.

Supplement Error Detection

For Java and .NET, you can supplement the methods that AppDynamics considers error methods from common error handling frameworks with your own, custom-defined error methods by defining a customer logger. When the application invokes the method in a business transaction, AppDynamics considers the transaction to be an error transaction for metric purposes.

In the custom logger configuration, you can specify the parameter that contains the exception or error information passed to the method. This information will appear as the error message in AppDynamics, which invokes toString() on the parameter to extract the message.

To configure a custom logger

1. In the Controller UI, on the Error Detection configuration tab, click Add Custom Logger Definition in the error configuration window.
2. Enter a descriptive name for the custom logger definition and use the settings to identify the classname and method for the custom logger. For more information on how to use the AppDynamics UI to identify classes and methods, see Data Collectors.
3. For the Method Parameter field, add a parameter definition for each parameter in the signature of the method.
   If the method is overloaded, create a logger definition for each form of the overloaded method where you want to detect errors. Your custom method must accept at least one parameter, which should be the parameter that conveys the logged error or exception information.
4. For the Exception Parameter field, identify the parameter in the method signature that contains the exception object by index number (0-based).
   This parameter, identified by an index, can be any type of object, including Arrays. If the object is not null, AppDynamics converts the object to a string, the result of which constitutes the error details. A business transaction is considered to be an error only if a non-null exception object is passed to the logger method.

Consider the following custom logger class and methods in a .NET application:

- Namespace and class = Logging.MyLogger
- logger.Error(string message, int param1)
- logger.Error(string message, int param1, int param2)
Notice that the Error method is overloaded. To capture errors logged in the second form of the method using the first parameter as the error to log. The following screenshot illustrates this configuration:

![Custom Logger Definition](image)

Notice that the exception parameter is set to 0, identifying the string message method parameter as the error message. When enabled, AppDynamics detects and reports the custom logger.

**Detect HTTP Responses or Redirect Pages as Errors**

In Java, .NET, Node.js, and Python you can configure errors based on HTTP response codes. If the error code is set as part of business transaction processing, the transaction is marked as an error. For PHP, it is possible to classify transactions as in error based on 4xx HTTP return codes using the `PHP Agent Configuration Settings`.

By default, AppDynamics captures HTTP error codes from 400 to 505. HTTP response codes may convey errors that occur at the business level of an application. For example, in an e-commerce application, a 522 error might indicate that an item is out of stock. For this case, you may want to include the error as a default transaction error indicator.

To exclude a return code add them to the error detection list by creating a custom error code range and then disabling that error code by clearing its `Enabled` checkbox. This in effect excludes the error code as an error indicator.

![Error Detection Using HTTP Return Codes](image)

For Java and .NET agents, you can specify custom redirect target error pages as error indicators. To specify a redirect page, click `Add Error Redirect Page` and add a name for the configuration and a regular expression to match the URL of the page, such as "AcmeErrorPage.jsp".
Ignore Exceptions and Log Messages as Error Indicators

Certain types of exceptions, loggers or log messages as transaction error indicators may not reflect events that should be counted as transaction errors in your environment. They may include exceptions raised by application framework code or by user login failures.

When you configure an exception to be ignored, the agent detects the exception, increments the exception count, and displays the exception in Exceptions lists in the UI, but the business transaction in which the error is thrown is not considered an "error transaction" for monitoring purposes. The transaction snapshot would not show the exception in the Summary or Error Details section and the user experience for the transaction instance would be unaffected by the exception.

To configure exceptions for the agent to ignore, click the Add New Exception to Ignore button in the error configuration window.

Enter the class name of the exception to be ignored and the match condition for the exception message. For the match condition, if you do not need to filter for specific messages in the exception, select "Is Not Empty". If you want to filter for Null, select "Is Not Empty" and use the gear icon to select NOT, which, in effect, tests for "is empty".

The following example directs the agent to ignore java.lang.RuntimeExceptions that wrap a javax.sql.SQLException only when the exception.getMessage() call in the root exception contains the string "format". (Other types of java.lang.Runtime exceptions will not be ignored.)

When you define the class of an exception to ignore by an exception chain, the exception message against which the match condition is applied must be in the root exception of the chain. The match is not applied to any nested exceptions.

In .NET and Java, you can specify that errors logged with certain loggers or logger categories be ignored. Click Add New Category/Logger to Ignore.

Error Detection Notes by Platform

The following sections list error detection configuration and error monitoring considerations, if any, by application platform.

Error Configuration for .NET

By default, the agent can instrument calls using NLog and Log4Net. It can also gather information from System Trace and the Event Log. To instrument calls using other loggers, add a custom logger definition. See Configuring a Custom Logger.

Messages logged as higher than ERROR include severe levels such as CRITICAL or FATAL.

If you do not want system trace or event log errors to be monitored, disable them by checking the appropriate box. System trace errors are anything written as an error message to the Listeners collection via TraceError.
Event log errors are anything written to the EventLog when the type is set to error. For example:

```csharp
myEventLog.WriteEntry(myMessage, EventLogEntryType.Error, myID);
```

Error Configuration for PHP

The PHP Agent instruments the PHP reporting facility. PHP applications can use trigger_error to report errors through that facility. PHP extensions and PHP itself can also use the PHP facility for reporting errors.

By default, the agent can check for PHP errors using multiple thresholds:

- If you select Error, the agent reports only messages and exceptions marked Error.
- If you select Warning, the agent reports only messages and exceptions marked Error and Warning.
- If you select Notice, the agent reports messages and exceptions marked Error, Warning and Notice.

Error Configuration for Node.js

The Node.js Agent reports exceptions thrown by the Node.js application or by Node.js itself.

You can configure certain exceptions or logged messages not to cause transactions to be reported as errors using the ‘Ignored Exceptions’ and ‘Ignored Messages’ lists. See Ignoring Exceptions and Log Messages as Error Indicators.

Error Configuration for Python Notes

By default, the Python Agent reports error transactions for unhandled exceptions, HTTP status codes greater than or equal to 400, and messages logged at ERROR or higher by the Python logging module.

If you do not want any logged errors to cause transactions to be reported as errors, clear the “Mark Business Transaction as error” checkbox. If you do not want to capture logged messages at all, clear the “Detect Errors” checkbox.

You can configure certain exceptions or logged messages not to cause transactions to be reported as errors using the ‘Ignored Exceptions’ and ‘Ignored Messages’ lists. See Ignoring Exceptions and Log Messages as Error Indicators.
Service Endpoint Detection

AppDynamics automatically detects service endpoints for Java applications. You can configure custom service endpoints for Java or .NET to monitor the performance of services.

You can also configure exclude rules for service endpoints, which prevent agents from registering service endpoints for methods that match the exclude rules. The order of precedence for match, exclude and automatic rules are:

1. Exclude rules
2. Custom service endpoint rules
3. Automatic discovery

Permissions

To configure service endpoints your user account must have the Configure Service Endpoints permission for the business application. For information on AppDynamics Role Based Access Control, see Application Permissions.

Configure Custom Service Endpoints

You configure custom service endpoints like you configure custom match rules for business transactions:

1. From the Service Endpoints pane, click Configure.
2. Click the Custom Service Endpoint tab.
3. Choose the tier on which the service runs and click the plus icon to add a service endpoint configuration.

The configuration settings are similar to business transaction entry point configuration settings. See Custom Match Rules.

As you choose a method for a service endpoint, avoid methods that occur within the following programming patterns:

- High-frequency loops, such as a for or while loop.
- Recursive functions.

Service endpoints inside such structures may cause performance overhead. Instead, consider adding a service endpoint at a point in the code upstream from a high-frequency loop or recursive function.

Configure service endpoint exclude rules in the same window as the custom service endpoints. Click Is Exclude Rule on the New Service Endpoint Definition window.

Automatic Discovery for Java

The Java Agent automatically discovers service endpoints. To configure service endpoint automatic detection settings, click to Configure on the Service Endpoints pane. You can configure the following settings:

- For existing service endpoint detection rules, you can enable or disable the detection rule.
- For servlet service endpoints, you can modify automatic naming as you would with servlet business transactions.

POJO service endpoints

For the Java Agent to detect Plain Old Java Objects (POJO) service endpoints, you must do the following:

- Create at least one POJO custom service endpoint definition and enable it. The agent only detects POJO service endpoints for
enabled custom service endpoint definitions.

- Verify automatic discovery for POJO service endpoints is enabled. It is enabled by default.

| When you disable automatic discovery for POJO service endpoints, the agent doesn't detect custom service endpoints. |

Asynchronous worker thread service endpoints

The Java Agent automatically detects asynchronous worker threads spawned within a transaction as service endpoints. The agent names worker thread service endpoints for the class name, for example, "worker1". There is no end-to-end latency metric for worker thread service endpoints.

To disable worker thread service endpoints, register the enable-async-service-endpoints app agent node property with a value of "false". See App Agent Node Properties.

Configure Service Endpoints for .NET

The .NET Agent doesn’t automatically discover service endpoints. To define a custom service endpoint for .NET, navigate to Configuration > Instrumentation > Service Endpoints > Custom Service Endpoints. Define the service endpoint as you would a custom match rule for a business transaction. See Service Endpoints and Transaction Detection Rules.

You can create service endpoints for the following entry point types:

- ASP.NET
- Message Queues
- POCO
- WCF
- Web Service

| The .NET Agent doesn't support service endpoints for MSMQ. |

By default, you need to restart the instrumented application for instrumentation changes such as adding a POCO service endpoint to take effect. You can enable Runtime Reinstrumentation for the .NET Agent so that you don't need to restart your application and the AppDynamics.Agent.Coordinator after instrumentation changes.

After you configure a custom service endpoint, you can monitor it on the service endpoint dashboard that shows KPIs and transaction snapshots where the service endpoint executed. You can also use the metric browser to analyze service endpoint performance. See Service Endpoints.

Tips for Defining .NET Service Endpoints

The methods that serve as the entry points for business transactions are often the same methods that you want to monitor as service endpoints as well. You can create service endpoints on the originating entry point for a business transaction.

- For .NET service endpoints, you can use the URL to define the service endpoint. For example, in the Travel Site tier above, set a URL match to "/Travel/Search"
- For WCF and ASP.NET web service entry points, you can use an existing transaction snapshot to find the URL for the service.
  1. Open a full transaction snapshot for the business transaction.
  2. From the upstream tier, click the exit call link, either WCF or Web Service.
     The exit call window shows the URL for the web service.
3. Open the service URL in a browser.
4. On the service page, use the URL to access the WSDL file.

**Service1 Service**

You have created a service.

To test this service, you will need to create a client and use it to call the service. You can do this using syntax:

```
svccutil.exe http://localhost:8205/Services/Service1.svc?wsdl
```

You can also access the service description as a single file:

```
http://localhost:8205/Services/Service1.svc?singleWsd1
```

This will generate a configuration file and a code file that contains the client class. Add the two files to the Service. For example:

5. The service name element shows the service name.

```
<wsdl:service name="Service1">
```

- Define POCO service endpoints exactly as you would a POCO custom match rule for a business transaction.
Data Collectors

On this page:

- Types of Data Collectors
- View Collected Data
- Configure a Data Collector
- Configuration Notes
- Java Data Collector Example
- PHP HTTP Data Collector Example

Works with:

Data collectors allow you to supplement business transaction and transaction analytics data with application data. The application data can add context to business transaction performance issues. For example, they show the values of particular parameters or return value for business transactions affected by poor performance.

This data shows the business context affected by performance issues, such as the specific user, order, or product.

Types of Data Collectors

Data collectors come in two forms:

- Method invocation data collectors capture code data such as method arguments, variables, and return values.
- HTTP data collectors capture the URLs, parameter values, headers, and cookies of HTTP messages exchanged in a business transaction.

For information on how transaction analytics uses data collectors, see Collect Transaction Analytics Data.

View Collected Data

When applied to business transactions, the data collectors supplement the information shown in transaction snapshots. The information captured by HTTP data collectors appears in the HTTP DATA and COOKIES sections, while method invocation data appears in the Business Data section. To view collected data, navigate to the Transaction Snapshots window. Double click on the transaction you want to view data for, then click Drill Down.

Configure a Data Collector

To configure data collectors, you need the Configure Diagnostic Data Collectors permission.

You can add a data collector directly from a transaction snapshot from the right-click menu for a method in a call graph.

Alternatively, you can add it manually from the Configuration > Instrumentation page. Click the Data Collector tab and click the Add button below the Method Invocation Data Collectors panel or the HTTP Request Data Collectors panel.

The new data collector configuration window appears as follows:
Note the following platform-specific considerations applicable to data collectors:

- For the C/C++ agent, you can create a method data collector using the `appd_bt_add_user_data` function(), not the Controller UI as described here. See Agent SDK for C and C++.
- For the Node.js agent, you can create a method data collector only using the `addSnapshotData()` Node.js API, not the Controller UI as described here. See Node.js Agent API Reference.
- For the Python agent, you can create a method data collector using the `add_snapshot_data()` Python agent API, not the Controller UI. See Python Agent API Reference.
- For PHP method data collectors, only the "with a Class Name that" option is valid. Also, you cannot add a method data collector to a standalone PHP function.

The general steps in configuring a data collector are:

1. Identifying the method on which to capture data. To do this, you define the method signature and optionally, filters based on the value of a code point in the method (such as return value or argument).
2. Specifying the actual code point that serves as the source of the data.
3. If the data collector applies to business transactions, choose the applicable business transactions.

Typically, creating a data collector requires knowledge of the code on which you are setting up the collector, whether based on access to the source code for the application or its documentation. For some application environments, including JVM 1.5 and .NET, you will need to restart the JVM or application server if your method invocation data collector results in modifications to instrumentation configuration (class name, method name, method parameters, and so on).

Configuration Notes

- The **Apply to new Business Transactions** option applies the collector to business transactions created after you have configured this collector. Otherwise, the data collector only applies to the business transactions you select in the subsequent data collector configuration screen.
- For **Class**, select the match condition that the data collector can use to identify the class, such as class name, implemented
interface name, and so on. If matching by class name, use the fully qualified name of the class, as appropriate for the
application platform. For example, the form of the equals field value would be:

- In Java: com.appdynamics.model.Item
- In .NET: Bookstore.Item
- In PHP: book

- Is this Method Overloaded: If this is an overloaded method, add parameters that identify the signature. You will need to create
  a data collector definition for each form of the method for which you want to capture data. For example, given the overloaded
  method in the following table, to capture data only for the second two forms, you would need to create two data collectors. The
  parameters to configure would be as shown:

<table>
<thead>
<tr>
<th>Signature</th>
<th>Parameters to configure</th>
</tr>
</thead>
<tbody>
<tr>
<td>getName()</td>
<td>None</td>
</tr>
<tr>
<td>getName(int studentId)</td>
<td>Param Index 0: java.lang.int</td>
</tr>
<tr>
<td>getName(int studentId, string name)</td>
<td>Param Index 0: java.lang.int</td>
</tr>
<tr>
<td></td>
<td>Param Index 1: java.lang.String</td>
</tr>
</tbody>
</table>

- You can refine the method selection to meet specific conditions. If you configure more than one match condition, all match
  conditions must be satisfied by the request for the data collector to be applied to the request.
- Once you identify the method, specify the code point from which you want to capture data, such as the method return value,
  argument, or a value captured by getter chain on the invoked object. You configure this code point in the Specify the Data to
  Collect from this Method Invocation section of the configuration settings.
- HTTP data collector can capture data from HTTP parameters, request attributes, cookies, and more. Notice that the
  Content-Length HTTP header is already captured in AppDynamics as the Average Request Size metric. However, you may
  choose to configure a data collector for this header to have the value appear in transaction snapshots, giving you insight, for
  example, on whether message size corresponds to slow performance.
- You can configure multiple data collectors. The effect of multiple data collectors is cumulative; if you add a custom data
  collector that does not include the collection of the URL HTTP request attribute, for example, but keep the Default HTTP
  Request Data Collector configuration in which the URL is configured to be collected, the URL is collected.

Java Data Collector Example

In this example, we'll set up a data collector on a Java application. We want to create a data collector on the method
getCartTotal(), shown in the following snippet, along with a method we'll use later as a data source, getUser().
package com.appdynamicspilot.model;
...
public class Cart implements java.io.Serializable {
  ...
  private Double fakeAmount = 0.0;
  ...
  private User user;
  ...
  public User getUser() {
    return user;
  }
  ...
  public Double getCartTotal() {
    if (fakeAmount == 0.0) {
      double total = 0;
      if (items != null) {
        for (Item item : items) {
          total += item.getPrice();
        }
      }
      return total;
    }
    return fakeAmount;
  }
  ...
}

To set up a sample data collector for this application:

1. Identify the method using the fully qualified name of the class and getCartTotal as the method.
1. Configure the return value data source as follows:

2. Choose return value as the source type. Often, the return values are rendered as strings using the `toString()` method. Since the return value in our example is numeric (a `double`), we can use a method in the Java base packages to render the number as a `long` value:

3. Capture the user name on the invoked object. The Cart class instantiates a User object based on the following class in the same package as Cart. Notice that the User class includes a method for returning the name of the user, `getCustomerName()`.
package com.appdynamicspilot.model;
...
public class User implements java.io.Serializable {
  ...
  private String customerName = null;
  ...

  public String getCustomerName() {
    return customerName;
  }
  public void setCustomerName(String customerName) {
    this.customerName = customerName;
  }
  ...
}

Using a getter chain, you can identify this method as another data source in the same data collector. For this case, select Invoked Object as the source of the data. The getter chain getUser().getCustomerName() the operation on the invoked object:

For an HTTP data collector, the user name is sometimes available as a variable value (e.g., userid) in the HTTP query or cookie. By creating a data collector on userid, you can similarly identify affected users in transaction snapshots for slow transactions with an HTTP data collector.

5. Optionally apply a capture condition. Our sample application differentiates users by types that correspond to service levels, diamond, platinum, and so on, as shown in the following snippet of the User class:
package com.appdynamicspilot.model;

public class User implements java.io.Serializable {
    public enum CUSTOMER_TYPE {
        DIAMOND, PLATINUM, GOLD, SILVER, BRONZE;
    ...
    public CUSTOMER_TYPE getCustomerType() {
        return this.customerType;
    }
}

To capture the user name and the cart total for only gold-level users, you could add a match condition that collects data from
the Invoked Object and uses the following getter chain as the operation on the invoked object, using the condition Equals
GOLD:

6. For the PHP agent, if a method return value collected by MIDC is not stored in any variable then it is seen as null in both
snapshot and Analytics data.

When complete, transaction snapshots for slow, very slow, and stalled transactions, the transaction snapshots will include user data
that shows the user name and cart total amount for GOLD customers.

PHP HTTP Data Collector Example

This example shows how the PHP Agent can be configured to capture information from an HTTP request. Note that data
collectors cannot be added to a standalone PHP function.

In the HTTP Data Collector configuration specify the request data that you want to display in the snapshot.

This configuration captures two HTTP parameters, the URL and session ID, the cookie name, the session keys, and the headers:
In the transaction snapshots, the URL and Session ID are displayed in the SUMMARY tab, and the configured HTTP parameters, session keys, and headers in the HTTP PARAMS tab, along with the request method and response code:

The cookie name and value are shown in the COOKIES tab.
You can control the data captured in call graphs with the Call Graph Settings window, described here.

Permissions
To configure call graph settings, you need the "Configure Call Graph Settings" permission.

Call Graph Granularity
To control the granularity for call graphs, use the following settings:

- **Control granularity for Methods**: To ensure low-performance overhead, choose a threshold in milliseconds for method execution time. Methods taking less than the time specified here are filtered out of the call graphs.
- **Control granularity for SQL calls**: You can specify a threshold for SQL queries. SQL queries taking less than the specified time in milliseconds are filtered out of the call graphs. Also, see Tune Java Agent Performance.

Only SQL Capture Settings apply to call graph configuration for the Node.js agent.

To access call graph configuration, click Configuration > Instrumentation and choose the Call Graph Settings tab. There are subtabs for each application type.

Exclude Packages or Namespaces from Call Graphs
A call graph can potentially contain hundreds of methods. You can exclude packages (Java) or namespaces (.NET) with classes that you do not want to monitor.

For Java, some packages are excluded by default. These are visible in the Excluded Packages list. The packages that are excluded by default cannot be removed. However, you can include a particular sub-package from an excluded package.

You can customize call graph instrumentation from the call graph instrumentation page. From there, choose from these configuration options:

- Use the Add Custom Package Exclude (Java) or Add Custom Namespace Exclude (.NET) configuration options to exclude specific packages or namespaces from call graphs.
- Use the Add Always Show Package/Class (Java) or Add Always Show Namespace/Class (.NET) configuration options to have a package or namespace always shown in call graphs.

When the Controller constructs a call graph it uses excluded packages and included sub packages to determine which calls to include. However, the Controller includes some calls even if they are listed among the excluded packages. For example, web service calls.
SQL Capture Settings

The SQL capture settings control whether SQL statements are captured and presented in the Controller UI with dynamic parameters bound to their runtime values. For example, consider Java code that constructs a SQL call as follows:

```java
stmt = new PreparedStatement("select * from user where ssn = ?")
stmt.bind(1, "123-123-1234")
stmt.execute()
```

With the capture raw SQL option enabled, AppDynamics captures and presents the SQL call in the following form:

```
select * from user where ssn = '123-123-1234'
```

If capture raw SQL is disabled, the SQL call appears in its original form, with question mark parameters not bound to values. Disabling capture-raw-sql and using question mark parameters in SQL prepared statements gives you a mechanism for preventing sensitive data from appearing in the Controller UI.

It is important to note that the sensitive values must be parameterized in the original, prepared statement form of the SQL statement, as shown above. The following statement results in the potentially sensitive information (social security number) appearing in the Controller UI whether capture raw SQL is enabled or disabled since the sensitive data is not parameterized.

```java
stmt = new PreparedStatement("select * from user where ssn = '123-123-1234'")
```

To configure SQL capture settings, in the Call Graph Settings tab, scroll down to the SQL Capture Settings section and choose one of the following options:

- **Capture Raw SQL**: Select this option to have SQL statements that are composed as prepared statements captured with dynamic parameters bound to runtime values. By default, private SQL data and queries that take less than 10 ms are not captured.

  When you enable Capture Raw SQL in .NET environments, the agent captures the parameters for ADO.NET stored procedure calls even though the parameters are not represented by question marks. It does not capture stored procedure local variables that are not available to the CLR.

- **Filter Parameter values**: Select this option to have SQL statements that are composed as prepared statements captured without dynamic parameters bound to runtime values.
Configure JMX Metrics from MBeans

On this page:
- JMX Metric Rules and Metrics
- Permissions
- Access the MBean Browser
- Configure JMX Metric Rules
- JMX Metric Settings
- Export and Import JMX Configuration
- Disable Transaction Monitoring for JMX Monitored Nodes
- MBean Getter Chains

Related pages:
- Monitor JMX
- Exclude JMX Metrics
- JMX Topics on the AppDynamics Community

This topic describes how to create persistent JMX metrics from MBean attributes. For information on monitoring JMX metrics, see Monitor JMX.

There are many helpful JMX topics on the AppDynamics community to help you with platform-specific troubleshooting tips.

JMX Metric Rules and Metrics

You can add persistent JMX-based metrics to AppDynamics using metric rules. Once you create a persistent JMX metric, you can:

- View it in the Metric Browser
- Add it to a Custom Dashboard
- Create a health rule for it so that you can receive alerts

You can use the MBean Browser or JMX Metrics Rules Panel to create new metrics. MBean query expressions are supported.

Permissions

To configure new JMX Metrics your user account must have "Configure JMX" permissions for the application. For information about configuring user permissions for applications in Roles and Permissions.

Access the MBean Browser

If the MBean is already monitored and you want to create a metric from one of its attributes, you can do so from the MBean Browser.

To create a metric from an existing MBean attribute:

1. Open the Node Dashboard of the node that hosts the MBean, and click the MBean tab and MBean Browser subtab.
2. Expand the domains listed in the left panel and select the MBean that contains the attribute of interest.
3. In the right panel, expand the Attributes section, select the attribute, and click Create Metric.

Configure JMX Metric Rules

To add an MBean and attributes, possibly based on complex matching patterns, go to Configuration > Instrumentation and click the JMX tab.

In the JMX Metric Configurations panel, click the Java platform to add the metric to a platform group. Alternatively, create a new group at the top of the platform list and add your metric to the group.

Groups give you a way to organize metrics as it makes sense for you. For example, you may choose to keep custom metrics with their
Java platform, or put all metrics you add into a “Custom” group to differentiate them from out-of-the-box metrics.

**JMX Metric Settings**

The general settings for adding instrumentation for an MBean are:

- **The Name** is the identifier you want to display in the UI for the MBean.
- An **Exclude Rule** excludes matched MBeans that would otherwise be included by other rules. See [Exclude MBean Attributes](#).
- **Enabled** means that you want this rule to be applied.
- The **Metric Path** determines where the metric appears in the metric browser. For example, in the screenshot below, the metric path is “Web Container Runtime” for JMX metric “Tomcat_HttpThreadPools”.

In the MBeans subpanel, add matching criteria to identify the MBeans that you want to monitor.

- **The Domain name** is the Java domain. This property must be the exact name; no wildcard characters are supported.
- The **Object Name Match Pattern** is the full object name pattern. The property may contain wildcard characters, such as the asterisk for matching all the name/value pairs. For example, specifying "jmx:type=Hello,"*" matches a JMX MBean ObjectName, "jmx:type=Hello,name=hello1,key1=value1".
- **The Instance Identifier** is the MBean ID.
- The **Advanced MBean Matching Criteria** is an optional control for matching against attribute values. Identify the attribute name and value to test, along with a matching criteria, such as a substring match.

For example, the following screenshot displays the MBean matching criteria for the “GCJMXMetric” rule.

For all matched MBeans, you can define one or more metrics for the attributes of those MBeans:

- **Metric Getter Chain** Expressions can be executed against any value. In addition, getter chains for Strings and Booleans are supported using implicit conversion. See [MBean Getter Chains](#).
- **Metric Time Rollup** determines how the metric will be aggregated over a period of time. You can choose to either average or sum the data points, or use the latest data point in the time interval.
- **Metric Cluster Rollup** defines how the metric will be aggregated for a tier, using the performance data for all the nodes in that tier. You can either average or sum the data.
- **Metric Aggregator Rollup** defines how the Agent rolls up multiple individual measurements (observations) into the observation that it reports once a one minute. For performance reasons, Agents report data to the Controller at one-minute intervals. Some metrics, such as Average Response Time, are measured (observed) many times in a minute. The Metric Aggregator Rollup setting determines how the Agent aggregates these metrics. You can average or sum observations on the data points or use the current observation. Alternatively, you can use the delta between the current and previous observation.

In the following configuration, for example, the maxThreads MBean attribute is mapped to the Maximum Threads metric in the JMX metrics browser:
Export and Import JMX Configuration

After modifying the JMX configuration, you can backup or transfer the configuration using the JMX Configurations export and import icons at the top of the JMX Metric Configuration panel on the JMX instrumentation page. The configuration is exported as an XML file.

Disable Transaction Monitoring for JMX Monitored Nodes

In some circumstances, such as for monitoring caches and message buses, you want to collect JMX metrics without the overhead of transaction monitoring.

You can do so by turning off transaction detection at the entry point. For information on disabling transaction detection, see Transaction Detection Rules.

MBean Getter Chains

Getter chains in custom MBean configuration specify the method that retrieves the value of interest.

Expressions can be executed against any value. In addition to getter chain support for numeric boxed primitives (Short, Integer, Long, and so on), Strings and Booleans are supported using implicit conversion.

Boolean and String are implicitly converted to an Integer, as follows:

- Booleans are automatically converted to 0 (false) and 1 (true).
- Strings are converted to numeric values.

To illustrate how to use a getter chain, given the following class with a getSomething() method that returns the time:

```java
package com.appdynamics.flexibleapp.mbean;

public class MyComplexObject {
    public long getSomething() {
        return System.currentTimeMillis();
    }
}
```

The metric getter chain configuration would be:
MBeans

MBean Matching Criteria

- Domain: com.appdynamics.flexibleapp
- Object Name Match Pattern: com.appdynamics.flexibleapp:type=Information

Advanced MBean Matching

Find MBeans where All of the following conditions apply:

Add Condition

Attributes

Define Metrics from MBean Attribute(s)

- MBean Attribute: MyComplexObject
- Metric Name: MyComplexObject

Advanced

- Metric Getter Chain: getSomething()
- Metric Time Rollup: Average the data points

Define how this metric is aggregated over time (for example,
Exclude JMX Metrics

On this page:
- Tuning What Metrics are Gathered
- Exclude a metric

This topic describes how to exclude MBean attributes from being monitored as JMX metrics.

For background information about JMX metrics see Monitor JMX.

Tuning What Metrics are Gathered

AppDynamics provides a default configuration for certain JMX metrics. However, in situations where an environment has many resources, there may be too many metrics gathered. AppDynamics lets you exclude resources and particular operations on resources.

Exclude a metric

For example, suppose you want to exclude monitoring for HTTP Thread Pools. Follow the procedure described in Create a new JMX Metrics Rule, using the following criteria:

1. Set the Exclude Rule option to Yes.
2. Provide the Object Name Match Pattern:

   Catalina:type=ThreadPool,*

3. Provide the Advanced MBean Matching value:

   http

This configuration directs AppDynamics to stop monitoring metrics for HTTP Thread Pools. You can clear the Enabled checkbox to disable the rule.
Exclude MBean Attributes

Some MBean attributes contain sensitive information that you do not want the Java Agent to report. You can configure the Java Agent to exclude these attributes using the <exclude object-name> setting in the app-agent-config.xml file.

To exclude an MBean attribute

2. The new configuration takes effect immediately if the agent-overwrite property is set to true in the app-agent-config.xml. If agent-overwrite is false, which is the default, then the new configuration will be ignored and you have to restart the agent. Set the property to true.

   ```xml
   <property name="agent-overwrite" value="true"/>
   ```

3. Locate the JMXService section. It looks like this:

   ```xml
   <agent-service name="JMXService" enabled="true">
   ```

4. In the JMXService <configuration> section add the <jmx-mbean-browser-excludes> section and the <exclude object-name> property as per the instructions in the comment.

   ```xml
   <configuration>
   <!--
   Use the below configuration sample to create rules to exclude MBean attributes from MBean Browser.
   <exclude object-name=<MBean name pattern> attributes=< * |comma separated list of attribute names> >
   The example below will exclude all attributes of MBeans that match "Catalina:*".
   <jmx-mbean-browser-excludes>
   <exclude object-name="Catalina:*" attributes="*"/>
   </jmx-mbean-browser-excludes>
   -->
   </configuration>
   ```

5. Save the file.
JMX Logging

JMX logs contain information about JMX interactions between AppDynamics agents and the monitored server's JMX domains.

JMX log entries are useful for diagnosing problems associated with JMX metrics or with JVM metrics that rely on JMX. For example, they can help you determine why MBeans or metrics created from MBeans do not appear in the Controller UI.

The name of the log is `JMX Year_mon_day_hr_min.#.log`, where # is the log set. See Agent Log Files for information about the structure of the log files into sets.

The JMX log file can reach a maximum of 5MB within a set.

Collecting JMX log (sample):

```
<ADRRAFAppender name="JMXLogger" fileName="JMX.log">
  <PatternLayout pattern="[%t] %d{DATE} %5p - %m%n" />
  <SizeBasedTriggeringPolicy size="20 MB" />
  <ADRolloverStrategy max="5" />
</ADRRAFAppender>

<AsyncLogger name="com.singularity.JMX" level="info" additivity="false">
  <AppenderRef ref="JMXLogger" />
</AsyncLogger>
```

These configurations can be used in log4j2.xml file.
Asynchronous Transaction Demarcators

To monitor end-to-end transaction performance for an asynchronous transaction, you must identify the demarcator for the transaction's logical endpoint. There are two ways you can define a transaction demarcator:

- For Java applications, you can specify a tier on which the end to end transaction processing is completed. This lets you determine the logical transaction response time for transactions that perform asynchronous backend calls (such as JMS calls or web service calls). When the last of the threads spawned by the transaction terminates on that tier, the agent considers the transaction complete.

  The Java Agent uses patent-pending heuristics to determine when the last thread associated with the Business Transaction has run. These heuristics are successful for many common cases. In complex environments with many thread handoffs unconstrained by frameworks, these heuristics may be inaccurate. For example Reactive Java, Akka, or Scala environments.

  You should validate that the last thread on tier mechanism returns an end to end time consistent with the end to end latency shown within full transaction snapshots.

- For Java or .NET applications, you can identify the method that acts as the logical endpoint for the transaction processing sequence. For a response handler, this could be a method that watches for spawned threads to complete and when done, assembles the response and sends it back to the client. The end to end transaction time includes the time it takes for this configured method to finish processing, not when the method is invoked.

  You may specify more than one endpoint demarcator for a particular business transaction. In this case, the first match ends the transaction for purposes of end-to-end latency monitoring. Be careful not to configure multiple demarcators that could be satisfied on different tiers for a given transaction.

  For the last thread on tier demarcator type, the transaction endpoint is considered to be the time when the thread that receives the traced transaction terminates or the latest points at which any descendant threads terminate.

  For the method-based transaction demarcator option, you can configure the demarcator to consider runtime state, such as the values of parameters passed to the method. This allows you to account for an application design in which the completion of a logical business transaction is signaled by the value of a method parameter or return value.

To create the configuration:

1. Open the Configuration > Instrumentation page and choose Asynchronous Transactions from the top menu. You likely need to expand the menu list (>>) to view the Asynchronous Transactions item.
2. Click Add and choose the demarcator method:
   - Java only: Transaction is complete when last thread on specified Tier is finished running.
   - Java or .NET: Transaction is complete when specified class/method is invoked.
3. Follow the instructions that appear onscreen for the option you choose. Note the following points:
   - You configure the class/method option using the standard method selector. See Configure Instrumentation for more information about identifying classes and methods in the AppDynamics configuration.
   - For the last thread on the tier approach, first, give a name for the transaction configuration and choose the tier.
4. Choose the business transaction where the configuration applies.

   The threads in the logical transaction processing flow must be traceable by AppDynamics, including the thread that contains the end-point method. If needed, configure custom thread correlation to ensure that all threads are properly traced. For more information on thread correlation, see Threading and the Java Agent and Thread Correlation for .NET.
Automatic Instrumentation of Specialist Packages/Frameworks

### Introduction

One of the goals of the AppDynamics Application Performance Management (APM) Platform is to provide maximum visibility into Enterprise IT systems for the minimum of set-up effort.

To achieve this, AppDynamics configures its agents to place instrumentation on entry points, exit points, and thread hand-offs right out of the box. The user interface facilitates configuring naming rules for transactions and backends.

Within this documentation, you can find pages about Supported Environments, Transaction Detection, and Backend Detection for many common frameworks (including servlet, JAX-RS, and JDBC) and many HTTP clients.

In addition, the following pages describe the specialized out-of-the-box configuration for instrumenting off-the-shelf products, packages, or frameworks whose entry and exit points are too complex for the more general documentation:

- IBM-BPM Support
- Spring Batch Support
- OSB Support
- Open Tracing Support
IBM® Business Process Manager (IBM BPM) is a comprehensive business process management platform. It provides tools to enable authoring, testing, and deployment of business processes, as well as management capabilities. For more information, see IBM documentation.

IBM BPM version 8.5.7 is used to define and execute Business Process Definitions (BPDs) which combine elements of human interaction Client-Side Human Services (CSHS) or ‘user tasks’ for example, manual loan approvals and so on with elements of system integration system tasks, for example, registering a loan approval in a CRM system and so on. These processes are critically important and it is, therefore, natural to use AppDynamics Business Transactions to monitor the system integration elements of the processes to ensure performance and reliability of these technical integrations. You can also use the AppDynamics Business Journeys capability to monitor the end to end progress of the processes, including the elements of human interaction.

AppDynamics Instrumentation Use Cases

In order to facilitate these monitoring use cases, the AppDynamics agent comes with out-of-the-box configuration necessary to:

- Appropriately start and name Business Transactions originated within IBM BPM to allow management of system tasks with App iQ APM
- Correlate the processing within IBM BPM to facilitate end-to-end tracing of these Business Transactions
- Collect sufficient process metadata to allow these Business Transactions to be used within Business Journeys to provide visibility into the progress of processes including the human interactions

Agent Installation and Setup

IBM BPM runs in WebSphere, see IBM WebSphere and InfoSphere Startup Settings for more details.

Business Process Definitions: System and User Tasks

A BPD process consists of the flow of a business process which may involve user interactions. It consists of User Tasks, System Tasks, and other logic.

The screenshot displays a sample process within the BPD designer:

User tasks, in turn, break down into one or more user interactions, such as submit a form and approve a decision. They can be implemented as a CSHS, one of which is shown below:
A CSHS always starts with a start event and consists of one or more activities each of which uses a system service as the underlying implementation, and one or more Coaches, which define the user interactions.

Business Transaction Support

**Business Transaction Support for Client Side Human Services**

**CSHS Transaction naming**
All Business Transactions detected within a CSHS are detected as servlet type transactions, and their naming can be
Since the Business Transaction is designed to monitor technical activity only, one CSHS may spawn multiple Business Transactions:

Any element inside a CSHS is associated with the following identifiers:
- Project Name
- BPD Process Name
- User Task Name
- CSHS Process Name
- Activity Name or Coach Name

Additionally, for Activity (Service), the activity has a service as its implementation which has its own identifier (Service Name).

These are annotated on screenshots of the IBM design tools:

**BPD Screen**
Within a CSHS, there can be one or more activity-services involved in the flow. Each of this hits a specific REST URL. Hence, an HTTP entry point is defined for each of these services.
For example, the default URL for the **Start BT** –
/teamworks/process.lsw

For example, the default URL for the **End BT** –
/teamworks/cfecontroller

Business Transactions originating from the start of a user task are detected as servlet Business Transactions and are named according to the following scheme:

/<Project Name>/<BPD Name>/<Task Name>/<CSHS Name>/Start:Event

For example (corresponding to the images shown above, BPD image and CSHS image), the Business Transactions would be named –

/My Hiring Sample/Standard HR Open New Position/Step: Submit job requisition/Client-Side Http Service/Start:Event
/My Hiring Sample/Standard HR Open New Position/Step: Submit job requisition/Client-Side Http Service/End:Event

**Business Transaction Detection and Naming at an Activity (Service) Node Within a CSHS**

Each user task in the BPD process has a CSHS as its implementation. Within a CSHS, there can be one or more activity-services involved in the flow. Each of them hits a specific REST URL. Hence, HTTP entry point is defined for each of these services.

For example, the default URL is –

/rest/bpm/wle/v1/coachflow/service/1.a22ce595-1573-4fdd-90f6-734884753610

Business Transactions originating within a CSHS activity service are detected as servlet Business Transactions and are named according to the following scheme:

/<Project Name>/<BPD Name>/<Task Name>/<CSHS Name>/<Activity Name>/<Implementation Service Name>:Service

For example (corresponding to the images BPD image and CSHS image), the Business Transactions would be named:
Business Transaction Detection at the CSHS Coach

A User Task in a BPD process has a CSHS as its implementation. Within a CSHS, there can be one or more UI Coach(es) involved in the flow. Each UI Coach represents the UI page(s) which opens up for the user to perform some actions (like completing a form, approving something, and so on). Each of these hits a specific REST–URL. Hence, the servlet entry point is used for the naming of the UI Coach.

For example, the default URL for this Business Transactions is –

/My Hiring Sample/Standard HR Open New Position/Step: Submit job requisition/Client-Side Http Service/Activity1/HttpClient IService:Service

Business Transaction Naming Scheme at Coach of CSHS:

The Business Transactions originating on entry to a UI coach are detected as Servlet Business Transactions –

/Project Name/BPD Name/Task Name/CSHS Name/Coach

For example (corresponding to the images BPD image and CSHS image), the business transaction would be named:

/My Hiring Sample/Standard HR Open New Position/Step: Submit job requisition/Client-Side Http Service/UI Coach:Coach

In-Process Correlation Between Elements within a CSHS

There can be more than one element involved within a User Task (CSHS) with no user interaction in between them. These form a part of the same business transaction. Service Endpoints give visibility of the individual steps within the resulting business transaction as illustrated below:
**Business Transaction Support for System Tasks within a BPD Process**

POJO business transactions are generated for System Tasks within a BPD process.

The POJO business transactions are named using four identifiers:

1. Project Name (<project-name>)
2. BPD Process Name (<bpd-name>)
3. System Task Name (<task-name>)
4. Implementation Process Name (<implementation-name>)

If any identifier's value is not available, it is replaced with `<UNKNOWN>` in the resulting transaction name.

For example, a business transaction detected for default configuration (project-name / bpd-name / task-name / implementation-name) is:

```
<implementation-name>
```

The default naming scheme can be customised using the `ibmbpm-systemtask-bt-naming` node property.

The property value is comma-separated identifiers chosen from project, bpd, task, implementation, and is order sensitive.

For example if,

- `value = "project, task"` means POJO Business Transactions are named as: `<project-name>/<task-name>`
- `value = "task, project"` means POJO Business Transactions are named as: `<task-name>/<project-name>`
- `value = "none"` means POJO Business Transactions are not detected (disabled)
value = "default" means all the identifiers are used for Business Transactions naming in a default order, that is, they are named as <project-name>/<bpd-name>/<task-name>/<implementation-name>

Set ibmbpm-systemtask-bt-naming property to "none" to disable detection of Business Transactions for system tasks.

Built-in Data Collectors

<table>
<thead>
<tr>
<th>PROCESS NAME</th>
<th>TASK ID</th>
<th>TASK NAME</th>
<th>BPD INSTANCE ID</th>
<th>BPD NAME</th>
<th>PROJECT NAME</th>
</tr>
</thead>
</table>

In addition to these, for a CSHS implementation of a User Task, if the IBM BPM specific naming Business Transactions naming scheme is enabled, an additional data collector, **ACTUAL URL**, will also be present, populated with the actual URL hit when the business transaction is initiated.

Additional data collectors defined for a User Task depending upon the element of the CSHS being called can be:

- COACH NAME for Coach
- EVENT TYPE for Start and End (value is either Start or End depending upon on the event)
- ACTIVITY NAME for Activity (Service)
- SERVICE NAME for Activity (Service – denoting the implementation service of the activity)

Sample Data Collectors

You can view the snapshot data collectors as –

- Data Collectors for System Task

- Data Collectors for Start Event of CSHS in User Task
- Data Collectors for Activity (Service) Element of CSHS in User Task

- Data Collectors for Coach Element of CSHS in User Task
**Other Configuration Notes**

There are some other configuration settings that may be helpful when implementing AppDynamics for IBM BPM monitoring:

- Increase the value of the node-property `max-business-transactions` if necessary, to accommodate the number of tasks involved in the BPD process (the default = 50).
- Add a node property `framework-support` with a value of “none” to disable CometD and GWT POJO Business Transactions a large number of which otherwise get detected. These correspond to activity within the IBM tooling and are not valuable for BPM monitoring.
- Exclude Servlet Business Transactions like `/portal/login`, `/webasset/`, which are not valuable for BPM monitoring.

**Node properties that control IBM BPM Instrumentation**
Node Properties available when using IBM-BPM support:

- `disable-ibmbpm-usertask-bt-naming`: Sets whether Business Transactions naming scheme for IBM-BPM UserTask Business Transactions should be disabled (value = true) or enabled (value = false).
  - When it is set to "true", the Business Transactions would be named as per the default URL and not the meaningful names.
- `disable-ibmbpm-usertask-bt-in-process-correlation`: Sets whether Business Transactions in-process correlation for IBM-BPM UserTask Business Transactions should be disabled (value = true) or enabled (value = false).
- `disable-ibmbpm-data-collectors`: Sets whether data-collectors for IBM-BPM task Business Transactions should be disabled (value = true) or enabled (value = false).
Spring Batch Support

Introduction

Spring Batch is an open source project for batch processing – execution of a series of jobs.
A Batch Job is composed of one or more Steps. Each step consists of a Tasklet (explicitly or implicitly using ItemReader and ItemWriter). Each level (Job, Step and Tasklet) has an execute method.
Step execution can be parallelised using multiple threads. Remote Chunking (processing of part of Step, chunk on a remote JVM) can also be done with the help of JMS.
Different steps in a job can be executed in separate threads use-case permitting.
From an APM perspective, tracking batch workloads at the Job level brings limited value, since batch Jobs are usually very long-running and an alert that the Job overran would come too late for remedial action to be taken.
For this reason, the AppDynamics Business Transactions should be placed at the Tasklet level, meaning one batch Job will be composed of many Tasklet Business Transaction executions.

Turn off spring batch instrumentation using the spring-batch-enabled node property.

Alerting on the response times of these Business transactions allows AppDynamics users to:

- Proactively detect that a Batch Job is progressing more slowly than usual, and thus predict if it will overrun any batch window
- Troubleshoot the root-cause of any unusual slow-downs in the progress of batch jobs in order that the issue can be remediated before the Job overruns

To achieve this, from release 4.5, the Java Agent:

- Creates BTs at Tasklet.execute, named after the Job, Step, and Tasklet
- Correlates chunks across thread boundaries or process boundaries via JMS
OSB Support

Introduction

Oracle Service Bus (OSB) is Oracle's Enterprise Service Bus (ESB) implementation. It is configuration-based and policy-driven and provides reliable service-oriented integration, service management, and traditional message brokering across heterogeneous IT environments.

This service-infrastructure software adheres to the SOA principles of building coarse-grained, loosely coupled, and standards-based services. Additionally, OSB acts as a message brokering, service monitoring, administration, dynamic routing, and message transformation layer to infrastructure.

OSB relies on Oracle WebLogic Server run-time facilities. OSB supports various service types and transports.

From agent release 4.5.8, Appdynamics supports Business Transaction entries and exits for the HTTP and JMS transports and correlates activity through the broker to give end-to-end transaction visibility. Outbound flows using other transports will be shown as uncorrelated custom backends on the flowmap. This behavior can be controlled using the osb-ignore-exit-types agent property.

For Business Transactions originating in OSB itself, the transaction naming scheme that will be used is dependent on the OSB Proxy Service Type.

<table>
<thead>
<tr>
<th>Service Type</th>
<th>Naming Scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>REST</td>
<td>WebService Entry</td>
</tr>
<tr>
<td>WSDL, Messaging, SOAP, XML</td>
<td>Servlet Entry</td>
</tr>
</tbody>
</table>

For more details on proxy service types and transports in OSB, refer to the Oracle documentation.
Open Tracing Support

On this page:
- Prerequisite
- Getting The AppDynamics OpenTracer
- Enabling or Disabling the Tracer
- Plugging the Tracer into Lightbend Telemetry

OpenTracing Overview

OpenTracing is an open standard for Tracing in-process and out-of-process distributed transactions. At a high level, it performs shallow stitching of method executions involved in a business flow, and hence enables measuring it end to end.

AppDynamics Tracer is an implementation based on version 0.31.0 of the OpenTracing standards.

Prerequisite

For the OpenTracer to work correctly, use the Executor strategy in the Java agent. You can specify it in app-agent-config.xml or as node property:

```xml
<!--As part of app-agent.config.xml-->
<property name="async-instrumentation-strategy" value="executor"/>
```

Or, just specify the following node property in the Controller:

```
async-instrumentation-strategy = "executor"
```

Getting The AppDynamics OpenTracer

**Installing the Dependency**

The OpenTracer jar can be accessed directly, or downloaded from Maven Central, or it can be downloaded from the AppDynamics portal. The library version changes with each new OpenTracer release, and is not tightly coupled to the version of the underlying agent, which must be a minimum of version 4.5.13.

```
Gradle
dependencies {
    compile group: 'com.appdynamics.agent', name: 'opentracer', version: '4.5.13.27526'
}
```
Enabling or Disabling the Tracer

By default when plugged in, OpenTracer is enabled. There are two ways to disable it:

- Programmatically you can enable and/or disable using:

  ```java
  AppdynamicsTracerConfiguration.appdTracingConfiguration().setTracingEnabled();
  ```

  Optionally,

- At startup, you can specify the system property to enable and/or disable using:

  ```
  -Dappdynamics.opentracing.enabled=false
  ```

Plugging the Tracer into Lightbend Telemetry

The first OpenTracing use case validated by AppDynamics is its use with Lightbend telemetry (also known as the Cinnamon agent), which allows transaction tracing through applications built on the Lightbend reactive platform, for example those built on Akka HTTP. For more details, refer to the Lightbend documentation, in particular as it relates to OpenTracing integration.

Full instructions as to how to set up Lightbend Telemetry are beyond the scope of the AppDynamics documentation. At a high level, the necessary steps for configuration are:

1. Configure the Lightbend telemetry agent
   For Lightbend instructions, see Instructions.
2. Enable tracing for Akka HTTP endpoints you wish the AppDynamics agent to trace
   For Lightbend instructions, see OpenTracing Configuration.

Ensure the active sampler is set to `const-sampler` in your `cinnamon.opentracing` file. (The low-overhead nature of the AppDynamics Java agent means that you can safely ignore the comment in the sample configuration.)
3. Plug the AppDynamics OpenTracing implementation into Lightbend Telemetry as a custom OpenTracing tracer using the AppdynamicsTracerFactory as described in the Lightbend documentation. A sample AppDynamics tracer factory is provided below:

```java
import com.appdynamics.opentracing.core.AppdynamicsTracerFactory;
import com.lightbend.cinnamon.opentracing.TracerFactory;
import io.opentracing.Tracer;

public class AppdynamicsTracerPlugin implements TracerFactory {
    @Override
    public Tracer create() {
        return AppdynamicsTracerFactory.getTracer();
    }
}
```
Development Level Monitoring

What is Development Level Monitoring?

AppDynamics monitors your production applications in a way that is designed to provide maximum visibility with the least amount of overhead.

This default mode, called production level monitoring, applies limits on the retention of certain types of information. In a testing environment or other environments where overhead is not an issue, you can enable development level monitoring. Development mode is intended for temporary use while setting up AppDynamics or for occasional application troubleshooting.

You apply development monitoring to a specific business transaction and originating node combination. An originating node is one or more of the nodes that serve as the entry points for the transaction.

Transactions originating on an enabled node are subject to development level monitoring on downstream nodes that participate in processing that business transaction. Note that this applies for that business application only, however. Processing for a continuing transaction in another business application is not monitored at the development level unless you have enabled development level monitoring at the continuing business application as well.

While development monitoring increases the retention of call graphs and SQL statement capture, certain limits still apply. The limits specify maximum thresholds for calls per minute, Java heap utilization, and snapshot segments generated across the Controller. If the thresholds are exceeded, development level monitoring is disabled. See Development Level Monitoring Limits for more information.

Effects of Development Mode

Enabling development level monitoring affects the capture of the following information:

- **Exit Calls**: AppDynamics increases information collected for exit calls to backend systems. For database backends, the agent collects all SQL statements without per-transaction limits. The agent also collects all JDBC and ADO.NET calls attached to methods, even when the call duration is less than 10 ms.
- **Snapshots**: The agent attempts to take a snapshot for every transaction, ignoring the values for the following agent node properties:
  - max-concurrent-snapshots
  - on-demand-snapshots

  Development mode does not guarantee snapshots for every transaction since it was not designed to capture data on every request.

- **Call Graphs**: The agent captures full call graphs in development mode.
Required Permissions

Users must belong to a role with the "Configure Monitoring Level (Production/Development)" permission for the application to set development level monitoring. See Roles and Permissions for more information.

Enable Development Level Monitoring

You enable development level monitoring in the context of a particular business transaction/originating node combination. To avoid exceeding development mode limits described in Development Level Monitoring Limits, it is recommended that you enable development monitoring at the smallest scope possible.

To enable development level monitoring:

1. In the business application for which you want to enable monitoring, click Configuration > Development Level Monitoring.
2. Enable the Development Level Monitoring switch at the top of the page.
   For agents earlier than version 4.3, enabling development level monitoring applies globally, rather than for a particular business transaction. For 4.3 agents and higher, you need to configure the business transaction and node to which the configuration applies, as described in the following step.
3. Select the Business Transaction for which you want to enable development monitoring.
4. Click the Enabled checkbox next to the originating nodes on which you want to enable development level monitoring.

The following screenshot shows the development level monitoring controls:

![Development Level Monitoring Controls](image)

Disable Development Level Monitoring

Even in pre-production environments, enabling development level monitoring would normally be a temporary measure done in select cases. When finished, disable development level monitoring for the business application, as described here.

To disable development level monitoring:

1. In the business application for which you want to disable development level monitoring, click Configuration > Development Level Monitoring.
2. Use the Development Level Monitoring switch at the top of the page to disable development level monitoring. The switch disables such monitoring for all business transactions in the business application.

Development mode may also be disabled automatically, as described in the next section.
Development Level Monitoring Limits

The following agent-applied and Controller-applied limits prevent excessive resource consumption during development mode.

App Agent Limits

For an app agent, the following limits apply:

- For each node, a maximum of 500 calls per minute. (See dev-mode-suspend-cpm.)
- For each JVM, the maximum heap utilization percentage of 90%. (See heap-storage-monitor-devmode-disable-trigger-pct.)

If a limit is exceeded, an agent log event is generated indicating that development mode has been disabled because limits were exceeded.

Controller Limits

The Controller monitors the number of snapshot segments collected during development monitoring across business transactions. A snapshot segment corresponds to the processing activities for a business transaction on a particular tier. A single business transaction that traverses many tiers can generate more data than several business transactions that only traverse a few tiers.

When the threshold of 5000 snapshot segments per minute is exceeded, the Controller disables development mode for the top contributing business transactions. The Controller disables development mode on as many business transactions as are needed to reduce the snapshot segment contribution by 20%.

For example, if a single business transaction is accountable for 25% of the snapshot segments when the threshold is exceeded, only it is disabled. As another example, given the following business transactions and the percentage of snapshot segments, they each contributed when the threshold was exceeded, the first two are disabled:

1. BT1: 17%
2. BT2: 16%
3. BT3: 15%
4. BT4: 15%
5. BT5: 13%
6. BT6: 12%
7. BT7: 12%

This Controller limit is not configurable.
App Server Agents Supported Environments

On this page:
- Java Agent Supported Platforms
- .NET Agent Support
- Node.js Agent Support
- PHP Agent Support
- Python Agent Support
- Apache Server Agent Support
- C/C++ Agent Supported Platforms
- Go SDK Support
- IIB Agent Support

This page provides an aggregated view of the supported environments for the app server agents.

Java Agent Supported Platforms

In the following tables, note that:
- A dash (“-“) in a table cell indicates that this column is not relevant or not supported for that particular environment.
- In cases where no version is provided, assume that all versions are supported. Contact AppDynamics Sales for confirmation.
- For environments that require additional configuration, a separate table describing or linking to configuration information follows the support matrix.
- For environments supported by AppDynamics End User Monitoring, see Supported Environments and Versions - Web EUM.
- For environments supported by AppDynamics Server Visibility, Standalone Machine Agent Requirements and Supported Environments.

JVM Support

The AppDynamics Java Agent uses the standard JVM Tool Interface (JVMTI) mechanism allowing it to instrument any software running on a JVM supporting this mechanism.

AppDynamics certifies the successful operation of the basic mechanisms of instrumentation used by the agent on the following Java runtimes. These capabilities are supported on both JRE or full JDK installations.

Where the agent supports the following advanced memory monitoring features, they are listed for the JVM: Object Instance Tracking (OIT), Automatic Leak Detection (ALD), Content Inspection (CI), and Access Tracking (AT).

<table>
<thead>
<tr>
<th>JVM</th>
<th>OS</th>
<th>Memory Monitoring Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon Corretto 8, 11</td>
<td>Linux, Windows</td>
<td></td>
</tr>
<tr>
<td>Azul Zing 15.x.</td>
<td>Linux x64</td>
<td>OIT, ALD</td>
</tr>
<tr>
<td>Azul Zulu 1.6, 1.7, 1.8, 9, 10, 11</td>
<td>Linux x64</td>
<td>OIT, ALD</td>
</tr>
<tr>
<td>JDK11 is supported from 4.5.6 onwards</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HP OpenVMS</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
IBM JVM 1.6.x, 1.7.x, 1.8.x

Object instance tracking, automatic leak detection, and custom memory structure monitoring are not supported with the AppDynamics IBM Java Agent. IBM JVMs can be instrumented with the AppDynamics Sun Java Agent to work around this limitation, however this only enables automatic leak detection and custom memory structure monitoring. Object instance tracking is not available. Working around this limitation can result in negative performance impact and is not recommended. In such cases, the IBM JVM needs to be restarted to enable custom memory structure monitoring.

Oracle Rockit JVM 28.1+

Linux
Intel 64,
Windows

Oracle/BEA JRockit 1.6

Oracle/Sun JVM 1.6, 1.7, 1.8, 9, 10, 11

Windows, Linux

OIT, ALD, CI, AT

Content Inspection and Access Tracking require a JVM restart.

Oracle/Sun JVM 1.6, 1.7, 1.8, 9, 10, 11, 12

Solaris Sparc 64, Windows, Linux

OIT, ALD, CI, AT

Content Inspection and Access Tracking require a JVM restart.

SAP JDK 6+

Windows, Solaris, Linux, HP-UX, i5/OS, AIX

JVM Application Server and Framework Support

AppDynamics supports the use of the Java Agent to instrument any application component running on a supported JVM, irrespective of how that component is built. The power of the AppDynamics platform is that it can automatically discover the topology and behavior of complex enterprise applications without requiring deep technical knowledge of the application's underlying code.

Frequently, Java-based systems employ standard framework code to implement business logic. Automatic instrumentation of framework code relies on knowledge of the business logic and programming patterns employed by the framework. AppDynamics instrumentation targets processing hand-offs between components, called entry points and exit points, either within the JVM or between JVMs. This includes hand-offs between frameworks in cases where multiple frameworks are being used together. This section covers the capabilities for frameworks for which AppDynamics provides automatic detection rules.

Monitoring application components built using frameworks not listed here may require custom configuration. The custom configuration may involve, for example, custom POJO entry or exit points. If you understand how the application behaves internally, you can easily configure this type of instrumentation. For more complex configuration tasks, contact your account representative to discuss how to engage the AppDynamics customer success organization.
JVM Language Frameworks Support

No additional configuration is required for these frameworks.

<table>
<thead>
<tr>
<th>Vendor</th>
<th>JVM Language Framework</th>
<th>Version</th>
<th>Correlation/Entry Points</th>
<th>Exit Points</th>
<th>Transports</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open Source</td>
<td>Akka Actor</td>
<td>2.1 – 2.5.x</td>
<td>Yes</td>
<td>Yes</td>
<td>Netty</td>
<td>4.3.1 required for 2.4.x  2.5x support includes Persistence  Remoting exit/entry supported</td>
</tr>
<tr>
<td>Open Source</td>
<td>Akka HTTP</td>
<td>Akka Actor 2.5.x  Akka HTTP upto 10.1.5  Scala 2.11, 2.12</td>
<td>Yes</td>
<td>Yes</td>
<td>HTTP</td>
<td>EUM is supported</td>
</tr>
<tr>
<td>Open Source</td>
<td>Groovy</td>
<td>-</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Open Source</td>
<td>Play for Scala  Play for Java</td>
<td>2.1 – 2.6  Scala 2.11, 2.12</td>
<td>Yes</td>
<td>-</td>
<td>HTTP over Netty server  Akka HTTP server</td>
<td>Includes framework specific entry and exit points  Play EUM-APM correlation supported</td>
</tr>
<tr>
<td>Open Source</td>
<td>Scala</td>
<td>2.11.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Open Source</td>
<td>Spray toolkit (Spray.io)</td>
<td>1.1.x  1.1.3</td>
<td>Yes</td>
<td>Yes</td>
<td>HTTP</td>
<td>Entry points are detected and configurable as servlet entry point and exit points as HTTP exits</td>
</tr>
<tr>
<td>Pivotal</td>
<td>Grails</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

Java Frameworks Support

The Java Agent supports these Java frameworks. Some require additional configuration as indicated in the Configuration Notes column.

<table>
<thead>
<tr>
<th>Vendor</th>
<th>Framework</th>
<th>Version</th>
<th>SOA protocol (WebServices)</th>
<th>Auto Naming</th>
<th>Entry Points</th>
<th>Exit Points</th>
<th>Detection</th>
<th>Configuration Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adobe</td>
<td>BlazeDS</td>
<td>-</td>
<td>HTTP and JMS adaptor</td>
<td>-</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
<td>Example Message Queue Backend Configuration</td>
</tr>
<tr>
<td>Adobe</td>
<td>ColdFusion</td>
<td>8.x, 9.x</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
<td>Configuration required for transaction discovery. See:  - Java Business Transaction Detection  - Servlet Entry Points</td>
</tr>
<tr>
<td>Apache</td>
<td>Cassandra with Thrift framework</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>Yes</td>
<td>Apache Thrift Entry and Exit points are detected</td>
<td></td>
</tr>
<tr>
<td>Apache</td>
<td>Struts</td>
<td>1.x, 2.x</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>-</td>
<td>Struts Actions are detected as entry points; struts invocation handler is instrumented</td>
<td>Struts Entry Points</td>
</tr>
<tr>
<td>Developer</td>
<td>Framework</td>
<td>Version</td>
<td>Protocol</td>
<td>Business Transaction Detection</td>
<td>Servlet Entry Points</td>
<td>Default Behavior</td>
<td>Notes</td>
<td></td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td>---------</td>
<td>----------</td>
<td>-------------------------------</td>
<td>----------------------</td>
<td>------------------</td>
<td>-------</td>
<td></td>
</tr>
</tbody>
</table>
| Apache     | Tapestry    | 5       | -        | Yes                           | -                    | Not by default   | See:  
  - Java Business Transaction Detection  
  - Servlet Entry Points |
| Apache     | Wicket      | -       | -        | No                            | Yes                  | Not by default   | See:  
  - Java Business Transaction Detection  
  - Servlet Entry Points |
| Apple      | WebObjects  | 5.4.3   | HTTP     | Yes                           | Yes                  | Yes              | Apple WebObjects Startup Settings |
| axonframework.org | Axon    | 2.x, 3.x | -        | Commands on the Command Bus continue existing Business Transactions | Correlation for Distributed Command Bus on JGroups and for Spring Cloud Connector transport as an exit | - | - |
| Open Source | CometD      | 2.6     | HTTP     | Yes                           | Yes                  | -               | See also "HTTP Exit Points" on Java Backend Detection. |
| Eclipse    | RCP (Rich Client Platform) | -       | -        | -                             | -                    | -               | - |
| Google     | Google Web Toolkit (GWT) | 2.5.1   | HTTP     | Yes                           | Yes                  | -               | - |
| JBoss      | JBossWS Native Stack | 4.x, 5.x | Native Stack | -                             | -                    | -               | - |
| IBM        | IBM-BPM     | 8.5.7, 8.6 | -       | Yes                           | Yes                  | Yes              | IBM-BPM Support |
| Open Source | Direct Web Remoting (DWR) | -       | -        | -                             | -                    | -               | - |
| Open Source | Eclipse Vert.x Core | 3.3.3-3.5.4 | HTTP | Yes                           | Yes                  | Yes              | EUM Correlation is supported |
| Open Source | Enterprise Java Beans (EJB) | 2.x, 3.x | -        | Yes                           | -                    | -               | EJB Entry Points |
| Open Source | Grails      | -       | -        | Yes                           | -                    | -               | Not by default |
| Open Source | Hibernate JMS Listeners | 1.x       | -        | -                             | -                    | -               | - |
| Open Source | Java Abstract Windowing Toolkit (AWT) | -       | -        | -                             | -                    | -               | - |
| Open Source | Java Server Faces (JSF) | 1.x, 2.x | -        | Yes                           | Yes                  | -               | Java Business Transaction Detection and Servlet Entry Points |
| Open Source | Java Server Pages | 2.x       | -        | Yes                           | -                    | -               | Servlet Entry Points |
| Open Source | Java Servlet API | 2.x, 3.0 | -        | -                             | -                    | -               | - |
| Open Source | Jersey      | 1.x, 2.x | REST, JAX-RS | Yes                           | Yes                  | No              | Not by default JAX-RS Support and node properties:  
  - rest-num-segments  
  - rest-transaction  
  - rest-uri-segment-scheme  
  See App Agent Node Properties Reference for information on the properties. |
### Application Servers

The Java Agent supports the following application servers. Some require additional configuration. Click the link on the server or OSGi Runtime for information about additional requirements or related configuration topics. The agent usually discovers application servers as an entry point.

<table>
<thead>
<tr>
<th>Vendor</th>
<th>Application Server / OSGi Runtime</th>
<th>Version</th>
<th>SOA Protocol</th>
<th>RMI Supported</th>
<th>JMX</th>
<th>Entry Points</th>
<th>Configuration Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adobe</td>
<td>Cold Fusion</td>
<td>8.x, 9.x</td>
<td>-</td>
<td>No</td>
<td>-</td>
<td>Yes</td>
<td>Requires configuration for transaction discovery; see Servlet Entry Points</td>
</tr>
<tr>
<td></td>
<td>Equinox</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>OSGi Infrastructure Configuration</td>
</tr>
<tr>
<td>Apache</td>
<td>Felix</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>OSGi Infrastructure Configuration</td>
</tr>
<tr>
<td>Apache</td>
<td>Sling</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>OSGi Infrastructure Configuration</td>
</tr>
<tr>
<td>Apache</td>
<td>Tomcat</td>
<td>5.x, 6.x, 7.x, 8.x, 9</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>Apache Tomcat Startup Settings</td>
<td></td>
</tr>
<tr>
<td>Apache</td>
<td>Resin</td>
<td>1.x - 4.x</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Resin Startup Settings</td>
</tr>
<tr>
<td>Eclipse</td>
<td>Jetty</td>
<td>6.x, 7.x, 8.x, 9x</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Jetty Startup Settings</td>
</tr>
<tr>
<td>IBM</td>
<td>InfoSphere</td>
<td>8.x</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>IBM WebSphere and InfoSphere Startup Settings</td>
</tr>
<tr>
<td>IBM</td>
<td>WebSphere</td>
<td>6.1, 7.x, 8.x, 9.x</td>
<td>JAX-WS</td>
<td>Yes, detect and correlate</td>
<td>Yes for WebSphere PMI</td>
<td>Yes</td>
<td>IBM WebSphere and InfoSphere Startup Settings</td>
</tr>
<tr>
<td>Open Source</td>
<td>Liferay Portal</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Open Source</td>
<td>JBoss Wildfly (formerly JBoss Application Server) Including versions shipped with JBoss EAP under JBoss Wildfly</td>
<td>4.x to 14.x</td>
<td>-</td>
<td>Yes</td>
<td>-</td>
<td>JBoss and Wildfly Startup Settings</td>
<td></td>
</tr>
<tr>
<td>Sun/Oracle</td>
<td>GlassFish Enterprise Server</td>
<td>2.x</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>Yes</td>
<td>GlassFish Startup Settings</td>
</tr>
<tr>
<td>Oracle</td>
<td>GlassFish Server and GlassFish Server Open Source Edition</td>
<td>3.x, 4.x</td>
<td>-</td>
<td>-</td>
<td>Yes for AMX</td>
<td>Yes</td>
<td>GlassFish Startup Settings</td>
</tr>
</tbody>
</table>

Open Source | JRuby HTTP | - | - | - | Yes | - | Not by default | See: Java Business Transaction Detection Servlet Entry Points Node property to disable Netty instrumentation: netty-enabled, by default it is true |
Open Source | Netty 3.x | HTTP | Yes | Yes | Yes | - | See: Java Business Transaction Detection Servlet Entry Points Node property to disable Netty instrumentation: netty-enabled, by default it is true |
Open Source | Spring Annotated Web Services 2.x+ | HTTP | Yes | Yes | No | - | See: Java Business Transaction Detection Servlet Entry Points Node property to disable Netty instrumentation: netty-enabled, by default it is true |
Open Source | WebSocket 1.0 (Java EE 7, JSR-356) | - | Yes, BT Naming not configurable | Yes, correlation not supported | Yes | Detection is automatic | Node property: websocket-ent ry-calls-enabled |
Oracle | Coherence with Spring Beans 2.x, 3.x | - | - | - | - | - | - |
Oracle | Swing (GUI) | - | - | - | - | - | - |
Oracle | WebCenter 10.0.2, 10.3.0 | - | - | - | - | - | - |
Spring | Spring MVC | - | - | - | Yes | - | Not by default | See App Agent Node Properties Reference. |
Oracle and BEA

<table>
<thead>
<tr>
<th>Software AG</th>
<th>WebLogic Server</th>
<th>9.x+</th>
<th>JAX-WS</th>
<th>Yes, detect and correlate for 10.x</th>
<th>Yes</th>
<th>Yes</th>
<th>Oracle WebLogic Startup Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>webMethods</td>
<td>9.5, 9.6</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>webMethods Startup Settings</td>
</tr>
<tr>
<td>Tibco</td>
<td>ActiveMatrix BusinessWorks Service Engine</td>
<td>5.x, 6.x</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>Tibco ActiveMatrix BusinessWorks Service Engine Settings</td>
</tr>
<tr>
<td>Application Server (OCAJ)</td>
<td>-</td>
<td>-</td>
<td>Yes, detect and correlate for 10.x</td>
<td>-</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Grails, with Tomcat 7.x, Glassfish v3, Weblogic 12.1.1 (12c)

Servlet 3.x detection is not supported.

PaaS Providers

<table>
<thead>
<tr>
<th>PaaS Provider</th>
<th>Buildpack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pivotal Cloud Foundry</td>
<td>Java Buildpack 3.4 and higher</td>
</tr>
<tr>
<td></td>
<td>See Using AppDynamics with Java Applications on Pivotal Cloud for more information.</td>
</tr>
<tr>
<td>Red Hat OpenShift 3</td>
<td>JBoss EAP 6.4 and WildFly 8.1 Docker images</td>
</tr>
<tr>
<td></td>
<td>For documentation and download information, see the AppDynamics Java APM Agent page on the Red Hat Customer Portal.</td>
</tr>
</tbody>
</table>

Message Oriented Middleware Support

The Java Agent supports the following message oriented middleware environments. Some require additional configuration as indicated in the Configuration Notes column. Message oriented middleware servers are usually found by the Java Agent as an entry point.

<table>
<thead>
<tr>
<th>Vendor</th>
<th>Messaging Server</th>
<th>Version</th>
<th>Protocol</th>
<th>Correlation/Entry Points</th>
<th>Exit Points</th>
<th>JMX</th>
<th>Configuration Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon</td>
<td>Simple Queue Service (SQS)</td>
<td>-</td>
<td>-</td>
<td>Yes (correlation only)</td>
<td>Yes</td>
<td>-</td>
<td>See &quot;Amazon Simple Queue Service Backends&quot; on Java Backend Detection</td>
</tr>
<tr>
<td>Amazon</td>
<td>Simple Notification Service (SNS)</td>
<td>-</td>
<td>-</td>
<td>No</td>
<td>Yes</td>
<td>-</td>
<td>See &quot;Amazon Simple Notification Service Backends&quot; on Java Backend Detection</td>
</tr>
<tr>
<td>Apache</td>
<td>ActiveMQ</td>
<td>5.x+</td>
<td>JMS 1.x</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Example Message Queue Backend Configuration</td>
</tr>
<tr>
<td>Apache</td>
<td>ActiveMQ</td>
<td>5.x+</td>
<td>STOMP</td>
<td>No</td>
<td>-</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Apache</td>
<td>ActiveMQ</td>
<td>5.8.x+</td>
<td>AMQP 1.0</td>
<td>No</td>
<td>-</td>
<td>Yes</td>
<td>Example Message Queue Backend Configuration</td>
</tr>
<tr>
<td>Apache</td>
<td>Axis</td>
<td>1.x, 2.x</td>
<td>JAX-WS</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
<td>Default exclude rules exist for Apache Axis, Axis2, and Axis Admin Servlets. See also &quot;Web Service Entry Points&quot; on Java Backend Detection.</td>
</tr>
<tr>
<td>Apache</td>
<td>Apache CXF</td>
<td>2.1</td>
<td>JAX-WS</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
<td>To enable correlation, set node property enable-soap-header-correlation=true.</td>
</tr>
<tr>
<td>Apache</td>
<td>Kafka</td>
<td>0.9.0.0 to 2.0.0</td>
<td>-</td>
<td>Yes</td>
<td>Yes</td>
<td>Kafka consumer entry points are disabled by default. Correlation is supported. See Apache Kafka Consumer Backends.</td>
<td></td>
</tr>
<tr>
<td>Apache</td>
<td>Synapse</td>
<td>2.1</td>
<td>HTTP</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
<td>To enable correlation, set node property enable-soap-header-correlation=true.</td>
</tr>
<tr>
<td>Fiorano</td>
<td>Fiorano MQ</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>IBM</td>
<td>IBM Web Application Server (WAS)</td>
<td>6.1+, 7.x</td>
<td>Embedded JMS</td>
<td>-</td>
<td>Yes</td>
<td>-</td>
<td>Example Message Queue Backend Configuration</td>
</tr>
<tr>
<td>IBM</td>
<td>IBM MQ (formerly IBM WebSphere MQ)</td>
<td>6+</td>
<td>JMS</td>
<td>Yes</td>
<td>Yes</td>
<td>Example Message Queue Backend Configuration</td>
<td></td>
</tr>
<tr>
<td>Mulesoft</td>
<td>Mule ESB</td>
<td>3.4, 3.6, 3.7, 3.8, 3.9, 4.1</td>
<td>HTTP, JMS</td>
<td>Yes</td>
<td>Yes</td>
<td>Mule ESB Startup Settings</td>
<td></td>
</tr>
<tr>
<td>Open Source</td>
<td>Eclipse Vert.x verticles</td>
<td>3.3.x, 3.4.x, 3.5.0, 3.6.0</td>
<td>-</td>
<td>Yes (correlation only)</td>
<td>Yes</td>
<td>The Java Agent detects messaging exit calls between verticles.</td>
<td></td>
</tr>
<tr>
<td>Open Source</td>
<td>Open MQ</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>---------</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td></td>
</tr>
<tr>
<td>Oracle</td>
<td>Java Message Service</td>
<td>2.0</td>
<td>JMS</td>
<td>Correlation of the listener is disabled by default</td>
<td>Yes</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Oracle</td>
<td>Oracle AQ</td>
<td>-</td>
<td>JMS</td>
<td>-</td>
<td>Yes</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Oracle</td>
<td>OSB deployed on WebLogic</td>
<td>12.2.1</td>
<td>HTTP</td>
<td>JMS</td>
<td>Yes</td>
<td>Yes</td>
<td>OSB Support</td>
</tr>
<tr>
<td>Oracle / BEA</td>
<td>WebLogic</td>
<td>9.x+</td>
<td>JMS 1.1</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Oracle WebLogic Startup Settings</td>
</tr>
<tr>
<td>Progress</td>
<td>SonicMQ</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Pivotal</td>
<td>RabbitMQ</td>
<td>-</td>
<td>HTTP</td>
<td>-</td>
<td>Yes</td>
<td>-</td>
<td>See &quot;RabbitMQ Backends&quot; on Java Backend Detection</td>
</tr>
<tr>
<td>Rabbit</td>
<td>RabbitMQ Spring Client</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
<td>See &quot;RabbitMQ Backends&quot; on Java Backend Detection</td>
</tr>
<tr>
<td>Red Hat</td>
<td>HornetQ (formerly JBoss Messaging and JBoss MQ)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Red Hat</td>
<td>JBoss A-MQ</td>
<td>4.x+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>-</td>
</tr>
<tr>
<td>Spring</td>
<td>Spring Integration</td>
<td>2.2.0+, 4.0+</td>
<td>JMS</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Spring Integration Support</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>See also &quot;Java Message Service Backends&quot; on Java Backend Detection</td>
</tr>
<tr>
<td>WSO2</td>
<td>ESB</td>
<td>4.7.0</td>
<td>-</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
<td>EUM Correlation is not supported</td>
</tr>
</tbody>
</table>

### JDBC Drivers and Database Servers Support

The Java Agent supports these JDBC driver and database server environments. AppDynamics can follow transactions using these drivers to the designated database.

<table>
<thead>
<tr>
<th>JDBC Vendor</th>
<th>Driver Version</th>
<th>Driver Type</th>
<th>Database Server</th>
<th>Database Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>10.9.1.0</td>
<td>Embedded or client</td>
<td>Derby</td>
<td>-</td>
</tr>
<tr>
<td>Apache</td>
<td>-</td>
<td>-</td>
<td>Cassandra</td>
<td>-</td>
</tr>
<tr>
<td>Progress</td>
<td>DataDirect</td>
<td>data connectivity for ODBC and JDBC driver access, data integration, and SaaS and cloud computing solutions</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>IBM</td>
<td>JDBC 3.0 version 3.57.82 or JDBC 4.0 version 4.7.85</td>
<td>DB2 Universal JDBC driver</td>
<td>DB2</td>
<td>9.x</td>
</tr>
<tr>
<td>IBM</td>
<td>JDBC 3.0 version 3.66.46 or JDBC 4.0 version 4.16.53</td>
<td>DB2 Universal JDBC driver</td>
<td>DB2</td>
<td>10.1</td>
</tr>
<tr>
<td>IBM</td>
<td>-</td>
<td>Type IV</td>
<td>Informix</td>
<td>-</td>
</tr>
<tr>
<td>Microsoft</td>
<td>4</td>
<td>Type II</td>
<td>MS SQL Server</td>
<td>2012</td>
</tr>
<tr>
<td>Oracle MySQL, MySQL Community</td>
<td>5.x</td>
<td>Type II, Type IV</td>
<td>MySQL</td>
<td>5.x</td>
</tr>
<tr>
<td>Oracle</td>
<td>RAC</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Oracle</td>
<td>9.x</td>
<td>Type II, Type IV</td>
<td>Oracle Database</td>
<td>8i+</td>
</tr>
<tr>
<td>Open Source PostgreSQL</td>
<td>42.2.5</td>
<td>Type IV</td>
<td>Postgres</td>
<td>8.x, 9.x, 11x</td>
</tr>
<tr>
<td>Sybase</td>
<td>jConnect</td>
<td>Type IV</td>
<td>Sybase</td>
<td>-</td>
</tr>
<tr>
<td>Teradata</td>
<td></td>
<td></td>
<td>Teradata</td>
<td>-</td>
</tr>
</tbody>
</table>

**Notes:**
- Type II is a C or OCI driver
Type IV is a thin database client and is a pure Java driver

NoSQL/Data Grids/Cache Servers Support

The Java Agent supports these NoSQL, data grids and cache server environments. Some require additional configuration. Click the link on the database, data grid or cache name in the following support matrix for information about additional configuration required or related configuration topics.

<table>
<thead>
<tr>
<th>Vendor</th>
<th>Database/Data Grid/Cache</th>
<th>Version</th>
<th>Correlation/Entry Points</th>
<th>JMX</th>
<th>Configuration Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon</td>
<td>DynamoDB</td>
<td>-</td>
<td>Exit Points</td>
<td>-</td>
<td>See &quot;Amazon Web Services&quot; on Java Backend Detection.</td>
</tr>
<tr>
<td>Amazon</td>
<td>Simple Storage Service (S3)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>&quot;Amazon Simple Storage Service Backends&quot; on Java Backend Detection.</td>
</tr>
<tr>
<td>Apache</td>
<td>Cassandra</td>
<td>1.x, 2.x</td>
<td>Correlation for Thrift drivers only</td>
<td>Yes</td>
<td>• &quot;Cassandra Backends&quot; on Java Backend Detection.</td>
</tr>
<tr>
<td>Apache</td>
<td>Lucene - Apache Solr</td>
<td>1.4.1</td>
<td>Entry Points</td>
<td>Yes</td>
<td>Apache Solr Startup Settings</td>
</tr>
<tr>
<td>JBoss</td>
<td>Cache TreeCache</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>JBoss Startup Settings</td>
</tr>
<tr>
<td>JBoss</td>
<td>Infinispan</td>
<td>5.3.0+</td>
<td>Correlation</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Open Source</td>
<td>Memcached</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Memcached Exit Points</td>
</tr>
<tr>
<td>Open Source</td>
<td>MongoDB</td>
<td>3.1</td>
<td>-</td>
<td>-</td>
<td>See &quot;MongoDB Backends&quot; on Java Backend Detection</td>
</tr>
<tr>
<td>Oracle</td>
<td>Coherence</td>
<td>3.7.1</td>
<td>Custom-Exit</td>
<td>Yes</td>
<td>Coherence Startup Settings</td>
</tr>
<tr>
<td>Red Hat</td>
<td>JBoss DataGrid</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>JBoss Startup Settings</td>
</tr>
<tr>
<td>JBoss</td>
<td>Cache TreeCache</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>JBoss</td>
<td>Infinispan</td>
<td>5.3.0+</td>
<td>Correlation</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Terracotta</td>
<td>EhCache</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

RPC/Web Services API/HTTP Client Support

The Java Agent supports these RPC, web services or API framework types. Some require additional configuration as indicated in the Configuration Notes column.

<table>
<thead>
<tr>
<th>Vendor</th>
<th>RPC/Web Services API Framework/HTTP Client Support</th>
<th>Version</th>
<th>SOA Protocol-WebServices</th>
<th>Auto Naming</th>
<th>Correlation/Entry Points</th>
<th>Exit Points</th>
<th>Configurable BT Naming Properties</th>
<th>Detection</th>
<th>Configur Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>Apache CXF</td>
<td>2.1</td>
<td>JAX-WS</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>See &quot;HTTP Backends ava Backe Detection</td>
</tr>
<tr>
<td>Apache</td>
<td>Apache HTTP Client</td>
<td>-</td>
<td>HTTPClient (now in Apache HTTP Components)</td>
<td>Yes</td>
<td>Yes (correlation only)</td>
<td>Yes</td>
<td>-</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Apache</td>
<td>Apache Async HTTP Client</td>
<td>4.1.x</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Apache</td>
<td>Ribbon HTTP Client</td>
<td>2.1.0</td>
<td>HTTP Client</td>
<td>Yes</td>
<td>Yes (correlation) Entry - NA</td>
<td>Yes</td>
<td>NA</td>
<td>Yes</td>
<td>Binary Ref Entry Point Apache Th</td>
</tr>
<tr>
<td>Apache</td>
<td>Apache Thrift</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Eclipse</td>
<td>Jetty</td>
<td>8.x, 9.x</td>
<td>HTTP Client</td>
<td>Yes</td>
<td>Yes (correlation only)</td>
<td>Yes</td>
<td>(ART supported)</td>
<td>-</td>
<td>See &quot;HTTP Backends ava Backe Detection</td>
</tr>
<tr>
<td>Vendor</td>
<td>Application</td>
<td>Version</td>
<td>JAX-RPC</td>
<td>IIOP</td>
<td>HTTP</td>
<td>Correlation</td>
<td>Entry</td>
<td>Default Configuration</td>
<td>StartUp Settings</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------------</td>
<td>---------</td>
<td>---------</td>
<td>------</td>
<td>------</td>
<td>-------------</td>
<td>-------</td>
<td>------------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>IBM</td>
<td>WebSphere</td>
<td>6.x, 7.x, 8.x</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>IBM WebSphere and InfoSphere Startup Settings also see D configuration excludes WebSphere classes</td>
<td></td>
</tr>
<tr>
<td>IBM</td>
<td>WebSphere</td>
<td>7.x, 8.x</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>IBM WebSphere and InfoSphere Startup Settings also see D configuration excludes WebSphere classes</td>
<td></td>
</tr>
<tr>
<td>Open Source</td>
<td>java.net.Http</td>
<td>-</td>
<td>HTTP</td>
<td>Yes</td>
<td>-</td>
<td>Yes</td>
<td>Yes</td>
<td>See &quot;HTTP Backends&quot; ava Back Detection.</td>
<td></td>
</tr>
<tr>
<td>Open Source</td>
<td>HttpClient</td>
<td>0.3-3</td>
<td>Oracle SOA (and potentially others that embed this library)</td>
<td>-</td>
<td>Correlation: Yes; Entry: No</td>
<td>Yes</td>
<td>-</td>
<td>Yes</td>
<td>Oracle WebLogic Startup Settings also see D configuration excludes WebSphere classes</td>
</tr>
<tr>
<td>Open Source</td>
<td>Grizzly</td>
<td>Grizzly Async HTTP Client (com.ning.http-client 1.6.x, 1.9.x, grizzly-http-client 1.1.x)</td>
<td>HTTP</td>
<td>-</td>
<td>Correlation: Yes; Entry: No</td>
<td>Yes</td>
<td>-</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Oracle</td>
<td>GlassFish Metro</td>
<td>-</td>
<td>JAX-WS</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Not by Default</td>
<td></td>
</tr>
<tr>
<td>Oracle</td>
<td>GlassFish Metro with Grails</td>
<td>-</td>
<td>JAX-WS</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Not by Default</td>
<td></td>
</tr>
<tr>
<td>Oracle</td>
<td>Oracle Application Server</td>
<td>ORMI</td>
<td>-</td>
<td>no</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Oracle</td>
<td>WebLogic</td>
<td>10.x</td>
<td>T3, IIOP</td>
<td>Yes</td>
<td>Correlation: Yes; Entry: No</td>
<td>Yes</td>
<td>-</td>
<td>Yes</td>
<td>Oracle WebLogic Startup Settings also see D configuration excludes WebSphere classes</td>
</tr>
<tr>
<td>Oracle/Sun</td>
<td>Java</td>
<td>11</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Yes (ART supported)</td>
<td></td>
</tr>
<tr>
<td>Oracle/Sun</td>
<td>Sun RMI</td>
<td>-</td>
<td>IIOP</td>
<td>-</td>
<td>Not by Default</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Oracle/Sun</td>
<td>Sun RMI</td>
<td>-</td>
<td>JRMP</td>
<td>-</td>
<td>No</td>
<td>Yes</td>
<td>host/port</td>
<td>Yes</td>
<td>Oracle WebLogic Startup Settings also see D configuration excludes WebSphere classes</td>
</tr>
<tr>
<td>Red Hat</td>
<td>JBoss A-MQ</td>
<td>4.x+</td>
<td>RMI</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>JBoss and Wildfly Sta Settings</td>
<td></td>
</tr>
<tr>
<td>Square</td>
<td>OkHttp</td>
<td>-</td>
<td>HTTP</td>
<td>Yes</td>
<td>Correlation: Yes; Entry: No</td>
<td>Yes</td>
<td>-</td>
<td>Synchronous only</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>Web Services</td>
<td>-</td>
<td>SOAP over HTTP</td>
<td>-</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
<td>Create Ma Rules for v Services</td>
<td></td>
</tr>
<tr>
<td>jersey.github.io</td>
<td>Reactive JAX-RS client API</td>
<td>2.25+</td>
<td>HTTP Client</td>
<td>Yes</td>
<td>Yes (correlation)</td>
<td>Yes</td>
<td>NA</td>
<td>Yes</td>
<td>&quot;Web Serv Entry Poin Java Back Detection</td>
</tr>
</tbody>
</table>

**Notes:**
- "Web Serv Entry Poin Java Back Detection"
Business Transaction Error Detection

The Java Agent supports the following logging frameworks for business transaction error detection:

- Apache Log4j and Log4j 2
- java.util.logging
- Simple Logging Facade for Java (SLF4J)
  - Support for the following method has been added: `public void error(String format, Object... argArray)`
- Logback

To instrument other types of loggers, see Error Detection.

.NET Agent Support

Supported Runtime Environments

This section lists the environments where the .NET Agent does some automatic discovery after little or no configuration.

OS Versions

- Microsoft Windows Server 2008 (32-bit and 64-bit)
- Microsoft Windows Server 2008 R2
- Microsoft Windows Server 2012
- Microsoft Windows Server 2012 R2
- Microsoft Windows Server 2016
- Microsoft Windows 7, 8, 8.1, 10

Microsoft .NET Frameworks

Microsoft .NET Framework versions 2.0, 3.0, 3.5, 4.0, 4.5, 4.5.2, 4.6, 4.7 on the following runtime environments:

- Microsoft IIS versions 6.0, 7.0, 7.5, 8.0, 8.5, 10
- Managed Windows Services
- Managed Standalone Applications
- Microsoft SharePoint 2010, 2013 as services running inside IIS
- Microsoft .NET Core 2.0/2.1 for Windows
- Microsoft .NET Core 2.2 for Windows is supported for .NET Agent versions 4.5.7 and later

Microsoft Windows Azure

- Azure App Services for .NET 4.6 environments in the Azure Portal
  - Web Apps
  - Web Jobs
  - API Apps
  - Container Services
  - For Azure App Services, the .NET Machine Agent disables certain .NET Machine Agent infrastructure monitoring features: CLR crash reporting, machine snapshots, and Windows performance counter monitoring.
- Azure Cloud Services
  - Web Roles
  - Worker Roles

Unsupported Frameworks

- Microsoft .NET versions 1.0, 1.1
- Unmanaged native code
Automatically Discovered Business Transactions

The .NET Agent discovers business transactions for the following frameworks by default. The agent enables detection without additional configuration.

<table>
<thead>
<tr>
<th>Type</th>
<th>Custom Configuration Options?</th>
<th>Downstream Correlation?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASP.NET*</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ASP.NET MVC 2</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ASP.NET MVC 3</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ASP.NET MVC 4</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ASP.NET MVC 5</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ASP.NET Core on the full framework</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Open Web Interface for .NET (OWIN) web API</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>.NET Remoting</td>
<td>No</td>
<td>See Enable Correlation for .NET Remoting.</td>
</tr>
<tr>
<td>Windows Communication Foundation (WCF)</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Web Services including SOAP</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Message Queues

<table>
<thead>
<tr>
<th>Type</th>
<th>Custom Configuration Options?</th>
<th>Downstream Correlation?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache ActiveMQ NMS framework and related MQs</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>IBM WebSphere MQ</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Microsoft Message Queuing (MSMQ)</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Microsoft Service Bus / Windows Azure Service Bus</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>NServiceBus over MSMQ or RabbitMQ transport</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>RabbitMQ</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>TIBCO Enterprise Message Service</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>TIBCO Rendezvous</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Windows Azure Queue</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

* The .NET Agent automatically discovers entry points for ASP.NET web forms with the Async property set to “true” in the Page directive.

Supported Loggers for the .NET Agent

- Log4Net
- NLog
- System Trace
- Windows Event Log

If you are using a different logger, see Error Detection.

Remote Service Detection

The .NET Agent automatically detects the following remote service types. The agent enables detection by default. You do not need to perform extra configuration.
<table>
<thead>
<tr>
<th>Type</th>
<th>Customizable Configuration Options?</th>
<th>Async Detection?*</th>
<th>Downstream Correlation?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Directory Services, including LDAP</td>
<td>No</td>
<td>No</td>
<td>N/A</td>
</tr>
<tr>
<td>HTTP</td>
<td>Yes</td>
<td>See Asynchronous Exit Points for .NET.</td>
<td>Yes</td>
</tr>
<tr>
<td>MongoDB: C# and .NET MongoDB Driver version 1.10, 2.0</td>
<td>No</td>
<td>See Asynchronous Exit Points for .NET.</td>
<td>N/A</td>
</tr>
<tr>
<td>.NET Remoting</td>
<td>Yes</td>
<td>No</td>
<td>See Enable Correlation for .NET Remoting.</td>
</tr>
<tr>
<td>WCF</td>
<td>Yes</td>
<td>See Asynchronous Exit Points for .NET.</td>
<td>Yes</td>
</tr>
<tr>
<td>WCF Data Services</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Web Services, including SOAP</td>
<td>Yes</td>
<td>See Asynchronous Exit Points for .NET.</td>
<td>Yes</td>
</tr>
<tr>
<td>Azure Service Fabric Remoting v1 and v2—for the .NET Microservices Agent</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Data Integration**

<table>
<thead>
<tr>
<th>Type</th>
<th>Customizable Configuration Options?</th>
<th>Async Detection?*</th>
<th>Downstream Correlation?</th>
</tr>
</thead>
</table>

**Message Queues**

<table>
<thead>
<tr>
<th>Type</th>
<th>Customizable Configuration Options?</th>
<th>Async Detection?*</th>
<th>Downstream Correlation?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache ActiveMQ NMS framework and related MQs</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>IBM WebSphere MQ (IBM XMS)</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Microsoft Message Queuing ( MSMQ)</td>
<td>Yes</td>
<td>See MSMQ Backends for .NET</td>
<td>See MSMQ Backends for .NET</td>
</tr>
<tr>
<td>Microsoft Service Bus / Windows Azure Service Bus</td>
<td>No</td>
<td>Async exit points only</td>
<td>Yes</td>
</tr>
<tr>
<td>NServiceBus over MSMQ or RabbitMQ transport</td>
<td>No</td>
<td>See NServiceBus Backends for .NET</td>
<td>Yes</td>
</tr>
<tr>
<td>RabbitMQ</td>
<td>See RabbitMQ Backends for .NET</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>TIBCO Enterprise Message Service</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>TIBCO Rendezvous</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Windows Azure Queue</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

* The agent discovers asynchronous transactions for the Microsoft .NET 4.5 framework. See [Asynchronous Exit Points for .NET](#).

**Supported Windows Azure Remote Services**

<table>
<thead>
<tr>
<th>Type</th>
<th>Customizable Configuration?</th>
<th>Downstream Correlation?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Azure Blob</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Azure Queue</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Microsoft Service Bus</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Cache Clients**

<table>
<thead>
<tr>
<th>Type</th>
<th>Customizable Configuration?</th>
<th>Async Detection?*</th>
<th>AppD for Databases?</th>
</tr>
</thead>
<tbody>
<tr>
<td>StackExchange.Redis</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>
Data Storage Detection

The .NET Agent automatically detects the following data storage types. The agent enables detection by default. You do not need to perform extra configuration.

<table>
<thead>
<tr>
<th>Type</th>
<th>Customizable Configuration?</th>
<th>Async Detection?*</th>
<th>AppD for Databases?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADO.NET (see supported clients below)</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Windows Azure Blob Storage</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Windows Azure File Storage</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Windows Azure Table Storage</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

* The agent discovers asynchronous transactions for the Microsoft .NET 4.5 framework. See Asynchronous Exit Points for .NET

Supported ADO.NET Clients

AppDynamics can monitor any ADO.NET client version and type. Clients we've tested include the following:

<table>
<thead>
<tr>
<th>Database Name</th>
<th>Database Version</th>
<th>Client Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oracle</td>
<td>10, 11, 12</td>
<td>ODP.NET</td>
</tr>
<tr>
<td>Oracle</td>
<td>10, 11, 12</td>
<td>Microsoft Provider for Oracle</td>
</tr>
<tr>
<td>MySQL</td>
<td>5.x</td>
<td>Connector/Net and ADO.NET</td>
</tr>
</tbody>
</table>

* Microsoft, SQL Server, and Windows are registered trademarks of Microsoft Corporation in the United States and other countries.

Node.js Agent Support

Node.js Versions

- The 4.5.11 Node.js agent supports Node.js v0.8, v0.10, v0.12, v4, v5, v6, v7, v8, v9, v10, and v11.

For agent versions 4.5.12 and on the npm install will stop and print a message for the two scenarios listed:

- nodejs version less than 6
  - This version of AppDynamics agent supports Node.js versions 6.0 and above. For older versions of Node.js use the the AppDynamics agent 4.5.11 by installing with 'npm install appdynamics@4.5.11'
- nodejs less than 8 on mac
  - This version of AppDynamics agent on Mac OS supports Node.js versions 8.0 and above. For older versions of Node.js use the the AppDynamics agent 4.5.11 by installing with 'npm install appdynamics@4.5.11'

Operating Systems

- The agent is compatible with any Linux distribution based on glibc 2.5+
- Mac OS X 10.8+ (Agent / Node.js version specific compatibility as noted)
- Windows Server 2008R2+ and newer for 64 bit applications for Node.js versions 0.12.0 and higher (Agent / Node.js version
Transaction Naming

<table>
<thead>
<tr>
<th>Entry Type</th>
<th>Default Transaction Naming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node.js Web</td>
<td>URI</td>
</tr>
</tbody>
</table>

HTTP Exit Points

**Supported HTTP Exit Points**

Node.js HTTP client library


Database Exit Points

- MongoDB
- MySQL
- PostgreSQL
- Riak - will be displayed as HTTP backends
- Couchbase
- DynamoDB using the official AWS SDK driver
- Cassandra

Cache Exit points

- Memcached
- Redis

PHP Agent Support

**PHP Versions**

The PHP agent supports PHP 5.6, 7.0, 7.1, 7.2, and 7.3.

The PHP agent does not:

- Monitor PHP applications in Zend Thread Safety (ZTS) mode. If you are using ZTS, AppDynamics suggests that you review your dependencies on ZTS to confirm that you actually need it, and if you do not, to switch to non-ZTS mode
- Support Zend Monitor
- Officially support plugins that encrypt and/or obfuscate PHP code, such as Zend Guard or ionCUBE Loader

**PHP Web Servers**

- Apache 2.2 and 2.4 in the following modes:
  - prefork mode using mod_php
  - worker MPM mode using `mod_fastcgi` with `php-fpm` or `mod_fcgid` with `php-cgi`
- Any web server compatible with `php-fpm`

**Operating Systems**
- Any Linux distribution based on glibc 2.5+
- Mac OS X 10.9+

**PHP Frameworks and Protocols**

<table>
<thead>
<tr>
<th>Framework/Protocol</th>
<th>Version</th>
<th>Entry Point Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Drupal</td>
<td>7, 8</td>
<td>Drupal</td>
</tr>
<tr>
<td>WordPress</td>
<td>3.4+, 4.x, 5.x</td>
<td>Wordpress</td>
</tr>
<tr>
<td>Zend</td>
<td>1, 2</td>
<td>PHP MVC</td>
</tr>
<tr>
<td>CodeIgniter</td>
<td>2.x, 3.x</td>
<td>PHP MVC</td>
</tr>
<tr>
<td>FuelPHP</td>
<td>1.5x, 1.6x</td>
<td>PHP MVC</td>
</tr>
<tr>
<td>Magento</td>
<td>1.5, 1.6, 1.7</td>
<td>PHP MVC</td>
</tr>
<tr>
<td>Symfony</td>
<td>1, 2</td>
<td>PHP MVC</td>
</tr>
<tr>
<td>CakePHP</td>
<td>2.x, 3.x</td>
<td>PHP MVC</td>
</tr>
<tr>
<td>Laravel</td>
<td>5.7</td>
<td>PHP MVC</td>
</tr>
<tr>
<td>HTTP</td>
<td></td>
<td>PHP Web</td>
</tr>
<tr>
<td>CLI</td>
<td></td>
<td>PHP CLI</td>
</tr>
</tbody>
</table>

If your PHP framework is not listed here, the agent detects your entry points as PHP Web and names the business transactions based on the first two segments of the URI — the default naming convention for PHP Web transactions. So it is still possible to monitor applications on unsupported frameworks. Laravel BTs are detected as symfony, as Laravel itself is built on top of Symfony.

**Transaction Naming**

<table>
<thead>
<tr>
<th>Framework/Environment</th>
<th>Default Transaction Naming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Drupal</td>
<td>page callback name</td>
</tr>
<tr>
<td>Wordpress</td>
<td>template name</td>
</tr>
<tr>
<td>PHP MVC Frameworks</td>
<td>controller:action</td>
</tr>
<tr>
<td>PHP Modular MVC Frameworks</td>
<td>module:controller:action</td>
</tr>
<tr>
<td>PHP Web</td>
<td>URI</td>
</tr>
<tr>
<td>PHP Web Service</td>
<td>service name.operation name</td>
</tr>
<tr>
<td>PHP CLI</td>
<td>last two segments of the script's directory path plus the name of the script</td>
</tr>
</tbody>
</table>

Virtual host prefixing is available for all supported entry point types except PHP CLI.

**PaaS Providers**

<table>
<thead>
<tr>
<th>PaaS Provider</th>
<th>Buildpack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pivotal Cloud Foundry</td>
<td><a href="https://github.com/Appdynamics/php-buildpack">https://github.com/Appdynamics/php-buildpack</a></td>
</tr>
<tr>
<td></td>
<td>See <a href="http://docs.pivotal.io/appdynamics/index.html">http://docs.pivotal.io/appdynamics/index.html</a> for information about integration with PCF.</td>
</tr>
</tbody>
</table>

**Exit Points**
Supported HTTP Exit Points

- curl/curl-multi
- drupal_http_request()
- fopen(), file_get_contents()
- Zend_HTTP_Client::request()

*The total time reported for a curl/multi_curl request in the Controller is the same as reported by the function curl_getinfo. Also, we report the following execution metrics in the exit call details for the curl/multi_curl request which are included in the total time:

  - namelookup_time
  - connect_time
  - pretransfer_time
  - redirect_time

Supported Database Exit Points

- MySQL old native driver (removed for PHP 7)
- MySQLi Extension
- OCI8
- PDO
- PostgreSQL accessed via PDO and pgsql extensions

*mysqli_multi_query is not supported.

Supported Cache Exit Points

- Memcache
- Memcached
- Predis 0.8.5 and 1.1.1, on PHP versions 5.6 and higher
- Phpredis 4.1

Although Predis is a full PHP client library, the PHP Agent supports Predis as an exit point only, not as an entry point.

Supported Web Service Exit Points

- PHP SOAPClient
- NuSOAP 0.9.5

Supported Message Queue Exit Points

- RabbitMQ

RabbitMQ support requires the amqp extension.

Opcode Cache Compatibility

- Alternative PHP Cache (APC)

Python Agent Support
Python Versions

- The Python agent supports CPython 2.6, 2.7, 3.4, 3.5, 3.6, and 3.7.

Operating Systems

- Any Linux distribution based on glibc 2.5+
- Mac OS X 10.8+

Python Frameworks and Protocols

<table>
<thead>
<tr>
<th>Framework/Protocol</th>
<th>Version</th>
<th>Entry Point Type</th>
<th>Default Transaction Naming</th>
</tr>
</thead>
<tbody>
<tr>
<td>WSGI</td>
<td>1.0</td>
<td>Python Web</td>
<td>First two segments of URI</td>
</tr>
<tr>
<td>Tornado</td>
<td>3.2 - 4.5</td>
<td>Python Web</td>
<td>First two segments of URI</td>
</tr>
</tbody>
</table>

AppDynamics has tested the Python Agent on Tornado, Django, Flask, CherryPy, Bottle, and Pyramid.

You can configure the agent to instrument any WSGI-based application or framework as Python Web, including but not limited to those listed below.

At present, the Python agent fully supports exception detection in Django, Flask, CherryPy, Bottle, Pyramid, and Tornado frameworks. Other WSGI frameworks and custom WSGI applications may install exception handlers that effectively hide some exceptions from the agent. In such cases, the agent will only detect exceptions during exit calls, uncaught exceptions which are propagated to the WSGI server, and exceptions reported via the custom business transaction API.

<table>
<thead>
<tr>
<th>WSGI-Based Frameworks</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bottle</td>
<td></td>
</tr>
<tr>
<td>CherryPy</td>
<td></td>
</tr>
<tr>
<td>Django</td>
<td></td>
</tr>
<tr>
<td>Flask</td>
<td></td>
</tr>
<tr>
<td>PasteDeploy</td>
<td></td>
</tr>
<tr>
<td>Pyramid</td>
<td></td>
</tr>
</tbody>
</table>

Database Exit Points

<table>
<thead>
<tr>
<th>Supported Database Exit Points</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>cx_Oracle</td>
<td>5.1.x</td>
</tr>
<tr>
<td>MongoDB</td>
<td>3.1+</td>
</tr>
<tr>
<td>MySQL-Python</td>
<td></td>
</tr>
<tr>
<td>mysqlclient</td>
<td></td>
</tr>
<tr>
<td>MySQL Connector/Python</td>
<td></td>
</tr>
<tr>
<td>Psycopg 2</td>
<td></td>
</tr>
<tr>
<td>PyMySql</td>
<td></td>
</tr>
<tr>
<td>TorMySql</td>
<td></td>
</tr>
</tbody>
</table>
HTTP Exit Points

<table>
<thead>
<tr>
<th>Supported HTTP Exit Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>httplib^</td>
</tr>
<tr>
<td>httplib2</td>
</tr>
<tr>
<td>requests</td>
</tr>
<tr>
<td>urllib</td>
</tr>
<tr>
<td>urllib2</td>
</tr>
<tr>
<td>urllib3</td>
</tr>
<tr>
<td>tornado.httpclient</td>
</tr>
</tbody>
</table>

^ The agent detects calls to any external library built on top of httplib. Therefore, backend calls to such services, such as boto, dropbox, python-twitter, and so on are detected and displayed as HTTP exit calls.

Cache Exit Points

<table>
<thead>
<tr>
<th>Supported Cache Exit Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memcache</td>
</tr>
<tr>
<td>Redis-py</td>
</tr>
</tbody>
</table>

Apache Server Agent Support

Apache Web Servers

<table>
<thead>
<tr>
<th>Supported Apache Web Server Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Apache HTTP Server 2.2.x</td>
</tr>
<tr>
<td>• Apache HTTP Server 2.4.x</td>
</tr>
<tr>
<td>• IBM HTTP Server 7.0 +</td>
</tr>
<tr>
<td>• Oracle HTTP Server 11g+</td>
</tr>
</tbody>
</table>

Operating Systems

• Any Linux distribution based on glibc 2.5+

Architectures

• 32-bit
• 64-bit

Automatically Discovered Business Transactions

The Apache Agent automatically discovers the following business transactions:

<table>
<thead>
<tr>
<th>Type</th>
<th>Custom Configuration Options</th>
<th>Downstream Correlation</th>
</tr>
</thead>
</table>
By default the agent excludes requests for the following static file types:
bmp
cab
class
conf
css
doc
gif
ico
jar
jpeg
jpg
js
mov
mp3
mp4
pdf
png
pps
properties
swf
tif
txt
zip

Remote Service Detection

Apache Modules

The Apache Agent automatically detects loaded Apache modules as remote services. The agent excludes a list of common modules from detection.

Show the list of excluded modules...
core.c
http_core.c
mod_access_compat.c
mod_actions.c
mod_alias.c
mod_allowmethods.c
mod_appdynamics.cpp
mod_auth_basic.c
mod_auth_digest.c
mod_authn_alias.c
mod_authn_anon.c
mod_authn_core.c
mod_authn_dbd.c
mod_authn_dbm.c
mod_authn_default.c
mod_authn_file.c
mod_authn_socache.c
For End User Monitoring, the Apache Agent does not support automatic injection of the Javascript `adrum` header and footer to instrument web pages.

### C/C++ Agent Supported Platforms

**Operating Systems**

- Any Linux distribution based on glibc 2.5+
- Windows Server 2008 R2 and above
  - Visual Studio 2015, or later
Go SDK Support

Operating Systems

- Any Linux distribution based on glibc 2.5+.

IIB Agent Support

IIB Versions

The IIB agent supports WMB v8, IIB v9, v10.

Operating Systems

The IIB agent supports the following operating systems:

- Linux x86-64 bit

IIB Node Types

The agent can continue business transactions detected upstream at the following node types:

- SOAPInput
- HTTPInput
- JMSInput
- MQInput

The agent can detect and tag exit calls for downstream correlation at the following node types:

- SOAPRequest
- HTTPRequest
- JMSOutput, JMSReply
- MQOutput, MQReply

For MQ, we use the MQRFH2 message header to provide correlation. Any applications consuming MQ messages from IIB with IIB agent must support the MQRFH2 header.

The agent can detect database backend calls for the following node types:

- DatabaseRetrieve
- DatabaseRoute

All nodes are represented within AppDynamics Business Transactions as Threads. The per node timings can be seen in the tree view of the Business Transaction dashboard, and in transaction snapshots.