



APPDYNAMICS

AppDynamics for Java

AppDynamics Pro Documentation

Version 3.8.x

1. AppDynamics for Java	5
1.1 Supported Environments and Versions for Java	5
1.2 Install the App Agent for Java	20
1.2.1 Multi-Agent Deployment for Java	26
1.2.2 Java Server-Specific Installation Settings	27
1.2.2.1 Apache Cassandra Startup Settings	27
1.2.2.2 Apache Tomcat Startup Settings	29
1.2.2.2.1 Tomcat as a Windows Service Configuration	33
1.2.2.3 Coherence Startup Settings	34
1.2.2.4 GlassFish Startup Settings	34
1.2.2.5 IBM WebSphere and InfoSphere Startup Settings	39
1.2.2.6 JBoss Startup Settings	44
1.2.2.7 Jetty Startup Settings	56
1.2.2.8 Mule ESB Startup Settings	56
1.2.2.9 Oracle WebLogic Startup Settings	57
1.2.2.10 OSGi Infrastructure Configuration	61
1.2.2.11 Resin Startup Settings	63
1.2.2.12 Solr Startup Settings	66
1.2.2.13 Standalone JVM Startup Settings	67
1.2.2.14 Tanuki Service Wrapper Configuration	68
1.2.2.15 Tibco ActiveMatrix BusinessWorks Service Engine Configuration	69
1.2.2.16 SUN JDK 1.6 on Linux	69
1.2.3 Enable SSL for Java	69
1.2.4 Upgrade the App Agent for Java	75
1.2.5 Uninstall the App Agent for Java	76
1.3 Configure AppDynamics for Java	76
1.3.1 Business Transaction Configuration Methodology for Java	76
1.3.2 Java Web Application Entry Points	86
1.3.2.1 Servlet Entry Points	87
1.3.2.1.1 Automatic Naming Configurations for Servlet-Based Business Transactions	89
1.3.2.1.2 Custom Naming Configurations for Servlet-Based Business Transactions	97
1.3.2.1.3 Custom Expressions for Naming Business Transactions	109
1.3.2.1.4 Advanced Servlet Transaction Detection Scenarios	110
1.3.2.2 Struts Entry Points	123
1.3.2.3 Web Service Entry Points	125
1.3.2.4 POJO Entry Points	127
1.3.2.5 Spring Bean Entry Points	135
1.3.2.6 EJB Entry Points	138
1.3.2.7 JMS Entry Points	140
1.3.2.8 Binary Remoting Entry Points for Apache Thrift	142
1.3.2.9 CometD Support	145
1.3.2.10 Mule ESB Support	146
1.3.2.11 JAX-RS Support	147
1.3.2.12 Spring Integration Support	148
1.3.2.13 Instrumenting Apple WebObjects Applications	151
1.3.3 Exclude Rule Examples for Java	152
1.3.4 Configure Multi-Threaded Transactions for Java	155
1.3.4.1 Configure End-to-End Message Transactions for Java	157
1.3.5 Configure Backend Detection for Java	160
1.3.5.1 Configure Custom Exit Points for Java	166

1.3.5.2 Configurations for Custom Exit Points for Java	172
1.3.5.3 HTTP Exit Points for Java	182
1.3.5.4 JDBC Exit Points for Java	188
1.3.5.5 Message Queue Exit Points for Java	192
1.3.5.6 Web Services Exit Points for Java	199
1.3.5.7 Cassandra Exit Points for Java	200
1.3.5.8 RMI Exit Points for Java	201
1.3.5.9 Thrift Exit Points for Java	202
1.3.6 Configure Memory Monitoring for Java	203
1.3.6.1 Configure Automatic Leak Detection for Java	203
1.3.6.2 Configure and Use Object Instance Tracking for Java	206
1.3.6.3 Configure and Use Custom Memory Structures for Java	208
1.3.7 Configure Background Tasks for Java	215
1.3.8 Import and Export Transaction Detection Configuration for Java	217
1.3.9 Getter Chains in Java Configurations	222
1.3.10 Code Metric Information Points for Java	227
1.3.11 Configure JMX Metrics from MBeans	228
1.3.11.1 Create, Import or Export JMX Metric Configurations	235
1.3.11.2 Exclude JMX Metrics	240
1.3.11.3 Exclude MBean Attributes	241
1.3.11.4 Configure JMX Without Transaction Monitoring	242
1.3.11.5 Resolve JMX Configuration Issues	242
1.3.11.6 MBean Getter Chains and Support for Boolean and String Attributes	248
1.3.12 Percentile Metrics	252
1.4 Monitor Java Applications	254
1.4.1 Monitor JVMs	254
1.4.1.1 JVM Crash Guard	261
1.4.2 Monitor Java App Servers	265
1.4.2.1 Monitor JMX MBeans	267
1.4.3 Trace MultiThreaded Transactions for Java	276
1.4.4 Service Endpoint Monitoring	285
1.4.5 Monitoring in a Development Environment	289
1.5 Troubleshoot Java Application Problems	291
1.5.1 Troubleshoot Slow Response Times for Java	291
1.5.2 Configure Diagnostic Sessions For Asynchronous Activity	306
1.5.3 Troubleshoot Java Memory Issues	307
1.5.3.1 Troubleshoot Java Memory Leaks	307
1.5.3.2 Troubleshoot Java Memory Thrash	315
1.5.4 Detect Code Deadlocks for Java	322
1.6 Tutorials for Java	324
1.6.1 Overview Tutorials for Java	324
1.6.1.1 Use AppDynamics for the First Time with Java	325
1.6.2 Monitoring Tutorials for Java	328
1.6.2.1 Tutorial for Java - Events	328
1.6.2.2 Tutorial for Java - Flow Maps	330
1.6.2.3 Tutorial for Java - Server Health	334
1.6.2.4 Tutorial for Java - Transaction Scorecards	338
1.6.3 Troubleshooting Tutorials for Java	341
1.6.3.1 Tutorial for Java - Business Transaction Health Drilldown	341
1.6.3.2 Tutorial for Java - Exceptions	341
1.6.3.3 Tutorial for Java - Slow Transactions	347
1.6.3.4 Tutorial for Java - Troubleshooting using Events	350

1.7 Administer App Agents for Java	357
1.7.1 Resolving Configuration Issues App Agent for Java	357
1.7.2 App Agent for Java Configuration Properties	359
1.7.3 Configure and Start an Agent Logging Session	371
1.7.4 Configure App Agent for Java in z-OS or Mainframe Environments	372
1.7.5 App Agent for Java Performance Tuning	373
1.7.6 Move an App Agent for Java Node to a New Application or Tier	375
1.7.7 App Agent for Java Diagnostic Data	376
1.7.8 App Agent for Java Directory Structure	378
1.7.9 IBM App Agent for Java	379
1.7.10 Configure App Agent for Java for Batch Processes	380
1.7.11 Configure App Agent for Java in Restricted Environments	382
1.7.12 Configure App Agent for Java on Multiple JVMs on the Same Machine that Serve Different Tiers	383
1.7.13 Configure App Agent for Java on Multiple JVMs on the Same Machine that Serves the Same Tier	384
1.7.14 Configure App Agent for Java to Use Existing System Properties	386
1.7.15 Administer App Agent for Java FAQ	389
1.7.16 Configure App Agent for Java for JVMs that are Dynamically Identified	390
1.7.17 Add the Agent into an Embedded JVM	390

AppDynamics for Java

This information covers using AppDynamics for Java applications and environments. For general information see [AppDynamics Essentials](#) and [AppDynamics Features](#).

Tutorials

Monitor Java Applications

Troubleshoot Java Application Problems

Configure AppDynamics for Java

Supported Environments and Versions for Java

Administer App Agents for Java

Supported Environments and Versions for Java

- Supported Platform Matrix for the App Agent for Java
 - JVM Support
 - JVM Language Frameworks Support
 - Application Servers
 - Application Server Configuration
 - Message Oriented Middleware Support
 - Message Oriented Middleware Configuration
 - JDBC Drivers and Database Servers Support
 - Business Transaction Error Detection
 - NoSQL/Data Grids/Cache Servers Support
 - NoSQL/Data Grids/Cache Servers Configuration
 - Java Frameworks Support
 - Java Frameworks Configuration
 - RPC/Web Services API Support
 - RPC/Web Services API Framework Configuration

Supported Platform Matrix for the App Agent for Java

This page documents known environments in which the App Agent for Java has been used to instrument applications. The App Agent for Java can target specific Java bytecode. This provides wide-ranging flexibility, so if an environment is not listed here, this does not preclude the App Agent for Java from being able to extract valuable performance metrics. Contact AppDynamics Support or Sales for additional details.

Notes:

- A dash "-" in a table cell indicates that this column is not relevant to that particular environment.
- In cases where no version is provided, assume that all versions are supported. Contact AppDynamics Support or 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](#).

JVM Support

These are the known JVM environments in which the App Agent for Java has been used to instrument applications.

Vendor	Implementation	Version	Operating System	Object Instance Tracking	Automatic Leak Detection	Custom Memory Structures		
						Content Inspection	Access Tracking	Requires JVM Restart?
Oracle	Java HotSpot	7 Update 45+	Solaris Sparc 64 Windows Linux	-	-	-	-	-
BEA	JRockit	1.5	-	-	Yes	Yes	Yes	Yes
BEA	JRockit	1.6, 1.7	-	-	Yes	Yes	-	-
Oracle	JRockit JVM	28.1+	Linux Intel 64 Windows	-	-	-	-	-
IBM	JVM	1.5.x, 1.6.x, 1.7.x	-	-	Yes	Yes	-	-
SUN	JVM	1.5, 1.6, 1.7	-	Yes	Yes	Yes	Yes	-
Open Source	OpenJDK	1.6	Linux, windows, everywhere	-	Yes	-	-	-

HP	OpenV MS	-	-	-	-	-	-	-
----	-------------	---	---	---	---	---	---	---

Notes:

- Both JDKs and JREs Supported
- For IBM JVM, a restart is required after configuring the custom memory structure.
- All JVMs must be restarted after enabling the Automatic Leak Detection feature.

JVM Language Frameworks Support

No additional configuration is required for these frameworks.

Vendor	JVM Language Framework	Version	Correlation/ Entry Points	Exit Points	Transports	Notes
Open Source / Typesafe Reactive Platform	Akka Actor	2.1 - 2.3	Yes	Yes	Netty	Remoting exit/entry supported. Persistence (experimental module in v2.3) is not currently supported.
Open Source	Groovy	-	Yes	Yes		
Open Source / Typesafe Reactive Platform	Play for Scala	2.1 - 2.3	Yes	-	HTTP over Netty	Includes framework specific entry points
Open Source / Typesafe Reactive Platform	Spray.io	-	No	No	No	Currently not supported
Pivotal	Grails	-	-	-	-	

The [Typesafe Reactive Platform](#) is a JVM-based runtime and collection of tools used to build [reactive](#) applications. This includes [Scala](#), [Play](#), [Akka](#), and [Spray.io](#).

Application Servers

These are the known application server environments in which the App Agent for Java has been used to instrument applications. Some require additional configuration. Click the link on the server or OSGi Runtime name in the following support matrix for information about additional configuration required or related configuration topics. Application servers are usually found by the App Agent for Java as an entry point.

Vendor	Application Server / OSGi Runtime	Version	SOA Protocol	RMI Supported	JMX	Entry Points
Apache	Felix	-	-	-	-	Yes
Apache	Sling	-	-	-	-	Yes
Apache	Tomcat	5.x, 6.x, 7.x	-	-	Yes	
Apache	Resin	1.x - 4.x	-	-	-	-
Adobe	Cold Fusion	8.x, 9.x	-	No	-	Yes
	Equinox	-	-	-	-	Yes
Eclipse	Jetty	6.x, 7.x	-	-	-	-
IBM	InfoSphere	8.x	-	-	-	Yes
IBM	WebSphere	6.1	JAX-WS	-	-	Yes
IBM	WebSphere	7.x	JAX-WS	Yes, detect and correlate	Yes for WebSphere PMI	Yes
Open Source	Liferay Portal	-	-	-	-	-
	GlassFish Enterprise Server	2.x	-	-	Yes	Yes
Oracle	GlassFish Server and GlassFish Server Open Source Edition	3.1+	-	-	Yes for AMX	Yes
Oracle and BEA	WebLogic Server	9.x+	JAX-WS	Yes, detect and correlate for 10.x	Yes	Yes

	Application Server (OC4J)	-	-	Yes, detect and correlate for 10.x	-	Yes
-	Grails, with Tomcat 7.x, Glassfish v3, Weblogic 12.1.1 (12c)	-	-	-	-	
-	JBoss Server	7+	-	-	-	Yes
-	JBoss Server	4.x, 5.x	-	Yes, detect and correlate	-	Yes
	JBoss AS	6.x, 7.x standalone)				
	JBoss EAP	6.11, 6.2.0				

Application Server Configuration

For application server environments that require additional configuration, this section provides some information and links to topics that help you configure the environment. Environments in the Application Server Support table that require additional configuration, link to the configuration table below.

Application Server	Topics for Required and Optional Configuration
Apache Felix	<ul style="list-style-type: none"> To configure Apache Felix for Glassfish To configure Felix for Jira or Confluence See also Unable to get metrics from the Java App Server Agent on GlassFish
Apache Sling	<ul style="list-style-type: none"> OSGi Infrastructure Configuration#To configure Apache Sling
Apache Tomcat	<ul style="list-style-type: none"> Apache Tomcat Startup Settings Tomcat as a Windows Service Configuration
Apache Resin	<ul style="list-style-type: none"> Resin Startup Settings

Apache Cold Fusion	Configuration is required for transaction discovery, see <ul style="list-style-type: none"> Servlet Entry Points
Equinox	<ul style="list-style-type: none"> To configure Eclipse Equinox
Eclipse Jetty	<ul style="list-style-type: none"> Jetty Startup Settings
IBM InfoSphere	<ul style="list-style-type: none"> IBM WebSphere and InfoSphere Startup Settings
IBM WebSphere	<ul style="list-style-type: none"> IBM WebSphere and InfoSphere Startup Settings
Sun GlassFish Enterprise Server	GlassFish JDBC connection pools can be manually configured using MBean attributes and custom JMX metrics <ul style="list-style-type: none"> GlassFish Startup Settings Modify GlassFish JVM Options
Oracle GlassFish Server	including GlassFish Server Open Source Edition <ul style="list-style-type: none"> GlassFish Startup Settings Modify GlassFish JVM Options
Oracle and BEA WebLogic Server	<ul style="list-style-type: none"> Oracle WebLogic Startup Settings
JBoss Server	<ul style="list-style-type: none"> JBoss Startup Settings

Message Oriented Middleware Support

These are the known message oriented middleware environments in which the App Agent for Java has been used to instrument applications. Some require additional configuration. Click the link on the messaging server name in the following support matrix for information about additional configuration required or related configuration topics. Message oriented middleware servers are usually found by the App Agent for Java as an entry point.

Vendor	Messaging Server	Version	Protocol	Correlation/Entry Points	Exit Points	JMX
Apache	ActiveMQ	5.x+	JMS 1.x	Yes	Yes	Yes
Apache	ActiveMQ	5.x+	STOMP	No	-	Yes
Apache	ActiveMQ	5.8.x+	AMQP 1.0	No	-	Yes
Apache	ActiveMQ	5.x+	SOAP	Yes	-	Yes
Apache	Axis	1.x, 2.x	JAX-WS	Yes	Yes	-

Apache	Apache CXF	2.1	JAX-WS	Yes	Yes	-
Apache	Synapse	2.1	HTTP	Yes	Yes	-
Fiorano	Fiorano MQ		-	-	-	-
IBM	IBM MQ	6.x, 7.x	-	-	-	-
IBM	IBM Web Application Server (WAS)	6.1+, 7.x	Embedded JMS	-	Yes	-
IBM	IBM WebSphere MQ	-	JMS	Yes	Yes	-
	JBoss MQ	4.x	-	-	-	Yes
JBoss	JBoss Messaging	5.x	-	-	-	Yes
JBoss	HornetQ	-	-	-	-	Yes
	Open MQ	-	-	-	-	-
Mulesoft	Mule ESB	3.4	HTTP	Yes	Yes	-
Oracle	Oracle AQ	-	JMS	-	Yes	-
Oracle / BEA	WebLogic	9.x+	JMS 1.1	Yes	Yes	Yes
Progress	SonicMQ	-	-	-	-	-
Pivotal	RabbitMQ	-	HTTP	-	Yes	-
Rabbit	RabbitMQ Spring Client	-	-	Yes	Yes	-
Spring	Spring Integration	2.2.0	JMS	Yes	Yes	Yes

Message Oriented Middleware Configuration

For message oriented middleware environments that require additional configuration, this section provides some information and links to topics that help you configure the environment.

Environments in the Message Oriented Middleware Support table that require additional configuration, link to the configuration table below.

Messaging Server	Topics for Required and Optional Configuration
------------------	--

Apache ActiveMQ	<ul style="list-style-type: none"> • JMS Message Queue Exit Points
Apache Axis	Default exclude rules exist for Apache Axis, Axis2, and Axis Admin Servlets. See also, <ul style="list-style-type: none"> • Web Service Entry Points
Apache Synapse	<ul style="list-style-type: none"> • To enable correlation, set node property enable-soap-header-correlation=true.
IBM MQ	No additional configuration is required. See also, Default Backends Discovered by the App Agent for Java
IBM Web Application Server	No additional configuration is required. See also, <ul style="list-style-type: none"> • JMS Message Queue Exit Points
IBM WebSphere MQ	<ul style="list-style-type: none"> • IBM Websphere MQ Message Queue Exit Points
Mule ESB	<ul style="list-style-type: none"> • Mule ESB Startup Settings • Mule ESB Support • See also HTTP Exit Points for Java
BEA WebLogic	<ul style="list-style-type: none"> • Oracle WebLogic Startup Settings
Pivotal RabbitMQ	No additional configuration is required. See also, <ul style="list-style-type: none"> • Default Backends Discovered by the App Agent for Java and • RabbitMQ Message Queue Exit Points
RabbitMQ Spring Client	No addition configuration is required, See also, <ul style="list-style-type: none"> • Message Queue Exit Points for Java
Spring Integration	<ul style="list-style-type: none"> • Spring Integration Support • See also, JMS Message Queue Exit Points

JDBC Drivers and Database Servers Support

These are the known JDBC driver and database server environments in which the App Agent for Java has been used to instrument applications. AppDynamics can follow transactions using these drivers to the designated database.

JDBC Vendor	Driver Version	Driver Type	Database Server	Database Version
Apache	10.9.1.0	Embedded or client	Derby	-

Apache	-	-	Cassandra	-
Progress	DataDirect	data connectivity for ODBC and JDBC driver access, data integration, and SaaS and cloud computing solutions	-	-
IBM	JDBC 3.0 version 3.57.82 or JDBC 4.0 version 4.7.85	DB2 Universal JDBC driver	DB2	9.x
IBM	JDBC 3.0 version 3.66.46 or JDBC 4.0 version 4.16.53	DB2 Universal JDBC driver	DB2	10.1
IBM	-	Type IV	Informix	-
Microsoft	4	Type II	MS SQL Server	2012*
Oracle MySQL, MySQL Community	5.x	Type II, Type IV	MySQL	5.x
Open Source	Connector/J 5.1.27	Type IV	MySQL	5.x
Open Source	-	Type IV	Postgres	8.x, 9.x
Oracle	9.x	Type II, Type IV	Oracle Database	8i+
Sybase	jConnect	Type IV	Sybase	-

Notes:

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

Business Transaction Error Detection

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

- Log4j
- java.util.logging

If you are using a different logger, see [Configuring Error Detection Using Custom Loggers](#).

NoSQL/Data Grids/Cache Servers Support

These are the known NoSQL, data grids and cache server environments in which the App Agent for Java has been used to instrument applications. 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.

Vendor	Database/Data Grid/Cache	Version	Correlation/Entry Points	JMX
Apache	Cassandra (Data Stax, REST) and Cassandra CQL3	-	Correlation	Yes
Apache	Apache Lucene - Apache Solr	1.4.1	Entry Points	Yes
JBoss	JBoss Cache TreeCache	-	-	-
Terracotta	EhCache	-	-	-
Open Source	Memcached	-	-	-
Open Source	MongoDB	-	-	-
Oracle	Coherence	3.7.1	Custom-Exit	Yes
JBoss	Infinispan	5.3.0+	Correlation	-

NoSQL/Data Grids/Cache Servers Configuration

For NoSQL, data grids, and cache server environments that require additional configuration, this section provides some information and links to topics that help you configure the environment.

Environments in the NoSQL/Data Grids/Cache Servers Support table that require additional configuration, link to the configuration table below.

Database/Data Grid/Cache	Topics for Required or Optional Configuration
Apache Cassandra (DataStax, REST) and Cassandra CQL3	<ul style="list-style-type: none"> Cassandra Exit Points for Java Apache Cassandra Startup Settings Default Backends Discovered by the App Agent for Java
Apache Lucene - Apache Solr	<ul style="list-style-type: none"> Solr Startup Settings
JBoss	<ul style="list-style-type: none"> JBoss Startup Settings
Terracotta EhCache	<ul style="list-style-type: none"> EhCache Exit Points
Open Source Memcached	<ul style="list-style-type: none"> Memcached Exit Points
Open Source MongoDB	<ul style="list-style-type: none"> Configurations for Custom Exit Points for Java

Oracle Coherence

- [Coherence Startup Settings](#)

Java Frameworks Support

These are the known Java framework environments in which the App Agent for Java has been used to instrument applications. Some require additional configuration. Click the link on the java framework name in the following support matrix for information about additional configuration required or related configuration topics.

Vendor	Framework	Version	SOA protocol (WebServices)	Auto Naming	Entry Points	Exit Points	Detection
Adobe	BlazeDS	-	HTTP and JMS adaptor	-	Yes		-
Adobe	ColdFusion	8.x, 9.x	-	-	Yes	-	Configuration required for transaction discovery
Apache	Cassandra with Thrift framework	-	-	-	Yes	Yes	Apache Thrift Entry and Exit points are detected
Apache	Struts	1.x, 2.x	-	-	Yes		Struts Actions are detected as entry points, struts invocation handler is instrumented
Apache	Tapestry	5	-	-	Yes	-	Not by default
	Wicket	-	-	No	Yes	-	Not by default

Apple	WebObjects	5.4.3	HTTP	Yes	Yes	-	Yes
	CometD	2.6	HTTP	Yes	Yes	-	-
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	-	-	-	-
Open Source	Direct Web Remoting (DWR)	-	-	-	-	-	-
Open Source	Enterprise Java Beans (EJB)	2.x, 3.x	-	-	Yes	-	-
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	-	Yes	Yes	-	Not by default
Open Source	Java Server Pages	2.x	-	Yes	-	-	Yes

Open Source	Java Servlet API	2.x	-	-	-	-	-
Open Source	Jersey	1.x, 2.x	REST, JAX-RS	Yes	Yes	No	Not by default
Open Source - Google	AngularJS	-	-	-	Yes	-	-
Oracle	Coherence with Spring Beans	2.x, 3.x	-	-	-	-	-
Oracle	Swing (GUI)	-	-	-	-	-	-
Oracle	WebCenter	10.0.2, 10.3.0	-	-	-	-	-
Open Source	JRuby HTTP	-	-	-	Yes	-	Not by default
Spring	Spring MVC	-	-	-	Yes	-	Not by default

Java Frameworks Configuration

For the Java framework environments that require additional configuration, this section provides some information and links to topics that help you configure the environment. Environments in the Java Frameworks Support table that require additional configuration, link to the configuration table below.

Java Framework	Topics for Required or Optional Configuration
Adobe BlazeDS	<ul style="list-style-type: none"> Message Queue Exit Points for Java
Adobe ColdFusion	Configuration is required for transaction discovery <ul style="list-style-type: none"> Java Web Application Entry Points Servlet Entry Points
Apache Cassandra with Thrift framework	No additional configuration is required. See also, <ul style="list-style-type: none"> Default Backends Discovered by the App Agent for Java
Apache Struts	<ul style="list-style-type: none"> Struts Entry Points

Apache Tapestry	<ul style="list-style-type: none"> • Java Web Application Entry Points • Servlet Entry Points
Wicket	<ul style="list-style-type: none"> • Java Web Application Entry Points • Servlet Entry Points
Apple WebObjects	Business transaction naming can be configured via getter-chains, see <ul style="list-style-type: none"> • Getter Chains in Java Configurations • Identify Transactions Based on POJO Method Invoked by a Servlet
CometD	<ul style="list-style-type: none"> • See also, HTTP Exit Points for Java
Open Source Enterprise Java Beans (EJB)	<ul style="list-style-type: none"> • EJB Entry Points
Open Source Hibernate JMS Listeners	No additional configuration is required. See also, <ul style="list-style-type: none"> • Advanced Options in Call Graphs
Open Source Java Server Faces (JSF)	<ul style="list-style-type: none"> • Java Web Application Entry Points and Servlet Entry Points
Open Source Java Server Pages	<ul style="list-style-type: none"> • Servlet Entry Points
Open Source Jersey	<ul style="list-style-type: none"> • JAX-RS Support and node properties: <ul style="list-style-type: none"> • rest-num-segments • rest-transaction • rest-uri-segment-scheme
Open Source JRuby HTTP	<ul style="list-style-type: none"> • Java Web Application Entry Points • Servlet Entry Points
Spring MVC	<ul style="list-style-type: none"> • Java Web Application Entry Points • Servlet Entry Points

RPC/Web Services API Support

These are the known Java framework environments in which the App Agent for Java has been used to instrument applications. Some require additional configuration. Click the link on the RPC, web services or API framework name in the following support matrix for information about additional configuration required or related configuration topics.

Vendor	RPC/Web Service s API Framework	Version	SOA Protocol - WebServices	Auto Naming	Correlation/Entry Points	Exit Points	Configurable BT Naming Properties	Detection
Apache	Apache CXF	2.1	JAX-WS	Yes	Yes	Yes	Yes	Yes

Apache	Apache Commons	-	HTTP Client	Yes	Yes	Yes	-	Yes
Apache	Apache Thrift	-	-	Yes	Yes	Yes	Yes	Yes
IBM	WebSphere	6.x	JAX-RPC	-	-	-	-	-
IBM	WebSphere	7.x	JAX-RPC	-	-	-	-	-
IBM	Websphere	7.x	IIOP	-	-	-	-	-
JBoss	JBoss	4.x, 5.x	RMI	Yes	Yes	Yes	Yes	Yes
Open Source	java.net. Http	-	HTTP	Yes	-	Yes	Yes	Yes
Oracle	GlassFish Metro	-	JAX-WS	-	-	-	-	-
Oracle	GlassFish Metro with Grails	-	JAX-WS	-	Yes	-	-	Not by Default
Oracle	Oracle Application Server	ORMI	-	no	-	-	-	-
Oracle	WebLogic	10.x	T3, IIOP	Yes	Correlation: Yes, Entry: No	Yes	-	Yes
Oracle	WebLogic	9.x, 10.x	JAX-RPC	-	-	-	-	-
Sun	Sun RMI	-	IIOP	-	Not by Default	-	-	-
Sun	Sun RMI	-	JRMP	-	By Default	Yes	host/port	Yes
-	Web Services	-	SOAP over HTTP	-	Yes	Yes	-	-

RPC/Web Services API Framework Configuration

For the RPC and web service API environment that require additional configuration, this section provides some information and links to topics that help you configure the environment.

Environments in the RPC/Web Services API Framework Support table that require additional configuration, link to the configuration table below.

RPC/Web Services API	Topics for Required or Optional Configuration
Apache Commons	<ul style="list-style-type: none"> • HTTP Exit Points for Java
Apache Thrift	<ul style="list-style-type: none"> • Binary Remoting Entry Points for Apache Thrift • Default Backends Discovered by the App Agent for Java
IBM WebSphere	<ul style="list-style-type: none"> • IBM WebSphere and InfoSphere Startup Settings, • App Agent for Java on z-OS or Mainframe Environments Configuration See also, <ul style="list-style-type: none"> • Unable to browse MBeans on WebSphere Application Server, • Default configuration excludes WebSphere classes
JBoss	<ul style="list-style-type: none"> • JBoss Startup Settings
Open Source java.net.Http	<ul style="list-style-type: none"> • HTTP Exit Points for Java
Oracle WebLogic	<ul style="list-style-type: none"> • Oracle WebLogic Startup Settings • Default configuration excludes WebLogic classes
Web Services	<ul style="list-style-type: none"> • Create Match Rules for Web Services • Web Service Entry Points • Web Services Exit Points for Java

Install the App Agent for Java

- [Overview of the App Agent for Java Installation Process](#)
- [Planning for Agent Installation](#)
 - [Important Files](#)
 - [To Install the Java App Server Agent](#)
 - [1. Download and unzip the App Agent for Java](#)
 - [2. Add the agent properties as a 'javaagent' argument to your JVM](#)
 - [3. Configure how the agent connects to the Controller](#)
 - [4. \(Only for Multi-tenant mode or SaaS Installations\): Configure Agent account information](#)
 - [5. Configure how the agent identifies the AppDynamics business application, tier, and node.](#)
 - [Automatic Naming for Application, Tier, and Node](#)
 - [Additional Installation Scenarios](#)

- 6. Verify agent configuration
- 7. Verify successful installation and reporting
 - a. Verify agent installation
 - b. Verify that the agent is reporting to the Controller
- [Example Configuration: App Agent for Java Deployment on a Single JVM](#)
- [Learn More](#)

The AppDynamics App Agent for Java identifies and tracks business transactions, captures statistics and diagnostic data, and analyzes and reports data to the Controller. The App Agent for Java uses [dynamic bytecode injection](#) to instrument a JVM and it runs as a part of the JVM process.

Caution: The AppDynamics Agent for Java may fail if there are other Application Performance Management (APM) products installed in the same JVM. They can coexist as long as the other APM does not interfere with the AppDynamics Agent for Java class transformations. We discourage the simultaneous use of other Byte Code Injection (BCI) agents.



Overview of the App Agent for Java Installation Process



Installing the App Agent for Java involves adding it as a javaagent ([Java Programming Language Agent](#)) on your JVM and setting up connection and identifying parameters for it to report data to the Controller.

Install the App Agent for Java as the same user or administrator of the JVM. Otherwise the agent may not have the correct write permissions for the system. The agent directories must have write permission so that AppDynamics can update the logs and other agent files.

Planning for Agent Installation

Before installing the App Agent for Java, be prepared with the following information.

	Planning Item	Description
	Where is the startup script for the JVM? If using a Java service wrapper, you need to know the location of the wrapper configuration.	This is where you can add startup arguments in the script file and system properties, if needed.
	What host and port is the Controller running on?	For SaaS customers, AppDynamics provides this information to you. For on-premise Controllers, this information is configured during Controller installation. See (Install the Controller on Linux or Install the Controller on Windows).

	To what AppDynamics business application does this JVM belong?	Usually, all JVMs in your distributed application infrastructure belong to the same AppDynamics business application. You assign a name to the business application. For details see Logical Model .
	To what AppDynamics tier does this JVM belong?	You assign a name to the tier. For details see Logical Model .

Important Files

In addition to the JVM startup script file, two other files are important during installation:

- The -javaagent argument uses the fully-qualified path of the javaagent.jar file. No separate classpath arguments need to be added.
- The <agent_home>/conf/controller-info.xml file is where you add the configuration mentioned in the planning list.

To Install the Java App Server Agent

1. Download and unzip the App Agent for Java

- Download the App Agent for Java ZIP file from [AppDynamics Download Center](#).
- Extract the ZIP file to the destination directory as the same user or administrator of the JVM.
Take note of the following:
 - Extract the agent to a directory that is outside of your container
 - All files should be readable by the agent
 - Runtime directory should be writable by the agent

Note: Do not unzip/install the agent into to the ..\tomcat\webapps directory. By default Tomcat tries to undeploy and deploy files under the webapps folder. To avoid the possibility that Tomcat will occasionally not restart, we recommend installing the agent to a directory outside of tomcat, such as \usr\local\agentsetup\AppServerAgent.

2. Add the agent properties as a 'javaagent' argument to your JVM

This step adds the agent to the startup script of your application server. Use the server-specific instructions below to add this argument for different Application Server JVMs:

3. Configure how the agent connects to the Controller

- Configure properties for the Controller host name and its port number.
- You can configure these two properties using either the controller-info.xml file or the JVM startup script:


Configure using controller-info.xml	Configure using System Properties	Required	Default
-------------------------------------	-----------------------------------	----------	---------

<controller-host>	-Dappdynamics.controller.hostName	Yes	None
<controller-port>	-Dappdynamics.controller.port	Yes	For On-premise Controller installations: By default, port 8090 is used for HTTP and 8181 is used for HTTPS communication. For SaaS Controller service: By default, port 80 is used for HTTP and 443 is used for HTTPS communication.

Optional settings for Agent-Controller communication

- To configure the Java Agent to use SSL, see [App Agent for Java Configuration Properties](#). See [Enable SSL \(Java\)](#) for instructions on *new SSL configuration in 3.7.6*
- To configure the Java Agent to use proxy settings see [App Agent for Java Configuration Properties](#)

4. (Only for Multi-tenant mode or SaaS Installations): Configure Agent account information

- This step is required only when the AppDynamics Controller is configured in [Controller Tenant Mode](#) or when you [Use a SaaS Controller](#).
 Skip this step if you are using single-tenant mode, which is the default in an on-premise installation.
- Specify the properties for Account Name and Account Key. This information is provided in the Welcome email from the AppDynamics Support Team. You can also find this information in the <controller-install>/initial_account_access_info.txt file.

Configure using controller-info.xml	Configure using System Properties	Required	Default
<account-name>	-Dappdynamics.agent.accountName	Required only if your Controller is configured for multi-tenant mode or your controller is hosted.	None.
<account-access-key>	-Dappdynamics.agent.accountAccessKey	Required only if your Controller is configured for multi-tenant mode or your controller is hosted.	None.

5. Configure how the agent identifies the AppDynamics business application, tier, and node.

To better understand agents and how they relate to business applications, tiers, and nodes see [Logical Model](#) and [Name Business Applications, Tiers, and Nodes](#).

You can configure these properties using either the controller-info.xml file or JVM startup script options. Use these guidelines when configuring agents:

- Configure items that are common for all the nodes in the controller-info.xml file.
- Configure information that is unique to a node in the startup script.

Configure using controller-info.xml	Configure using System Properties	Required	Default
<application-name>	-Dappdynamics.agent.applicationName	Yes, unless you use automatic naming	None
<tier-name>	-Dappdynamics.agent.tierName	Yes, unless you use automatic naming	None
<node-name>	-Dappdynamics.agent.nodeName	Yes, unless you use automatic naming	None

Automatic Naming for Application, Tier, and Node

The App Agent for Java javaagent command accepts an argument named uniqueID that AppDynamics uses to automatically name the node and tier for this agent. For example, using this command argument AppDynamics will name the node and tier "my-app-jvm1":

```
-javaagent:<agent_home>/javaagent.jar=uniqueID=<my-app-jvm1>
```

When uniqueID is used and the application name is not provided either through the system property or in the controller-info.xml, AppDynamics creates a new business application called "MyApp".

The naming mechanism is used by the Agent Download Wizard process. See [Quick Install](#).

Additional Installation Scenarios

Refer to the links below for typical installation scenarios, especially for cases where there are multiple JVMs on the same machine:

- [Configure App Agent for Java on Multiple JVMs on the Same Machine that Serves the Same Tier](#)
- [Configure App Agent for Java on Multiple JVMs on the Same Machine that Serve Different Tiers](#)
- [Configure App Agent for Java to Use Existing System Properties](#)
- [App Agent for Java on z-OS or Mainframe Environments Configuration](#)
- [Configure App Agent for Java for Batch Processes](#)

- [Add the Agent into an Embedded JVM](#)

6. Verify agent configuration


- Ensure that you have added `-javaagent` argument in your JVM startup script. This is not a `-D` system property but a different standard argument for all JVMs v1.5 and higher.
- Ensure that you have added all mandatory items either in the Agent controller-info.xml file or in the JVM startup script file.
- The user running the JVM process/application server process is the user accessing the Java Agent installation.

7. Verify successful installation and reporting

a. Verify agent installation

After a successful install, your agent logs, located at `<agent_home>/logs`, should contain following message:

```
Started AppDynamics Java Agent Successfully
```

 If the agent log file is not present, the App Agent for Java may not be accessing the `javaagent` command properties. The application server log file where STDOUT is logged will have the fallback log messages, for further troubleshooting.

b. Verify that the agent is reporting to the Controller

Use the AppDynamics UI, to verify that the Java Agent is able to connect to the Controller:

- Point your browser to: `http://<controller-host>:<controller-port>/controller`
- Provide the admin credentials to log into the AppDynamics UI.
- Select the application. In the left navigation pane, click **Servers -> App Servers -> <tier> -> <node>**. Click the Agents tab and App Server Agent subtab. An agent successfully reporting to the Controller will be listed and the Reporting property shows an "up" arrow symbol. For more details see [Verify App Agent-Controller Communication](#).
- When deploying multiple agents for the same tier, see if you get the exact number of nodes reporting in the same tier.

Example Configuration: App Agent for Java Deployment on a Single JVM

The following example shows a sample deployment of the App Agent for Java for the ACME Bookstore.

- Add the `javaagent` argument to the start-up script of the JVM:

```
java -javaagent:/home/appdynamics/AppServerAgent/javaagent.jar
```

- Define the five mandatory items for agent configuration in the Agent controller-info.xml file:

```
<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>
```

Learn More

- [App Agent for Java Configuration Properties](#)
- [Uninstall the App Agent for Java](#)
- [Logical Model](#)

Multi-Agent Deployment for Java

- [Deployment Procedure](#)
 - [To Deploy Java App Agents](#)
 - [To Deploy Standalone Machine Agents](#)
- [Sample Deployment Solutions](#)
- [Learn More](#)

This topic describes the high-level procedures for deploying multiple AppDynamics app agents and machine agents on Java platforms.

Deployment Procedure

To Deploy Java App Agents

1. Download the latest agent ZIP file from <http://download.appdynamics.com/>.
2. Update deployment artifacts to use the downloaded agent.
3. Unzip the downloaded app agent file on the destination machine in the desired app agent directories.
4. Modify the app-agent-config.xml file with any custom settings for the node, tier or app.
5. Do one of the following for each application server:
 - Set the application name, tier name, node name, controller host and controller port properties in the <Agent_Installation_Directory>/conf/controller-info.xml file.
OR
 - Set these properties as system startup properties in the application server startup script using the -D option.
See [App Agent for Java Configuration Properties](#) for more information about these properties.
6. Restart the application servers to make the changes take effect.

See [Install the App Agent for Java](#) for detailed instructions on installing the app agent.

To Deploy Standalone Machine Agents

1. Download the latest AppDynamics Standalone Machine Agent ZIP file from <http://download.appdynamics.com/>.
2. Unzip the downloaded file on the destination machine in the desired directories.
3. Modify the <Machine_Agent_Installation_Directory>/conf/controller-info.xml files to set the application name, tier name, node name, Controller host and Controller port properties. Note that there are no -D settings allowed for standalone machine agents, unlike app agents.
4. Configure the startup script for the machine to start the machine agent every time the machine reboots. For example, you could add the machine startup command to .bashrc.

To handle large values for metrics, run the standalone machine agent using a 64-bit JDK.

Sample Deployment Solutions

You can download some sample solutions that our customers have created to perform multi-agent AppDynamics rollouts.

Use these samples for ideas on how to automate AppDynamics agent deployment for your environment. All of these samples deploy the agents independently of the application deployment.

- ChefExample1 and ChefExample2 use Opscode Chef recipes to automate deployment on Java platforms. See <http://www.opscode.com/chef/> for information about Chef.
- JavaExample1 uses a script, configuration file and package repository.


Click below to download the samples.

- [ChefExample1.tar](#)
- [ChefExample2.tar](#)
- [JavaExample1.tar](#)

Learn More

- <https://github.com/edmunds/cookbook-appdynamics>

Java Server-Specific Installation Settings

 If you are a Self-Service Trial user, add the App Agent for Java javaagent argument to your JVM start script where <my-app-jvm1> is the name you use for the application running on that JVM.

```
-javaagent:<agent_home>/javaagent.jar=uniqueID=<my-app-jvm1>
```

See [Name Business Applications, Tiers, and Nodes](#).

Apache Cassandra Startup Settings

- [To add the javaagent command in a Windows environment](#)
- [To add the javaagent command in a Linux environment](#)

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


To add the javaagent command in a Windows environment

1. Open the apache-cassandra-x.x.x\bin\cassandra.bat file.
2. Add the AppDynamics javaagent to the JAVA_OPTS variable. Make sure to include the drive in the full path to the App Server 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
. . .
. . .
```

 If you are a Self-Service Trial user, add the App Agent for Java javaagent argument to your JVM start script where <my-app-jvm1> is the name you use for the application running on that JVM.

```
-javaagent:<agent_home>\javaagent.jar=uniqueID=<my-app-jvm1>
```

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


To add the javaagent command 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
```

 If you are a Self-Service Trial user, add the App Agent for Java javaagent argument to your JVM start script where <my-app-jvm1> is the name you use for the application running on that JVM.


```
-javaagent:<agent_home>/javaagent.jar=uniqueID=<my-app-jvm1>
```

3. Restart the Cassandra server for the changes to take effect.

Apache Tomcat Startup Settings

- [To add the javaagent command in a Windows environment](#)
- [To add the javaagent command in a Linux environment](#)


The AppDynamics Java App Server Agent bootstraps using the javaagent command line option. Add this option to your Tomcat catalina.sh or catalina.bat file.

If you are using Tomcat as a Windows service, see [Tomcat as a Windows Service Configuration](#).

To add the javaagent command in a Windows environment

1. Open the **catalina.bat** file, located at <apache_version_tomcat_install_dir>\bin.
2. Add following javaagent argument to the beginning of your application server start script.

```
if "%1"=="stop" goto skip_agent
set JAVA_OPTS=%JAVA_OPTS% -javaagent:"Drive:<agent_home>\javaagent.jar"
:skip_agent
```

 If you are a Self-Service Trial user, add the App Agent for Java javaagent argument to your JVM start script where <my-app-jvm1> is the name you use for the application running on that JVM.

```
-javaagent:"Drive:<agent_home>\javaagent.jar=uniqueID=<my-app-jvm1>"
```

The javaagent argument references the full path to the App Server Agent installation directory, including the drive. For details see the screen captures.

2a. Sample Tomcat 5.x catalina.bat file

```

catalina.bat
88 rem Add on extra jar file to CLASSPATH
89 rem Note that there are no quotes as we do not want to introduce random
90 rem quotes into the CLASSPATH
91 if "%CLASSPATH%" == "" goto emptyClasspath
92 set CLASSPATH=%CLASSPATH%;
93 :emptyClasspath
94 set CLASSPATH=%CLASSPATH%;%CATALINA_HOME%\bin\bootstrap.jar
95
96 if not "%CATALINA_BASE%" == "" goto getBase
97 set CATALINA_BASE=%CATALINA_HOME%
98 :getBase
99
100 if not "%CATALINA_TMPDIR%" == "" goto getTmpdir
101 set CATALINA_TMPDIR=%CATALINA_BASE%\temp
102 :getTmpdir
103
104 if not exist "%CATALINA_HOME%\bin\tomcat-juli.jar" goto noJuli
105 set JAVA_OPTS=%JAVA_OPTS% -Djava.util.logging.manager=org.apache.juli.ClassLoaderLogManager -Djava.util.logging.config.file="%CATALINA_BASE%\
106 :noJuli
107
108 if "%1"=="stop" goto skip_agent
109 set JAVA_OPTS=%JAVA_OPTS% -javaagent:"D:\Appdynamics\192\javaagent.jar"
110 :skip_agent
111
112 rem ----- Execute The Requested Command -----
113
114 echo Using CATALINA_BASE:  %CATALINA_BASE%
115 echo Using CATALINA_HOME:  %CATALINA_HOME%
116 echo Using CATALINA_TMPDIR: %CATALINA_TMPDIR%
117 if ""%1"" == ""debug"" goto use_jdk
118 echo Using JRE_HOME:       %JRE_HOME%
119 goto java_dir_displayed
120 :use_jdk
121 echo Using JAVA_HOME:      %JAVA_HOME%
122 :java_dir_displayed
123 echo Using CLASSPATH:      %CLASSPATH%
124
125 set _EXECJAVA=%_RUNJAVA%
126 set MAINCLASS=org.apache.catalina.startup.Bootstrap
127 set ACTION=start
128 set SECURITY_POLICY_FILE=
129 set DEBUG_OPTS=
130 set JPDA=
131
132 if not ""%1"" == ""jpda"" goto noJpda
133 set JPDA=jpda

```

2b. Sample Tomcat 6.x catalina.bat file

```

catalina.bat
118
119 if not "%CATALINA_BASE%" == "" goto gotBase
120 set CATALINA_BASE=%CATALINA_HOME%
121 :gotBase
122
123 if not "%CATALINA_TMPDIR%" == "" goto gotTmpdir
124 set CATALINA_TMPDIR=%CATALINA_BASE%\temp
125 :gotTmpdir
126
127 if not "%LOGGING_CONFIG%" == "" goto noJuliConfig
128 set LOGGING_CONFIG=-Dnop
129 if not exist "%CATALINA_BASE%\conf\logging.properties" goto noJuliConfig
130 set LOGGING_CONFIG=-Djava.util.logging.config.file="%CATALINA_BASE%\conf\logging.properties"
131 :noJuliConfig
132 set JAVA_OPTS=%JAVA_OPTS% %LOGGING_CONFIG%
133
134 if not "%LOGGING_MANAGER%" == "" goto noJuliManager
135 set LOGGING_MANAGER=-Djava.util.logging.manager=org.apache.juli.ClassLoaderLogManager
136 :noJuliManager
137 set JAVA_OPTS=%JAVA_OPTS% %LOGGING_MANAGER%
138
139 if "%1"=="stop" goto skip_agent
140 set JAVA_OPTS=%JAVA_OPTS% -javaagent:"D:\Appdynamics\192\javaagent.jar"
141 :skip_agent
142
143
144 rem ----- Execute The Requested Command -----
145
146 echo Using CATALINA_BASE:  %CATALINA_BASE%
147 echo Using CATALINA_HOME:  %CATALINA_HOME%
148 echo Using CATALINA_TMPDIR: %CATALINA_TMPDIR%
149 if "%1"=="debug" goto use_jdk
150 echo Using JRE_HOME:       %JRE_HOME%
151 goto java_dir_displayed
152 :use_jdk
153 echo Using JAVA_HOME:      %JAVA_HOME%
154 :java_dir_displayed
155

```

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

To add the javaagent command in a Linux environment

1. Open the catalina.sh file located at <apache_version_tomcat_install_dir>/bin).
2. Add the following commands at the beginning of your application server start script.

```

if [ "$1" = "start" -o "$1" = "run" ]; then
export JAVA_OPTS="$JAVA_OPTS -javaagent:agent_install_dir/javaagent.jar"
fi

```

The javaagent argument references the full path to the App Server Agent installation directory.

⚠ If you are a Self-Service Trial user, add the App Agent for Java javaagent argument to your JVM start script where <my-app-jvm1> is the name you use for the application running on that JVM.

```

-javaagent:<agent_home>/javaagent.jar=uniqueID=<my-app-jvm1>

```

For details see the screen captures.

2a. Sample Tomcat 5.x catalina.sh file

```

catalina.sh
148 have_tty=0
149 if [ "$tty" != "not a tty" ]; then
150     have_tty=1
151 fi
152
153 # For Cygwin, switch paths to Windows format before running java
154 if $cygwin; then
155     JAVA_HOME=`cygpath --absolute --windows "$JAVA_HOME"`
156     JRE_HOME=`cygpath --absolute --windows "$JRE_HOME"`
157     CATALINA_HOME=`cygpath --absolute --windows "$CATALINA_HOME"`
158     CATALINA_BASE=`cygpath --absolute --windows "$CATALINA_BASE"`
159     CATALINA_TMPDIR=`cygpath --absolute --windows "$CATALINA_TMPDIR"`
160     CLASSPATH=`cygpath --path --windows "$CLASSPATH"`
161     JAVA_ENDORSED_DIRS=`cygpath --path --windows "$JAVA_ENDORSED_DIRS"`
162 fi
163
164 # Set juli LogManager if it is present
165 if [ -r "$CATALINA_HOME/bin/tomcat-juli.jar" ]; then
166     JAVA_OPTS="$JAVA_OPTS -Djava.util.logging.manager=org.apache.juli.ClassLoaderLogManager"
167     LOGGING_CONFIG="-Djava.util.logging.config.file=$CATALINA_BASE/conf/logging.properties"
168 else
169     # Bugzilla 45585
170     LOGGING_CONFIG="-Dnop"
171 fi
172
173 if [ "$1" = "start" -o "$1" = "run" ]; then
174     export JAVA_OPTS="$JAVA_OPTS -javaagent:/xnt/agenttest/agent192-2/javaagent.jar"
175 fi
176
177 # ----- Execute The Requested Command -----
178
179 # Bugzilla 37848: only output this if we have a TTY
180 if [ $have_tty -eq 1 ]; then
181     echo "Using CATALINA_BASE: $CATALINA_BASE"
182     echo "Using CATALINA_HOME: $CATALINA_HOME"
183     echo "Using CATALINA_TMPDIR: $CATALINA_TMPDIR"
184     if [ "$1" = "debug" ]; then
185         echo "Using JAVA_HOME: $JAVA_HOME"
186     else
187         echo "Using JRE_HOME: $JRE_HOME"
188     fi
189 fi

```

2b. Sample Tomcat 6.x catalina.sh file

```

catalina.sh
199 fi
200
201 # Set juli LogManager config file if it is present and an override has not been issued
202 if [ -z "$LOGGING_CONFIG" ]; then
203     if [ -r "$CATALINA_BASE/conf/logging.properties" ]; then
204         LOGGING_CONFIG="-Djava.util.logging.config.file=$CATALINA_BASE/conf/logging.properties"
205     else
206         # Bugzilla 45585
207         LOGGING_CONFIG="-Dnop"
208     fi
209 fi
210
211 if [ -z "$LOGGING_MANAGER" ]; then
212     JAVA_OPTS="$JAVA_OPTS -Djava.util.logging.manager=org.apache.juli.ClassLoaderLogManager"
213 else
214     JAVA_OPTS="$JAVA_OPTS $LOGGING_MANAGER"
215 fi
216
217 if [ "$1" = "start" -o "$1" = "run" ]; then
218     export JAVA_OPTS="$JAVA_OPTS -javaagent:/mnt/agenttest/agent192/javaagent.jar"
219 fi
220
221 # ----- Execute The Requested Command -----
222
223 # Bugzilla 37848: only output this if we have a TTY
224 if [ $have_tty -eq 1 ]; then
225     echo "Using CATALINA_BASE:  $CATALINA_BASE"
226     echo "Using CATALINA_HOME:   $CATALINA_HOME"
227     echo "Using CATALINA_TMPDIR:  $CATALINA_TMPDIR"
228     if [ "$1" = "debug" ]; then
229         echo "Using JAVA_HOME:         $JAVA_HOME"
230     else
231         echo "Using JRE_HOME:           $JRE_HOME"
232     fi
233     echo "Using CLASSPATH:          $CLASSPATH"
234 fi
235
236 if [ "$1" = "jps" ]; then
237     if [ -z "$JPDA_TRANSPORT" ]; then

```

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

Tomcat as a Windows Service Configuration

- To install the javaagent as a Tomcat Windows service

The AppDynamics Java App Server Agent bootstraps using the javaagent command line option. Add this option to your Tomcat properties.

If you are not running Tomcat as a Windows service, see [Apache Tomcat Startup Settings](#).

To install the javaagent as a Tomcat Windows service

These instructions apply to Apache Tomcat 6.x or later versions.

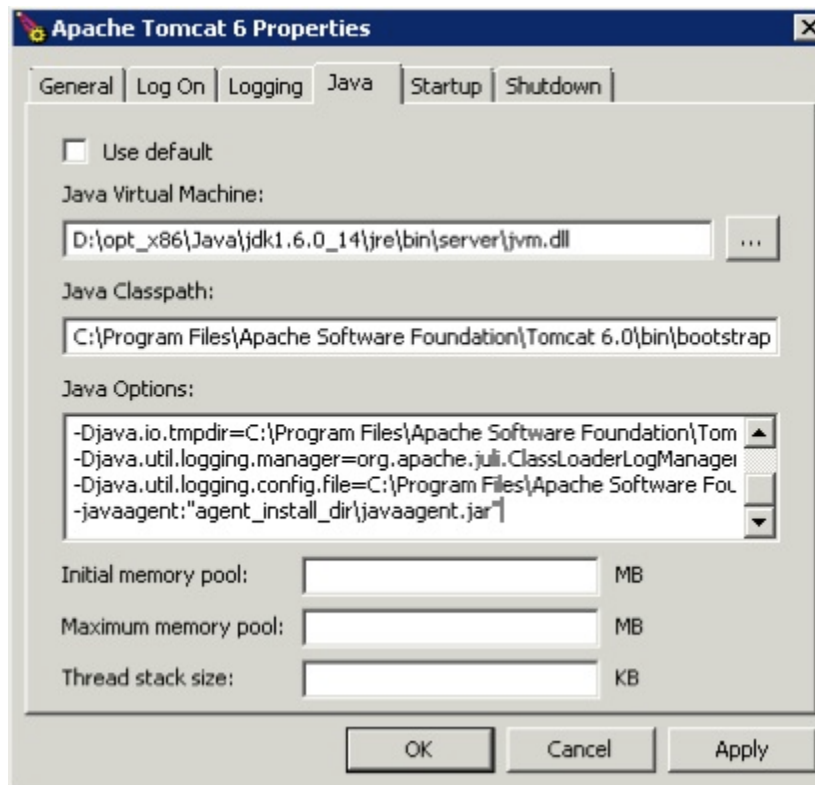
1. Ensure that you are using administrator privileges.
2. Click **Programs -> Apache Tomcat**.
3. Run **Configure Tomcat**.
4. Click the **Java** tab.
5. In the **Java Options** add:

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

! If you are a Self-Service Trial user, add the App Agent for Java javaagent argument to your JVM start script where <my-app-jvm1> is the name you use for the application running on that JVM.

```
-javaagent:"<agent_home>\javaagent.jar=uniqueID=<my-app-jvm1>"
```

For details see the following screenshot.



6. Restart the Tomcat service. The application server must be restarted for the changes to take effect.

Coherence Startup Settings

In the cache-server.sh file update the following:

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

GlassFish Startup Settings


- To add the javaagent command in a GlassFish 3.0 environment
- To add the javaagent command in a GlassFish 3.1 environment
- To verify the Agent configuration
- About Glassfish AMX Support

The AppDynamics Java App Server Agent bootstraps using the javaagent command line option.

To add the javaagent command in a GlassFish 3.0 environment

1. If you are using **GlassFish v3.0**, first configure the OSGi containers. For details see [OSGi Infrastructure Configuration](#).
2. Log into the **GlassFish domain** where you want to install the App Server Agent.
3. In the left navigation tree **Common Tasks** section, click **Application Server**. The Application Server Settings dialog opens.
4. In the **JVM Settings** tab, click **JVM Options**.
5. Click **Add JVM Option** and add an entry for the javaagent argument. The javaagent argument contains the full path, including the drive, of the App Server Agent installation directory.

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

 If you are a Self-Service Trial user, add the App Agent for Java javaagent argument to your JVM start script where <my-app-jvm1> is the name you use for the application running on that JVM.

```
-javaagent:<drive>:\<agent_home>\javaagent.jar=uniqueID=<my-app-jvm1>
```




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

To add the javaagent command in a GlassFish 3.1 environment

1. If you are using **GlassFish v3.0**, first configure the OSGi containers. For details see [OSGi Infrastructure Configuration](#).
2. Log into the **GlassFish domain** where you want to install the App Server Agent.

Note: Remember to turn on remote administration by entering the following command from the <Controller_home>/appserver/glassfish/bin directory:

```
asadmin enable-secure-admin
```

3. In the **Configurations** section in the left navigation tree, click **server-config** and then click **JVM Settings**.
4. On the **JVM Settings** tab, click **JVM Options**.


5. Click **Add JVM Option** and add an entry for the javaagent argument as follows:

For Windows:

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

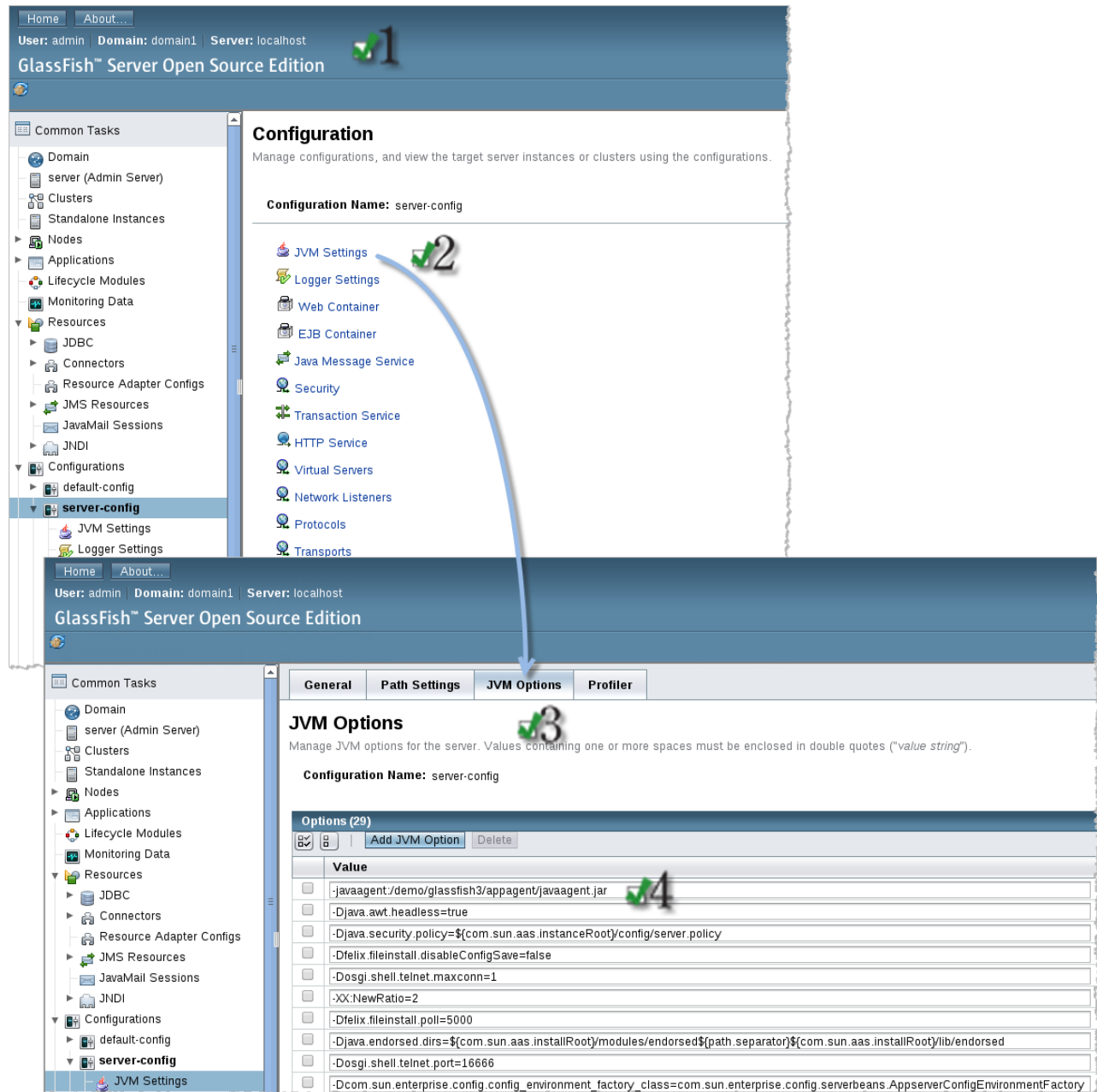
For Linux:

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

 If you are a Self-Service Trial user, add the App Agent for Java javaagent argument to your JVM start script where <my-app-jvm1> is the name you use for the application running on that JVM.

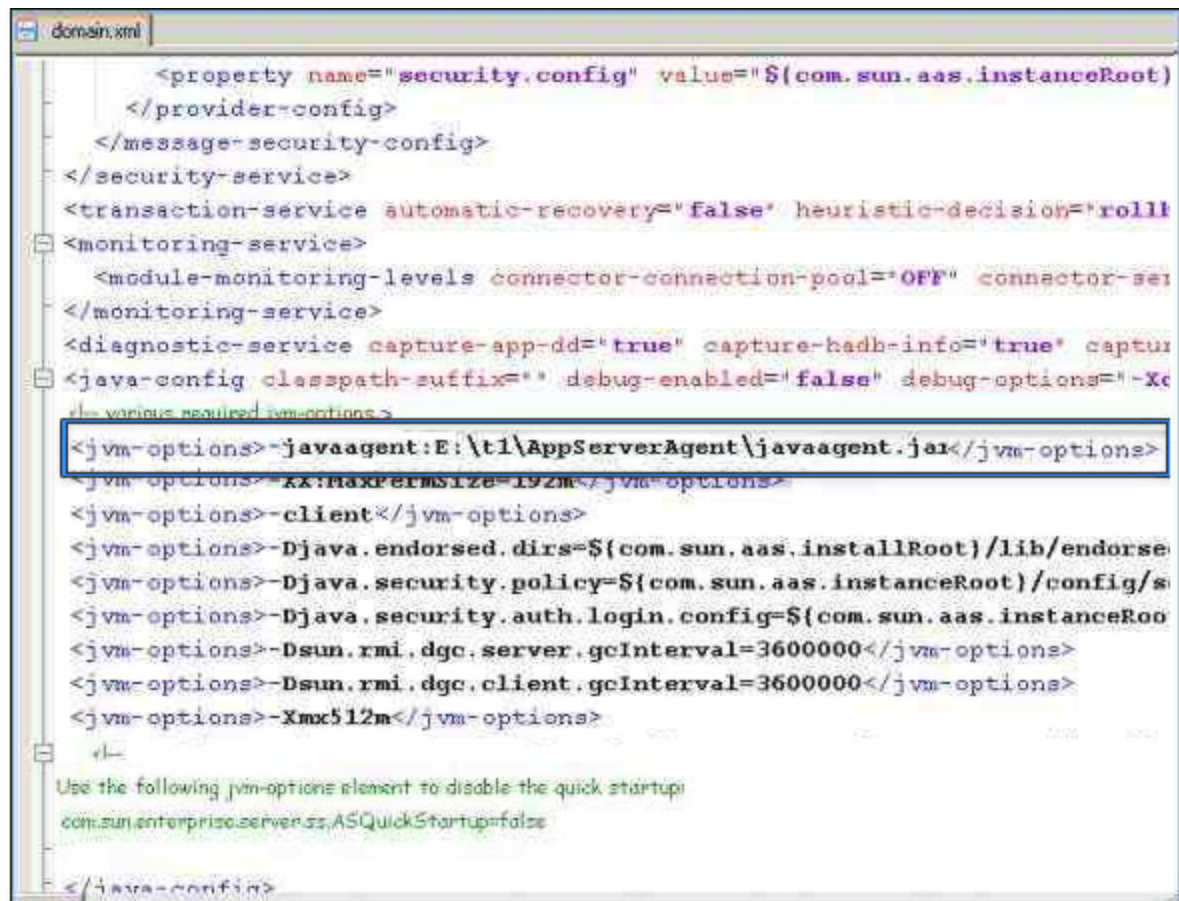
```
-javaagent:<drive>:\<agent_home>\javaagent.jar=uniqueID=<my-app-jvm1>
```

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



To verify the Agent configuration

To verify this configuration, look at the domain.xml file located at <glassfish_install_dir>\domains\<domain_name>. The domain.xml file should have an entry as shown in the following screenshot.



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

- To add the javaagent command in a WebSphere 7.x and InfoSphere 8.x environment
- To add the javaagent command in a WebSphere 6.x environment
- To add the javaagent command in a WebSphere 5.x environment
- To verify the Agent configuration
- Security Requirements
- Running WebSphere with Security Enabled

The AppDynamics Java App Server Agent bootstraps using the javaagent command line option.

To add the javaagent command in a WebSphere 7.x and InfoSphere 8.x environment

1. Log in to the **Administrator** console of the WebSphere node where you want to install the App Server Agent.
2. In the Administration Console click **Servers**.


3. Expand **Server Type** and click **WebSphere application servers**.
4. Click the name of your server.
5. Expand **Java and Process Management** and click **Process Definition**.
6. Under the **Additional Properties** section, click **Java Virtual Machine**.
7. Enter the javaagent option with the full path to the AppDynamics 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
```

 If you are a Self-Service Trial user, add the App Agent for Java javaagent argument to your JVM start script where <my-app-jvm1> is the name you use for the application running on that JVM.

```
-javaagent:<drive>:\<agent_home>\javaagent.jar=uniqueID=<my-app-jvm1>
```

8. Click **OK**.

To add the javaagent command in a WebSphere 6.x environment

1. Log in to the **Administrator** console of the WebSphere node where you want to install the App Server Agent.
2. In the left navigation tree, click **Servers -> Application servers**.
3. Click the name of your server in the list of servers.

For quick access, place your bookmarks here in the bookmarks bar.

Integrated Solutions Console Welcome jpowar [Help](#) | [Logout](#) 

View: All tasks ▾

- Welcome
- Guided Activities
- Servers
 - Application servers
 - Web servers
 - WebSphere MQ servers
- Applications
 - Enterprise Applications
 - Install New Application
- Resources
- Security
- Environment

Application servers

Application servers

Use this page to view a list of the application servers in your environment. You can also use this page to change the status of a specific application server.

Preferences

Name	Node	Version
server1	ww-84f6cf8ea82aNode01	Base 6.1.0.0

Total 1

4. In the "Configuration" tab, click **Java and Process Management**.

Application servers > server1

Use this page to configure an application server. An application server is a server that provides services required to run enterprise applications.

Runtime | **Configuration**

General Properties

Name
server1

Node name
rajendraNode01

☐ Run in development mode

☒ Parallel start

☐ Start components as needed

Access to internal server classes
Allow

Server-specific Application Settings

ClassLoader policy
Multiple

Class loading mode
Classes loaded with parent class loader first

Apply OK Reset Cancel

Container Settings

- Session management
- + SIP Container Settings
- + Web Container Settings
- + Portlet Container Settings
- + EJB Container Settings
- + Container Services
- + Business Process Services

Applications

- Installed applications

Server messaging

- Messaging engines
- Messaging engine inbound transports
- WebSphere MQ link inbound transports
- SIB service

Server Infrastructure

- + Java and Process Management
- + Administration

Communications

- + Ports

5. Enter the javaagent option with the full path to the AppDynamics 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
```

⚠ If you are a Self-Service Trial user, add the App Agent for Java javaagent argument to your JVM start script where <my-app-jvm1> is the name you use for the application running on that JVM.

```
-javaagent:<drive>:\<agent_home>\javaagent.jar=uniqueID=<my-app-jvm1>
```

8. Click **OK**.

The screenshot shows the 'General Properties' tab of the WebSphere Administrative Console. The 'Classpath' and 'Boot Classpath' fields are empty. Below them are checkboxes for 'Verbose class loading', 'Verbose garbage collection', and 'Verbose JNI'. The 'Initial Heap Size' and 'Maximum Heap Size' fields are also empty. There are checkboxes for 'Run HProf' and 'Debug Mode'. The 'HProf Arguments' field is empty. The 'Debug arguments' field contains '-Djava.compile=NONE -xdeb'. The 'Generic JVM arguments' field is highlighted with a green oval and contains '-javaagent:E:\test1\AppServer\javaagent.jar=uniqueID=AppServer1'. The 'Additional Properties' tab is also visible, showing a 'Custom Properties' section.

To add the javaagent command in a WebSphere 5.x environment


1. Log in to the **Administrator** console of the WebSphere node where you want to install the App Server Agent.
2. In the Administrative Console, click **Servers**.
3. Select **Application Servers**.
4. Click the name of your server.
5. Under **Additional Properties**, select **Process Definition**.
6. On the next page, under **Additional Properties** select **Java Virtual Machine**.
7. Enter the javaagent option with the full path to the AppDynamics 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
```

 If you are a Self-Service Trial user, add the App Agent for Java javaagent argument to your JVM start script where <my-app-jvm1> is the name you use for the application running on that JVM.

```
-javaagent:<drive>:\<agent_home>\javaagent.jar=uniqueID=<my-app-jvm1>
```

8. Click **OK**.

To verify the Agent configuration

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

```
<jvmEntries ...&nbsp;
genericJvmArguments='-javaagent:E:\test1\AppServerAgent\javaagent.jar'
disableJIT="false"/>
```

Security Requirements

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 within the interaction between WebSphere and the Java agent. Make the change listed below to the server.policy file, which is located in <websphere_home>/properties or in <websphere_profile_home>/properties. Add the following block to the WebSphere server.policy file:

```
grant codeBase "file:\* AGENT_DEPLOYMENT_DIRECTORY \*/-"
{
    permission java.security.AllPermission;
};
```

WebSphere in z-OS or Mainframe Environments

See [Configure App Agent for Java in z-OS or Mainframe Environments](#).

JBoss Startup Settings


- To add the javaagent command in a Windows environment
- To add the javaagent command in a Linux environment for JBoss 5.x
- To add the javaagent command in a Linux environment for JBoss AS 6.x
- To add the javaagent command in a Linux environment for JBoss EAP 6.1.1, EAP 6.2.0, and JBoss AS 7.0.x (standalone)
- To add the javaagent command in a Linux environment for JBoss 7.1.1
- To add the javaagent command in a Linux environment for JBoss AS 7.x
- To add the javaagent command in a Windows environment for JBoss AS 7.x
- To add the javaagent command to RHEL JBoss EAP 6.x, JBoss AS 7.0.x, JBoss 8 (Domain Mode)
 - Revise the Domain.xml file
 - Revise the Host.xml file
- To add the agent to JBoss 7.2 (standalone)
- Fix Linkage Error

The AppDynamics Java App Server Agent bootstraps using the javaagent command line option. Add this option to your JBoss server run.sh or run.bat file.

To add the javaagent command in a Windows environment

1. Open the server run.bat file, located at <jboss_version_install_directory>\bin.
2. Add the following javaagent argument at the beginning of your app server start script.

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

 If you are a Self-Service Trial user, add the App Agent for Java javaagent argument to your JVM start script where <my-app-jvm1> is the name you use for the application running on that JVM.

```
-javaagent:"<drive>:\<agent_home>\javaagent.jar=uniqueID=<my-app-jvm1>"
```

The javaagent argument references the full path of the App Server Agent installation directory, including the drive. For details see the screen captures.

3. Sample JBoss 4.x run.bat file
 - a. Sample JBoss 4.x run.bat file:

```

run.bat
62
63 rem If JBOSS_CLASSPATH is empty, don't include it, as this will
64 rem result in including the local directory, which makes error tracking
65 rem harder.
66 if "%JBOSS_CLASSPATH%" == "" (
67     set JBOSS_CLASSPATH=%JAVAC_JAR%;%RUNJAR%
68 ) else (
69     set JBOSS_CLASSPATH=%JBOSS_CLASSPATH%;%JAVAC_JAR%;%RUNJAR%
70 )
71
72 rem Setup JBoss specific properties
73 set JAVA_OPTS=%JAVA_OPTS% -Dprogram.name=%PROGNAME%
74 set JBOSS_HOME=%DIRNAME%\..
75
76 rem Add -server to the JVM options, if supported
77 "%JAVA%" -version 2>&1 | findstr /I hotspot > nul
78 if not errorlevel == 1 (set JAVA_OPTS=%JAVA_OPTS% -server)
79
80 rem JVM memory allocation pool parameters. Modify as appropriate.
81 set JAVA_OPTS=%JAVA_OPTS% -Xms128m -Xmx512m
82
83 rem With Sun JVMs reduce the RMI GCs to once per hour
84 set JAVA_OPTS=%JAVA_OPTS% -Dsun.rmi.dgc.client.gcInterval=3600000 -Dsun.rmi.dgc.server.gcInterval=3600000
85
86 set JAVA_OPTS=%JAVA_OPTS% -javaagent:"E:\tl\AppServerAgent\javaagent.jar"
87 rem JPIA options. Uncomment and modify as appropriate to enable remote debugging.
88 rem set JAVA_OPTS=%JAVA_OPTS% -Xdebug -Xrunjdwp:transport=dt_socket,address=8787,server=y,suspend=y %JAVA_OPTS%
89
90 rem Setup the java endorsed dirs
91 set JBOSS_ENDORSED_DIRS=%JBOSS_HOME%\lib\endorsed
92
93 echo =====
94 echo.
95 echo.      JBoss Bootstrap Environment
96 echo.
97 echo.      JBOSS_HOME: %JBOSS_HOME%
98 echo.
99 echo.      JAVA: %JAVA%
100 echo.
101 echo.      JAVA_OPTS: %JAVA_OPTS%
102

```

b. Sample JBoss 5.x run.bat file

```

run.bat
97
98 rem If JBOSS_CLASSPATH empty, don't include it, as this will
99 rem result in including the local directory in the classpath, which makes
100 rem error tracking harder.
101 if "%JBOSS_CLASSPATH%" == "" (
102     set "RUN_CLASSPATH=%RUNJAR%"
103 ) else (
104     set "RUN_CLASSPATH=%JBOSS_CLASSPATH%;%RUNJAR%"
105 )
106
107 set JBOSS_CLASSPATH=%RUN_CLASSPATH%
108
109 rem Setup JBoss specific properties
110 rem JVM memory allocation pool parameters. Modify as appropriate.
111 set JAVA_OPTS=%JAVA_OPTS% -Xms128m -Xmx512m -XX:MaxPermSize=256m
112
113 rem Warn when resolving remote XML dtd/schemas
114 set JAVA_OPTS=%JAVA_OPTS% -Dorg.jboss.resolver.warning=true
115
116 rem With Sun JVMs reduce the RMI GCs to once per hour
117 set JAVA_OPTS=%JAVA_OPTS% -Dsun.rmi.dgc.client.gcInterval=3600000 -Dsun.rmi.dgc.server.gcInterval=3600000
118
119 set JAVA_OPTS=%JAVA_OPTS% -javaagent:"E:\t1\AppServerAgent\javaagent.jar"
120 rem JPDA options. Uncomment and modify as appropriate to enable remote debugging.
121 rem set JAVA_OPTS=%JAVA_OPTS% -Xdebug -Xrunjdwp:transport=dt_socket,address=8787,server=y,suspend=y
122
123 rem Setup the java endorsed dirs
124 set JBOSS_ENDORSED_DIRS=%JBOSS_HOME%\lib\endorsed
125
126 echo =====
127 echo.
128 echo JBoss Bootstrap Environment
129 echo.
130 echo JBOSS_HOME: %JBOSS_HOME%
131 echo.
132 echo JAVA: %JAVA%
133 echo.
134 echo JAVA_OPTS: %JAVA_OPTS%
135 echo.
136 echo CLASSPATH: %JBOSS_CLASSPATH%

```

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

To add the javaagent command in a Linux environment for JBoss 5.x

1. Open the server run.sh file, located at <jboss_version_install_dir>/bin.
2. Add the following javaagent argument to the server start script.

```
export JAVA_OPTS="$JAVA_OPTS -javaagent: /<agent_home>/javaagent.jar"
```

- a. Sample JBoss 5.x run.sh file

```

130     fi
131
132     # Enable -server if we have Hotspot, unless we can't
133     if [ "x$HAS_HOTSPOT" != "x" ]; then
134         # MacOS does not support -server flag
135         if [ "$darwin" != "true" ]; then
136             JAVA_OPTS="-server $JAVA_OPTS"
137         fi
138     fi
139 fi
140
141 # Setup JBoss sepecific properties
142 JAVA_OPTS="-Dprogram.name=$PROGNAME $JAVA_OPTS"
143
144 # Setup the java endorsed dirs
145 JBOSS_ENDORSED_DIRS="$JBOSS_HOME/lib/endorsed"
146
147 # For Cygwin, switch paths to Windows format before running java
148 if $cygwin; then
149     JBOSS_HOME=`cygpath --path --windows "$JBOSS_HOME"`
150     JAVA_HOME=`cygpath --path --windows "$JAVA_HOME"`
151     JBOSS_CLASSPATH=`cygpath --path --windows "$JBOSS_CLASSPATH"`
152     JBOSS_ENDORSED_DIRS=`cygpath --path --windows "$JBOSS_ENDORSED_DIRS"`
153 fi
154 export JAVA_OPTS="$JAVA_OPTS -javaagent:/home/AppServerAgent/javaagent.jar"
155 # Display our environment
156 echo "=====
157 echo ""
158 echo " JBoss Bootstrap Environment"
159 echo ""
160 echo " JBOSS_HOME: $JBOSS_HOME"

```

- b. ⚠ If you are a Self-Service Trial user, add the App Agent for Java javaagent argument to your JVM start script where <my-app-jvm1> is the name you use for the application running on that JVM.

```
-javaagent:<agent_home>/javaagent.jar=uniqueID=<my-app-jvm1>
```

3. Restart the application server. You must restart the application server for the changes to take effect.

To add the javaagent command in a Linux environment for JBoss AS 6.x

1. Open the server run.sh file, located at <jboss_version_install_dir>/bin.
2. Add the following Java environment variables to the server start script.

```

JAVA_OPTS="$JAVA_OPTS
-Djava.util.logging.manager=org.jboss.logmanager.LogManager"
JAVA_ARGS="$JAVA_OPTS
-Dorg.jboss.logging.Logger.pluginClass=org.jboss.logging.logmanager.Logger
PluginImpl"
JBOSS_CLASSPATH=<path>"jboss-logmanager.jar"

```

3. Add the following javaagent argument to the server start script.

```
export JAVA_OPTS="$JAVA_OPTS -javaagent:/agent_install_dir/javaagent.jar"
```

4. Restart the application server. You must restart the application server for the changes to take effect.

To add the javaagent command in a Linux environment for JBoss EAP 6.1.1, EAP 6.2.0, and JBoss AS 7.0.x (standalone)

1. Open the standalone.sh file.
2. Add the following and save the file:

```

JAVA_OPTS="$JAVA_OPTS
-Djava.util.logging.manager=org.jboss.logmanager.LogManager
-Xbootclasspath/p:/<path/to/jboss-eap-6.1.1>/modules/system/layers/base/or
g.jboss.logmanager/main/jboss-logmanager-1.4.3.Final-redhat-1.jar:/<path/t
o/jboss-eap-6.1.1>/modules/system/layers/base/org.jboss.log4j/logmanager/m
ain/log4j-jboss-logmanager-1.0.2.Final-redhat-1.jar"JAVA_OPTS="$JAVA_OPTS
-javaagent:/<path/to/appdynamics>/javaagent.jar"
JAVA_OPTS="$JAVA_OPTS
-Djboss.modules.system.pkgs=org.jboss.byteman,com.appdynamics,com.appdynam
ics.,com.singularity,com.singularity."

```

Note: Substitute <path/to/jboss-eap-6.1.1> and <path/to/appdynamics> with the correct respective paths for your installation.

3. Restart the application server. You must restart the application server for the changes to take effect.
4. Integrate error detection with the jboss log manager logging implementation using a custom logger definition as described in [Configuring Error Detection Using Custom Loggers](#).

To add the javaagent command in a Linux environment for JBoss 7.1.1

1. Open the standalone.sh file
2. Search for the following line in standalone.sh.

```
# Setup the JVM
```

3. Add the following above that section and save the file.

```

export JAVA_OPTS="$JAVA_OPTS -javaagent:<agent-path>/javaagent.jar
-Dorg.jboss.boot.log.file=$JBOSS_HOME/standalone/log/boot.log
-Djava.util.logging.manager=org.jboss.logmanager.LogManager"
export JAVA_OPTS="$JAVA_OPTS
-Xbootclasspath/p:$JBOSS_HOME/modules/org/apache/log4j/main/log4j-1.2.16.j
ar:$JBOSS_HOME/modules/org/jboss/logmanager/log4j/main/jboss-logmanager-lo
g4j-1.0.0.GA.jar:$JBOSS_HOME/modules/org/jboss/logmanager/main/jboss-logma
nager-1.2.2.GA.jar
-Dlogging.configuration=file:$JBOSS_HOME/standalone/configuration/logging.
properties"
#export JAVA_OPTS="$JAVA_OPTS
-Xbootclasspath/p:$JBOSS_HOME/modules/org/jboss/logmanager/main/jboss-logm
anager-1.2.2.GA.jar
-Dlogging.configuration=file:$JBOSS_HOME/standalone/configuration/logging.
properties"

```

Note: Substitute <agent-path> with the correct path for your installation.

4. Open `standalone.conf` and search for the following.

```
# Uncomment the following line to prevent manipulation of JVM options
```

5. Add the following above that section and save the file.

```

if [ "x$JBOSS_MODULES_SYSTEM_PKGS" = "x" ]; then
JBOSS_MODULES_SYSTEM_PKGS="org.jboss.byteman,com.appdynamics,com.appdynami
cs.,com.singularity,com.singularity.,org.jboss.logmanager"
# JBOSS_MODULES_SYSTEM_PKGS="org.jboss.byteman"
fi

```

6. Restart the application server. You must restart the application server for the changes to take effect.

To add the javaagent command in a Linux environment for JBoss AS 7.x

1. Open the `standalone.conf` file.
2. Search for the following line in `standalone.conf`.

```
JBOSS_MODULES_SYSTEM_PKGS="org.jboss.byteman"
```

3. Add the `com.singularity` and `org.jboss.logmanager` packages to that line as follows:

```
JBOSS_MODULES_SYSTEM_PKGS="org.jboss.byteman,com.singularity,org.jboss.log
manager"
```

4. Add the following to the end of the `standalone.conf` file in the `JAVA_OPTS` section.

```
-Djava.util.logging.manager=org.jboss.logmanager.LogManager
-Xbootclasspath/p:<JBoss-DIR>/modules/org/jboss/logmanager/main/jboss-logmanager-1.2.2.GA.jar
:<JBoss-DIR>/modules/org/jboss/logmanager/log4j/main/jboss-logmanager-log4j-1.0.0.GA.jar
:<JBoss-DIR>/modules/org/jboss/logmanager/log4j/main/log4j-1.2.16.jar
```

Note: The path for the necessary JAR files may differ for different versions. Provide the correct path of these JAR files for your version. If any of the packages are not available with the JBoss ZIP, download the missing package and add it to the path.

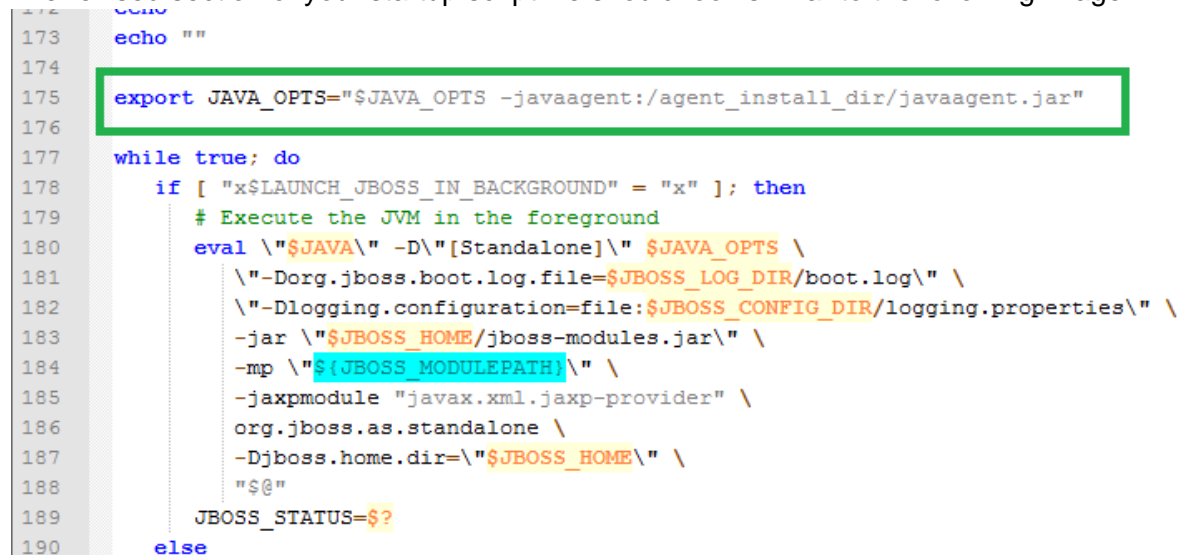
5. In the standalone.sh file, add the following javaagent argument.

```
export JAVA_OPTS="$JAVA_OPTS -javaagent:/agent_install_dir/javaagent.jar"
```

above the following section of standalone.sh

```
...
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\" \
```

The revised section of your startup script file should look similar to the following image:



```
173 echo ""
174
175 export JAVA_OPTS="$JAVA_OPTS -javaagent:/agent_install_dir/javaagent.jar"
176
177 while true; do
178   if [ "x$LAUNCH_JBOSS_IN_BACKGROUND" = "x" ]; then
179     # Execute the JVM in the foreground
180     eval \"\$JAVA\" -D\"[Standalone]\" \"\$JAVA_OPTS \
181       \"-Dorg.jboss.boot.log.file=$JBoss_LOG_DIR/boot.log\" \
182       \"-Dlogging.configuration=file:$JBoss_CONFIG_DIR/logging.properties\" \
183       -jar \"\$JBoss_HOME/jboss-modules.jar\" \
184       -mp \"${JBoss_MODULEPATH}\" \
185       -jaxpmodule "javax.xml.jaxp-provider" \
186       org.jboss.as.standalone \
187       -Djboss.home.dir=\"\$JBoss_HOME\" \
188       \"$@"
189     JBoss_STATUS=$?
190   else
```

6. Restart the application server.

To add the javaagent command in a Windows environment for JBoss AS 7.x

1. Open the bin\standalone.conf file.

2. Search for the line `JBOSS_MODULES_SYSTEM_PKGS="org.jboss.byteman"` and add the `com.singularity` and `org.jboss.logmanager` packages to that line as follows:

```
JBOSS_MODULES_SYSTEM_PKGS="org.jboss.byteman,com.singularity,org.jboss.logmanager"
```

3. Save the file.
4. Open the `standalone.bat` file.
5. Add the following `javaagent` argument to the `standalone.bat` file.

```
:RESTART
"%JAVA%" \-javaagent:<AGENT-DIR>javaagent.jar %JAVA_OPTS%
"-Dorg.jboss.boot.log.file=%JBOSS_HOME%\standalone\log\boot.log"
"-Dlogging.configuration=[file:%JBOSS_HOME%\standalone/configuration\logging.properties]" \-jar "%JBOSS_HOME%\jboss-modules.jar"
```

6. Save the file.
7. Restart the application server. The application server must be restarted for the changes to take effect.

To add the `javaagent` command to RHEL JBoss EAP 6.x, JBoss AS 7.0.x, JBoss 8 (Domain Mode)

You must add a system property to allow the `com.singularity` classes in the AppDynamics agent to be found from any class loader.

Add the `jvm` options specifying the location of the agent jar file, the application name, and the tier name.

- If all the server instances in the server group are part of the same business application, then configure `-Dappdynamics.agent.applicationName` in `domain.xml`; otherwise, configure the application name in the `host.xml` file for each specific server.
- If all the server instances in the server group are part of the same tier then configure `-Dappdynamics.agent.tierName` in `domain.xml`, otherwise configure the tier name in `host.xml` for each specific server.

`-Dappdynamics.agent.applicationName` tells the AppDynamics agents the name of the Business Application to be used to connect to the AppDynamics Controller.

`-Dappdynamics.agent.tierName` tells the AppDynamics agents the name of the tier to use to connect to the AppDynamics Controller.

Revise the JBoss `domain.xml` and `host.xml` files as indicated in the following sections and then restart the application server.

Revise the `Domain.xml` file

1. Locate and edit `domain.xml`, usually located under `$JBOSS_HOME/domain/configuration/`.
2. Add the following system property, `<property name="jboss.modules.system.pkgs" value="com.singularity" />` in the `<system-properties>` element.


```
<system-properties>
  <!-- IPv4 is not required, but setting this helps avoid unintended
  use of IPv6 -->
  <property name="java.net.preferIPv4Stack" value="true"/>
  <property name="jboss.modules.system.pkgs"
  value="com.singularity"/>
</system-properties>
```

This property tells the JBoss modules to allow the com.singularity classes in the AppDynamics App Agent for Java to be found from any class loader. This is required for the agent to run.

3. Under the server group name where you want to enable your AppDynamics agents, add the JVM options using the appropriate values for your agent location, JBoss application name, and tier name.

```
<server-group name="main-server-group" profile="full">
  <jvm name="default">
    <heap size="1303m" max-size="1303m"/>
    <permgen max-size="256m"/>
    <jvm-options>
      <option value="-javaagent:<agent_install_dir>/javaagent.jar"/>
      <option value="-Dappdynamics.agent.applicationName=JBOSS-EAP-APP"/>
      <option value="-Dappdynamics.agent.tierName=JBOSS-EAP-TIER"/>
    </jvm-options>
  </jvm>
  <socket-binding-group ref="full-sockets"/>
</server-group>
```

Revise the Host.xml file

Add the -Dappdynamics.agent.nodeName jvm option in the host.xml file (usually located under \$JBOSS_HOME/domain/configuration/). This option tells the AppDynamics agent the node name to use to connect to the AppDynamics Controller. Use the appropriate values for your node names.

For example:

```
<servers>
  <server name="server-one" group="main-server-group">
    <!-- Remote JPDA debugging for a specific server
    <option
value="-agentlib:jdwp=transport=dt_socket,address=8787,server=y,suspend=n"/>
    -->
    <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">
    <!-- server-two avoids port conflicts by incrementing the ports in
    the default socket-group declared in the server-group -->
    <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">
    <!-- server-three avoids port conflicts by incrementing the ports in
    the default socket-group declared in the server-group -->
    <socket-bindings port-offset="250"/>
  </server>
</servers>
```

To add the agent to JBoss 7.2 (standalone)

1. Add the following to the standalone.sh file.

```

AD_AGENT_HOME="/Users/jack.ginnever/Downloads/AppD-Downloads/AppServerAgent/3.8.1.0/AppServerAgent"
AD_CONT_HOST="localhost"
AD_CONT_POST="8090"
AD_APPL_NAME="JBossAS"
AD_APPL_TIER="standalone"
AD_APPL_NODE="jboss_node"

AD_OPTS=" -javaagent:$AD_AGENT_HOME/javaagent.jar \
-Dappdynamics.controller.hostName=$AD_CONT_HOST \
-Dappdynamics.controller.port=$AD_CONT_POST \
-Dappdynamics.agent.applicationName=$AD_APPL_NAME \
-Dappdynamics.agent.tierName=$AD_APPL_TIER \
-Dappdynamics.agent.nodeName=$AD_APPL_NODE "
# Fix up the Loggers and Bootclasspath
AD_OPTS="$AD_OPTS
-Dorg.jboss.boot.log.file=$JBASS_HOME/standalone/log/boot.log \
-Djava.util.logging.manager=org.jboss.logmanager.LogManager"
AD_OPTS="$AD_OPTS
-Xbootclasspath/p:$JBASS_HOME/modules/org/apache/log4j/main/log4j-1.2.16.jar:$JBASS_HOME/modules/org/jboss/log4j/logmanager/main/log4j-jboss-logmanager-1.0.1.Final.jar:$JBASS_HOME/modules/org/jboss/logmanager/main/jboss-logmanager-1.4.0.Final.jar
-Dlogging.configuration=file:$JBASS_HOME/standalone/configuration/logging.properties"

JAVA_OPTS="$AD_OPTS $JAVA_OPTS"

```

2. Add the following to the standalone.conf file and save it.

```

# --> Commented out original mod to JBOSS_MODULES_SYSTEM_PKGS
#
# if [ "x$JBOSS_MODULES_SYSTEM_PKGS" = "x" ]; then
#   JBOSS_MODULES_SYSTEM_PKGS="org.jboss.byteman"
# fi
#
# --> Replaced with following mod to JBOSS_MODULES_SYSTEM_PKGS
#
if [ "x$JBOSS_MODULES_SYSTEM_PKGS" = "x" ]; then
JBOSS_MODULES_SYSTEM_PKGS="org.jboss.byteman,com.appdynamics,com.appdynamics.,com.singularity,com.singularity."
fi

```

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

Fix Linkage Error

If you see this error:

```
Caused by: java.lang.LinkageError: loader constraint violation in interface
itable initialization: when resolving method
"com.microsoft.sqlserver.jdbc.SQLServerXAConnection.getXAResource()Ljavax/transaction/xa/XAResource;" the class loader (instance of
org/jboss/modules/ModuleClassLoader) of the current class,
com/microsoft/sqlserver/jdbc/SQLServerXAConnection, and the class loader
(instance of <bootloader>) for interface javax/sql/XAConnection have different
Class objects for the type javax/transaction/xa/XAResource used in the signature
```

Add the following JVM option to the start-up script and restart the server:

```
-Dappdynamics.bciengine.class.lookahead=!*
```

The error indicates a race condition while loading classes and this flag controls the class loading hierarchy thus overcoming the class loading problems.

Jetty Startup Settings


- [To add the javaagent command in a Jetty environment](#)

The AppDynamics Java App Server Agent bootstraps using the javaagent command line option. Add this option to your jetty.sh file.

To add the javaagent command in a Jetty environment

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
```

 If you are a Self-Service Trial user, add the App Agent for Java javaagent argument to your JVM start script where <my-app-jvm1> is the name you use for the application running on that JVM.

```
-javaagent:<agent_home>/javaagent.jar=uniqueID=<my-app-jvm1>
```

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

Mule ESB Startup Settings

The AppDynamics Java App Server Agent bootstraps using the javaagent command line option. Mule ESB 3.X or later uses a Tanuki configuration environment. To add the AppDynamics App Agent for Java to your Mule ESB environment, add this option to the Tanuki Service wrapper.conf file.

To configure the Tanuki Service Wrapper

1. Open the wrapper.conf file.
2. Use the wrapper.java.additional.<n> property to add the javaagent option.
3. On Linux systems, in the <MULE_HOME>/conf/wrapper.conf file, add the following

```
wrapper.java.additional.4=-javaagent:"/Users/hbrien/Software/mule-enterprise-standalone-3.3.1/app_agent/javaagent.jar
wrapper.java.additional.4.stripquotes=TRU
wrapper.java.additional.X=-javaagent:"/opt/app_agent/javaagent.jar

wrapper.java.additional.X.stripquotes=TRUE
```

Where "X" is a unique integer among the other properties in the file.

Learn More

Mule ESB Support

Oracle WebLogic Startup Settings


- [To add the javaagent command in a Windows environment](#)
- [To add the javaagent command in a application running as a Windows service](#)
- [To add the javaagent command in a Linux environment](#)

The AppDynamics Java App Server Agent bootstraps using the javaagent command line option. Add this option to your startWebLogic.sh or startWebLogic.cmd file.

To add the javaagent command in a Windows environment

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 beginning of your application server start script.

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

 If you are a Self-Service Trial user, add the App Agent for Java javaagent argument to your JVM start script where <my-app-jvm1> is the name you use for the application running on that JVM.

```
-javaagent:"<drive>:\<agent_home>\javaagent.jar=uniqueID=<my-app-jvm1>"
```

The javaagent argument references the full path of the App Server Agent installation directory, including the drive.

2a. Sample WebLogic v9.x startWebLogic.cmd file

```

startWebLogic.cmd - Notepad
File Edit Format View Help

if NOT "%Interface%"==" " (
) else (
    set IFNAME=%Interface%
)
set IFNAME=

@REM Set IP Mask.
if NOT "%NetMask%"==" " (
) else (
    set IPMASK=%NetMask%
)
set IPMASK=

@REM Perform IP Migration if SERVER_IP is set by node manager.
if NOT "%SERVER_IP%"==" " (
    call "%WL_HOME%\common\bin\wlsifconfig.cmd" -addif "%IFNAME%" "%SERVER_IP%" "%IPMASK%"
)

set JAVA_OPTIONS=%JAVA_OPTIONS% -javaagent:C:\users\Arunkumar\Documents\bea9x\singularity\Agent_wl_9_springs_domain\javaagent.jar
@REM START WEBLOGIC

echo starting weblogic with java version:
%JAVA_HOME%\bin\java %JAVA_VM% -version

if "%WLS_REDIRECT_LOG%"==" " (
    echo Starting WLS with line:
    echo %JAVA_HOME%\bin\java %JAVA_VM% %MEM_ARGS% %JAVA_OPTIONS% -Dweblogic.Name=%SERVER_NAME% -Djava.security.policy=%WL_HOME%

```

2b. Sample WebLogic v10.x startWebLogic.cmd file

```

startWebLogic.cmd
165 ) else (
166     set IPMASK=
167 )
168
169 @REM Perform IP Migration if SERVER_IP is set by node manager.
170
171 if NOT "%SERVER_IP%"==" " (
172     call "%WL_HOME%\common\bin\wlsifconfig.cmd" -addif "%IFNAME%" "%SERVER_IP%" "%IPMASK%"
173 )
174
175 set JAVA_OPTIONS=%JAVA_OPTIONS% -javaagent:"E:\tl\AppServerAgent\javaagent.jar"
176 @REM START WEBLOGIC
177
178 echo starting weblogic with Java version:
179
180 %JAVA_HOME%\bin\java %JAVA_VM% -version
181
182 if "%WLS_REDIRECT_LOG%"==" " (
183     echo Starting WLS with line:
184     echo %JAVA_HOME%\bin\java %JAVA_VM% %MEM_ARGS% %JAVA_OPTIONS% -Dweblogic.Name=%SERVER_
185     %JAVA_HOME%\bin\java %JAVA_VM% %MEM_ARGS% %JAVA_OPTIONS% -Dweblogic.Name=%SERVER_NAME%
186 ) else (
187     echo Redirecting output from WLS window to %WLS_REDIRECT_LOG%
188     %JAVA_HOME%\bin\java %JAVA_VM% %MEM_ARGS% %JAVA_OPTIONS% -Dweblogic.Name=%SERVER_NAME%
189 )

```

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

To add the javaagent command in a 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 AppDynamics Agent Start
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. Remove the existing Windows Service for your application. From the command line, run this script.

```
install_XXXXXX_Server_Start_Win_Service.cmd
XXXXXX_xxxxxx_Production_Server R
```

4. Install the Windows Service for your application to include the AppDynamics agent JAVA_OPTIONS argument.

```
install_XXXXXX_Server_Start_Win_Service.cmd
XXXXXX_xxxx_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 = XXXXX_xxxx_Production_Server.

7. Ensure that your application is working properly.

For more information, see [Creating a Server-Specific Script](#) in the Oracle documentation.

To add the javaagent command in a Linux environment

1. Open the startWebLogic.sh file, located at
<weblogic_<version#>_install_dir>/user_projects/domains/<domain_name>/bin.

2. Add the following lines of code to the beginning of your application server start script.


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

⚠ If you are a Self-Service Trial user, add the App Agent for Java javaagent argument to your JVM start script where <my-app-jvm1> is the name you use for the application running on that JVM.

```
-javaagent:<agent_home>/javaagent.jar=uniqueID=<my-app-jvm1>
```

The javaagent argument references the full path of the App Server Agent installation directory. For details see the screen captures.

2a. Sample WebLogic v9.x startWebLogic.sh file



```
167 if [ "${SERVER_IP}" != "" ] ; then
168     ${WL_HOME}/common/bin/wlsifconfig.sh -addif "${IFNAME}" "${SERVER_IP}" "${IPMASK}"
169 fi
170
171 # START WEBLOGIC
172
173 echo "starting weblogic with Java version:"
174
175 ${JAVA_HOME}/bin/java ${JAVA_VM} -version
176
177 if [ "${WLS_REDIRECT_LOG}" = "" ] ; then
178     echo "Starting WLS with line:"
179     echo "${JAVA_HOME}/bin/java ${JAVA_VM} ${MEM_ARGS} ${JAVA_OPTIONS} -Dweblogic.Name=
180     ${JAVA_HOME}/bin/java ${JAVA_VM} ${MEM_ARGS} ${JAVA_OPTIONS} -Dweblogic.Name=
181 else
182     echo "Redirecting output from WLS window to ${WLS_REDIRECT_LOG}"
183     ${JAVA_HOME}/bin/java ${JAVA_VM} ${MEM_ARGS} ${JAVA_OPTIONS} -Dweblogic.Name=
184 fi
185
186 stopAll
187
188 popd
189 export JAVA_OPTS="$JAVA_OPTS -javaagent:E:/t1/AppServerAgent/javaagent.jar"
190 # Exit this script only if we have been told to exit.
191
192 if [ "${doExitFlag}" = "true" ] ; then
193     exit
194 fi
```

2b. Sample WebLogic v10.x startWebLogic.sh file


```

startWebLogic.sh
167 if [ "${SERVER_IP}" != "" ] ; then
168     ${WL_HOME}/common/bin/wlsifconfig.sh -addif "${IFNAME}" "${SERVER_IP}" "${IPMASK}"
169 fi
170
171 # START WEBLOGIC
172
173 echo "starting weblogic with Java version:"
174
175 ${JAVA_HOME}/bin/java ${JAVA_VM} -version
176
177 if [ "${WLS_REDIRECT_LOG}" = "" ] ; then
178     echo "Starting WLS with line:"
179     echo "${JAVA_HOME}/bin/java ${JAVA_VM} ${MEM_ARGS} ${JAVA_OPTIONS} -Dweblogic.Name="
180     ${JAVA_HOME}/bin/java ${JAVA_VM} ${MEM_ARGS} ${JAVA_OPTIONS} -Dweblogic.Name=
181 else
182     echo "Redirecting output from WLS window to ${WLS_REDIRECT_LOG}"
183     ${JAVA_HOME}/bin/java ${JAVA_VM} ${MEM_ARGS} ${JAVA_OPTIONS} -Dweblogic.Name=
184 fi
185
186 stopAll
187
188 popd
189 export JAVA_OPTS="${JAVA_OPTS} -javaagent:E:/tl/AppServerAgent/javaagent.jar"
190 # Exit this script only if we have been told to exit.
191
192 if [ "${doExitFlag}" = "true" ] ; then
193     exit
194 fi

```

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

OSGi Infrastructure Configuration

- Configuring OSGi Containers
 - To configure Eclipse Equinox
 - To configure Apache Sling
 - To configure Apache Felix for GlassFish
 - For GlassFish 3.1.2
 - To configure Felix for Jira or Confluence
 - To configure other OSGi-based containers

Configuring OSGi Containers

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 App Server Agent classes.

For more information see [GlassFish OSGi Configuration per Domain](#).

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

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

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

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

org.osgi.framework.bootdelegation=com.sun.btrace., **com.singularity**.

To configure Eclipse Equinox

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

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

For more information see [Getting Started with Equinox](#).

To configure Apache Sling

1. Open the sling.properties file. The location of the sling.properties 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.*
```

To configure Apache Felix for GlassFish

1. Open the config.properties file, located at <glassfish-install>/glassfish/osgi/felix/conf.
2. Add following package prefix to the config.properties file.

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

For GlassFish 3.1.2

Add:

```
com.singularity.*
```

to the boot delegation list in the <GlassFish_Home_Directory>\glassfish\config\osgi.properties file.
For example:

```
org.osgi.framework.bootdelegation=${eclipselink.bootdelegation}, com.sun.btrace,
com.singularity.*
```

To configure Felix for Jira or Confluence

For Jira 5.1.8 and newer, Confluence 5.3 and newer

1. Update the jira startup script (e.g. catalina) with the following java system property:

```
-Datlassian.org.osgi.framework.bootdelegation=META-INF.services,com.yourkit,com.
singularity.*,com.jprofiler,com.jprofiler.*,org.apache.xerces,org.apache.xerces.
*,org.apache.xalan,org.apache.xalan.*,sun.*,com.sun.jndi,com.icl.saxon,com.icl.s
axon.*,javax.servlet,javax.servlet.*,com.sun.xml.bind.*
```

2. Update the Java options:

- For Linux: JAVA_OPTS=
- For Windows: set JAVA_OPTS=%JAVA_OPTS%

```
-javaagent:/root/AppServerAgent/javaagent.jar"
```

To configure other OSGi-based containers

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

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

Resin Startup Settings

- [To Configure Resin 1.x - 3.x](#)
 - [To add the javaagent command in a Windows environment](#)
 - [To add the javaagent command in a Linux environment](#)
- [To Configure Resin 4.x](#)


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

To Configure Resin 1.x - 3.x

To add the javaagent command in a Windows environment

1. Open the resin.bat file, located at <Resin_installation_directory>/bin.
2. Add following javaagent argument to the beginning of your application server start script.

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

 If you are a Self-Service Trial user, add the App Agent for Java javaagent argument to your JVM start script where <my-app-jvm1> is the name you use for the application running on that JVM.

```
-javaagent:"<drive>:\<agent_home>\javaagent.jar=uniqueID=<my-app-jvm1>"
```


The javaagent argument references the full path of the App Server Agent installation directory, including the drive.

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

To add the javaagent command in a Linux environment

1. Open the resin.sh file, located at <Resin_Installation_Directory>/bin.
2. Add the following javaagent argument to the beginning of your application server start script.

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

 If you are a Self-Service Trial user, add the App Agent for Java javaagent argument to your JVM start script where <my-app-jvm1> is the name you use for the application running on that JVM.

```
-javaagent:<agent_home>/javaagent.jar=uniqueID=<my-app-jvm1>
```

The javaagent argument references the full path of the App Server Agent installation directory. See the following screenshot.

```

/mnt/resin-pro-4.0.18/bin/resin.sh - root@>ec2-184-72-162-149.compute-1.amazonaws.com

#
# See contrib/init.resin for /etc/rc.d/init.d startup script
#
# resin.sh      -- execs resin in the foreground
# resin.sh start -- starts resin in the background
# resin.sh stop  -- stops resin
# resin.sh restart -- restarts resin
#
# resin.sh will return a status code if the wrapper detects an error, but
# some errors, like bind exceptions or Java errors, are not detected.
#
# To install, you'll need to configure JAVA_HOME and RESIN_HOME and
# copy contrib/init.resin to /etc/rc.d/init.d/resin.  Then
# use "unix# /sbin/chkconfig resin on"

if test -n "${JAVA_HOME}"; then
    if test -z "${JAVA_EXE}"; then
        JAVA_EXE=${JAVA_HOME}/bin/java
    fi
fi

if test -z "${JAVA_EXE}"; then
    JAVA_EXE=java
fi

#
# trace script and simlinks to find the wrapper
#
if test -z "${RESIN_HOME}"; then
    script="/bin/ls -l $0 | awk '{ print $NF; }'"

    while test -h "$script"
    do
        script="/bin/ls -l $script | awk '{ print $NF; }'"
    done

    bin=`dirname $script`
    RESIN_HOME="$bin/.."
fi

exec $JAVA_EXE -javaagent:"/mnt/agent/3.2/javaagent.jar" -jar ${RESIN_HOME}/lib/resin.jar $*

```

Line: 42/42 Column: 57 Character: 32 (0x20)

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

To Configure Resin 4.x

1. To install the App Server Agent into Resin 4.X or later, edit the `./conf/resin.xml` file and add:

```
<jvm-arg>-Xmx512m</jvm-arg>
<jvm-arg>-javaagent:<$appagent_location>/javaagent.jar</jvm-arg>
```

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

Solr Startup Settings


- To add the javaagent command in a Windows environment
- To add the javaagent command in a Linux environment

The AppDynamics Java App Server Agent bootstraps using the javaagent command line option. Add this option to your Solr server.

To add the javaagent command in a Windows environment

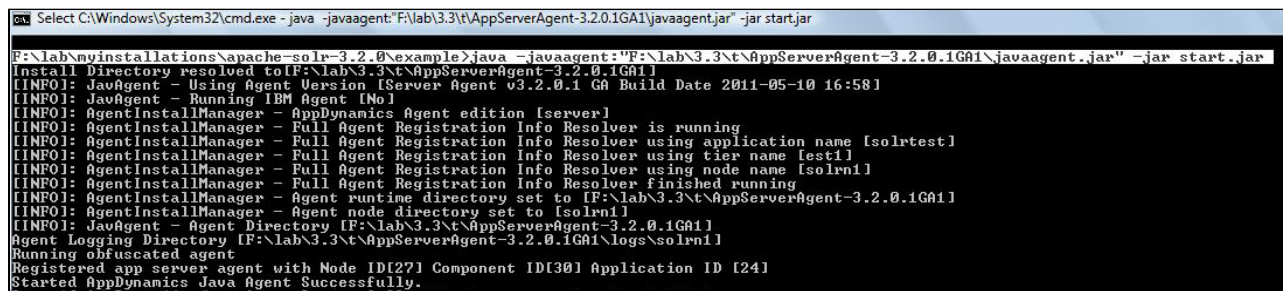
1. Open the Windows command line utility.
2. Execute the following commands to add the javaagent argument to the Solr server:

```
>cd $Solr_Installation_Directory
>java -javaagent:"<drive>:\<agent_home>\javaagent.jar" -jar start.jar
```

 If you are a Self-Service Trial user, add the App Agent for Java javaagent argument to your JVM start script where <my-app-jvm1> is the name you use for the application running on that JVM.

```
-javaagent:"<drive>:\<agent_home>\javaagent.jar=uniqueID=<my-app-jvm1>"
```

The javaagent argument references the full path to the App Server Agent installation directory, including the drive. For details see the screenshots.



```
cmd Select C:\Windows\System32\cmd.exe - java -javaagent:"F:\lab\3.3\t\AppServerAgent-3.2.0.1GA1\javaagent.jar" -jar start.jar
F:\lab\myinstallations\apache-solr-3.2.0\example>java -javaagent:"F:\lab\3.3\t\AppServerAgent-3.2.0.1GA1\javaagent.jar" -jar start.jar
Install Directory resolved to[F:\lab\3.3\t\AppServerAgent-3.2.0.1GA1]
[INFO]: JavAgent - Using Agent Version [Server Agent v3.2.0.1 GA Build Date 2011-05-10 16:58]
[INFO]: JavAgent - Running IBM Agent [No]
[INFO]: AgentInstallManager - AppDynamics Agent edition [server]
[INFO]: AgentInstallManager - Full Agent Registration Info Resolver is running
[INFO]: AgentInstallManager - Full Agent Registration Info Resolver using application name [solrtest]
[INFO]: AgentInstallManager - Full Agent Registration Info Resolver using tier name [est1]
[INFO]: AgentInstallManager - Full Agent Registration Info Resolver using node name [solrn1]
[INFO]: AgentInstallManager - Full Agent Registration Info Resolver finished running
[INFO]: AgentInstallManager - Agent runtime directory set to [F:\lab\3.3\t\AppServerAgent-3.2.0.1GA1]
[INFO]: AgentInstallManager - Agent node directory set to [solrn1]
[INFO]: JavAgent - Agent Directory [F:\lab\3.3\t\AppServerAgent-3.2.0.1GA1]
Agent Logging Directory [F:\lab\3.3\t\AppServerAgent-3.2.0.1GA1\logs\solrn1]
Running obfuscated agent
Registered app server agent with Node ID[271] Component ID[30] Application ID [24]
Started AppDynamics Java Agent Successfully.
```

To add the javaagent command in a Linux environment

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


```
>cd $Solr_Installation_Directory
>java -javaagent:"<agent_home>/javaagent.jar" -jar start.jar
```

⚠ If you are a Self-Service Trial user, add the App Agent for Java javaagent argument to your JVM start script where <my-app-jvm1> is the name you use for the application running on that JVM.

```
-javaagent:<agent_home>/javaagent.jar=uniqueID=<my-app-jvm1>
```

The javaagent argument references the full path to the App Server Agent installation directory. For details see the screenshot.

```
root@domU-12-31-39-05-75-42: /mnt/solr/apache-solr-3.2.0/example
root@domU-12-31-39-05-75-42:/mnt/solr/apache-solr-3.2.0# ls
CHANGES.txt LICENSE.txt NOTICE.txt README.txt client contrib dist docs example
root@domU-12-31-39-05-75-42:/mnt/solr/apache-solr-3.2.0# cd example/
root@domU-12-31-39-05-75-42:/mnt/solr/apache-solr-3.2.0/example# java -javaagent:"/mnt/agent/3.2/test/javaagent.jar" -jar start.jar
Install Directory resolved to [/mnt/agent/3.2/test]
[INFO]: JavAgent - Using Agent Version [Server Agent v3.2.1.0 GA Build Date 2011-06-03 20:53]
[INFO]: JavAgent - Running IBM Agent [No]
[INFO]: AgentInstallManager - AppDynamics Agent edition [server]
[INFO]: AgentInstallManager - Full Agent Registration Info Resolver is running
[INFO]: AgentInstallManager - Full Agent Registration Info Resolver using application name [casstest]
[INFO]: AgentInstallManager - Full Agent Registration Info Resolver using tier name [test1]
[INFO]: AgentInstallManager - Full Agent Registration Info Resolver using node name [nodetestcassedra]
[INFO]: AgentInstallManager - Full Agent Registration Info Resolver finished running
[INFO]: AgentInstallManager - Agent runtime directory set to [/mnt/agent/3.2/test]
[INFO]: AgentInstallManager - Agent node directory set to [nodetestcassedra]
[INFO]: JavAgent - Agent Directory [/mnt/agent/3.2/test]
Agent Logging Directory [/mnt/agent/3.2/test/logs/nodetestcassedra]
Running obfuscated agent
Registered app server agent with Node ID[28] Component ID[31] Application ID [25]
Started AppDynamics Java Agent Successfully.
2011-06-08 12:24:06.068:INFO::Logging to STDERR via org.mortbay.log.StdErrLog
2011-06-08 12:24:06.359:INFO::jetty-6.1-SNAPSHOT
```

Standalone JVM Startup Settings

- To add the javaagent command in a Windows environment
- To add the javaagent command in a Linux environment

AppDynamics works just as well with JVMs that are not application servers or containers.

The AppDynamics Java App Server Agent bootstraps using the javaagent command line option, a standard Java option and can be used with any JVM. Add this option to your standalone JVM.


To add the javaagent command in a Windows environment

1. Open the command line utility for Windows.
2. Add javaagent argument to the standalone JVM:

```
>java -javaagent:"Drive:\agent_home\javaagent.jar"
<fully_qualified_class_name_with_main_method>
```

For example:

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

 If you are a Self-Service Trial user, add the App Agent for Java javaagent argument to your JVM start script where <my-app-jvm1> is the name you use for the application running on that JVM.

```
-javaagent:"<drive>:\<agent_home>\javaagent.jar=uniqueID=<my-app-jvm1>"
```

The javaagent argument references the full path to the App Server Agent installation directory, including the drive.


To add the javaagent command in a Linux environment

1. Open the terminal.
2. Add the javaagent argument to the standalone JVM:

```
>java -javaagent:"/agent_install_dir/javaagent.jar"  
<fully_qualified_class_name_with_main_method>
```

For example:

```
>java -javaagent:"/mnt/AppDynamics/agentDir/javaagent.jar" com.main.HelloWorld
```

 If you are a Self-Service Trial user, add the App Agent for Java javaagent argument to your JVM start script where <my-app-jvm1> is the name you use for the application running on that JVM.

```
-javaagent:<agent_home>/javaagent.jar=uniqueID=<my-app-jvm1>
```

The javaagent argument references the full path to the App Server Agent installation directory.

Tanuki Service Wrapper Configuration


- [To configure the Tanuki Service Wrapper](#)

The AppDynamics Java App Server Agent bootstraps using the javaagent command line option. Add this option to the Tanuki Service wrapper.conf file.

To configure the Tanuki Service Wrapper

1. Open the wrapper.conf file.
2. Use the wrapper.java.additional.<n> property to add the javaagent option.


```
wrapper.java.additional.6=-javaagent:/C:/agent/javaagent.jar
```

 If you are a Self-Service Trial user, add the App Agent for Java javaagent argument to your JVM start script where <my-app-jvm1> is the name you use for the application running on that JVM.

```
-javaagent:<agent_home>/javaagent.jar=uniqueID=<my-app-jvm1>
```

For more information see:

- [Tanuki Service Wrapper Properties](#)
- [Example Configuration](#)
- [More Help On Tanuki Service Wrapper](#)

Tibco ActiveMatrix BusinessWorks Service Engine Configuration

There are typically two scripts associated with the Tibco ActiveMatrix BusinessWorks Services engine.

- my_application.sh
- my_application.tra

The JVM that runs the services start with a command line tool called bwengine(.exe).

Add the following to the .tra file:

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

See also: <https://docs.tibco.com/> and <https://tibbr.tibcommunity.com/>.

SUN JDK 1.6 on Linux

The App Agent for Java calls an API to collect CPU times for threads. Because of a [Sun JDK 1.6 problem on Linux](#) the API may take too long, and you may see the following error:

```
threads blocking on
sun.management.ThreadImpl.getThreadTotalCpuTime0(Native Method).
```

If this error occurs, the App Agent for Java by default will disable CPU time collection for threads. You can force the agent to continue to collect CPU times for threads using the [thread-cpu-capture-overhead-threshold-in-ms](#) node property, but this may cause unacceptable overhead on your application. We recommend you use this [Java HotSpot VM option](#) instead to speed up the API call itself.

```
-XX:+UseLinuxPosixThreadCPUClocks
```

Enable SSL for Java

- [Before You Begin](#)
 - [SSL Utilities](#)
 - [Keystore Certificate Extractor Utility](#)
 - [Password Encryption Utility](#)
- [SaaS Controller](#)
 - [Sample SaaS SSL controller.xml configuration](#)
 - [Sample SaaS SSL JVM startup script configuration](#)
- [On-Premise Controller with a Trusted CA Signed Certificate](#)
 - [Sample on-premise SSL controller.xml configuration for a CA signed certificate](#)
 - [Sample on-premise SSL JVM startup script configuration for a CA signed certificate](#)
- [On-Premise Controller with an Internally Signed Certificate](#)
 - [Sample on-premise SSL controller.xml configuration for an internally signed certificate](#)
 - [Sample on-premise SSL JVM startup script configuration for an internally signed certificate](#)
- [On-Premise Controller with a Self-Signed Certificate](#)
 - [Sample on-premise SSL controller.xml configuration for a self-signed certificate](#)
 - [Sample on-premise SSL JVM startup script configuration for a self-signed certificate](#)
- [Learn More](#)

This topic covers how to configure the App Agent for Java (the agent) to connect to the Controller using SSL. It assumes that you use a SaaS Controller or have configured the on-premise Controller to use SSL.

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

Before You Begin

Before you configure the agent to enable SSL, gather the following information:

- Identify if the Controller is SaaS or on-premise.
- Identify the Controller SSL port.
 - For SaaS Controllers the SSL port is 443.
 - For on-premise Controllers the default SSL port is 8181, but you may configure the Controller to listen for SSL on another port.
- 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.
- Decide how to manage the configurations. See [Where to Configure App Agent Properties](#):
 - Add the configuration parameters to <agent install directory>/conf/controller-info.xml.
 - Or
 - Include system properties in the -javaagent argument in your JVM startup script.

SSL Utilities

We provide two utilities to help you implement SSL.

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 install directory>/utils/keystorereader/kr.jar
```

i 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 install directory>/appserver/glassfish/domains/domain1/appagent
```

To use the Keystore Certificate Extractor, execute kr.jar and pass the following parameters:

- The full path to the Controller's keystore:

```
<controller install  
directory>/appserver/glassfish/domains/domain1/config/keystore.jks
```

- The truststore output file name. By default the agent looks for cacerts.jks.
- 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 install  
directory>/appserver/glassfish/domains/domain1/config/keystore.jks cacerts.jks  
<controller certificate password>
```

Password Encryption Utility

The Password Encryption Utility encrypts the Controller's certificate password so you can add it to the controller-info.xml file. It installs to the following location:

```
<agent install directory>/utils/encryptor/encrypt.jar
```

To use the Password Encryption Utility, execute encrypt.jar and pass the clear text password as a parameter. The utility returns the encrypted password:

```
java -jar <agent install directory>/utils/encryptor/encrypt.jar <controller  
certificate password>  
  
Encrypted password is nkV/LwhLMLFjfNTbh0DLow==
```

SaaS Controller

1. Update the JVM startup script or controller-info.xml to use SSL enabled settings. See [App Agent for Java Configuration Properties](#).

- Set the Controller Port Property to 443. See [Controller Port Property](#).
- Set the Controller SSL Enabled Property to true. See [Controller SSL Enabled Property](#).

2. Save your changes.

3. Restart the JVM.

The agent detects SaaS implementations based upon the controller host URL, which must contain ".saas.appdynamics.com". It also checks for an account-name and an access-key. If all three elements exist, the agent connects with the SaaS Controller via SSL.

Sample SaaS SSL controller.xml configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<controller-info>

  <controller-host>mycompany.saas.appdynamics.com</controller-host>

  <controller-port>443</controller-port>

  <controller-ssl-enabled>true</controller-ssl-enabled>
  ...
  <account-name>mycompany</account-name>

  <account-access-key>xxxxxxxxxxxxxx</account-access-key>
  ...
</controller-info>
```

Sample SaaS SSL JVM startup script configuration

```
java -javaagent:/home/appdynamics/AppServerAgent/
-Dappdynamics.controller.hostName=<controller domain>
-Dappdynamics.controller.port=443 -Dappdynamics.controller.ssl.enabled=true ...
-Dappdynamics.agent.accountName=<account name>
-Dappdynamics.agent.accountAccessKey=<access key>
```

On-Premise Controller with a Trusted CA Signed Certificate

1. Update your JVM startup script or controller-info.xml to use SSL enabled settings. See [App Agent for Java Configuration Properties](#).

- Set the Controller Port Property to the on-premise SSL port. See [Controller Port Property](#).
- Set the Controller SSL Enabled Property to true. See [Controller SSL Enabled Property](#).
- To configure the agent to perform full validation of the Controller certificate, set the Force Default SSL Certificate Validation app agent node property to true. See [Force Default SSL Certificate Validation Property](#).

2. Save your changes.

3. Restart the JVM.

The agent connects to the Controller over SSL. Because the Force Default SSL Certificate

Validation app agent node property is set to true, the agent enforces the trust chain using the default Java truststore.

Sample on-premise SSL controller.xml configuration for a CA signed certificate

```
<?xml version="1.0" encoding="UTF-8"?>
<controller-info>

  <controller-host>mycontroller.mycompany.com</controller-host>

  <controller-port>8181</controller-port>

  <controller-ssl-enabled>true</controller-ssl-enabled>

  <force-default-certificate-validation>true</force-default-certificate-validation
>

  ...
</controller-info>
```

Sample on-premise SSL JVM startup script configuration for a CA signed certificate

```
java -javaagent:/home/appdynamics/AppServerAgent/
-Dappdynamics.controller.hostName=<controller domain>
-Dappdynamics.controller.port=443 -Dappdynamics.controller.ssl.enabled=true
-Dappdynamics.force.default.ssl.certificate.validation=true ...
```

On-Premise Controller with an Internally Signed Certificate

1. Obtain the root CA certificate from your internal resource. By default the agent looks for a Java truststore named cacerts.jks.

To import a certificate to a truststore, run the following command:

```
keytool -import -alias rootCA -file <certificate file name> -keystore
cacerts.jks -storepass <truststore password>
```

i This command creates the truststore cacerts.jks if it does not exist and assigns it the password you specify.

2. Copy the truststore file to the agent configuration directory:


```
cp cacerts.jks <agent install directory>/conf/cacerts.jks
```

3. Encrypt the truststore password. See [Password Encryption Utility](#).

4. Update your JVM startup script or controller-info.xml to use SSL enabled settings. See [App](#)

Agent for Java Configuration Properties.

- Set the Controller Port Property to the on-premise SSL port. See [Controller Port Property](#).
- Set the Controller SSL Enabled Property to true. See [Controller SSL Enabled Property](#).
- Set the Controller Keystore Password Property to the encrypted password. See [Controller Keystore Password Property](#).

 You must configure this property in the controller-info.xml. It is not available as a system property in the JVM startup script.

5. Restart the JVM.

The agent detects the cacerts.jks truststore in its configuration directory and uses it to enforce the trust chain when connecting to the Controller over SSL.

Sample on-premise SSL controller.xml configuration for an internally signed certificate

```
<?xml version="1.0" encoding="UTF-8"?>
<controller-info>

  <controller-host>mycontroller.mycompany.com</controller-host>

  <controller-port>8181</controller-port>

  <controller-ssl-enabled>true</controller-ssl-enabled>

  <controller-keystore-password>nkV/LwhLMLFjfNTbh0DLow==</controller-keystore-password>

  ...
</controller-info>
```

Sample on-premise SSL JVM startup script configuration for an internally signed certificate

```
java -javaagent:/home/appdynamics/AppServerAgent/
-Dappdynamics.controller.hostName=<controller domain>
-Dappdynamics.controller.port=443 -Dappdynamics.controller.ssl.enabled=true ...
```


On-Premise Controller with a Self-Signed Certificate

1. Extract the Controller's self-signed Certificate to a truststore named cacerts.jks. See [Keystore Certificate Extractor Utility](#).
2. Copy the truststore file to the agent configuration directory:

```
cp cacerts.jks <agent install directory>/conf/cacerts.jks
```

3. Encrypt the truststore password. See [Password Encryption Utility](#).
4. Update your JVM startup script or controller-info.xml to use SSL enabled settings. See [App Agent for Java Configuration Properties](#).

- Set the Controller Port Property to the on-premise SSL port. See [Controller Port Property](#).
- Set the Controller SSL Enabled Property to true. See [Controller SSL Enabled Property](#).
- Set the Controller Keystore Password Property to the encrypted password. See [Controller Keystore Password Property](#).

 You must configure this property in the controller-info.xml. It is not available as a system property in the JVM startup script.

5. Restart the JVM.

The agent detects the cacerts.jks truststore in its configuration directory and uses it to enforce the trust chain when connecting to the Controller over SSL.

Sample on-premise SSL controller.xml configuration for a self-signed certificate

```
<?xml version="1.0" encoding="UTF-8"?>
<controller-info>

  <controller-host>mycontroller.mycompany.com</controller-host>

  <controller-port>8181</controller-port>

  <controller-ssl-enabled>true</controller-ssl-enabled>

  <controller-keystore-password>nkV/LwhLMLFjfNTbh0DLow==</controller-keystore-pas
sword>

  ...
</controller-info>
```

Sample on-premise SSL JVM startup script configuration for a self-signed certificate

```
java \-javaagent:/home/appdynamics/AppServerAgent/
\ -Dappdynamics.controller.hostName=<controller domain>
\ -Dappdynamics.controller.port=443 \ -Dappdynamics.controller.ssl.enabled=true
...
```

Learn More

[Implement SSL](#)

[Install the App Agent for Java](#)

Upgrade the App Agent for Java

This topic provides instructions for upgrading the App Agent for Java.

Upgrade Sequence

If you are upgrading both the Controller and agents, first upgrade the Controller and then upgrade the Agents.

To upgrade the App Agent for Java

1. Shut down the application server where the App Agent for Java and the optional machine agent is installed.
2. Create a backup copy of the current agent installation directory and move the backup directory to a new location.
3. Download the latest release from [AppDynamics Download Center](#).
4. Extract the Agent binaries to a new directory and rename old directory. Then rename the new directory the same name as the original one. Copy the controller-info.xml from the old Agent directory to the new Agent directory. At the end, you should have the new files using the same directory path as the previous one.
5. If you previously made changes to the `<App_Server_Agent_Installation_Directory>/conf/app-agent-config.xml` file, copy those changes to the new file.

Using the same directory path avoids the task of manually changing the agent-related configurations in your JVM startup script.

6. Restart the application server.

Uninstall the App Agent for Java

- [To uninstall the Java Agent](#)
- [Learn More](#)

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 app agent runs on, the app agent will reappear in the UI the next time it connects to the Controller.

To prevent an app agent from connecting to the Controller, you need to remove the app 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 an App Agent for Java from the JVM configuration.

To uninstall the Java Agent

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

Learn More

- [Manage App Agents](#)
- [Install the App Agent for Java](#)
- [App Agent for Java Configuration Properties](#)

Configure AppDynamics for Java

See also, [App Agent for Java Configuration Properties](#)

Business Transaction Configuration Methodology for Java

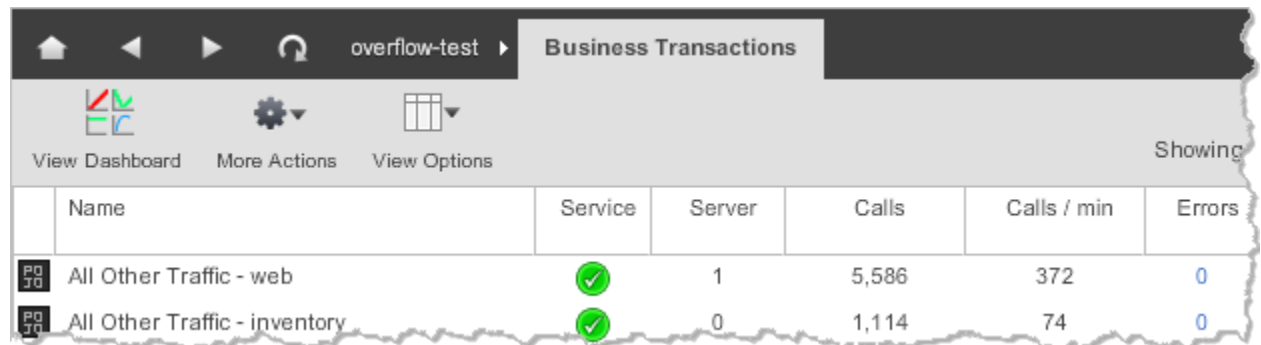
- [Modifying the Default Business Transaction Configuration](#)

- Suggested Methodology
- 1. Confirm Business Relevance
- 2. Review the Architecture
- 3. Modify Automatic Detection Criteria
 - Disable Entry Point Discovery
 - Customize Detection Settings
 - Combine Business Transactions
 - Use Dynamic Values to Split a Business Transaction
 - Collect Extra Traffic into a Catch-all Transaction
- 4. Exclude Business Transactions
 - Exclude Action in the UI
 - Configure Exclude Rules
 - Change the Default Exclude Rule Settings
 - Create New Exclude Rules
- 5. Rename Business Transactions
- 6. Group Business Transactions
- 7. Delete Old Unwanted Business Transactions

Modifying the Default Business Transaction Configuration

Sometimes you need to fine-tune your business transaction configuration, such as when:

- You do not see business transactions that you expect to see based on how your application should be represented in AppDynamics. See [Organizing Traffic as Business Transactions](#).
- You see the [All Other Traffic](#) business transaction in the [Business Transactions List](#), for example:

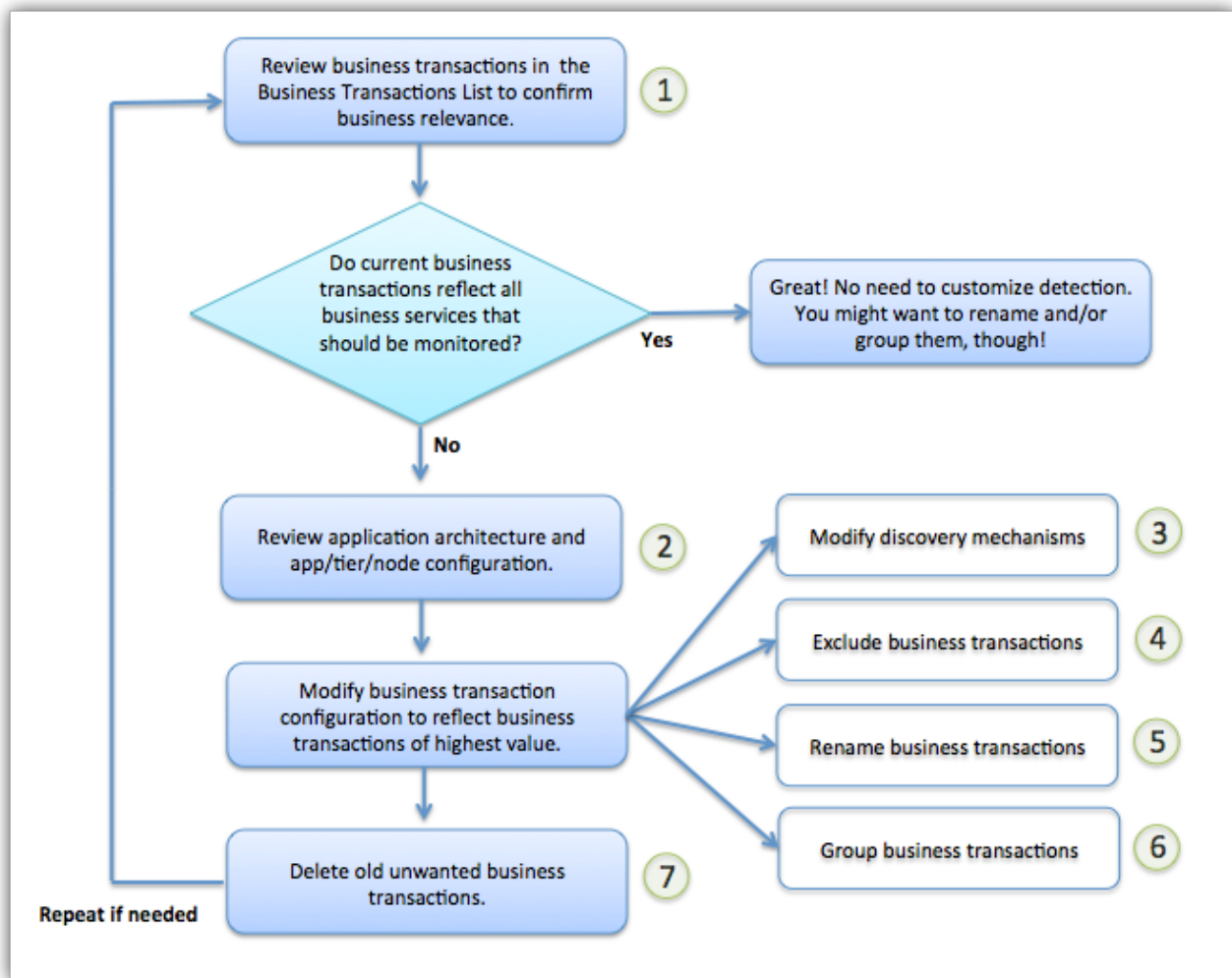


	Name	Service	Server	Calls	Calls / min	Errors
PD 76	All Other Traffic - web	✓	1	5,586	372	0
PD 76	All Other Traffic - inventory	✓	0	1,114	74	0

AppDynamics creates the [All Other Traffic](#) business transaction when a business transaction limit is reached. You can modify the transaction discovery configuration to reduce the number of business transactions.

To modify your business transaction configuration follow this suggested methodology.

Suggested Methodology



Confirm

1. Confirm Business Relevance

The first step in analyzing the business transaction configuration that is right for your application is to confirm which transactions you want to monitor. Talk with your application developers and architects about which are the most important processes to monitor. The discussion will help you identify the correct entry points, which define the beginnings of your business transactions. Be sure you are measuring the right things. See [Organizing Traffic as Business Transactions](#).

Once you know what you want to monitor, you can examine the business transactions being detected and determine your next steps.

Review

2. Review the Architecture

If you are missing an expected business transaction, review the application architecture and make sure the expected business transaction is not part of another

transaction initiated by another tier. Also make sure the application/tier/node configuration is correct. See [Mapping Application Services to the AppDynamics Model](#).

A blue button with rounded corners and a subtle gradient, containing the word "Modify" in a sans-serif font.

3. Modify Automatic Detection Criteria

While AppDynamics discovers many business transactions automatically, you may need to modify these mechanisms to detect additional ones or to disable ones that are not critical to monitor.

Configuration is hierarchical for a business application and its tiers, and has both a "global" scope and a more granular "custom" scope. See [Hierarchical Configuration Model](#) and [How AppDynamics Identifies Business Transactions Using Entry Points](#).

To change discovery mechanisms you can:







- [Disable Entry Point Discovery](#)
- [Customize Detection Settings](#)
- [Combine Business Transactions](#)
- [Use Dynamic Values to Split a Business Transaction](#)

Disable Entry Point Discovery

In the Transaction Detection window for the application or tier, you can completely disable transaction monitoring for a type of entry point. You may want to do this when:

- You know that all the business transactions of a specific entry point type don't need to be monitored.
- You want to use custom transaction discovery configurations instead.

Another example is when Servlets implement Spring Beans and you are interested in monitoring the transaction starting at the Spring Bean level. In this case you can disable Servlet discovery and then only the Spring Beans, which are enabled by default, are discovered and monitored. See [Exclude Spring Beans Of Specific Packages](#).

▼ Entry Points		
Type	Transaction Monitoring	Automatic Transaction Detection
 Servlet	<input type="checkbox"/> Enabled	<input type="checkbox"/> Discover Transactions automatically for all Servlet requests Configure Naming <input type="checkbox"/> Enable Servlet Filter Detection 
 Struts Action	<input checked="" type="checkbox"/> Enabled	<input checked="" type="checkbox"/> Discover Transactions automatically for all Struts Action invocations Transactions will be named: ActionName.MethodName
 Web Service	<input checked="" type="checkbox"/> Enabled	<input checked="" type="checkbox"/> Discover Transactions automatically for all Web Service requests Transactions will be named: ServiceName.OperationName
 POJO	<input checked="" type="checkbox"/> Enabled	Any Java method can be the entry point for a Business Transaction. The class to which the method belongs to can be picked using different parameters like its name, its super class name, the interfaces it implements, or the annotations it has.
 Spring Bean	<input checked="" type="checkbox"/> Enabled	<input type="checkbox"/> Discover Transactions automatically for all Spring Bean invocations Transactions will be named: BeanName.MethodName

Customize Detection Settings

You can use custom match rules on entry points to establish a set of transactions that give a good and distinct representation of the business activity while not being too granular. See [Configure Business Transaction Detection](#) and [Java Web Application Entry Points](#).

For example, by default AppDynamics uses the first two segments of a URI to identify Servlet-based business transactions. Depending on your application code, you may need to use either fewer or more segments of a URI to get the correct granularity to identify the transactions.

For another example, you may have a business transaction initiated by code that does not use a standard framework, and you need to define a custom [POJO entry point](#).

New Business Transaction Match Rule - POJO

Name

CustomPOJO

Enabled

☒

Background Task

☐

Transaction Match Cri...

Transaction Splitting

Exclude Rule

Define match criteria for a POJO method which be will an entry point for a Business Transaction

☒

Match Classes *

with a Class Name that

Contains

☒

Method Name *

Equals

Starts With

Ends With

Contains

Matches Reg Ex

Is in List

Combine Business Transactions

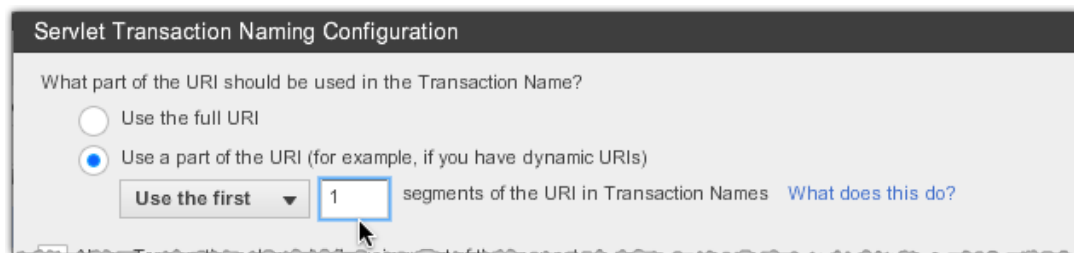
You can combine multiple business transactions by changing the auto-discovery rules at the global application or tier level and/or by creating custom match rules at more granular levels. Both techniques help you configure what business transactions to monitor.

For example, you may have many Servlet-based business transactions that are auto-discovered using the first two segments of the URI.

```
http://www.myapp.com/users/user01
http://www.myapp.com/users/user02
http://www.myapp.com/users/user03
etc...
```

If it makes more sense to use only the first segment, you can change the auto-discovery rule to specify only the first segment.

```
http://www.myapp.com/users
```



You can also combine multiple transactions using custom match rules on entry points. See [Configure Business Transaction Detection](#) and [Java Web Application Entry Points](#).

Use Dynamic Values to Split a Business Transaction

The process of using a dynamic value to customize business transaction discovery is called transaction splitting. Transaction splitting allows you to fine-tune transaction detection or exclusion based on a parameter or user data.

For example, you could create a new business transaction configuration that uses the "color" parameter to separate products/outdoor transactions.

See more Servlet examples:

- [Automatic Naming Configurations for Servlet-Based Business Transactions](#)
- [Custom Naming Configurations for Servlet-Based Business Transactions](#)
- [Advanced Servlet Transaction Detection Scenarios](#)

Collect Extra Traffic into a Catch-all Transaction

After configuring the most important entry points, consider creating a new "catch-all" rule to collect and name all other website traffic. This will help manage the number of business transactions you see in lists and flow maps.

Create a custom match "catch-all" rule where:

- Entry point type is "Servlet" (the most commonly encountered entry point type in Java environments)
- Priority is "0"
- Match value is "URI is not empty"

Make sure the Priority is "0" so that it will be discovered last. Any operations that don't match your customized detection rules will be discovered by this "catch-all" rule.

Note: The "catch-all" rule also captures all web service and Struts traffic. This is because custom rules are evaluated before exclude rules, and there are default exclude rules for web service and Struts traffic. The "catch-all" rule will evaluate first and put what would normally be excluded by default into the "catch-all" business transaction.

Exclude

4. Exclude Business Transactions

There are two ways to exclude auto-discovered business transactions that you do not need to monitor.

- The [Exclude Action in the UI](#): Excluding transactions from the UI is helpful if you think you may want to resume monitoring the transaction in the future, as the underlying configuration is still present. For example, a transaction may

not have traffic now but you anticipate that it will in the future.

- [Configure Exclude Rules](#): One reason to use exclude rules over the Exclude action in the UI is mainly for efficiency, as you can exclude whole classes of business transactions using string [match rule conditions](#). See [Creating Exclude Rules](#).

Exclude Action in the UI

When you exclude a business transaction from the UI, AppDynamics retains existing metrics for the business transaction, but no longer monitors it. Excluded transactions do not count against the maximum number of transactions allowed per application or per agent. Excluding transactions using this method is helpful if you think you may want to resume monitoring the transaction in the future.

The exclude action in the UI is most useful for web service, POJO, and EJB business transactions, where a single class or web service detects many methods/operations, and you want to monitor some but not all of them. In this case it's onerous to set up custom exclude rules, and much simpler to select the subset you do not want and exclude them via the UI.

In the Business Transaction List you can select one or more transactions and either right-click **Exclude Transactions** or select **Actions -> Exclude Transactions**.

To resume monitoring the business transaction, you can "un-exclude" it from the View Excluded Transactions window.

See [Excluding Business Transactions from the UI](#).

Configure Exclude Rules

Exclude rules prevent detection of business transactions that match certain criteria. You might want to use exclude rules in the following situations:

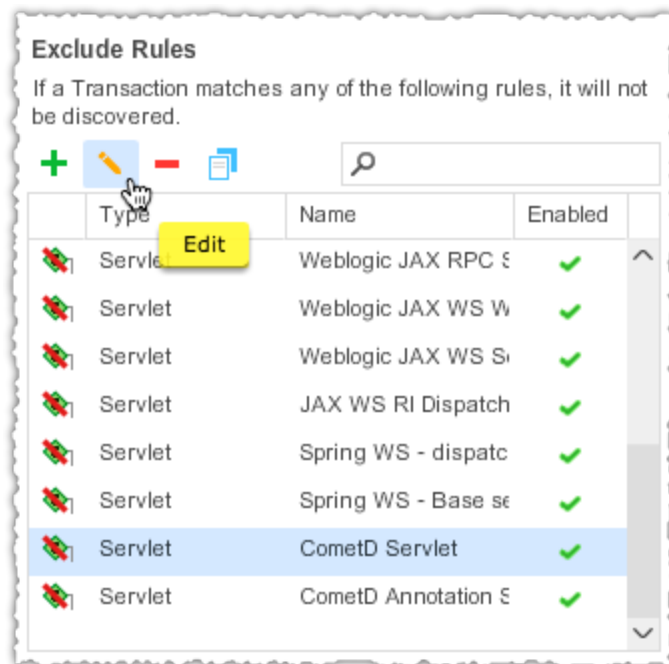
- AppDynamics is detecting business transactions that you are not interested in monitoring.
- A detected business transaction has no traffic and probably will not have interesting traffic in the future.
- You need to trim the total number of business transactions in order to stay under the agent and Controller limits.
- You need to substitute a default entry point with a more appropriate entry point using a custom match rule.

To customize exclude rules you can:

- [Change the Default Exclude Rule Settings](#)
and
- [Create New Exclude Rules](#)

Change the Default Exclude Rule Settings

Several entry points are excluded by default and you can edit them in the Transaction Detection panel.



Create New Exclude Rules

You can create new exclude rules using custom match conditions that provides fine granularity over business transaction detection.

New Exclude Business Transaction Match Rule - Servlet

Name:

Enabled: ☒

Transaction Match Criteria

☐ Method: GET

☒ URI: Equals

☐ HTTP Parameter (Both GET query parameters and POST parameters can be used): Check for parameter value

Parameter Name: Value: Equals

☐ Header: Check for parameter value

Parameter Name: Value: Equals

☐ Hostname: Equals

☐ Port: Equals

☐ Class Name: Equals

☐ Servlet Name: Equals

☐ Cookie: Check for cookie existence

Cookie Name:

Buttons: Cancel, Create Exclude Rule

For example an exclude rule can:

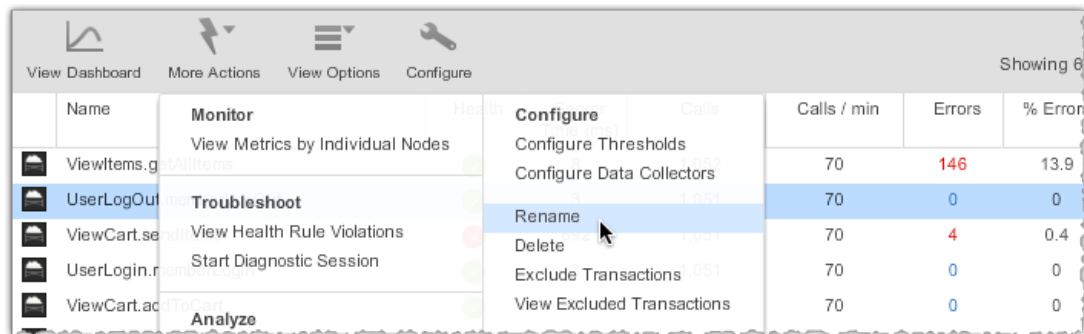
- Skip an EJB control class and use the business logic classes
- Behave like a filter that allows eligible requests and ignores the rest
- Exclude Spring Beans of specific packages.

See [Creating Exclude Rules](#) and [Exclude Rule Examples for Java](#).

Rename

5. Rename Business Transactions

By default, all business transactions are identified using default naming schemes for different types of requests. For ease of use you can change the label associated with the name. Use the **Action -> Rename** menu in the [Business Transaction Dashboard](#) or the [Business Transaction List](#) to give a user-friendly name to the transaction.

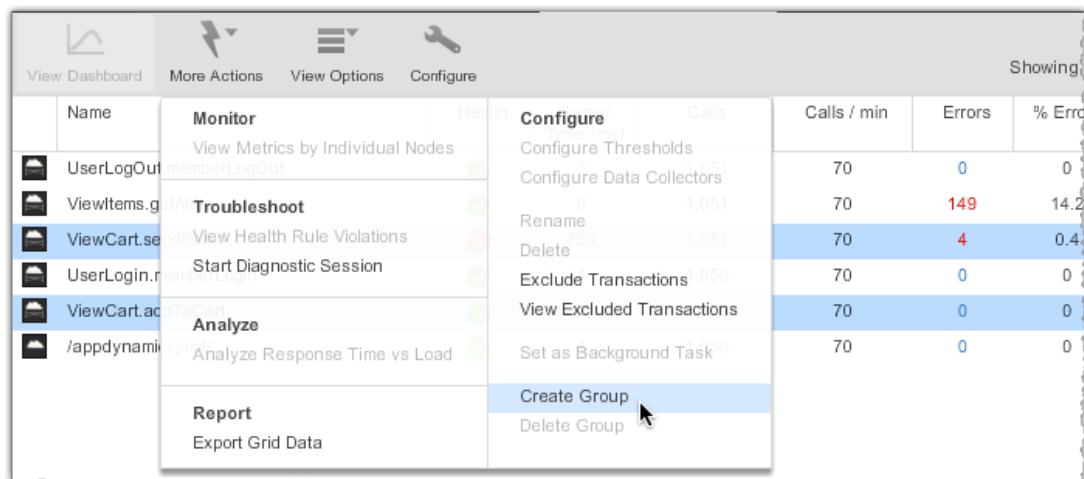


Name	Monitor	Health	Configure	Calls	Calls / min	Errors	% Error
ViewItems.getAl...	View Metrics by Individual Nodes	Green	Configure Thresholds	70	146	13.9	
UserLogout	Troubleshoot	Green	Configure Data Collectors	70	0	0	
ViewCart.send...	View Health Rule Violations	Red	Rename	70	4	0.4	
UserLogin.m...	Start Diagnostic Session	Green	Delete	70	0	0	
ViewCart.ad...	Analyze	Green	Exclude Transactions	70	0	0	
			View Excluded Transactions	70	0	0	

Group

6. Group Business Transactions

When multiple business transactions are similar and you want to roll up their metrics, you can put multiple business transactions into a group. You get metrics for each transaction and for the group as a whole. Grouping in the UI makes your lists and flowcharts easier to read by reducing visual clutter.



Name	Monitor	Health	Configure	Calls	Calls / min	Errors	% Error
UserLogout	View Metrics by Individual Nodes	Green	Configure Thresholds	70	0	0	
ViewItems.getAl...	Troubleshoot	Green	Configure Data Collectors	70	149	14.2	
ViewCart.send...	View Health Rule Violations	Red	Rename	70	4	0.4	
UserLogin.m...	Start Diagnostic Session	Green	Delete	70	0	0	
ViewCart.ad...	Analyze	Green	Exclude Transactions	70	0	0	
/appdynamic...	Analyze Response Time vs Load	Green	View Excluded Transactions	70	0	0	
			Set as Background Task	70	0	0	
			Create Group				
			Delete Group				

See [Organizing Business Transactions into Groups](#) and [Grouping Business](#)

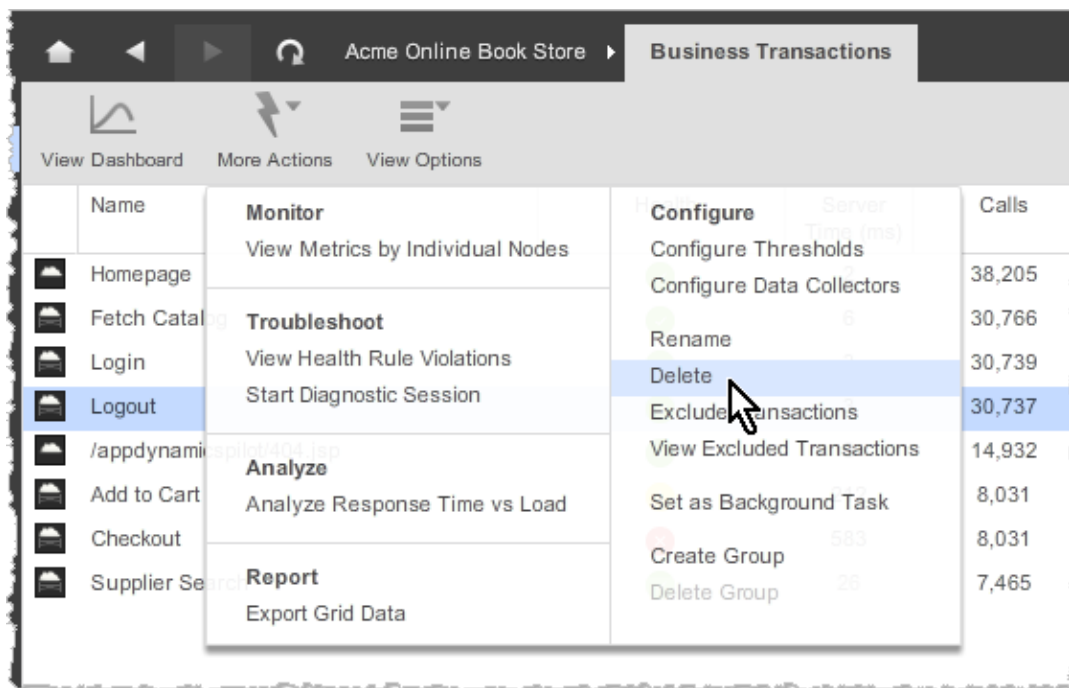
Transactions.

Delete

7. Delete Old Unwanted Business Transactions

After you have modified the business transaction discovery configurations, you then need to delete the old, unwanted business transactions. If you delete a transaction and you have not changed the configuration, it will be rediscovered. However, when you have revised the transaction discovery rules properly for what you want to see and then delete the unwanted business transaction, it won't be rediscovered.

You can delete transactions from the **Business Transaction List** using **More Actions**.



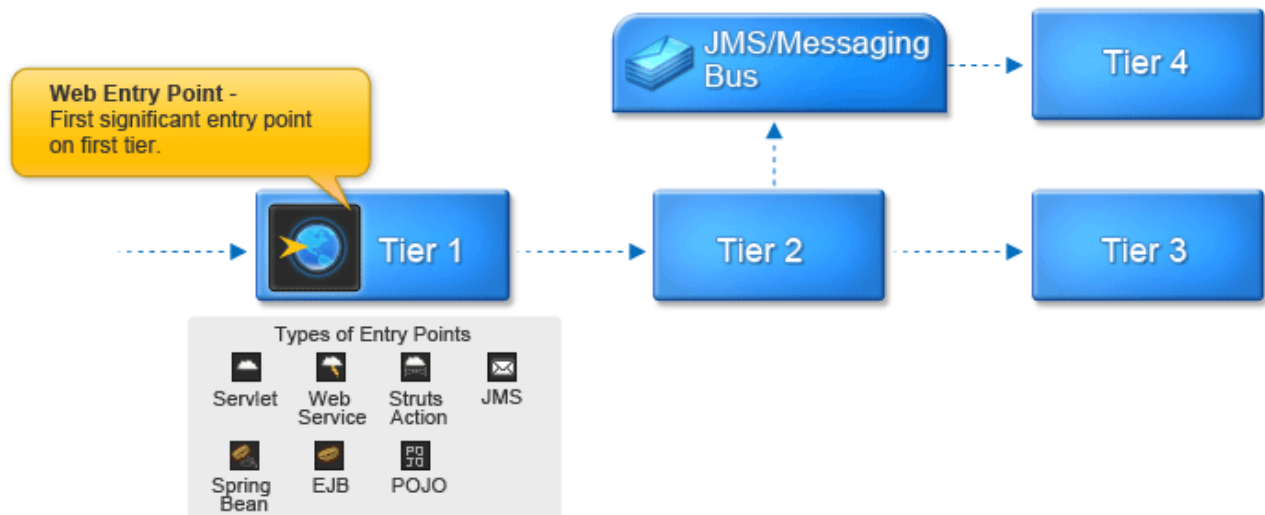
Java Web Application Entry Points

- [Tiers and Web Application Entry Points](#)
- [Other Web Application Frameworks Based on Servlets or Servlets Filter](#)
- [Learn More](#)

This section discusses web application entry points for different types of business transactions.

Tiers and Web Application Entry Points

A tier can have multiple entry points.



For example, for Java frameworks a combination of pure Servlets or JSPs, Struts, Web services, Servlet filters, etc. may all co-exist 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 about Java entry points see:

Other Web Application Frameworks Based on Servlets or Servlets Filter

AppDynamics provides out-of-the-box support for most of the common web frameworks that are based on Servlets or Servlet Filters. When using any of the frameworks listed below, refer to the [Servlet discovery rules](#) to configure transaction discovery.

- Spring MVC
- Wicket
- Java Server Faces (JSF)
- JRuby
- Grails
- Groovy
- Tapestry
- ColdFusion

Learn More

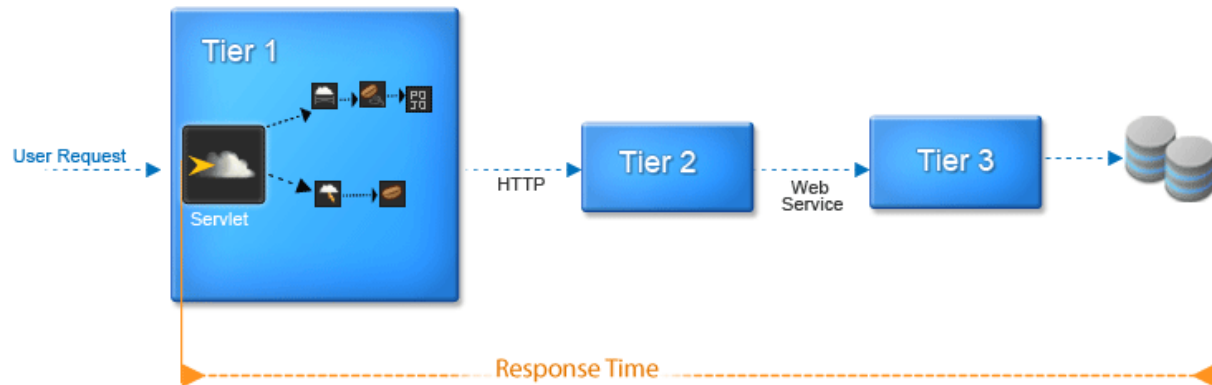
Servlet Entry Points

- [Servlet-Based Business Transactions](#)
- [Identify Business Transactions Based on REST-Style URLs](#)
- [Additional Servlet-Based Business Transaction Detection Scenarios](#)
- [Learn More](#)

You can configure transaction entry points for Servlet-based methods that may or may not be used as part of an application framework.

Servlet-Based Business Transactions

AppDynamics allows you to configure a transaction entry point on the invocation of the service method of a Servlet. The response time for the Servlet transaction is measured when the Servlet entry point is invoked.



As discussed in [Configure Business Transaction Detection](#) and [Business Transaction Configuration Methodology for Java](#), AppDynamics automatically identifies business transactions, and if the defaults are not ideal for monitoring your application, you can change them. For Servlet-based transactions you can:

- Identify a Servlet request based on a particular segment of its URI.
- Identify a Servlet request based on part of its HTTP request, such as header values, parameters, etc.

There are two scopes to identifying business transactions:

- **Automatic transaction naming configurations** (sometimes called "global discovery rules" and "auto-detection scheme") by default apply across the entire business application and can be overridden at the tier level. AppDynamics provides default, out-of-the-box automatic configurations. You can modify the automatic global transaction naming configuration across the entire business application or tier in the Transaction Detection Entry Points and Transaction Naming Configuration windows. See [Automatic Naming Configurations for Servlet-Based Business Transactions](#).
- **Custom transaction naming configurations** (sometimes called "custom match rules") by default apply across the entire business application and can be overridden at the tier level. AppDynamics provides some POJO custom match rules that are disabled by default. When you need different configurations for different parts of the application or different web contexts, add custom match rules in the Transaction Detection and New Business Transaction Match Rule window. See [Custom Naming Configurations for Servlet-Based Business Transactions](#).

As you might expect, you can use a combination of the global and specific configurations to effectively identify and represent the most important business transactions in your application. See [When to Use Custom Naming Configurations Instead of Automatic Global Configurations](#).

Identify Business Transactions Based on REST-Style URLs

REST applications typically have dynamic URLs where the application semantics are a part of the URL. Sometimes you need to "skip over" a dynamic part of a URL when identifying business transactions.

To configure business transactions based on dynamic URLs, see:

- For global configurations:
 - [Skip a Segment of the URL](#)
 - [Use Multiple Segments in a URL](#)
- For custom configurations:
 - [Custom Naming Configurations for Servlet-Based Business Transactions](#)

Additional Servlet-Based Business Transaction Detection Scenarios

For other detection scenarios see:

Learn More

For more information see:

- [Automatic Naming Configurations for Servlet-Based Business Transactions](#)
- [Custom Naming Configurations for Servlet-Based Business Transactions](#)
- [Use Parts of HTTP Requests to Name Global Servlet Entry Points](#)
- [Configure Business Transaction Detection](#)

Automatic Naming Configurations for Servlet-Based Business Transactions

- [Default Automatic Naming Identification for Servlet-Based Transactions](#)
- [Modify the Automatic Global Naming](#)
 - [To modify the automatic global naming for Servlet-based transactions](#)
- [Use Different Segments of the URI to Globally Identify Business Transactions](#)
 - [Use the Full, First or Last Segments of the URI](#)
 - [Skip a Segment of the URI](#)
 - [Use Multiple Segments in a URI](#)
- [Use Parts of HTTP Requests to Automatically Identify Business Transactions](#)
 - [To modify the automatic global naming for Servlet-based transactions](#)
 - [Use URI Segment Numbers](#)
 - [Use HTTP Parameter Values](#)
 - [Use Header Values](#)
 - [Use a Referer Header](#)
 - [Use Cookie Values](#)
 - [Use Session Attribute Values](#)
 - [Use Methods](#)
 - [Use Request Host](#)
 - [Use Request Originating Address](#)
 - [Use Custom Expressions](#)
 - [Use Part of an HTTP Parameter Value to Split a Business Transaction](#)
- [Learn More](#)

You can configure the automatic global business transaction naming criteria for Servlet-based applications. Automatic transaction naming configurations (sometimes called "global discovery rules" and "auto-detection scheme") by default apply across the entire business application and can be overridden at the tier level. You can modify the automatic global transaction naming

configuration across the entire business application or tier in the Transaction Detection Entry Points and Transaction Naming Configuration windows.

You can both modify the global automatic discovery rules and/or create custom match rules. See [When to Use Custom Naming Configurations Instead of Automatic Global Configurations](#).

Default Automatic Naming Identification for Servlet-Based Transactions

By default AppDynamics automatically discovers and identifies all Servlet-based business transactions using the first two segments of the URI.

For example, if the URI for a checkout operation in an online store is

`http://acmeonline.com/store/checkout`

AppDynamics automatically names the business transaction "store/checkout" and all checkout requests are handled in one business transaction.

By default AppDynamics does not use parts of the HTTP request to identify business transactions.

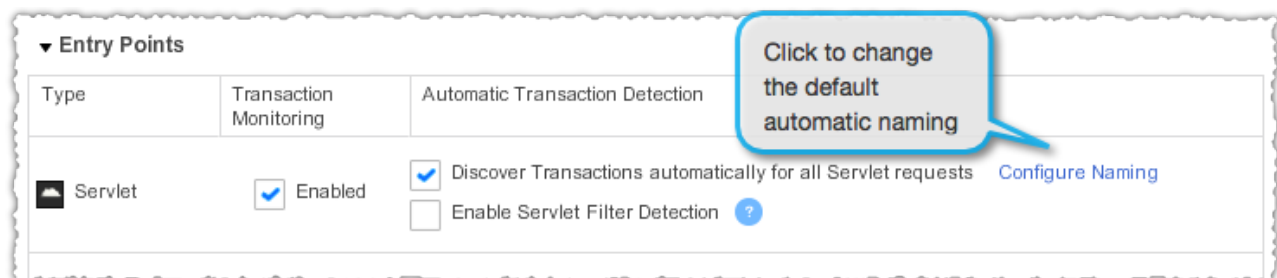
Modify the Automatic Global Naming

You can modify the URI and/or specify parts of the HTTP request.

- Change which segments of the URI are used to automatically identify business transactions. See the section [Use Different Segments of the URI to Identify Business Transactions](#).
- When the URI does not contain enough information to effectively identify the business transaction, you can use headers, parameters, cookies, and other parts of HTTP requests. See the section [Use Parts of HTTP Requests to Identify Transactions](#).

To modify the automatic global naming for Servlet-based transactions

1. In the left navigation pane click **Configure -> Instrumentation**.
2. In the Transaction Detection tab select the application or a tier.
3. Click the Java - Transaction Detection sub-tab.
4. Expand the Entry Points panel.
5. In the Servlet section, confirm that **Transaction Monitoring** and **Discover Transactions automatically for all Servlet requests** are both enabled.
6. In the Servlet section, click **Configure Naming**.



Use Different Segments of the URI to Globally Identify Business Transactions

You can change the automatic global naming configuration to fine-tune how AppDynamics identifies Servlet-based business transactions. You can:

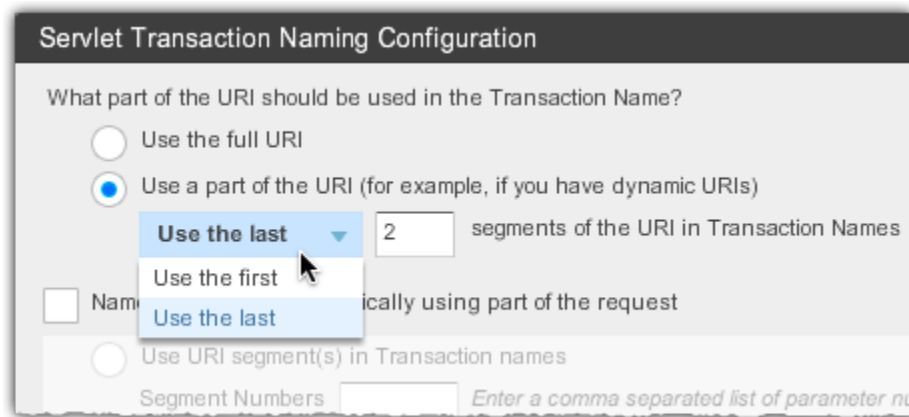
- [Use the Full, First or Last Segments](#)
- [Skip a Segment](#)
- [Use Multiple Segments](#)

Use the Full, First or Last Segments of the URI

For example the following URI represents the checkout operation in ACME Online:

`http://acmeonline.com/web/store/checkout`

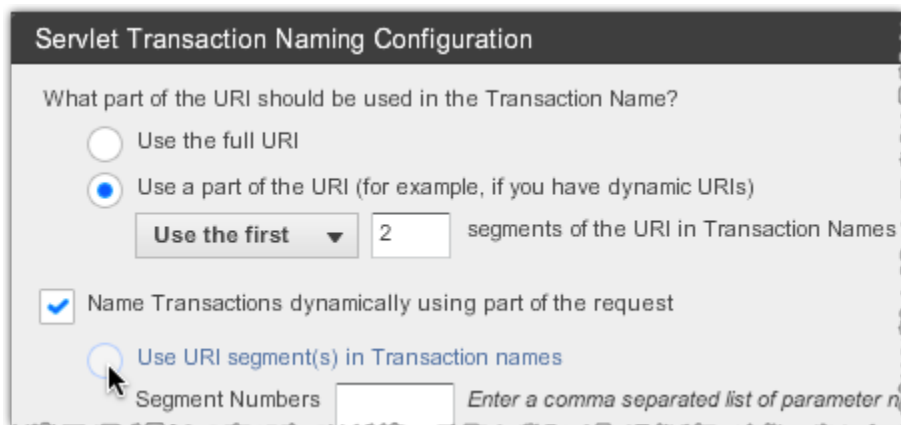
By default a business transaction is identified by the first two segments of the URI: "/web/store". However this does not indicate the business functionality of the operation, such as checkout, add to cart, etc. Another approach is to identify the business transaction using the last two segments of the URI: "store/checkout".



At the global scope you modify URI detection to:

- Use the full URI
- Use either the first or last segments of the URI, and specify how many segments

If you need more granularity on the URI, such as to use non-contiguous segments, you can click **Name Transactions dynamically using part of the request** and specify the segments with the **Use URI segments in Transaction names** option as described in the following sections.



Skip a Segment of the URI

Suppose a customer ID or the order ID is part of the URI, and you don't need that information to identify a business transaction. For example the following URL represents the checkout transaction invoked by a customer with ID 1234:

```
http://acmeonline.com/store/cust1234/checkout
```

The default discovery mechanism names this business transaction `"/store/cust1234"`. Ideally, all the customers performing a checkout operation should also be part of the checkout business transaction. Therefore, a better approach is to name this transaction `"/store/checkout"`.

To configure the transaction to be named `"/store/checkout"`, use the first segment of the URI and split that URI using the third segment, thus avoiding the second (dynamic) segment:

Servlet Transaction Naming Configuration

What part of the URI should be used in the Transaction Name?

☐ Use the full URI

☒ Use a part of the URI (for example, if you have dynamic URIs)

Use the first segments of the URI in Transaction Names [What does this do?](#)

☒ Name Transactions dynamically using part of the request

☒ Use URI segment(s) in Transaction names

Segment Numbers Enter a comma separated list of parameter numbers (e.g. 1,3,4)

This configuration modifies the default global discovery for all Servlet based transactions. The requests will be named as `"/store.checkout"` transaction.

Use Multiple Segments in a URI

Sometimes you need to use multiple or non-contiguous segments to globally identify a business transaction. For example for the following URL it is best to use segments 1,3, and 5:

```
http://acmeonline.com/user/foo@bar.com/profile/profile2345/edit
```

You would specify the first segment in the **What part of the URI..** section and specify the others in the **Name Transactions dynamically..** section:

Servlet Transaction Naming Configuration

What part of the URI should be used in the Transaction Name?

☐ Use the full URI

☒ Use a part of the URI (for example, if you have dynamic URIs)

Use the first segments of the URI in Transaction Names [What does this do?](#)

☒ Name Transactions dynamically using part of the request

☒ Use URI segment(s) in Transaction names

Segment Numbers Enter a comma separated list of parameter numbers (e.g. 1,3,4)

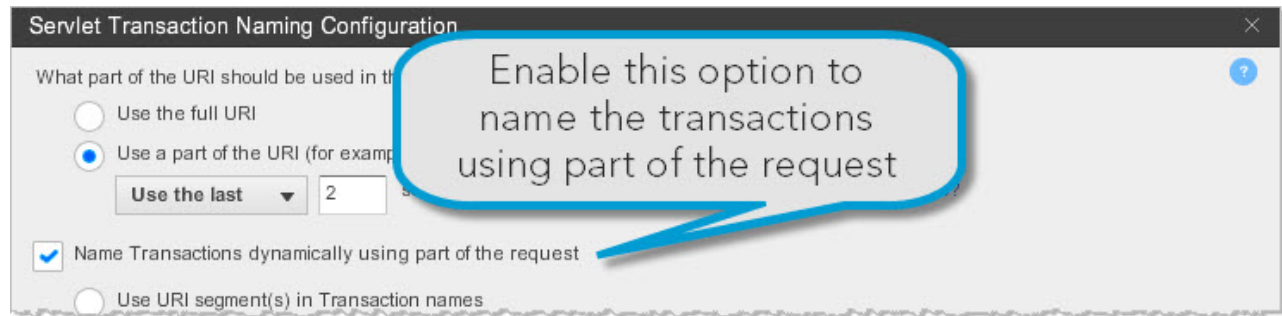
This configuration modifies the default global discovery for all Servlet based transactions. The requests will be named as `"/user.profile.edit"` transaction.

Use Parts of HTTP Requests to Automatically Identify Business Transactions

You can configure the global naming for your Servlet based business transactions using headers, parameters, cookies, and other parts of HTTP requests.

To identify Servlet based business transactions using particular parts of the HTTP request, use

the **Name Transactions dynamically using part of the request** option:



You can "split" the transactions using different parts of your request data like header, URI, sessions, etc. For more information see [Transaction Splitting for Dynamic Discovery](#).

To modify the automatic global naming for Servlet-based transactions

1. In the left navigation pane click **Configure -> Instrumentation**.
2. In the Transaction Detection tab select the application or a tier.
3. Click the Java - Transaction Detection sub tab.
4. Expand the Entry Points panel.
5. In the Servlet section, confirm that **Transaction Monitoring** and **Discover Transactions automatically for all Servlet requests** are both enabled.
6. In the Servlet section, click **Configure Naming**.
7. In the Servlet Transaction Naming Configuration window, click **Name Transactions dynamically using part of the request**.

Servlet Transaction Naming Configuration

What part of the URI should be used in the Transaction Name?

☐ Use the full URI

☒ Use a part of the URI (for example, if you have dynamic URIs)

Use the first segments of the URI in Transaction Names [What does this do?](#)

☒ Name Transactions dynamically using part of the request

☒ Use URI segment(s) in Transaction names

Segment Numbers Enter a comma separated list of parameter numbers (e.g. 1,3,4)

☐ Use a parameter value in Transaction names

Parameter Name

☐ Use a header value in Transaction names

Header Name

☐ Use a cookie value in Transaction names

Cookie Name

☐ Use a session attribute value in Transaction names

Session Attribute Key

☐ Use the request method (GET/POST/PUT) in Transaction names

☐ Use the request host Transaction in names

☐ Use the request originating address in Transaction names

☐ Apply a custom expression on HttpServletRequest and use the result in Transaction Names [Explain This](#)

Cancel Save

The following sections describe the various options for using parts of the HTTP requests.

Use URI Segment Numbers

You can name your transaction dynamically using URL segments. See [Use Different Segments of the URI to Automatically Identify Business Transactions](#).

Use HTTP Parameter Values

You can name a business transaction based on the value of a particular parameter in your request data.

For another example, consider the following URL:

```
http://acmeonline.com/orders/process?type=creditcard
```

The ideal naming configuration uses the combination of the parameter value for the parameter "type" and the last two segments to name the business transaction "/orders/process.creditcard". This configuration ensures that the credit card orders are differentiated from other orders.

Servlet Transaction Naming Configuration

What part of the URI should be used in the Transaction Name?

☐ Use the full URI

☒ Use a part of the URI (for example, if you have dynamic URIs)

Use the first ▼ 2 segments of the URI in Transaction Names

☒ Name Transactions dynamically using part of the request

☐ Use URI segment(s) in Transaction names

Segment Numbers Enter a comma separated list of parameter numbers

☒ Use a parameter value in Transaction names

Parameter Name

☐ Use a header value in Transaction names

For another example, in dispatcher-style Servlet patterns where the Servlet dispatches requests based on a query parameter, configure both the URI segments and the dynamic parameter. For the following URL:

`http://acmeonline.com/dispatcher?action=checkout`

You can identify the transaction based on the segments and the value of the "action" parameter. For example:

Servlet Transaction Naming Configuration

What part of the URI should be used in the Transaction Name?

☐ Use the full URI

☒ Use a part of the URI (for example, if you have dynamic URIs)

Use the first ▼ 2 segments of the URI in Transaction Names

☒ Name Transactions dynamically using part of the request

☐ Use URI segment(s) in Transaction names

Segment Numbers Enter a comma separated list of parameter numbers

☒ Use a parameter value in Transaction names

Parameter Name

☐ Use a header value in Transaction names

AppDynamics automatically identifies the business transaction based on the value of the "action" parameter, "<Segments_of_URI>.checkout".

You can also identify a business transaction based on multiple parameters by providing the comma-separated names of those parameters.

Use Header Values

You can also name your transaction based on the value of header(s) of your request data.

For example, to identify the requests based on "header1" for ACME Online:

1. Set the URI identification option.
2. Enable the **Use header value in transaction names** option and enter the header name "header1" .

The business transaction named "<Segments_of_URI>.<header1>".

You can name a business transaction based on multiple headers by providing the comma-separated names of those headers.

Use a Referer Header

If you are working on any of the following web technologies like Java Servlets, JSP, Struts, Springs or .NET, and if you would like to know which page has the current requested URL reference, you can get it by configuring the transactions on the REFERER header.

1. Set URI identification option.
2. Enable the **Use header value in transaction names** option and enter the header name "referer" .

This configures names of all transactions using the referring URL.

Use Cookie Values

You can also name your transaction based on the value of a particular cookie in your request data.

For example: For ACME Online, you can identify only those transactions when the "Gold" customers invoke the checkout operation.

1. Set the URI identification option.
2. Enable the **Use cookie value in transaction names** option and provide the cookie name.

You can name a business transaction based on multiple cookies by providing the comma-separated names of the cookies.

Use Session Attribute Values

You can name a business transaction based on the value of a particular session attribute key in your request data.

- Set the URI identification option.
- Enable the **Use session attribute values in transaction names** option and provide the session attribute key.

Use Methods

You can name a business transaction based on the GET/POST/PUT methods.

1. Set the URI identification option.
2. Enable the **Specify request method in transaction names** option.

Use Request Host

You can name a business transaction based on the request host.

1. Set the URI identification option.
2. Enable the **Use request host in transaction names** option.

Use Request Originating Address

You can name a business transaction based on the request's originating address.

1. Set the URI identification option.
2. Enable the **Use request originating address in transaction names** option.

Use Custom Expressions

You can use a custom expression to specify the name of business transaction.

1. Set the URI identification option.
2. Enter the expression. See [Custom Expressions for Naming Business Transactions](#).

Use Part of an HTTP Parameter Value to Split a Business Transaction

For example, given an HTTP parameter, `eventSource`, that has values like:

```
TabBar:AccountTab:AccountTab_AccountNumberSearchItem_Button_act
```

Assuming `getParameter` returns a `java.lang.string()`, you can use the following expression in the **Apply a custom expression on HttpServletRequest and use the result in Transaction Names** field:

```
{getParameter(eventSource).split(":").[0]}
```

AppDynamics uses everything up to the first colon ':' to name the business transaction, in this example "TabBar".

Additional Servlet-Based Business Transaction Detection Scenarios

For other detection scenarios see:

[Learn More](#)

For more information see:

- [Custom Naming Configurations for Servlet-Based Business Transactions](#)
- [Use Parts of HTTP Requests to Name Global Servlet Entry Points](#)
- [Configure Business Transaction Detection](#)
- [Group URI Patterns for All Servlet Entry Points](#)
- [Group URI Patterns for a Custom Servlet](#)

Custom Naming Configurations for Servlet-Based Business Transactions

- [Custom Match Rules for Specific Contexts](#)
 - [To access business transaction detection custom configuration](#)
- [Match on HTTP Methods](#)
- [Match on URIs](#)
 - [Split a URI Using Request Data](#)
 - [Split Based on the First Segments of the URI](#)
 - [Split Based on the Last Segments of the URI](#)
 - [Split Based on URI Segment Numbers](#)
 - [Split Based on Parameter Values](#)

- Split Based on Header Values
- Split Based on Cookie Values
- Split Based on an HTTP Method
- Split Based on the Request Host
- Split Based on the Request Originating Address
- Split Based on a Custom Expression
- Match on an HTTP GET or POST Parameter
 - To identify transactions based on the POST parameter
- Match on a Header Parameter Name or Value
- Match on a Hostname or Port
- Match on a Class or Servlet Name
- Match on a Cookie Name or Value
- Match on Information in the Body of the Servlet Request
- [Learn More](#)

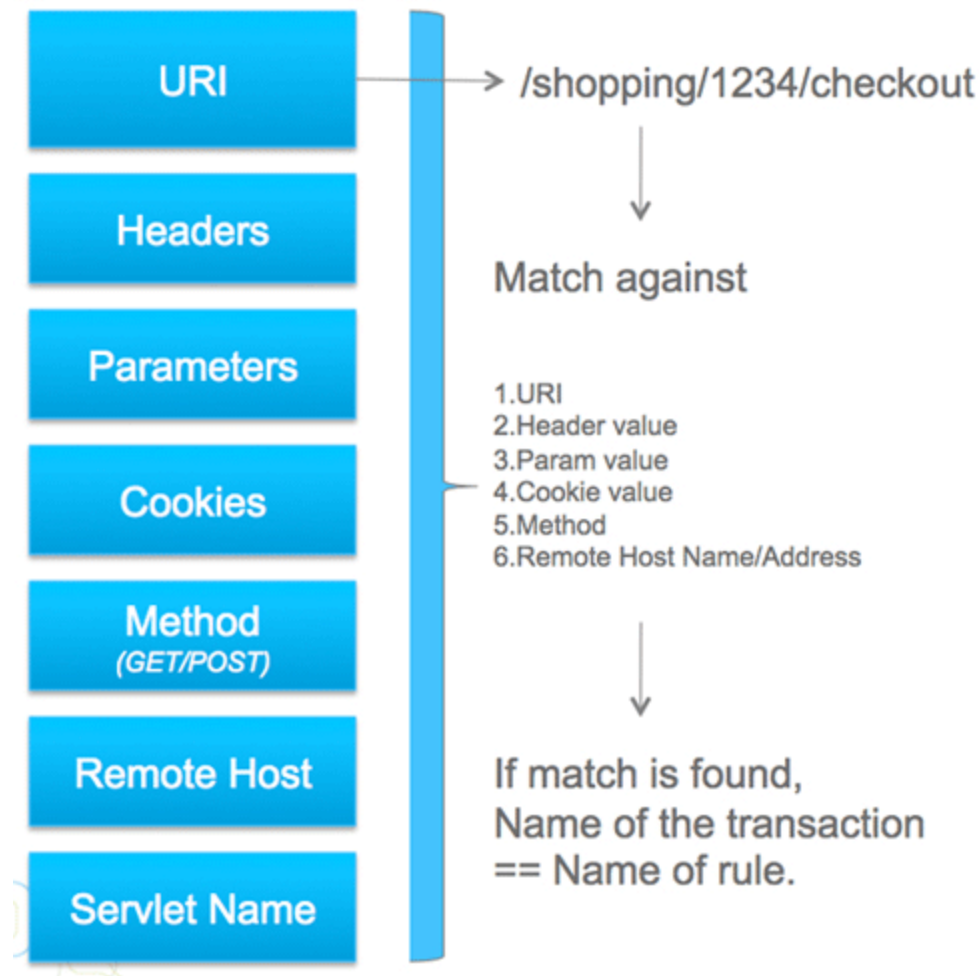
This topic describes how to configure entry point detection for a particular Servlet-based transaction using a custom match rule.

To configure at the global application or tier level see [Automatic Naming Configurations for Servlet-Based Business Transactions](#).

For context see [Configure Business Transaction Detection](#) and [Servlet Entry Points](#).

Custom Match Rules for Specific Contexts

To handle situations when web contexts represent different parts of the same business application and therefore require different naming configurations, use custom match rules for each of the web contexts or URIs. Custom match rules let you create mutually exclusive business transactions for specific contexts.



See [Custom Match Rules](#) for general information.

You can use the [transaction splitting option](#) to separate business transactions based on URI patterns, request data, or payload.

To access business transaction detection custom configuration

1. From the left navigation pane select **Configure -> Instrumentation**.
2. Click the **Transaction Detection** tab if it is not already selected.
3. Select the tier for which you want to configure the transactions or select the application if you want to configure at the application level. For information about inheritance for transaction detection, see [Hierarchical Configuration Model](#).
4. Click the Java - Transaction Detection subtab.
5. In the Custom Match Rules panel click **Add** (the + sign).
6. To configure a rule for Servlets, select **Servlet** the drop-down list and click **Next**.

The New Business Transaction Match Rule - Servlet window opens.

7. Give the rule a name.

8. If you have multiple custom match rules, change the priority based on the evaluation order of the rule. Higher numbers will evaluate first, with 0 being the last custom rule to be evaluated. For details see [Sequence and Precedence for Auto-Naming and Custom Match Rules](#).

Match on HTTP Methods

You can use GET, POST, PUT, and DELETE methods to identify a business transaction.

Match on URIs

By default AppDynamics automatically discovers and identifies all Servlet-based business transactions using the first two segments of the URI. You can [change the global default](#) or use a different pattern for a custom match rule.

A custom rule can match a URI based on one of the following expressions:

- Equals
- Starts With
- Ends With
- Contains
- Matches Reg Ex
- Is in List
- Is Not Empty

In addition you can configure a NOT condition:

Split a URI Using Request Data

You can split business transactions by adding attributes to either full or partial URI's for a Servlet request. You can also append a key-value pair to a URI discovered name. This splits the URI into a separate transaction.

1. First you configure the URI match. See the previous section.
2. Select the Split Transactions using Request Data tab and click the **Split Transactions using request data** option.
3. Provide the details for dynamically splitting based on the following:
 - [First Segments of the URI](#)
 - [Last Segments of the URI](#)
 - [Segment Numbers](#)
 - [Parameter Values](#)

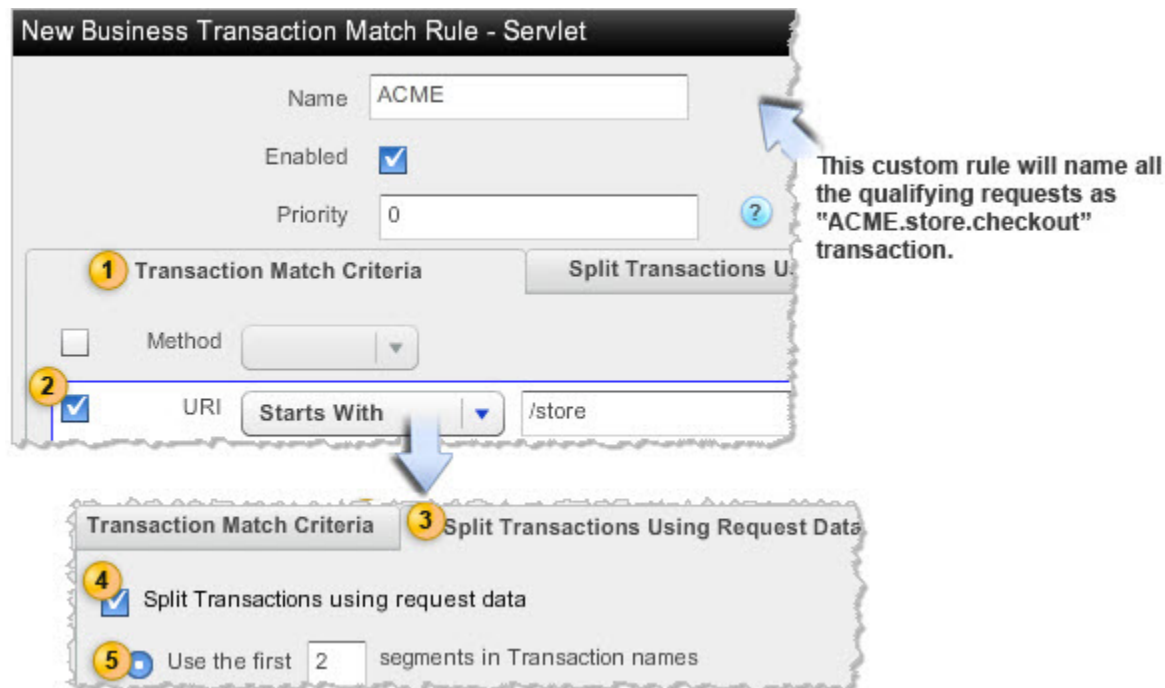
- Header Values
- Cookie Values
- HTTP Methods
- Request Host
- Request Originating Address
- Use a Custom Expression

Split Based on the First Segments of the URI

Consider the following request URI:

`http://acmeonline.com/store/checkout`

Note that the "first two segments" is the same as the default global configuration. You may want to use a similar custom match rule when you have a few different rules and you want to control the order (Priority) by which the rules execute.



Split Based on the Last Segments of the URI

Consider the following request URI:

`http://acmeonline.com/web/store/checkout`

Using the default global discovery rules, AppDynamics automatically identifies this business transaction as: `/web/store`. However the default does not accurately identify the functionality used by the store, such as checkout or add to cart etc.

You can reset it to **Use the last 2 segments** so that the business transaction is named "`<Name_Of_The_Custom_Rule>.store.checkout`".

Qualifying requests will be grouped into "ACMEOnline.store.checkout" transaction.

New Business Transaction Match Rule - Servlet

Name

Enabled ☒

Priority

1 Transaction Match Criteria

☐ Method

☒ **2** URI

3 Split Transactions Using Request Data

4 ☒ Split Transactions using request data

☐ Use the first segments in Transaction names

5 ☒ Use the last segments in Transaction names

Split Based on URI Segment Numbers

You can name your transaction dynamically using a particular segment number.

For example: For ACME Online, the checkout operation has following URL:

`http://www.acmeonline.com/shopping/customer1234/checkout`

To correctly identify the user requests for "checkout" functionality, provide the segment numbers ("1,3"). This custom match rule groups all the qualifying requests into a "<Name_Of_The_Custom_Rule>.shopping.checkout" transaction.

Qualifying requests will be grouped into "ACMEOnline.shopping.checkout" transaction.

New Business Transaction Match Rule - Servlet

Name: ACMEOnline

Enabled: ☒

Priority: 0

1 Transaction Match Criteria

2 Method: ☐ URI: ☒ Contains: /shopping

3 Split Transactions Using Request Data

4 ☒ Split Transactions using request data

5 Use URI segment(s) in Transaction names

Segment Numbers: 1,3

For another example:

New Business Transaction Match Rule - Servlet

Name: ACME

Enabled: ☒

Priority: 0

Transaction Match Criteria

☐ Method: GET

☒ URI: Equals: /user

Split Transactions Using Request Data

☒ Split Transactions using request data

☐ Use the first segments in Transaction names

☐ Use the last segments in Transaction names

☒ Use URI segment(s) in Transaction names

Segment Numbers: 1,3,5

Split Based on Parameter Values

You can name a business transaction based on the value of a particular parameter in the request data.

For example, ACME Online's checkout action results in following URL:

`http://acmeonline.com/orders/process?type=creditcard`

You want to use the combination of the parameter value for the parameter "type" and the last two segments to identify a business transaction called "/orders/process.creditcard". This ensures that the credit card orders are differentiated from other orders. On the "Transaction Match Criteria" section, select the matching option for URI (in this case, "orders").

1. Define the URI.
2. For the **Split Transactions using Request Data** option, enter the parameter name ("type").

This custom rule will group all those requests for which the value of parameter "type" is "creditcard" into a "<Name_Of_Custom_Rule>.creditcard" business transaction.

A custom rule may also identify other requests that use the same parameter. You can later choose to [exclude these transactions](#).

You can split based on multiple parameters by entering the comma-separated names of those parameters.

Qualifying requests will be grouped into "ACMEOnline.creditcard" transaction.

New Business Transaction Match Rule - Servlet

Name ACMEOnline

Enabled ☒

Priority 0

1 Transaction Match Criteria

2 Method ☐ URI Contains /orders/process

3 Split Transactions Using Request Data

4 Split Transactions using request data

5 Use a parameter value in Transaction names

Parameter Name type

You can also provide comma-separated list of parameter names.

Split Based on Header Values

You can also name your transaction based on the value of header(s) of your request data, including the host name.

For example, to identify the requests based on the host name for ACME Online:

1. Define the URI.
2. For the **Split Transactions using Request Data** option, enter the header name ("Host").

This custom rule will put qualifying requests into a business transaction named

"<Name_Of_Custom_Rule>.<Host_Name>".

Qualifying requests will be grouped into "ACMEOnline.<Host_Name>" transaction.

New Business Transaction Match Rule - Servlet

Name ACMEOnline

Enabled ☒

Priority 0

1 Transaction Match Criteria

2 Method

3 Split Transactions Using Request Data

4 Split Transactions using request data

5 Use a header value in Transaction names

Header Name Host

You can also provide comma-separated list of header names.

You can also name your transaction based on multiple headers by entering the comma-separated names of those headers.

Split Based on Cookie Values

You can also name your transaction based on the value of a particular cookie in your request data.

For example, you can identify only those business transactions where the "Gold" customers invoke the checkout operation.

To do this, specify a custom rule as follows:

1. Define the URI.
2. For the **Split Transactions using Request Data** option, enter the name of the cookie.

This configuration will also return those transactions for which the customer priority is not "Gold". You can choose to [exclude such transactions](#).

You can also name your transaction based on multiple cookies by providing the comma-separated names of those cookies.

Split Based on an HTTP Method

You can name a business transaction based on GET/POST/PUT methods.

To do this, specify a custom rule as follows:

1. Define the URI.
2. For the **Split Transactions using Request Data** option, select **Use request methods (GET/PUT/POST)** in the transaction names.

Qualifying requests will be grouped into "ACMEOnline.<Method_Name>" transaction.

New Business Transaction Match Rule - Servlet

Name: ACMEOnline

Enabled: ☒

Priority: 0

Transaction Match Criteria

☐ Method

☒ URI: Contains /orders/process

Split Transactions Using Request Data

☒ Split Transactions using request data

☒ Use the request method (GET/POST/PUT) in Transaction names

Split Based on the Request Host

You can name a business transaction based on the request host.

1. Define the URI.
2. For the **Split Transactions using Request Data** option, select **Use request host in transaction names**.

Split Based on the Request Originating Address

You can name a business transaction based on the request's originating address.

1. Define the URI.
2. For the **Split Transactions using Request Data** option, select **Use the request originating address in transaction names**.

Split Based on a Custom Expression

You can use a custom expression to name a business transaction.

1. Define the URI.
2. For the **Split Transactions using Request Data** option, select **Apply a custom expression on HttpServletRequest and use the result in Transaction Names**.
3. Enter the custom expression. See [Custom Expressions for Naming Business Transactions](#).

Match on an HTTP GET or POST Parameter

To identify transactions based on the POST parameter

For example, ACME Online's checkout operation for any item in the "Book" category results in a POST parameter ("itemid") whose value is "Book".

To identify only those requests that belong to a "BuyBook" business transaction, configure the custom match rule:

1. In the Transaction Match Criteria tab, define the URI as "/shopping".
2. In the same tab, check the **HTTP Parameter** option.
3. Select **Check for parameter value**.
4. Enter the **Parameter Name**; in this example it is "itemid".
5. Set **Value Equals** to "Book".

New Business Transaction Match Rule - Servlet

Name: Buybook

Enabled: ☒

Priority: 0

Transaction Match Criteria | Split Transactions Using Request Data | Split Transactions Using Request Data

☐ Method: GET

☒ URI: Equals

☒ HTTP Parameter (Both GET query parameters and POST parameters can be used): Check for parameter value

Parameter Name: itemid

Value: Equals Book

With this custom match rule, every time a request matches the custom parameter rule it is identified as part of the BuyBook business transaction.

Match on a Header Parameter Name or Value

A custom rule can match on a header parameter name.

1. In the Transaction Match Criteria tab, check **Header**.
2. Select **Check for parameter existence** and enter the parameter name.

A custom rule can match on a header parameter name and value.

1. In the Transaction Match Criteria tab, check **Header**.
2. Select **Check for parameter value** and enter the parameter name.

3. Enter a **Value** based on one of the following expressions:

- Equals
- Starts With
- Ends With
- Contains
- Matches Reg Ex
- Is in List
- Is Not Empty

Match on a Hostname or Port

A custom rule can match a hostname or port based on one of the following expressions:

- Equals
- Starts With
- Ends With
- Contains
- Matches Reg Ex
- Is in List
- Is Not Empty

In addition you can configure a NOT condition.

Match on a Class or Servlet Name

A custom rule can match a class name or Servlet name on one of the following expressions:

- Equals
- Starts With
- Ends With
- Contains
- Matches Reg Ex
- Is in List
- Is Not Empty

In addition you can configure a NOT condition.

Match on a Cookie Name or Value

A custom rule can match on a cookie name.

1. In the Transaction Match Criteria tab, check **Cookie**.
2. Select **Check for cookie existence** and enter the cookie name.

A custom rule can match on the value of a cookie.

1. In the Transaction Match Criteria tab, check **Cookie**.
2. Select **Check for cookie value** and enter the cookie name.
3. Enter a **Value** based on one of the following expressions:
 - Equals
 - Starts With
 - Ends With
 - Contains

- Matches Reg Ex
- Is in List
- Is Not Empty

Match on Information in the Body of the Servlet Request

In certain situations, you might have an application that receives an XML/JSON payload as part of the POST request.

If the category of processing is a part of the XML/JSON, neither the URI nor the parameters/headers have enough information to name the transaction. Only the XML contents that can provide correct naming configuration.

See [Identify Transactions Based on DOM Parsing Incoming XML Payload and Identify Transactions for Java XML Binding Frameworks](#).

Learn More

- [Configure Business Transaction Detection](#)
- [Business Transaction Configuration Methodology for Java](#)
- [Servlet Entry Points](#)
- [Automatic Naming Configurations for Servlet-Based Business Transactions](#)

Custom Expressions for Naming Business Transactions

You can create custom expressions to name transactions based on the evaluation of a custom expression on the `HttpServletRequestObject`.

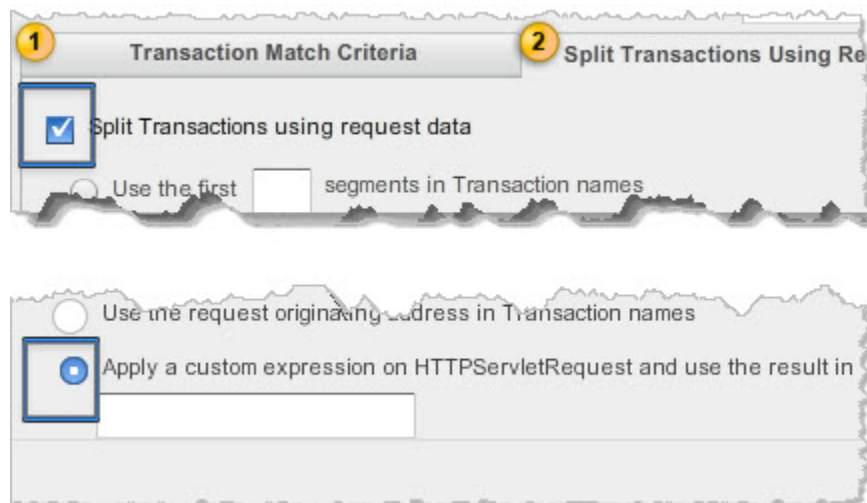
Suppose you want to monitor the `HttpServletRequest` request variable to identify the names and values of all the request parameters passed with the request.

The following example shows a custom rule configuration based on the expression:

```
${getParameter(myParam)}-${getUserPrincipal().getName() }
```

which evaluates to:

```
request.getParameter("myParam")+"-"+request.getUserPrincipal()
```



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).

Request Attributes in the Custom Expression

A custom expression can have a combination of any of the following getter chains on the request attributes:

Getters on Request Attributes	Transaction Identification
<code>getAuthType()</code>	Use this option to monitor secure (or insecure) communications.
<code>getContextPath()</code>	Identify the user requests based on the portion of the URI.
<code>getHeader()</code>	Identify the requests based on request headers.
<code>getMethod()</code>	Identify user requests invoked by a particular method.
<code>getPathInfo()</code>	Identify user requests based on the extra path information associated with the URL (sent by the client when the request was made).
<code>getQueryString()</code>	Identify the requests based on the query string contained in the request URL after the path.
<code>getRemoteUser()</code>	Identify the user requests based on the login of the user making this request.
<code>getRequestedSessionId()</code>	Identify user requests based on the session id specified by the client.
<code>getUserPrincipal()</code>	Identify user requests based on the current authenticated user.

For example, the following custom expression can be used to name the business transactions using the combination of header and a particular parameter:

```
${getHeader(header1)}-${getParameter(myParam)}
```

The identified transaction will be named based on the result of following expression in your application code:

```
request.getHeader("header1")\+ "-"\+ request.getParameter("myParam")
```

Advanced Servlet Transaction Detection Scenarios

See also:

[Custom Naming Configurations for Servlet-Based Business Transactions](#)

Servlet Entry Points

Identify Transactions Based on DOM Parsing Incoming XML Payload

- Business Transactions and XML Payload
 - Identifying the Business Transaction Using the XPath Expression
 - To configure a custom match rule
- [Learn More](#)

This topic describes how to identify transactions when an XML is posted to a Servlet.

Business Transactions and XML Payload

The XML contains the naming information for the Business Transaction. The Servlet uses a DOM parser to parse the posted XML into a DOM object.



Posted XML is parsed into a DOM object

Identifying the Business Transaction Using the XPath Expression

For example, the following XML posts an order for three items. The order uses credit card processing.

```
<acme>
  <order>
    <type>creditcard</type>
    <item>Item1</item>
    <item>Item2</item>
    <item>Item3</item>
  </order>
</acme>
```

The URL is:

```
http://acmeonline.com/store
```

The doPost method of the Servlet is:

```
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)
    .....
}
```

The XPath expression "//order/type" on this XML payload evaluates to "creditcard".

This value correctly identifies the type of the order and therefore should be used to name the "Order" transaction.

To identify the Business Transactions in this manner, first configure a custom match rule that automatically intercepts the method that parses the XML and gets the DOM object.

You use the XPath expression in the custom rule so that it names the transaction, for example "Store.order.creditcard". Though the name is not obtained until the XML is parsed, AppDynamics measures the duration of the business transaction to include the execution of the doPost() method.

To configure a custom match rule

1. Navigate to the custom rule section for Servlets.
2. In the **Transaction Match Criteria** tab, specify the URI.
3. In the **Split Transactions Using Payloads** tab, enable **Split transactions using XML/JSON Payload or a Java method invocation**.
4. Set the split mechanism to **XPath Expressions**.
5. Enter the XPath expression that you want to set as the Entry Point. The result of the XPath expression will be appended to the name of the Business Transaction.

New Business Transaction Match Rule - Servlet

Name:

Enabled: ☒

Priority:

1 Transaction Match Criteria

☐ Method:

2 ☒ URI:

3 Split Transactions Using Request Data

4 ☒ Split Transactions using XML/JSON Payload or a Java method invocation

Split Mechanism: **5** ☒ XPath Expressions ☐ Java XML Binding ☐ JSON ☐ POJO Method Call

[Help with XPATH Splitting](#)

XPath Expression:

☐ Ignore if expression does not evaluate

This custom rule ensures that all qualifying requests are named as "Store.Order.creditcard" transaction.

You can use one or more XPath expressions to chain the names generated for the Business Transaction.

If the expression does not evaluate to a value, the transaction will not be identified.

Learn More

- [Servlet Entry Points](#)

Identify Transactions Based on POJO Method Invoked by a Servlet

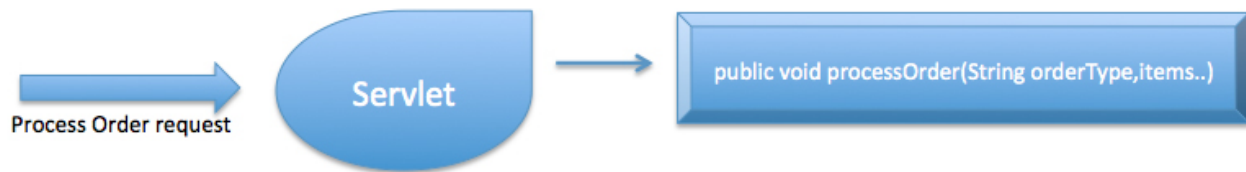
- [Using a Java Method to Name a Transaction](#)
 - [To configure the custom match rule](#)
- [Learn More](#)

Using a Java Method to Name a Transaction

You can use a Java method to name the transaction where:

- You might not have a clear URI pattern or
- You are using XML/JSON frameworks that are currently not supported by AppDynamics.

The following illustration shows how the Servlet that invokes the POJO method holds the transaction name.



For example, consider the `processOrder()` method. The order is parsed using any type of method and eventually when the `processOrder()` method is invoked, a correct approach to name transaction is to capture the first parameter to the `processOrder()` method.

The following URL is used by these requests: `http://acmeonline.com/store`. Following code snippet shows the `doPost()` method of the Servlet:

```
public void doPost(HttpServletRequest req, HttpServletResponse resp)
{
    //process the data from the sevlet request and get the orderType and the items
    processOrder(orderType,item)
    .....
}
public void processOrder(String orderType,String item)
{
    //process order
}
```

The `processOrder()` method has the information which can correctly derive the type of the order and also the name of the transaction. To identify these requests as a single transaction, configure a custom match rule.

To configure the custom match rule

1. Go to the custom rule section for Servlet Entry Points
2. In the **Transaction Match Criteria** tab, specify the URI.
3. In the **Split Transactions Using Payload** tab, enable Split transactions using XML/JSON Payload or a Java method invocation.
4. Select POJO Method Call as the split mechanism .
5. Enter the class and the method name.
6. If the method is overloaded, also specify the details for the arguments.
7. Optionally, you can name your transactions by defining multiple methods in a getter chain in the Method Call Chain field.

The following screenshot displays the configuration of a custom match rule which will name all the qualifying requests into a "Store.order.creditcard" transaction:

New Business Transaction Match Rule - Servlet

Name:

Enabled: ☒

Priority:

Transaction Match Criteria

☐ Method:

☒ URI:

Split Transactions Using Request Data

Split Transactions Using Payload

☒ Split Transactions using XML/JSON Payload or a Java method invocation

Split Mechanism: ☐ XPath Expressions ☐ Java XML Binding ☐ JSON ☒ POJO Method Call

[Help with POJO Splitting](#)

Class Name:

Method Name:

Return Type: ☐

Number of Arguments:

Argument Index:

Method Call Chain:

For example: `getPerson().getAddress().getStreet()`

This custom rule ensures that the processOrder method is automatically intercepted.

Although the name is not obtained till the processOrder() method is called, the time for the transaction will include all of the doGet() method.

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.

Learn More

- [Servlet Entry Points](#)

Identify Transactions for Java XML Binding Frameworks

- [To Configure the Custom Match Rule](#)

Supported Java XML data binding frameworks

[Learn More](#)

The following illustration shows the situation in which an XML is posted to a Servlet. The Servlet

uses an XML-to-Java binding framework, such as XMLBeans or Castor, to unmarshal and read the posted XML payload.



The XML payload contains the naming information for the transactions.

In the following example, an XML payload posts an order for three items. It uses a credit card to process the order.

The URL is: <http://acmeonline.com/store>

```
<acme>
  <order>
    <type>creditcard</type>
    <item>Item1</item>
    <item>Item2</item>
    <item>Item3</item>
  </order>
</acme>
```

The following code snippet shows the doPost() method of the Servlet:

```
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)

    ...
}
```

After the posted XML is unmarshalled to the PurchaseOrder data object, the getOrderType() method should be used to identify the type of the order.

To Configure the Custom Match Rule

1. Navigate to the custom rule section for [Servlet Entry Points](#).
2. In the **Transaction Match Criteria** tab, specify the URI.
3. In the **Split Transactions Using Payload** tab, check Split transactions using XML/JSON Payload or a Java method invocation.
4. Select Java XML Binding as the split mechanism.
5. Enter the class name and the method name.

The screenshot below shows a custom match rule which identifies the business transaction for this example as "Store.order.creditcard":

New Business Transaction Match Rule - Servlet

Name:

Enabled: ☒

Priority:

Transaction Match Criteria

☐ Method:

☒ URI:

Split Transactions Using Payload

☒ Split Transactions using XML/JSON Payload or a Java method invocation

Split Mechanism: ☐ XPath Expressions ☒ Java XML Binding ☐ JSON ☐ POJO Method Call

[Help with Java XML Binding Splitting](#) ?

Unmarshaled Class Name:

Method Call Chain:

For example: `getPerson().getAddress().getStreet()`

This custom rule ensures that the method in XMLBeans (which unmarshals XML to Java objects) is automatically intercepted. It also ensures that the `getOrderType()` method is applied on the Java data object only if it is the `PurchaseOrder` data object.

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.

Supported Java XML data binding frameworks

The following Java XML data binding frameworks are supported:

- Castor

- JAXB
- JibX
- XMLBeans
- XStream

Learn More

- [Servlet Entry Points](#)

Identify Transactions Based on JSON Payload

- [Example Configuration for a JSON Payload](#)
- [Learn More](#)

Example Configuration for a JSON Payload

The following illustration shows a JSON payload posted to a Servlet and the Servlet unmarshalls the payload.



The JSON contains the naming information for the transactions.

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:

```
public void doPost(HttpServletRequest req, HttpServletResponse resp)
{
    //create JSONObject from servlet input stream
    String orderType = jsonObject.get("type");\\
    //read the item for the order\\
    processOrder(orderType,item)\\
    .....\\
}
```

After the posted JSON payload is unmarshalled to the JSON object, the "type" key is required to identify the type of the order. In this case, this key uniquely identifies the business transaction.

To configure this rule:

1. Go to the custom rule section for [Servlet Entry Points](#).
2. Under "Transaction Match Criteria" specify the URI.
3. Under "Split Transactions Using Payloads", enable "Split transactions using XML/JSON Payload or a Java method invocation".
4. Select the split mechanism as "JSON".
5. Enter the JSON object key.

The following screenshot displays the configuration for a custom match rule that will name all the qualifying requests into a single transaction called "Store.Order.creditcard".

New Business Transaction Match Rule - Servlet

This custom rule ensures that all qualifying requests are named as "Store.Order.creditcard" transaction.

Name:

Enabled: ☒

Priority:

1 Transaction Match Criteria **Split Transactions Using Request Data** **3 Split Transactions Using Payload**

☐ Method

2 ☒ URI

4 ☒ Split Transactions using XML/JSON Payload or a Java method invocation

Split Mechanism: ☐ XPath Expressions ☐ Java XML Binding **5** ☒ JSON ☐ POJO Method Call

NOTE: The 'enable-json-bci-rules' agent property must be set to true for each node in this Tier for this match rule.

[Help with JSON Splitting](#) ?

JSON Object Key:

6. Set the property "enable-json-bci-rules" to "true" for each node to enable this custom rule. See [App Agent Node Properties](#).

This configuration ensures that the JSONObject and the get("\$JSON_Object_Key") method on this object are intercepted automatically to get the name of the transaction. Although the transaction name is not obtained until the JSON object is unmarshalled, the response time for the transaction will be calculated from the doGet method.

Learn More

- [Servlet Entry Points](#)
- [App Agent Node Properties](#)

Identify Transactions Based on URL Segment and HTTP Parameter

This topic describes a method you can use to identify transactions using both a URL Segment and HTTP Parameter split to get split transactions based on page context or action.

Use Custom Expressions

Use the "Apply a custom expression on HttpServletRequest and use the result in Transaction Names" option for Servlet naming with the following expression. Using this method, the transaction is named immediately.

```
${getParameter(myparam)}-${getRequestURI().split('/')[1,2]}
```

The above expression generates paramvalue-secondsegment-thirdsegment.

Servlet Transaction Naming Configuration

What part of the URI should be used in the Transaction Name?

☐ Use the full URI

☒ Use a part of the URI (for example, if you have dynamic URIs)

Use the first segments of the URI in Transaction Names [What does this do?](#)

☒ Name Transactions dynamically using part of the request

☐ Use URI segment(s) in Transaction names
Segment Numbers *Enter a comma separated list of parameter numbers (e.g. 1,3,4)*

☐ Use a parameter value in Transaction names
Parameter Name

☐ Use a header value in Transaction names
Header Name

☐ Use a cookie value in Transaction names
Cookie Name

☐ Use a session attribute value in Transaction names
Session Attribute Key

☐ Use the request method (GET/POST/PUT) in Transaction names

☐ Use the request host Transaction in names

☐ Use the request originating address in Transaction names

☒ Apply a custom expression on HttpServletRequest and use the result in Transaction Names [Explain This](#)

enter expression here

Identify Transactions for Grails Applications

Configure transaction identification for Grails applications by changing the default naming scheme to use the full URI.

To configure business transactions for a Grails application

1. On the left hand side navigation of the AppDynamics User Interface, go to **Configure -> Instrumentation**.
2. Select the **Transaction Detection** section.
3. Select the tier for which you want to enable identification.
4. Go to the **Entry Points** section.
5. Under the **Servlet** section, click **Configure Naming**.
This opens the configuration screen for modifying all the Servlets based transactions.
6. Select the **Use the Full URI** option.

New Business Transaction Match Rule - Servlet

Name:

Enabled: ☒

Priority:

Transaction Match Criteria | Split Transactions Using Request Data | Split Transactions Using Payload

☒ Split Transactions using XML/JSON Payload or a Java method invocation

Split Mechanism: ☐ XPath Expressions ☐ Java XML Binding ☐ JSON ☒ POJO Method Call

[Help with POJO Splitting](#)

Class Name:

Method Name:

Return Type:

Number of Arguments:

Argument Index:

Method Call Chain: [+](#)

For example: getPerson().getAddress().getStreet()

This custom rule generates business transactions that are named in the following pattern: "<Name of the Custom Rule>.<path to jsp>".

You can later rename these business transactions to a more user-friendly name if you like.

Struts Entry Points

- **Struts-based Transactions**
 - Struts Request Names
- Custom Match Rules for Struts Transactions
- Exclude Rules for Struts Actions or Methods
 - Problem When Custom-built Dispatch Servlet is Not Excluded

Struts-based Transactions

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.

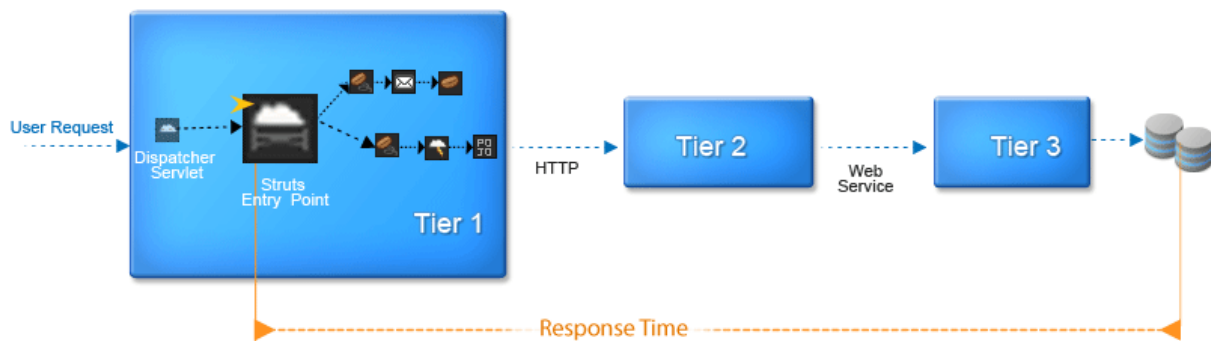
AppDynamics supports following versions of Struts:

- 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.

The following figure shows the identification process for Struts Action entry points.



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 [Business Transaction List Operations](#).

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 creating the rule 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 an exclude rule. See [Exclude Rules](#). The criteria for Struts exclude rules are the same as those for custom match rules.

Problem When Custom-built Dispatch Servlet is Not Excluded

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 App Agent for Java 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.

Web Service Entry Points

- [Web Services-based Transactions](#)
 - [Default Naming](#)
 - [Custom Match Rules for Web Services](#)
 - [Exclude Rules](#)
 - [Transaction Splitting for Web Services Based on Payload](#)

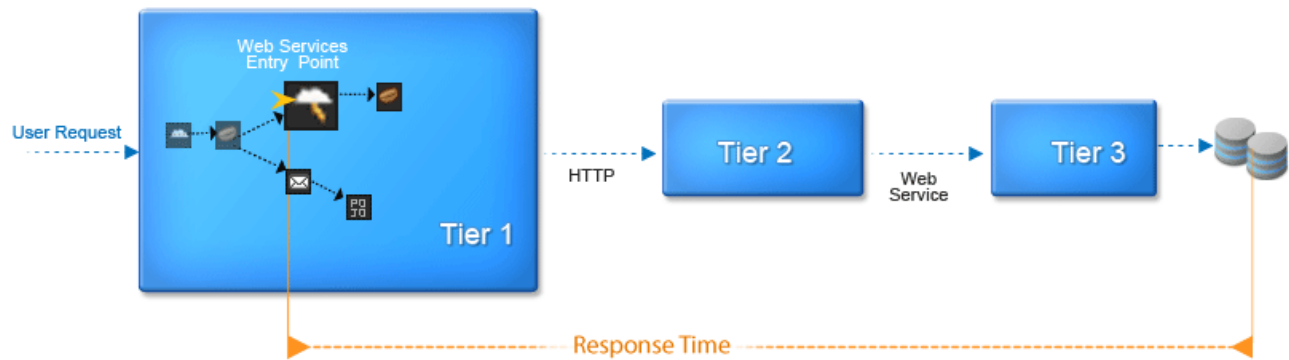
This topic discusses Web Service Entry Points.

Web Services-based Transactions

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 end point 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 [Business Transaction List Operations](#).

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":

New Business Transaction Match Rule - Web Service

Name:

Enabled: ☒

Business Transaction Match Criteria

<input checked="" type="checkbox"/>	Web Service Name	Equals	<input type="text" value="CartService"/>
<input checked="" type="checkbox"/>	Operation Name	Is Not Empty	<input type="text"/>

Cancel Create Custom Match Rule

Exclude Rules

To exclude specific Web Services or operation names from detection, add an exclude rule. See [Exclude 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.

POJO Entry Points

- [Considerations for Defining a POJO Entry Point](#)
- [Defining a POJO Entry Point](#)
 - [To define a POJO Entry Point](#)
 - [Defining an Entry Point for a Method Using the Fully-Qualified Class Name](#)
 - [Defining an Entry Point for Classes that have Methods with Multiple Parameters](#)
 - [Defining an Entry Point for a Method that Extends a Super Class](#)
 - [Defining an Entry Point for a Method Using an Interface](#)
 - [Defining an Entry Point for a Method Using Annotations](#)
 - [Dynamically Identifying POJO Transactions Using Transaction Splitting](#)
 - [Defining an Entry Point based on a Parameter Value](#)
 - [To configure transaction splitting](#)
 - [Using a Method parameter for dynamically naming the transactions](#)
 - [Exclude Rules for POJO Transactions](#)
- [Identifying a POJO Transaction as a Background Task](#)
- [Learn More](#)

Not all business processing can be implemented using Web entry points for popular frameworks. Your application may perform batch processing in all types of containers. You may be using a framework that AppDynamics does not automatically detect. Or maybe you are using pure Java.

In these situations, to enable detection of your business transaction, configure a [custom match rule](#) for a POJO (Plan Old Java Object) entry point. The rule should be defined on the class/method that is the most appropriate entry point. Someone who is familiar with your application code should help make this determination. See [Considerations for Defining a POJO Entry Point](#).

AppDynamics measures performance data for POJO transactions as for any other transactions. 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.

Considerations for Defining a POJO Entry Point

The POJO entry point is the Java method that starts the transaction.

The most important consideration in defining a POJO entry point is to choose a method that begins and ends every time the specific business transaction is invoked.

For example, consider the method execution sequence:

```
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.

Defining a POJO Entry Point

To define a POJO Entry Point

1. Click **Configure -> Instrumentation**.
2. In the Transaction Detection tab select the tier.
3. Click **Use Custom Configuration for this Tier**.
4. Scroll down to the Custom Rules panel and click **Add** (the plus sign).
5. From the Entry Point Type dropdown menu select **POJO** and click **Next**.
6. In the **New Business Transaction Match Rule - POJO** window set the criteria for identifying the entry point.
7. Save the configuration.

 If you are running on IBM JVM v1.5 or v1.6, you must restart the JVM after defining the custom match rules.

You may optionally refine the naming of a POJO-based transaction by transaction splitting.

See the following sections for examples:

- [Defining an Entry Point for a Method Using the Fully-Qualified Class Name](#)
- [Defining an Entry Point for a Method Using a Superclass](#)
- [Defining an Entry Point for a Method Using an Interface](#)
- [Defining an Entry Point for a Method Using Annotations](#)
- [Identify a POJO Transaction as a Background Task](#)

See also [Custom Match Rules](#).

Defining an Entry Point for a Method Using the Fully-Qualified Class Name

For the class named "Foo" and method `doWork()`, match on the fully-qualified class name and method name:

New Business Transaction Match Rule - POJO

Name:

Enabled: ☒

Background Task: ☐

Transaction Match Criteria | Transaction Splitting | Exclude Rule

Define match criteria for a POJO method which will be an entry point for a Business Transaction

☒ Match Classes * with a Class Name that Equals com.acme.Foo

☒ Method Name * Equals doWork

☐ NOT Condition
Selecting this will reverse the condition and return true if NOT (condition)

Click to create a Not Equal condition

Cancel Create Custom Match Rule

After you define and save the custom match rule for the POJO transaction, performance data for it displays in the [Business Transactions List](#).

	Name	Service Lev...	Time (ms)	Calls	Calls / min	Errors	Error %	Slow Reque...	Very Slow ...	Stalled ...	Tier	Type
POJO	ProcessOrder	✓	678	260	130	0	0	62	22	0	ACMETier	POJO

Business transaction is identified using the custom rule.

The name of the POJO-based business transaction is the name of the custom match rule or entry point definition.

In the example above, AppDynamics shows that the "doWork" method of class "Foo" was invoked 260 times and had an average response time of 678 ms (for the selected time range). Out of the total invocations, 62 invocations were slow and 22 invocations were very slow (these slow or very slow identification is based on the thresholds set for the business transaction).

Defining an Entry Point for Classes that have Methods with Multiple Parameters

For example, we want to instrument one of more methods in the following class:

```
class A
{
    public void m1();
    public void m1(String a);
}
```

```
public void m1(String a, com.mycompany.MyObject b);
}
```

- To instrument all the methods in the class, create a POJO-based business transaction match rule as follows:

Match Classes with a Class Name that Equals A
Method Name Equals m1(java.lang.String)

- To instrument only the method with one parameter, create a POJO-based business transaction match rule as follows:

Match Classes with a Class Name that Equals A
Method Name Equals m1(java.lang.String)

- To instrument only the method with two parameters, create a POJO-based business transaction match rule as follows:

Match Classes with a Class Name that Equals A
Method Name Equals m1(java.lang.String, com.mycompany.MyObject)

Defining an Entry Point for a Method that Extends a Super Class

For example, the entry point is based on the com.acme.AbstractProcessor super class, which defines a process() method, which is extended by its child classes: SalesProcessor, InventoryProcessor, BacklogProcessor.

Define the custom rule on the method defined in the super class:

The screenshot shows a window titled "New Business Transaction Match Rule - POJO". It contains several fields and checkboxes. The "Name" field is set to "Process". The "Enabled" checkbox is checked, and the "Background Task" checkbox is unchecked. Below these are three tabs: "Transaction Match Criteria", "Transaction Splitting", and "Exclude Rule". The "Transaction Match Criteria" tab is active, showing a section titled "Define match criteria for a POJO method which will be an entry point for a Business Transaction". This section contains two rows of criteria. The first row is "Match Classes *" with a dropdown menu set to "that extends a Super Class that", a comparison operator dropdown set to "Equals", and a text field containing "com.acme.AbstractProcessor". The second row is "Method Name *" with a dropdown menu set to "Equals" and a text field containing "process". At the bottom right of the dialog are "Cancel" and "Create Custom Match Rule" buttons.

Defining an Entry Point for a Method Using an Interface

Define a custom rule matching on an interface named com.acme.IProcessor, which defines a process() method that is implemented by the SalesProcessor, InventoryProcessor, BacklogProcessor classes.

New Business Transaction Match Rule - POJO

Name:

Enabled: ☒

Background Task: ☐

Transaction Match Criteria | Transaction Splitting | Exclude Rule

Define match criteria for a POJO method which be will an entry point for a Business Transaction

Match Classes * ☐ that implements an interface which Equals

Method Name * ☐ Equals

Cancel Create Custom Match Rule

Defining an Entry Point for a Method Using Annotations

For example, if all processor classes are annotated with "@com.acme.Processor", a custom rule should be defined using annotation.

New Business Transaction Match Rule - POJO

Name:

Enabled: ☒

Background Task: ☐

Transaction Match Criteria | Transaction Splitting | Exclude Rule

Define match criteria for a POJO method which be will an entry point for a Business Transaction

Match Classes * ☐ that has an Annotation which Equals

Method Name * ☐ Equals

Cancel Create Custom Match Rule

By default, in these cases the business transaction started when any process() is invoked is named Process, based on the name of the custom rule. To refine the transaction name to reflect the specific method called (Process.SalesProcessor, Process.InventoryProcessor, Process.BacklogProcessor) use transaction splitting.

Dynamically Identifying POJO Transactions Using Transaction Splitting

By default, when you create a custom match rule for POJO transactions, all the qualifying requests are identified by the name of the custom match rule.

However, in many situations it is preferable to split POJO-based transactions, especially for nodes that execute scheduled jobs.

For example, if multiple classes have the same method and are instrumented using the same rule, when the method is invoked the class name of the instance being invoked can be used to classify the request.

If you split the transaction based on the simple class name, instead of one business transaction named Process, the transaction that is started when the process() method is invoked is named based on the rule name combined with the class name: either Process.SalesProcessor, Process.InventoryProcessor, or Process.BacklogProcessor.

Defining an Entry Point based on a Parameter Value

In some cases you want to split the transaction based on the value of a parameter in the entry point method. For example, you could configure the split on the following process() method:

```
public void process(String jobType,String otherParameters...)
```

where the jobType parameter could be Sales, Inventory or Backlog.

The screenshot shows a configuration window titled "New Business Transaction Match Rule - POJO". It has three tabs: "Transaction Match Criteria", "Transaction Splitting", and "Exclude Rule". The "Transaction Match Criteria" tab is active. It contains several checkboxes and input fields. The first checkbox, "Use a method parameter to name Transactions", is checked. Below it, "Parameter Index" is set to 0 and "Getter Chain to run on method parameter" is set to toString(). There is a green plus icon to the right of this section. Below that, "Use the POJO Object Instance to name Transactions" is unchecked, with a text field containing "getX().getY()" and another green plus icon. Several other checkboxes for naming conventions (Full Class Name, Simple Class Name, Thread ID, Thread Name, Method Name, Simple Class Name and Method Name, Full Class Name and Method Name) are all unchecked. At the bottom right are "Cancel" and "Create Custom Match Rule" buttons.

You can name POJO transactions dynamically using the following mechanisms:

- method parameter
- POJO object instance
- fully qualified class name
- simple class name
- thread ID
- thread name
- method name
- simple class name and method name
- full class name and method name

In all cases, the name of the rule is prepended to the dynamically-generated name to form the business transaction name.

To configure transaction splitting

1. In the Transaction Splitting tab of the **New Business Transaction Match Rule - POJO** window, click **Split POJO Transactions using one of the following mechanisms...**

2. Click the mechanism to use to split the transaction.

- If you are specifying a method parameter, enter the zero-based parameter index of the parameter.
- If the parameter is a complex type, specify the getter chain to use used to derive the transaction name.
- If you are specifying a POJO object instance, specify the getter chain. See [Getter Chains in Java Configurations](#).

3. Click **Save**.

The screenshot shows the 'New Business Transaction Match Rule - POJO' window with the 'Transaction Splitting' tab active. The 'Name' field is empty, 'Enabled' is checked, and 'Background Task' is unchecked. Under 'Transaction Match Criteria', the 'Transaction Splitting' sub-tab is selected. The following options are listed:

- ☐ Use a method parameter to name Transactions
 - Parameter index: 0
 - Getter Chain to run on method parameter: toString()
- ☐ Use the POJO Object Instance to name Transactions
 - Define match criteria for a POJO method which be will an entry point for a Business Transaction: getX().getY()
- ☒ Use the Simple Class Name in Transaction names
- ☐ Use the Full Class Name in Transaction names
- ☐ Use the Thread ID in Transaction names
- ☐ Use the Thread Name in Transaction names
- ☐ Use the Method Name in Transaction names
- ☐ Use the Simple Class Name and Method Name in Transaction names
- ☐ Use the Full Clas Name and Method Name in Transaction names

A callout box points to the selected option, stating: "For example, when selected, the business transaction list dashboard displays the following business transactions: Process.SalesProcessor, Process.InventoryProcessor, Process.BacklogProcessor".

Buttons at the bottom: Cancel, Create Custom Match Rule.

Using a Method parameter for dynamically naming the transactions

Suppose in the ACME Online example, instead of the super-class or interface, the type of the processing is passed in as a parameter.

For example:

```
public void process(String jobType,String otherParameters...)
```

In this case, it would be appropriate to name the transaction based on the value of Job type. This Job type is passed as the parameter.

To specify a custom rule for method parameter:

1. Specify the details for the custom rule in the **Transaction Match Criteria** tab.

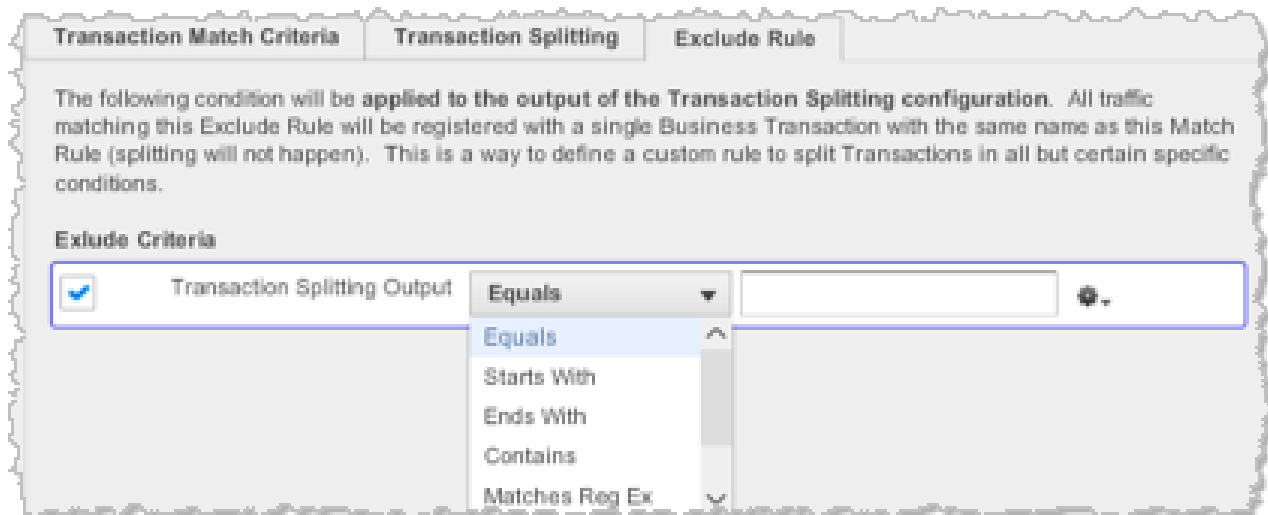
2. Click the **Transaction Splitting** tab.
3. Click **Split POJO Transactions using one of the following mechanisms...**
4. Select the option for the method parameter.
5. Specify the details for the parameters.

You can use a getter chain if the parameter is of complex type in order to derive a string value, which can then be used for the transaction name. See [Getter Chains in Java Configurations](#).

The screenshot shows the 'New Business Transaction Match Rule - POJO' dialog box. The 'Transaction Splitting' tab is active. Under the 'Split POJO Transactions using one of the following mechanisms...' section, the first option, 'Use a method parameter to name Transactions', is selected with a checked checkbox. Below this, the 'Parameter index' is set to 0, and the 'Getter Chain to run on method parameter' is set to 'toString()'. There is a green '+' icon to the right of this option. Below this, there is an unchecked checkbox for 'Use the POJO Object Instance to name Transactions', followed by a text field containing 'getX().getY()' and a green '+' icon. Further down, there are several other unchecked checkboxes for different naming conventions: 'Use the Full Class Name in Transaction names', 'Use the Simple Class Name in Transaction names', 'Use the Thread ID in Transaction names', 'Use the Thread Name in Transaction names', 'Use the Method Name in Transaction names', 'Use the Simple Class Name and Method Name in Transaction names', and 'Use the Full Class Name and Method Name in Transaction names'. At the bottom right, there are two buttons: 'Cancel' and 'Create Custom Match Rule'.

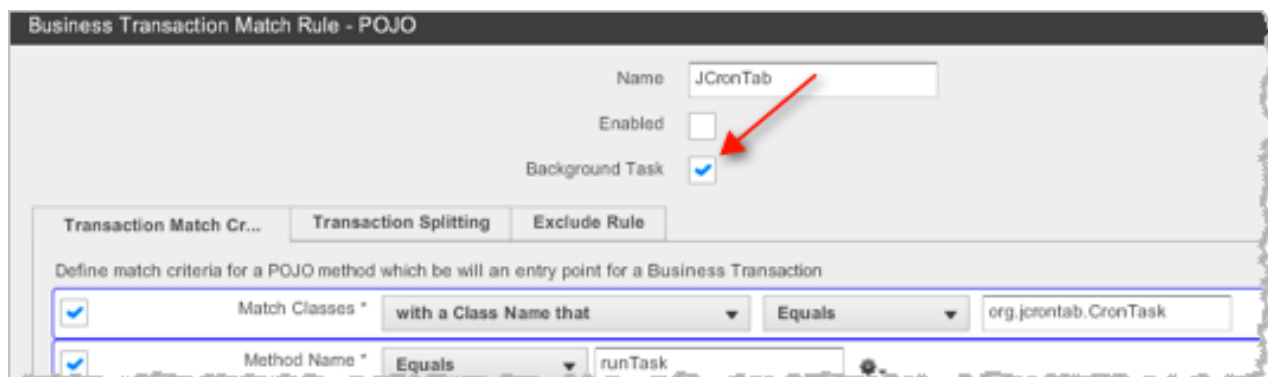
Exclude Rules for POJO Transactions

To prevent configured transaction splitting from being applied in certain situations, create an exclude rule defined on the output of the transaction splitting.



Identifying a POJO Transaction as a Background Task

When you want to specify that a POJO transaction is a background task, check **Background Task**.



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. See [Background Task Monitoring](#).

Learn More

Identify Transactions Based on POJO Method Invoked by a Servlet

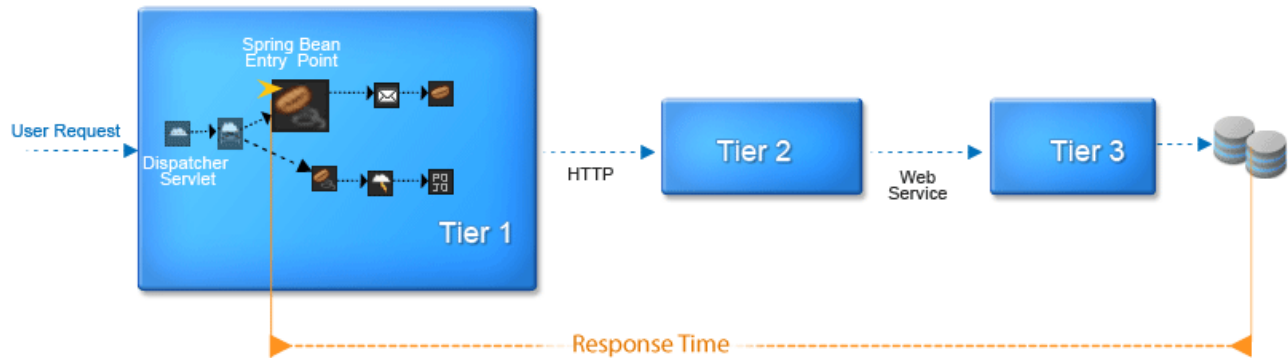
Spring Bean Entry Points

- [Spring Bean-based Transactions](#)
- [Default Naming for Spring Bean Requests](#)
 - [To Enable Auto-discovery for Spring Bean entry points](#)
- [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.



Default Naming for Spring Bean Requests

When the automatic discovery for a Spring Bean based request is turned on, 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.

To Enable Auto-discovery for Spring Bean entry points

1. Access the transaction detection configurations screen and select the tier to configure. See [To Access Business Transaction Detection Configuration](#)
2. In the Spring Bean entry in the Entry Points section check the Automatic Transaction Detection check box.

▼ Entry Points

Type	Transaction Monitoring	Automatic Transaction Detection
Servlet	<input checked="" type="checkbox"/> Enabled	<input checked="" type="checkbox"/> Discover Transactions automatically for all Servlet requests <input type="checkbox"/> Enable Servlet Filter Detection
Struts Action	<input checked="" type="checkbox"/> Enabled	<input checked="" type="checkbox"/> Discover Transactions automatically for all Struts Action invocations Transactions will be named: ActionName.MethodName
Web Service	<input checked="" type="checkbox"/> Enabled	<input checked="" type="checkbox"/> Discover Transactions automatically for all Web Service requests Transactions will be named: ServiceName.OperationName
POJO	<input checked="" type="checkbox"/> Enabled	Any Java method can be the entry point for a Business Transaction. The class to which the method belongs to can be picked using different parameters like its name, its super class name, the interfaces it implements, or the annotations it has.
Spring Bean	<input checked="" type="checkbox"/> Enabled	<input checked="" type="checkbox"/> Discover Transactions automatically for all Spring Bean invocations Transactions will be named: BeanName.MethodName
EJB	<input checked="" type="checkbox"/> Enabled	<input type="checkbox"/> Discover Transactions automatically for all EJB invocations Transactions will be named: EJBName.MethodName

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. See [Custom Match Rules](#).

The following example creates a custom match rule for the placeOrder method in the orderManager bean.

New Business Transaction Match Rule - Spring Bean

Name:

Enabled: ☒

Business Transaction Match Criteria

<input checked="" type="checkbox"/>	Bean ID	Contains	<input type="text" value="orderManager"/>
<input checked="" type="checkbox"/>	Method	Equals	<input type="text" value="placeOrder"/>
<input type="checkbox"/>	Class Name	Equals	<input type="text"/>
<input type="checkbox"/>	Extends	Equals	<input type="text"/>
<input type="checkbox"/>	Implements	Equals	<input type="text"/>

This custom rule will name all the qualifying requests as "ACME.orderManager.placeOrder".

Exclude Rules Spring Bean Transactions

To exclude specific Spring Bean transactions from detection add an exclude rule. See [Exclude](#)

Rules. The criteria for Spring Bean exclude rules are the same as those for custom match rules.

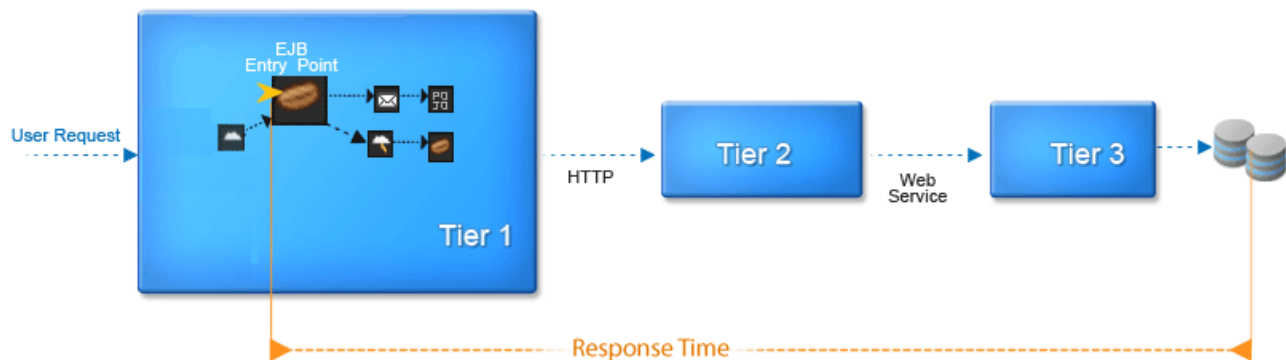
EJB Entry Points

- EJB-Based Business Transactions
- Default Naming for EJB Entry Points
 - To Enable the Auto-discovery for EJB Transactions
- Custom Match Rules for EJB based Transactions
- Exclude Rules for EJB Transactions

This topic describes how to configure transaction entry points for the EJB based requests.

EJB-Based Business 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 <EJBName>.<MethodName>. 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.

Keep in mind the following before you enable auto-discovery for EJB based transactions:

- 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. This is because AppDynamics supports distributed transaction correlation.
- 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.

To Enable the Auto-discovery for EJB Transactions

1. Access the transaction detection configurations screen and select the tier to configure. See [To Access Business Transaction Detection Configuration](#).
2. In the EJB entry in the Entry Points section check the Automatic Transaction Detection check

box.

Type	Transaction Monitoring	Automatic Transaction Detection
Servlet	<input checked="" type="checkbox"/> Enabled	<input checked="" type="checkbox"/> Discover Transactions automatically <input type="checkbox"/> Enable Servlet Filter Detection
Struts Action	<input checked="" type="checkbox"/> Enabled	<input checked="" type="checkbox"/> Discover Transactions automatically Transactions will be named: Action
Web Service	<input checked="" type="checkbox"/> Enabled	<input checked="" type="checkbox"/> Discover Transactions automatically Transactions will be named: Service
POJO	<input checked="" type="checkbox"/> Enabled	Any Java method can be the entry point for a Transaction. The class to which the method is picked using different parameters like name, the interfaces it implements, or the package it belongs to.
Spring Bean	<input checked="" type="checkbox"/> Enabled	<input type="checkbox"/> Discover Transactions automatically Transactions will be named: BeanName
EJB	<input checked="" type="checkbox"/> Enabled	<input type="checkbox"/> Discover Transactions automatically Transactions will be named: EJBName

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 a EJB based transaction. See [Custom Match Rules](#).

The following example creates a custom match rule for the receiveOrder method in the TrackOrder bean. The transactions are named "ACME_EJB.TrackOrder.receiveOrder".

New Business Transaction Match Rule - EJB

Name:

Enabled: ☒

Business Transaction Match Criteria

<input checked="" type="checkbox"/>	EJB Name	Equals	<input type="text" value="TrackOrder"/>	
<input checked="" type="checkbox"/>	Method	Equals	<input type="text" value="receiveOrder"/>	
<input type="checkbox"/>	EJB Type	Message Driven		
<input type="checkbox"/>	Class Name	Equals	<input type="text"/>	
<input type="checkbox"/>	Extends	Equals	<input type="text"/>	
<input type="checkbox"/>	Implements	Equals	<input type="text"/>	

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 an exclude rule. See [Exclude Rules](#). The criteria for EJB exclude rules are the same as those for custom match rules.

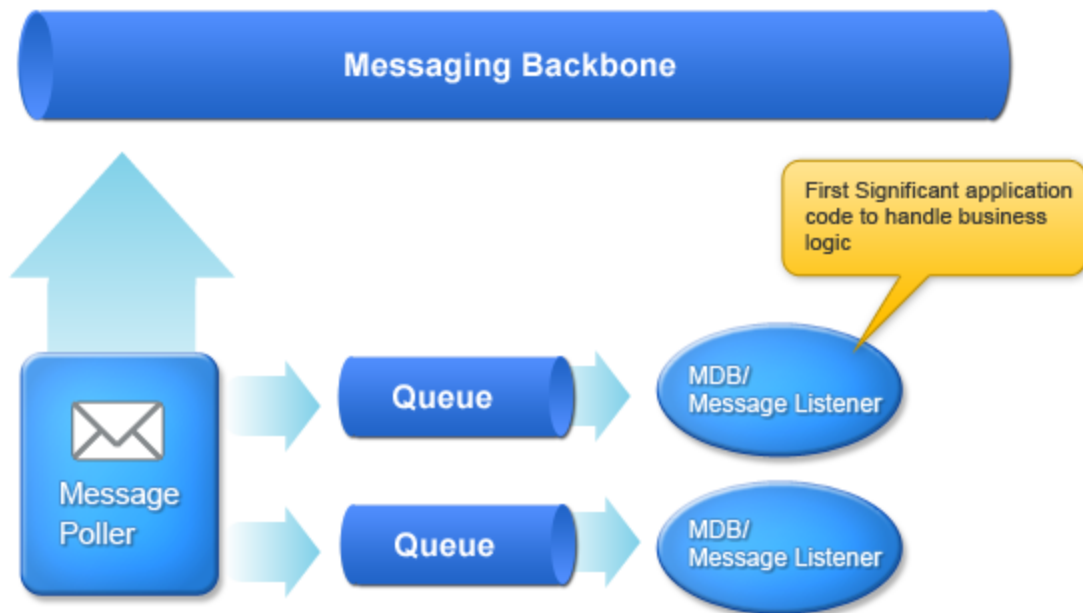
JMS Entry Points

- [Messaging Entry Points](#)
 - [Default Naming Conventions](#)
 - [To Access the Default Configuration for Messaging Entry Points](#)
 - [Custom Match Rules for Messaging Entry Points](#)
 - [Grouping Example](#)
 - [Message Payload Example](#)
- [Learn More](#)

This topic discusses messaging entry points configuration.

Messaging Entry Points

When an application uses asynchronous message listeners or message driven beans, such as JMS or equivalent MQ providers as the primary trigger for business processing on the entry point tier AppDynamics can intercept the message listener invocations and track them as business transactions. This is relevant only for the entry point tier.



You can define messaging entry points for queue listeners to monitor Service Level Agreements (SLAs). An SLA is often reflected in the rate of processing by a queue listener. When you monitor a messaging entry point, you can track the rate of processing by a particular queue listener.

Default Naming Conventions

AppDynamics automatically detects and names the messaging entry points. When a message listener is invoked, the transaction is named after the destination name (the queue name) or the listener class name if the destination name is not available.

To Access the Default Configuration for Messaging Entry Points

1. Access the business transaction configuration page.
See [To Access Business Transaction Detection Configuration](#).
2. Scroll down to message entry point entry for your framework.

Custom Match Rules for Messaging Entry Points

If you want finer control over naming messaging requests, you can use custom match rules either to specify names for your messaging entry points or to group multiple queue invocations into a single business transaction.

See [To Create Custom Match Rules](#) for general information about configuring custom match rules.

Grouping Example

The following custom match rule groups all transactions to queues starting with the word "Event"

New Business Transaction Match Rule - JMS

Name:

Enabled: ☒

Business Transaction Match Criteria

☒ Message Destination: Queue Starts With Event

☐ Message Property: Check for property existence
Property Type: BOOLEAN
Property Name:

☐ Message Content: Equals
Applies to text messages only

Message Payload Example

The following custom rule uses message properties (headers) to define the custom match rule. You can also use the message content. You can also use message properties or message content to define custom exclude rules to exclude certain messaging entry points from detection.

New Business Transaction Match Rule - JMS

Name:

Enabled: ☒

Business Transaction Match Criteria

☒ Message Destination: Queue Starts With Event

☒ Message Property: Check for property value
Property Type: BOOLEAN
Property Name: customerType
Property Value: Priority

☐ Message Content: Equals
Applies to text messages only

Learn More

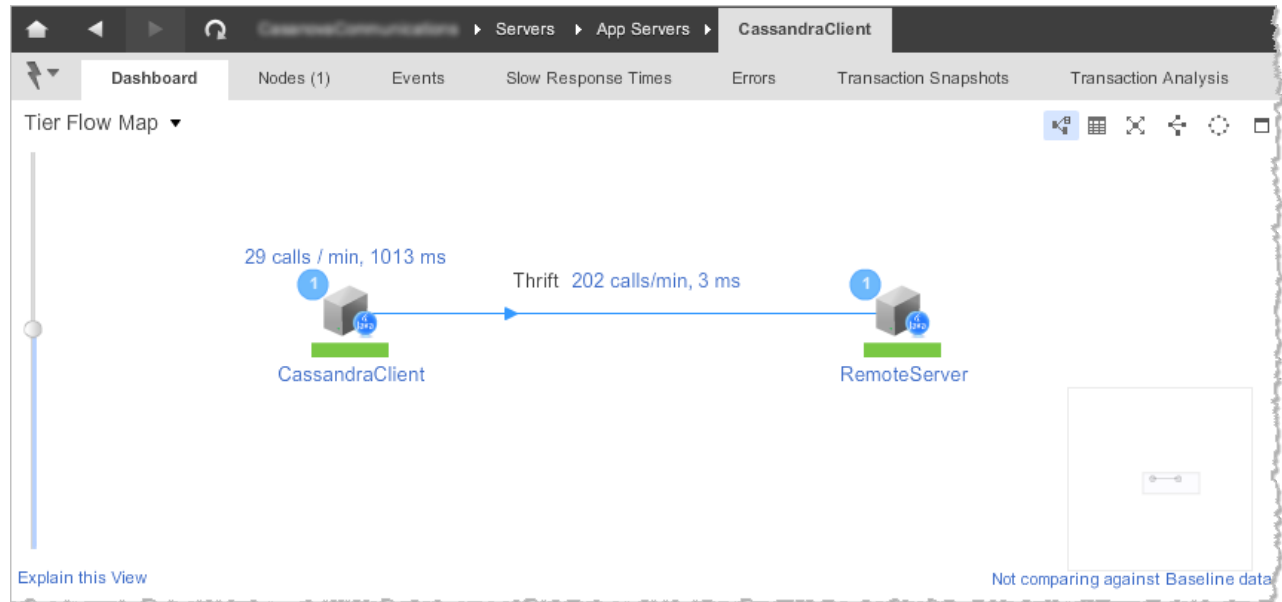
- [Configure Business Transaction Detection](#)

Binary Remoting Entry Points for Apache Thrift

- [Default Naming for Binary Remoting \(Thrift\) Entry Points](#)
- [Enabling Auto-discovery for Binary Remoting \(Thrift\) Entry Points](#)
- [Creating Custom Match Rules for Binary Remoting \(Thrift\) Requests](#)

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 transaction entry points for Cassandra with Thrift framework applications.



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
```

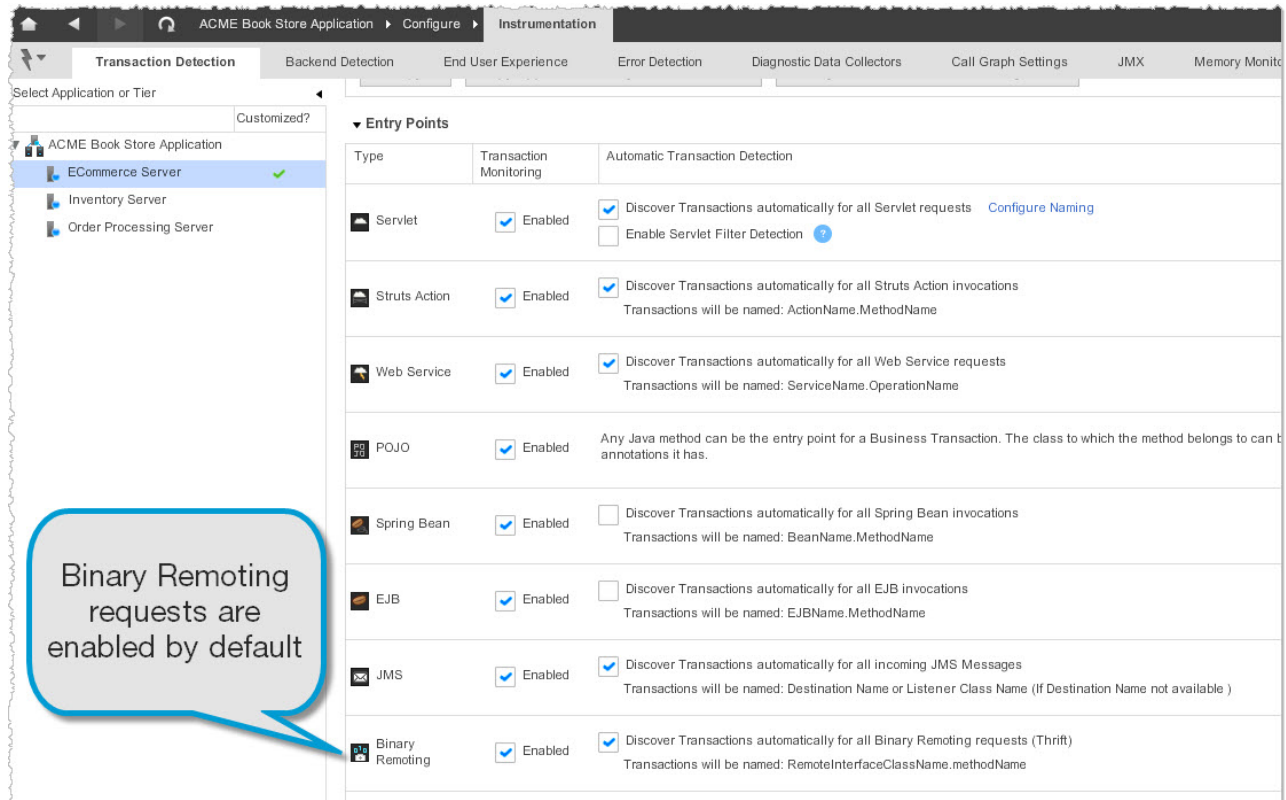
By default, transaction discovery for Binary Remoting (Thrift)-based request is enabled.

Enabling Auto-discovery 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**, you should ensure they have been enabled as follows.

1. Access the transaction detection configuration window and select the tier to configure. See [To Access Business Transaction Detection Configuration](#).

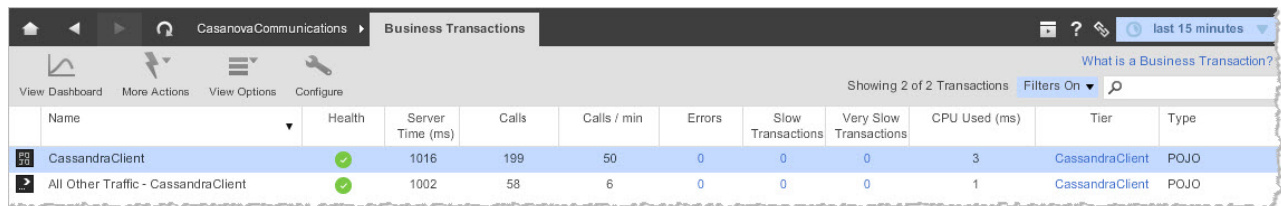
2. In the **Binary Remoting** entry of the **Entry Points** section, click **Automatic Transaction Detection**.



Binary Remoting requests are enabled by default

Type	Transaction Monitoring	Automatic Transaction Detection
Servlet	<input checked="" type="checkbox"/> Enabled	<input checked="" type="checkbox"/> Discover Transactions automatically for all Servlet requests Configure Naming <input type="checkbox"/> Enable Servlet Filter Detection
Struts Action	<input checked="" type="checkbox"/> Enabled	<input checked="" type="checkbox"/> Discover Transactions automatically for all Struts Action invocations Transactions will be named: ActionName.MethodName
Web Service	<input checked="" type="checkbox"/> Enabled	<input checked="" type="checkbox"/> Discover Transactions automatically for all Web Service requests Transactions will be named: ServiceName.OperationName
POJO	<input checked="" type="checkbox"/> Enabled	Any Java method can be the entry point for a Business Transaction. The class to which the method belongs to can be annotated with the @BusinessTransaction annotation.
Spring Bean	<input checked="" type="checkbox"/> Enabled	<input type="checkbox"/> Discover Transactions automatically for all Spring Bean invocations Transactions will be named: BeanName.MethodName
EJB	<input checked="" type="checkbox"/> Enabled	<input type="checkbox"/> Discover Transactions automatically for all EJB invocations Transactions will be named: EJBName.MethodName
JMS	<input checked="" type="checkbox"/> Enabled	<input checked="" type="checkbox"/> Discover Transactions automatically for all incoming JMS Messages Transactions will be named: Destination Name or Listener Class Name (If Destination Name not available)
Binary Remoting	<input checked="" type="checkbox"/> Enabled	<input checked="" type="checkbox"/> Discover Transactions automatically for all Binary Remoting requests (Thrift) Transactions will be named: RemoteInterfaceClassName.methodName

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.



Name	Health	Server Time (ms)	Calls	Calls / min	Errors	Slow Transactions	Very Slow Transactions	CPU Used (ms)	Tier	Type
CassandraClient		1016	199	50	0	0	0	3	CassandraClient	POJO
All Other Traffic - CassandraClient		1002	58	6	0	0	0	1	CassandraClient	POJO

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.

To enable detection of your binary remoting business transactions, configure a [custom match rule](#). The rule should be defined on the class/method that is the most appropriate entry point. Someone who is familiar with your application code should help make this determination.

A custom match rule lets you specify customized names for your Binary Remoting (Thrift) based requests. You can also group multiple Thrift invocations into a single business transaction using custom match rules. See [Custom Match Rules](#) for information about accessing the configuration windows.

The matching criteria for creating the custom entry point rule for a Thrift request are the POJO class name and method name of the business transaction initiating the binary remote call. Transaction splitting and exclude rules are also supported. 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 consistent with those for POJO entry points.

CometD Support

CometD is a scalable HTTP-based event routing bus for transporting asynchronous messages over HTTP. CometD provides both a JavaScript API and Java API and is used for applications such as multi-player games and chat rooms. AppDynamics supports both the

WebSocket and HTTP long-polling client transports. CometD sends messages to channels and we can now track messages through the channels.

New rules defined in the agent:

- New POJO rules to gather message activity
- Business transaction split configuration on the POJO rules to determine which channel the message was published to, tracks business transaction per channel
- New servlet exclude rules - since the CometD Servlet implements the different transports activity, you need to exclude the Servlets implementing the CometD transports. CometD is generally contained within a Jetty container, you need to exclude the Jetty container tracking in order to see the CometD messages contained within the transaction. [CometD servlets](#) are excluded by default. The following are the Jetty Servlet exclude rules:
 - EQUALS, org.eclipse.jetty.server.handler.ContextHandler
 - EQUALS, org.eclipse.jetty.servlet.ServletContextHandler
 - ENDSWITH, jetty.plugin.JettyWebAppContext



	Name	Health	Server Time (ms)	Calls	Calls / min
PO TO	CometD./market/nasdaq	✓	0	565	283
PO TO	CometD./market/nikkei_225	✓	0	439	439
PO TO	CometD./market/nyse	✓	0	36	18
PO TO	CometD./meta/connect	✓	0	20	10
PO TO	/cometd	✓	16	8	4
PO TO	CometD./meta/handshake	✓	2	8	3
PO TO	CometD./foobar/java_client	✓	1	5	2
PO TO	CometD./meta/subscribe	✓	0	4	1

Mule ESB Support

Mule ESB 3.4 and previous releases are supported.

Mule ESB (Enterprise Service Bus) is an integration platform with many different connectors for integrating applications and supporting service oriented architectures. By default, AppDynamics detects the Mule ESB endpoint. However, in some situations, you may need to create a [servlet exclude rule](#).

The App Agent for Java supports Mule ESB as follows:

Tracks Business Transactions to Remote Services and Tiers

No configuration required: The App Agent for Java detects business application calls through Mule ESB service connections to remote services and tiers. Mule ESB is automatically detected and continuing tiers are recognized. Asynchronous correlation is enabled by default. Business transaction naming is dependent on business transaction discovery.

Detects incoming HTTP to the Mule ESB HTTP endpoint when it performs as a servlet

No configuration required: If your application takes the output of the Mule ESB HTTP endpoint and makes it perform like a servlet, the App Agent for Java detects incoming HTTP to the Mule HTTP Endpoint as a servlet.

Detects SOAP or RESTful operations beyond the Mule ESB HTTP Endpoint

Servlet Exclude Rule Required: Mule ESB can function as a transport connector. In some cases the Mule ESB HTTP endpoint is an entry point to another application component. For example, the Mule ESB HTTP endpoint can be an entry point to an application that ends with a CXF SOAP service or a JAX-RS RESTful endpoint further in the tier. By default, the App Agent for Java treats the Mule ESB HTTP output as a servlet, an endpoint in the application and doesn't see the CXF SOAP or JAX-RS RESTful operation further in the flow. In order to see the SOAP or RESTful operation, we need to exclude the Mule ESB HTTP servlet.

For example, we have an application using Mule ESB services at this URI: `http://muleapp1:8080/`

There are some HTTP response servlets on: `http://muleapp1:8080/httpResponseService/`

and there is a CXF SOAP endpoint

on: `http://muleapp1:8080/webservicesendpoint/`

To see the CXF SOAP endpoint, we need to create a servlet exclude rule on
uri:`http://muleapp1:8080/webservicesendpoint/`

We do not need to create a servlet exclude rule when the Mule ESB HTTP endpoint continues to another tier or for exit points within Mule ESB.

See also, [Mule ESB Startup Settings](#).

JAX-RS Support

AppDynamics Agent for Java 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](#)

Examples

Using Default Settings

By default the business transaction is named using the first two segments of the REST URI and the name of the HTTP method, separated by a period. The application name part of the URI is not used.

```
rest-transaction-naming={rest-uri}.{http-method}
```

For example, using default settings, when the App Agent for Java detects this REST resource as an entry point,

`/REST/MyApplication/MyResource/resource`

The agent uses the URI of the REST resource to name the business transaction.

The REST resource is accessible using the HTTP method **GET**.

So, the business transaction name would be,

MyResource/resource.GET

where **MyResource/resource** is the first two segments of the REST URI
and **GET** is the HTTP method.

Using rest-transaction-naming Properties

Using the rest-transaction-naming property, you can name the business transaction using a

number of properties. For this example, we'll use the following agent node property:

```
rest-transaction-naming={class-name}/{method-name}
```

So, when the App Agent for Java 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`.

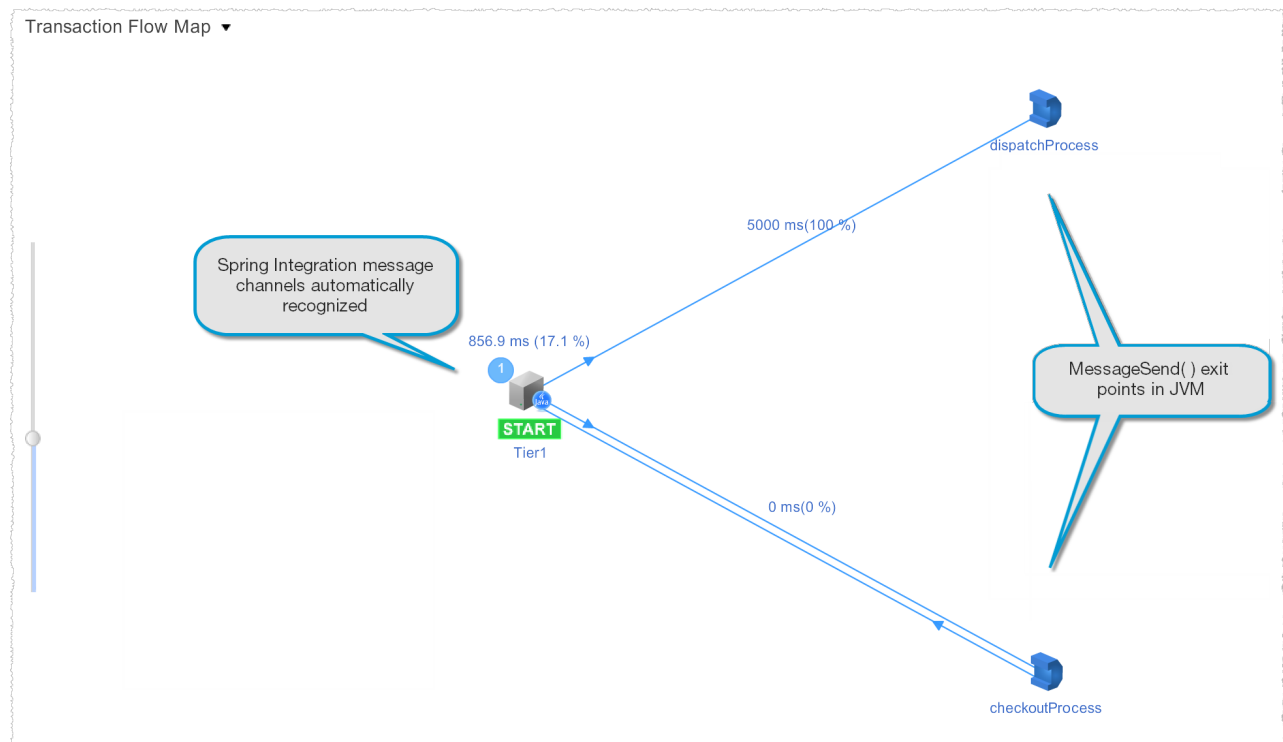
Learn More

[App Agent Node Properties Reference](#)

Spring Integration Support

In Spring-based applications, [Spring Integration](#) enables lightweight messaging and supports integration with external systems via declarative adapters.

The App Agent for Java by default automatically discovers exits for all Spring Integration Release 2.2.0 channels except 'DirectChannel.'



See the [Sample Application Flow XML](#) below.

AppDynamics Pro Agent for Java supports tracking application flow through Spring Integration [messaging channels](#). The App Agent for Java Spring Integration support is based on the MessageHandler interface.

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 ends when the iteration ends.

Entry points

Originating transactions begin with `MessageHandler.handleMessage()` implementations. If the incoming message is already recognized by the App Agent for Java then a continuing transaction is started.

Exit points

Exit points are based on `MessageChannel.send()`. Most of these message channels are typically inside the JVM so the Application Flow Maps shows a link from the tier to the message channel component name (bean name) and back.

Spring Integration Support Customizations

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 App Agent for Java tracks a pollable channel loop only if it happens inside a class/method combination similar to that defined in the following example. Configure the [spring-integration-receive-marker-classes](#) node property for each class/method combination that polls messages in a loop, then only those class/methods identified in this node property are tracked.

```
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.

Sample Application Flow XML

The following XML specifies the integration flow configuration of the application tracked in the image above.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/integration"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:beans="http://www.springframework.org/schema/beans"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/integration
    http://www.springframework.org/schema/integration/spring-integration.xsd">

  <channel id="inputChannel">
    <queue capacity="100"/>
  </channel>

  <channel id="out"/>

  <bridge input-channel="inputChannel" output-channel="out">
    <poller max-messages-per-poll="10" fixed-rate="5000"/>
  </bridge>

  <channel id="outputChannel">
    <queue capacity="100"/>
  </channel>

  <service-activator input-channel="out"
    output-channel="outputChannel"
    ref="helloService"
    method="sayHello"/>

  <channel id="out2">
    <queue capacity="100"/>
  </channel>

  <bridge input-channel="outputChannel" output-channel="out2">
    <poller max-messages-per-poll="10" fixed-rate="5000"/>
  </bridge>

  <beans:bean id="helloService"
    class="org.springframework.integration.samples.helloworld.HelloService"/>

</beans:beans>
```

In the following statements, you can see Spring Integration channels that are recognized and identified in the Application Flow Map visualized above:

```
<channel id="out">
```

```
<channel id="outputChannel">
```

Instrumenting Apple WebObjects Applications

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

After installing WebObjects, you can find most of the artifacts in the following directories:

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

We will use one of the developer examples in the following section to illustrate how to instrument an application created with Apple WebObjects.

Instrumenting a Sample App

When you run the HelloWorld application,

```
/Developer/Examples/JavaWebObjects/HelloWorld,
```

a script file is generated

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

Open up the generated script

file, `/Developer/Examples/JavaWebObjects/HelloWorld/dist/legacy/HelloWorld.woa/HelloWorld`, and towards the end of the file, line 310 in the following example, you'll see the Java execute line:

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

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

- [Getter Chains in Java Configurations](#)
- [Identify Transactions Based on POJO Method Invoked by a Servlet](#)

Exclude Rule Examples for Java

- [Exclude Business Transactions Using Exclude Rules](#)
 - [Change the Default Exclude Rule Settings](#)
 - [Use the Next Layer of Application Logic](#)
 - [Use an Exclude Rule as a Filter](#)
 - [Exclude Spring Beans of Specific Packages](#)
- [Learn More](#)

Exclude Business Transactions Using Exclude Rules

Exclude rules prevent detection of business transactions that match certain criteria. You might want to use exclude rules in the following situations:

- AppDynamics is detecting business transactions that you are not interested in monitoring.
- You need to trim the total number of business transactions in order to stay under the agent

and controller limits.

- You need to substitute a default entry point with a more appropriate entry using a custom match rule.

You can customize existing rules:

- [Change the Default Exclude Rule Settings](#)

and/or setting up new rules:

- [Use the Next Layer of Application Logic](#)
- [Use an Exclude Rule as a Filter](#)
- [Exclude Spring Beans of Specific Packages](#)

Change the Default Exclude Rule Settings

AppDynamics has a default set of exclude rules for Servlets. Sometimes you need to adjust the default exclude rules by disabling a default rule or adding your own exclude rules.

Several entry points are excluded by default as shown here:

Exclude Rules

If a Transaction matches any of the following rules, it will not be discovered.









+

✎

✖

📄

🔍

	Type	Name	Enable ▲
	Servlet	XFire web-services servlet	✓
	Servlet	Apache Axis2 Servlet	✓
	Servlet	Apache Axis2 Admin Servlet	✓
	Servlet	Struts Action Servlet	✓
	Servlet	Websphere web-services Servlet	✓
	Servlet	Websphere web-services axis Servlet	✓
	Servlet	JBoss web-services servlet	✓
	Servlet	Apache Axis Servlet	✓

Use the Next Layer of Application Logic

When the incoming request starts at some control logic in your code that triggers different business logic based on payload data, you may prefer the business logic class and method as your business transaction names. You can exclude the control logic from being identified as the business transaction, and have the business logic be identified instead.

For example, when the control logic and business logic are built as EJBs, enabling EJB discovery will result in business transactions based on the control class and method. Instead, you can create an EJB exclude rule using the match criteria **Class Name Equals** the control class name and then the business logic APIs can be discovered as business transactions. This type of exclude rule looks similar to this:

Use an Exclude Rule as a Filter

Another reason to use an exclude rule is to use it like a filter, where you allow the eligible requests and ignore everything else.

For example, you want to use default discovery rules. Your application is receiving URI ranges that start with : /a, /b ... /z, however you want to monitor URIs only when they start with /a and /b. The easiest way to achieve this is to create a Servlet exclude rule that does a Match : Doesn't Start With /a or /b as shown here:

Exclude Spring Beans of Specific Packages

In this scenario, an application has a couple of Spring Beans of the class `java.lang.String` and `java.util.ArrayList`. Because of this, all instances of these classes are being mismarked as Spring Beans with the same IDs. To fix this behavior and exclude specific Spring Bean IDs, you can define an exclude rule as follows:

1. In the left navigation panel, click **Configure -> Instrumentation**.
2. Select the Transaction Detection tab.
3. Select your tier.
4. Navigate to the Exclude Rules section.
5. Click on "+" to create a new rule and select **Spring Bean** from the **Entry Point Type** menu.
6. Provide a name for the exclude rule.
7. In the popup, enable the **Bean ID** match criteria, use the **Equals** option and type the Spring Bean ID that you want to exclude from discovery as a business transaction.

The screenshot shows a dialog box titled "New Exclude Business Transaction Match Rule - Spring Bean". It has a "Name" field with the value "Exclude-yourID" and an "Enabled" checkbox that is checked. Below this is a section titled "Business Transaction Match Criteria" with a table of criteria. The first row, "Bean ID", is checked and has a dropdown set to "Equals" and a text field with "yourID". The other rows, "Method", "Class Name", "Extends", and "Implements", are not checked and have dropdowns set to "Equals" and empty text fields. At the bottom right are "Cancel" and "Create Exclude Rule" buttons.

Criteria	Match Criteria	Value
<input checked="" type="checkbox"/> Bean ID	Equals	yourID
<input type="checkbox"/> Method	Equals	
<input type="checkbox"/> Class Name	Equals	
<input type="checkbox"/> Extends	Equals	
<input type="checkbox"/> Implements	Equals	

Learn More

- [Business Transaction Configuration Methodology for Java](#)
- [Configure Business Transaction Detection](#)
- [Match Rule Conditions](#)
- [Regular Expressions In Match Conditions](#)

Configure Multi-Threaded Transactions for Java

- [Default Configuration](#)
- [Custom Configuration](#)
 - [Managing Thread Correlation Using a Node Property](#)
- [Enabling and Disabling Asynchronous Monitoring](#)
 - [To Disable the New Method of Asynchronous Monitoring](#)

- [To Disable all Asynchronous Monitoring](#)
- [To Enable Asynchronous Monitoring](#)
- [Learn More](#)

AppDynamics collects and reports key performance metrics for individual threads in multi-threaded Java applications. See [Trace MultiThreaded Transactions for Java](#) for details on where these metrics are reported.


In addition to support for applications written in Java, applications running in the JVM that are written with Groovy are also supported. Groovy classes can be intercepted, servlets are detected by default, and exit calls and thread correlation are supported without any additional configuration.

Default Configuration

Classes for multi-threaded correlation are configured in the <excludes> child elements of the <fork-config> element in the <App_Server_Agent_Installation_Directory>/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/" />
  <!-- exclude weblogic and websphere -->
  <excludes filter-type="STARTSWITH"
filter-value="com.bea/, com.weblogic/, weblogic/, com.ibm/, net/sf/, com/mchange/" />
  . . .
```

Note: 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.

You can also explicitly include sub-packages and subclasses of excluded packages and classes. Use the <includes> or <include> child element to specify the included packages and classes.

Use the <excludes> and <includes> elements to specify a comma-separated list of classes or packages. Use the <exclude> and <include> elements to specify a single class or package

Managing Thread Correlation Using a Node Property

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](#).

Enabling and Disabling Asynchronous Monitoring

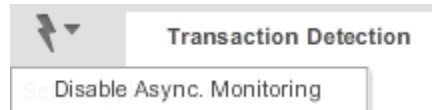
 You should disable monitoring of multi-threaded transactions on all agents if all of your agents

and your controller are not at AppDynamics version 3.6 or higher. The previous methodology for monitoring asynchronous communications was problematic. You can enable the feature after all of your agents have been upgraded.

You must restart the agent after you enable or disable this feature.

To Disable the New Method of Asynchronous Monitoring

1. In the left navigation pane click **Configure->Instrumentation->Transaction Detection**.
2. From the Actions menu in the upper left corner click **Disable Async Monitoring**.



To Disable all Asynchronous Monitoring

Set the App 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
```

or

Add the following line under the fork-config section of the app-agent-config.xml file.

```
<exclude filter-type="REGEX" filter-value=".*"/>
```

To Enable Asynchronous Monitoring

1. In the left navigation pane click **Configure->Instrumentation->Transaction Detection**.
2. From the Actions menu in the upper-left corner click **Enable Async Monitoring**.



Learn More

- [Configure Business Transaction Detection](#)
- [App Agent Node Properties](#)

Configure End-to-End Message Transactions for Java

- [About End-to-End Message Monitoring](#)
- [Configure End-to-End Performance Monitoring](#)
 - [To enable end-to-end message transaction monitoring](#)
- [End-to-End Performance Metrics](#)
- [Learn More](#)

About End-to-End Message Monitoring

As described in [Configure Multi-Threaded Transactions for Java](#), AppDynamics can automatically detect and monitor asynchronous threads spawned in Java and .NET applications as part of a business transaction. The request response times for such business transactions reflect the length of time from when the entry point thread receives the request until it responds to the client. In a highly distributed, asynchronous application, however, this response time doesn't always reflect the actual amount time it takes to fully process a request.

For example, consider an asynchronous application with an entry point method in a request handler that spawns multiple threads, including one to serve as the final response handler. The request handler thread then returns a preliminary response to the client, stopping the clock for purposes of measuring the response time of the 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 time it takes for the complete, logical transaction that the AppDynamics user wants to monitor does not match the reported response time metric for the business transaction.

End-to-end metrics give you a way to measure the processing time for business transactions for which response time alone does not reflect the entire request processing workflow. The end-to-end message metrics show how long it takes to process the end-to-end transactions, the number of end-to-end transactions per minute, and the number of slow end-to-end message processing events.

Configure End-to-End Performance Monitoring

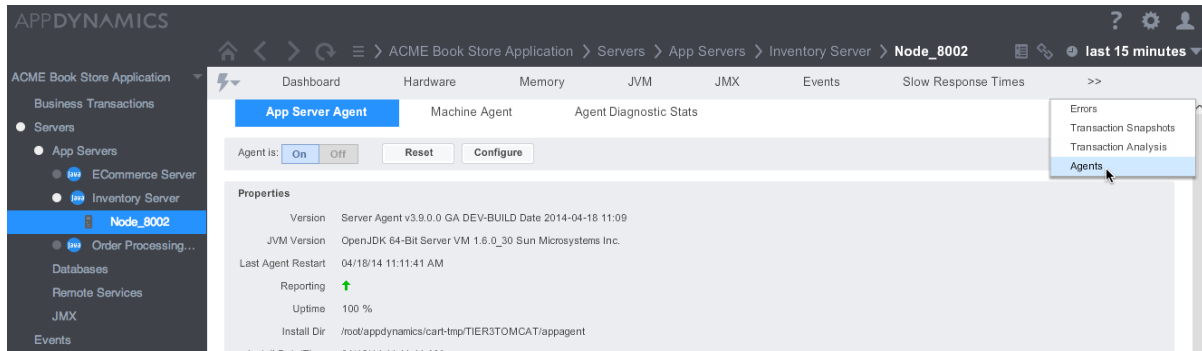
To use end-to-end message monitoring, you need to identify a method in your application that acts as the logical end point in the transaction processing sequence. For our sample response handler, this could be the method in the response handler that waits for threads to complete, assembles the response and sends it to the client.

To enable end-to-end message monitoring in AppDynamics:

- Make sure that the threads in the logical transaction processing flow are traced by AppDynamics, including the thread that contains the end-point method. If needed, configure custom correlation to ensure that all threads are properly traced. For more information, see in [Configure Multi-Threaded Transactions for Java](#).
- Configure the end-to-end message settings in the App Agent for the node that contains the business transaction entry point, as described here

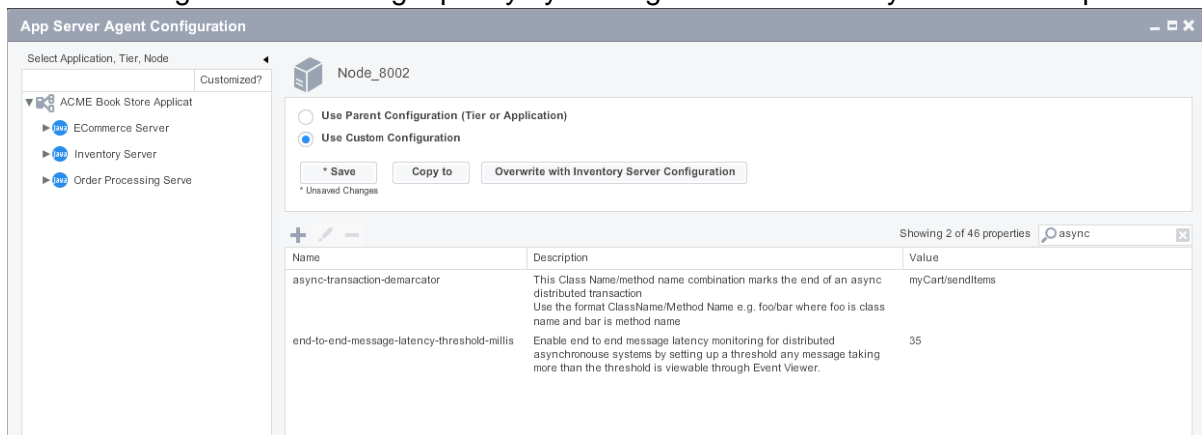
To enable end-to-end message transaction monitoring

1. In the application navigation tree, navigate to the node that contains the entry point for the messages for which you want to measure end-to-end performance.
2. Click the **Agents** tab.
If the Agents item is not visible in the tab bar, expand the tab bar, as shown:



3. Click the **Configure** button.
4. Click **Use Custom Configuration**.
5. Set values for the following agent properties by double-clicking the property and entering new values for each:
 - a. **async-transaction-demarcator**: Specifies the class name and name for the method that serves as the termination point for the end-to-end transaction. It should be in the format: `class_name/method_name`
 - b. **end-to-end-message-latency-threshold-millis**: Optionally, set a value in milliseconds that, if exceeded by the time it takes to process an end-to-end message, causes the transaction to be considered a slow end-to-end message and sends an event to the Controller.

You can navigate to the settings quickly by filtering the view with "async". For example:



6. Click **Save** to apply your changes.

End-to-end latency metrics should now appear for any business transactions that has an entry point on the configured node and invokes the transaction demarcator method. Values for end-to-end message transaction performance should also appear in the metric browser view for the node and for the overall application.

End-to-End Performance Metrics

The overall application performance metrics for end-to-end message transactions are:

- **Average End to End Latency**: The average time in milliseconds spent processing end-to-end message transactions over the selected time frame.
- **End to End Messages per Minute**: The average number of transactions that are measured as end-to-end message transactions per minute over the selected time frame.

- **Number of Slow End to End Messages:** The number of end-to-end message transactions that exceeded the configured end-to-end latency threshold over the selected time frame.

For information on how to accessing the overall application performance metrics, see [Metric Browser](#).

Learn More

- [Measure Distributed Transaction Performance](#)
- [Configure Multi-Threaded Transactions for Java](#)

Configure Backend Detection for Java

- [Types of Exit Points](#)
- [View the Discovery Rules](#)
- [Revise Backend Discovery Rules](#)
 - [Change the Default Discovery Rules](#)
 - [Add Backend Discovery Rules](#)
 - [Add Custom Exit Points](#)
- [Propagate Changes to Other Tiers or Applications](#)
- [Learn More](#)

To review general information about monitoring databases and remote services (collectively known as backends) and for an overview of backend configuration see [Backend Monitoring](#).

Types of Exit Points

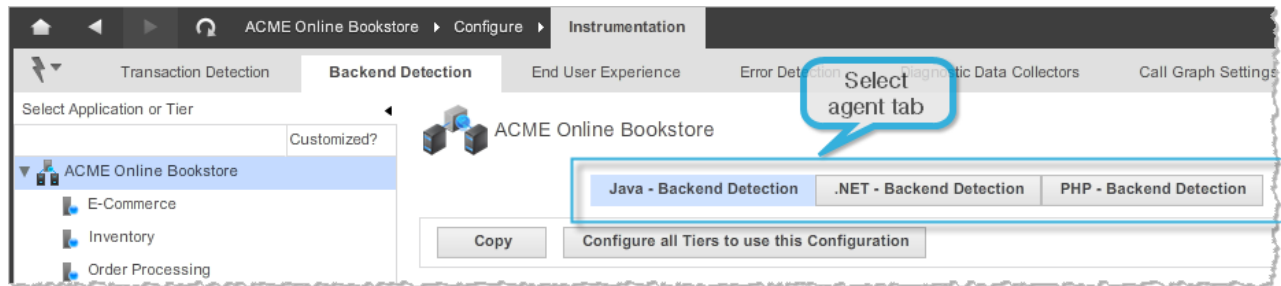
Each automatically discovered backend type has a default discovery rule and a set of configurable properties. See the following:

- [Configure Custom Exit Points for Java](#)
- [Configurations for Custom Exit Points for Java](#)
- [HTTP Exit Points for Java](#)
- [JDBC Exit Points for Java](#)
- [Message Queue Exit Points for Java](#)
- [Web Services Exit Points for Java](#)
- [Cassandra Exit Points for Java](#)
- [RMI Exit Points for Java](#)
- [Thrift Exit Points for Java](#)

View the Discovery Rules

To view the discovery rules for an automatically discovered backend, access the backend configuration window using these steps:

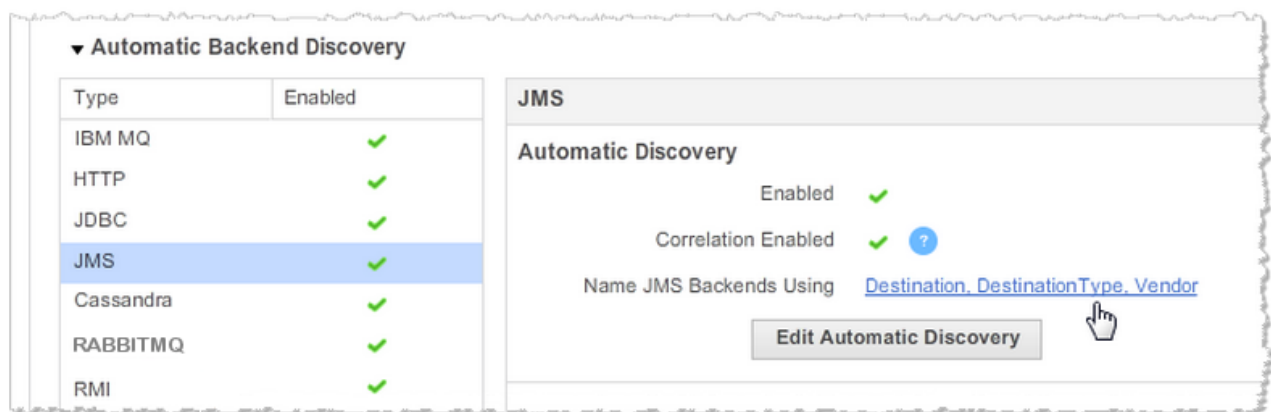
1. Select the application.
2. In the left navigation pane, click **Configure -> Instrumentation**.
3. Select the **Backend Detection** tab.
4. Select the application and the tab corresponding to your agent platform (Java, .NET, PHP).



The Automatic Backend Discovery default configurations for that agent are listed.

5. In the Automatic Backend Discovery list, click the backend type to view.

A summary of the configuration appears on the right. For example, the following figure shows that JMS backends are auto-discovered using the Destination, Destination Type, and Vendor.



Revise Backend Discovery Rules

If the default settings don't give you exactly what you need, you can refine the configuration in the following ways:

- [Change the default discovery rules:](#)
 - Enable or disable one or more of the properties
 - Use one or more specified segments of a property
 - Run a regular expression on a property
 - Execute a method on a property
- [Add new discovery rules](#)
- [Add custom exit points](#)

The precise configurations vary according to the backend type. These general features are configurable:

- **Discovery Enabled** - You can enable and disable automatic discovery for the backend type. Undiscovered backends are not monitored.
- **Correlation Enabled** - You can enable and disable correlation. Correlation enables AppDynamics to tag, trace, and learn about application calls to and through the backend to other remote services or tiers. For example, if a call is made from Tier1 -> Backend1 -> Tier2, Tier2 knows about the transaction flow because the agent "tags" the outbound call

from Backend1 to identify it as related to the same transaction that called Backend1 from Tier1. If you do not care about activity downstream from the backend, you may want to disable correlation.

- Backend Naming - You can configure how backends are named.

Change the Default Discovery Rules

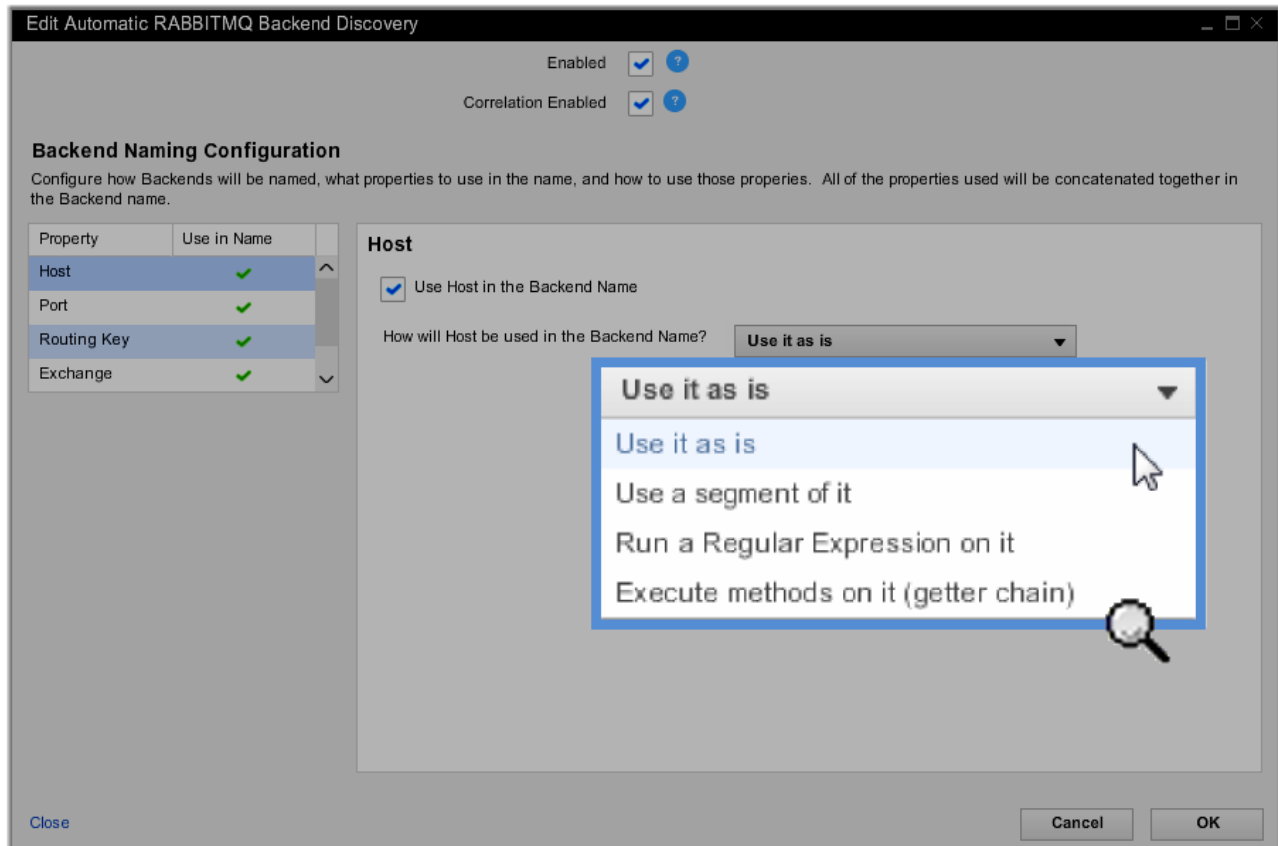
When you need to revise the default set of discovery rules, in many cases, you can achieve the visibility you need by making adjustments to the default automatic discovery rules. For some scenarios, you might want to disable some or all of the default rules and create custom rules for detecting all your backends. AppDynamics provides flexibility for configuring backend detection.

For example, detection of HTTP backends is enabled by default. In Java environments, HTTP backends are identified by the host and port and correlation with the application is enabled. To change the discovery rule for HTTP backends in some way, such as disabling correlation, omitting a property from the detected name, or using only certain segments of a property in the name, you edit the HTTP automatic discovery rule.

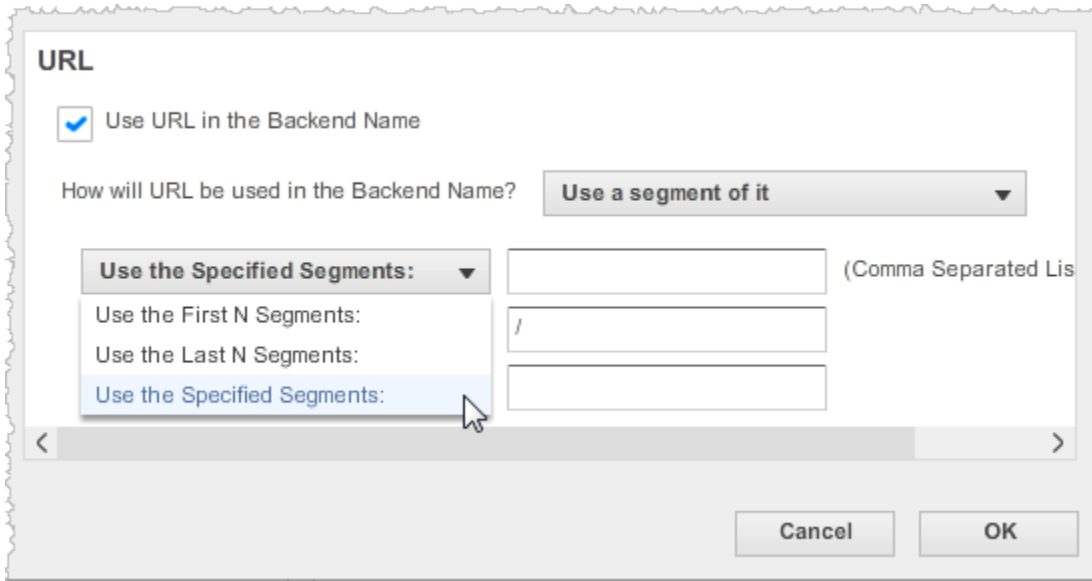
Review the rules for each exit point to determine the best course of action for your application if the default discovery rules do not give the results you need for your monitoring strategy.

To change default backend automatic discovery rules

1. From the left-hand navigation panel, select **Configure -> Instrumentation**. Then select the **Backend Detection** tab and the application or tier you want to configure.
2. In the Automatic Backend Discovery list, select the backend type to modify. The rule summary appears in the Automatic Discovery panel on the right.
3. Click **Edit Automatic Discovery**. The Edit Automatic Backend Discovery Rule window appears.
4. For each property that you want to configure:
 - Select the property in the property list.
 - Check the property check box to use the property for detection; clear the check box to omit it.
 - If you are using the property, choose how the property is used from the drop-down list.



- If you have a complex property, such as the URL, destination, or a query string, and you want to eliminate some parts of it or need some additional manipulation you can use an option from the second drop-down list such as **Run a Regular Expression on it** or **Execute methods on it (getter chain)**. Each option has associated configuration parameters. For example, you have options for manipulating the segments of the **URL**.



5. Check **Enabled** to enable the rule; clear the check box to disable it.

6. Check **Correlation Enabled** to enable correlation.
7. Click **OK**.

Add Backend Discovery Rules

AppDynamics provides the additional flexibility to create new custom discovery rules for the automatically discovered backend types. Custom rules include the following settings:

- **Name** for the custom rule.
- **Priority** used to set precedence for custom rules.
- **Match Conditions** used to identify which backends are subject to the custom naming rules.
- **Backend Naming Configuration** used to name the backends matching the match conditions.

The window for adding custom discovery rules looks like this:

Create Custom HTTP Backend Discovery Rule

Name:

Enabled: ☒ ?

Correlation Enabled: ☒ ?

Priority: ?

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.

<input type="checkbox"/>	Host	Equals	<input type="text"/>	
<input type="checkbox"/>	Port	Equals	<input type="text"/>	
<input type="checkbox"/>	URL	Equals	<input type="text"/>	
<input type="checkbox"/>	Query String	Equals	<input type="text"/>	

Backend Naming Configuration

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.

Property	Use in Name
Host	<input checked="" type="checkbox"/>
Port	<input checked="" type="checkbox"/>
URL	<input type="checkbox"/>
Query String	<input type="checkbox"/>

Host

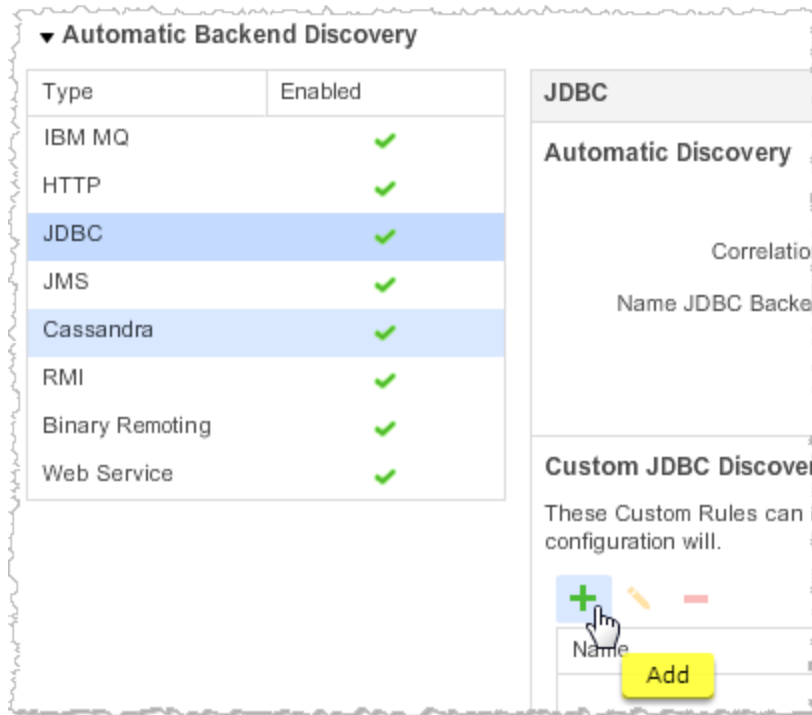
☒ Use Host in the Backend Name

How will Host be used in the Backend Name?

Close Cancel OK

To create a custom discovery rule for an automatically discovered backend type, use these steps:

1. In the Automatic Backend Discovery list, select the backend type. The Custom Discovery Rules editor appears in the right panel below the Automatic Discovery panel.
2. Click **Add** (the + icon) to create a new rule or select an existing rule from the list and click the edit icon to modify one.



3. Enter a name for the custom rule.
4. Confirm the settings for **Enabled** and **Correlation Enabled** (if applicable).
4. Enter the priority for the 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.
5. In the next section, configure the match conditions.
Match conditions are used to identify which backends should use the custom rule. Backends that do not meet all the defined match conditions are discovered according to the default rule.
7. In the next section, configure the naming for the backends matching the rule. The naming configuration must include the property used by the match condition.
8. Save the configuration.
See specific exit points for examples.

Add Custom Exit Points

When your application has backends that are not automatically discovered, you can enable discovery using custom exit points. To do this, you need to know the class and method used to identify the backend. See [Configure Custom Exit Points for Java](#).

Propagate Changes to Other Tiers or Applications

When you have made changes to the backend detection rules, you may want to propagate your changes to other tiers or applications.

To copy an entire backend detection configuration to all tiers

1. Access the backend detection window. See [View the Discovery Rules](#).
2. In the left panel select the application or tier whose configuration you want to copy.
3. Click **Configure all Tiers to use this Configuration**.

To copy an entire backend detection configuration to another tier or application

1. Access the backend detection window. See [View the Discovery Rules](#).
2. In the left panel select the application or tier whose configuration you want to copy.
3. Click **Copy**.
4. In the Application/Tier browser, choose the application or tier to copy the configuration to.
5. Click **OK**.

Learn More

- [Backend Monitoring](#)
- [Monitor Databases](#)
- [Monitor Remote Services](#)
- [Hierarchical Configuration Model](#)
- [Configure Custom Exit Points](#)
- [Configurations for Custom Exit Points for Java](#)

Configure Custom Exit Points for Java

- [Default Backends Discovered by the App Agent for Java](#)
- [Configure Custom Exit Points for Java Backends](#)
 - [To create a custom exit point](#)
 - [To split an exit point](#)
 - [To group an exit point](#)
 - [To define custom metrics for a custom exit point](#)
 - [To define transaction snapshot data collected](#)
- [Learn More](#)

AppDynamics provides default automatic discovery for commonly-used backends. If a backend used in your environment is not discovered, first compare the list of default backends to determine whether you need to modify the default configuration. If it is not on the list then configure a custom exit point according to these instructions.

Default Backends Discovered by the App Agent for Java

The default backends for Java are:

- HTTP
- JDBC
- JMS
- Cassandra
- IBM MQ
- RABBITMQ
- RMI
- Thrift

- Web Service

To configure a default backend see [Configure Backend Detection for Java](#).

Configure Custom Exit Points for Java Backends

Configure Custom Exit Points

Use custom exit points to identify backend types that are not automatically detected, such as file systems, mainframes, and so on. For example, you can define a custom exit point to monitor the 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. You define a custom exit point by specifying the class and method used to identify the backend. If the method is overloaded, you need to add the parameters to identify the method uniquely.

You can restrict the method invocations for which you want AppDynamics to collect metrics by specifying match conditions for the method. The match conditions can be based on a parameter or the invoked object.

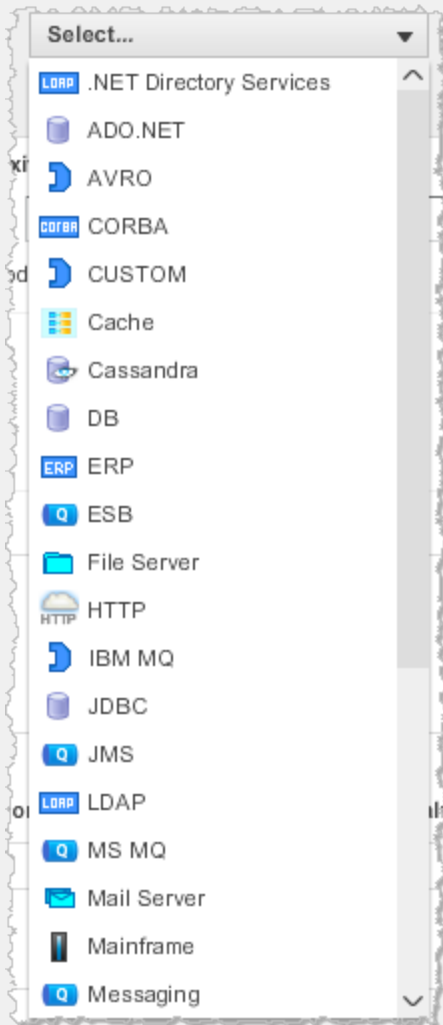
You can also optionally split the exit point based on a method parameter, the return value, or the invoked object.

You can also configure custom metrics and transaction snapshot data to collect for the backend.

To create a custom exit point

1. From the left navigation pane, click **Configure -> Instrumentation** and select the **Backend Detection** tab.
2. Select the application or tier for which you are configuring the custom exit point.
3. Ensure **Use Custom Configuration for this Tier** is selected.
Backend detection configuration is applied on a hierarchical inheritance model. See [Hierarchical Configuration Model](#).
4. Scroll down to **Custom Exit Points** and click **Add** (the + icon).
5. In the **Create Custom Exit Point** window, click the **Identification** tab if it is not selected.
6. Enter a name for the exit point. This is the name that identifies the backend.
7. Select the type of backend from the **Type** drop-down menu.

This field controls the icon and name that appears on the flow maps and dashboards. Some of the values are shown in this screen shot:



If the type is not listed, you can check **Use Custom** and enter a string to be used as the name on the dashboards.

8. Configure the class and method name that identify the custom exit point.

If the method is overloaded, check the Overloaded check box and add the parameters.

9. If you want to restrict metric collection based on a set of criteria that are evaluated at runtime, click **Add Match Condition** and define the match condition(s).

For example, you may want to collect data only if the value of a specific method parameter contains a certain value.

10. Click **Save**.

The following screenshot shows a custom exit point for a Cache type backend. This exit point is defined on the `getAll()` method of the specified class. The exit point appears in flow maps as an unresolved backend named `CoherenceGetAll`.

Create Custom Exit Point

Name: Type: ☐ Use custom

Identification Custom Metrics Snapshot Data

Define the class and method name which, when called, will be identified as a Custom Exit Point:

Class: equals

Method Name: ☐ Is this Method Overloaded?

Method Parameters (optional):

Add Parameter

Match Conditions (optional):

Add Match Condition

Calls to the specified class and method name can be further split based by a combination of method parameters and return value:

Name	Description
<input type="text"/>	<input type="text"/>

Add Edit Delete

Cancel Save

To split an exit point

1. In the Backend Detection configuration window, click **Add**.
2. Enter a display name for the split exit point.
3. Specify the source of the data (parameter, return value, or invoked object).
4. Specify the operation to invoke on the source of the data: **Use toString()** or **Use Getter Chain** (for complex objects).
5. Click **Save**.

The following example shows a split configuration of the previously created CoherenceGetAll exit point based on the getCacheName() method of the invoked object.

Edit Custom Exit Point Identifier

Specify the parameter index or indicate if it the return value of the diagnostic data to be collected. Simple getters without parameters can be used on the paramter or the return value to be displayed against the display name spe here.

Display Name

Create your own name for the data collected. This will be the display name for the data in Request Snapshot

Collect Data From

- ☐ Method Parameter @ Index:
- ☐ Return Value
- ☒ Invoked Object

Operation on Invoked Object

- ☐ Use toString()
- ☒ Use Getter Chain
for example: getAccount().getBalance()

To group an exit point

You can group methods as a single exit point if the methods point to the same key.

For example, ACME Online has an exit point for NamedCache.getAll. This exit point has a split configuration of getCacheName() on the invoked object as illustrated in the previous screen shot.

Suppose we also define an exit point for NamedCache.entrySet. This is another exit point, but it has the split configuration that has getCacheName() method of the invoked object.

Edit Custom Exit Point

Name: Type:

Identification Custom Metrics Snapshot Data

Define the class and method name which, when called, will be identified as a Custom Exit Point:

Class equals

Method Name ☐ Is this Method Overloaded?

Method Parameters (optional)

Match Conditions (optional)

Calls to the specified class and method name can be further split based by a combination of method parameters and

Name	Description
CacheName	Collect data from the invoked object and capture the result of getCacheName().

If the `getAll()` and the `entrySet()` methods point to the same cache name, they will point to the same backend.

Matching name-value pairs identify the back-end. In this case, only one key, the cache name, has to match. So, here both exit points have the same name for the cache and they resolve to the same backend.

To define custom metrics for a custom exit point

Custom metrics are collected in addition to the standard metrics.

The result of the data collected from the method invocation must be an integer value, which is either averaged or added per minute, depending on your data roll-up selection.

To configure custom business metrics that can be generated from the Java method invocation:

1. Click the **Custom Metrics** tab.
2. Click **Add**.
3. In the **Add Custom Metric** window type a name for the metric.
4. Select **Collect Data From** to specify the source of the metric data.

5. Select **Operation on Method Parameter** to specify how the metric data is processed.
6. Select how the data should be rolled up (average or sum) from the Data Rollup drop-down menu.
7. Click **Create Custom Metric**.

To define transaction snapshot data collected

1. Click the **Snapshot Data** tab.
2. Click **Add**.
3. In the Add Snapshot Data window, enter a display name for the snapshot data.
4. Select the Collect Data From radio button to specify the source of the snapshot data.
5. Select the Operation on Method Parameter to specify how the snapshot data is processed.
5. Click **Save**.

Learn More

- [Configurations for Custom Exit Points for Java](#)

Configurations for Custom Exit Points for Java

- [Caching System Backends](#)
 - [Coherence Exit Points](#)
 - [Sample Coherence Exit Point Configuration](#)
 - [Memcached Exit Points](#)
 - [Sample Memcached Exit Point Configuration](#)
 - [EhCache Exit Points](#)
 - [Sample EhCache Exit Point Configuration](#)
- [SAP Exit Points](#)
 - [Sample SAP Exit Point Configuration](#)
- [Mail Exit Points](#)
 - [Sample Mail Exit Point Configuration](#)
- [LDAP Exit Points](#)
 - [Sample LDAP Exit Point Configuration](#)
- [MongoDB Exit Points](#)
 - [Sample MongoDB Exit Point Configuration](#)
- [Learn More](#)

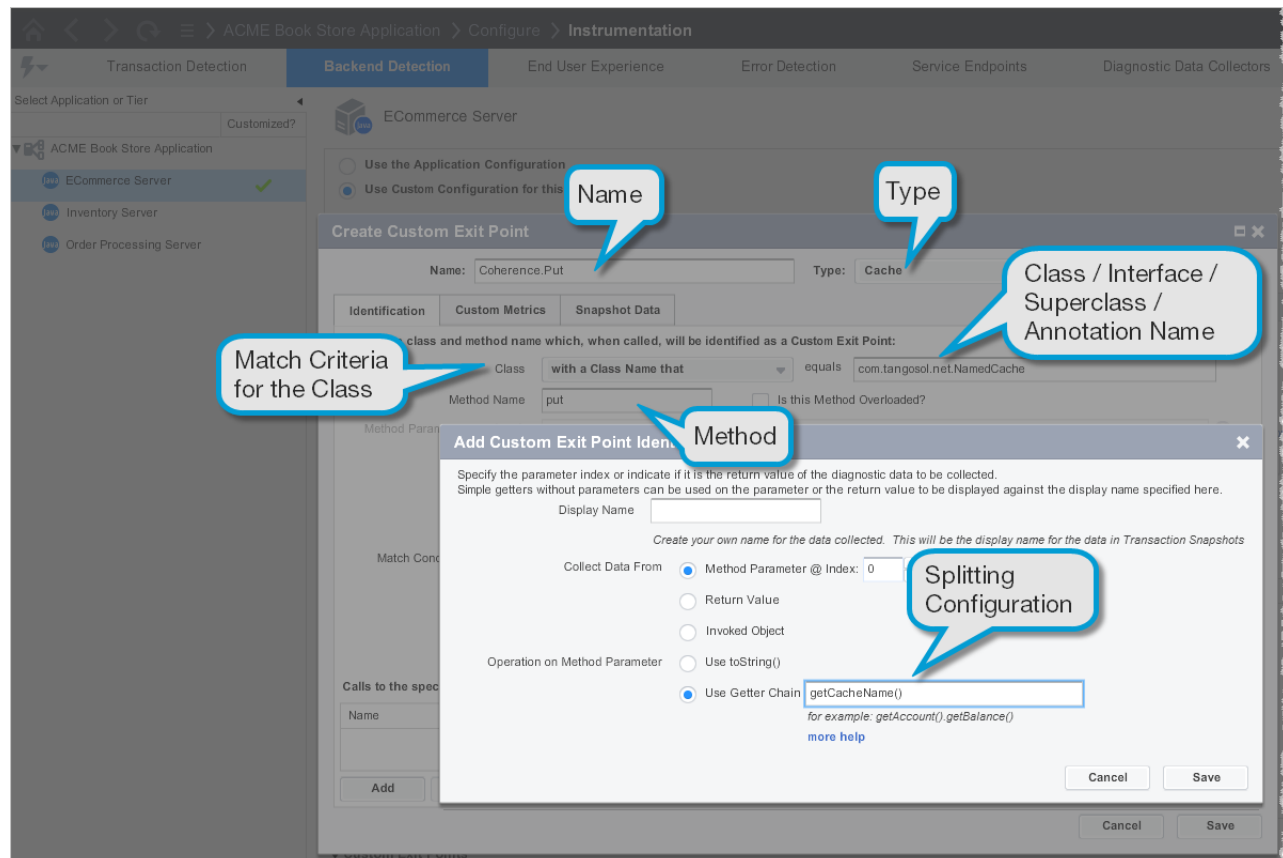
This topic describes custom exit point configurations for specific backends in Java environments. To implement these exit points, create custom exit points using the configuration described. To learn how to create custom exit points, see [To create a custom exit point](#).

Caching System Backends

Coherence Exit Points

Name of the Exit Point	Type	Method Name	Match Criteria for the Class	Class/Interface/Superclass/Annotation Name	Splitting Configuration
Coherence.Put	Cache	put	that implements an interface which	com.tangosol.net.NamedCache	getCacheName()
Coherence.PutAll	Cache	putAll	that implements an interface which	com.tangosol.net.NamedCache	getCacheName()
Coherence.EntrySet	Cache	entrySet	that implements an interface which	com.tangosol.net.NamedCache	getCacheName()
Coherence.KeySet	Cache	keySet	that implements an interface which	com.tangosol.net.NamedCache	getCacheName()
Coherence.Get	Cache	get	that implements an interface which	com.tangosol.net.NamedCache	getCacheName()
Coherence.Remove	Cache	remove	that implements an interface which	com.tangosol.net.NamedCache	getCacheName()

Sample Coherence Exit Point Configuration



Memcached Exit Points

Name of the Exit Point	Type	Method Name	Match Criteria value for the Class	Class/Interface/ Superclass/Annotation Name
Memcached.Add	Cache	add	With a class name that	net.spy.memcached.MemcachedClient
Memcached.Set	Cache	set	With a class name that	net.spy.memcached.MemcachedClient
Memcached.Replace	Cache	replace	With a class name that	net.spy.memcached.MemcachedClient
Memcached.CompareAndSwap	Cache	cas	With a class name that	net.spy.memcached.MemcachedClient
Memcached.Get	Cache	get	With a class name that	net.spy.memcached.MemcachedClient

Memcached.Remove	Cache	remove	With a class name that	net.spy.memcached.MemcachedClient
------------------	-------	--------	------------------------	-----------------------------------

Sample Memcached Exit Point Configuration

The screenshot shows the 'Create Custom Exit Point' dialog in the AppDynamics Instrumentation configuration interface. The dialog is titled 'Create Custom Exit Point' and has tabs for 'Identification', 'Custom Metrics', and 'Snapshot Data'. The 'Identification' tab is selected. The dialog contains the following fields and callouts:

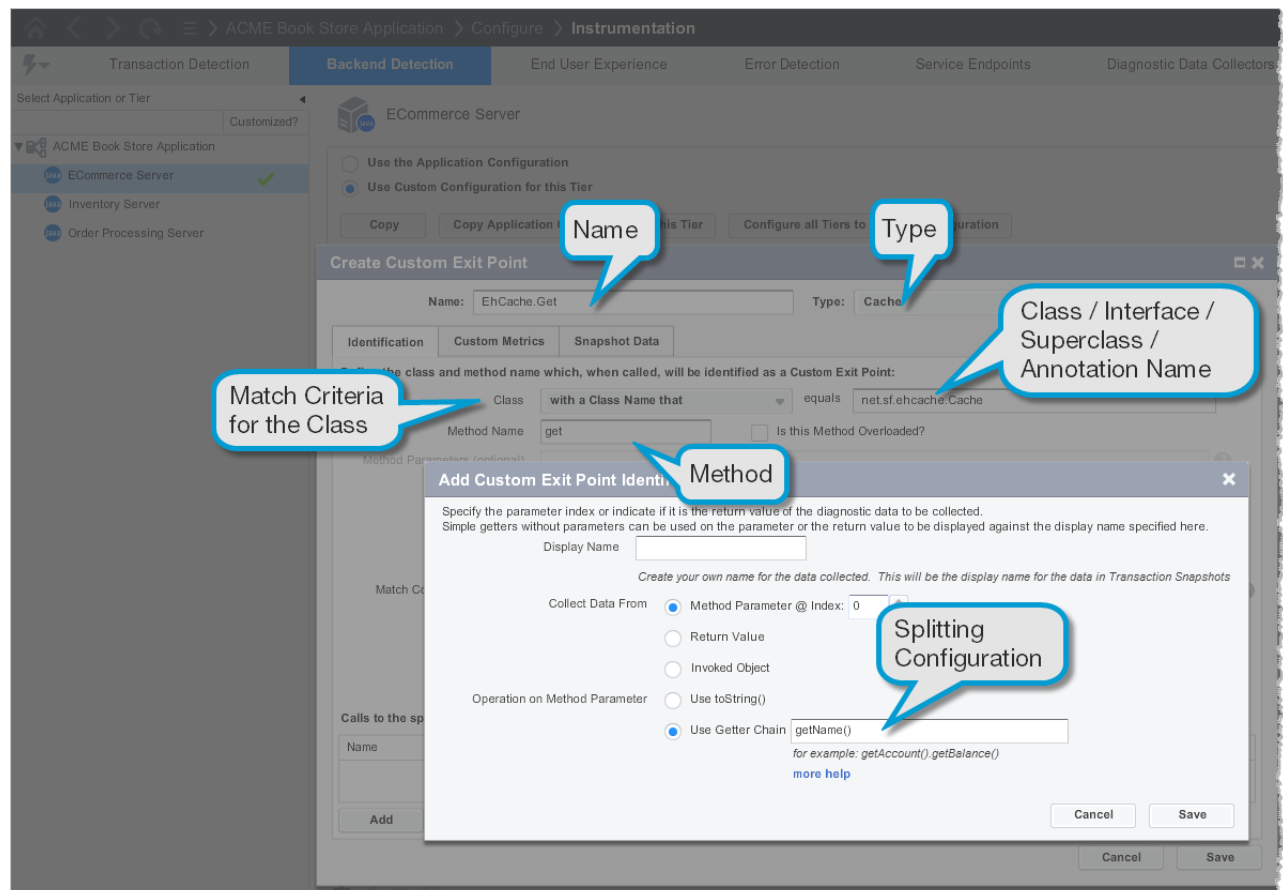
- Name:** Memcached.Add (Callout: Name)
- Type:** Cache (Callout: Type)
- Match Criteria for the Class:** with a Class Name that (Callout: Match Criteria for the Class)
- Method Name:** add (Callout: Method Name)
- Class / Interface / Superclass / Annotation Name:** net.spy.memcached.MemcachedClient (Callout: Class / Interface / Superclass / Annotation Name)
- Match Conditions (optional):** (Empty text area)
- Is this Method Overloaded?:** (Checkbox, unchecked)
- Buttons:** Add, Edit, Delete, Cancel, Save

EhCache Exit Points

Name of the Exit Point	Type	Method Name	Match Criteria value for the Class	Class/Interface/ Superclass/ Annotation Name	Splitting Configuration
EhCache.Get	Cache	get	With a class name that	net.sf.ehcache.Cache	getName()
EhCache.Put	Cache	put	With a class name that	net.sf.ehcache.Cache	getName()
EhCache.PutIfAbsent	Cache	putIfAbsent	With a class name that	net.sf.ehcache.Cache	getName()
EhCache.PutQuiet	Cache	putQuiet	With a class name that	net.sf.ehcache.Cache	getName()

EhCache.Remove	Cache	remove	With a class name that	net.sf.ehcache.Cache	getName()
EhCache.RemoveAll	Cache	removeAll	With a class name that	net.sf.ehcache.Cache	getName()
EhCache.RemoveQuiet	Cache	removeQuiet	With a class name that	net.sf.ehcache.Cache	getName()
EhCache.Replace	Cache	replace	With a class name that	net.sf.ehcache.Cache	getName()

Sample EhCache Exit Point Configuration



SAP Exit Points

Name of the Exit Point	Type	Method Name	Match Criteria value for the Class	Class/Interface/ Superclass/Annotation Name
SAP.Execute	SAP	execute	With a class name that	com.sap.mw.jco .rfc.MiddlewareRFC\$Client

SAP.Connect	SAP	connect	With a class name that	com.sap.mw.jco .rfc.MiddlewareR FC\$Client
SAP.Disconnect	SAP	disconnect	With a class name that	com.sap.mw.jco .rfc.MiddlewareR FC\$Client
SAP.Reset	SAP	reset	With a class name that	com.sap.mw.jco .rfc.MiddlewareR FC\$Client
SAP.CreateTID	SAP	createTID	With a class name that	com.sap.mw.jco .rfc.MiddlewareR FC\$Client
SAP.ConfirmTID	SAP	confirmTID	With a class name that	com.sap.mw.jco .rfc.MiddlewareR FC\$Client

Sample SAP Exit Point Configuration

The screenshot shows the 'Create Custom Exit Point' dialog box in the AppDynamics configuration interface. The dialog is titled 'Create Custom Exit Point' and has tabs for 'Identification', 'Custom Metrics', and 'Snapshot Data'. The 'Identification' tab is active. The 'Name' field is set to 'SAP.Execute' and the 'Type' is set to 'SAP'. The 'Class' field is set to 'with a Class Name that' and the 'Method Name' is set to 'execute'. The 'Match Criteria for the Class' is set to 'equals' and the 'Class / Interface / Superclass / Annotation Name' is set to 'com.sap.mw.jco.rfc.MiddlewareRFC\$Client'. The 'Method' field is set to 'execute'. The 'Match Conditions (optional)' field is empty. The 'Add Parameter' button is visible. The 'Add Match Condition' button is visible. The 'Calls to the specified class and method name can be further split based by a combination of method parameters and return value:' section is visible. The 'Add', 'Edit', and 'Delete' buttons are visible at the bottom. The 'Cancel' and 'Save' buttons are visible at the bottom right.

Annotations:

- Name:** Points to the 'Name' field containing 'SAP.Execute'.
- Type:** Points to the 'Type' field containing 'SAP'.
- Match Criteria for the Class:** Points to the 'Match Criteria for the Class' dropdown menu.
- Method:** Points to the 'Method Name' field containing 'execute'.
- Class / Interface / Superclass / Annotation Name:** Points to the 'Class' field containing 'com.sap.mw.jco.rfc.MiddlewareRFC\$Client'.

Mail Exit Points

Name of the Exit Point	Type	Method Name	Match Criteria value for the Class	Class/Interface/ Superclass/Annotation Name
MailExitPoint.Send	Mail Server	send	With a class name that	javax.mail.Transport
MailExitPoint.SendMessage	Mail Server	sendMessage	With a class name that	javax.mail.Transport

Sample Mail Exit Point Configuration

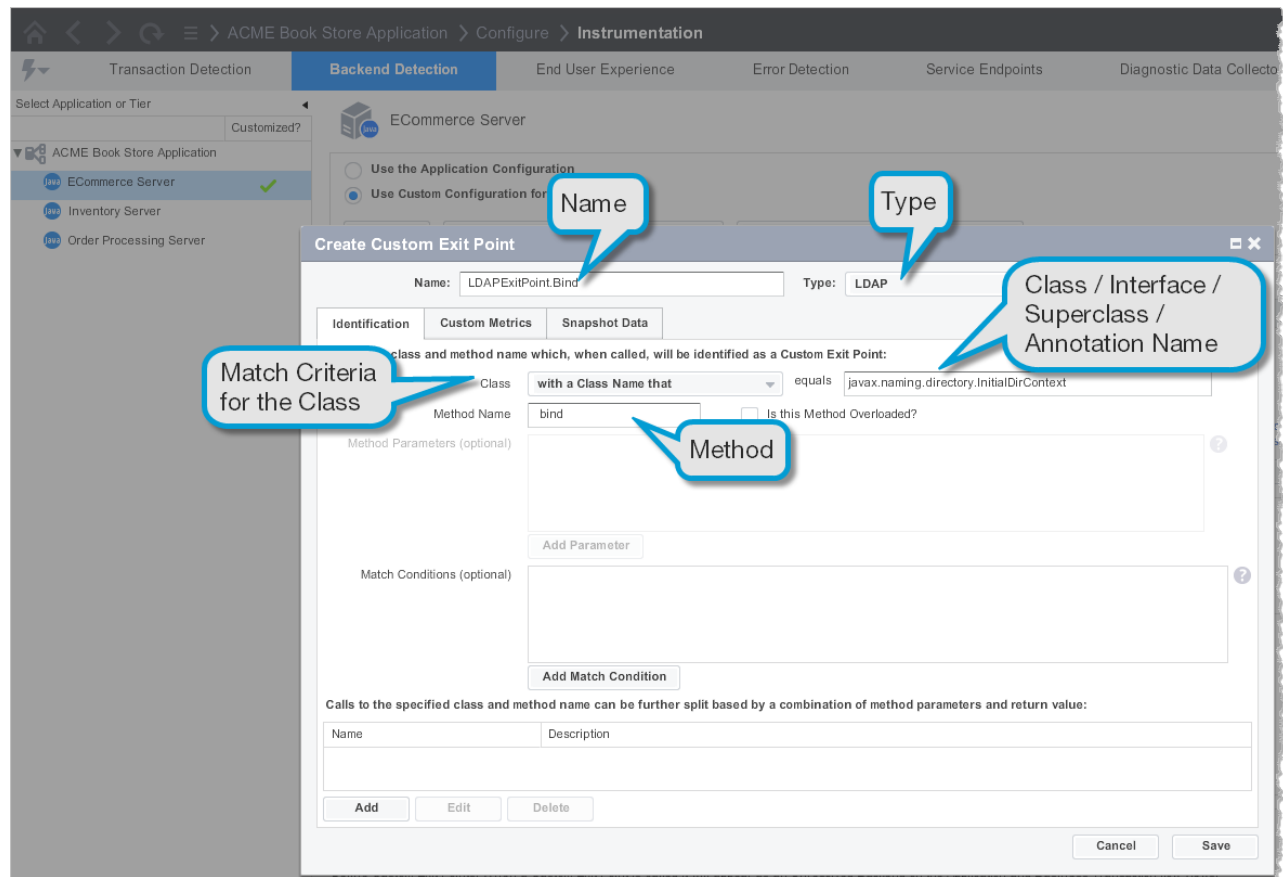
The screenshot shows the 'Create Custom Exit Point' dialog box in the AppDynamics configuration interface. The dialog is titled 'Create Custom Exit Point' and has tabs for 'Identification', 'Custom Metrics', and 'Snapshot Data'. The 'Identification' tab is active. It contains fields for 'Name' (MailExitPoint.Send), 'Type' (Mail Server), 'Class' (with a dropdown menu set to 'with a Class Name that'), 'Method Name' (send), and 'Match Criteria' (equals). The 'Class' field is highlighted with a blue callout bubble labeled 'Match Criteria for the Class'. The 'Method Name' field is highlighted with a blue callout bubble labeled 'Method'. The 'Type' field is highlighted with a blue callout bubble labeled 'Type'. The 'Class' field is also highlighted with a blue callout bubble labeled 'Class / Interface / Superclass / Annotation Name'. The 'Match Criteria' field is highlighted with a blue callout bubble labeled 'Match Criteria for the Class'. The 'Name' field is highlighted with a blue callout bubble labeled 'Name'. The 'Type' field is highlighted with a blue callout bubble labeled 'Type'. The 'Class' field is highlighted with a blue callout bubble labeled 'Class / Interface / Superclass / Annotation Name'. The 'Method Name' field is highlighted with a blue callout bubble labeled 'Method'. The 'Match Criteria' field is highlighted with a blue callout bubble labeled 'Match Criteria for the Class'. The 'Name' field is highlighted with a blue callout bubble labeled 'Name'. The 'Type' field is highlighted with a blue callout bubble labeled 'Type'. The 'Class' field is highlighted with a blue callout bubble labeled 'Class / Interface / Superclass / Annotation Name'. The 'Method Name' field is highlighted with a blue callout bubble labeled 'Method'. The 'Match Criteria' field is highlighted with a blue callout bubble labeled 'Match Criteria for the Class'.

LDAP Exit Points

Name of the Exit Point	Type	Method Name	Match Criteria value for the Class	Class/Interface/ Superclass/Annotation Name
LDAPExitPoint.Bind	LDAP	bind	With a class name that	javax.naming.directory.InitialDirContext
LDAPExitPoint.Rebind	LDAP	rebind	With a class name that	javax.naming.directory.InitialDirContext

LDAPExitPoint. Search	LDAP	search	With a class name that	javax.naming.dir ectory.InitialDirC ontext
LDAPExitPoint. ModifyAttributes	LDAP	modifyAttributes	With a class name that	javax.naming.dir ectory.InitialDirC ontext
LDAPExitPoint. GetNextBatch	LDAP	getNextBatch	With a class name that	com.sun.jndi.Ida p.LdapNamingE numeration
LDAPExitPoint. NextAux	LDAP	nextAux	With a class name that	com.sun.jndi.Ida p.LdapNamingE numeration
LDAPExitPoint. CreatePooledCo nnection	LDAP	createPooledCo nnection	With a class name that	com.sun.jndi.Ida p.LdapClientFact ory
LDAPExitPoint. Search	LDAP	search	With a class name that	com.sun.jndi.Ida p.LdapClientFact ory
LDAPExitPoint. Modify	LDAP	modify	With a class name that	com.sun.jndi.Ida p.LdapClientFact ory

Sample LDAP Exit Point Configuration

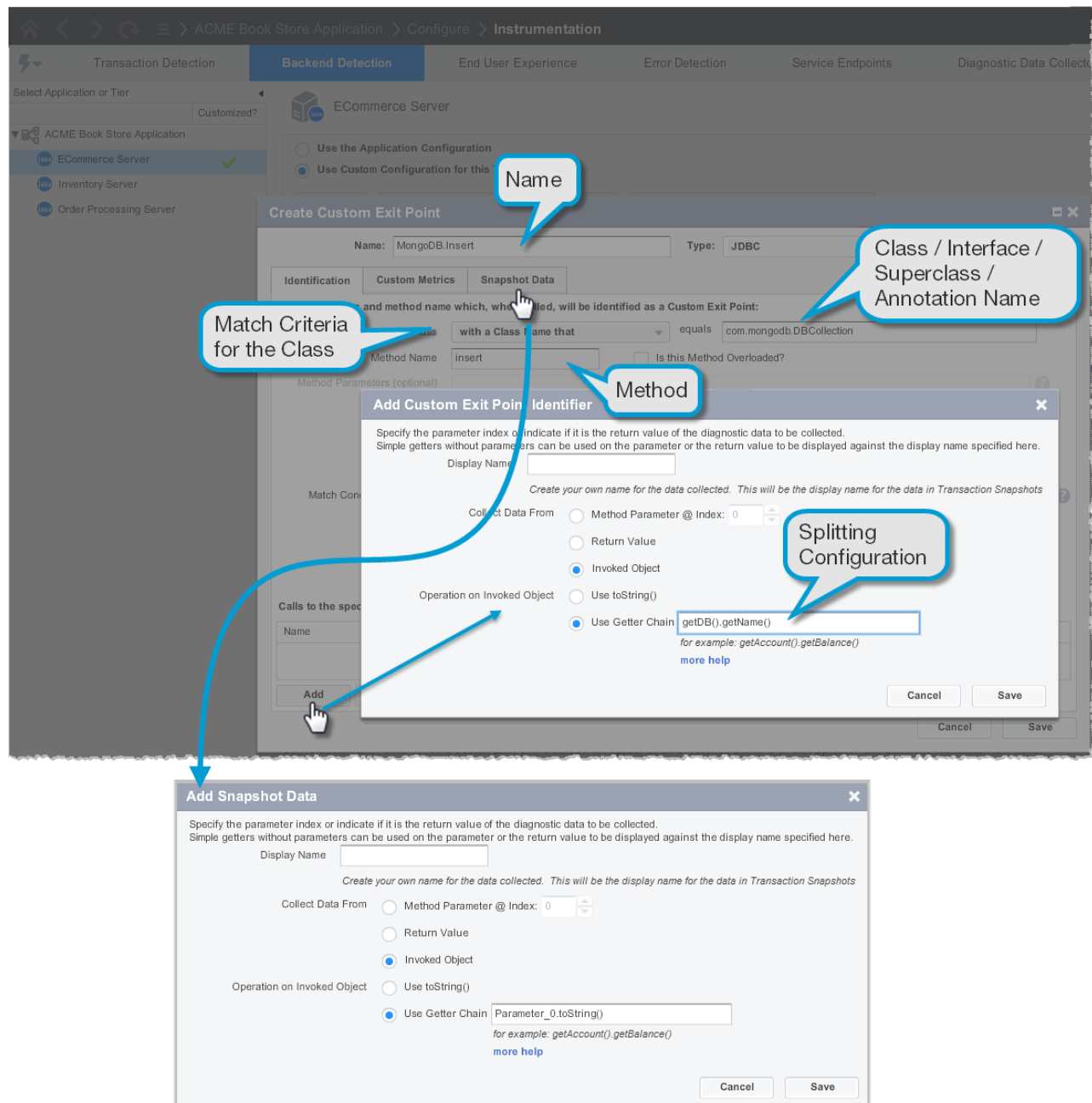


MongoDB Exit Points

Name of the Exit Point	Type	Method Name	Match Criteria value for the Class	Class/Interface/ Superclass/Annotation Name	Splitting configuration/ Custom Exit Point Identifier		Snapshot Data
					Collect Data From value for the Class	Operation on Invoked Object value for the Class	
MongoDB.Insert	JDBC	insert	With a class name that	com.mongodb.DB Collection	Invoked_ Object.	getDB().getName()	Parameter_0.toString()

MongoDB.Find	JDBC	find	With a class name that	com.mongodb.DB Collection	Invoked_Object.	getDB().getName()	Parameter_0.toString()
MongoDB.Update	JDBC	update	With a class name that	com.mongodb.DB Collection	Invoked_Object.	getDB().getName()	Parameter_0.toString()
MongoDB.Remove	JDBC	remove	With a class name that	com.mongodb.DB Collection	Invoked_Object.	getDB().getName()	Parameter_0.toString()
MongoDB.Apply	JDBC	apply	With a class name that	com.mongodb.DB Collection	Invoked_Object.	getDB().getName()	Parameter_0.toString()

Sample MongoDB Exit Point Configuration



Learn More

- [Configure Backend Detection \(Java\)](#)
- [Configure Custom Exit Points](#)

HTTP Exit Points for Java

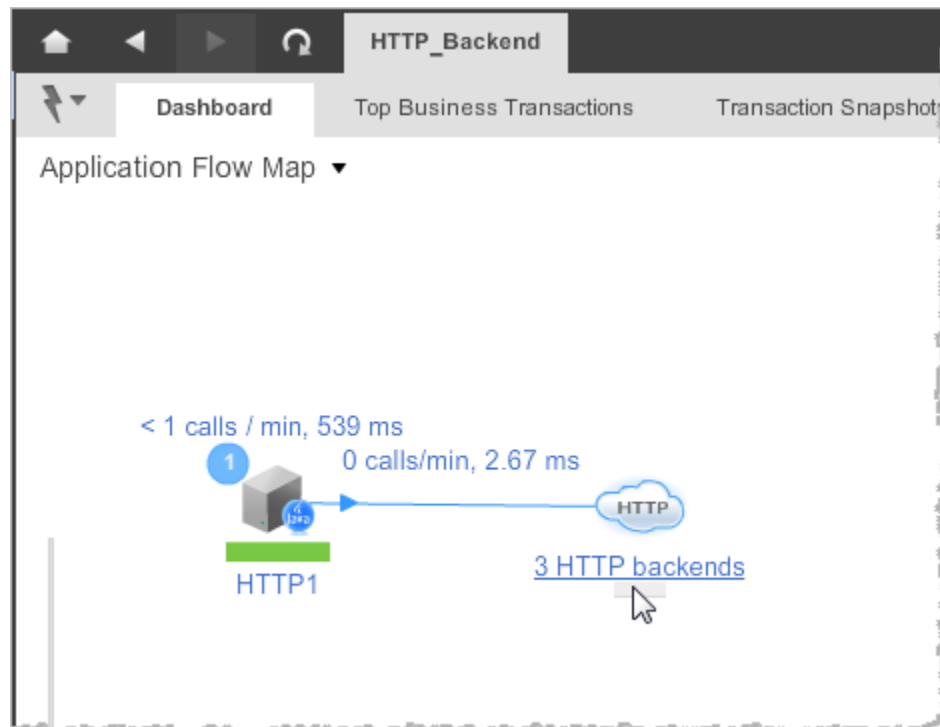
- [Automatic Discovery and Default Naming](#)
- [HTTP Configurable Properties](#)
- [Changing the Default HTTP Automatic Discovery and Naming](#)
 - [Examples](#)
 - [HTTP Service](#)

- [HTTP Backends With Different Formats](#)
- [Learn More](#)

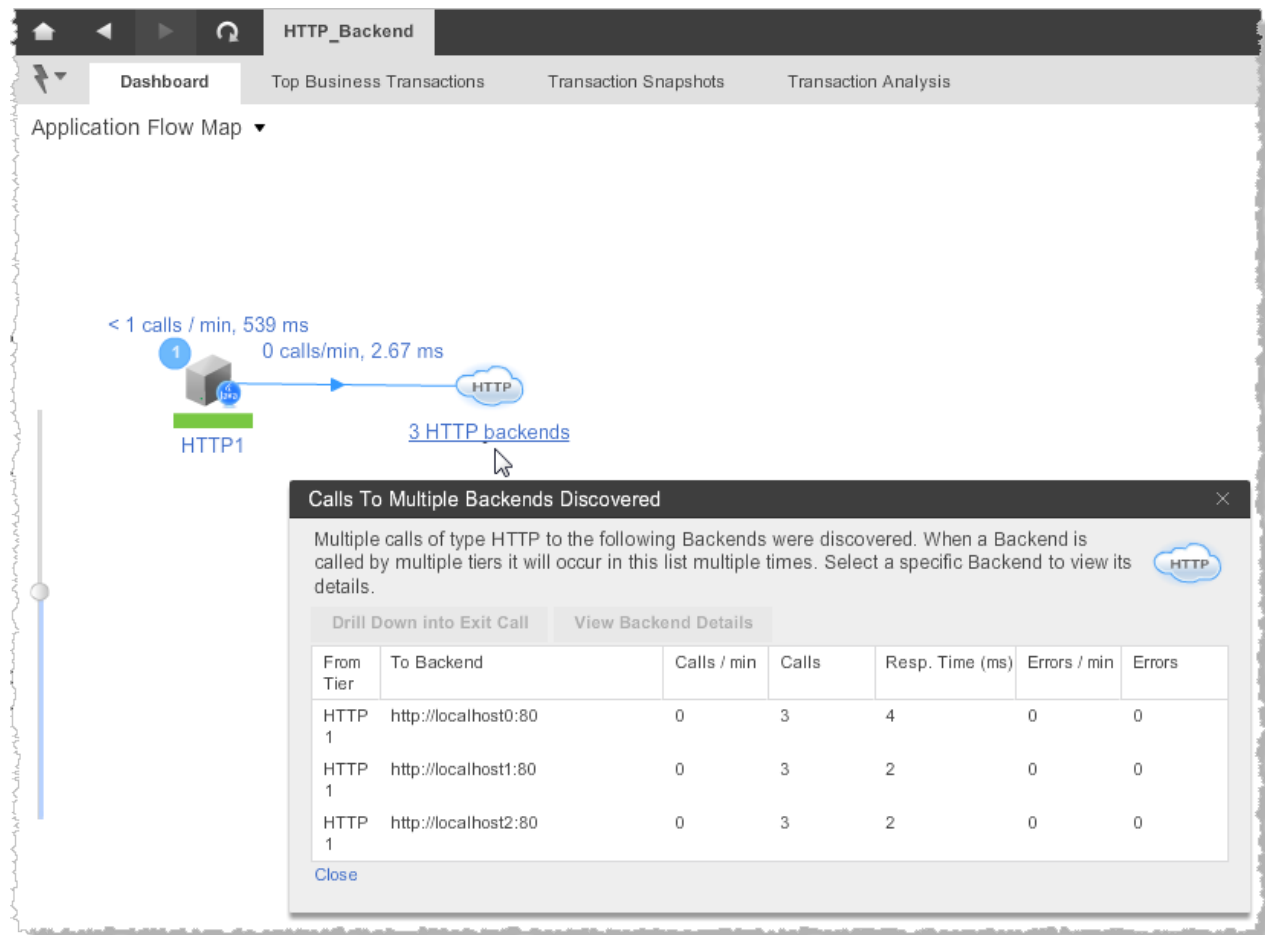
This topic explains HTTP exit point configuration. To review general information about monitoring databases and remote services (collectively known as backends) and for an overview of backend configuration see [Backend Monitoring](#). For configuration procedures, see [Configure Backend Detection \(Java\)](#).

Automatic Discovery and Default Naming

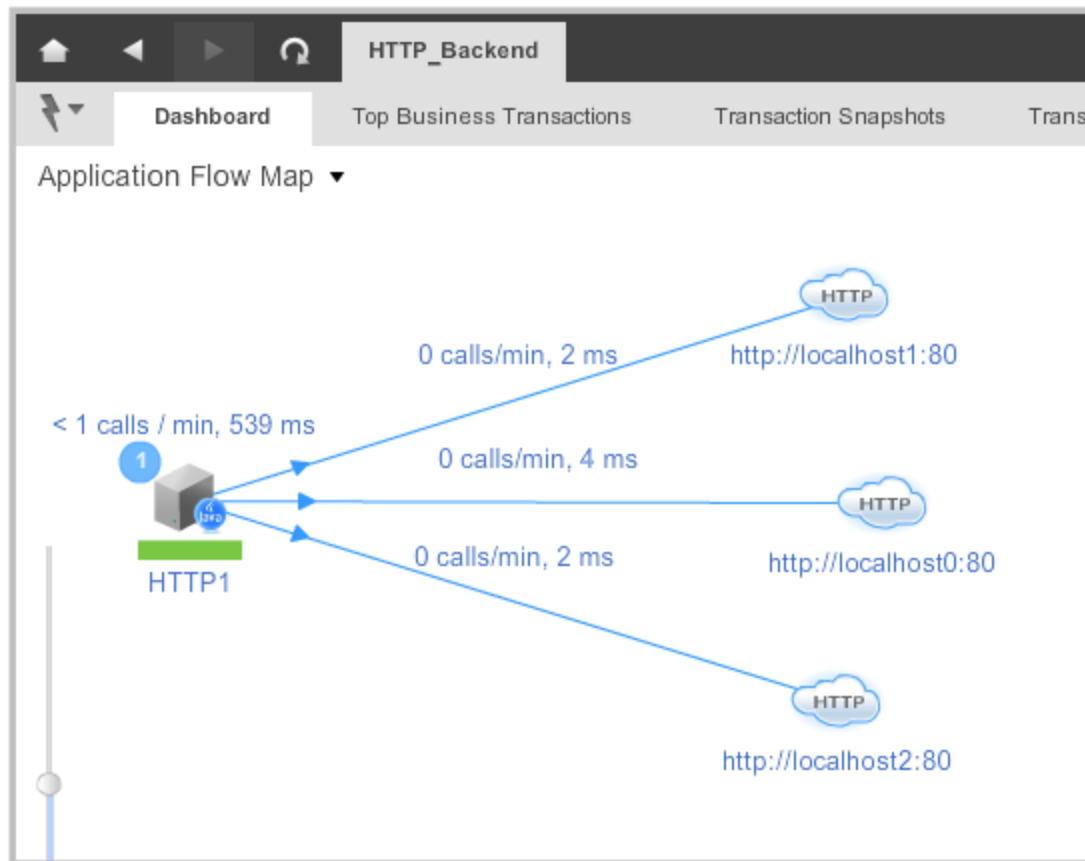
By default, AppDynamics automatically detects and identifies HTTP exit points (backends). HTTP exit point activity includes all HTTP calls done outside of a web service call. Web service calls are not considered an HTTP exit point. The Host and Port properties are enabled in the default HTTP automatic discovery rule. From the enabled properties AppDynamics derives a display name, for example: "myHTTPHost:5000". By default, AppDynamics groups HTTP backends together on the application flow map as shown:



Click the HTTP Backends label to see the individual backends listed.



If you [edit the flow map](#) to ungroup the HTTP backends, you can see each separate HTTP backend and how it uses the default discovery rule for naming.



HTTP Configurable Properties

You can enable or disable the use of the following properties for HTTP exit points.

Configurable Properties	Used by Default in Detection and Naming
Host	Yes
Port	Yes
URL	No
Query String	No

For procedures, see [Configure Backend Detection \(Java\)](#)

Changing the Default HTTP Automatic Discovery and Naming

Depending on exactly what you need to monitor, there may be times when you want to change the default HTTP configuration. When you see things such as EC2 host names, file paths, and ports in the backend name, changing the default discovery rule may help. 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. Doing this enables you to monitor the key performance indicators (KPIs) that are of most interest to you.

Examples

HTTP Service

Consider a scenario 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 be of no use since the IP addresses are transient and all performance numbers would be irrelevant after the IP addresses recycle. Therefore, you want to monitor by service name. To do this you need to avoid using the Host and Port properties in your configuration and use only the URL property.

1. Edit the **Automatic Backend Discovery** rule for HTTP. See [Configure Backend Detection \(Java\)](#) for details on accessing this screen.

Select Application or Tier

Customized?

ACME Book Store Application

Java - Backend Detection .NET - Backend Detection PHP - Backend Detection

Copy Configure all Tiers to use this Configuration

▼ Automatic Backend Discovery

Type	Enabled
IBM MQ	✓
HTTP	✓
JDBC	✓
JMS	✓
Cassandra	✓
RMI	✓
Binary Remoting	✓
Web Service	✓

HTTP

Automatic Discovery

Enabled ✓

Correlation Enabled ✓

Name HTTP Backends Using Host, Port

Edit Automatic Discovery

Custom HTTP Discovery Rules

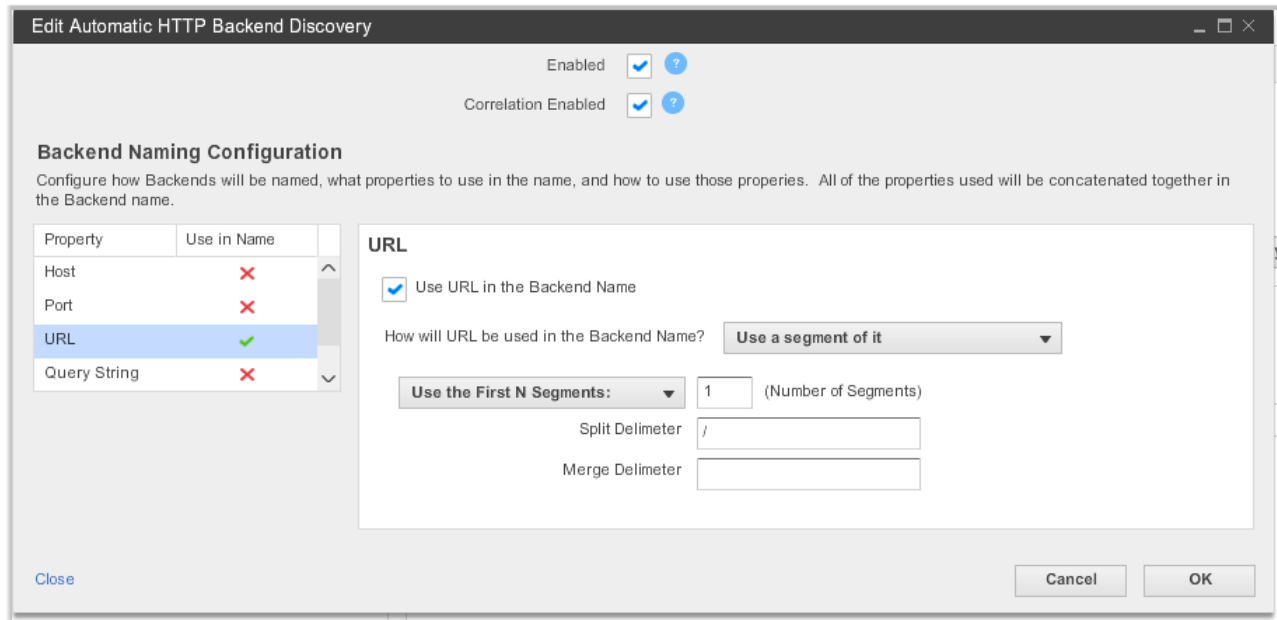
These Custom Rules can be used to name backends differently than the Automatic Discovery will.

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**, then specify that the first segment should be used. In this case the split delimiter is a / (slash). The backend naming configuration looks similar to the following:



A similar technique can be used 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]
```

What you care about in this case is the "create" service, so your configuration is the same as in the previous screen shot.

6. Once you change the configuration, you should delete all HTTP backends. Then, as the agent rediscovers the backends according to the changed configuration, you see only the service backends in the flow map.

HTTP Backends With Different Formats

Consider a set of HTTP backends that have different formats, for example, some are prefixed with ec2storage, some are salesforce.com or localhost and so on,. In this case, you don't change the automatic discovery rule. Instead you create a custom rule. This is because you need different rules for the different URL formats as follows:

- For format "ec2storage/servicename", you need to use the URL
- For format "salesforce.com", you want to use the host name
- For the other backends, you might want to use the query string

In some cases, your HTTP backend discovery configuration might consist of a combination of the default rule and custom rules. A custom rule to handle the "ec2storage" URLs might look similar to the following:

Create Custom HTTP Backend Discovery Rule

Name

Enabled ☒ ?

Correlation Enabled ☒ ?

Priority ?

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 have at least one condition.

<input checked="" type="checkbox"/>	Host	Contains	<input type="text" value="_ec2storage"/>	⚙
<input type="checkbox"/>	Port	Equals	<input type="text"/>	⚙
<input type="checkbox"/>	URL	Equals	<input type="text"/>	⚙
<input type="checkbox"/>	Query String	Equals	<input type="text"/>	⚙

Backend Naming Configuration

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 to form the Backend name.

Property	Use in Name
Host	<input type="checkbox"/>
Port	<input type="checkbox"/>
URL	<input checked="" type="checkbox"/>
Query String	<input type="checkbox"/>

URL

☒ Use URL in the Backend Name

How will URL be used in the Backend Name?

Learn More

- [Configure Backend Detection \(Java\)](#)
- [Backend Monitoring](#)
- [Monitor Remote Services](#)
- [Configure Backend Detection](#)
- [Flow Maps](#)

JDBC Exit Points for Java

- [Auto-Discovery and Default Naming](#)
- [JDBC Configurable Properties](#)
- [Changing the Default JDBC Automatic Discovery and Naming](#)
 - [Examples](#)
 - [Multiple Databases from Same Vendor](#)
 - [JDBC with Complex URLs](#)
 - [EC2 Hosted Databases](#)
- [Learn More](#)

This topic covers JDBC exit point configuration. To review general information about monitoring databases and remote services (collectively known as backends) and for an overview of backend configuration see [Backend Monitoring](#). For configuration procedures, see [Configure Backend Detection \(Java\)](#).

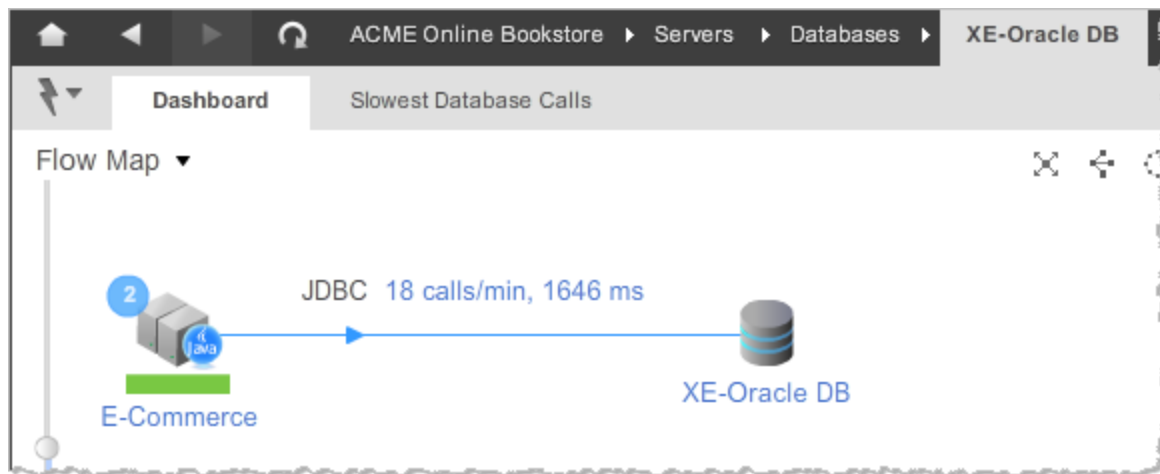
Auto-Discovery and Default Naming

JDBC backend activity consists of all JDBC calls including inserts, queries, updates, getting

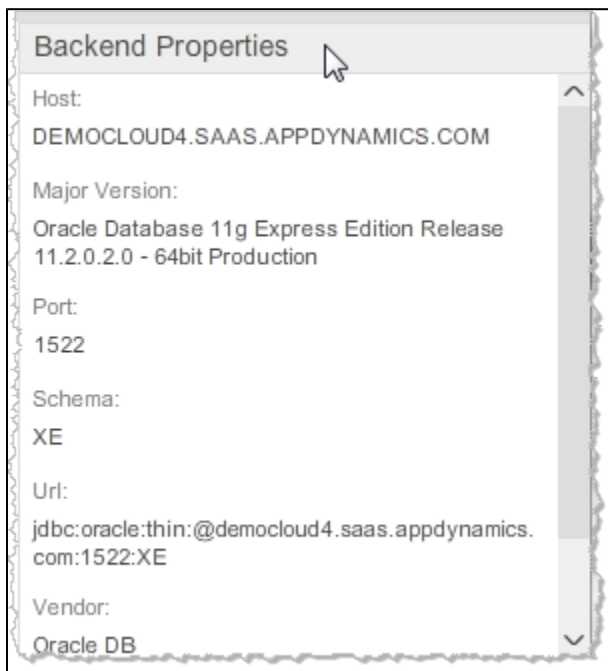
connections from connection pools, and so on. By default, JDBC backends are identified using the logical properties of the database:

- URL
- Host name
- Port number
- Database schema
- Version
- Vendor

From these properties AppDynamics derives a display name, for example "XE-Oracle DB". You can see an Oracle JDBC database on the following flow map.



You can see the values of the database properties on the database dashboard.



JDBC Configurable Properties

You can enable or disable the use of the following properties for JDBC exit points.

Configurable Properties	Property Used by Default in Detection and Naming	Description
URL	Yes	JDBC URL provided to the driver
Host	Yes	Database host
Port	Yes	Database port
Database	Yes	Database schema
Version	Yes	Database version as reported by JDBC driver
Vendor	Yes	Database vendor as reported by JDBC driver

Changing the Default JDBC Automatic Discovery and Naming

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.

Examples

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=titanpfmcl01)(PORT=1
521)))(CONNECT_DATA=(SERVICE_NAME=trafrefpfm01.global.trafigura.com)))(DESCRIPTI
ON=(ADDRESS_LIST=(LOAD_BALANCE=ON)(ADDRESS=(PROTOCOL=TCP)(HOST=titanpfmcl02)(POR
T = 1521)))(CONNECT_DATA=(SERVICE_NAME=trafrefpfm01.global.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:

titanpfmcl01-trafrefpfm01.global.trafigura.com-titanpfmcl02-trafrefpfm01.global.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:

Enabled ☒ ?

Correlation Enabled ☒ ?

Backend Naming Configuration

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.

Property	Use in Name
Host	<input type="checkbox"/>
Port	<input type="checkbox"/>
URL	<input checked="" type="checkbox"/>
Query String	<input type="checkbox"/>

URL

☒ Use URL in the Backend Name

How will URL be used in the Backend Name? Run a Regular Expression on it

Regular Expression

Regular Expression Groups (Comma Separated List of list)

Merge Delimiter

Also see [Regular Expressions In Match Conditions](#).

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.

Create Custom JDBC Backend Discovery Rule

Name:

Enabled: ☒ ?

Correlation Enabled: ☐ Correlation is not supported for JDBC Backends. ?

Priority: ?

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.

Condition	Operator	Value
<input type="checkbox"/> URL	Equals	<input type="text"/>
<input checked="" type="checkbox"/> Host	Contains	EC2
<input type="checkbox"/> Port	Equals	<input type="text"/>
<input type="checkbox"/> Database	Equals	<input type="text"/>
<input type="checkbox"/> Version	Equals	<input type="text"/>
<input type="checkbox"/> Vendor	Equals	<input type="text"/>

Assuming host names of the format "EC2-segment2-segment3", use the following naming configuration:

Backend Naming Configuration

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.

Property	Use in Name
URL	✗
Host	✓
Port	✗
Database	✗
Version	✗
Vendor	✗

Host

☒ Use Host in the Backend Name

How will Host be used in the Backend Name?

1 (Number of Segments)

Split Delimiter:

Merge Delimiter:

Close Cancel OK

This configuration results in a single database icon on the flow map named "EC2".

Learn More

- [Configure Backend Detection \(Java\)](#)
- [Backend Monitoring](#)
- [Monitor Databases](#)

Message Queue Exit Points for Java

- [JMS Message Queue Exit Points](#)
 - [JMS Auto-Discovery and Default Naming](#)
 - [JMS Configurable Properties](#)
 - [Changing the Default JMS Automatic Discovery and Naming](#)
 - [Examples](#)
 - [Monitor the Server by Ignoring the JMS Queue Name](#)
 - [Temporary Queues](#)
 - [Session ID in the Queue Name](#)

- IBM Websphere MQ Message Queue Exit Points
 - IBM Websphere MQ Auto-Discovery and Default Naming
 - IBM MQ Configurable Properties
 - Example: Monitor the Server by Ignoring the MQ Queue Name
- RabbitMQ Message Queue Exit Points
 - RabbitMQ Auto-Discovery and Default Naming
 - RabbitMQ Configurable Properties
 - Changing the Default RabbitMQ Automatic Discovery and Naming
 - Example: Monitor the Server by Ignoring the RabbitMQ Queue Name
- [Learn More](#)

This topic covers message queue exit point configuration, including JMS, IBM MQ, and RabbitMQ. To review general information about monitoring databases and remote services (collectively known as backends) and for an overview of backend configuration see [Backend Monitoring](#). For configuration procedures, see [Configure Backend Detection \(Java\)](#).

For a list of supported message-oriented middleware products, see [Supported Application Servers and Portals for the App Agent for Java](#)

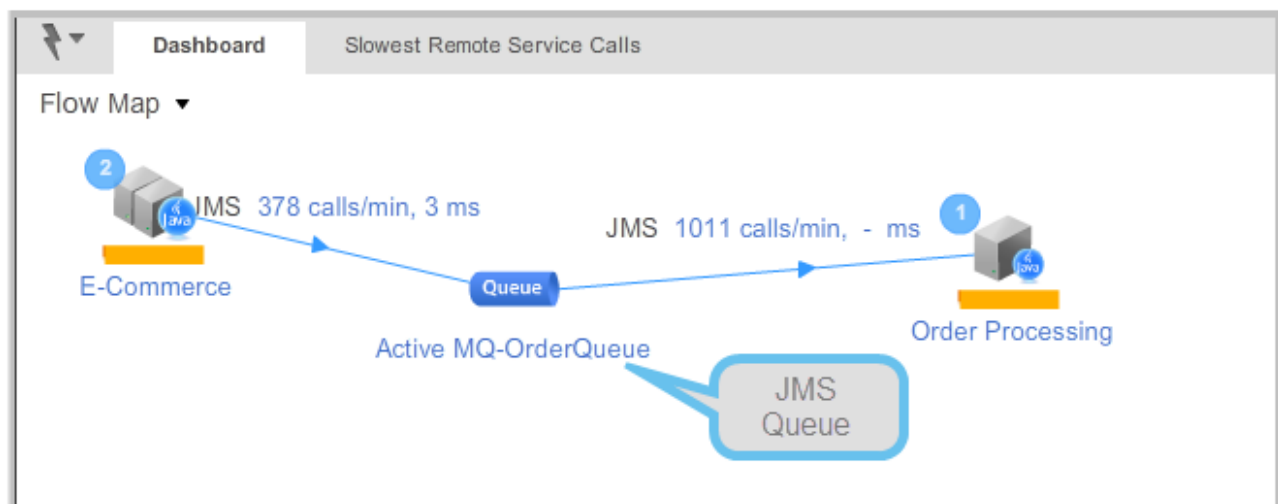
JMS Message Queue Exit Points

JMS Auto-Discovery and Default Naming

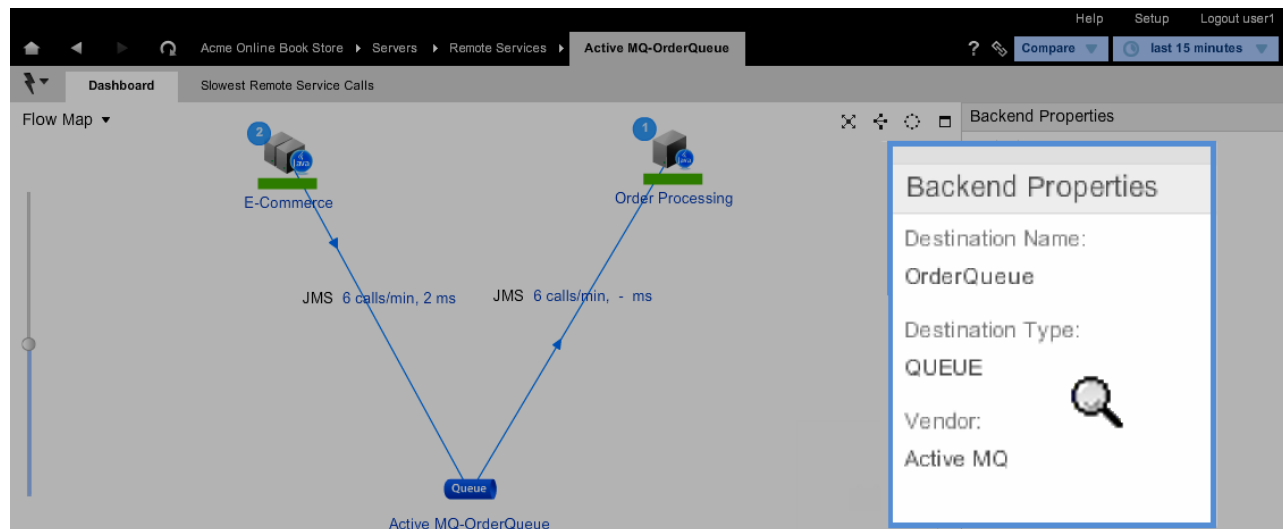
JMS backend activity includes all JMS message send and publish activity. By default, JMS backends are identified using the logical properties of the JMS server, such as: vendor, destination name, and destination type.

The default configuration uses all three properties of the JMS queue.

From the enabled properties AppDynamics derives a display name, for example, ActiveMQ-OrderQueue.



The Backend Properties are visible on the Remote Services dashboard.



JMS Configurable Properties

The following properties can be configured to refine the identification of JMS backends.

Configurable Properties	Property Used by Default in Detection and Naming
Destination	Yes
Destination Type	Yes
Vendor	Yes

Changing the Default JMS Automatic Discovery and Naming

Depending on exactly what you need to monitor, there may be times when you want to change the default JMS configuration. In most cases, you can generate the right name and the correct number of backends to monitor by editing the automatic discovery rule. For example, you can disable use of the Vendor property. If you do, then JMS backends with the same destination and destination type are identified as the same backend and the metrics for calls with the same destination and destination type are aggregated into one backend. Changing the default discovery rule can enable you to monitor the key performance indicators (KPIs) that are of most interest to you.

Examples

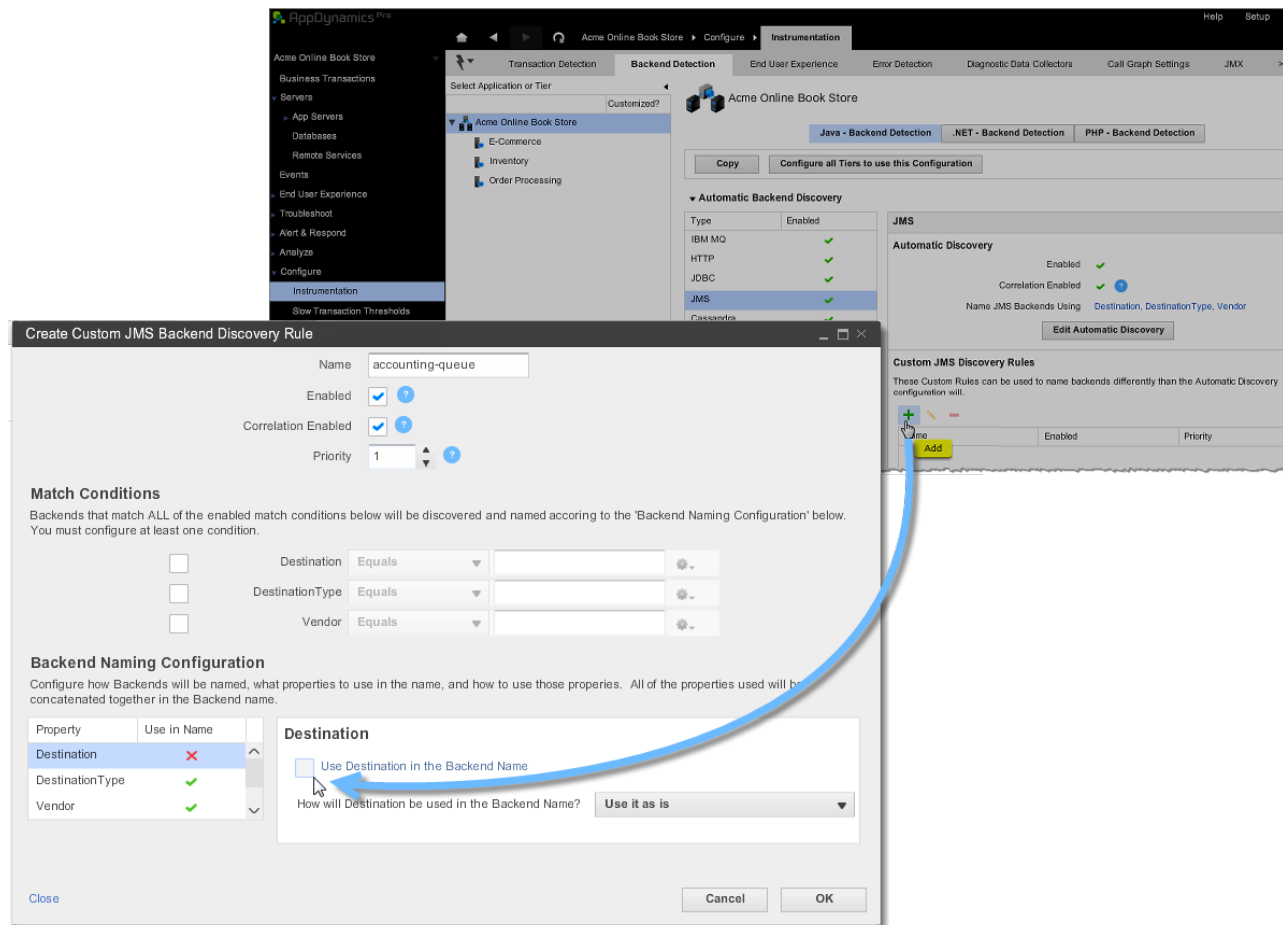
Monitor the Server by Ignoring the JMS Queue Name

In this 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. This configuration looks similar to the following:



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

Name:

Enabled: ☒ ?

Correlation Enabled: ☒ ?

Priority: ?

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 those properties. All of the properties used will be concatenated together in the Backend name.

Property	Use in Name
Destination	<input checked="" type="checkbox"/>
DestinationType	<input checked="" type="checkbox"/>
Vendor	<input checked="" type="checkbox"/>

Destination

☒ Use Destination in the Backend Name

How will Destination be used in the Backend Name? **Run a Regular Expression on it**

Regular Expression

Regular Expression Groups (Comma Separated List of list)

[Close](#) [Cancel](#) [OK](#)

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.

IBM Websphere MQ Message Queue Exit Points

IBM Websphere MQ Auto-Discovery and Default Naming

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 Configurable Properties

You can enable or disable the use of the following properties for IBM MQ exit points.

Configurable Properties	Property Used by Default in Detection and Naming
Destination	Yes

Destination Type	Yes
Host	Yes
Port	Yes
Major Version	Yes
Vendor	Yes

Example: Monitor the Server by Ignoring the MQ Queue Name

In this 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:

Edit Automatic IBM MQ Backend Discovery

Enabled ☒ ?

Correlation Enabled ☒ ?

Backend Naming Configuration

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.

Property	Use in Name
Destination	✗
DestinationType	✗
Host	✓
Port	✓
Major Version	✗
Vendor	✗

Vendor

☐ Use Vendor in the Backend Name

How will Vendor be used in the Backend Name? Use it as is

Close Cancel OK

RabbitMQ Message Queue Exit Points

RabbitMQ Auto-Discovery and Default Naming

RabbitMQ is an open source, commercially supported, messaging middleware that runs on many different operating systems. By default, the App Agent for Java discovers exit points utilizing the RabbitMQ Java API, which is usually shipped as an amqp-client.jar. By default, RabbitMQ

backends are identified by Host, Port, Routing Key, and Exchange. By default, the name would resemble this:

```
amqp://guest@127.0.0.1:5672/exchange/task_queue
```

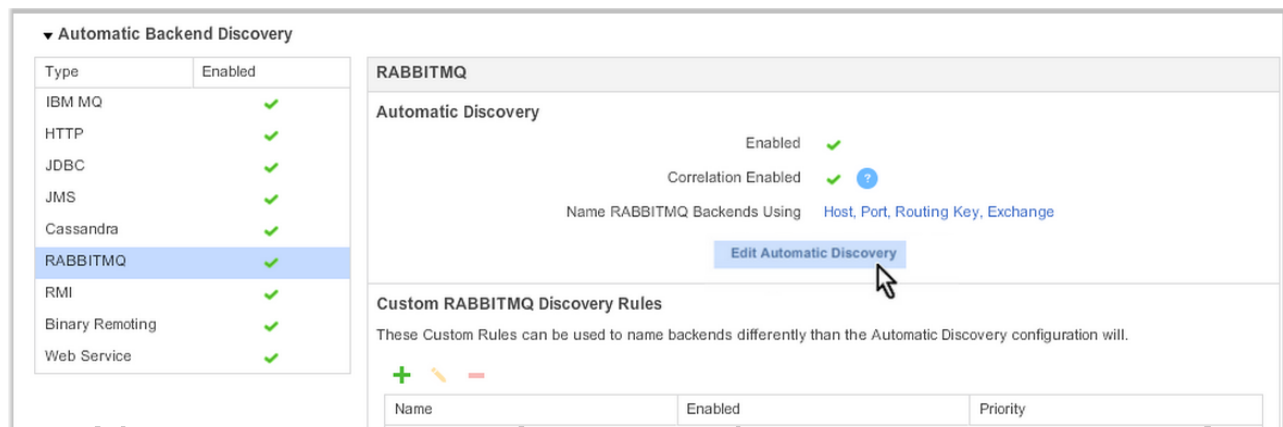
RabbitMQ Configurable Properties

You can enable or disable the use of the following properties for RabbitMQ exit points.

Configurable Properties	Property Used by Default in Detection and Naming
Host	Yes
Port	Yes
Routing Key	Yes
Exchange	Yes

Changing the Default RabbitMQ Automatic Discovery and Naming

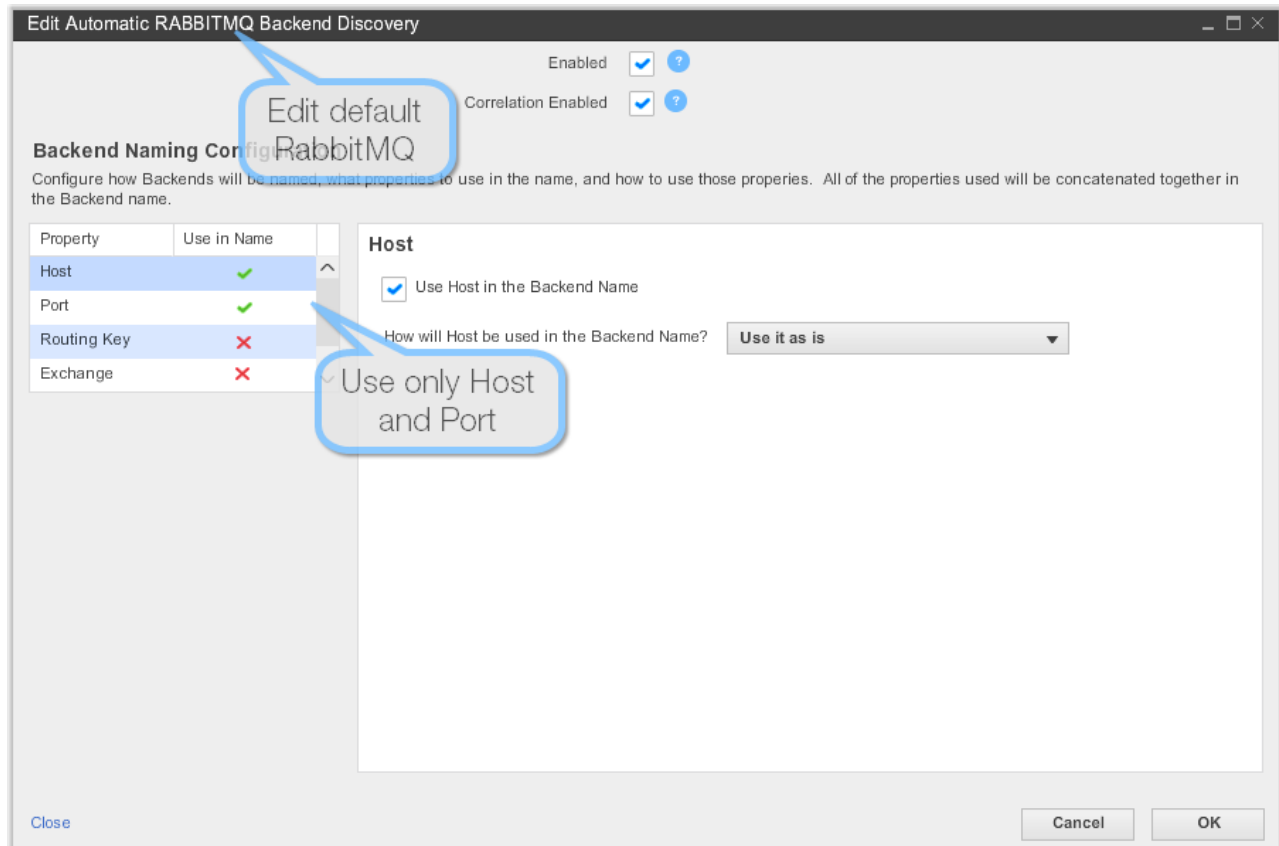
You can change the default RabbitMQ automatic discovery attributes by clicking **Edit Automatic Discovery**.



A window appears where you can edit the automatic RabbitMQ backend discovery rules.

Example: Monitor the Server by Ignoring the RabbitMQ Queue Name

Configuring properties, such as host and port, is useful where you have lots of queues and want to monitor the health of the server and not the message queue. In this situation, a discovery rule using only host and port might be the most useful strategy as follows:



Learn More

- [Configure Backend Detection \(Java\)](#)
- [Backend Monitoring](#)
- [Monitor Remote Services](#)
- [Configure Backend Detection](#)
- [Flow Maps](#)
- [AppDynamics eXchange RabbitMQ Monitoring Extension](#) for visibility into the queue.

Web Services Exit Points for Java

- [Auto-Discovery and Default Naming](#)
- [Web Services Configurable Properties](#)
- [Changing the Default Web Service Automatic Discovery and Naming](#)
 - [Examples](#)
 - [Web Services Using HTTP for Transport](#)
- [Learn More](#)

This topic covers web services exit point configuration. To review general information about monitoring databases and remote services (collectively known as backends) and for an overview of backend configuration see [Backend Monitoring](#). For configuration procedures, see [Configure Backend Detection \(Java\)](#).

Auto-Discovery and Default Naming

Web service backend activity includes all web service invocations. Web service backends are identified using the web service name.

Web Services Configurable Properties

Configurable Properties	Property Used by Default in Detection and Naming
Service	Yes
URL	No
Operation	No
Soap Action	No
Vendor	No

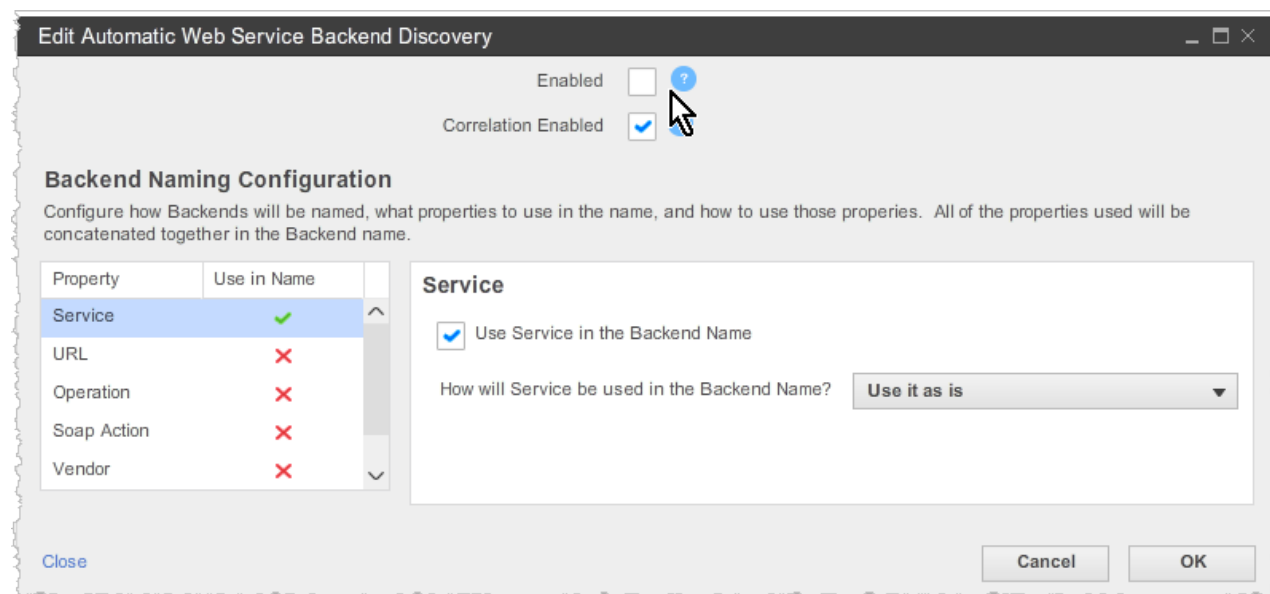
Changing the Default Web Service Automatic Discovery and Naming

Depending on exactly what you need to monitor, there may be times when you want to change the default configuration. In most cases, you can generate the right name and the correct number of backends to monitor by editing the automatic discovery rule.

Examples

Web Services Using HTTP for Transport

You can disable automatic discovery of a backend type entirely if this better suits your monitoring needs. For example, if your web services use HTTP for transport and you are fine with them being discovered as HTTP backends, you can disable discovery of the Web Service type backends.



Learn More

- [Configure Backend Detection \(Java\)](#)
- [Backend Monitoring](#)
- [Monitor Remote Services](#)

Cassandra Exit Points for Java

- [Auto-Discovery and Default Naming](#)
- [Cassandra Configurable Properties](#)
- [Changing the Default Cassandra Automatic Discovery and Naming](#)
- [Learn More](#)

This topic covers Cassandra exit point configuration. To review general information about monitoring databases and remote services (collectively known as backends) and for an overview of backend configuration see [Backend Monitoring](#). For configuration procedures, see [Configure Backend Detection \(Java\)](#).

Auto-Discovery and Default Naming

By default, AppDynamics automatically detects and identifies Cassandra databases and Cassandra CQL3.

Cassandra Configurable Properties

Configurable Properties for Database	Configurable Properties for CQL3	Property Used by Default in Detection and Naming
Host	Host	Yes
Port	Port	Yes
transport		Yes
keyspace		Yes

Changing the Default Cassandra Automatic Discovery and Naming

Depending on what you need to monitor, you can change the default configuration by disabling one or more properties.

Learn More

- [Configure Backend Detection \(Java\)](#)
- [Backend Monitoring](#)
- [Configure Backend Detection](#)

RMI Exit Points for Java

- [Auto-Discovery and Default Naming](#)
- [RMI Properties](#)
- [Changing the Default RMI Automatic Discovery and Naming](#)
- [Learn More](#)

This topic covers Java Remote Method Invocation (RMI) exit point configuration. To review general information about monitoring databases and remote services (collectively known as backends) see [Backend Monitoring](#).

Auto-Discovery and Default Naming

The App Agent for Java automatically discovers backends called using the standard Java RMI

API. For a list of supported RMI frameworks, see [Supported Application Servers and Portals for the App Agent for Java](#)

RMI Properties

Configurable Properties	Property Used by Default in Detection and Naming
URL	Yes

Changing the Default RMI Automatic Discovery and Naming

Depending on what you need to monitor, you can change the default configuration to use a portion of the URL.

Learn More

- [Configure Backend Detection \(Java\)](#)
- [Backend Monitoring](#)
- [Monitor Remote Services](#)

Thrift Exit Points for Java

- [Automatic Discovery and Default Naming](#)
- [Thrift Configurable Properties](#)
- [Changing the Default Thrift Automatic Discovery and Naming](#)
- [Learn More](#)

This topic explains Thrift exit point configuration. To review general information about monitoring databases and remote services (collectively known as backends) and for an overview of backend configuration see [Backend Monitoring](#). For configuration procedures, see [Configure Backend Detection \(Java\)](#).

Automatic Discovery and Default Naming

By default, AppDynamics automatically detects and identifies Apache Thrift exit points (backends). See [Apache Thrift](#) for details.

Thrift Configurable Properties

You can enable or disable the use of the following properties for Thrift exit points.

Configurable Properties	Used by Default in Detection and Naming
Host	Yes
Port	Yes
transport	Yes

Changing the Default Thrift Automatic Discovery and Naming

Depending on what you need to monitor, you can change the default configuration by disabling one or more properties.

Learn More

- [Configure Backend Detection \(Java\)](#)
- [Backend Monitoring](#)
- [Monitor Remote Services](#)

Configure Memory Monitoring for Java

See:

- [Configure and Use Custom Memory Structures for Java](#)
- [Configure and Use Object Instance Tracking for Java](#)

Configure Automatic Leak Detection for Java

- [Prerequisites for Automatic Leak Detection](#)
- [Memory Leaks in a Java Environment](#)
- [AppDynamics Java Automatic Leak Detection](#)
 - [Automatic Leak Detection Support](#)
 - [Conditions for Troubleshooting Java Memory Leaks](#)
- [Starting Automatic Leak Detection](#)
 - [To start automatic leak detection on a node](#)
- [Learn More](#)

This topic helps you understand how to configure automatic leak detection.

Prerequisites for Automatic Leak Detection

Automatic leak detection can only be used with specific JVMs. See [JVM Support](#).

Memory Leaks in a Java Environment

While the JVM's garbage collection greatly reduces the opportunities for memory leaks to be introduced into a codebase, it does not eliminate them completely. For example, consider a web page whose code adds the current user object to a static set. In this case, the size of the set grows over time and could eventually use up significant amounts of memory. In general, leaks occur when an application code puts objects in a static collection and does not remove them even when they are no longer needed.

In high workload production environments if the collection is frequently updated, it may cause the applications to crash due to insufficient memory. It could also result in system performance degradation as the operating system starts paging memory to disk.

AppDynamics Java Automatic Leak Detection

AppDynamics automatically tracks every Java collection (for example, HashMap and ArrayList) that meets a set of criteria defined below. The collection size is tracked and a linear regression model identifies whether the collection is potentially leaking. You can then identify the root cause of the leak by tracking frequent accesses of the collection over a period of time.

Once a collection is qualified, its size, or number of elements, is monitored for long term growth trend. A positive growth indicates that the collection is potentially leaking!

Once a leaking collection is identified, the agent automatically triggers diagnostics every 30

minutes to capture a shallow content dump and activity traces of the code path and business transactions that are accessing the collection. By drilling down into any leaking collection monitored by the agent, you can manually trigger Content Summary Capture and Access Tracking sessions. See [Configure Automatic Leak Detection for Java](#)

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 [Configure and Use Custom Memory Structures for Java](#).

Automatic Leak Detection Support

Ensure AppDynamics supports Automatic Leak Detection on your JVM. See [JVM Support](#).

Conditions for Troubleshooting Java Memory Leaks

Automatic Leak Detection uses On Demand Capture Sessions to capture any actively used collections (i.e. any class that implements JDK Map or Collection interface) during the Capture period (default is 10 minutes) and then qualifies them based on the following criteria:

For a collection object to be identified and monitored, it must meet the following conditions:

- The collection has been alive for at least N minutes. Default is 30 minutes, configurable with the [minimum-age-for-evaluation-in-minutes](#) node property.
- The collection has at least N elements. Default is 1000 elements, configurable with the [minimum-number-of-elements-in-collection-to-deep-size](#) node property.
- The collection Deep Size is at least N MB. Default is 5 MB, configurable with the [minimum-size-for-evaluation-in-mb](#) property.
The Deep Size is calculated by traversing recursive object graphs of all the objects in the collection.

See [App Agent Node Properties](#) and [App Agent Node Properties Reference by Type](#).

Starting Automatic Leak Detection

To start automatic leak detection on a node

1. In the left navigation pane, click **Servers -> App Servers -> <tier> -> <node>**. The Node Dashboard opens.
2. Click the Memory tab.
3. Click the Automatic Leak Detection subtab.
4. Click **ON**.

Automatic Leak Detection is available for Sun JVM version 1.5 or higher, JRockit 1.6 or higher. The following collection packages are supported - JDK, Apache Commons, backport-utils (edu.emory.*) and concurrent (EDU.oswego).

Automatic Leak Detection uses **On Demand Capture Sessions** to capture any class that implements JDK 'Map' or 'Collection' interface during the 'Capture' period (default is 10 minutes) and then qualifies them based on the following criteria:

- The Collection has **been alive for at least N minutes** (default is 20 minutes, and is configurable when starting a capture session)
- The Collection has **at least N elements** (default is 1000 elements, and is configurable at the Node level: Actions -> Configure App Server Agent)
- The Collection **Deep Size is at least N MB** (default is 5 MB, and is configurable at the Node level: Actions -> Configure App Server Agent). The **Deep Size** is calculated by traversing the recursive object graphs of all objects in the Collection.

Once a collection is qualified its size (number of elements) will be monitored for long term growth trend. **A positive growth will mark the collection as potentially leaking!**

Once a leaking Collection is identified, the Agent automatically triggers diagnostics every 30 minutes to capture a shallow Content Dump and Activity Traces (i.e. code path and BT that are accessing the collection). Content Summary and Access Tracking sessions can also be triggered manually for any collection that's monitored by Agent.

When to trigger On Demand Capture Sessions

A good time to trigger a short capture session is during periods of growth in heap utilization %.

When to use Automatic Leak Detection vs Custom Memory Structures

Automatic Leak Detection tracks only Map and Collection libraries and the data captured is valid only for a given JVM Session. Custom Memory Structures can be configured to monitor any custom object (not just maps and collections) created by app and the size data can be traced across JVM restarts. Automatic Leak Detection is typically used to identify leaks and Custom Memory Structures is used to track large coarse grained custom cache objects.

AppDynamics begins to automatically track the top 20 application classes and the top 20 system (core Java) classes in the heap.

Class	Collection Size	Potentially Leaking	Object Creation Time	JVM Start Time	Object Instance ID	Status	Collection Size Trend
java.util.concurrent.ConcurrentHashMap	2615	Yes	05/08/13 5:38:37 PM	05/08/13 5:32:04 PM	51		

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 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.

Learn More

- [Troubleshoot Java Memory Leaks](#)

Configure and Use Object Instance Tracking for Java

- [Prerequisites for Object Instance Tracking](#)
 - [Specifying the Classpath](#)
- [Starting Object Instance Tracking](#)
 - [To start object instance tracking on a node](#)
- [Tracking Specific Classes](#)
 - [To track instances of custom classes](#)
- [Learn More](#)

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 [Troubleshoot Java Memory Thrash](#).

Prerequisites for Object Instance Tracking

- Object Instance Tracking can be used only for Sun JVM v1.6.x and later.
- If you are running with the JDK then tools.jar will probably be setup correctly, but if you are running with the JRE you must add tools.jar to JRE_HOME/lib/ext and restart the JVM for this feature to start working. You can find the tools.jar file in JAVA_HOME/lib/tools.jar.
- In some cases 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 a JDK tool, set the classpath using the -classpath option. This sets the classpath for the application only. For example:

On Windows

```
java -classpath <complete-path-to-tools.jar>;%CLASSPATH% -jar myApp.jar
```

OR

On Unix

```
java -classpath <complete-path-to-tools.jar>:$CLASSPATH -jar myApp.jar
```

Alternatively, you can set the CLASSPATH variable for your entire environment. For example:

On Windows

```
SET CLASSPATH=%CLASSPATH%;%JAVA_HOME%\lib\tools.jar
```

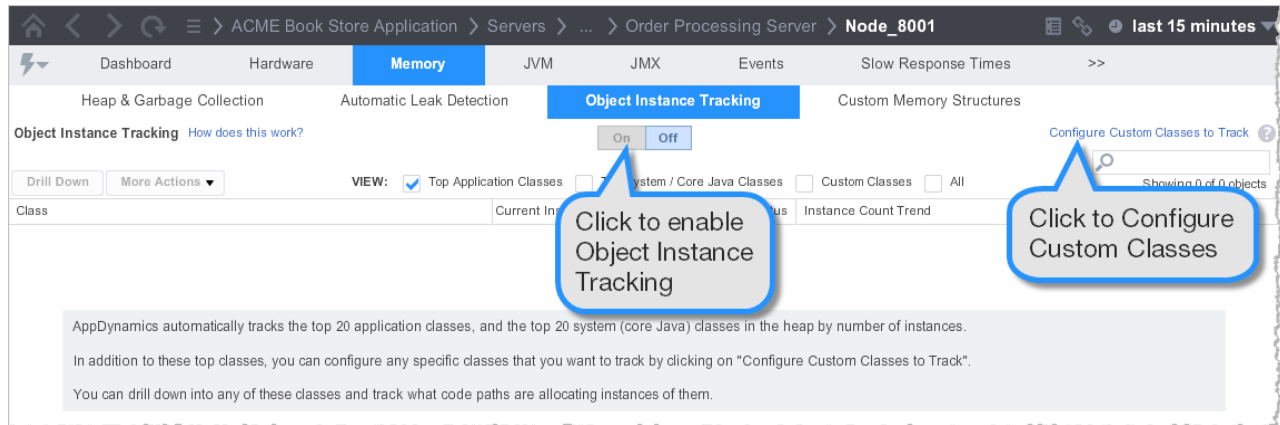
On Unix

```
CLASSPATH=$CLASSPATH:$JAVA_HOME/lib/tools.jar
```

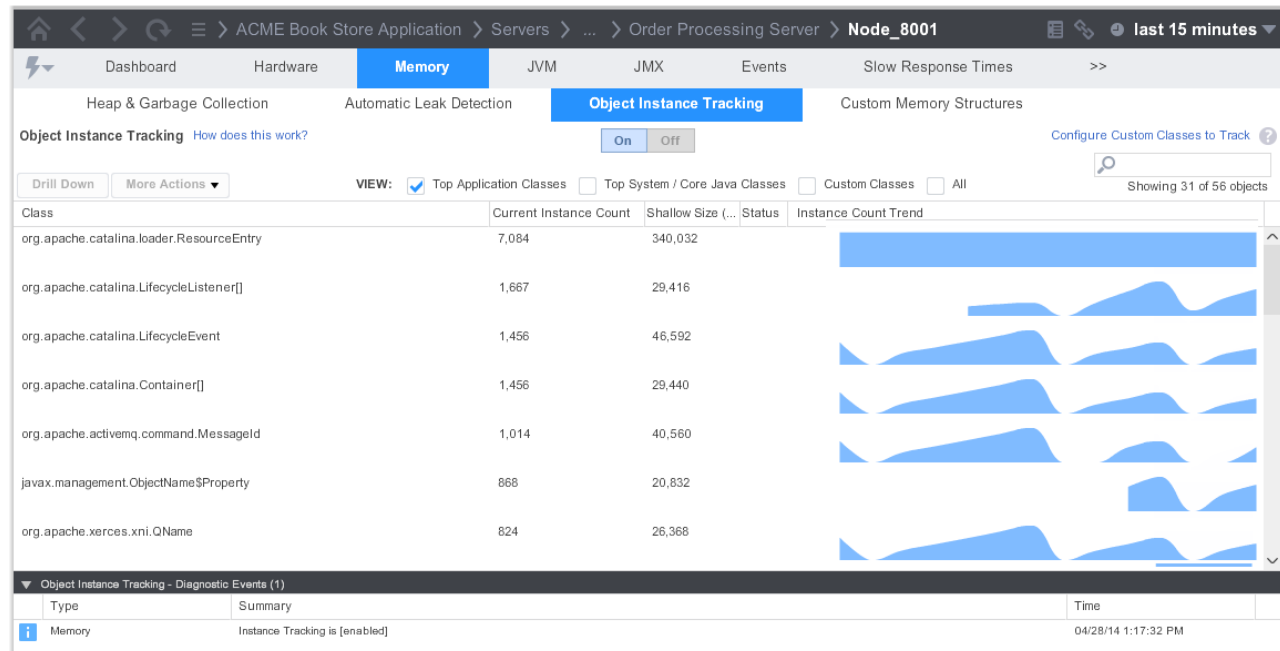
Starting Object Instance Tracking

To start object instance tracking on a node

1. In the left navigation pane, click **Servers** -> **App Servers** -> <tier> -> <node>. The Node Dashboard opens.
2. Click the Memory tab.
3. Click the Object Instance Tracking subtab.
4. Click **ON**.



AppDynamics begins to automatically track the top 20 application classes and the top 20 system (core Java) classes in the heap.



Tracking Specific Classes

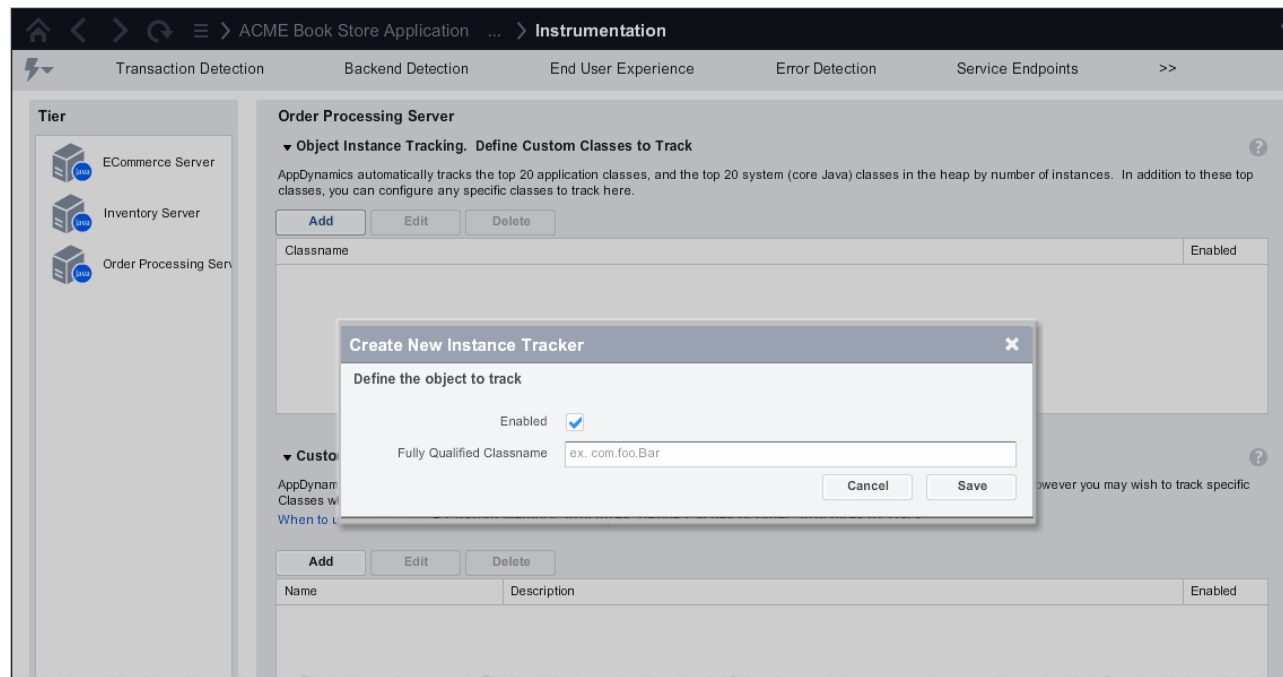
Check against the required set of classes to enable instance tracking for each set. For improved performance, only the top 20 application classes and the top 20 system (core Java) classes in the heap are tracked automatically.

Use the Configure Custom Classes to Track option to specify instances of specific classes.

Classes configured here are only tracked if their instance count is among the top 1000 instance counts in the JVM.

To track instances of custom classes

1. In the left navigation pane, click **Servers -> App Servers -> <tier> -> <node>**. The Node Dashboard opens.
2. From the Object Instance Tracking subtab, click **Configure Custom Classes To Track** on the rightmost corner of the window.
3. In the Object Instance Tracking - Define Custom Classes to Track section for the tier, click **Add**.



4. In the Create New Instance Tracker window, check **Enabled**.
5. Enter the fully-qualified class name of the class to track.
6. Click **Save**.

You can also access this configuration page by selecting **Configuration -> Instrumentation -> Memory Monitoring**.

You can edit or delete the object tracing configuration after it has been created.

Learn More

- [Troubleshoot Java Memory Thrash](#)

Configure and Use Custom Memory Structures for Java

- [Custom Memory Structures and Memory Leaks](#)
 - [Using Automatic Leak Detection vs Monitoring Custom Memory Structures](#)
 - [To identify custom memory structures](#)
 - [To Add a Custom Memory Structure](#)
- [Identifying Potential Memory Leaks](#)

- [Diagnosing memory leaks](#)
- [Isolating a leaking collection](#)
- [Access Tracking](#)
- [Learn More](#)

This topic describes how to configure custom memory structures and monitor large coarse grained custom cache objects.

AppDynamics provides different levels of memory monitoring for multiple JVMs. Ensure custom memory structures are supported in your JVM environment. See [JVM Support](#).



CPU Overhead Caution

Due to high CPU usage, Custom Memory Structures monitoring should only be enabled while debugging a problem.

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:

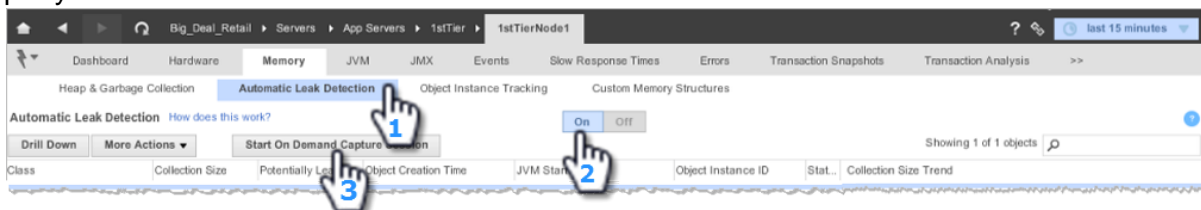
- Cache access for slow, very slow, and stalled business transactions
- Usage statistics, rolled up to the Business Transaction level
- Keys being accessed
- Deep size of internal cache structures

Using Automatic Leak Detection vs Monitoring Custom Memory Structures

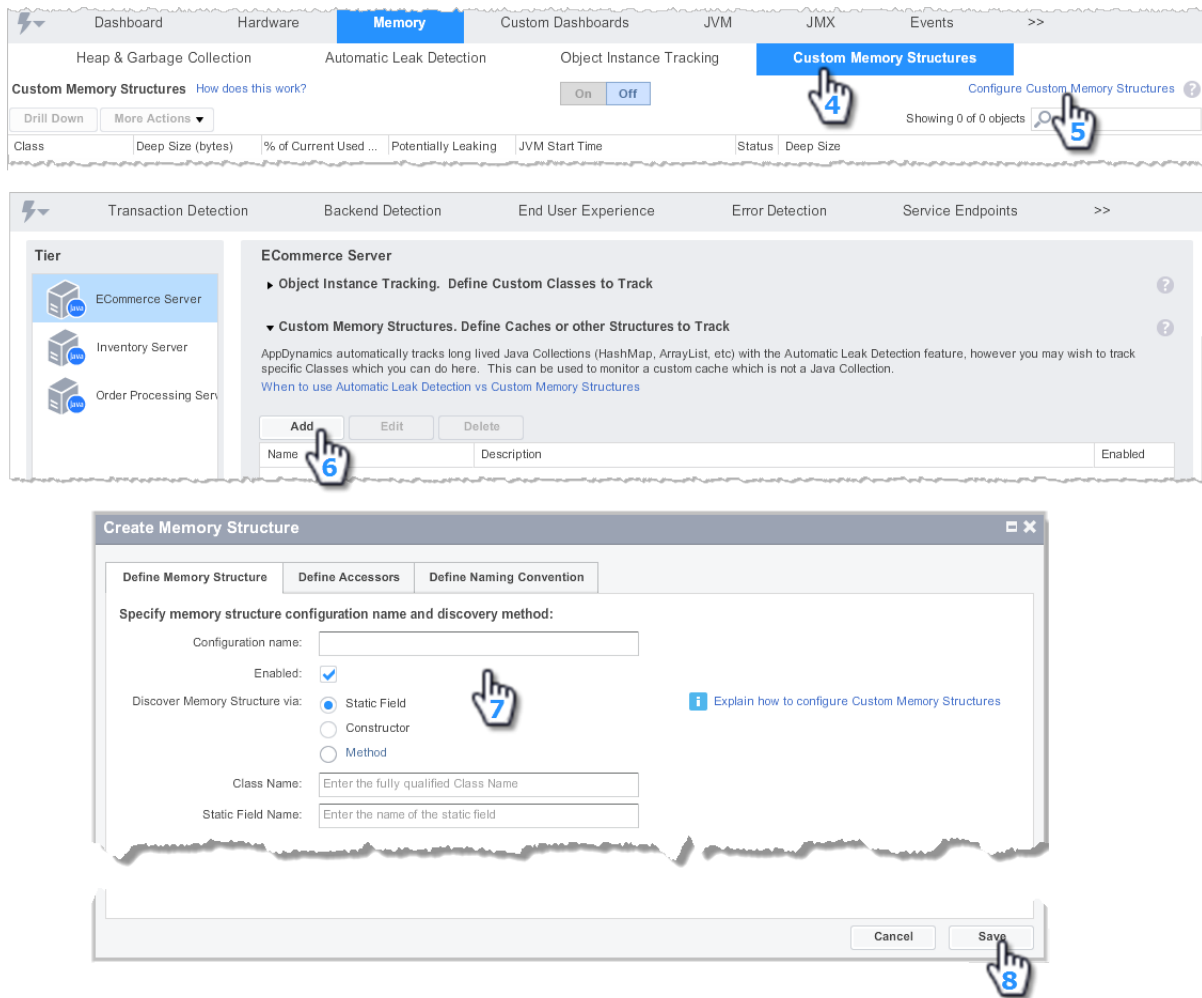
The automatic leak detection feature captures memory usage data for all map and collection libraries in a JVM session. However, custom memory structures might or might not contain collections objects. For example, you may have a custom cache or a third party cache like Ehcache for which you want to collect 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 and custom memory structures is 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.

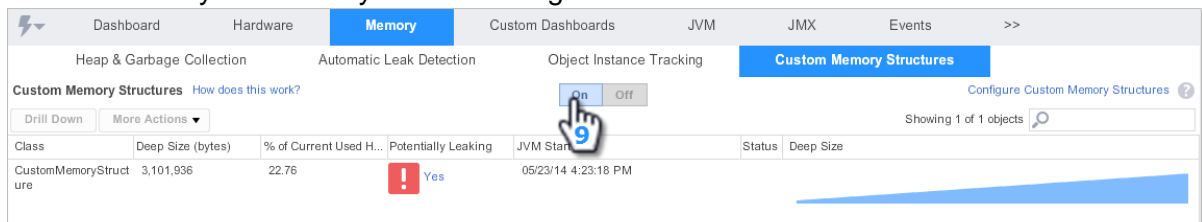
1. On the Node Dashboard, use the Automatic Leak Detection, On Demand Capture Session feature to determine which classes aren't being monitored, for example, custom or third party caches such as EhCache.



2. Configure Custom Memory Structures and then restart the JVM if necessary.



3. Turn on Custom Memory Structures monitoring to detect potential memory leaks in the custom memory structures you have configured.



4. Drill down into leaking memory structures for details that will help you determine where the leak is.

To identify custom memory structures

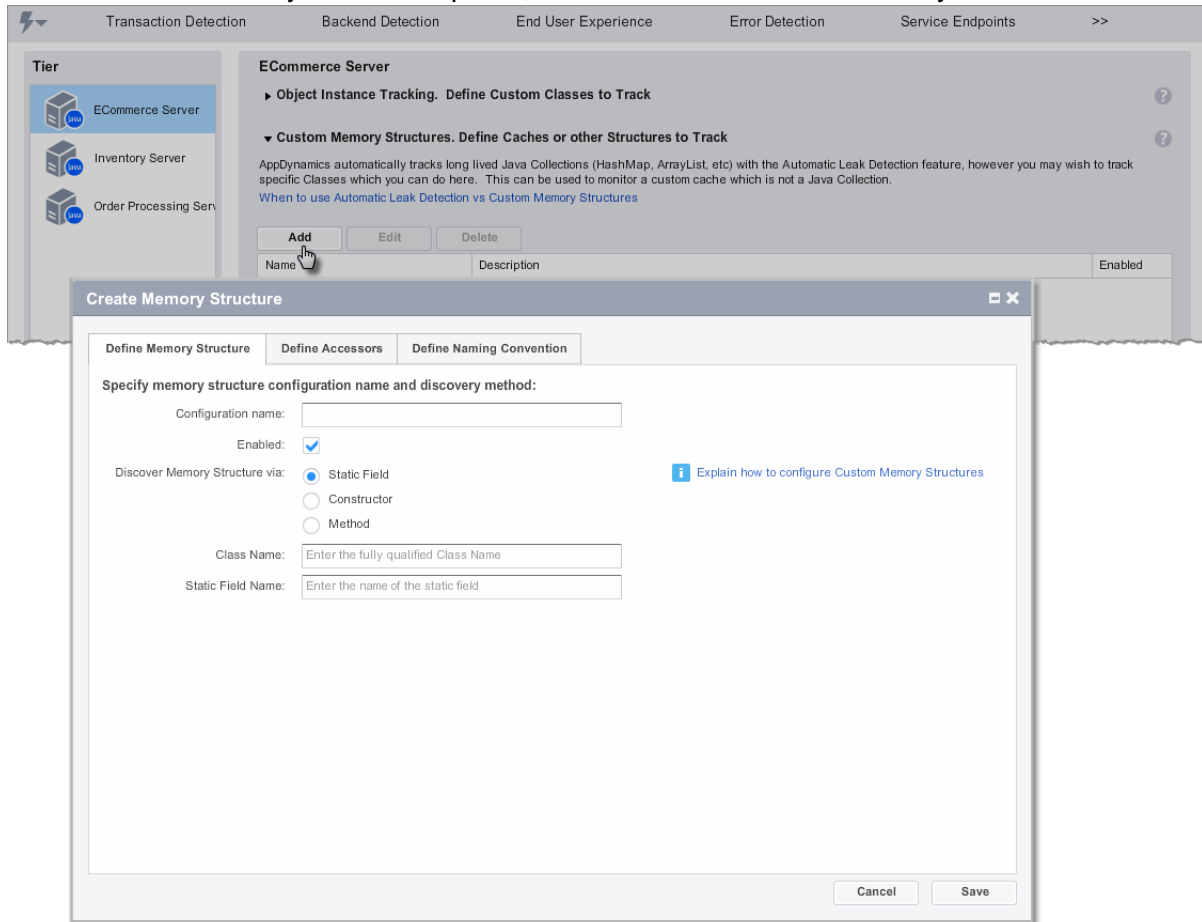
1. On the Automatic Leak Detection subtab of the Memory tab, click **On**.
2. Click **Start On Demand Capture Session** to capture information on which classes are accessing which collections objects. Use this information to identify custom memory structures.

AppDynamics captures the top 1000 classes, by instance count.

To Add a Custom Memory Structure

These instructions provide an alternate method to accessing the Custom Memory Structures pane than the workflow above shows. Use the method that is most convenient to you.

1. From the left navigation pane select **Configure -> Instrumentation**.
2. In the Tier panel, click the tier for which you want to configure a custom memory structure and then click **Use Custom Configuration for this Tier**.
3. On the top menu, click the Memory Monitoring tab.
4. In the Custom Memory Structures panel, click **Add** to add a new memory structure.



- a. In the Create Memory Structure window

- Specify the configuration name.
- Click **Enabled**.
- Specify the discovery method.

The discovery method provides three options to monitor the custom memory structure. The discovery method determines how the agent gets a reference to the custom memory structure. AppDynamics needs this reference to monitor the size of the structure. Select one of the three options for the discovery method:

- Discover using Static Field.
- Discover using Constructor.
- Discover using Method.

In many cases, especially with caches, the object for which a reference is needed is created early in the life cycle of the application.

Example for using static field	Example for using Constructor	Example for using method
<pre>public class CacheManag er { private static Map userCache< String> User>; }</pre>	<pre>public class CustomerCa che { public CustomerCa che(); }</pre>	<pre>public Class CacheManag er{ public List<Order >; getOrderCa che(); {} }</pre>
Notes: Monitors deep size of this Map.	Notes: Monitors deep size of CustomerCache object(s).	Notes: Monitors deep size of this list.

Restart the JVM after the discovery methods are configured to get the references for the object.

- b. (Optional) Define accessors.

Click **Define Accessors** to define the methods used to access the custom memory structure. This information is used to capture the code paths accessing the custom memory structure.

- c. (Optional) Define the naming convention.

Click **Define Naming Convention**. These configurations differentiate between custom memory structures.

There are situations where more than one custom Caches are used, but only few of them need monitoring. In such a case, use the **Getter Chain** option to distinguish amongst such caches. For all other cases, use either value of the field on the object or a specific string as the object name.

- d. Click **Save** to save the configuration.

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:

Class	Deep Size (bytes)	% of Current Used H...	Potentially Leaking	JVM Start Time	Status	Deep Size
CustomMemoryStruct ure	3,101,936	22.76	Yes	05/23/14 4:23:18 PM		

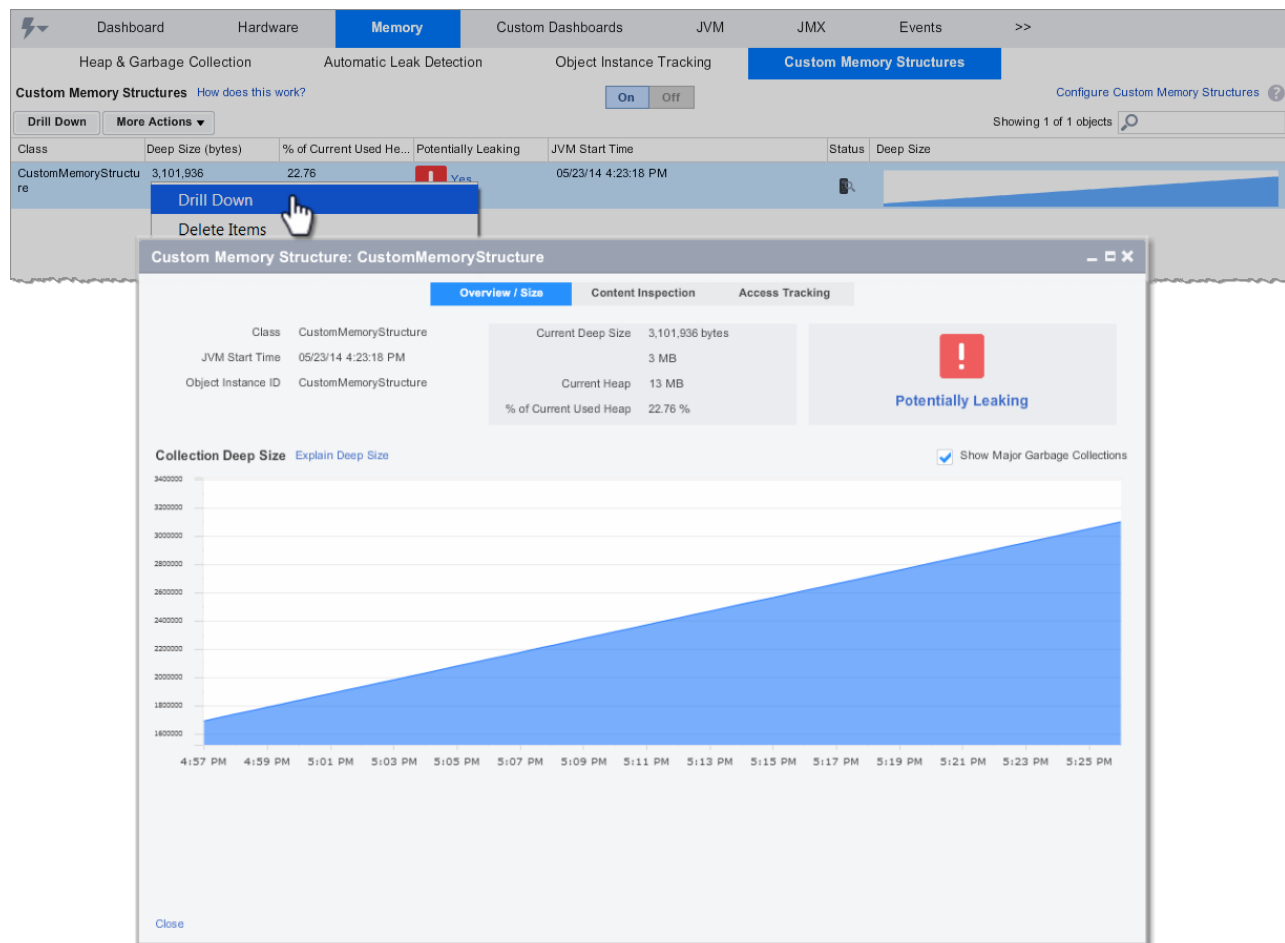
The Custom Memory Structures dashboard provides the following information:

- **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 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.

Custom Memory Structure: CustomMemoryStructure

Overview / Size **Content Inspection** Access Tracking

Time

- 05/27/14 2:31:52 PM
- 05/27/14 2:32:52 PM
- 05/27/14 3:01:52 PM
- 05/27/14 3:02:52 PM
- 05/27/14 3:31:52 PM
- 05/27/14 3:32:52 PM
- 05/27/14 4:01:52 PM
- 05/27/14 4:02:52 PM**
- 05/27/14 4:31:52 PM
- 05/27/14 4:32:52 PM

Refresh

Query for the latest available Content Summaries

Start Content Summary Capture Session

Start a session to capture the summary of the contents of this Collection

Dump Contents to Disk

Dump the full contents of this collection to the AppDynamics App Server agent log directory.

Content Summary

Export ☐ Hide java.util.*

Classname	Count	Size (bytes)
java.lang.String	9052	7,096,768
memorymonitoring.CustomMemoryStructure\$Node	9052	217,248
memorymonitoring.CustomMemoryStructure	1	24

Deep Size: 7,314,040 bytes 7 MB

Close

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.

Custom Memory Structure: CustomMemoryStructure

Overview / Size

Content Inspection

Access Tracking

Session Time

05/27/14 2:33:47 PM
05/27/14 3:03:47 PM
05/27/14 3:33:47 PM
05/27/14 4:03:47 PM
05/27/14 4:33:47 PM

Refresh

Query for data from the most recent Access Tracking Sessions

Start Access Tracking Session

This will start a session to track code accessing this Collection (get(), put(), etc)

Code Paths and Business Transactions accessing this Collection (get(), put(), etc)

05/27/14 3:33:47 PM

Export

Code Paths

Code Path	Occurrences
memorymonitoring.CustomMemoryStructure.addNewEntries(CustomMemoryStructure.java) at memorymonitoring.App.addToCustomMemoryStructure(App.java:24) at memorymonitoring.App.main(App.java:9) at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method) at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57) at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43) at java.lang.reflect.Method.invoke(Method.java:601) at com.intellij.rt.execution.application.AppMain.main(AppMain.java:120)	100

Business Transactions accessing this Collection

Transaction Name	Occurrences
POJO	100

Close

Learn More

- [Troubleshoot Java Memory Leaks](#)

Configure Background Tasks for Java

- [Pre-Configured Frameworks for Java Background Tasks](#)
- [Enabling Automatic Discovery for Background Tasks](#)
 - [To enable discovery for a background task using a common framework](#)
- [Configuring Background Batch or Shell Files using a Main Method](#)
 - [To instrument the main method of a background task](#)
- [Learn More](#)

In a Java environment background tasks are detected using POJO entry points. It is the same basic procedure as defining entry points for business transactions, except that you check the Background Task check box. For instructions see [Configure Background Tasks](#).

Pre-Configured Frameworks for Java Background Tasks

When enabled, AppDynamics provides discovery for the following Java background-processing task frameworks:

- Quartz
- Cron4J
- JCronTab
- JavaTimer

Configure Background Tasks

Enabling Automatic Discovery for Background Tasks

Automatic discovery of background tasks is disabled by default. When you know that there are background tasks in your application environment and you want to monitor them, first enable automatic discovery so that AppDynamics will detect the task.

AppDynamics provides preconfigured support for some common frameworks. If your application is not using one of the default frameworks you can create a custom match rule.

To enable discovery for a background task using a common framework

1. In the left navigation pane, click **Configure -> Instrumentation**.
2. On the Transaction Detection tab, select the tier for which you want to enable monitoring.
3. Click **Use Custom Configuration for this Tier**.
4. Scroll down to the Custom Match Rules pane.
5. Do one of the following
 - If you are using a pre-configured framework, select the row of the framework and click the pencil icon, or double-click on the row to open the Business Transaction Match Rule window. By default the values are populated with rule name and the class and method names for the particular framework. Verify that those are the correct names for your environment.
 - OR
 - If you are using a custom framework, select the match criteria and enter the Class Name and Method Name.

The Background Task check box should be already checked.

6. Check **Enabled**.

7. Click **Save**.


The custom match rule for the background task will take effect and the background task will display in the Business Transaction List.

Once you enable discovery, every background task is identified based on following attributes:

- Implementation class name
- Parameter to the execution method name

Configuring Background Batch or Shell Files using a Main Method

Sometimes background tasks are defined in batch or shell files in which the main method triggers the background processing. In this situation, the response time of the batch process is the duration of the execution of the main method.

 **IMPORTANT:** Instrument the main method only when the duration of the batch process is

equal to the duration of the main method. Otherwise choose another method that accurately represents the unit of work for the background process.

To instrument the main method of a background task

1. In the left navigation pane, click **Configure -> Instrumentation**.
2. In the **Transaction Detection** section select the tier for which you want to instrument the main method.
3. In the **Custom Rules** section click **Add** (the "+" icon).
4. From the **Entry Point** type drop down list, click **POJO**.
5. Enter a name for the custom rule.
6. Check **Background Task**.
7. Check **Enabled**.
8. Enter "main" as the match value for **Method Name**.

New Business Transaction Match Rule - POJO

Name: OvernightBatchRun

Enabled: ☒

Background Task: ☒

Transaction Match Crit... | Transaction Splitting | Exclude Rule

Define match criteria for a POJO method which be will an entry point for a Business Transaction

☒ Match Classes * with a Class Name that Equals com.foo.OvernightOrderProce:

☒ Method Name * Equals main

Cancel Create Custom Match Rule

9. Save the changes.
10. To ensure that the name of the script file is automatically picked up as a background task, configure your Java Agent for that node. See [Configure App Agent for Java for Batch Processes](#).

Learn More

- [Configure Background Tasks](#)
- [Configure App Agent for Java for Batch Processes](#)
- [POJO Entry Points](#)

Import and Export Transaction Detection Configuration for Java

- [Import and Export Auto-Detected Entry Point Configurations](#)
 - To import or export the configurations for all the auto-detected entry-points to or from an application

- To import or export the configuration for a single auto-detected entry point type to or from an application
- To import or export the configurations for all the auto-detected entry-points to or from a tier
- To import or export the configuration for a single auto-detected entry point type to or from a tier
- Import and Export Custom Match and Exclude Rules
 - Import and Export Custom Match Rules
 - To import or export a single custom match rule to or from an application
 - To import or export a single custom match rule to or from a tier
 - Import and Export Exclude Rules
 - To import or export a single exclude rule to or from an application
 - To import or export a single exclude rule to or from a tier
- Overwrite Parameter
- Learn More








You can export your transaction detection configurations from one application to another using the AppDynamics REST API. This capability allows you to re-use transaction detection configurations in different applications instead of re-configuring each application manually using the AppDynamics console.

You can export from an application or tier configuration and import to an application or tier configuration.

This feature is available for Java platforms only.

You can import and export:

- **auto-detected entry point configurations**

▼ Entry Points		
Type	Transaction Monitoring	Automatic Transaction Detection
 Servlet	<input checked="" type="checkbox"/> Enabled	<input checked="" type="checkbox"/> Discover Transactions automatically for all Servlet requests Configure Naming <input type="checkbox"/> Enable Servlet Filter Detection 
 Struts Action	<input checked="" type="checkbox"/> Enabled	<input checked="" type="checkbox"/> Discover Transactions automatically for all Struts Action invocations Transactions will be named: ActionName.MethodName
 Web Service	<input checked="" type="checkbox"/> Enabled	<input checked="" type="checkbox"/> Discover Transactions automatically for all Web Service requests Transactions will be named: ServiceName.OperationName
 POJO	<input checked="" type="checkbox"/> Enabled	Any Java method can be the entry point for a Business Transaction. The class to which the method belongs to can be picked using different parameters like its name, its super class name, the interfaces it implements, or the annotations it has.
 Spring Bean	<input checked="" type="checkbox"/> Enabled	<input type="checkbox"/> Discover Transactions automatically for all Spring Bean invocations Transactions will be named: BeanName.MethodName
 EJB	<input checked="" type="checkbox"/> Enabled	<input type="checkbox"/> Discover Transactions automatically for all EJB invocations Transactions will be named: EJBName.MethodName

- **custom match rules**

Custom Match Rules
Configure Rules to identify transactions

custom rule names

	Type	Name	Priority	Enabled
	Servlet	ProductSearch	10	✗
	POJO	JavaTimer		✗
	POJO	JCronTab		✗
	POJO	Cron4J		✗
	POJO	ProductSearchPoj		✓
	POJO	Quartz		✗

- **exclude rules**

Exclude Rules
If a Transaction matches any of the following rules, it will not be discovered.

exclude rule names

	Type	Name	Enabled
	Servlet	Apache Axis Servlet	✓
	Servlet	Apache Axis2 Servlet	✓
	Servlet	Apache Axis2 Admin Servlet	✓
	Servlet	Struts Action Servlet	✓
	Servlet	Websphere web-services Servlet	✓
	Servlet	Websphere web-services axis Se	✓
	Servlet	JBoss web-services servlet	✓

To export use the HTTP GET method. The URI is the application (and optionally the tier) from which you are exporting the configuration. The configuration is exported to an xml file. If necessary, you can edit the xml file before you import it. For example, if you have exported the configuration of all the auto-detected entry-points and you do not want to import all of them, you can delete the ones you do not want from the file before you import it.

To import use the HTTP POST method. The URI is the application (and optionally the tier) to which you are importing the configuration. Use UTF-8 URL encoding of the URI before posting; for example, do not replace a space (" ") with "%20" in the URI.

For information about overwriting a configuration with the same name see [Overwrite Parameter](#).

Import and Export Auto-Detected Entry Point Configurations

You can import and export all your entry point configurations or one entry point configuration in a single request. Lists of multiple entry point names are not supported.

For requests that specify a single entry point configuration, the entry-point-type-name is the name displayed in the Type column of the Entry Points List in the Instrumentation->Transaction Detection tab in the AppDynamics console.

To import or export the configurations for all the auto-detected entry-points to or from an application

`http://<controller-host>:<controller-port>/controller/transactiondetection/<application-name>/auto`

For example:

```
http://op2.appdynamics.com:80/controller/transactiondetection/ACME
Book Store Application/auto
```

produces the output in [auto_app_all.xml](#).

To import or export the configuration for a single auto-detected entry point type to or from an application

`http://<controller-host>:<controller-port>/controller/transactiondetection/<application-name>/auto/<entry-point-type-name>`

For example:

```
http://op2.appdynamics.com:80/controller/transactiondetection/ACME
Book Store Application/auto/Servlet
```

produces the output in [auto_app_servlet.xml](#).

To import or export the configurations for all the auto-detected entry-points to or from a tier

`http://<controller-host>:<controller-port>/controller/transactiondetection/<application-name>/<tier-name>/auto`

For example:

```
http://op2.appdynamics.com:80/controller/transactiondetection/ACME
Book Store Application/ECommerce Server/auto/
```

produces the output in [auto_tier_all.xml](#).

To import or export the configuration for a single auto-detected entry point type to or from a tier

`http://<controller-host>:<controller-port>/controller/transactiondetection/<application-name>/<tier-name>/auto/<entry-point-type-name>`

For example:

```
http://op2.appdynamics.com:80/controller/transactiondetection/ACME
Book Store Application/ECommerce Server/auto/Servlet
```

produces the output in [auto_tier_servlet.xml](#).

Import and Export Custom Match and Exclude Rules

The URLs for both the import and export operations are identical.

To create the XML file do one of the following:

- EASY: Create the rule on a local controller and export it.

or

- DIFFICULT: Write the XML for the rule from scratch using a text editor.

To import the rule to the destination controller, use an HTTP POST operation attaching the XML file that describes the rule as attachment.

Import and Export Custom Match Rules

You can individual import and export custom match rules for servlet and POJO type entry points.

The custom-rule-name is the name displayed in the custom rule list in Instrumentation->Transaction Detection tab in the AppDynamics console.

To import or export a single custom match rule to or from an application

`http://<controller-host>:<controller-port>/controller/transactiondetection/<application-name>/<entry-point-type-name>/custom/<custom-rule-name>`

For example:

```
http://op2.appdynamics.com/controller/transactiondetection/ACME Book
Store Application/pojo/custom/JavaTimer
```

produces the output in [custom_app_single.xml](#).

To import or export a single custom match rule to or from a tier

`http://<controller-host>:<controller-port>/controller/transactiondetection/<application-name>/<tier-name>/<entry-point-type-name>/custom/<custom-rule-name>`

For example:

```
http://op2.appdynamics.com:80/controller/transactiondetection/ACME
Book Store Application/ECommerce Server/pojo/custom/Quartz
```

produces the output in [custom_tier_single.xml](#).

Import and Export Exclude Rules

You can import and export exclude rules for servlet and POJO type entry points.

The exclude-rule-name is the name displayed in the exclude rule list in the Instrumentation->Transaction Detection tab in the AppDynamics console.

To import or export a single exclude rule to or from an application

`http://<controller-host>:<controller-port>/controller/transactiondetection/<application-name>/<entry`

-point-type-name>/<exclude-rule-name>

For example:

```
http://op2.appdynamics.com:80/controller/transactiondetection/ACME
Book Store Application/exclude/servlet/Apache Axis Servlet
```

produces the output in [exclude_app_single](#).

To import or export a single exclude rule to or from a tier

http://<controller-host>:<controller-port>/controller/transactiondetection/<application-name>/<tier-name>/servlet/<exclude-rule-name>

For example:

```
http://op2.appdynamics.com:80/controller/transactiondetection/ACME
Book Store Application/ECommerce Server/exclude/servlet/Struts Action
Servlet
```

produces the output in [exclude_tier_single](#).

Overwrite Parameter

Use the overwrite parameter to overwrite a configuration of the same name. Without this parameter, if the import encounters a configuration for a component of the same name, the request will fail.

For example, to import a configuration for a POJO custom match rule named "JavaTimer" to an application that has an existing "JavaTimer" custom match rule use:

```
http://op2.appdynamics.com:80/controller/transactiondetection/ACME
Book Store Application/pojo/custom/JavaTimer?overwrite=true
```

The default is overwrite=false.

Learn More

- [Configure Business Transaction Detection](#)
- [Use the AppDynamics REST API](#)
- [Import and Export Health Rule Configurations](#)

Getter Chains in Java Configurations

- [Using Getter Chains](#)
- [Separators in Getter Chains](#)
- [Escaping Special Characters](#)
- [Getter Chain Examples](#)
- [Braces Enclosing Getter Chains](#)

- [Learn More](#)

This topic provides some guidance and examples of the correct syntax for using getter chains in AppDynamics configurations.

Using Getter Chains

You can use getter chains to:

- Create a new JMX Metric Rule and define metrics from MBean attributes. See [MBean Getter Chains and Support for Boolean and String Attributes](#).
- Configure method invocation data collectors. See [Configure Data Collectors#To use a getter chain to specify the data collection on method invocation](#).
- 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 [Identify Transactions Based on POJO Method Invoked by a Servlet](#).

Note: If a getter chain calls on a method that does a lot of processing, such as making numerous SQL calls, it can degrade the performance of the application and the App Agent for Java. Ideally, use getter chains only with simple MBean gets.

An example of a simple getter would be just getting a property from a bean, such as `getName()`.

```
public class MyBean
{
    private String name;
    public void setName(String name)
    { this.name = name; }
    public String getName()
    { return this.name; }
    public String getValue()
    { // Open up a database connection and run a big query // Process the
      result set performing some complex maths on the data // etc. return
      calculatedValue; }
}
```

Separators in Getter Chains

The following special characters are used as separators:

- comma (,) for separating parameters
- forward slash (/) for separating a type declaration from a value in a parameter
- Dot (.) for separating the methods and properties in the getter chain
- Dot (.) when representing "anything" must be escaped.

Escaping Special Characters

- If a slash or a comma character is used in a string parameter, use the backslash (\) escape character.

- If a literal dot (.) is used in a string parameter, use the backslash escape character before the dot. For example, a dot (.) when representing any character must be escaped using the backslash (\) escape character.

For example, in the following getter chain, both the backslash (\) and the dot (.) are escaped.

```
getHeader(hostid).split("\\\\\.")[1]
```

Getter Chain Examples

- Getter chain with integer parameters in the substring method using the forward slash as the type separator:

```
getAddress(appdynamics, sf).substring(int/0, int/10)
```

- Getter chain with various non-string parameter types:

```
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:

```
getUrl().split(\/) # node slash is escaped by a backward slash
```

- Getter chain with an array element:

```
getUrl().split(\/).[4]
```

- Getter chain with multiple array elements separated by commas:

```
getUrl().split(\/).[1,3]
```

- Getter chain retrieves property values, such as the length of an array:

```
getUrl().split(\.).length
```

- Getter chain using backslash to escape the dot in the string parameter; the call is `getParam (a.b.c)`.

```
getAddress().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).

```
getName(suze\\.smith).getClass().getSimpleName()
```

- The following getter chain is from a transaction splitting rule on URIs that use a semicolon as a delimiter; for example:

```
/my-webapp/xyz;jsessionid=BE7F31CC0235C796BF8C6DF3766A1D00?act=Add&uid=c42ab7ad-48a7-4353-bb11-0dfeabb798b5
```

The getter chain splits on the API name, so the resulting split transactions are "API.abc", API."xyz" and so on.

The call gets the URI using `getRequestURI()` and then splits it using the escaped forward slash. From the resulting array it takes the third entry (as the split treats the first slash as a separator) and inserts what before the slash (in this case, nothing) into the first entry. Then it splits this result using the semicolon, getting the first entry of the resulting array, which in this case contains the API name.

```
getRequestURI().split(\/).[2].split(;).[0]
```

Tip: When using `string.split()`, remember that it takes a regex and you have to escape any special regex characters.

For example, if you want to split on left square bracket ([):

```
Java syntax: split("[")
Getter chain syntax: split([)
```

Braces Enclosing Getter Chains

In most cases braces `{ }` are not used to enclose getter chains in AppDynamics configurations. An exception is the use of a getter chain in a custom expression on the `HttpRequest` object.

Custom expressions on the HTTP request are configurable 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. In these cases, braces are required to delineate the boundaries of the getter chains.

Business Transaction Match Rule - Servlet

Name

Enabled ☐

Priority

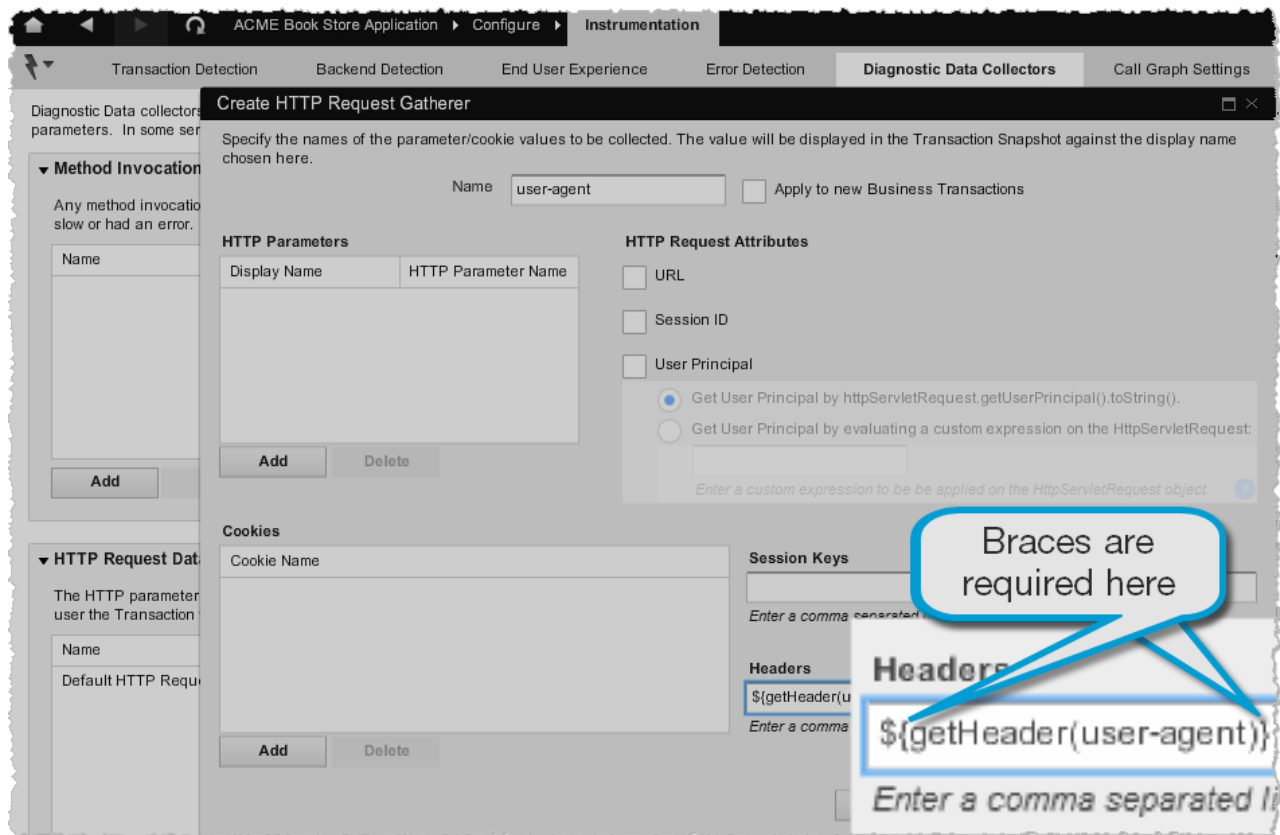
Transaction Match Criteria Split Transactions Using Request Data Split Transactions Using Payload

☐ Use the first segments in Transaction names
☐ Use the last segments in Transaction names
☐ Use URI segment(s) in Transaction names
 Segment Numbers *Enter a comma separated list of parameter numbers (e.g. 1,3,4)*
☐ Use a parameter value in Transaction names
 Parameter Name
☐ Use a header value in Transaction names
 Header Name
☐ Use a cookie value in Transaction names
 Cookie Name
☐ Use a session attribute value in Transaction names
 Session Attribute Key
☐ Use the request method (GET/POST/PUT) in Transaction names
☐ Use the request host Transaction in names
☐ Use the request originating address in Transaction names
☒ Apply a custom expression on the HttpServletRequest and use the result

Braces are required here

`${getRequestURI().split(V).[2].split(:).[0]}`

Getter chains in custom expressions on the HTTP request in diagnostic data collector should also be enclosed in braces:



Learn More

- [Configure Business Transaction Detection](#)
- [Configure Data Collectors](#)

Code Metric Information Points for Java

- [Code Metric Information Points for Java System Classes](#)
 - [To instrument a Java system class](#)
 - [Learn More](#)

Code Metric Information Points for Java System Classes

System classes like `java.lang.*` are by default excluded by AppDynamics. To enable instrumentation for a system class, use code metric information points.

The overhead of instrumenting Java system classes is based on the number of calls. AppDynamics recommends that you instrument only a small number of nodes and monitor the performance for these nodes before adding configuring all the 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 the metric.
2. Add the fully-qualified system class name to the override exclude section in the XML file. For example, to configure the `java.lang.Socket` class connect method, modify following element:

```
<override-system-exclude filter-type="equals" filter-value="java.lang.Socket"/>
```

3. Restart those JVMs for which you have modified the XML file.

Learn More

- [Code Metrics](#)
- [Configure Code Metric Information Points](#)

Configure JMX Metrics from MBeans

- [JMX Metric Rules and Metrics](#)
 - [Using the JMX Metric Rules Configuration Panel](#)
 - [Using the MBean Browser to Add an MBean Attribute](#)
 - [To create a metric from an MBean attribute in the MBean Browser](#)
- [Learn More](#)



Utilizing JMX Metrics in Troubleshooting

This topic describes how to create persistent JMX metrics from MBean attributes.

There are two ways to add MBean metrics:

- [Using the JMX Metric Rules Configuration Panel](#) for multiple attributes
- [Using the MBean Browser to Add an MBean Attribute](#) for specific attributes

For background information about creating JMX metrics see [Monitor JVMs](#) and [Monitor JMX MBeans](#).

JMX Metric Rules and Metrics

A JMX Metric Rule maps a set of MBean attributes from one or more MBeans into AppDynamics persistent metrics. You configure a metric rule that creates one or more metrics in the AppDynamics system. You may want to create new metrics if the preconfigured metrics do not provide sufficient visibility into the health of your system.

After the MBean attribute is configured to provide a persistent metric in AppDynamics, you can use

it to configure health rules. For details see [Health Rules](#).

To view the MBeans that are reporting currently in your managed environment use the [Metric Browser](#).

You can use the [JMX Metrics Rules Panel](#) or the [Using the MBean Browser](#) to create new metrics. MBean query expressions are supported.

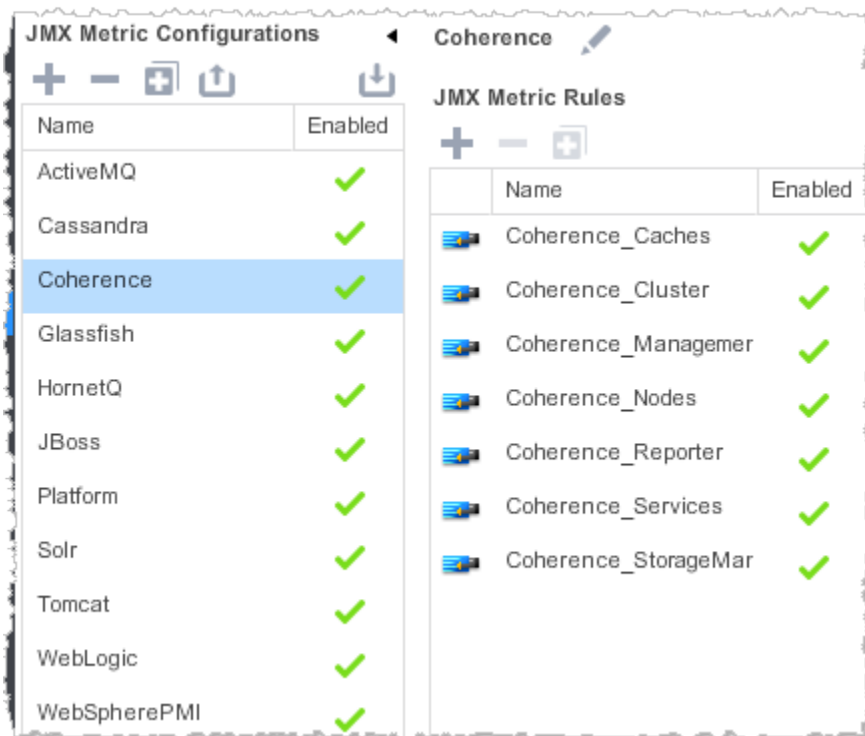
Required User Permissions

- In order to configure new JMX Metrics your user account must have "Configure JMX" permissions for the application.
For information about configuring user permissions for applications, see [To Configure the Default Application Permissions](#).

Using the JMX Metric Rules Configuration Panel

The JMX Metric Rules Panel is the best way to create metrics for multiple attributes based on the same MBean or for complex matching patterns.

- In the left navigation pane, click **Configure -> Instrumentation**.
- Click the **JMX** tab.
- In the **JMX Metric Configurations** panel, click the Java platform for which you are configuring metrics.



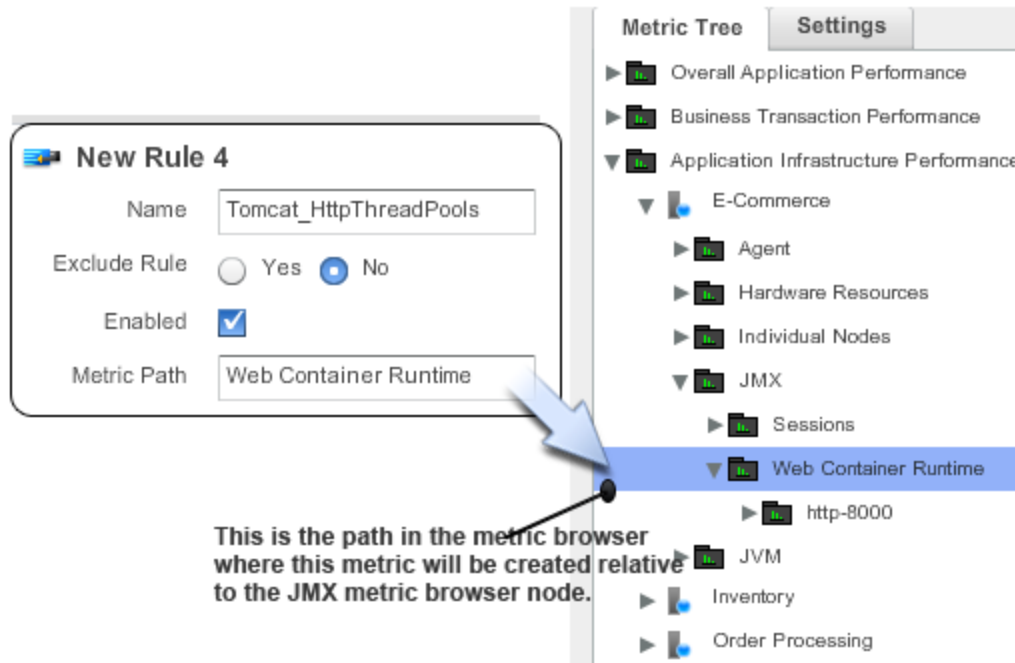
4. In the **JMX Metric Rules** panel, click the **Add** (the + icon). The **New Rule #** panel opens. The rule is given the next incremented number.

5. Provide the name and settings for this rule:

- The **Name** is the identifier you want to display in the UI.

- An **Exclude Rule** is used for excluding existing rules, so leave the default **No** option.
- **Enabled** means that you want this rule to run, so leave it selected.
- The **Metric Path** is the category as shown in the Metric Browser where the metrics will be displayed. A metric path groups the metrics and is relative to the Metric Browser node.

For example, the following screenshot displays how the JMX Metric Rule "Tomcat_HttpThreadPools" is defined for the ACME Online demo. The metric path is "Web Container Runtime", the category on Metric Browser where all metrics configured under the "Tomcat_HttpThreadPools" Metric Rule will be available.



6. 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 optional for more complex matching. Use one of the following:
 - any-substring
 - final-substring
 - equals
 - initial-substring
- Click **Add Condition**.

For example, the following screenshot displays the MBean matching criteria for the

"Tomcat_HTTPThreadPools" rule.

For all MBeans that match the preceding criteria, you can define one or more metrics for the attributes of those MBeans.

7. In the **Attributes** panel click **Add Attribute** to specify the MBean attributes.

8. Provide the name of the attribute and the metric name.

The metric name is used to represent the metric in the Metric Browser.

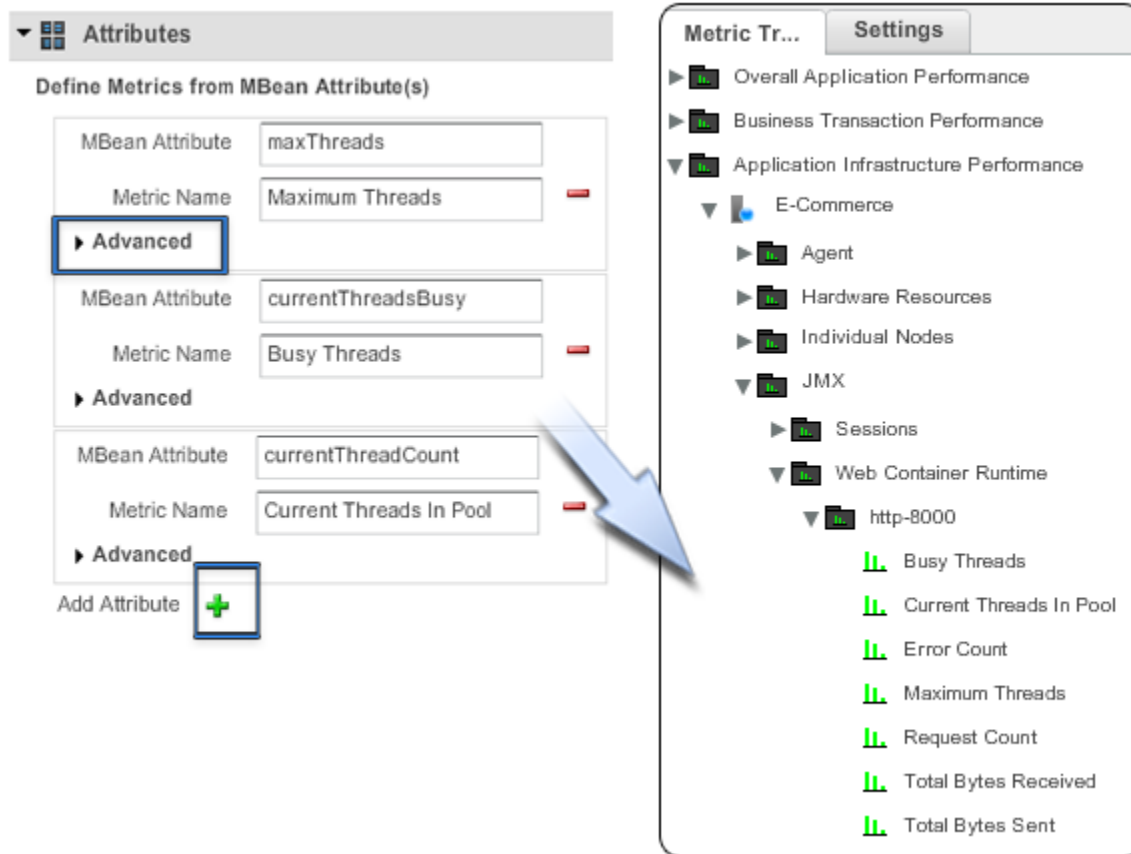
9. If needed, specify Advanced properties for the attribute.

- **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 and Support for Boolean and String Attributes](#).
- **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.

10. Click **Add Attribute** to define another metric from the same MBean.

11. Click **Create JMX Rule**.

The following screenshot shows how the MBean attributes configured for the Tomcat_HttpThreadPools rule will be displayed in the Metric Browser.

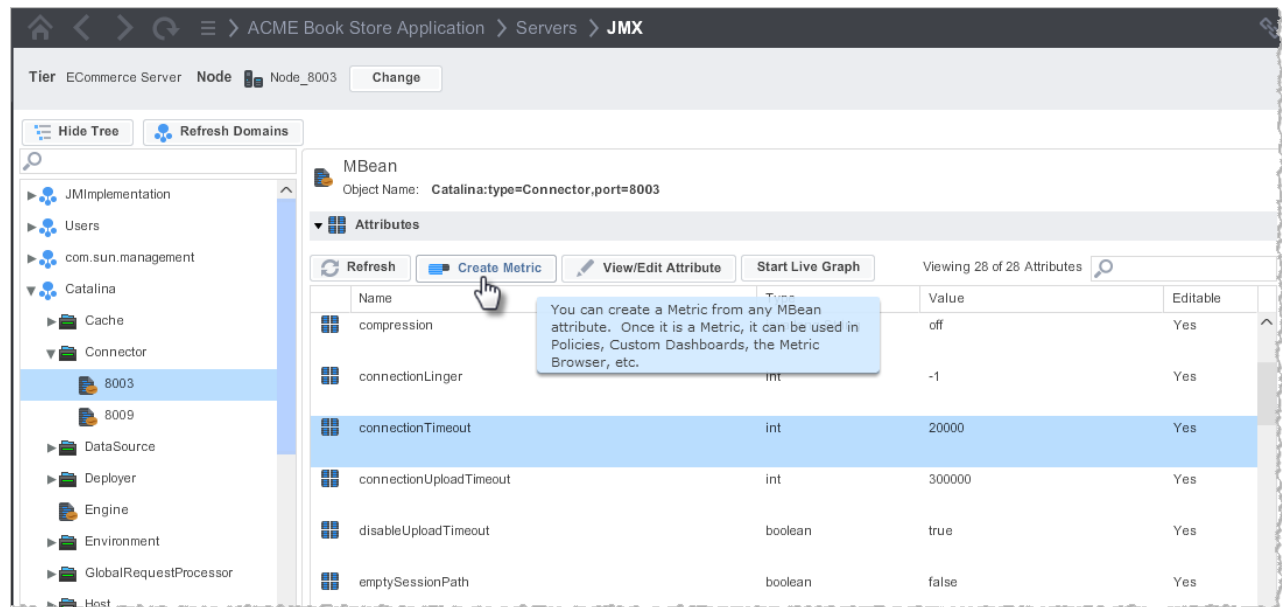


Using the MBean Browser to Add an MBean Attribute

You may know exactly which particular MBean attribute you want to monitor. You can select the attribute in the **MBean Browser** and create a **JMX Metric Rule** for it.

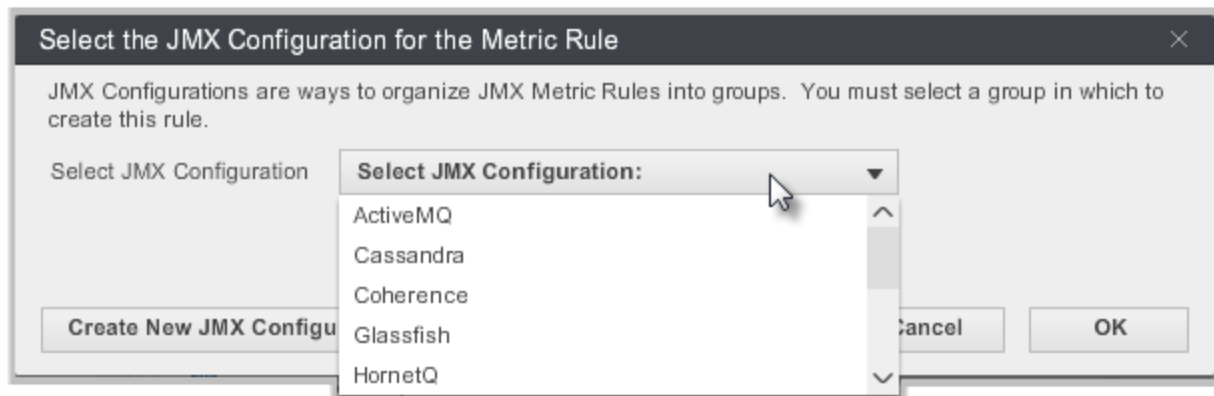
To create a metric from an MBean attribute in the MBean Browser

1. Navigate to the attribute and select it in the MBean Browser.



2. Click **Create Metric**.

3. In the **Select JMX Configuration for the Metric Rule** window, select the group in which to create the rule from the **Select JMX Configuration** pulldown. The categories that already exist on your system are listed. This example uses the Tomcat JMX configuration. Click **OK**.



Alternatively, you can create a new group with an name of your choosing. Click **Create New JMX Configuration** to make a new category. This is useful if you want to separate out the custom metrics from the out-of-the-box metrics.

The **Create JMX Metric Rule from an MBean** window opens.

Create JMX Metric Rule from an MBean

New Rule for MBean
Catalina:Catalina:type=Connector,port=8000

Name: New Rule for MBean Catalina:Catalina:type=Connector,port=8000

Exclude Rule: ☐ Yes ☒ No

Enabled: ☒

Metric Path:
This is the path in the metric browser where this metric will be created relative to the JMX metric browser node.

MBeans

MBean Matching Criteria

Domain: Catalina

Object Name Match Pattern: Catalina:type=Connector,port=8000

► Advanced MBean Matching

Attributes

Define Metrics from MBean Attribute(s)

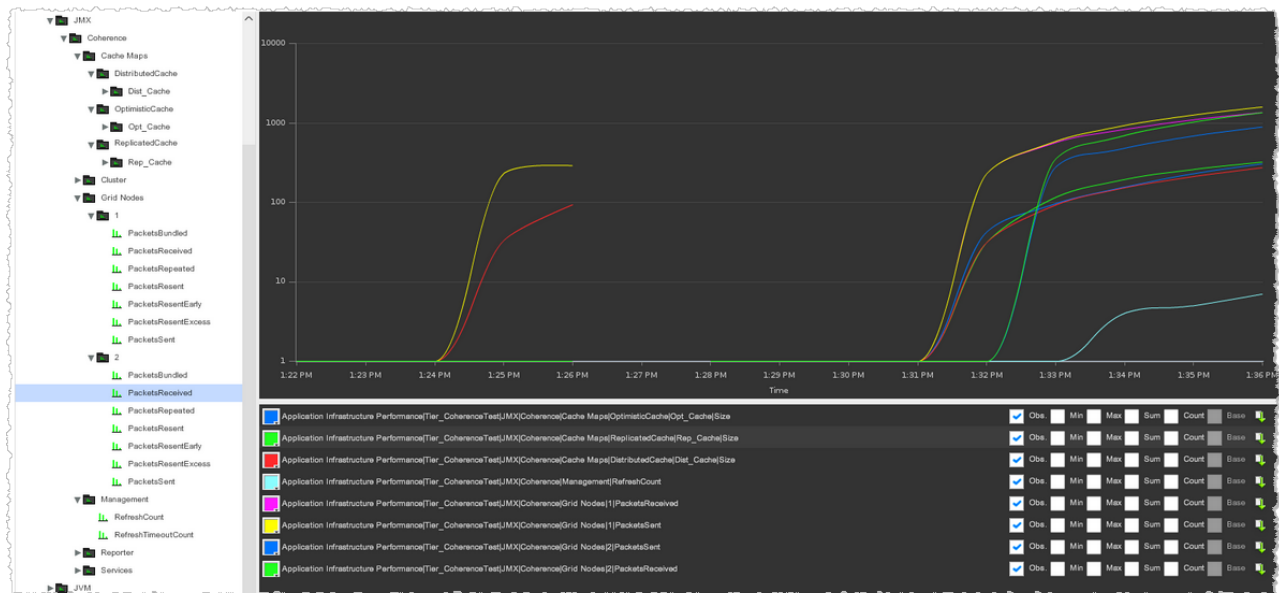
MBean Attribute	Metric Name
connectionTimeout	connectionTimeout

► Advanced

Add Attribute +

Cancel Create JMX Rule

4. Supply the **Metric Path**, the category as shown in the Metric Browser where the metrics will be displayed. For more discussion of the Metric Path see Step 5 of the previous section.
5. Review the **MBean Matching Criteria** and modify it as needed. Since this is the MBean you selected, you probably do not need to change it.
6. Review the **MBean Attribute** and **Metric Name**. This is the MBean attribute you originally selected. By default the name is the same as the attribute. You can change it if you want to be more specific about its use.
7. Review the **Advanced** panel rollup criteria and update as needed. For more information about these options see Step 9 of the previous section.
8. Click **Add Attribute** to define another metric from the same MBean.
9. Click **Create JMX Rule**. The new metric displays in the **JMX Metric Browser**.



Learn More

- [Monitor JVMs](#)
- [Monitor JMX MBeans](#)
- [Exclude JMX Metrics](#)

Create, Import or Export JMX Metric Configurations

- [To create a New JMX Configuration](#)
- [To import a JMX metrics configuration](#)
- [To export a JMX metrics configuration](#)
- [To use a pre-3.3 configuration file for JMX metrics](#)

This topic describes how to import or export your existing JMX configurations or create new JMX configurations.

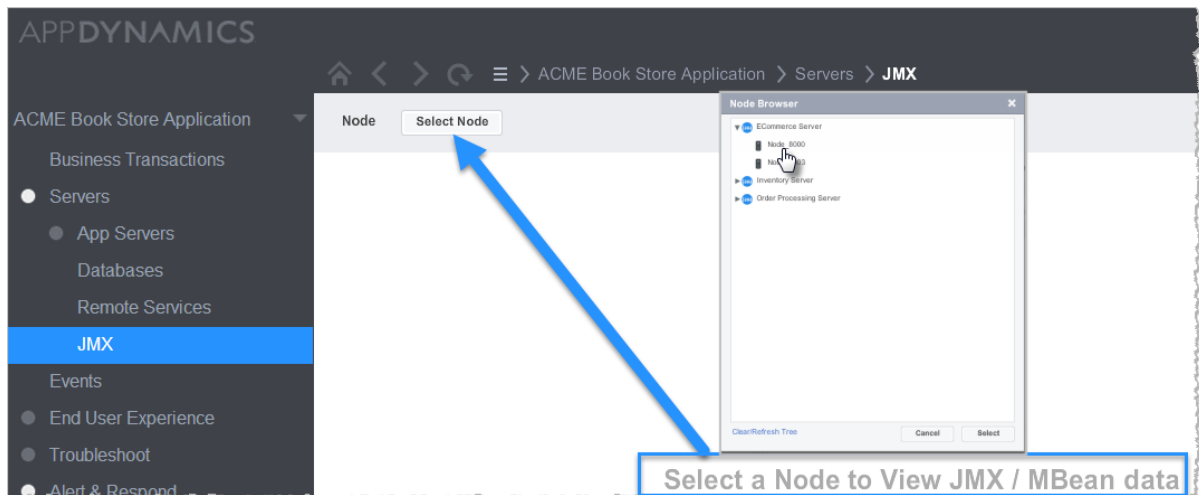
Prerequisite for Configuring JMX Metric Rules

- In order to configure JMX metric rules, your user account must have "Configure JMX" permissions for the application. For information about configuring user permissions for applications, see [To Configure the Default Application Permissions](#).

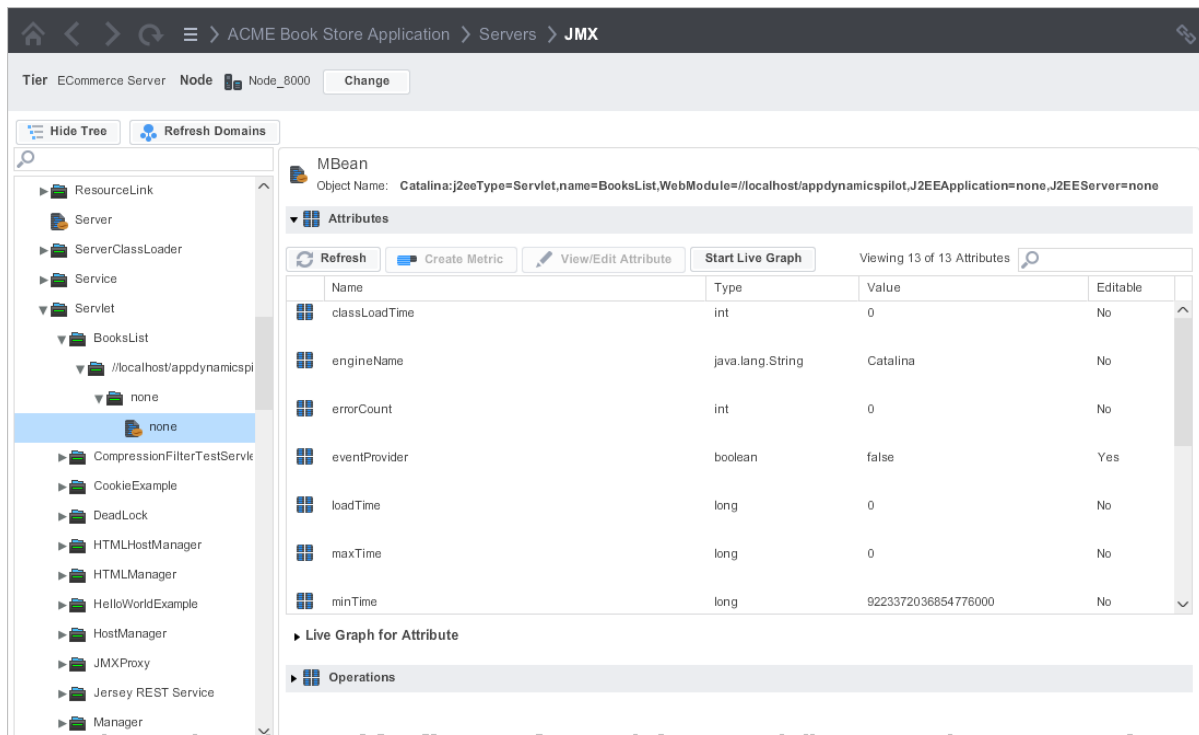
To create a New JMX Configuration

This is useful if you want to separate out the custom metrics from the out-of-the-box metrics.

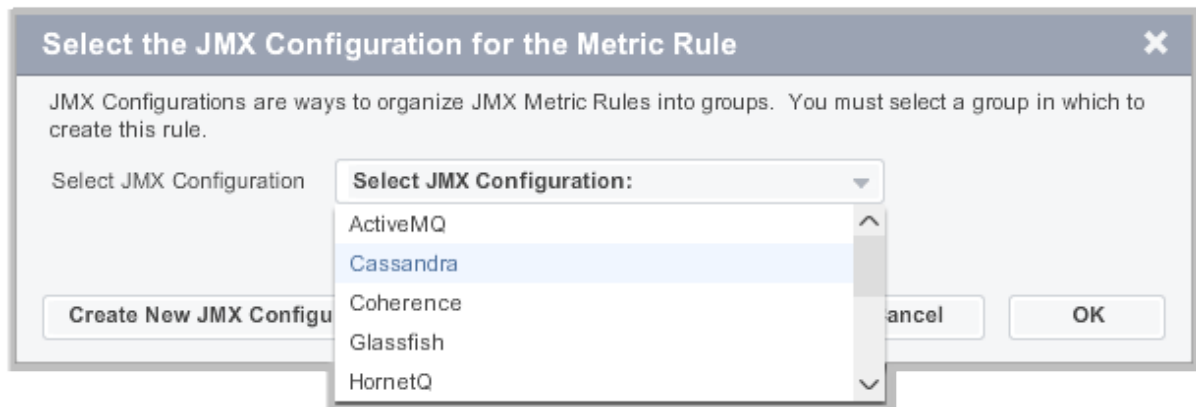
1. In the left navigation menu, click **Servers -> JMX**.



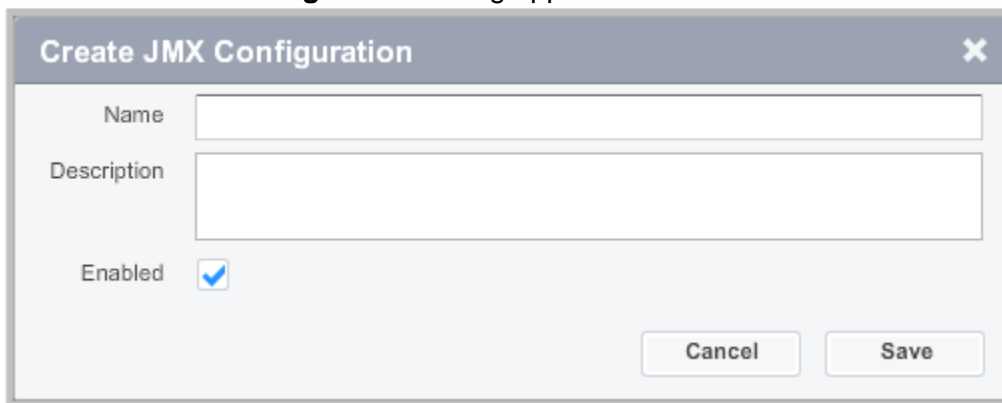
2. Click **Select Node**, in the **Node Browser** click the node, and then click **Select**. All the Domains for the node appear.
3. In the Domain tree, expand the domains to locate the MBean you want to use as the basis for a JMX metric.



4. Click the MBean attribute you want to work with, and then click **Create New JMX Configuration**.



The **Create JMX Configuration** dialog appears.



5. Enter the **Name** and **Description** for the JMX configuration, and click **Enabled** if you want the new configuration to be immediately accessible and usable.

To import a JMX metrics configuration

1. Click **Configure -> Instrumentation**.
2. Click the **JMX** tab.
3. Click the **Import JMX Configuration** icon.



4. In the JMX Configuration Import screen, click **Select JMX Config. File** and select the XML configuration file for your JMX metrics.



5. Click **Import**.

To export a JMX metrics configuration

1. Click **Configure -> Instrumentation**.
2. Click the **JMX** tab.
3. Click the **Export JMX Configuration** icon.

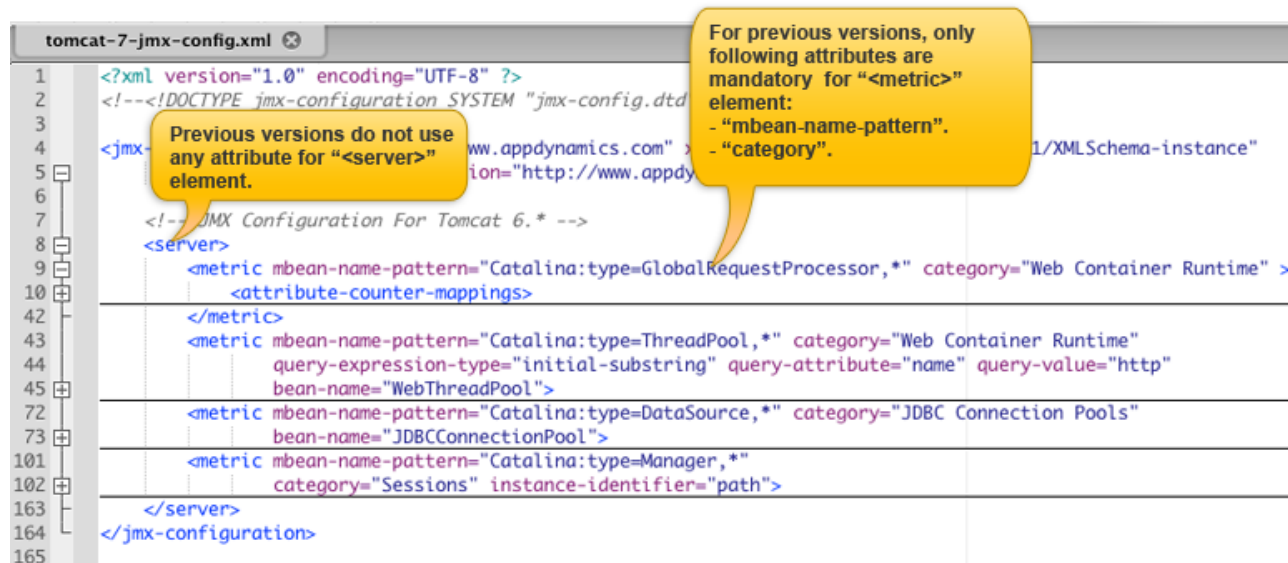


The configuration is downloaded as an XML file.

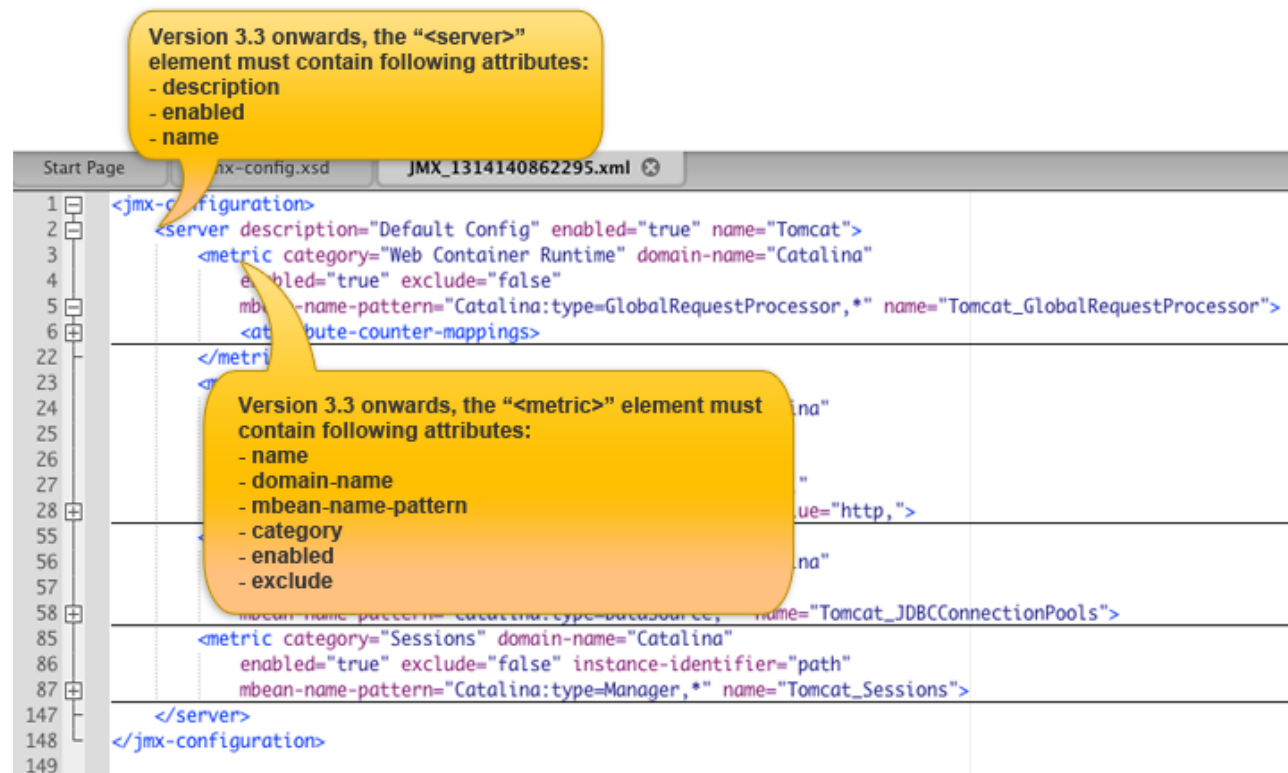
To use a pre-3.3 configuration file for JMX metrics

In AppDynamics Pro Version 3.3 the structure of the XML-based configuration file for JMX metrics changed. As a result, if you use a pre-3.3 version, you must provide additional attributes in the configuration file.

The following screenshot shows a sample XML configuration file for the JMX metrics for pre-3.3 versions of AppDynamics:



Beginning with AppDynamics version 3.3, the structure for this XML file is the following:



To be able to import your existing configurations for JMX metrics, add the following attributes for <server> element and **each** of the <metric> elements in your XML file.

The attributes for each element are listed below:

Attributes for the <server> element

Attribute Name	Attribute Type	Allowed Values	Mandatory/Optional
description	String		Mandatory
enabled	Boolean	true/false	Mandatory
name	String		Mandatory

Attributes for each <metric> element

For each <metric> element, add the mandatory attributes from the following list to your existing configuration file:

Attribute Name	Attribute Type	Allowed Values	Mandatory/Optional
name	String		Mandatory
domain-name	String	true/false	Mandatory
mbean-name-pattern	String		Mandatory

category	String		Mandatory <i>All those <metric> elements that have same value for this attribute will be grouped together on the metric browser.</i>
enabled	Boolean	true/false	Mandatory
exclude	Boolean	true/false	Mandatory
bean-name	String		Optional
query-attribute	String		Optional
query-expression-type		Use any one from the following values: <ul style="list-style-type: none"> • any-substring • final-substring • equals • initial-substring 	Optional
query-value	String		Optional
instance-identifier	String		Optional
instance-name	String		Optional

Exclude JMX Metrics

- [Tuning What Metrics are Gathered](#)
 - [To exclude a metric](#)
- [Learn More](#)

This topic describes how to exclude MBean attributes from being monitored as JMX metrics.

For background information about JMX metrics see [Monitor JVMs](#) and [Monitor JMX MBeans](#).

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.

To 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 causes AppDynamics to stop monitoring metrics for HTTP Thread Pools.

Later, if you want to see all HTTP Thread Pool metrics, clear the Enabled checkbox to disable the rule.

Learn More

- [Monitor JVMs](#)
- [Monitor JMX MBeans](#)
- [Configure JMX Metrics from MBeans](#)

Exclude MBean Attributes

- [Excluding MBean Attributes from the MBean Browser](#)
 - [To exclude an MBean attribute](#)
- [Learn More](#)

Excluding MBean Attributes from the MBean Browser

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

1. Open the `AppServerAgent/conf/app-agent-config.xml` file.
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`.

```
<property name="agent-overwrite" value="true"/>
```

3. Locate the `JMXService` section. It looks like this:

```
<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.

```
<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>
```

4. Save the file.

Learn More

- [App Agent for Java Directory Structure](#)

Configure JMX Without Transaction Monitoring

- [Collect Metrics without Transaction Monitoring](#)
 - [To turn off transaction detection](#)
- [Learn More](#)

Collect Metrics without Transaction Monitoring

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 this by turning off transaction detection at the entry point.

To turn off transaction detection

1. In the left navigation panel, click **Configure -> Instrumentation**.
2. In the **Select Application or Tier** panel, select the application.
3. In the right panel, click **Use Custom Configuration for this Tier**.
4. Expand the **Entry Points** list if it is not already expanded.
5. Clear all the relevant **Enabled** checkboxes in the **Transaction Monitoring** column.

Transaction monitoring on all selected entry points in the application is disabled. Exit point detection remains enabled.

Learn More

- [Configure Business Transaction Detection](#)

Resolve JMX Configuration Issues

- [Unable to browse MBeans on WebSphere Application Server \(WAS\)](#).

- Unable to get metrics from the App Agent for Java App on a GlassFish server
 - Unable to get JMX metrics for database connections on GlassFish

[Learn More](#)

This topic describes how to resolve issues that may prevent AppDynamics from properly reporting JMX MBean metrics.

Unable to browse MBeans on WebSphere Application Server (WAS).

In certain situations, you may encounter the following exception in the agent.log file for App Agent for Java App deployed on the WebSphere Application Server (WAS).

```
[AD Thread-Transient Event Channel Poller0] 17 Aug 2011 08:14:08,031
ERROR JMXTransientOperationsHandler - Error trying to lookup clz -
java.lang.ClassNotFoundException: com.ibm.ws.security.core.SecurityContext
```

To resolve this issue:

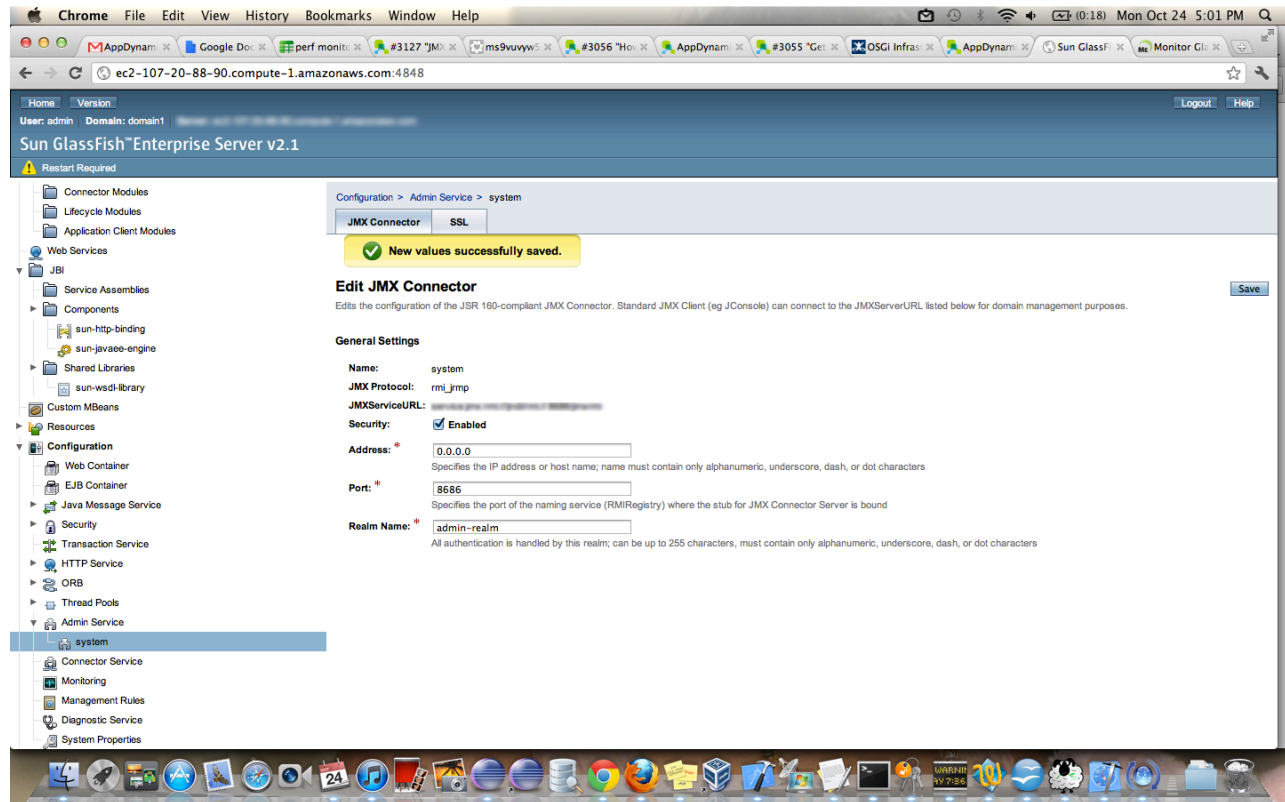
1. From the WAS administration console, navigate to the JVM settings for the server of interest: **Application servers -> <server> -> Process Definition -> Java Virtual Machine**.
2. Remove the following setting from the generic JVM settings:

```
-Djavax.management.builder.initial = -Dcom.sun.management.jmxremote
```

Unable to get metrics from the App Agent for Java App on a GlassFish server

Under some situations JMX metrics from GlassFish are not reported. Also some metrics may not be enabled by default. Try these solutions:

- 1) Confirm that JMX monitoring is enabled in the GlassFish server. Refer to the following screenshot:



2) Copy the text below into an mbean-servers.xml file in the following directory:

```
<App_Agent_Dir>/conf/jmx/
```

```
<?xml version="1.0" encoding="UTF-8" ?>
<!--<!DOCTYPE servers SYSTEM "mbean-servers.dtd" -->

<servers xmlns="http://www.appdynamics.com"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.appdynamics.com
mbean-servers.xsd">
  <!--
  <server mbean-server-name="WebSphere"
mbean-name-pattern="WebSphere:*,type=Server,j2eeType=J2EEServer"
version-attribute="platformVersion" version-startsWith="7"
config-file="servers/websphere-7-jmx-config.xml" />

  <server mbean-server-name="WebSphere"
mbean-name-pattern="WebSphere:*,type=Server,j2eeType=J2EEServer"
version-attribute="platformVersion" version-startsWith="6"
config-file="servers/websphere-7-jmx-config.xml" />
  -->
  <server mbean-server-name="WebSphere"
mbean-name-pattern="WebSphere:*,type=Server"
config-file="servers/websphere-7-jmx-config.xml" />
  <server mbean-server-name="JBoss_4"
mbean-name-pattern="jboss.management.local:j2eeType=J2EEServer,name=Local"
```



```

version-attribute="serverVersion" version-startsWith="4"
config-file="servers/jboss-4-jmx-config.xml" />
    <server mbean-server-name="JBoss_5"
mbean-name-pattern="jboss.management.local:j2eeType=J2EEServer,name=Local"
version-attribute="serverVersion" version-startsWith="5"
config-file="servers/jboss-5-jmx-config.xml" />
    <server mbean-server-name="JBoss_6"
mbean-name-pattern="jboss.management.local:j2eeType=J2EEServer,name=Local"
version-attribute="serverVersion" version-startsWith="6"
config-file="servers/jboss-5-jmx-config.xml" />
    <server mbean-server-name="Tomcat_5.5"
mbean-name-pattern="Catalina:type=Server" version-attribute="serverInfo"
version-startsWith="Apache Tomcat/5.5"
config-file="servers/tomcat-5-jmx-config.xml" />
    <server mbean-server-name="Tomcat_6.0"
mbean-name-pattern="Catalina:type=Server" version-attribute="serverInfo"
version-startsWith="Apache Tomcat/6.0"
config-file="servers/tomcat-6-jmx-config.xml" />
    <server mbean-server-name="Tomcat_7"
mbean-name-pattern="Catalina:type=Server" version-attribute="serverInfo"
version-startsWith="Apache Tomcat/7"
config-file="servers/tomcat-7-jmx-config.xml" />
    <server mbean-server-name="Sun GlassFish_2.1"
mbean-name-pattern="com.sun.appserv:j2eeType=J2EEServer,name=server,category=run
time" config-file="servers/glassfish-v2-jmx-config.xml" />
    <server mbean-server-name="WebLogic_10"
mbean-server-lookup-string="java:comp/jmx/runtime"
mbean-name-pattern="com.bea:*,Type=ServerRuntime"
version-attribute="WeblogicVersion" version-startsWith="WebLogic Server 10"
config-file="servers/weblogic-10-jmx-config.xml" />
    <server mbean-server-name="WebLogic_9"
mbean-server-lookup-string="java:comp/jmx/runtime"
mbean-name-pattern="com.bea:*,Type=ServerRuntime"
version-attribute="WeblogicVersion" version-startsWith="WebLogic Server 9"
config-file="servers/weblogic-9-jmx-config.xml" />
    <server mbean-server-name="ActiveMQ_5.3.2"
mbean-name-pattern="org.apache.activemq:*"
config-file="servers/activemq-5.3.2-jmx-config.xml" />
    <server mbean-server-name="Apache Solr 1.4.1" mbean-name-pattern="solr:*"
config-file="servers\solr-1.4.1-jmx-config.xml" />
    <server mbean-server-name="Apache Cassandra 0.7.0"
mbean-name-pattern="org.apache.cassandra.net:*"
config-file="servers\cassandra-0.7.0-jmx-config.xml" />
    <server mbean-server-name="Apache Cassandra 0.7.0"
mbean-name-pattern="org.apache.cassandra.db:*"
config-file="servers\cassandra-0.7.0-jmx-config.xml" />
    <server mbean-server-name="Apache Cassandra 0.7.0"
mbean-name-pattern="org.apache.cassandra.request:*"
config-file="servers\cassandra-0.7.0-jmx-config.xml" />
    <server mbean-server-name="Apache Cassandra 0.7.0"
mbean-name-pattern="org.apache.cassandra.internal:*"
config-file="servers\cassandra-0.7.0-jmx-config.xml" />
    <!-- If you are using Platform MBean server to report activemq metrics then
you may uncomment the following line.
-->
<!--

```

```

    <server mbean-server-name="Platform"
mbean-name-pattern="org.apache.activemq:*"
config-file="servers\activemq-5.3.2-jmx-config.xml" />
-->

    <!-- If your app publishes custom jmx metrics to platform jmx server then
you may modify the platform-jmx-config.xml
    and update the mbean-name-pattern in the following line to start recording
your metrics by appdynamics agent
-->

    <!--
    <server mbean-server-name="Platform" mbean-name-pattern="com.foo.myjmx:*"
config-file="servers\platform-jmx-config.xml" />
-->

```

```
</servers>
```

You should see a new JMX node in the metrics tree.

Unable to get JMX metrics for database connections on GlassFish

JDBC connection pool metrics are not configured out-of-the-box for GlassFish. To configure them, uncomment the JDBC connection pool section and provide the relevant information in the following file:

```
<app_agent_install>/conf/jmx/servers/glassfish-v2-jmx-config.xml
```

Uncomment the following section and follow the instructions provided in the file.

```

<!-- The following config can be uncommented to monitor glassfish JDBC
connection pool. Please set the name of the connection
pool (not the datasource name) and enable monitoring for the JDBC Pools on
glassfish admin console. -->
<!--
<metric
mbean-name-pattern="com.sun.appserv:type=jdbc-connection-pool,category=monitor,n
ame=<set the name of pool>,*"
category="JDBC Connection Pools">
<attribute-counter-mappings>
<attribute-counter-mapping>
<attribute-name>numconnused-current</attribute-name>
<counter-name>Connections In Use</counter-name>
<counter-type>average</counter-type>
<time-rollup-type>average</time-rollup-type>
<cluster-rollup-type>individual</cluster-rollup-type>
</attribute-counter-mapping>
<attribute-counter-mapping>
<attribute-name>numconnused-highwatermark</attribute-name>
<counter-name>Max Connections Used</counter-name>
<counter-type>observation</counter-type>
<time-rollup-type>average</time-rollup-type>
<cluster-rollup-type>individual</cluster-rollup-type>
</attribute-counter-mapping>
<attribute-counter-mapping>
<attribute-name>numpotentialconnleak-count</attribute-name>
<counter-name>Potential Leaks</counter-name>
<counter-type>observation</counter-type>
<time-rollup-type>average</time-rollup-type>
<cluster-rollup-type>individual</cluster-rollup-type>
</attribute-counter-mapping>
<attribute-counter-mapping>
<attribute-name>averageconnwaittime-count</attribute-name>
<counter-name>Avg Wait Time Millis</counter-name>
<counter-type>observation</counter-type>
<time-rollup-type>average</time-rollup-type>
<cluster-rollup-type>individual</cluster-rollup-type>
</attribute-counter-mapping>
<attribute-counter-mapping>
<attribute-name>waitqueuelength-count</attribute-name>
<counter-name>Current Wait Queue Length</counter-name>
<counter-type>observation</counter-type>
<time-rollup-type>average</time-rollup-type>
<cluster-rollup-type>individual</cluster-rollup-type>
</attribute-counter-mapping>
</attribute-counter-mappings>
</metric>

```

Learn More

- [IBM WebSphere and InfoSphere Startup Settings](#)
- [GlassFish Startup Settings](#)

MBean Getter Chains and Support for Boolean and String Attributes

- i** In addition to getter chain support for numeric boxed primitives (Short, Integer, Long, etc.), Strings and Booleans are supported using implicit conversion. Expressions can be executed against any value.

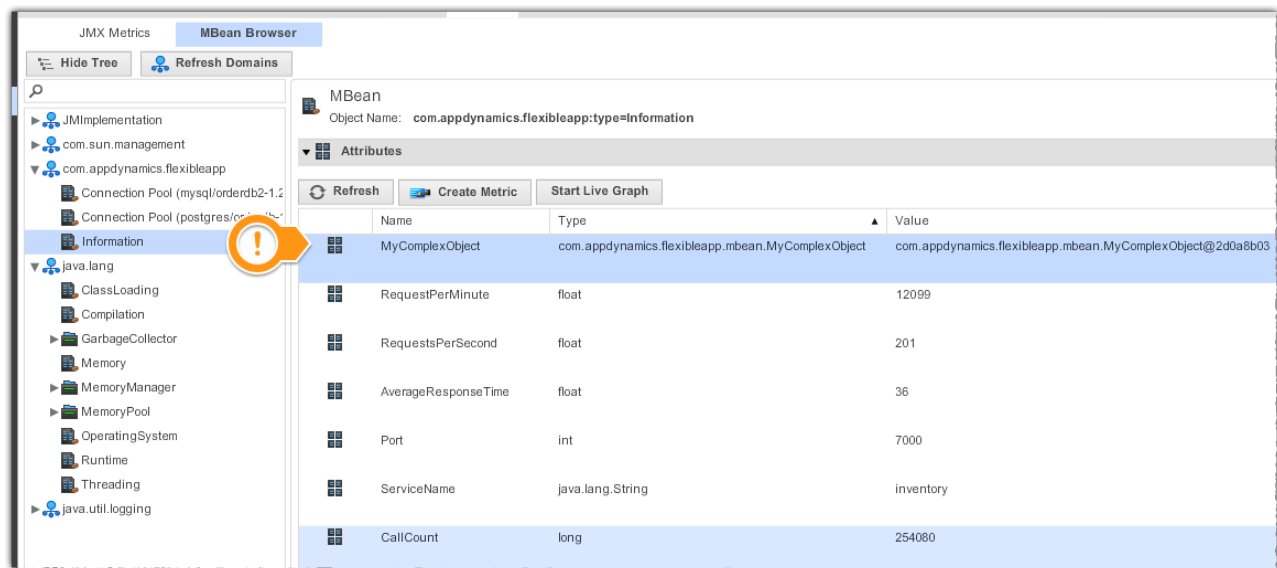
Prior to 3.7.7, AppDynamics supported Integer MBean attributes. In 3.7.7 support was added for Boolean and String attributes using the new getter chain field that implicitly converts the Boolean or String to an Integer. Booleans are automatically converted to 0 (false) and 1 (true). Strings are converted to numeric values.

Example Metric Getter Chain

1. Given the following MBean code:

```
1 package com.appdynamics.flexibleapp.mbean;
2
3 /**
4  * User: [REDACTED]
5  * Date: 10/23/12
6  * Time: 4:06 PM
7  */
8 public class MyComplexObject
9 {
10     public long getSomething()
11     {
12         return System.currentTimeMillis();
13     }
14
15     public AnotherComplexObject getSecond()
16     {
17         return new AnotherComplexObject();
18     }
19 }
20
```

2. Locate the MBean in the MBean Browser:



The screenshot shows the AppDynamics MBean Browser interface. On the left, a tree view displays the hierarchy of MBeans, with 'Information' selected under 'com.appdynamics.flexibleapp'. An orange callout bubble with an exclamation mark points to the 'Information' MBean. The main panel shows the 'Attributes' of the selected MBean, with a table listing various metrics and their values.

Name	Type	Value
MyComplexObject	com.appdynamics.flexibleapp.mbean.MyComplexObject	com.appdynamics.flexibleapp.mbean.MyComplexObject@2d0a8b03
RequestPerMinute	float	12099
RequestsPerSecond	float	201
AverageResponseTime	float	36
Port	int	7000
ServiceName	java.lang.String	inventory
CallCount	long	254080

3. Create a new JMX Metric Rule and add the getter chain information in new Metric Getter Chain field:

Create JMX Metric Rule from an MBean

New Rule for MBean
com.appdynamics.flexibleapp:com.appdyna
namics.flexibleapp:type=Information

Name: New Rule for MBean com.appdynamics.flexibleapp:com.appdyna

Exclude Rule: ☐ Yes ☒ No

Enabled: ☒

Metric Path: jmx
This is the path in the metric browser where this metric will be created relative to the JMX metric browser node.

MBeans

MBean Matching Criteria

Domain: com.appdynamics.flexibleapp

Object Name Match Pattern: com.appdynamics.flexibleapp:type=Information

► Advanced MBean Matching

Attributes

Define Metrics from MBean Attribute(s)

MBean Attribute: MyComplexObject

Metric Name: MyComplexObject

▼ **Advanced**

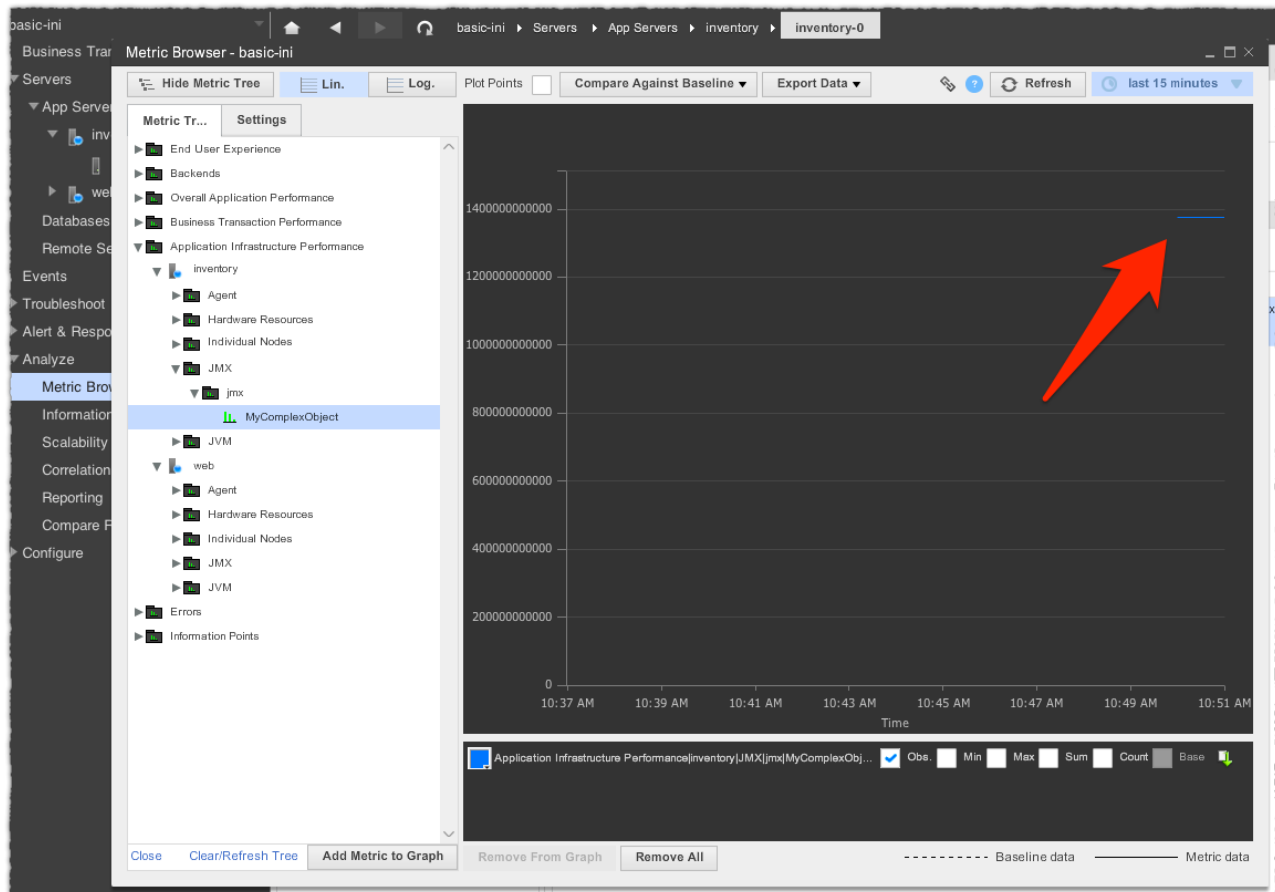
Metric Getter Chain: getSomething()

Metric Time Rollup: Average the data points
Define how this metric is aggregated over time (for example, from 60 one min data points to one 1 hour data point)

Metric Cluster Rollup: Average Data from all Nodes
Define how this metric is aggregated for a Tier from its Nodes (for example average response time for a Tier using the avp values from all of its individual nodes)

Add Metric Getter Chain

4. The new metric shows up in the Metric Tree:



Learn More

- [Getter Chains in Java Configurations](#)

Percentile Metrics

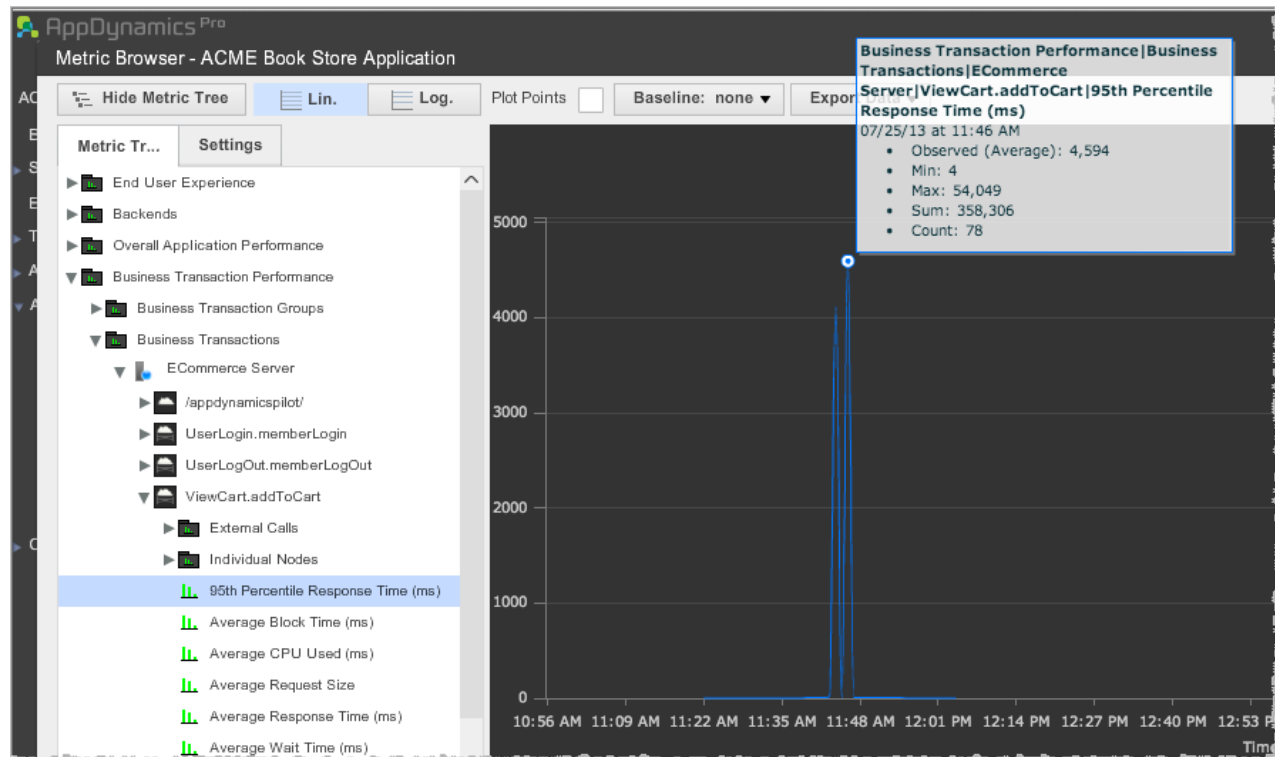
Demo of Percentile Metrics

Percentile metrics are available for business transaction response time. Percentiles are generally a better indicator than averages because they are not sensitive to outliers while averages can get distorted. Percentile metrics can provide a sense of how the response times are distributed. From this information you can extrapolate the percentage of responses that are within and outside of acceptable ranges.

Percentiles describe distributions of real world data sets in ways that are less sensitive to the effect of outliers in the data set than simpler calculations such as the mean. Percentile metrics can provide you with answers to questions such as the following:

- What is the 95th percentile latency of transactions for a single web server?
- What is the 95th percentile latency of transactions for the entire web site (over all the servers)?
- What is the 95th percentile latency of transactions for the website during the last one hour?

Percentile metrics for business transaction response time are displayed in the Metric Browser or Custom Dashboard.



Percentile metrics can be interpreted as follows: the 95th percentile point indicates that 95% of all response times were less than the metric value. It provides a sense of how the response times are distributed. For example if the metric Observed (Average) or percentile value is 300ms, it implies that 95% of the business transaction response times are less than 300ms, and therefore implies that 95% of the business transaction response times are within acceptable ranges. It also implies that 5% of the requests have been taking more time and could be a cause for concern.

Enabling Percentile Metrics

There are three agent node properties that you can use to enable and configure how percentile metrics are collected. You do not need to restart the agent after making changes to these properties.

- **disable-percentile-metrics:** Set this property to 'false' to view the percentile metrics.
- **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 for the machine where the agent is running, but may offer better percentiles depending on how the response times are distributed.
- **percentiles-to-report:** By default, the system will capture the 95th percentile metrics. You can change the percentile captured here.

Learn More

[Metric Browser](#)

[Business Metrics](#)

[App Agent Node Properties Reference](#)

[App Agent Node Properties](#)

Monitor Java Applications

Monitor JVMs

- [Infrastructure Monitoring in a Java Environment](#)
- [JVM Key Performance Indicators](#)
 - [Memory Usage and Garbage Collection](#)
 - [To view heap usage, garbage collection, and memory pools](#)
 - [Heap Usage](#)
 - [Garbage Collection](#)
 - [Memory Pools](#)
 - [Classes, Garbage Collection, Memory, Threads, and Process CPU Usage Metrics](#)
 - [To view classes, garbage collection, memory, threads, and process CPU usage metrics](#)
- [Alert for JVM Health](#)
- [Monitor JVM Configuration Changes](#)
- [Detect Memory Leaks](#)
 - [Automatic Leak Detection](#)
 - [To enable automatic leak detection](#)
- [Detect Memory Thrash](#)
 - [Object Instance Tracking](#)
 - [To monitor Java object instances](#)
- [Monitor Long-lived Collections](#)
 - [To view or configure custom memory structures](#)
- [Learn More](#)



JVM/container configuration can often be a root cause for slow performance because not enough resources are available to the application.

Infrastructure Monitoring in a Java Environment

A Java application environment has multiple functional subsystems. These are usually instrumented using JMX (Java Management Extensions) or IBM Performance Monitoring Infrastructure (PMI). AppDynamics automatically discovers JMX and PMI attributes.

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.

For more information about JMX, see the [JMX overview and tutorial](#). To learn about PMI see [Writing PMI Applications Using the JMX Interface](#).

JVM Key Performance Indicators

There are often thousands of attributes, however, you may not need to know about all of them. By default, AppDynamics monitors the attributes that most clearly represent key performance indicators and provide useful information about short and long term trends. The preconfigured JVM metrics include:

- Total classes loaded and how many are currently loaded
- Thread usage
- Percent CPU process usage
- On a per-node basis:
 - Heap usage
 - Garbage collection
 - Memory pools and caching
 - Java object instances

You can configure additional monitoring for:

- Automatic leak detection
- Custom memory structures

Memory Usage and Garbage Collection

Monitoring garbage collection and memory usage can help you identify memory leaks or memory thrash that can have a negative impact application performance.

Important Information

The agent cannot capture memory statistics if the application is configured to use G1GC (-XX:+UseG1GC) and using JDK version = 1.6.0_26-b03. You should either remove -XX:+UseG1GC from the application startup options or upgrade the JDK to version 1.6.0_32 or higher.

To view heap usage, garbage collection, and memory pools

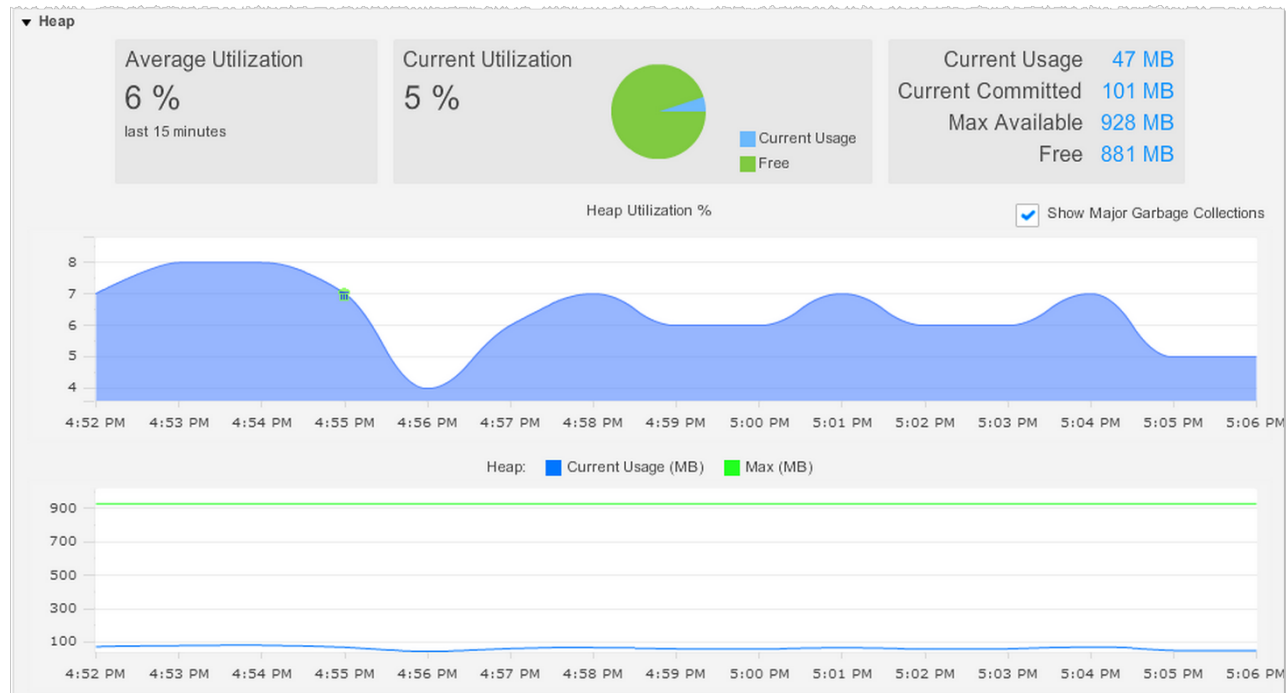
1. In the left navigation pane, click **Servers - > App Servers -> <tier> -> <node>**. The Node

- Dashboard opens.
2. On the Node Dashboard, click the Memory tab.
3. On the Memory tab, click the Heap & Garbage Collection subtab. The panels show data about the current usage.

See [Node Dashboard](#) for a complete description of this dashboard.

Heap Usage

The Heap panel shows data about the current usage.

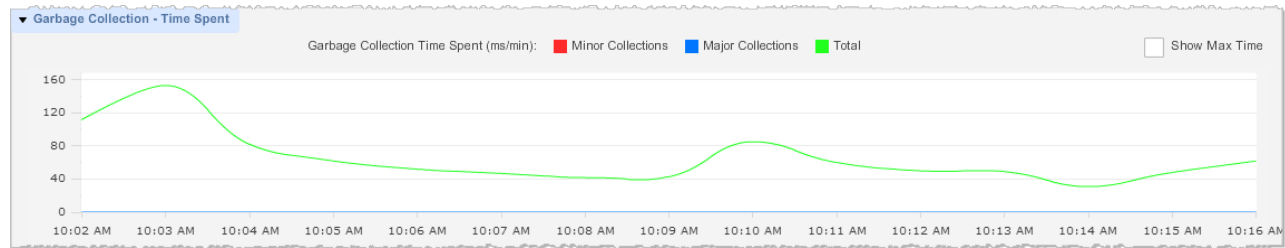


Garbage Collection

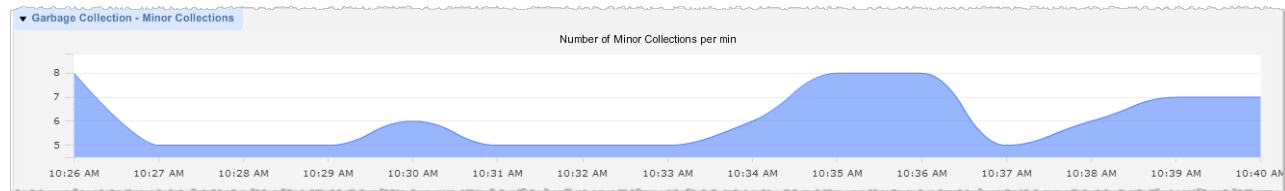
Java garbage collection refers to how the JVM monitors the objects in memory to find any objects which are no longer being referenced by the running application. Unused objects are deleted from memory to make room for new objects. For details see the Java documentation for [Tuning Garbage Collection](#).

Garbage collection is a well-known mechanism provided by Java Virtual Machine to reclaim heap space from objects that are eligible for garbage collection. The process of scanning and deleting objects can cause pauses in the application. Because this can be an issue for applications with large amounts of data, multiple threads, and high transaction rates, AppDynamics captures performance data about the duration of the pauses for garbage collection.

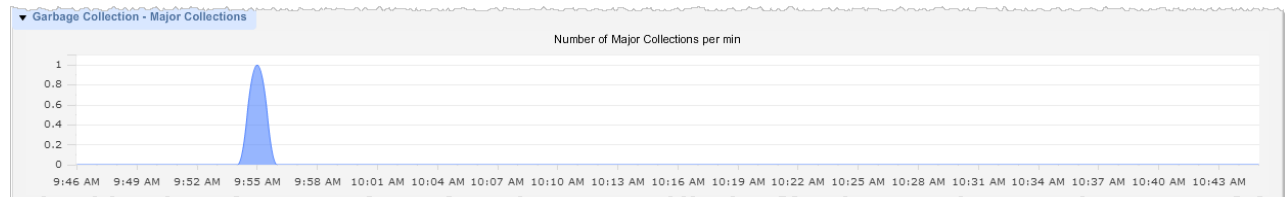
Below the Heap panel, the Garbage Collection - Time Spent panel shows how much time, in milliseconds, it takes to complete both minor and major collections.



The Garbage Collection - Minor Collections panel shows the number of minor collections per minute. The effectiveness of minor collections indicates better performance for your application.

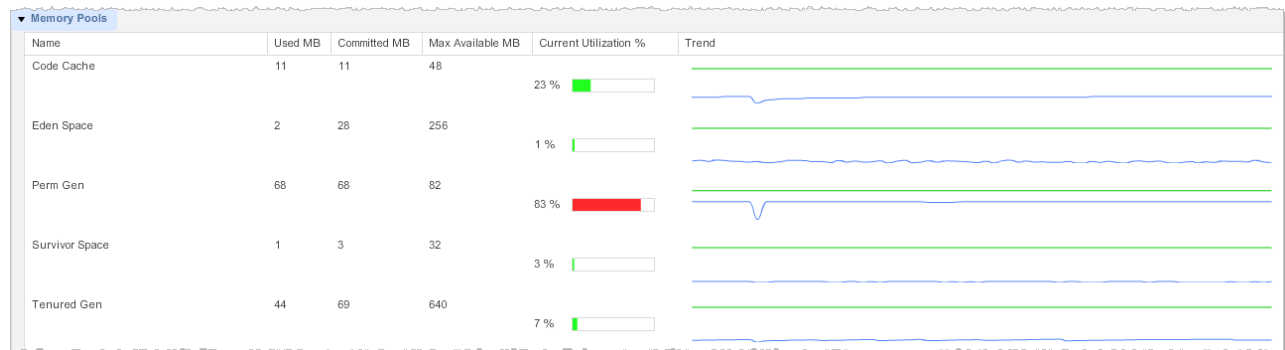


The Garbage Collection - Major Collections panel shows the number of major collections per minute.



Memory Pools

The Memory Pools panel shows usage and trends about the Java memory pools.



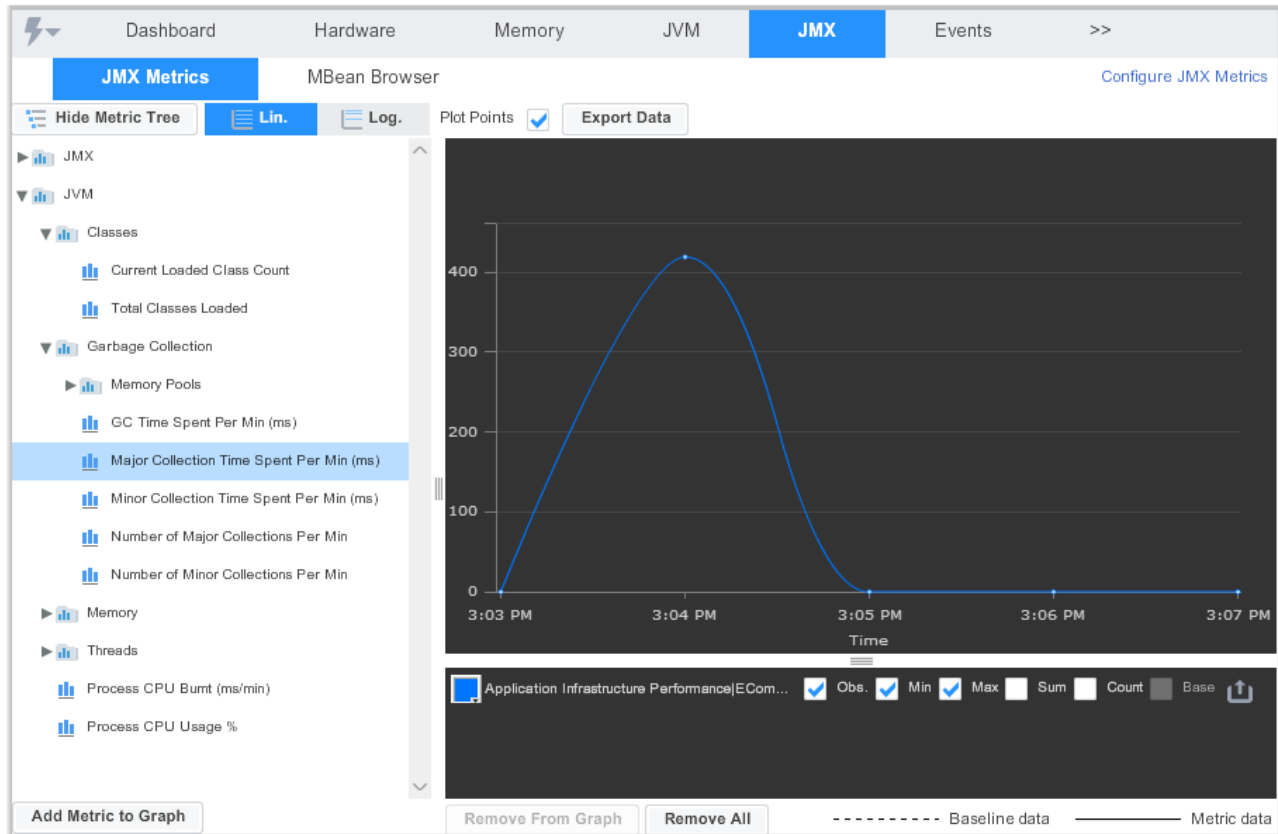
Classes, Garbage Collection, Memory, Threads, and Process CPU Usage Metrics

Information on JVM classes, garbage, threads and process CPU usage is available on the JMX Metrics subtab of the Node Dashboard JMX tab.

To view classes, garbage collection, memory, threads, and process CPU usage metrics

1. In the left navigation pane, click **Servers - > App Servers -> <tier> -> <node>**. The Node Dashboard opens.
2. In the Node Dashboard, click the JMX tab.

3. In the JMX Metrics subtab metric tree, click an item and drag it to the line graph to plot current metric data.



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](#).

Monitor JVM Configuration Changes

The JVM tab of the Node Dashboard displays the JVM version, startup options, system options and environment properties for the node.

The screenshot shows the AppDynamics JVM monitoring interface for a specific Node (Node_8000). The interface is divided into several sections:

- Properties:** Displays basic information about the JVM agent, including Version (Server Agent v3.8.0.1 GA #2014-04-07_23-46-01), JVM Version (Java HotSpot(TM) 64-Bit Server VM 1.6.0_45 Sun Microsystems Inc.), Process ID (18958), and IP Addresses (10.150.5.132).
- JVM Startup Options:** Shows the command-line options used to start the JVM, such as `-Dappdynamics.agent.uniqueHostId=cart-machine`, `-Dcatalina.base=/home/appd/cart-tmp/TIER1TOMCAT`, and `-Xms128m`.
- JVM System Options:** A table listing system properties and their values.

Name	Value
appdynamics.agent.uniqueHostId	cart-machine
appdynamics.viewer.port	8990
catalina.base	/home/appd/cart-tmp/TIER1TOMCAT
catalina.home	/home/appd/cart-tmp/TIER1TOMCAT
does.not	matter2
file.encoding	UTF-8
sun.jvm	1.6
- Environment Properties:** A table listing environment properties and their values.

Name	Value
CVS_RSH	ssh
G_BROKEN_FILENAMES	1

Changes to the application configuration generate events that can be viewed in the Events list.

For more information, see [Monitor Application Change Events](#).

Detect Memory Leaks

By monitoring JVM heap utilization and memory pool usage you can identify potential memory leaks. Consistently increasing heap valleys may indicate either an improper heap configuration or a memory leak. You might identify potential memory leaks by analyzing the usage pattern of either the survivor space or the old generation. To troubleshoot memory leaks see [Troubleshoot Java Memory Leaks](#).

Automatic Leak Detection

AppDynamics supports automatic leak detection for some JVMs as listed in [JVM Support](#). By default this functionality is not enabled, because using this mode results in higher overhead on the JVM. AppDynamics recommends that you enable leak detection mode only when you suspect a memory leak problem and that you turn it off once the leak is identified and remedied.

Memory leaks occur when an unused object's references are never freed. These are the most common occurrences in collections classes, such as HashMap. This is caused when an application code puts objects in collections but does not remove them even when they are not being actively used. In production environments with high workloads, a frequently accessed collection with a memory leak can cause the application to crash.

AppDynamics automatically tracks every Java collection (HashMap, ArrayList, and so on) that has been alive in the heap for more than 30 minutes. The collection size is tracked and a linear regression model identifies if the collection is potentially leaking. You can then identify the root

cause of the leak by tracking frequent accesses of the collection over a period of time.

View this data on the Node Dashboard on the Automatic Leak Detection subtab of the Memory tab. See [Node Dashboard](#) for a complete description of this dashboard and its tabs.

To enable automatic leak detection

To enable automatic leak detection follow the instructions at [Troubleshoot Java Memory Leaks](#).

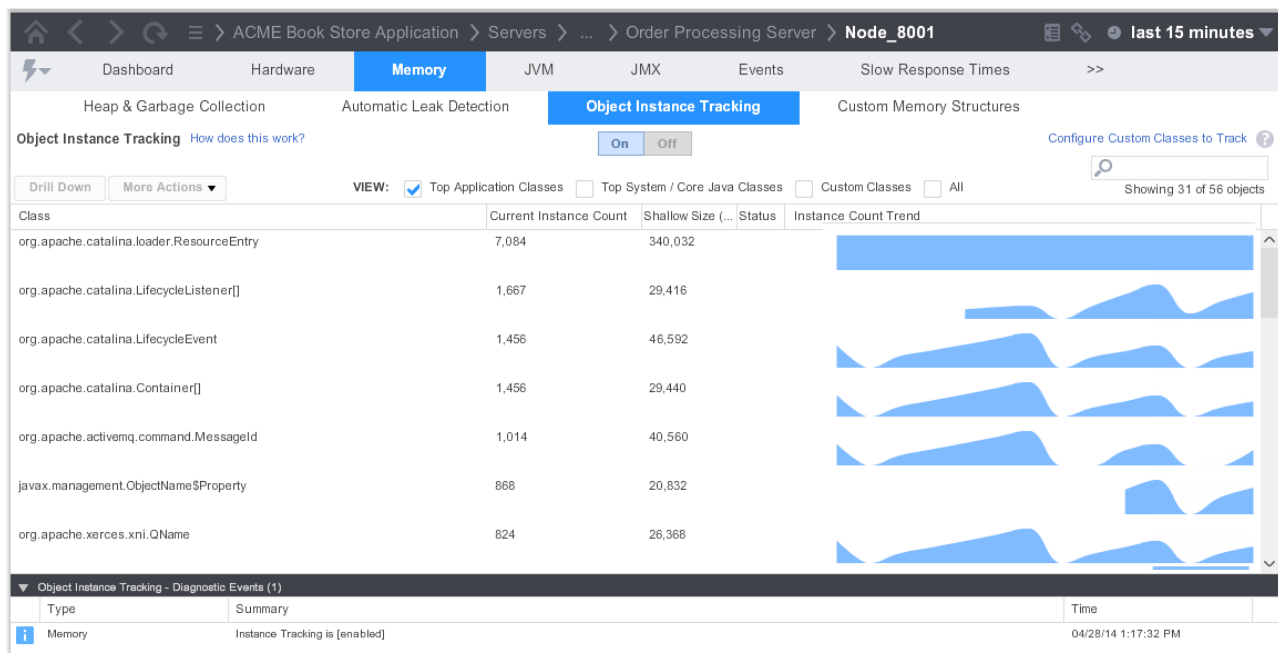
Detect Memory Thrash

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 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. To troubleshoot memory thrash, see [Troubleshoot Java Memory Thrash](#).

Object Instance Tracking

The Object Instance Tracking subtab helps you isolate the root cause of possible memory thrash. By default, AppDynamics tracks the object instances for the top 20 core Java classes and the top 20 application classes. For the list of the supported JVMs see the [Compatibility Matrix for Memory Monitoring](#).

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.



To monitor Java object instances

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 [Configure Object Instance Tracking \(Java\)](#).

Monitor Long-lived Collections

AppDynamics automatically tracks long lived Java collections (HashMap, ArrayList, and so on) with Automatic Leak Detection. You can also configure tracking of 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. Custom memory structures are used as caching solutions. 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 objects that would be tracked using automatic leak detection. It is therefore important to manage and track these 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 internal cache structures

Ensure your custom memory structures are supported on your JVM, see [JVM Support](#).

To view or configure custom memory structures

1. In the Node Dashboard, click the Memory tab.
3. On the Memory tab, click the Custom Memory Structures subtab.

For details see [Configure and Use Custom Memory Structures for Java](#).

Learn More

- [Configure Policies](#)
- [Supported Environments and Versions](#)
- [Infrastructure Metrics](#)
- [Monitor Events](#)

JVM Crash Guard

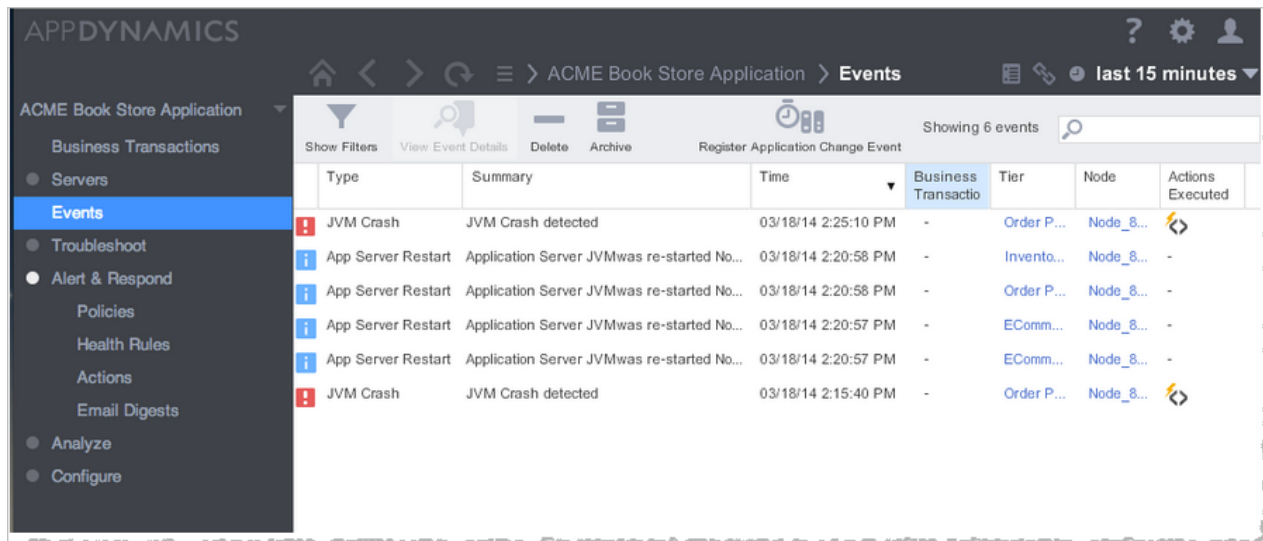
- [To start monitoring for JVM Crashes](#)

When a JVM crash occurs, you need to be notified as soon as possible. Learning of a JVM crash is very critical because it may be a sign of a severe runtime problem in an application.

Furthermore, you may want to take remediation steps once you are aware that a crash event has occurred. JVM Crash is a new event type, implemented as part of JVM Crash Guard, that you can activate to provide you with the critical information you need to expeditiously handle JVM crashes.

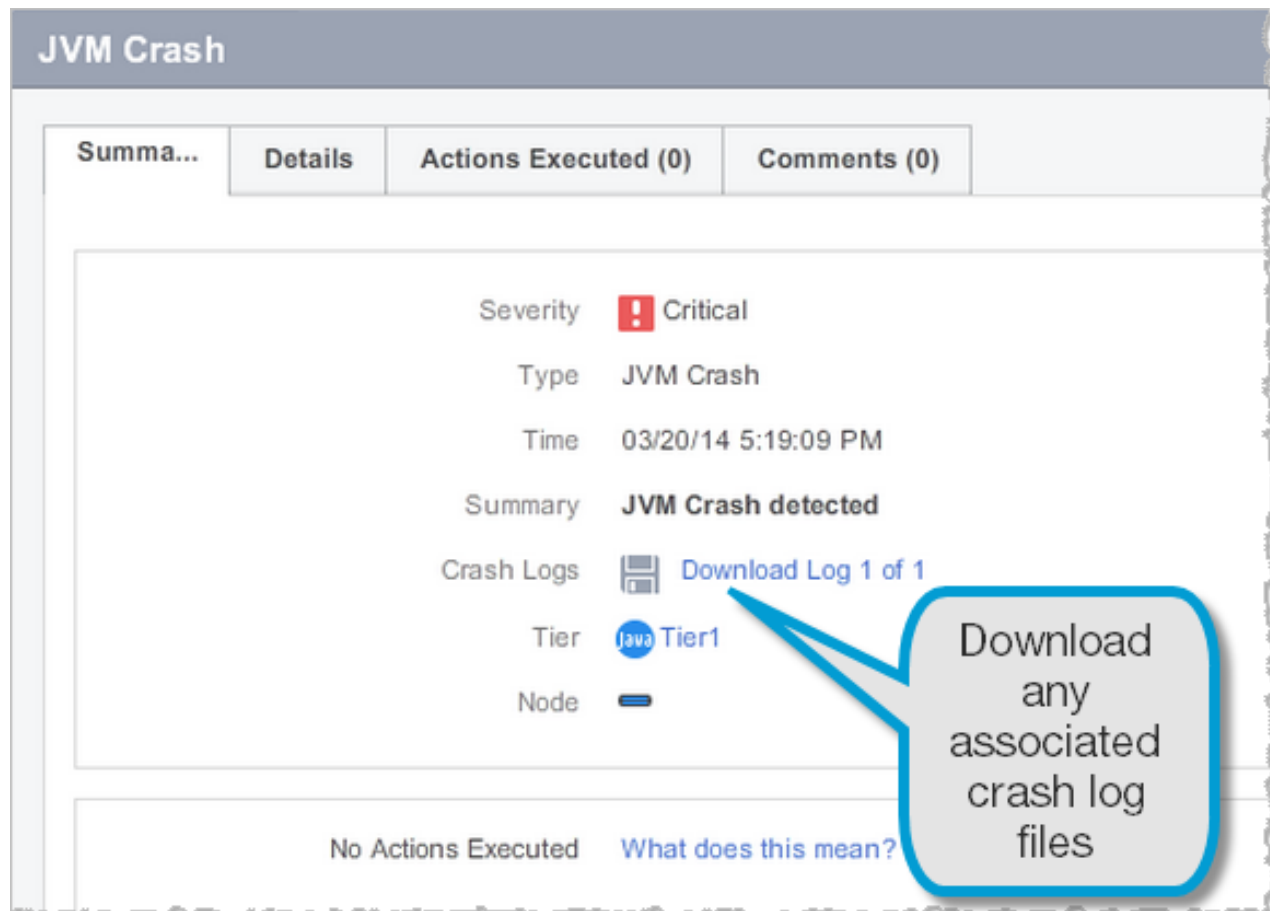
The following image shows the Events window where notification of two JVM Crash events

detected is displayed.



Double-clicking the JVM Crash event on the Events window displays more information to assist you in troubleshooting the underlying reason for the JVM crash.

On the Summary page you can download any logs associated with the JVM Crash event.



The JVM Crash window also displays information about actions executed as a result of the crash. These are actions that you specify when creating the policy that is triggered by a JVM crash event.

The screenshot shows the 'JVM Crash' window with the 'Details' tab selected. The window displays the following information:

Summary	Details	Actions Executed (0)	Comments (0)
<p>Copy to Clipboard</p> <p>Command line</p> <pre> /Library/Java/JavaVirtualMachines/jdk1.7.0_45.jdk/Contents/Home/jre/bin/java -Ddoes.not=matter1 -Ddoes.not=matter2 -javaagent:/Users/bwinslow/git/cart-mp/TIER1TOMCAT/appagent/javaagent.jar -Dappdynamics.agent.uniqueHostId=cart-machine -Dappdynamics.bciengine.write2disk=/Users/bwinslow/git/cart-bci -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/Users/bwinslow/git/cart-mp/TIER1TOMCAT/logs -D-XX:+UseG1GC -XX:PermSize=256m -XX:ErrorFile=/Users/bwinslow/crashlogs/java_error%p.log -Xmx512m -Dminimum_age_for_evaluation_in_mins=0 -DlistenPort=8000 -DshutdownPort=8180 -Djava.util.logging.manager=org.apache.juli.ClassLoaderLogManager -Djava.util.logging.config.file=/Users/bwinslow/git/cart-mp/TIER1TOMCAT/conf/logging.properties -Dcom.appdynamics.bounded.collections.default.policy=WarnIfBoundedExceeded -Dcatalina.base=/Users/bwinslow/git/cart-mp/TIER1TOMCAT -Dappdynamics.viewer.port=8990 -Dcom.sun.management.jmxremote.port=9004 -Dcom.sun.management.jmxremote.authenticate=false -Dappdynamics.enable.missed.class.scan=false -Dappdynamics.enable.missed.class.scan.freq=120 -Dappdynamics.avoid.boot.class.loader.lookup=true -Dappdynamics.background.transaction.reporter.queue.type=circular -Dappdynamics.async.all.metrics=sometimes -Dappdynamics.background.transaction.reporter.initial.circular.queue.size=1000 -Dappdynamics.hotspot.enabled=false -Dappdynamics.bciengine.should.implement.new.interfaces=true -Dappdynamics.socket.collection.bci.enable=false -Dappdynamics.agent.runtime.dir=/Users/bwinslow/AgentRuntime -Dappdynamics.object.monitor.hang.detect=600 -Dappdynamics.scheduler.time.adjuster.required=sync -classpath /Library/Java/JavaVirtualMachines/jdk1.7.0_45.jdk/Contents/Home/lib/tools.jar:/Users/bwinslow/git/cart-mp/TIER1TOMCAT/bin/bootstrap.jar org.apache.catalina.startup.Bootstrap </pre> <p>Crash Date/Time Thu Mar 27 15:55:43 PDT 2014</p> <p>Crash Reason SIGSEGV (0xb) at pc=0x0000000010e5046fd, pid=36832, tid=67075</p> <p>File_0 1395960944535/java_error36832.log</p> <p>Host name osx11bwins.local</p> <p>IP Address 192.168.1.9</p> <p>PID 36832</p> <p>appName ACME Book Store Application</p> <p>nodeName Node_8000</p> <p>tierName ECommerce Server</p>			

Close

The JVM Crash event captures the following information: timestamp, crash reason, host name, IP address, process ID, application name, node name, and tier name and displays them on the details page.

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, Segmentation Fault, etc... To facilitate the discovery and display of the reason for the JVM crash, JVM Crash Guard provides full support for:

- Hotspot JVM error log analysis
- IBM JVM System Dump log analysis
- Jrockit JVM error log analysis

To start monitoring for JVM Crashes

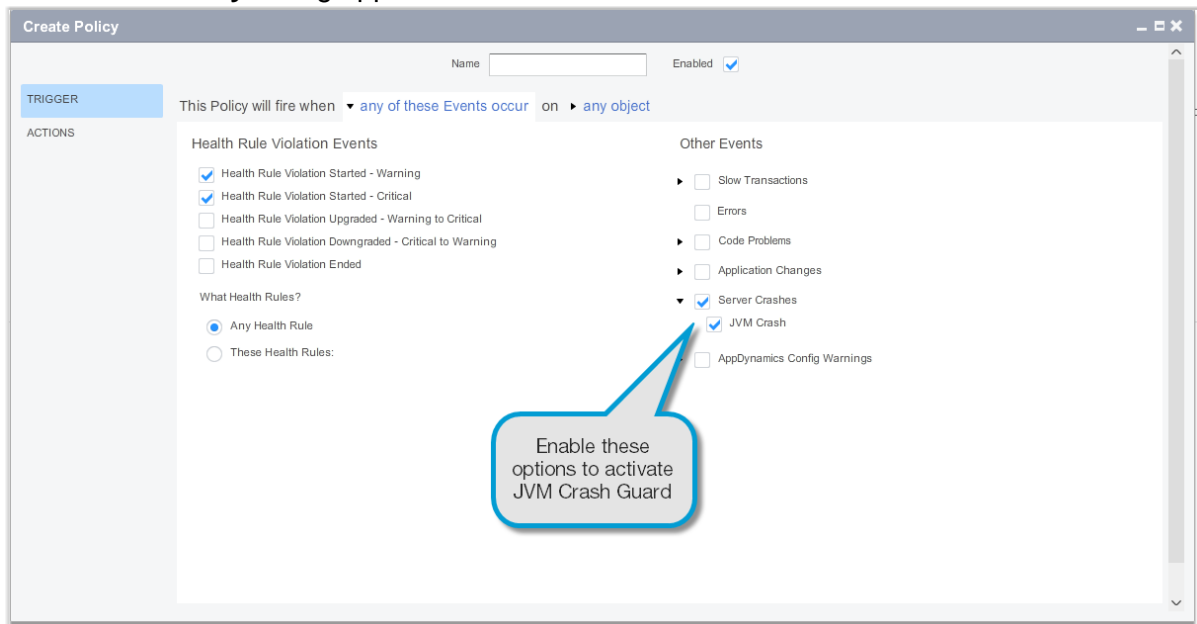


Prerequisite

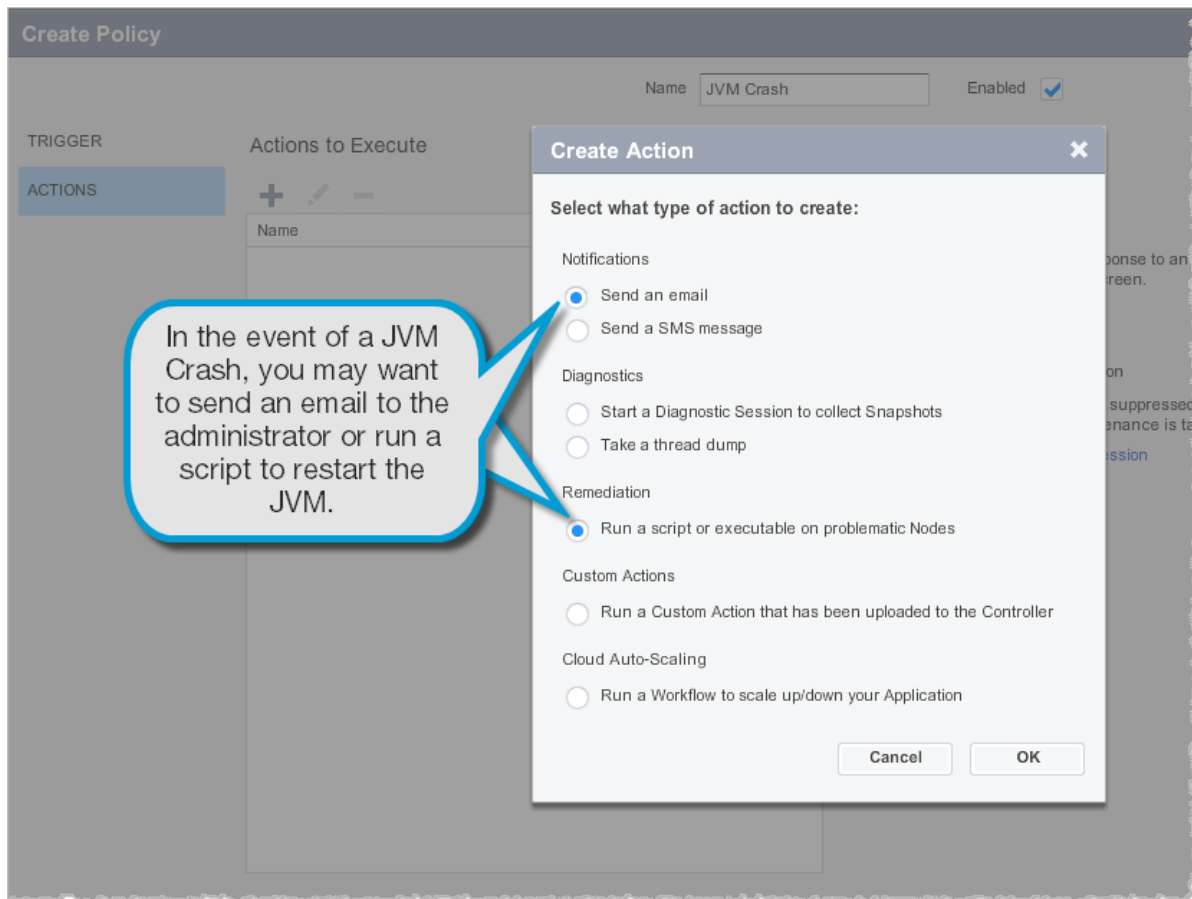
JVM Crash Guard is a policy trigger that works with the Standalone Machine Agent to fire an AppDynamics policy when a JVM Crash event occurs. You must therefore have a Standalone Machine Agent installed on the system which you want to monitor for JVM crashes. On Windows, the Standalone Machine Agent must run in Administrator mode.

1. From the left-hand navigation menu, click **Alert & Respond ->Policies** and then click **Create a Policy**.
OR

Navigate to the **Policies** tab and then click **Create a Policy**.
The **Create Policy** dialog appears.

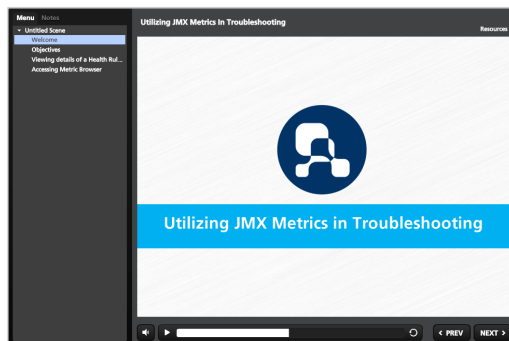


2. In the **Other Events** section, expand the **Server Crashes** option and click **JVM Crash**.
The JVM Crash event then becomes a trigger to fire a policy.
3. Proceed as usual to create the Policy. For more information on creating Policies, see [Policies](#).



Monitor Java App Servers

- Infrastructure Monitoring in a Java Environment
- App Server Key Performance Indicators
- Alerting for App Server Health
- Learn More



Utilizing JMX Metrics in Troubleshooting

Infrastructure Monitoring in a Java Environment

A Java application environment has multiple functional subsystems. These are usually instrumented using JMX (Java Management Extensions) or IBM Performance Monitoring Infrastructure (PMI). AppDynamics automatically discovers JMX and PMI attributes.

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.

For more information about JMX, see the [JMX overview and tutorial](#). To learn about PMI see [Writing PMI Applications Using the JMX Interface](#).

App Server Key Performance Indicators

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 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 see the JMX metrics discovered in a node, see the JMX tab on the Node Dashboard.

To learn how to customize additional MBean attributes for long-term monitoring, see [Configure JMX Metrics from MBeans](#).

Alerting for App Server Health

AppDynamics discovers metrics for most Java platforms and applications. Some environments however are not instrumented by default, yet they have MBeans. For those situations you can enable monitoring using the MBean Browser. For details see [Monitor JMX MBeans](#).

In addition to the preconfigured metrics, you may be interested in additional JVM or Java container metrics. You can add custom metrics using JMX MBean attributes in the Metric Browser. To

customize which MBean attributes are monitored, see [Configure JMX Metrics from MBeans](#).

Once you add a custom metric you can create a custom health rule for it and receive alerts if conditions indicate problems. For details see [Alert and Respond](#).

AppDynamics also provides the Application Server Agent API (Agent API) for access to metrics that are not supported by default or by MBeans. You can use the Agent API to:

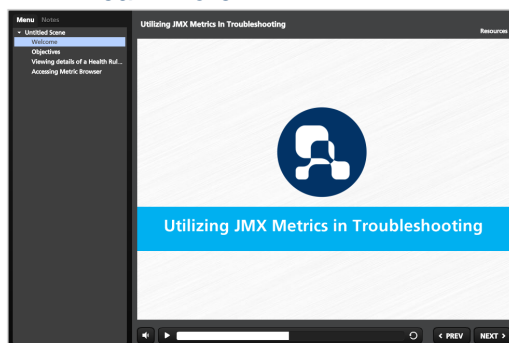
- Inject custom events and report on them
- Create and report on new metrics
- Correlate distributed transactions when using protocols that AppDynamics does not support

Learn More

- [Configure JMX Metrics from MBeans](#)
- [Monitor JMX MBeans](#)
- [Configure Health Rules](#)
- [Supported Environments and Versions](#)

Monitor JMX MBeans

- [JMX and MBeans Monitoring Application Infrastructure](#)
 - [Prerequisites for JMX Monitoring](#)
 - [Preconfigured JMX Metrics](#)
 - [To view the configuration of the preconfigured JMX metrics](#)
- [Using AppDynamics for JMX Monitoring](#)
 - [To view JMX metrics in the Metrics Browser](#)
- [Trending MBeans Using Live Graphs](#)
 - [To monitor the real-time trend of an MBean](#)
- [Working with MBean Values](#)
 - [To View and Edit the MBean Attribute Values](#)
 - [To invoke MBean Operations](#)
 - [To view Complex MBean Attributes](#)
- [Configuring New JMX Metrics](#)
- [Reusing JMX Metric Configurations](#)
- [Understanding JMX Metrics](#)
- [Learn More](#)



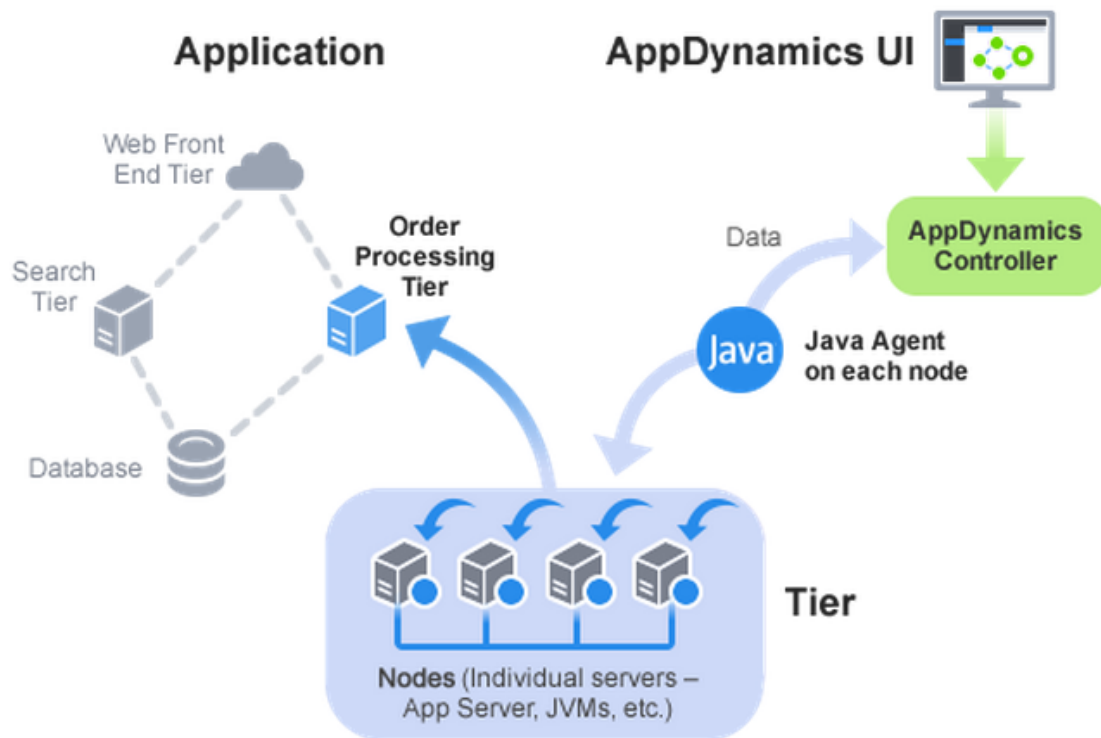
Utilizing JMX Metrics in Troubleshooting

This topic discusses how to provide visibility into the JMX metrics for your JVM and application server.

JMX and MBeans Monitoring Application Infrastructure

As discussed at [Monitor JVMs](#) and [Monitor Java App Servers](#), AppDynamics uses JMX (Java Management Extensions) to monitor Java applications.

JMX and MBean Monitoring uses the Java Agent, which works like this:



JMX uses objects called MBeans (Managed Beans) to expose data and resources from your application. You can use one or more MBean attributes to create persistent JMX metrics in AppDynamics. In addition, you can import and export JMX metric configurations from one version or instance of AppDynamics to another.

Prerequisites for JMX Monitoring

AppDynamics can capture MBean data, when these conditions are met:

- The monitored system must be running on Java 1.5 or later.
- Each monitored Java process must enable JMX. See [the JMX documentation](#).

Additional MBean data may be available when a monitored business application exposes Managed Beans (MBeans) using standard JMX. See [the MBean documentation](#).

Preconfigured JMX Metrics

AppDynamics provides preconfigured JMX metrics for several common app server environments:

- Apache ActiveMQ
- Cassandra
- Coherence
- GlassFish
- HornetQ
- JBoss
- Apache Solr
- Apache Tomcat
- Oracle WebLogic Server
- WebSphere PMI

For application server environments that are not instrumented by default, you can configure new JMX metrics configurations. You can also add new JMX metric rules. See [Configure JMX Metrics from MBeans](#). You can also add new metric rules to the existing set of configurations. For example, Glassfish JDBC connection pools can be manually configured using MBean attributes and custom JMX metrics.

To view the configuration of the preconfigured JMX metrics

1. In the left navigation pane, click **Configure -> Instrumentation** and select the **JMX** tab.
2. The list of **JMX Metric Configurations** appears.
Click a metric configuration to view the preconfigured JMX metrics for that app server.
3. For example, selecting Cassandra shows the preconfigured JMX Metric Rules for Apache Cassandra.
Double-click a metric rule to see configuration details such as the MBeans matching criteria and the MBean attributes being used to define the metric.

The screenshot displays the AppDynamics web interface for configuring JMX metrics. The left navigation pane shows the 'Configure' section with 'Instrumentation' selected. The main area shows a list of 'JMX Metric Configurations' for 'WebSpherePMI'. A callout points to the list with the text 'Add, delete, copy, export and import'. Another callout points to a specific configuration 'WebSpherePMI_ThreadPools' with the text 'Click to edit, add, conditions, and define metrics'. The right-hand panel shows the configuration details for 'WebSpherePMI_ThreadPools', including 'MBean Matching Criteria' and 'Attributes'. A callout points to the 'MBean Matching Criteria' section with the text 'Edit, add conditions, define metrics'. Another callout points to the 'Attributes' section with the text 'Click to see these options'. The 'Attributes' section shows 'Define Metrics from MBean Attribute(s)' with 'PoolSize' selected.

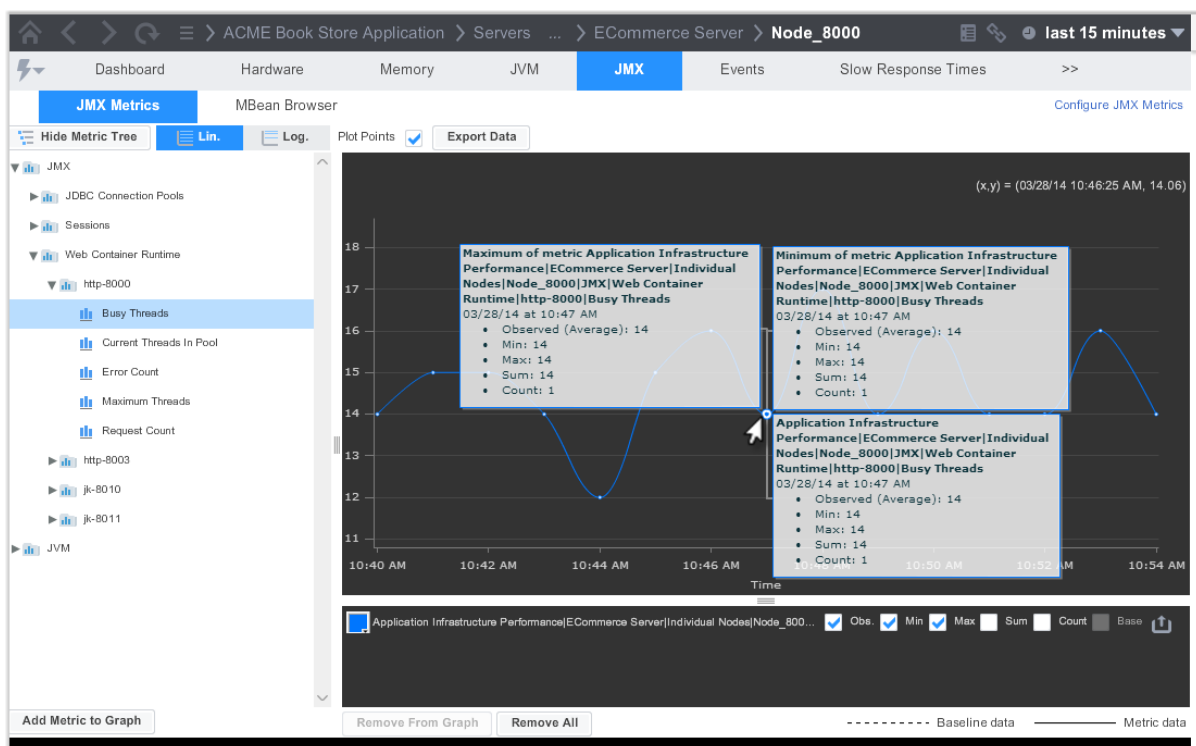
You can view, delete, and edit the existing JMX metric rules.

Using AppDynamics for JMX Monitoring

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

1. In the left navigation pane, click **Servers -> App Servers -> <Tier> -> <Node>**. The Node Dashboard opens.
2. Click the **JMX** tab. The JMX Metrics browser opens and displays the MBeans in a Metric Tree.
3. To monitor a particular metric, double-click or drag and drop the metric onto the graph panel.
4. Browse the default JMX metrics.



5. You can perform all the operations that are provided by the Metric Browser such as:
 - Drill-down
 - Analyze the transaction snapshot for a selected time duration
 - Set the selected time range as a global time range

Trending MBeans Using Live Graphs

You can monitor the trend of a particular MBean attribute over time using the **Live Graph**.

To monitor the real-time trend of an MBean

1. In the left navigation pane, click **Servers -> App Servers -> <Tier> -> <Node>**. The Node Dashboard opens.
2. Click the **JMX** tab.

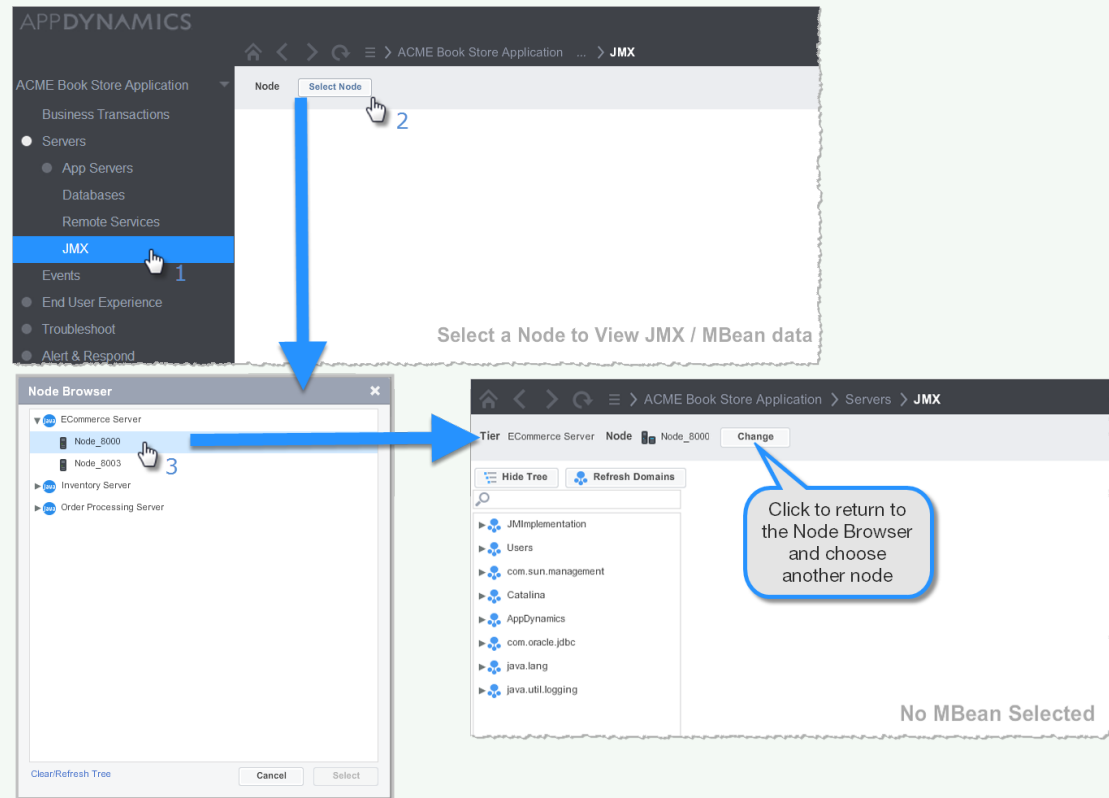
- Click the **MBean Browser** sub-tab.



Alternate Path to JMX Metrics

Alternatively, in the left navigation pane, click **Servers -> App Servers -> JMX**. The JMX window appears

If you haven't already selected a node, you are prompted to select a node and then a tier and then the JMX window appears



- Select the domain for which you want to monitor MBeans. For a description of domains see [Monitor JVMs](#).
- In the domain tree, expand the domains to find and then select the MBean that is of interest to you.
- Expand the **Attributes** section and then choose an attribute of the MBean.
- Click **Start Live Graph for Attribute** and then click **Start Live Graph**. You can see the runtime values.
- Select an attribute and click **Live Graph for Attribute** to see a larger view of a particular graph.

The screenshot shows the AppDynamics JMX interface. The top navigation bar includes Dashboard, Hardware, Memory, JVM, JMX, Events, and Slow Response Times. The left sidebar shows the ACME Book Store Application hierarchy, with Node_8000 selected. The main panel displays the MBean Browser for the MBean Object Name: Catalina:type=Connector,port=8000. A table lists 28 attributes, with 'acceptCount' selected. A callout box explains the 'Start Live Graph' button: 'Start graphing the current values of the selected MBean attribute. The graph, shown at the bottom of this screen, will continually update while you are on this screen.' Below the table, a live graph for 'acceptCount' is shown, with a callout box stating 'Live Graph for Attribute: acceptCount'.

Name	Type	Value	Editable	Live Graph
acceptCount	int	100	Yes	
allowTrace	boolean	false	Yes	
bufferSize	int	2048	Yes	
compression	java.lang.String	off	Yes	
connectionLinger	int	-1	Yes	
connectionTimeout	int	20000	Yes	
connectionTimeout	int	300000	Yes	

Working with MBean Values

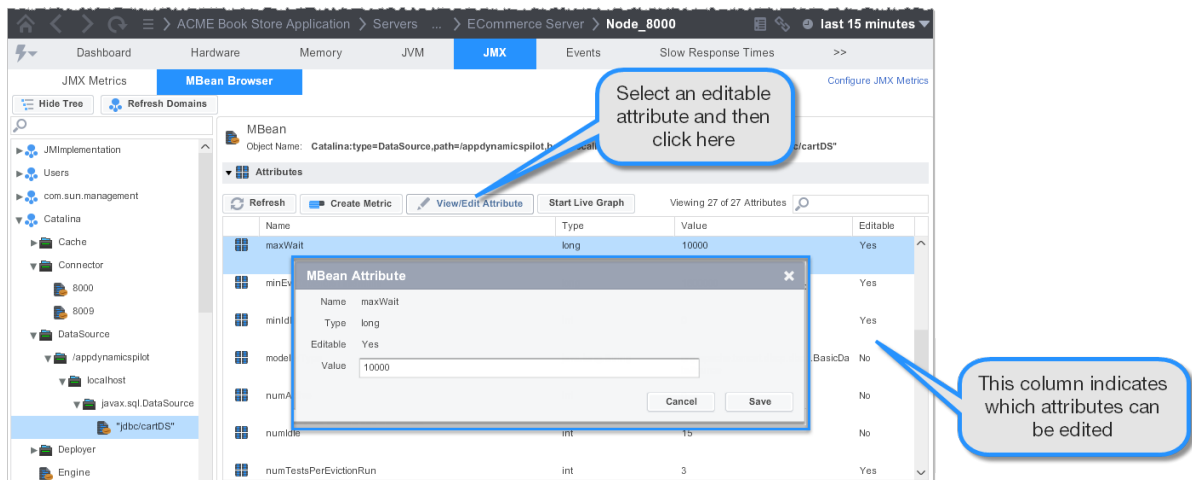
When troubleshooting or monitoring a Java-based system, you may want to change the values of composite mBeans and execute mBean methods. Using the JMX window, you can accomplish these tasks.

i Prerequisite for Setting MBean Attributes and Invoking Operations

- To change the value of an MBean attribute or invoke operation, your user account must have "Set JMX MBean Attributes and Invoke Operations" permissions for the application. For information about configuring user permissions for applications, see [To Configure the Default Application Permissions](#).

To View and Edit the MBean Attribute Values

- From the **JMX** window, select **MBean Browser**.
- In the Domain tree, search and find the MBean that interests you.
- Select an editable attribute, one that has **Yes** in the **Editable** column, and then click **View/Edit Attribute**.
- In the **MBean Attribute** window that displays, you see the current value of the MBean Attribute.

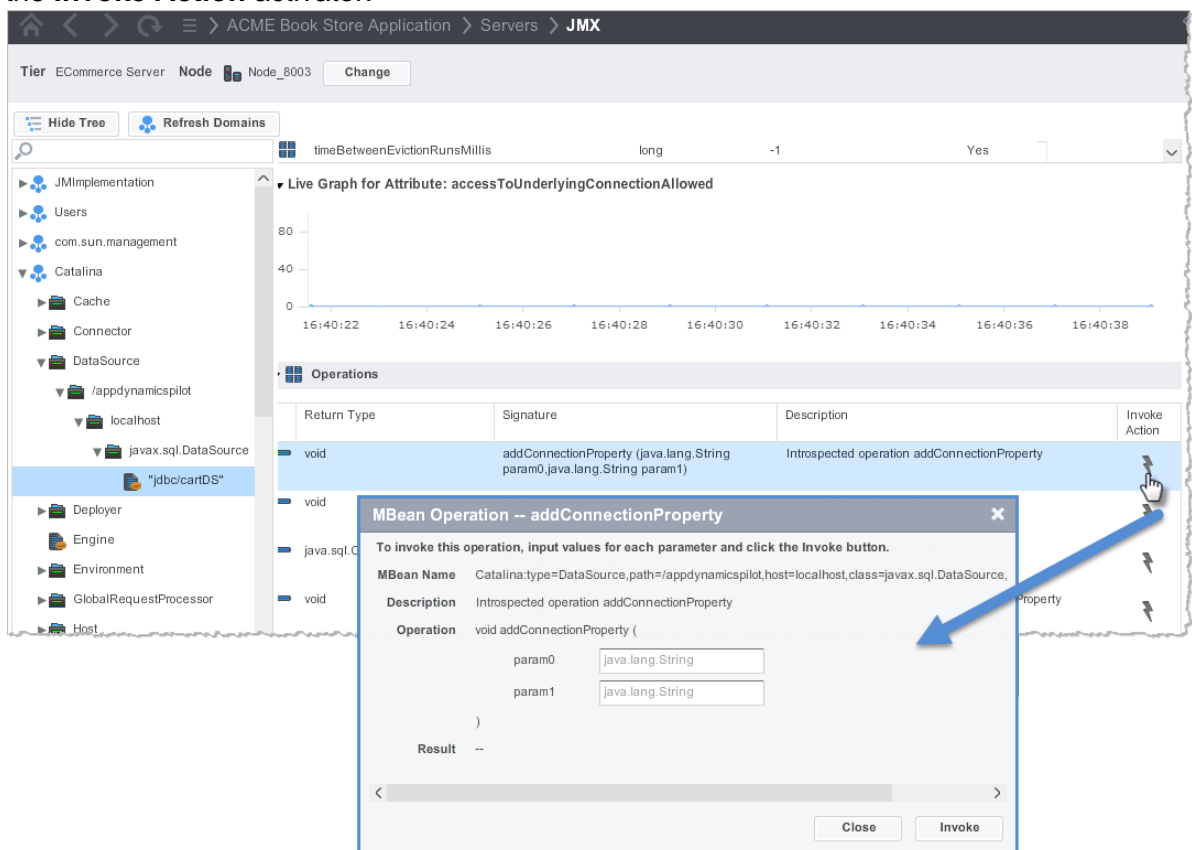


5. You can change the value of an editable MBean Attribute by entering a new value in the **Value** field.

To invoke MBean Operations

Using the JMX viewer, you can invoke an mBean operation, specify standard java language strings for the parameters, and view the return values from the mBean invocation.

1. From the **JMX** window, select **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**. Click **OK** to invoke the operation.

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.

MBean Operation -- isServiced [X]

To invoke this operation, input values for each parameter and click the Invoke button.

MBean Name Catalina:type=Deployer,host=localhost

Description Add a web application name to the serviced list

Operation boolean isServiced (

name

)

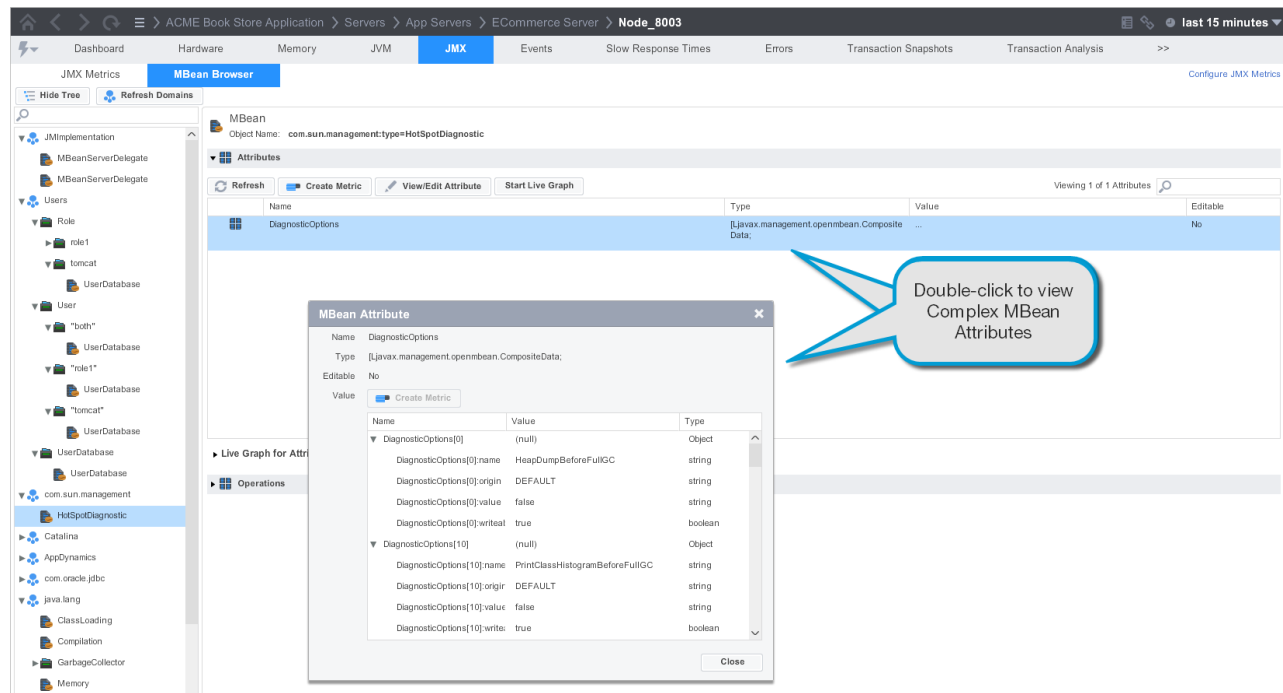
Result Operation Successfully Invoked:

(copy)

Close Invoke

To view Complex MBean Attributes

When the MBean is a complex type, you can view its details by double-clicking it as shown below.



Configuring New JMX Metrics

Required User 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, see [To Configure the Default Application Permissions](#).

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.

You can create a JMX metric from any MBean attribute or set of attributes. 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

The JMX Metrics Configuration panel is the central configuration interface for all of the JMX metrics that AppDynamics reports. You can use the MBean Browser to view MBeans exposed in your environment. From there, you can access the JMX Metrics Configuration panel by selecting an MBean attribute and clicking **Create Metric**.

For details, see [Configure JMX Metrics from MBeans](#).

Reusing JMX Metric Configurations

Once you create a custom JMX metric configuration, you can keep the configuration for upgrade or other purposes. The JMX metric information is stored in an XML file that you can export and then import to another AppDynamics system. For instructions see [Create, Import or Export JMX Metric Configurations](#).

Understanding 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](#), in the [ActiveMQ Features documentation](#)
- Cassandra, see [Cassandra Metrics](#)
- Coherence, see the Coherence MBeans Reference in Appendix A of the [Coherence Management Guide](#)
- GlassFish, see the Oracle Sun Glassfish Administration Guide where you can find a [Metrics Information Reference](#)
- HornetQ, see [Using Management Via JMX](#)
- JBoss, see [An Introduction to JMX](#)
- Apache Solr, see [Quick Demo](#) in the Solr JMX documentation
- Apache Tomcat, see the descriptions of [JMX MBeans for Catalina](#) in the mbeans-descriptor.xml file for each package
- Oracle WebLogic Server, see [Understanding JMX](#)
- WebSphere PMI, see [PMI data organization](#) in the IBM Websphere Application documentation

Learn More

- [Configure JMX Metrics from MBeans](#)
- [Monitor JVMs](#)
- [Create, Import or Export JMX Metric Configurations](#)
- [Configure JMX Without Transaction Monitoring](#)

Trace MultiThreaded Transactions for Java

- [Thread Visibility and Metrics](#)
 - [Thread Metrics](#)
 - [Asynchronous Activity in Dashboards](#)
 - [Application Dashboard](#)
 - [Business Transaction Dashboard](#)
 - [Threads and Thread Tasks in the Metric Browser](#)
- [Threads in Call Graphs](#)
 - [To Drill Down into Downstream Calls on a Thread](#)
- [Thread Metrics in Health Rules](#)
- [Learn More](#)

Multithreaded programming techniques are common in applications that require asynchronous processing. Although each thread has its own call stack, multiple threads can access shared data. This creates two potential problems: visibility and access.

- A visibility problem occurs if thread A reads shared data which is later changed by thread B, and thread A is not aware of the change.
- An access problem occurs if several threads are trying to access and change the same

shared data at the same time.

Visibility and access problems can lead to:

- Liveness failure: Application performance becomes sluggish or stops processing also known as a deadlock.
- Safety failure: Race condition that results in difficult to discover programming errors.

Thread contention can occur when multiple threads attempt to access a synchronized method or block at the same time. If a thread remains in the synchronized method or blocks for a long time, the other threads must wait for access to shared resources. This situation has an adverse effect on application performance. Call graphs for multi-threaded transactions enable you to trace thread creation in a business transaction and provide an aggregated view of the overall processing for transactions that spawn threads for concurrent processing.

AppDynamics monitors asynchronous activities as first class entities with their own metrics to give you the information you need to see and act to correct these performance issues.

Thread Visibility and Metrics

When applications spawn threads to perform concurrent tasks, you can monitor each thread as a separate entity, including exit calls and policies associated with a specific thread. By default, all Runnables, Callables and Threads are instrumented, except those that are explicitly excluded.

AppDynamics provides the flexibility to adjust the default monitoring to match the needs of your specific applications. In some environments, if the default settings lead to too many classes being instrumented, you can create custom rules to exclude unnecessary classes. If you do not want to monitor any threads, you can completely disable asynchronous monitoring. This requires an agent restart. See [Configure Multi-Threaded Transactions for Java](#).

AppDynamics provides thread visibility in dashboards, the metric browser and snapshots.

Thread Metrics

For each asynchronous thread spawned in the course of executing a business transaction, AppDynamics collects and reports metrics such as the following:

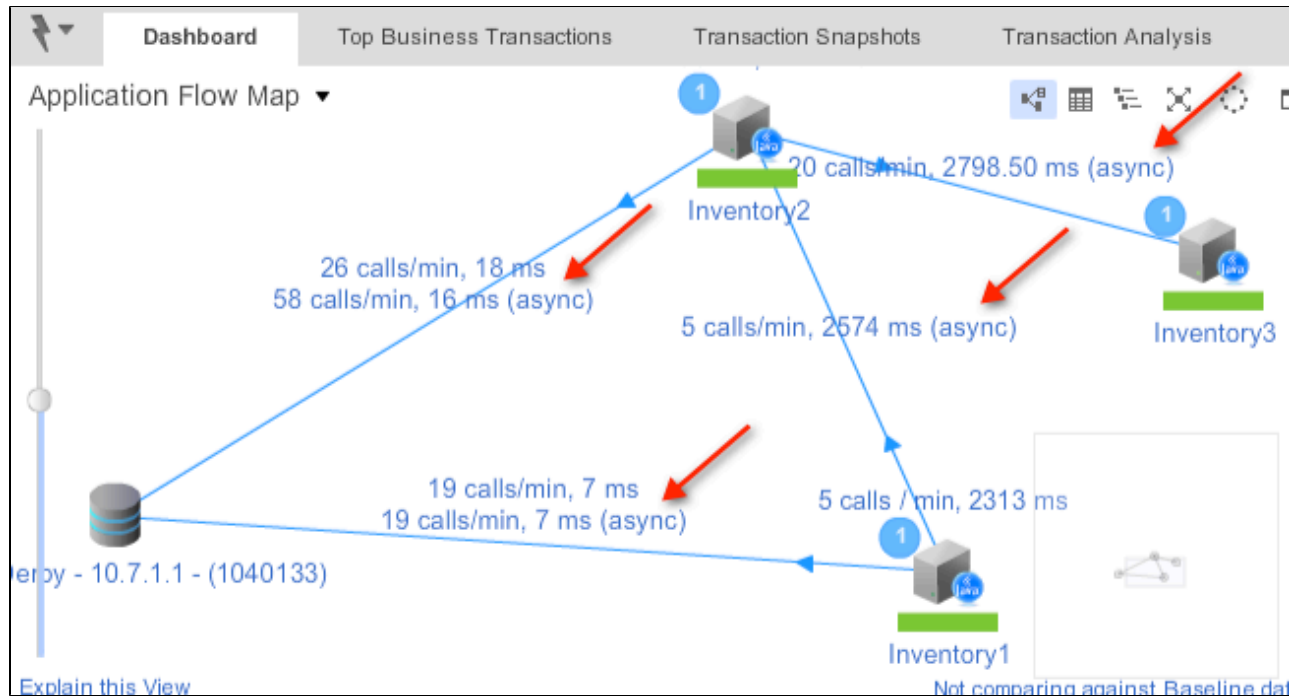
- Average response time
- Calls per minute
- Errors per minute

Asynchronous Activity in Dashboards

AppDynamics detects asynchronous calls in an application and labels them as "async" in the dashboards that display the asynchronous activity.

Application Dashboard

In the following Application Flow Map, you can see the calls per minute and average response time displayed on each flow line where asynchronous activities are detected. These metrics aggregate the metrics for asynchronous activities across all business transactions.



For improved visibility, you can set the flow map lines to display as dotted lines. See [To enable dotted flow line](#).

Business Transaction Dashboard

Asynchronous activity can be viewed in a hierarchical format with the originating activities encapsulating their respective spawned asynchronous activities. The tree view of a multi-threaded business transaction flow shows the hierarchical view as well as the errors and time spent in asynchronous calls for separate business transactions. To expand the tree to see all the calls, right-click a tier name and select **Expand All**.

Dashboard

Events

Slow Response Times

Errors

Transaction Snapshots

Transaction Analysis

Transaction Flow - Tree View

Inventory2

2525.0 ms

100.0 %

72

5

0

0

JDBC call to Apache Der

14.0 ms

2.2 %

290

19

0

0

InventorySupplierLookup

1593 ms

async

72

5

0

0

HTTP call to Inventor

1593 ms

async

72

5

0

0

InventoryJobController

108 ms

async

72

5

0

0

QueryRunner

84 ms

async

72

5

0

0

LookupServlet\$1

506 ms

async

72

5

0

0

InventoryRequestBaseR

2006 ms

async

72

5

0

0

InventoryAggregation

2008 ms

async

72

5

0

0

The following metrics are visualized in the Transaction Flow Tree View:

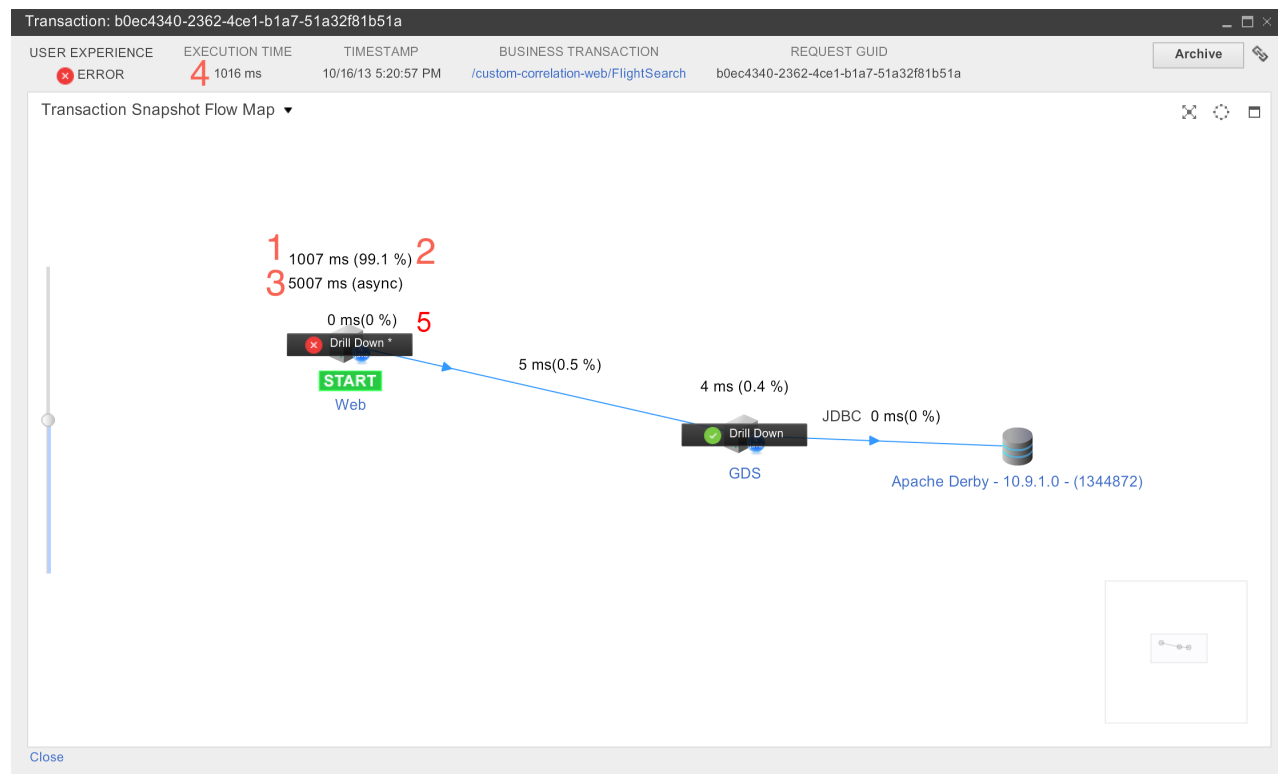
Metric Name	Explanation
-------------	-------------

Time Spent (ms)	Average time spent by the specific activity and any spawned asynchronous activities. Percentage metrics are used to represent the fraction of time spent in a specific activity. Asynchronous activities do not have a percentage breakdown because each asynchronous activity is linked to the originating business transaction, but represents a separate logical entity throughout the execution of the business transaction.
Calls	Number of calls made by a particular activity such as an asynchronous activity.
Calls/min	Number of calls made per minute for a particular activity such as an asynchronous thread.
Errors	Number of calls for a particular activity which resulted in errors.
Errors/min	Number of calls made per minute for a particular activity which resulted in errors.

Trends for baselines are visualized using the data for the originating business transaction. Metrics for the asynchronous activities are not used in calculation of these trends.

The Transaction Scorecard reflects only the data for the originating business transaction. The scorecard metrics are not inclusive of the metrics for any asynchronous threads being spawned by the originating business transaction.

The Transaction Snapshot Flow Map for a transaction with asynchronous activity displays both synchronous time and time spent in asynchronous activity. The following are example screen shots of transactions with asynchronous activities:



On the transaction flow map, AppDynamics displays the following metrics:

Metric Name	Explanation
1. Tier Response Time (ms)	Time spent processing at a particular tier for this business transaction. Only present for originating tier snapshots. (first in chain)
2. Percentage of Time Spent (%)	Percentage metric represents the fraction of time spent processing at a particular tier or in communication with other tiers/backends from the entire execution lifespan of a business transaction. Only present for "first in chain" snapshots. This metric does not include the processing time of the asynchronous activities

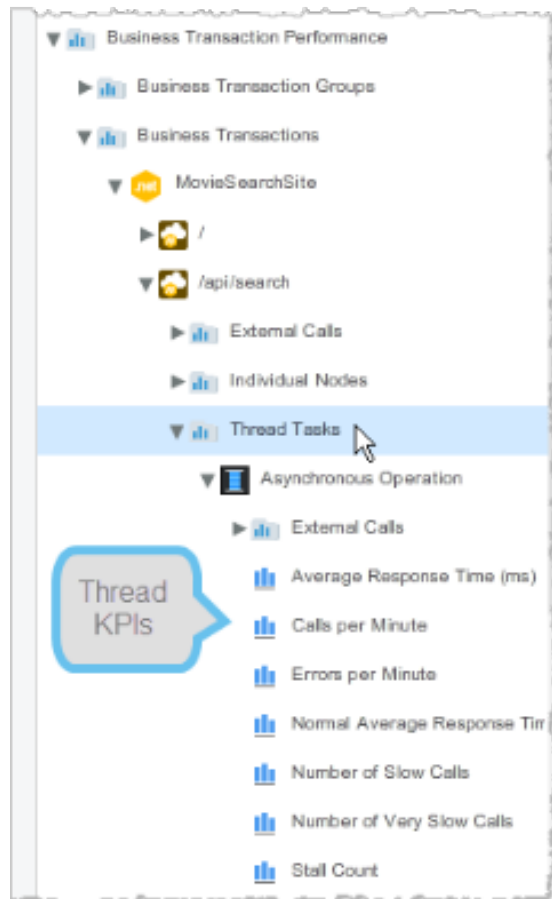
3. Asynchronous Activity Processing Time (ms)	<p>Processing time of all asynchronous activities at this tier. This metric does not contribute to the overall tier response time because the activity is asynchronous by nature. This metric is calculated by adding the execution times of all asynchronous activities at a tier and the time spent in communication between other tiers and backends as follows:</p> <p><i>Asynchronous Activity Processing Time = Asynchronous-activity-1-processing-time + Asynchronous-activity-2-processing-time + so on.</i></p>
4. Execution Time (ms)	<p>Time spent processing by the business transaction in all affected tiers and communication with other tiers and backends. This metric does not include processing time of the asynchronous activities. However, in the case of Wait-for-Completion, the originating business transaction will take a longer time processing the request due to blocking and waiting for all the activities to complete before proceeding.</p> <p>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:</p> <p><i>Execution Time = Time-spent-processing-in-Tier-1 + Time-spent-processing-in-Tier-2 + Time-spent-communicating-with-Tier-2 + so on.</i></p>
5. Call back to same tier (ms)	<p>This label and corresponding value are not always present. When a tier makes an exit call and the call is received back by the same tier then this is displayed. The metric value corresponds to the time spent in the call from the moment the call went out of the tier until the point the call returned back to the caller. If the label contains "async" then the exit call was made asynchronously.</p>

Threads and Thread Tasks in the Metric Browser

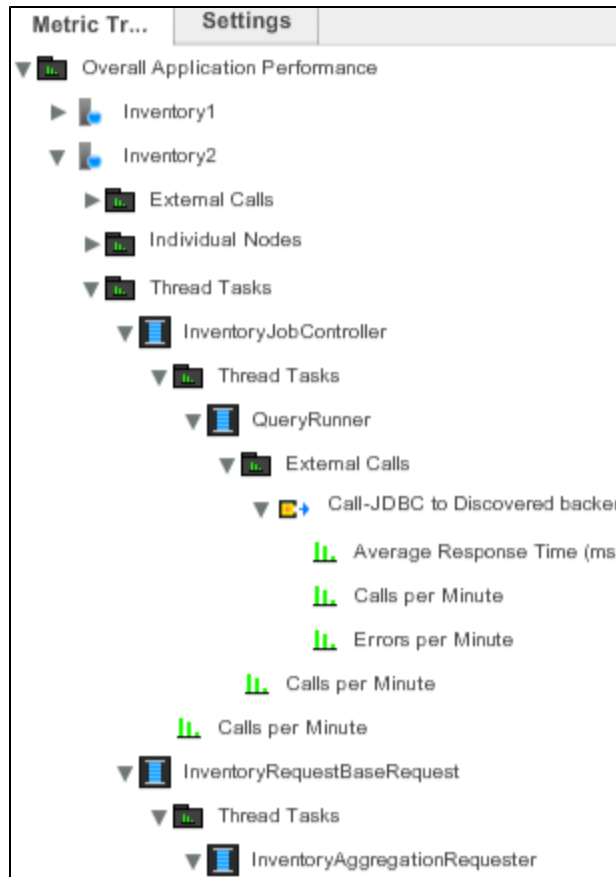
In a multi-threaded transaction, AppDynamics reports key business transaction performance

metrics for individual threads in a Thread Tasks branch of the tier in the Metric Browser. The Thread Tasks branch is created only for multi-threaded transactions.

The Metric Browser path is **Business Transaction Performance -> Business Transactions -> tier-name -> business-transaction-name -> Thread Tasks** as shown here:



Thread Tasks are also reported in tiers under Overall Application Performance, where you can see metrics on specific calls made by each thread in a node or in a tier.



Threads in Call Graphs

When you drill down in a transaction snapshot for a tier with multiple calls, AppDynamics displays the list of calls that you can drill down into.

Select a Call to Drill Down into				
Multiple calls were made to this Tier as part of this Transaction.				
Drill Down into Call		Show:	All Calls	Originating from: Show All
	Exe Time (ms)	Summary	Exit Calls	Start Time
!	10530 ms	Call from (end user)	4 Async. Activity calls (32 ms. max, 8.3 ms. avg.), and 4 JDBC calls (68 ms. max, 17.0 ms. av	10/17/12 1:14:11.247 AM
✓	501 ms	Async Activity (LookupServlet\$1)	No exit calls made.	10/17/12 1:14:21.253 AM
✓	2049 ms	Async Activity (InventoryRequestBaseReq	1 Async. Activity call (0 ms.)	10/17/12 1:14:21.538 AM
!	10422 ms	Async Activity (InventorySupplierLookup)	1 HTTP call (10420 ms.)	10/17/12 1:14:21.713 AM
✓	60 ms	Async Activity (InventoryJobController)	1 Async. Activity call (0 ms.)	10/17/12 1:14:21.716 AM
✓	47 ms	Async Activity (QueryRunner)	4 JDBC calls (10 ms. max, 2.5 ms. avg.)	10/17/12 1:14:21.727 AM

Select a call from the list and double-click or click **Drill Down into Call** to access the call graph.

Diagnostic sessions are automatically triggered based on the average response time of the originating thread of a business transaction. See [Diagnostic Sessions](#).

To configure snapshots based on KPIs of an asynchronous thread, use a custom health rule based on the thread KPI of interest and set up a policy to trigger a diagnostic session on the business transaction. See [Diagnostic Sessions](#).

To Drill Down into Downstream Calls on a Thread

If the call graph indicates Async Activity in the Exit Call/Threads column, you can drill down further into the downstream call on the thread:

Call Drill Down. Exe Time: 10404 ms Timestamp: 10/17/12 1:14:21 AM BT: /inventory_check2/ GUID: aba3c074-a748-41ab-a320-dc6e464e9d11

Execution Time: 10404 ms. Node InvServer3. Timestamp: 10/17/12 1:14:21 AM.

Set as Root Reset Root (?) Show Filters

Name	Time (ms)	Exit Calls / Threads
Servlet - LookupServlet:doGet	20 ms (self) 0.2 %	
Servlet - LookupServlet:processRequest:111	0 ms (self) 0 %	
Servlet - LookupServlet:sleep:205	0 ms (self) 0 %	
java.lang.Thread:sleep	9980 ms (self) 95.9 %	
Proxy For Servlet - LookupServlet:<init>:118	22 ms (self) 0.2 %	Async. Activity
Servlet - LookupServlet:sleep:205	0 ms (self) 0 %	
java.lang.Thread:sleep	185 ms (self) 1.8 %	
Servlet - LookupServlet:do1:163	0 ms (self) 0 %	
Servlet - LookupServlet:do2:168	0 ms (self) 0 %	
Constructor of com.includepkg.InventoryRequestBaseRequest:15	0 ms (self) 0 %	
Constructor of com.includepkg.InventoryRequestBaseRequest:20	0 ms (self) 0 %	
Constructor of com.includepkg.InventoryRequestSubmitter:13	13 ms (self) 0.1 %	Async. Activity
com.includepkg.AnnotatedCustomJob:process:14	0 ms (self) 0 %	
com.async.DB:runQuery:187	0 ms (self) 0 %	
com.async.DB\$DbOperation:execute:30	0 ms (self) 0 %	
com.async.DB\$DbOperation:runUpdateQuery:125	25 ms (self) 0.2 %	JDBC
Proxy For Servlet - LookupServlet:<init>:143	9 ms (self) 0.1 %	Async. Activity

1. Click **Async Activity** in the Exit Calls/Threads Column for the call that you want to drill down from.
2. In the Exit Calls and Async Activities window, click **Drill Down into Downstream Call**.

Exit Calls and Async Activities at LookupServlet\$1.<init>

Type	Details	Count	Time (ms)	% Time	From Tier	To Tier	Downstream Call Time (ms)
Async. Activity	Asynchronous activ	1	1	0.1	Inventory2	Inventory2	501 ms

1 ms

Details

Asynchronous activity identified

Drill Down into Downstream Call

A call graph for the downstream call opens.

Thread Metrics in Health Rules

You can create a custom health rule 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](#).

Learn More

- [Configure Multi-Threaded Transactions for Java](#)
- [Metric Browser](#)
- [Call Graphs](#)
- [Health Rules](#)
- [Configure Health Rules](#)
- [Configure Diagnostic Sessions For Asynchronous Activity](#)

Service Endpoint Monitoring

- [Service Endpoints for Monitoring Specific Services](#)
- [Understand Service Endpoints](#)
- [View Service Endpoint Metrics](#)
 - [Key Performance Indicators and Transaction Scorecard](#)
 - [Service Endpoints in the Metric Browser](#)
- [Configure Service Endpoints](#)
 - [Prerequisites for Configuring Service Endpoints](#)
 - [To configure service endpoints](#)
- [Learn More](#)

Service Endpoints for Monitoring Specific Services

In complex, large-scale applications, some application services may span multiple tiers. If you are an owner of an application service, you may need metrics on that specific service as opposed to metrics from across an entire business transaction or entire tier. Service endpoints allow you to obtain a subset of metrics and associated snapshots for your service so that you can focus on the information truly of interest to you. Using service endpoints, you can define a set of entry points into your specific service to create a customized view in the AppDynamics UI that displays the key performance indicators, associated snapshots, and metrics that affect only that service. You can understand the performance of your service and quickly drill down to snapshots that affect your service, instead of sifting through snapshots for the entire tier or business transaction.



Understand Service Endpoints

Service endpoints are similar to business transactions except that they report metrics only at the entry point and do not track metrics for any downstream segments. Service endpoints are not automatically detected like business transactions are detected; you must specify the entry points for service endpoints. Service endpoints support the same entry point types as business transactions and you configure them in a similar way.

Snapshot information for the service endpoint refers back to the originating business transaction snapshot that goes through service endpoint-defined entry points.

All normal metric operations of metrics are observed for metrics collected by the service endpoint; this includes metrics registration and metric rollups for tiers, limits on number of metrics, and other standard operations. Captured metrics for service endpoints are limited to entry point metrics. Custom metrics are not supported on service endpoints.

Diagnostic sessions cannot be started on the service endpoints; however, you can create diagnostic sessions on the originating business transaction.

- ✔ Service endpoints are supported only on the App Agent for Java at this time.

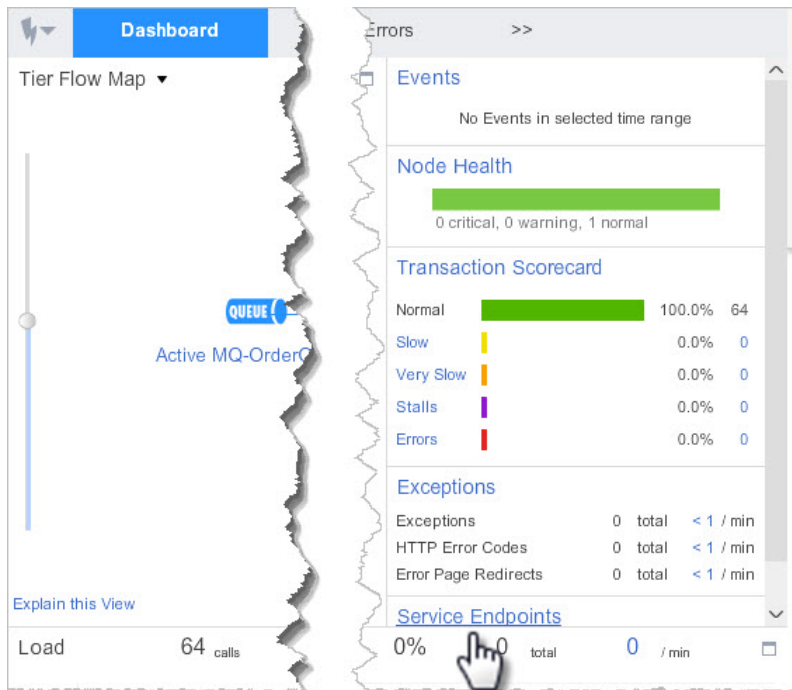
Additional load on the system for service endpoints is negligible. For example, with 1000 agents,

additional metric traffic amounts to only 30K. Approximately three metrics are captured per service endpoint and the agent has a default limit of 100 service endpoints.

View Service Endpoint Metrics

Key Performance Indicators and Transaction Scorecard

1. On the left-hand navigation menu, click a **Tier**.
2. On the right-hand side of the Tier Flow Map, click **Service Endpoints**.



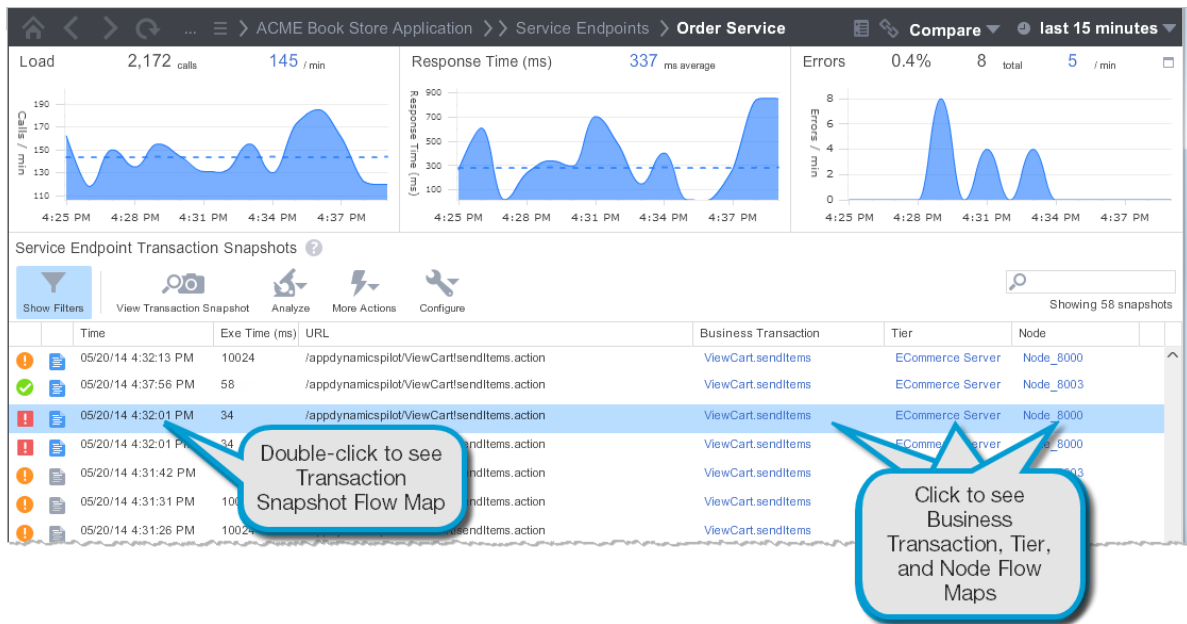
Key performance indicator metrics along with transaction scorecards display.

The screenshot shows the 'Service Endpoints' page for the 'Order Service'. The table displays the following data:

Name	Response	Calls	Calls / min	Errors	% Errors	Type
Order Service	206	2294	153	5	3	POJO

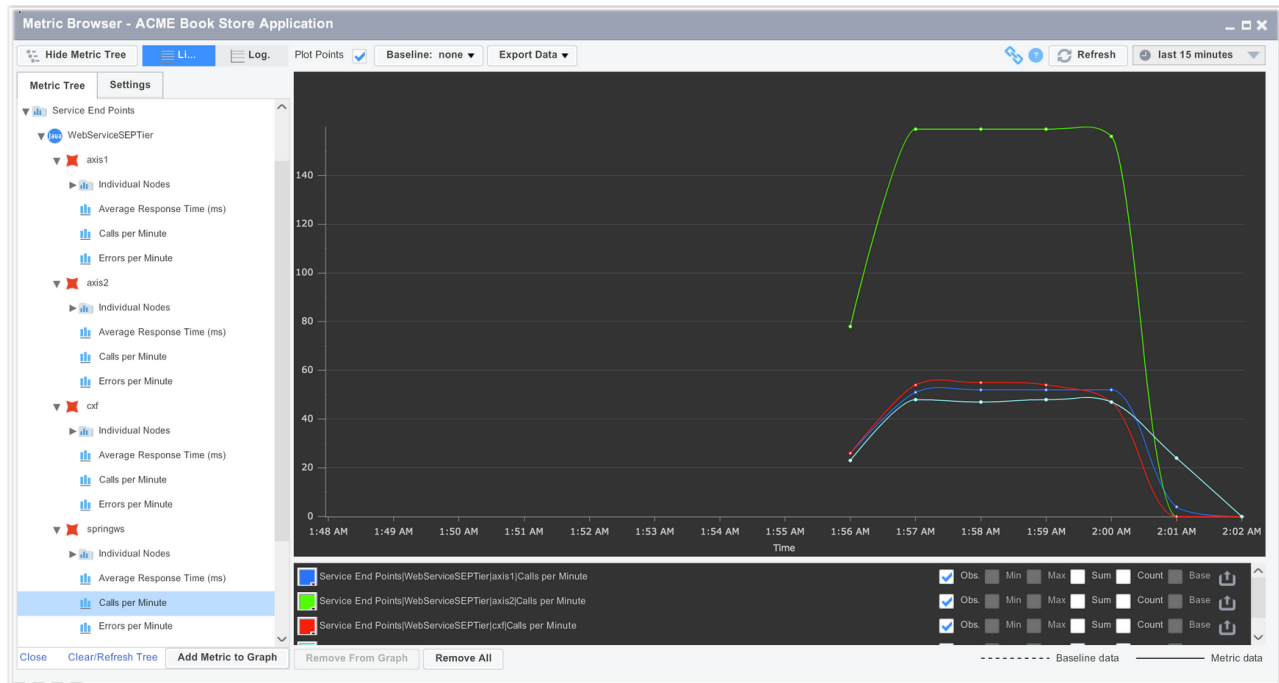
The information displayed relates only to the service endpoints entry points defined for the tier.

3. Double-click a service endpoint name to see the Service Endpoints Transaction Snapshots.
4. Click the snapshot or double-click the Business Transaction, Tier, or Node links for additional details. These details help you to troubleshoot problems with your service.



Service Endpoints in the Metric Browser

At a glance, you can see the performance of your service by looking at the service endpoints in the Metric Browser.



Configure Service Endpoints

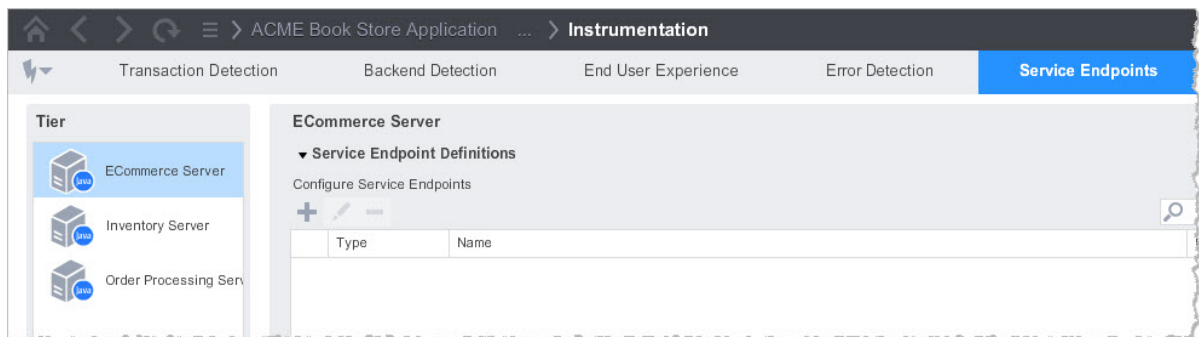
Prerequisites for Configuring Service Endpoints

In order to configure Service Endpoints, your user account must have "Configure Service Endpoints" permissions for the application. For information about configuring user permissions for

applications, see [To Configure the Default Application Permissions](#).

To configure service endpoints

1. Determine which object you want to instrument.
You can instrument service endpoints just as you would business transactions, on the same objects, using the same matching and exclude rules, etc...
2. From the left-hand navigation menu, click **Configure -> Instrumentation -> <Tier> -> Service Endpoints**, where <Tier> is the tier on which the service runs that interests you.
Or
From the left-hand navigation menu, click the tier on which the service runs that interests you, and then click **Service Endpoints -> Configure**.



3. Select the tier where you want to insert the service endpoint.
4. On the **Service Endpoints Definition** panel, click the **+**, and then in the dialog that appears, select the **Entry Point Type**.
5. Define the service endpoint as you would a business transaction.
For information on defining business transactions, see [Configure Business Transaction Detection](#).

Learn More

- [Configure Business Transaction Detection](#)

Monitoring in a Development Environment

Using AppDynamics in a non-production environment is useful for capturing and troubleshooting issues while your applications are under development before you move them into production. To facilitate greater flexibility into the amount of visibility into your environment, AppDynamics provides the following monitoring levels when using the App Agent for Java.

- **Production** is the normal operational mode that optimizes agent performance for your environment, where you have set the balance between transaction visibility and overhead.
- **Development** can be used in non-production environments where overhead is less of a concern. When Development mode is enabled, the system relaxes the various limits associated with data capture. AppDynamics disables all limits that are intended to reduce overhead in order to increase the amount of data captured, such as disabling limits to the number of call graphs and SQL statements to capture. As a safeguard, when the load

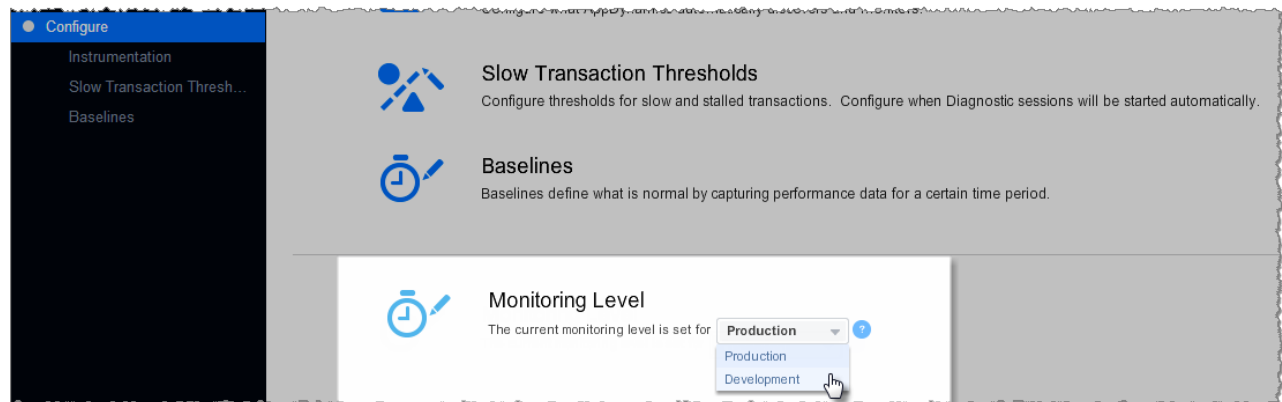
reaches the maximum number of calls per minute you pre-defined or heap utilization reaches the maximum value you defined for Development, AppDynamics re-enables all of the limits.

To Change the Monitoring Level

Prerequisite for Changing the Development Level

- In order to change the Monitoring Level, your user account must have "Configure Monitoring Level (Production/Development)" permissions for the application. For information about configuring user permissions for applications, see [To Configure the Default Application Permissions](#).
- Define the maximum number-of-calls-per-minute safeguard using the [dev-mode-suspend-cpm](#) app agent node property.
- Optionally, define the maximum Java heap utilization percentage for development mode, using the [heap-storage-monitor-devmode-disable-trigger-pct](#) app agent node property.

1. In the left-hand navigation menu of the Application Dashboard, click **Configure**.
2. Click the monitoring level and choose either **Production** or **Development**.



Effects of Using Development Mode

The Development monitoring level controls a set of agent property values that affect the agent behavior to increase visibility, resolution, and decrease data latency at the expense of overhead. The following describes the effects of using Development mode:

- **SQL and JDBC Captures:** When in Development mode any constraints on SQL captures and all other exit calls as well are relaxed. All SQL statements are collected, without a per transaction limit, as well all JDBC calls attached to the method are collected, even if the duration of the call is less than < 10 ms, which is the normal cut off point for JDBC call collection.
- **Call Graphs:** All call graphs are captured during Development mode.
- **Snapshots:** A snapshot is taken for every transaction, including slow and error

transactions.

App Agent Node Properties Ignored During Development Mode

While in Development mode, the values for the following app agent node properties are ignored:

- [max-concurrent-snapshots](#)
- [on-demand-snapshots](#)

Troubleshoot Java Application Problems

Troubleshoot Slow Response Times for Java

- [How Do You Know Response Time is Slow?](#)
- [Troubleshooting Steps](#)
 - [Step 1 - Slow or stalled business transactions?](#)
 - [Step 2 - Slow DB or remote service calls?](#)
 - [Step 3 - Affects 1 or more nodes?](#)
 - [Step 4 - Backend problem?](#)
 - [Step 5 - CPU saturated?](#)
 - [Step 6 - Significant garbage collection activity?](#)
 - [Step 7 - Memory leak?](#)
 - [Step 8 - Resource leak?](#)
 - [None of the above?](#)

How Do You Know Response Time is Slow?

There are many ways you can learn that your application's response time is slow:

- You received an email or SMS alert from AppDynamics (see [Alert and Respond](#)). The alert provides details about the problem that triggered the alert. If the problem is related to slow response time, start troubleshooting at [Step 1](#).
- Someone reported a problem such as "it's taking a long time to check out" or "the app timed out when I tried to add an item to the cart."
The problem is slow response time related to one or more business transactions. Start troubleshooting at [Step 1](#).
- A custom dashboard shows a problem. If the problem is related to slow response time, start troubleshooting at [Step 1](#).
- You are looking at the Application Dashboard for a business application, shown below:



1. Look at the traffic flow lines in the flow map, the Business Transaction Health pane, the Transaction Scorecard pane, and the Response Time graph. If you see problems (yellow or red lines, spikes in response time), the problem is slow response time, and is probably related to your AppDynamics business application. Start troubleshooting at [Step 1](#).
2. Look at the Tier icons in the flow map. If you see yellow or red, you have a problem with one or more nodes, which may or may not result in slow response time. Start troubleshooting at [Step 3](#).
3. Look at the Events pane. If you see red or yellow, the problem reflects a change in application state that is of potential interest, which may or may not result in slow response time. See [Tutorial for Java - Troubleshooting using Events](#).
4. Look at the Server Health pane. If you see red or yellow, the problem reflects a server health rule violation, which may or may not result in slow response time. See [Tutorial for Java - Server Health](#).

Need more help?

- [Application Dashboard](#)
- [Flow Maps](#)

Troubleshooting Steps

The following steps help you determine whether your problem is related to your business application or to your hardware or software infrastructure. Each step shows you how to display detailed information to help you pinpoint the source of the problem and quickly resolve it.

Step 1 - Slow or stalled business transactions ?

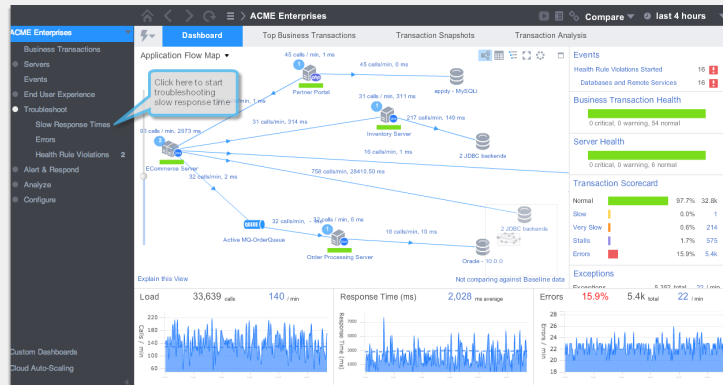
Are there any slow or stalled business transactions?

✓ How do I know?

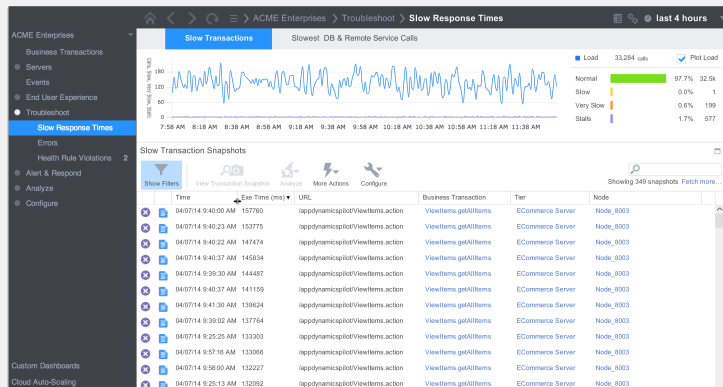
How do I know if business transactions are slow or stalled?

1. Click **Troubleshoot -> Slow Response Times**

You can also access this information from tabs in the various dashboards.



2. Click the **Slow Transactions** tab if it is not selected.



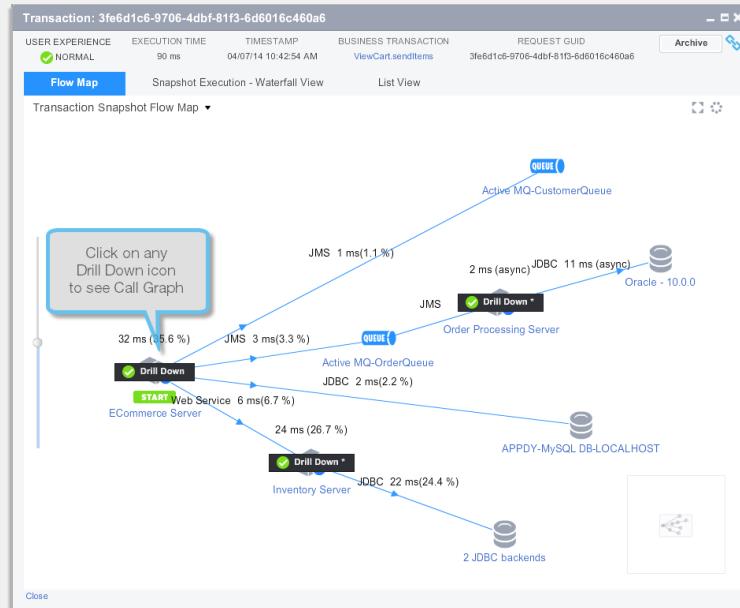
- In the upper pane AppDynamics displays a graph of the slow, very slow, and stalled transactions for the time period specified in the Time Range drop-down menu. If the load is not displayed, you can click the Plot Load checkbox at the upper right to see the load.
- In the lower pane AppDynamics displays the transaction snapshots for slow, very slow, and stalled transactions.

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](#).

Yes – You have one or more slow or stalled transactions, and need to 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. You see the Transaction Flow Map. Choose the Flow Map tab if it is not already selected.



3. Click a Drill Down icon to display a call graph for a problematic part of the transaction. Once you are in the call graph you can look for methods that have a significant response time. In this example, the executeQuery method is responsible for 54.5% of response time.

Call Drill Down: 22 ms 04/07/14 10:42:55 AM BT: ViewCart.sendItems GUID: 3fe6d1c6-9706-4dbf-81f3-6d6016c460a6

Execution Time: 22 ms. Node: Node_0002. Timestamp: 04/07/14 10:42:55 AM.

NAME	Time (ms)	CPU time	Exit Calls
Web Service - org.apache.axis2.receivers.AbstractOutboundMessageReceiver.receive:39	0 ms (self)	0.1%	1
Web Service - org.apache.axis2.receivers.RPCMessageReceiver.invokeBusinessLogic:115	0 ms (self)	0.0%	1
Proxy For Spring Bean - orderServiceTarget.createOrder	0 ms (self)	0.0%	1
Spring Bean - orderServiceTarget.createOrder	0 ms (self)	0.0%	1
Spring Bean - dataService.getConnection:880	1 ms (self)	4.5%	1
Proxy For Spring Bean - orderServiceTarget.invoke	0 ms (self)	0.0%	1
Spring Bean - orderServiceTarget.createOrder:22	3 ms (self)	13.8%	1
com.appdynamics.inventory.QueryExecutor.executeQuery:45	12 ms (self)	54.5%	1
Spring Bean - transactionManager.doCommit:578	2 ms (self)	9.1%	1
com.ctc.wstx.sax.BaseStreamWriter.flush:269	0 ms (self)	0.0%	1
com.ctc.wstx.sax.BaseStreamWriter.flush:184	0 ms (self)	0.0%	1
com.ctc.wstx.sax.UTFWriter.flush:92	1 ms (self)	4.5%	1

This query is taking 54.5% of the execution time

- Click the Information icon in the right column to see more details. Provide this information to the personnel responsible for addressing this issue.

executeQuery:45

Name: com.appdynamics.jdbc.MPreparedStatement.executeQuery
Type: POJO
Class: com.appdynamics.jdbc.MPreparedStatement
Method: executeQuery
Line Number: 45

Execution Time

Self Time: 12 ms
Total Time: 12 ms

54.5 %
54.5 %

CPU Time

CPU: 0 ms, Block: 0 ms, Wait: 10 ms

Exit Calls

Made 1 JDBC exit call(s).

Close

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**.

Need more help?

- Transaction Snapshots

Step 2 - Slow DB or remote service calls?

Is there one or more slow DB or remote service call?

▼ How do I know?

How do I know if I have slow DB or remote service calls?

- Click **Troubleshoot -> Slow Response Times**, then click the **Slowest DB & Remote Service Calls** tab if it is not selected.

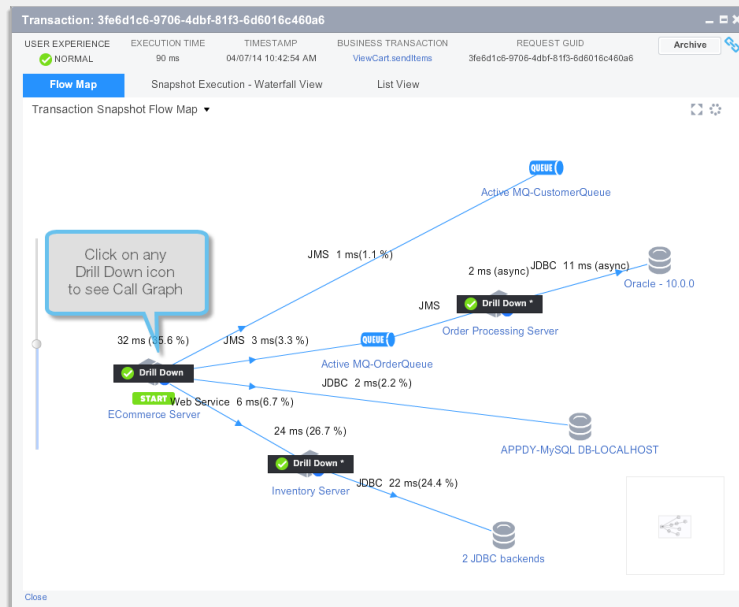
Slow Transactions		Slowest DB & Remote Service Calls			
Call Type		These are the calls with largest observed individual execution time (Max Time) during the specified time range.			
	Call	Avg. Time per Call	Number of Calls	Max Time (ms)	Snapshots
All Calls	SELECT COUNT(1) COUNT FROM ITEM IT1, ITEM IT2	113745.5	576	157726	View snapshots
JDBC	ORDERSERVICE.CREATEORDER	8660	242	10050	View snapshots
JMS	INSERT INTO ORDERREQUEST (ITEM_ID, NOTES) VALUES (?, ?)	302.7	6730	10001	View snapshots
DB	ORDERQUEUE	130	1	130	View snapshots
HTTP	DELETE FROM CART	0.6	7807	69	View snapshots
	INSERT INTO CART (ITEM_ID, USER_ID) VALUES (?, ?)	0.5	7423	80	View snapshots
	INSERT INTO ORDERS (QUANTITY, CREATEDON, ITEMID) VALUES	0.5	6180	61	View snapshots
	DB TRANSACTION COMMIT	0.4	3475	38	View snapshots
	PREPARED STATEMENT BATCH	0.4	6204	38	View snapshots
	SELECT ITEM0_ID AS ID0_0_ ITEM0_QUANTITY AS QUANTITY0_0	0.4	5654	37	View snapshots
Call Details		Correlated Snapshots			
		SELECT COUNT(1) COUNT FROM ITEM IT1, ITEM IT2			

If you see one or more slow calls on this page, the answer to this question is Yes. Otherwise, the answer is No.

No – Go to [Step 3](#).

Yes – You have one or more slow DB or remote service calls, and need to drill down to find the root cause.

1. In the Call Type panel select the type of call for which you want to see information, or select All Calls.
2. Sort by Average Time per Call to display the slowest calls at the top of the list.
3. To see transaction snapshots for the business transaction that is correlated with a slow call, you can:
 - Click the **View Snapshots** link in the right column to display correlated snapshots in a new window.
 - Select the call and click the **Correlated Snapshots** tab in the lower panel to display correlated snapshots at the bottom of the screen.
4. Select a snapshot from the list and click View Transaction Snapshot. Choose the Flow Map tab if it is not already selected, then click a Drill Down icon to display a call graph for a problematic part of the transaction.



5. Once you are in the call graph you can look for methods that have a significant response time. In this example, an Oracle query is responsible for 99.7% of response time. Provide this information to the personnel responsible for addressing this issue.

Transaction: 9144f609-f716-4fa3-815f-23f473247a69		
Call Call Down. Exe Time: 9968 ms. Timestamp: 03/27/14 4:49:37 PM. BT: Fetch Catalog GUID: 9144f609-f716-4fa3-815f-23f473247a69		
SUMMARY	Execution Time: 9968 ms. Node: E-Commerce-Node-8000. Timestamp: 03/27/14 4:50:27 PM	
CALL GRAPH	Set as Root. Reset Root. (?)	
HOT SPOTS		
SQL CALLS		
HTTP PARAMS		
COOKIES		
USER DATA		
ERROR DETAILS		
HARDWARE / MEM		
NODE PROBLEMS		
ADDITIONAL DATA		
	Name	Time (ms)
	StrutsActionProxy.execute	21 ms (self) 0.2 %
	StrutsActionProxy.execute	6 ms (self) 0.1 %
	Proxy For Spring Bean - ItemServiceTarget.getItems	0 ms (self) 0 %
	Proxy For Spring Bean - ItemServiceTarget.invoke	0 ms (self) 0 %
	Spring Bean - ItemServiceTarget.getItems	0 ms (self) 0 %
	Spring Bean - ItemPersistence.getItems	0 ms (self) 0 %
	Spring Bean - oracleQueryExecutor.executeOracleQuery	9841 ms (self) 99.7 %

If there are multiple slow calls, repeat this step until you have resolved them all. However, there may be additional problems you haven't resolved. If any of the tier icons on the flow map show yellow or red, continue to [Step 3](#). Otherwise, you have isolated the problem and don't need to continue with the rest of the steps below.

Need more help?

- [Business Transaction Dashboard](#)
- [Business Transaction Monitoring](#)
- [Business Transactions List](#)
- [Transaction Snapshots](#)

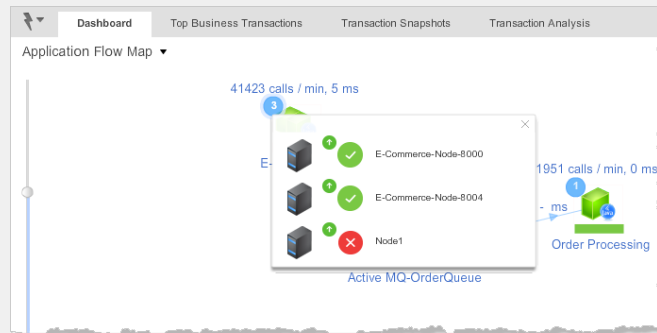
Step 3 - Affects 1 or more nodes?

Is the problem affecting all nodes in the slow tier?

▼ [How do I know?](#)

How do I know if the problem is affecting all nodes?

- In the Application or Tier Flow Map, click the number that represents how many nodes are in the tier. This provides a quick overview of the health of each node in the tier. The small circle icon indicates whether the server is up with the agent reporting, and the larger circle icon indicates Health Rule violation status.



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. (Because only one node is affected, the problem is probably not related to the code itself.)

In the left navigation pane, click **Servers -> App Servers -> <slow tier> -> <problematic node>** to display the Node Dashboard (flow map).



- Click the Dashboard tab to get a view of the overall health of the node.
- Click the Hardware tab; if the problem is hardware-related, contact your IT department.
- Click the Memory tab; sort on various column headings to determine if you need to add memory to the node, configure additional memory for the application, or take some other corrective action.

You have isolated the problem and don't need to continue with the rest of the steps below.

Need more help?

- [Node Dashboard](#)

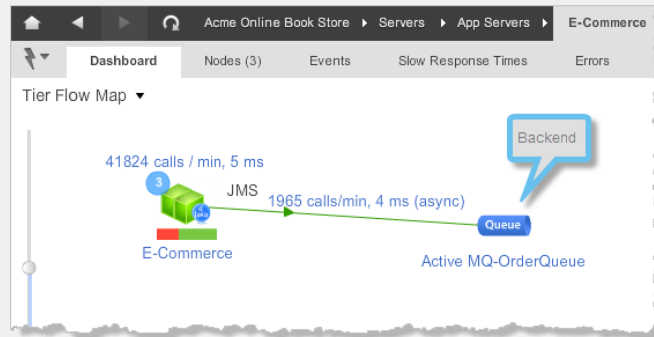
Step 4 - Backend problem?

Are the nodes in the slow tier linked to a backend (database or other remote service) that might be causing your problem?

How do I know?

How do I know if the nodes are linked to a backend (database or other remote service) that might be causing my problem?

- Display the Tier Flow Map. If any nodes are linked to a backend, links to those backends are displayed in the flow map.

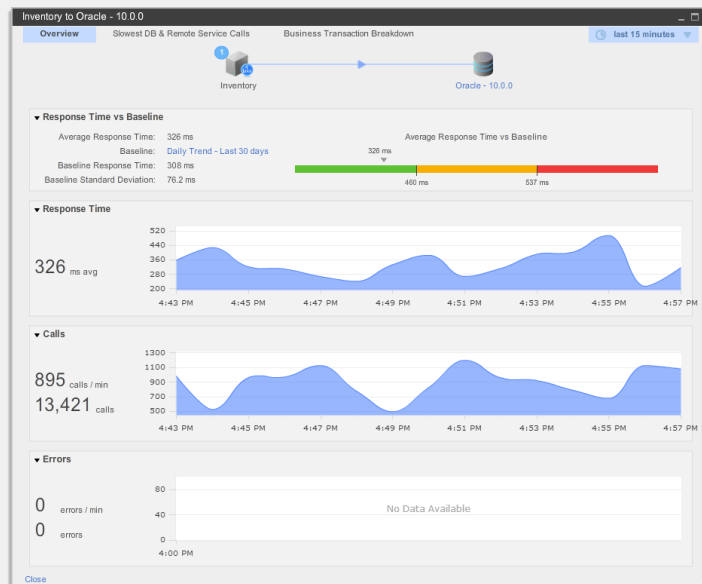


If a backend or the line connecting to a backend is yellow or red, the answer to this question is Yes. Otherwise, the answer is No.

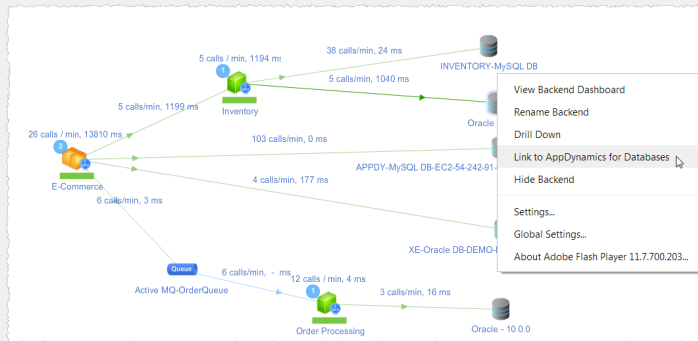
No – Go to [Step 5](#).

Yes –

- Click the line connecting to the backend to see an information window about the backend. (The contents of the information window vary depending on the type of backend.) Use the various tabs to find the source of the issue, or contact the team responsible for that backend.



If the backend is a database, right-click the database icon. You have a number of options that let you see the dashboard, drill down, etc. If you have AppDynamics for Databases, choose Link to AppDynamics for Databases. You can use AppDynamics for Databases to diagnose and resolve any backend issues, or work with your internal DBAs to troubleshoot the database, which is not instrumented in AppDynamics.



You have isolated the problem and don't need to continue with the rest of the steps below.

Need more help?

- [Backend Monitoring](#)
- [Configure Backend Detection for Java](#)
- [AppDynamics for Databases](#)

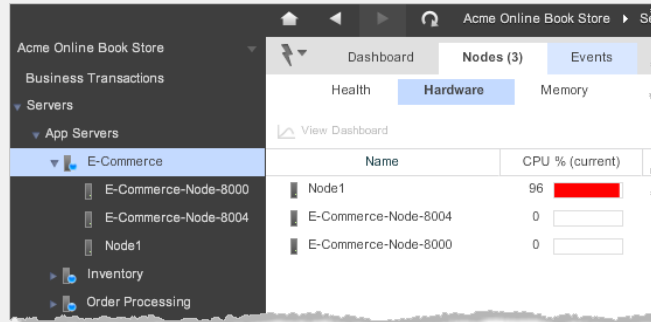
Step 5 - CPU saturated?

Is the CPU of the JVM saturated?

▼ [How do I know?](#)

How do I know if the CPU of the JVM is saturated?

1. Display the Tier Flow Map.
2. Click the Nodes tab, and then click the Hardware tab.
3. Sort by CPU % (current). [Show me how.](#)



If the CPU % is 90 or higher, the answer to this question is Yes. Otherwise, the answer is No.

Yes – Go to [Step 6](#).

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.

You have isolated the problem and don't need to continue with the rest of the steps below.

Need more help?

- [Monitor JVMs](#)

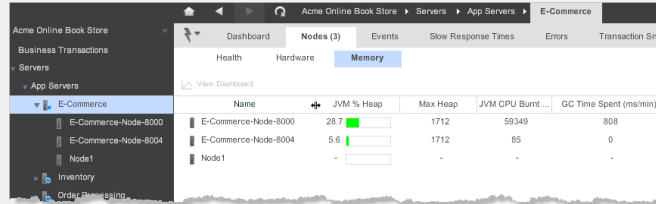
**Step 6 -
Significant
garbage
collection
activity?**

Is there significant garbage collection activity?

✓ [How do I know?](#)

How do I know if there is significant garbage collection activity?

- Display the Tier Flow Map.
- Click the Nodes tab, and then click the Memory tab.
- Sort by GC Time Spent to see how many milliseconds per minute is being spent on GC; 60,000 indicates 100%. [Show me how.](#)



Name	JVM % Heap	Max Heap	JVM CPU Burst ...	GC Time Spent (ms/min)
E-Commerce-Node-8000	28.7	1712	59349	808
E-Commerce-Node-8004	5.6	1712	85	0
Node1	-	-	-	-

If GC Time Spent is higher than 500 ms, the answer to the question in Step 5 is Yes. Otherwise, the answer is No.

Yes – Go to [Step 7](#).

No – Go to [Step 8](#).

Need more help?

- [Memory Usage and Garbage Collection](#)

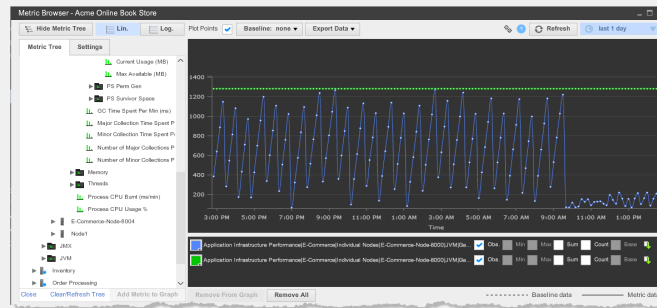
**Step 7 -
Memory
leak?**

Is there a memory leak?

✓ [How do I know?](#)

How do I know if there is 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 scroll down to display the Memory Pool graphs at the bottom of the window.
3. Double-click the PS Old Gen memory pools. **Show me how.**



If memory is not being released (use is trending upward), the answer to this question is Yes. Otherwise, the answer is No.

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 and don't need to continue with the rest of the steps below.

Need more help?

- [Troubleshoot Java Memory Leaks](#)
- [Troubleshoot Java Memory Thrash](#)
- [Configure and Use Object Instance Tracking for Java](#)

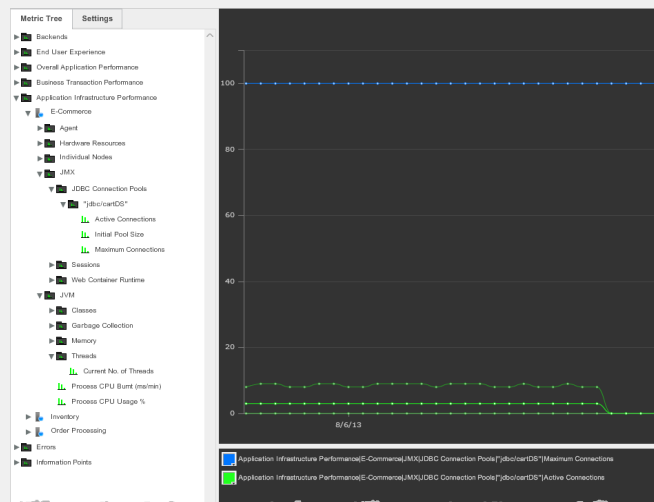
Step 8 - Resource leak?

Is there a resource leak?

▼ [How do I know?](#)

How do I know if there is a resource leak?

1. In the left Navigation pane, go to (for example) **Analyze -> Metric Browser -> Application Infrastructure Performance <slow tier> -> Individual Nodes -> <Problematic node> -> JMX -> JDBC Connection Pools -> <Pool name>**
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.

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](#).

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.

Need more help?

- [Diagnostic Actions](#)

None of the above?

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 a community discussion topic. These discussions are monitored by customers, partners, and AppDynamics staff. Of course, you can also contact AppDynamics support.

Need more help?

- [AppDynamics Pro Documentation](#)
- [Community Discussion Boards](#) (If you don't see AppDynamics Pro as a topic, click Sign In at the upper right corner of the screen.)

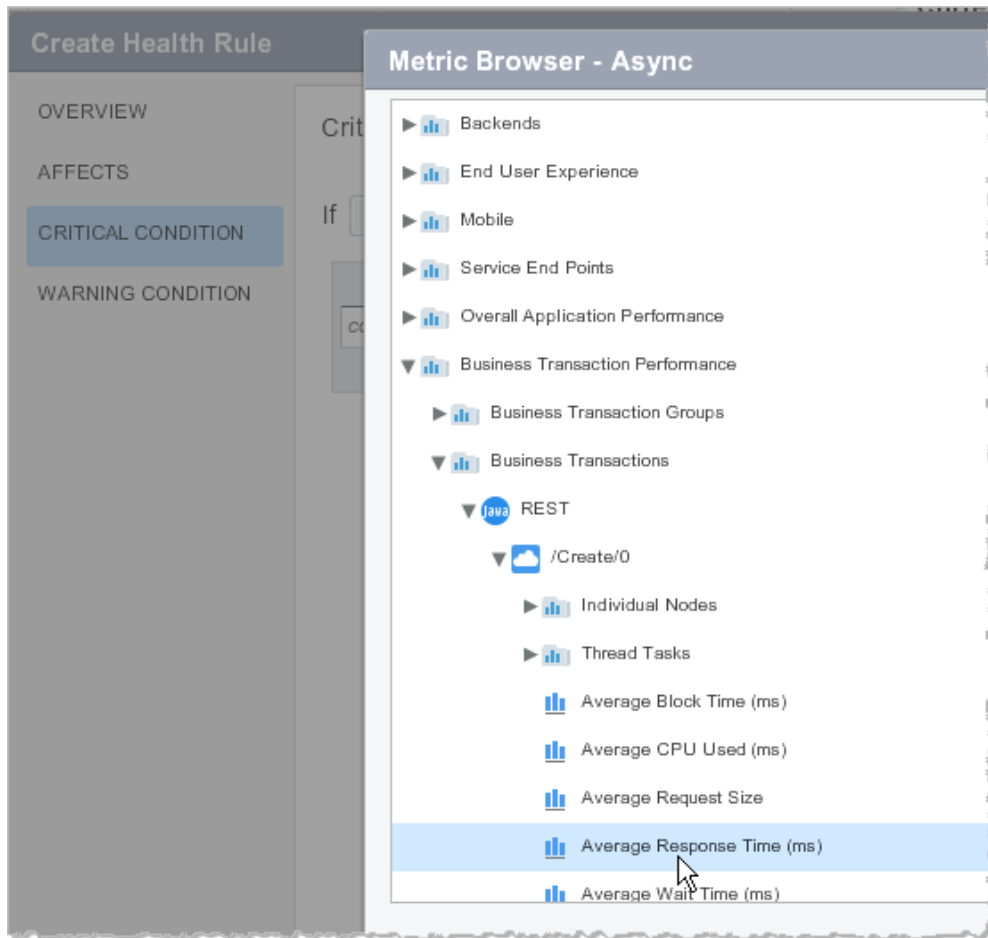
Configure Diagnostic Sessions For Asynchronous Activity

Automatic Diagnostic Sessions For Asynchronous Activity

Diagnostic sessions are triggered based on the performance metrics for a business transaction. The average response time of a business transaction does not include the execution time of its asynchronous activity. If you have asynchronous processing in your application, it might be possible for the originating transaction to execute within normal bounds even though the asynchronous activity takes longer than normal. To diagnose an issue like this, you can create a custom health rule based on the average response time (or other performance metric) of the asynchronous activity and use that health rule to set up a policy that triggers a diagnostic session on the transaction. The general steps to do this are described in the following example that uses a metric for an async thread task.

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.

Troubleshoot Java Memory Issues

Troubleshoot Java Memory Leaks

- [Memory Leaks in a Java Environment](#)
- [AppDynamics Java Automatic Leak Detection](#)
 - [Automatic Leak Detection Support](#)
 - [Conditions for Troubleshooting Java Memory Leaks](#)
- [Workflow to Troubleshoot Memory Leaks](#)
 - [Monitor Memory for Potential JVM Leaks](#)
 - [Enable Memory Leak Detection](#)
 - [Troubleshoot Memory Leaks](#)
 - [Select the Collection Object to Monitor](#)
 - [Use Content Inspection](#)
 - [Use Access Tracking](#)
- [Learn More](#)

Memory Leaks in a Java Environment

While the JVM's garbage collection greatly reduces the opportunities for memory leaks to be introduced into a codebase, it does not eliminate them completely. For example, consider a web page whose code adds the current user object to a static set. In this case, the size of the set grows over time and could eventually use up significant amounts of memory. In general, leaks occur when an application code puts objects in a static collection and does not remove them even when they are no longer needed.

In high workload production environments if the collection is frequently updated, it may cause the applications to crash due to insufficient memory. It could also result in system performance degradation as the operating system starts paging memory to disk.

AppDynamics Java Automatic Leak Detection

AppDynamics automatically tracks every Java collection (for example, HashMap and ArrayList) that meets a set of criteria defined below. The collection size is tracked and a linear regression model identifies whether the collection is potentially leaking. You can then identify the root cause of the leak by tracking frequent accesses of the collection over a period of time.

Once a collection is qualified, its size, or number of elements, is monitored for long term growth trend. A positive growth indicates that the collection is potentially leaking!

Once a leaking collection is identified, the agent automatically triggers diagnostics every 30 minutes to capture a shallow content dump and activity traces of the code path and business transactions that are accessing the collection. By drilling down into any leaking collection monitored by the agent, you can manually trigger Content Summary Capture and Access Tracking sessions. See [Configure Automatic Leak Detection for Java](#)

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 [Configure and Use Custom Memory Structures for Java](#).

Automatic Leak Detection Support

Ensure AppDynamics supports Automatic Leak Detection on your JVM. See [JVM Support](#).

Conditions for Troubleshooting Java Memory Leaks

Automatic Leak Detection uses On Demand Capture Sessions to capture any actively used collections (i.e. any class that implements JDK Map or Collection interface) during the Capture period (default is 10 minutes) and then qualifies them based on the following criteria:

For a collection object to be identified and monitored, it must meet the following conditions:

- The collection has been alive for at least N minutes. Default is 30 minutes, configurable with the [minimum-age-for-evaluation-in-minutes](#) node property.
- The collection has at least N elements. Default is 1000 elements, configurable with the [minimum-number-of-elements-in-collection-to-deep-size](#) node property.
- The collection Deep Size is at least N MB. Default is 5 MB, configurable with the [minimum-size-for-evaluation-in-mb](#) property.

The Deep Size is calculated by traversing recursive object graphs of all the objects in

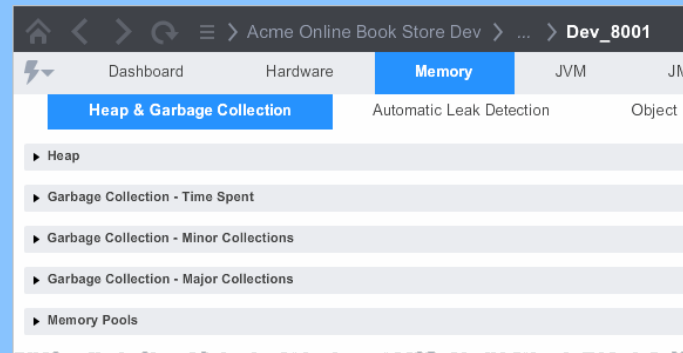
the collection.

See [App Agent Node Properties](#) and [App Agent Node Properties Reference by Type](#).

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:

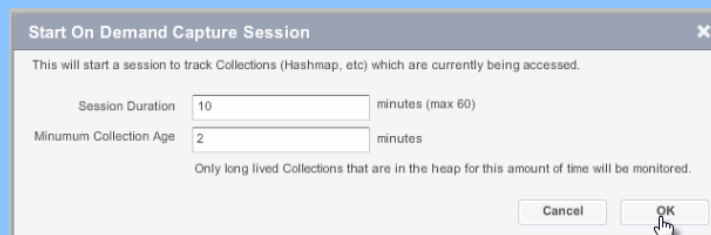
Monitor JVM Memory for potential memory leak



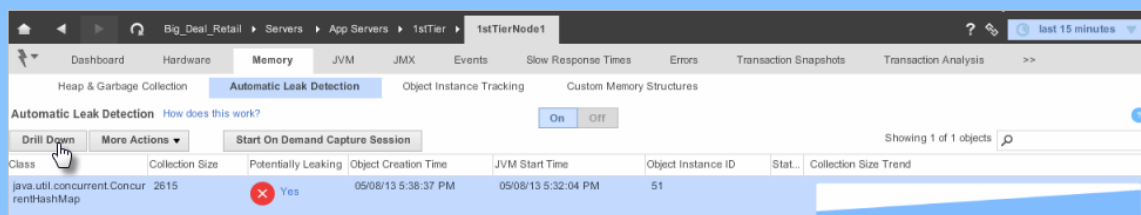
Enable Automatic Leak Detection



Start On Demand Capture Session

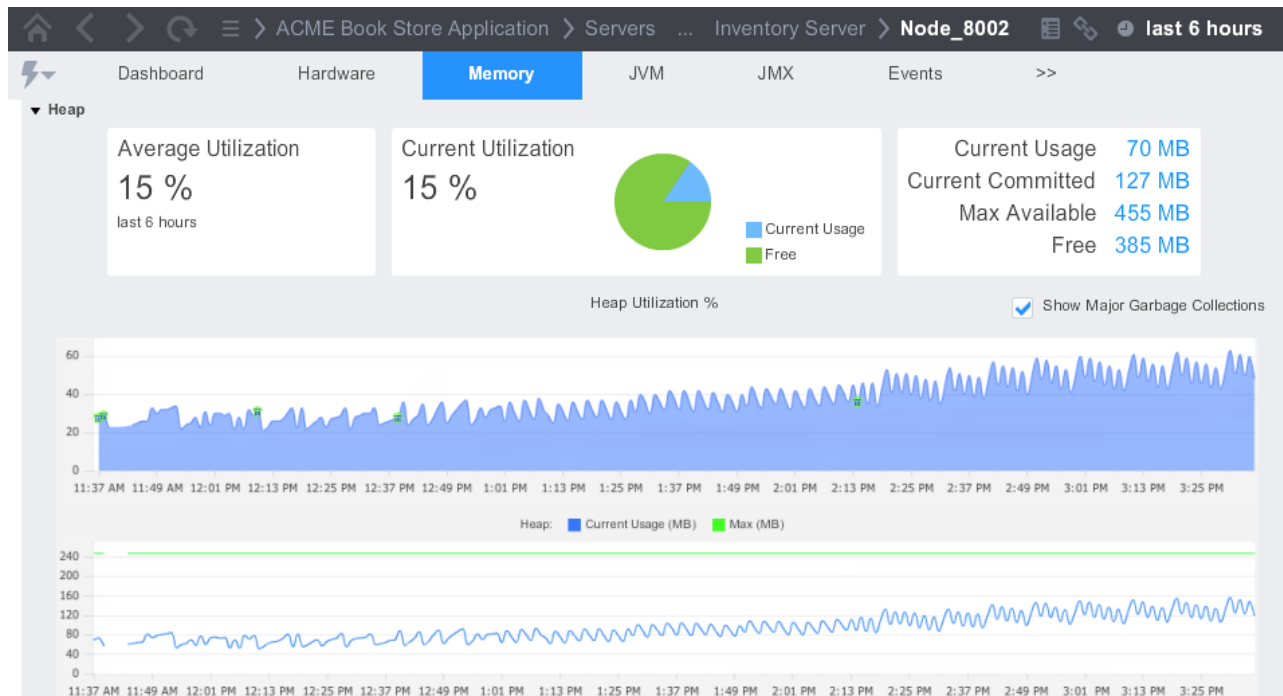


Detect and troubleshoot leaking condition



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.

Class	Collection Size	Potentially Leaking	Object Creation Time	JVM Start Time	Object Instance ID	Status	Collection Size Trend
java.util.concurrent.ConcurrentHashMap	2615	Yes	05/08/13 5:38:37 PM	05/08/13 5:32:04 PM	51		

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 potential memory leak.

of Elements in a Collection

Class	Collection Size	Potentially Leaking	Object Creation Time	JVM Start Time	Object Insta...	Status	Collection Size Trend
-------	-----------------	---------------------	----------------------	----------------	-----------------	--------	-----------------------

Leak Indicator

Diagnostic Session Indicator

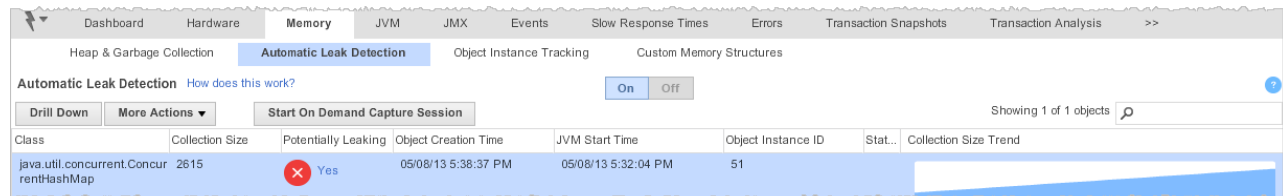
Tip: To identify long-lived collections compare the JVM start time and Object Creation Time.

If you cannot see any captured collections, ensure that you have correct configuration for detecting potential memory leaks.

Enable Memory Leak Detection

Before enabling memory leak detection, identify the potential JVMs that may have a leak. See [Detect Memory Leaks](#).

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 a higher overhead. Turn on Automatic Leak Detection mode only when a memory leak problem is identified and then start an On Demand Capture Session to start monitoring frequently used collections and detect leaking collections.



Turn this mode off after you have identified and resolved the leak.

To achieve optimum performance, start diagnosis on an individual collection at a time.

Troubleshoot Memory Leaks

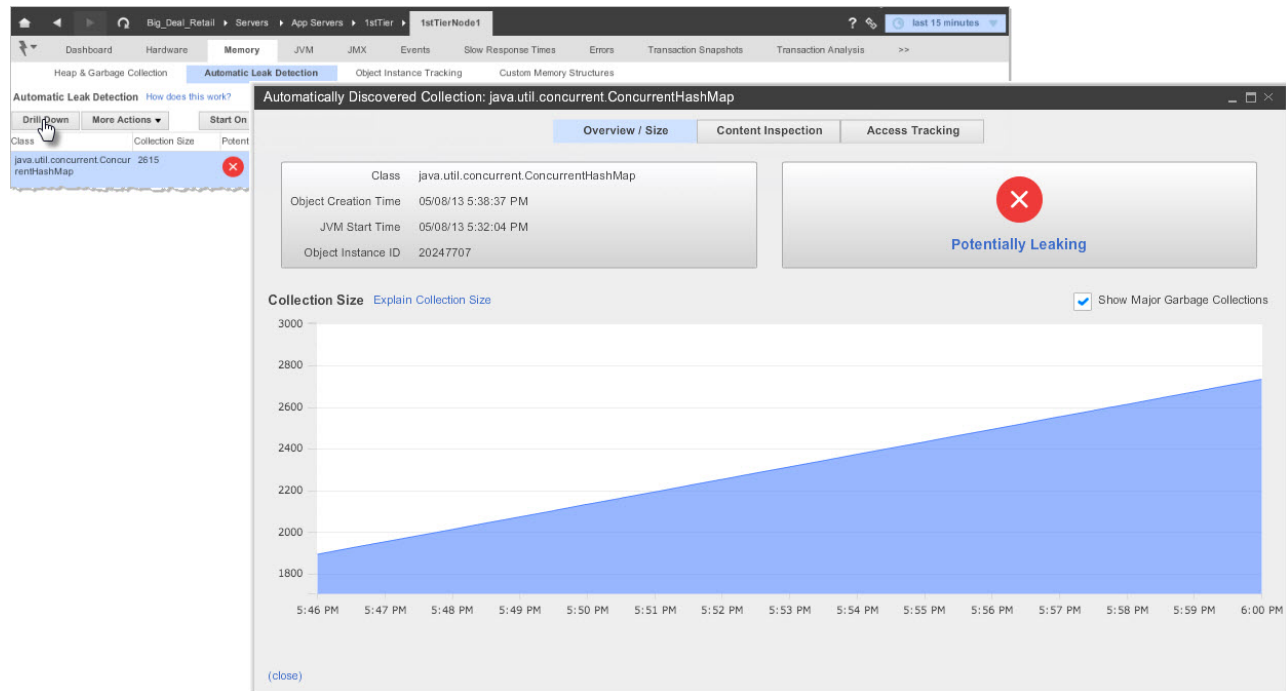
After detecting a potential memory leak, troubleshooting the leak involves performing the following three actions:

- [Select the Collection Object that you want to monitor](#)
- [Use Content Inspection](#)
- [Use Access Tracking](#)

Select the Collection Object to Monitor

1. On the Automatic Leak Detection dashboard, select the name of the class that you want to monitor.
2. Click **Drill Down** on the top left-hand side of the memory leak dashboard. Alternatively right-click the class name and click **Drill Down**.

IMPORTANT: To achieve optimum performance, start the troubleshooting session on a single collection object at a time.



Use Content Inspection

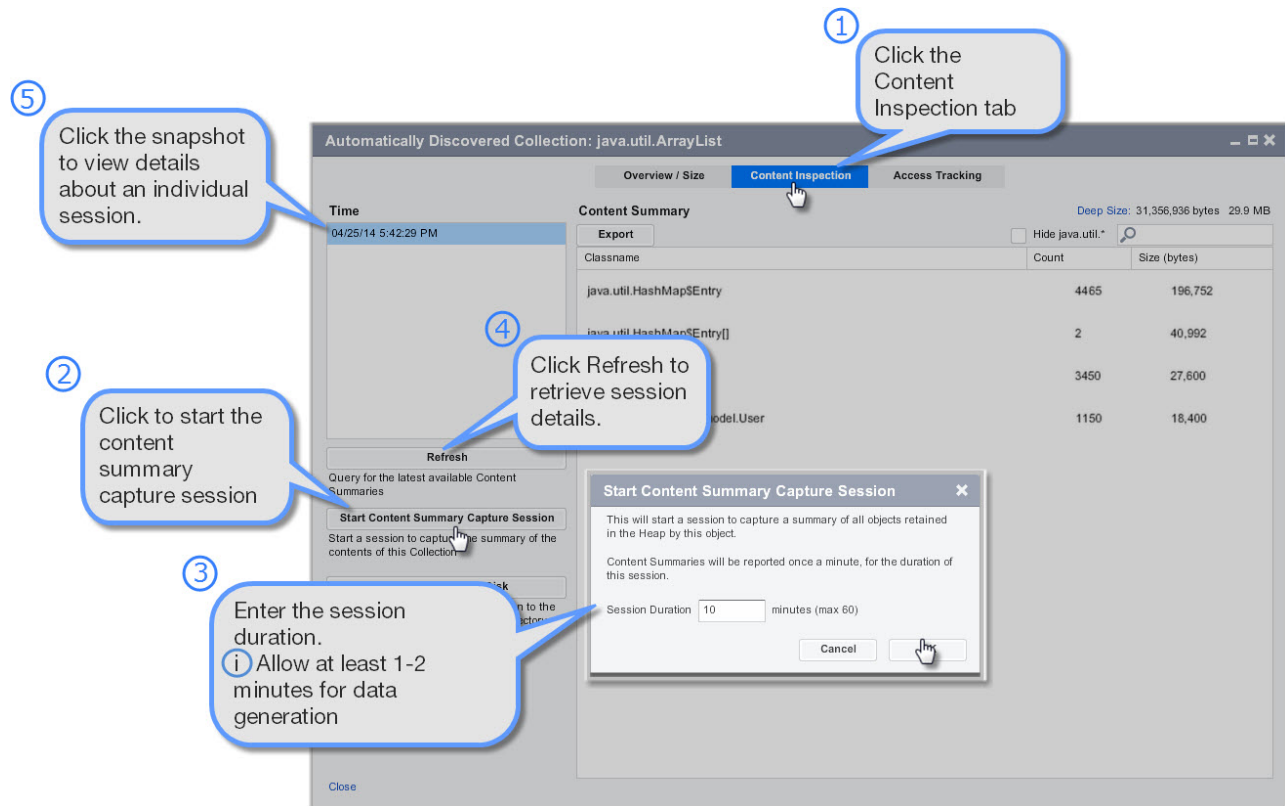
Use Content Inspection to identify which part of the application the collection belongs to so that you can start troubleshooting. It allows monitoring histograms of all the elements in a particular collection.

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. 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.

Exporting Troubleshooting Information

You can also export the troubleshooting information into Excel files using the Export button under Content Summary.



Use Access Tracking

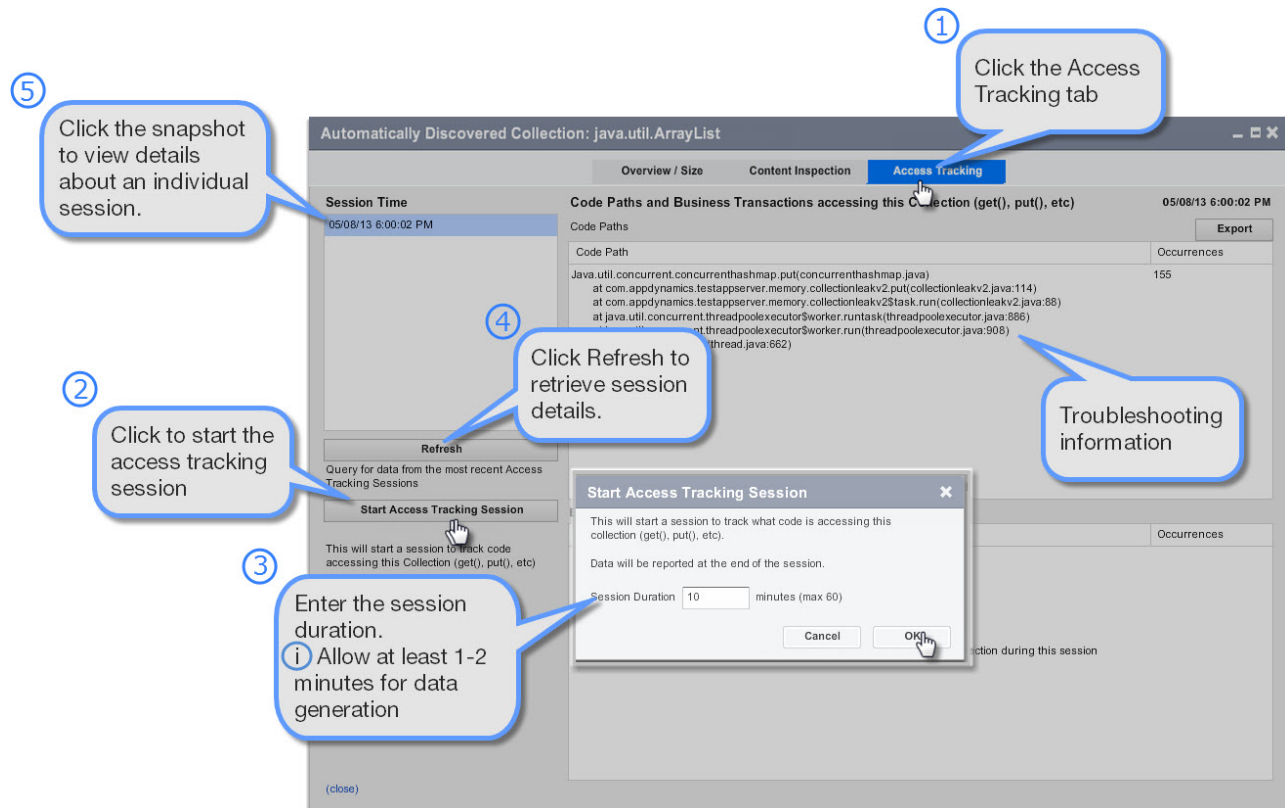
Use Access Tracking to view the actual code paths and business transactions accessing the collections object.

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.

i Exporting Troubleshooting Information

You can also export the troubleshooting information into Excel files using the Export button under Content Summary.



Learn More

- [App Agent Node Properties](#)
- [Monitor JVMs](#)
- [Metric Browser](#)

Troubleshoot Java Memory Thrash

- [Memory Thrash and Object Instance Tracking](#)
- [Prerequisites for Object Instance Tracking](#)
 - [Specifying the Classpath](#)
- [Workflow for Detecting and Troubleshooting Memory Thrash](#)
- [Analyzing Memory Thrash](#)
 - [To analyze memory thrash problems](#)
 - [To verify memory thrash](#)
 - [Troubleshooting Java Memory Thrash Using Allocation Tracking](#)
 - [To use allocation tracking](#)

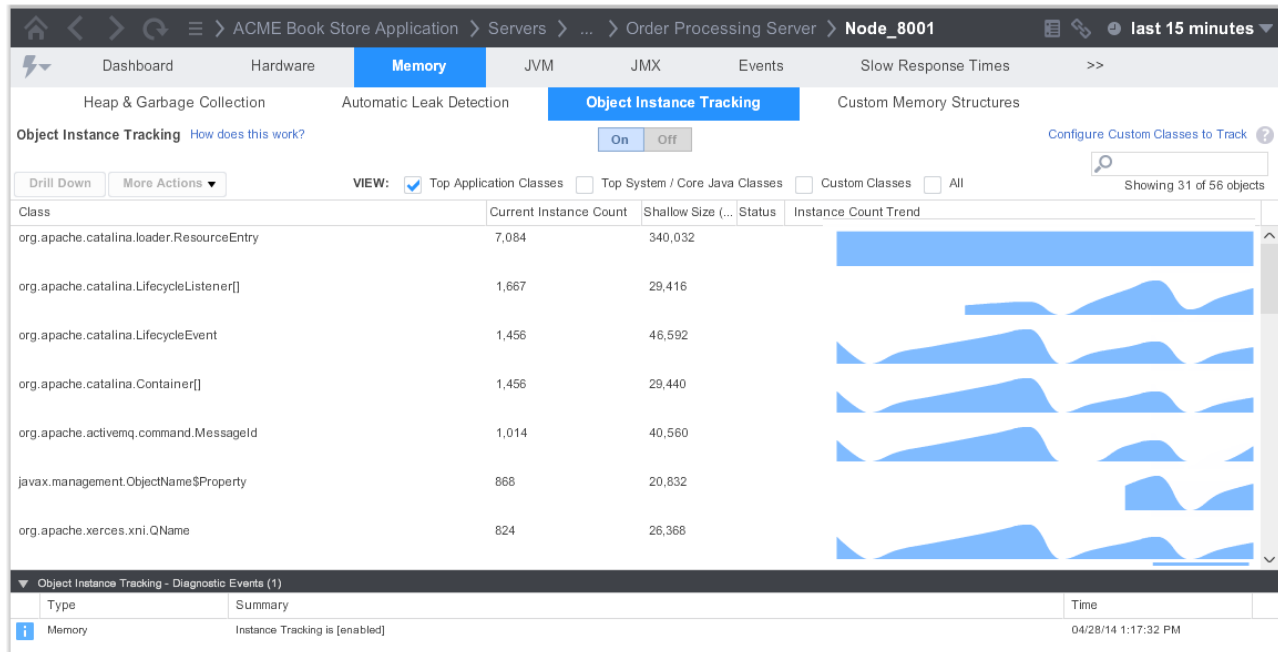
Memory Thrash and Object Instance Tracking

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 [Configure and Use Object Instance Tracking for Java](#).

AppDynamics automatically tracks the following classes:

- Application Classes
- System Classes

The Object Instance Tracking feature maps a histogram of every object in the JVM. The Object Instance Tracking dashboard not only provides the number of instances for a particular class but also provides the shallow memory size (the memory footprint of the object and the primitives it contains) used by all the instances.



Prerequisites for Object Instance Tracking

- Object Instance Tracking can be used only for Sun JVM v1.6.x and later.
- If you are running with the JDK then tools.jar will probably be setup correctly, but if you are running with the JRE you must add tools.jar to JRE_HOME/lib/ext and restart the JVM for this feature to start working. You can find the tools.jar file in JAVA_HOME/lib/tools.jar.
- 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 a JDK tool, set the classpath using the -classpath option. This sets the classpath for the application only. For example:

On Windows

```
java -classpath <complete-path-to-tools.jar>;%CLASSPATH% -jar myApp.jar
```

OR

On Unix

```
java -classpath <complete-path-to-tools.jar>:$CLASSPATH -jar myApp.jar
```

Alternatively, you can set the CLASSPATH variable for your entire environment. For example:

On Windows


```
SET CLASSPATH=%CLASSPATH%;%JAVA_HOME%\lib\tools.jar
```

On Unix

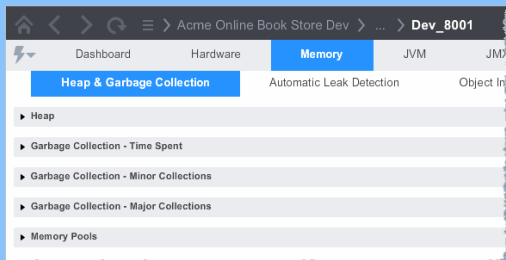
```
CLASSPATH=$CLASSPATH:$JAVA_HOME/lib/tools.jar
```

Workflow for Detecting and Troubleshooting Memory Thrash

The following diagram outlines the workflow for monitoring and troubleshooting memory thrash problems in a production environment.

-  To monitor memory leaks, on the node dashboard, use the Memory -> Automatic Leak Detection subtab. See [Troubleshoot Java Memory Leaks](#).

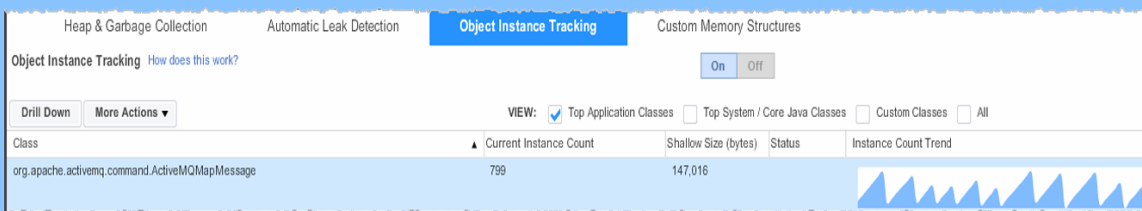
Monitor JVM Memory



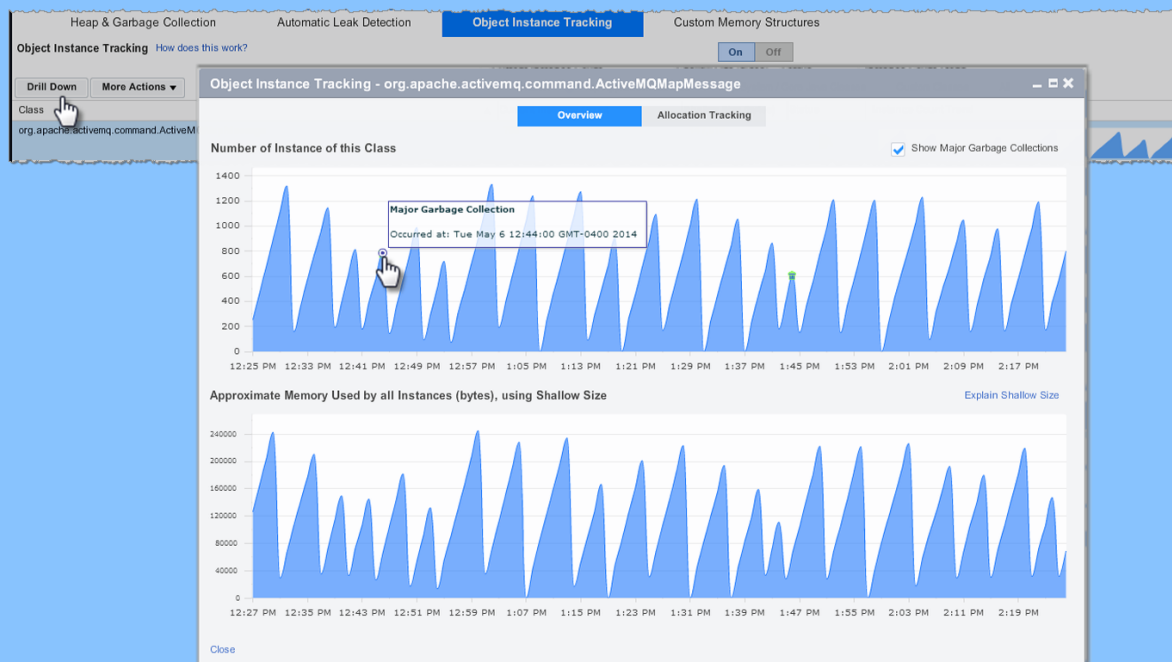
Enable Object Instance Tracking

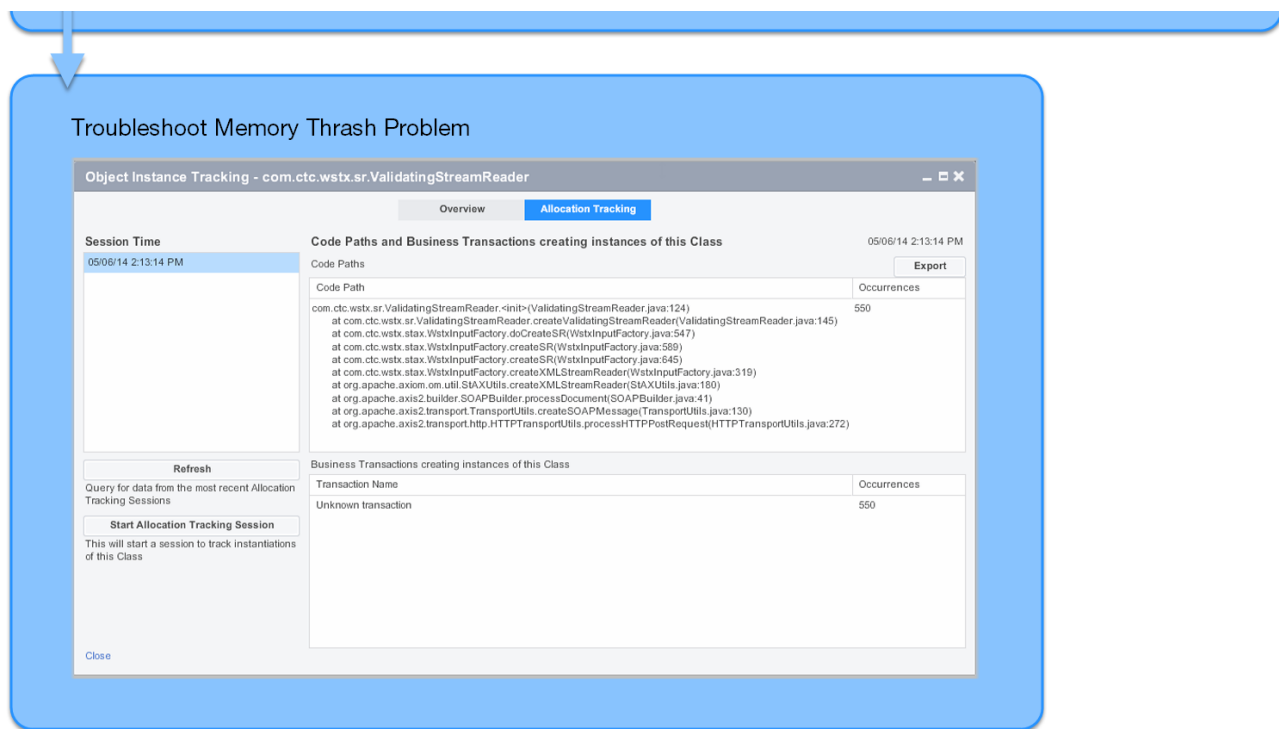


Detect Memory Thrash Problem



Verify Memory Thrash Problem





Analyzing Memory Thrash


To analyze memory thrash problems

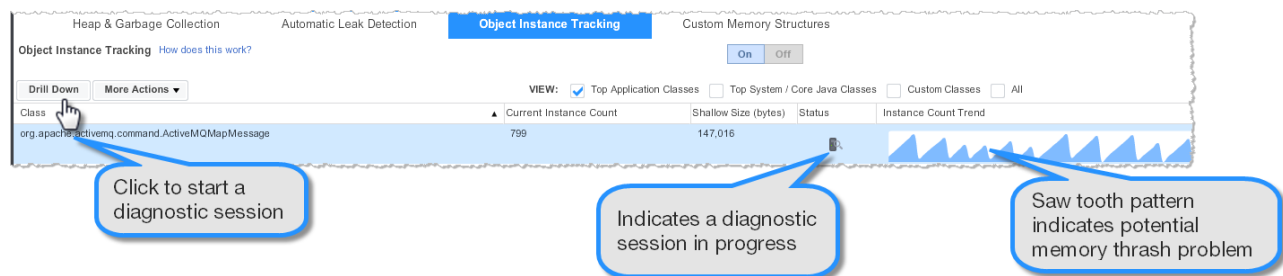
Once a memory thrash problem is identified in a particular collection, start the diagnostic session by drilling down into the suspected problematic class.

1. 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.

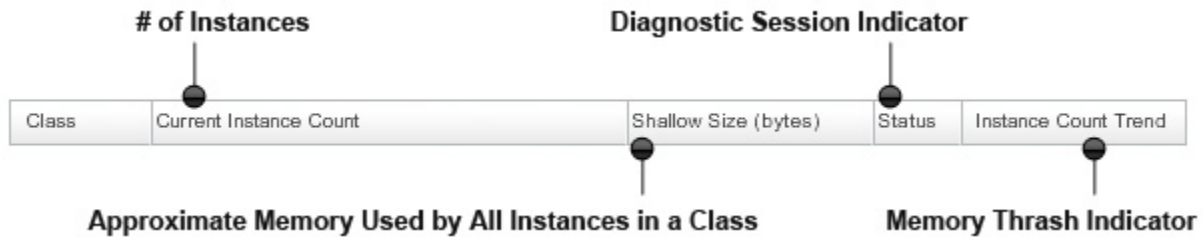
i For optimal performance, trigger a drill down action on a single instance or class name at a time.

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.

The following provides more detail on the meaning of each of the columns on the Object Instance Tracking dashboard.



Prime indicators of memory thrash problems are:

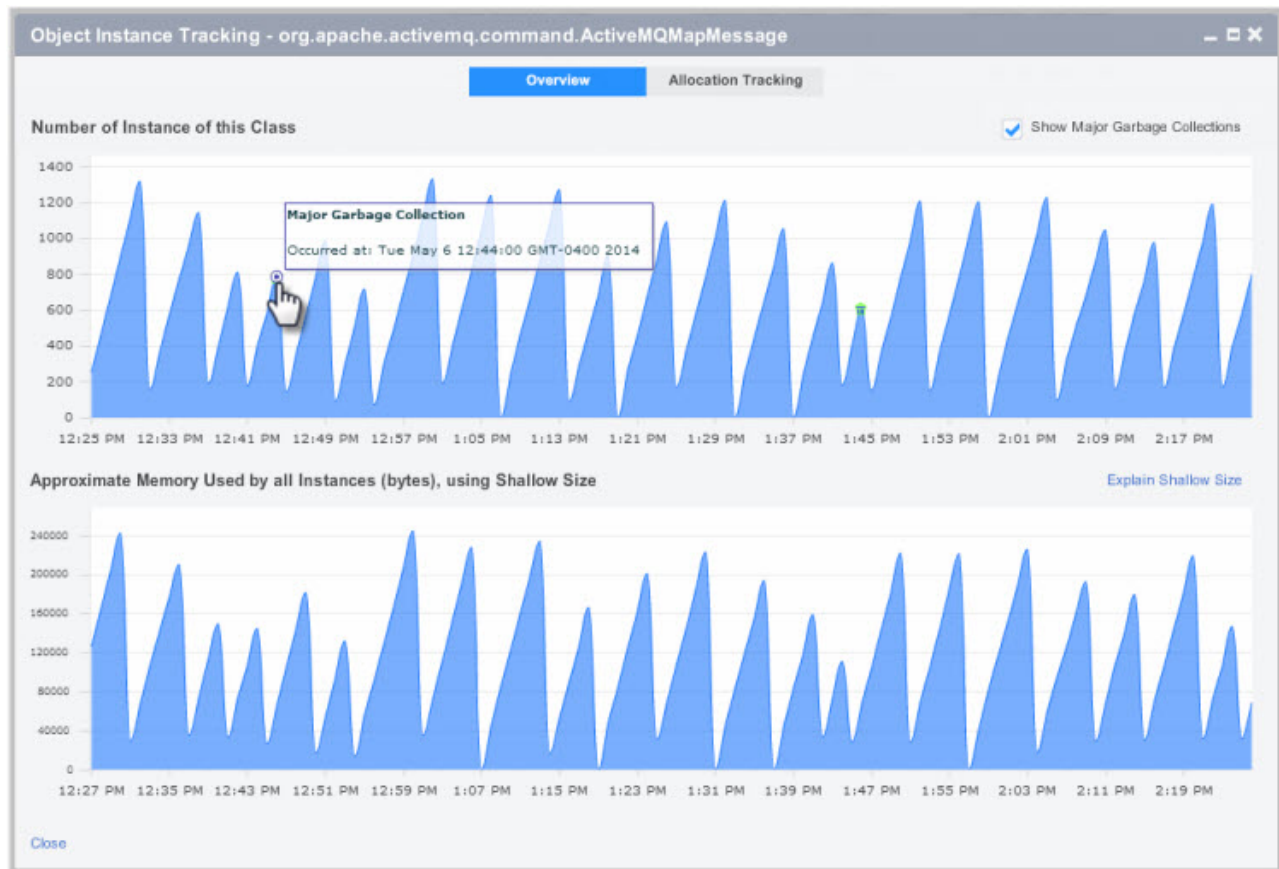
- **Current Instance Count:** A high number indicates possible allocation of large number of temporary objects.
- **Shallow Size:** A large number for shallow size signals potential memory thrash.
- **Instance Count Trend:** A saw wave is an instant indication of memory thrash.

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](#).

To verify memory thrash

1. Select the class name to monitor and click **Drill Down** at the top of the Object Instance Tracking dashboard.
2. 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 potential leak and not a memory thrash problem. See [Troubleshoot Java Memory Leaks](#).

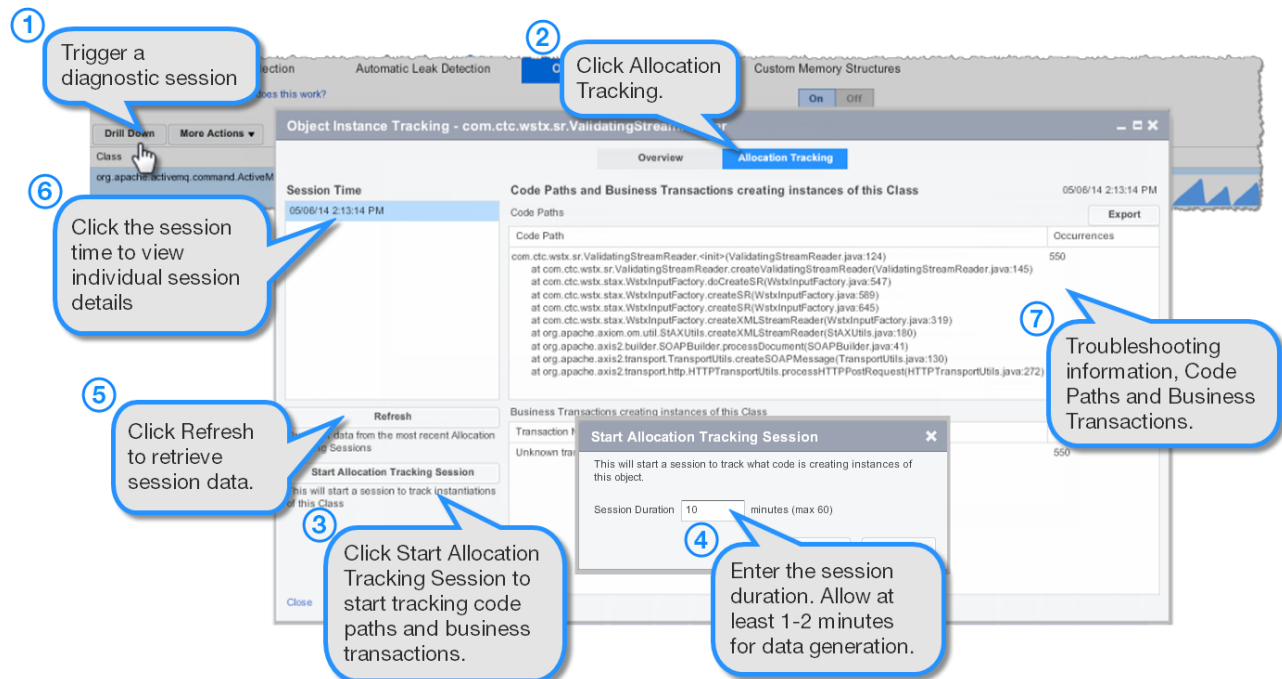
Troubleshooting Java Memory Thrash Using Allocation Tracking

Allocation Tracking tracks all the code paths and those business transactions that are allocating instances of a particular class.

Allocation tracking 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 1-2 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.



Detect Code Deadlocks for Java

- Code Deadlocks and their Causes
 - Finding Deadlocks using the Events List
 - To Examine a Code Deadlock
 - Finding Deadlocks Using the REST API
- Learn More

i Read a real-life story about how AppDynamics helped identify code deadlocks and reduce the risk to revenue!

By default the agent detects code deadlocks. You can find deadlocks and see their details using the Events list or the REST API.

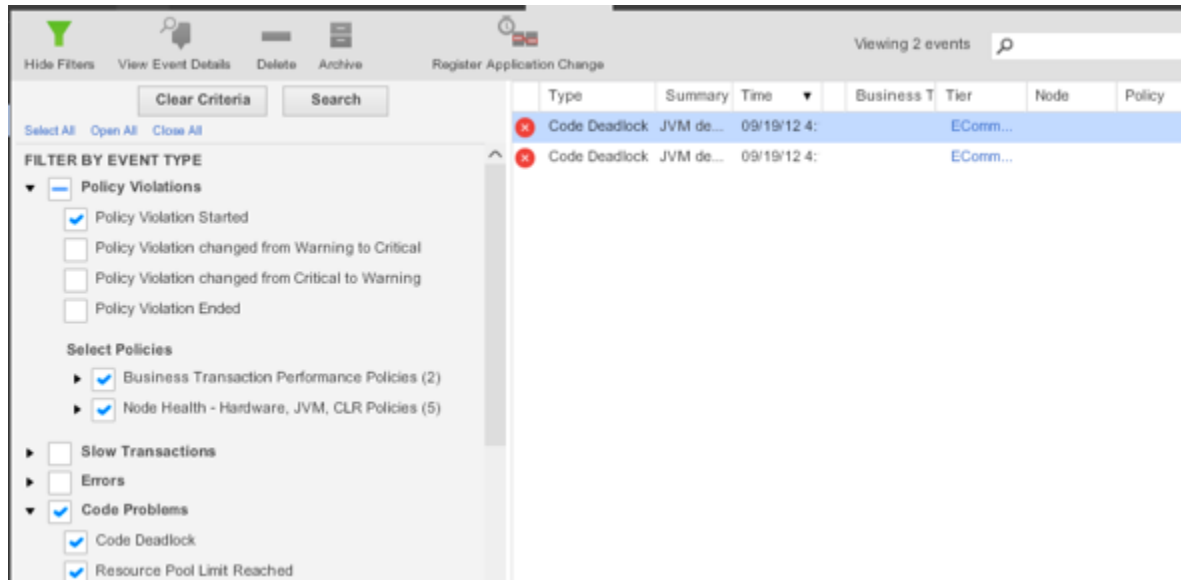
Code Deadlocks and their Causes

In a multi-threading development environment, 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. See [Filter and Analyze Events](#). The following list shows two deadlocks in the ECommerce tier.



Hide Filters View Event Details Delete Archive Register Application Change Viewing 2 events

Clear Criteria Search

Select All Open All Close All

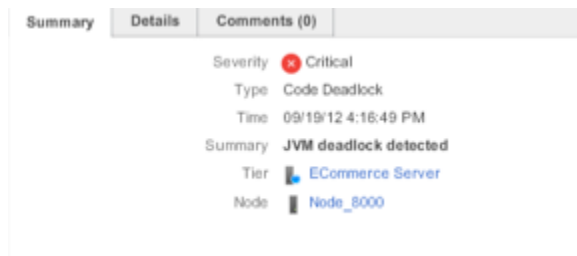
FILTER BY EVENT TYPE

- ☒ Policy Violations
 - ☒ Policy Violation Started
 - ☐ Policy Violation changed from Warning to Critical
 - ☐ Policy Violation changed from Critical to Warning
 - ☐ Policy Violation Ended
- Select Policies**
 - ☒ Business Transaction Performance Policies (2)
 - ☒ Node Health - Hardware, JVM, CLR Policies (5)
- ☐ Slow Transactions
- ☐ Errors
- ☒ Code Problems
 - ☒ Code Deadlock
 - ☒ Resource Pool Limit Reached

Type	Summary	Time	Business T	Tier	Node	Policy
Code Deadlock	JVM de...	09/19/12 4:				EComm...
Code Deadlock	JVM de...	09/19/12 4:				EComm...

To Examine a Code Deadlock

1. Double-click the deadlock event in the events list.
The Code Deadlock **Summary** tab displays.



Summary Details Comments (0)

Severity Critical

Type Code Deadlock

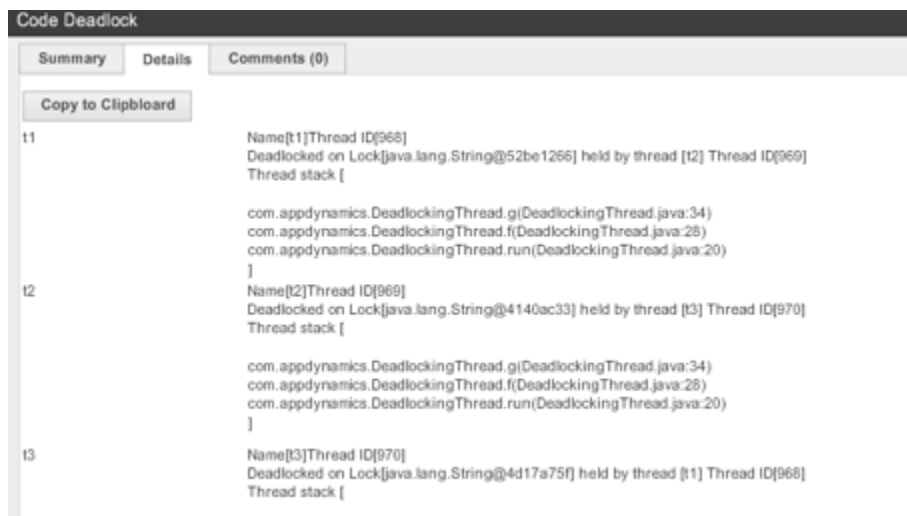
Time 09/19/12 4:16:49 PM

Summary **JVM deadlock detected**

Tier ECommerce Server

Node Node_8000

2. To see details about the deadlock click the **Details** tab and scroll down.



Code Deadlock

Summary Details Comments (0)

Copy to Clipboard

```

11 Name[t1]Thread ID[968]
   Deadlocked on Lock[java.lang.String@52be1266] held by thread [t2] Thread ID[969]
   Thread stack [
       com.appdynamics.DeadlockingThread.g(DeadlockingThread.java:34)
       com.appdynamics.DeadlockingThread.f(DeadlockingThread.java:28)
       com.appdynamics.DeadlockingThread.run(DeadlockingThread.java:20)
   ]
12 Name[t2]Thread ID[969]
   Deadlocked on Lock[java.lang.String@4140ac33] held by thread [t3] Thread ID[970]
   Thread stack [
       com.appdynamics.DeadlockingThread.g(DeadlockingThread.java:34)
       com.appdynamics.DeadlockingThread.f(DeadlockingThread.java:28)
       com.appdynamics.DeadlockingThread.run(DeadlockingThread.java:20)
   ]
13 Name[t3]Thread ID[970]
   Deadlocked on Lock[java.lang.String@4d17a75f] held by thread [t1] Thread ID[968]
   Thread stack [

```

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](#).

Learn More

- [Use the AppDynamics REST API](#)

Tutorials for Java

This section provides tutorials for tasks in AppDynamics.

Quick Tour of the User Interface

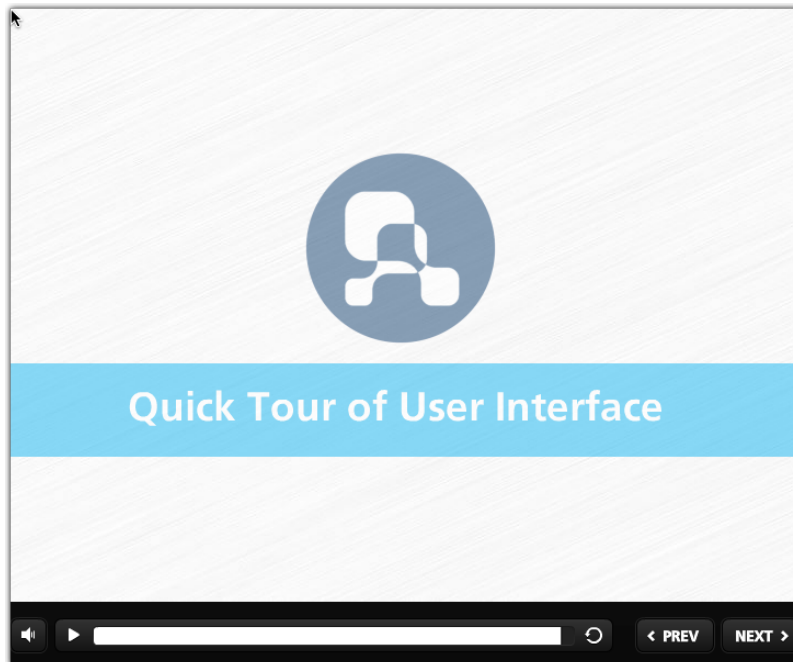


Troubleshooting Application Errors

Identifying and troubleshooting errors in your Java application.

Overview Tutorials for Java

Quick Tour of the User Interface



Use AppDynamics for the First Time with Java

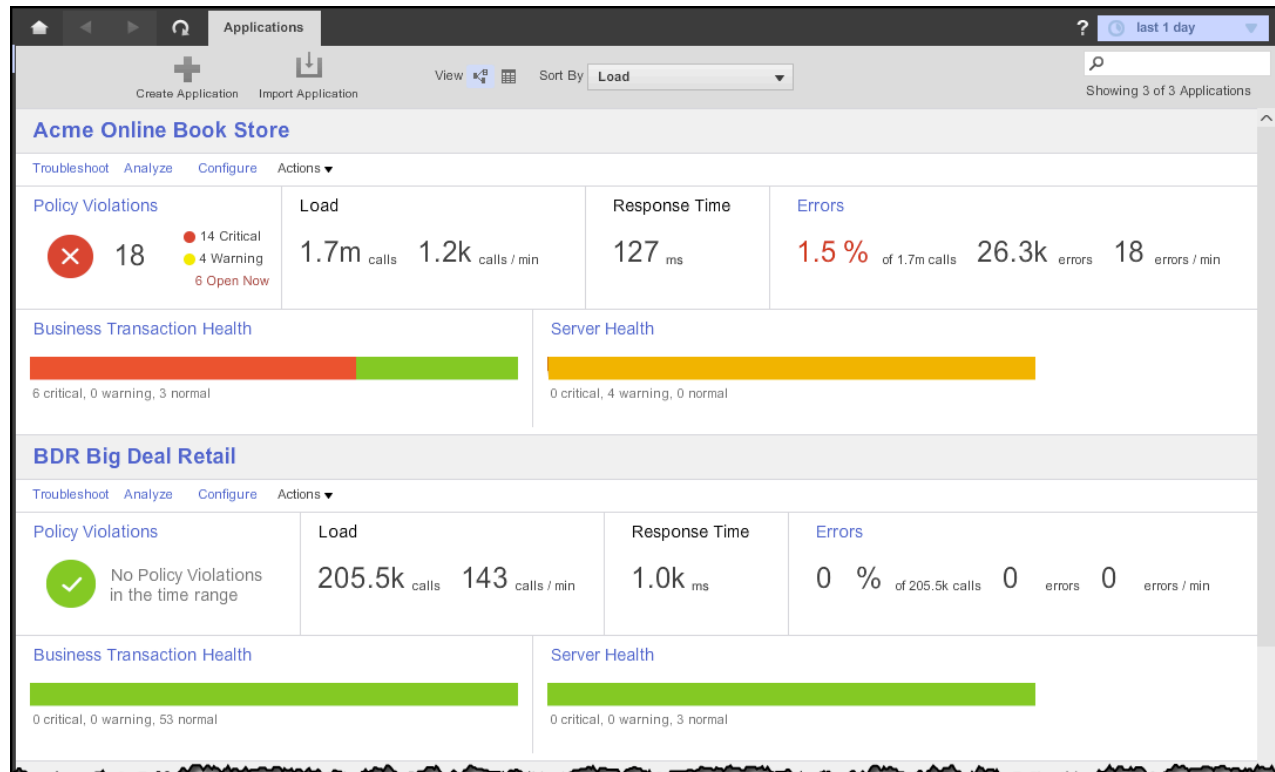
- [All Applications Dashboard](#)
- [Application Dashboard](#)
 - [Time Range](#)
 - [Flow Map and KPIs](#)
 - [Events](#)
 - [Transaction Scorecard](#)
 - [Exceptions and Errors](#)
- [More Tutorials](#)

This topic assumes that an application is already configured in AppDynamics, and uses the Acme Online application as the example. It also assumes that you have already logged in to AppDynamics.

This topic gives you an overview of how AppDynamics detects actual and potential problems that users may experience in your application - transactions that are slow, stalled or have errors - and helps you easily identify the root causes.

All Applications Dashboard

When you log into the Controller UI you see the All Applications dashboard.



The All Applications dashboard shows high-level performance information about one or more business applications. Load, response time, and errors are standard metrics that AppDynamics calls "key performance indicators" or "KPIs". The others are:

Health Rule Violations and Policies: AppDynamics lets you [define a health rule](#), which consists of a condition or a set of conditions based on metrics exceeding predefined thresholds or dynamic baselines. You can then use health rules in policies to automate optional remedial actions to take if the conditions trigger. AppDynamics also provides default health rules to help you get started.

Business Transaction Health: The health indicators are a visual summary of the extent to which a business transaction is experiencing critical and warning health rule violations. See the [slow transactions tutorial](#).

Server Health: Additional visual indicators that track how well the server infrastructure is performing. See the [server health tutorial](#).

Application Dashboard

Click an application to monitor, one that has some traffic running through it. The Application dashboard gives you a view of how well the application is performing.



You see the dashboard for your application. The flow map on the left gives you an overview of your servers (application servers, databases, remote servers such as message queues, etc.) and metrics for the calls between them. Click, hold and move the icons around to arrange the flow map. Use the scale slider and mini-map to change the view.

Time Range

From the time range drop-down in the upper-right corner select the time range over which to monitor - the last 15 minutes, the last couple of hours, the last couple of days or weeks. Try a few different time ranges and see how the dashboard data changes.

Flow Map and KPIs

In the flow map, click any of the blue lines to see more detail on the aggregated key performance metrics (load, average response time and errors) between the two servers. See the [flow maps tutorial](#).

The graphs at the bottom of the dashboard show the key performance indicators over the selected time range for the entire application.

Events

An event represents a change in application state. The Events pane lists the important events occurring in the application environment. See the [events tutorial](#).

Transaction Scorecard

The Transaction Scorecard panel shows metrics about business transactions within the specified time range, covering the percentage of instances that are normal slow, very slow, stalled or have errors. Slow and very slow transactions have completed. Stalled transactions never completed or

timed out. [Configurable thresholds](#) define the level of performance for the slow, very slow and stalled categories. See the [Transaction Scorecard tutorial](#).

Exceptions and Errors

An exception is a code-logged message outside the context of a business transaction. An error is a departure from the expected behavior of a business transaction, which prevents the transaction from working properly. See the [exceptions tutorial](#).

More Tutorials

Monitoring Tutorials for Java

Tutorial for Java - Events

- [Monitoring Events](#)
- [Filtering Events](#)

Monitoring Events

1. From an application, tier or node dashboard, look at the **Events** panel.

The Events Panel shows the type of event, a synopsis, the timestamp when the event occurred, what business transaction the event is associated with (if any), the source of the event by tier and node.

Type	Summary	Time	Business Transaction	Tier	Node
Code Deadlock	JVM deadlock det...	09/24/12 9:44:36 AM		2Tier	node2
Code Deadlock	JVM deadlock det...	09/24/12 9:39:36 AM		2Tier	node2
Code Deadlock	JVM deadlock det...	09/24/12 9:34:36 AM		2Tier	node2
Code Deadlock	JVM deadlock det...	09/24/12 9:29:36 AM		2Tier	node2
Code Deadlock	JVM deadlock det...	09/24/12 9:24:36 AM		2Tier	node2
Code Deadlock	JVM deadlock det...	09/24/12 9:19:36 AM		2Tier	node2
Slow Requests - Very Slow	/processorder/elect...	09/24/12 9:17:08 AM	/processorder/elect...	1Tier	node1
Slow Requests - Very Slow	/product/outdoorM...	09/24/12 9:17:08 AM	/product/outdoor...	1Tier	node1

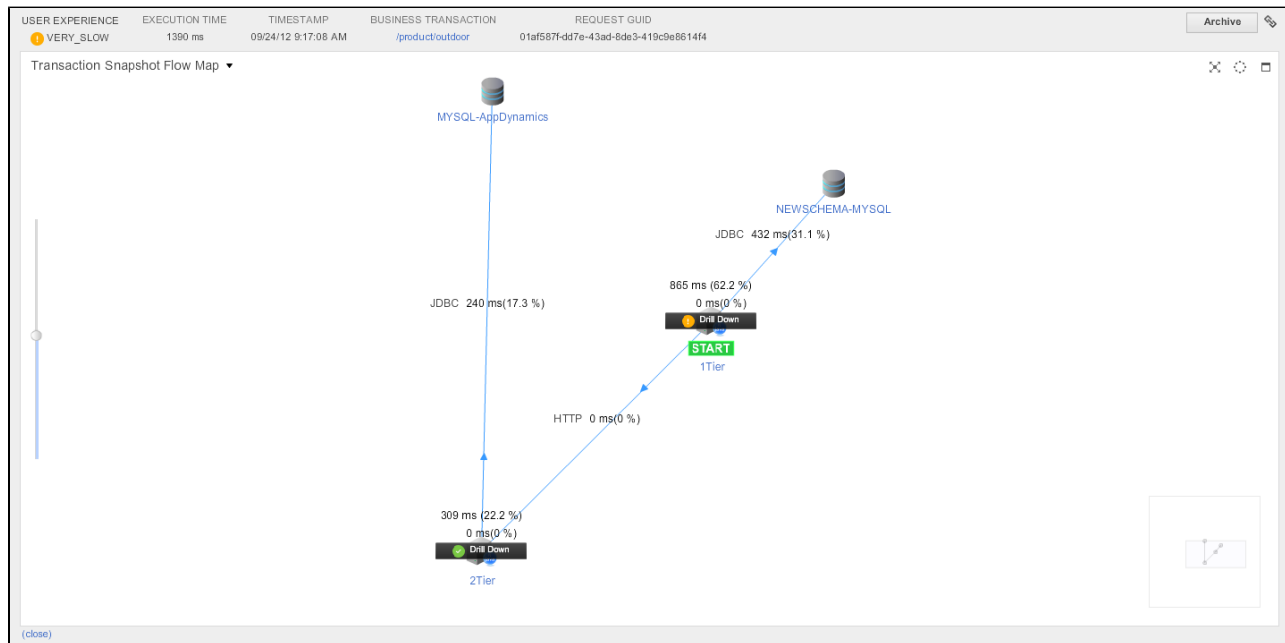
The **Events** panel shows all the events that are monitored by AppDynamics. There are several types of events:

- **Health Rule Violation Events** include business transaction health rule events such as average response time, and server health events such as Java VM Heap Utilization Thresholds. To change what generates a health rule event, see [Health Rules](#).
- **Slow Transaction Events** occur when slow, very slow or stalled transactions are detected. See [Configure Thresholds](#).
- **Error Events** occur when application exceptions are thrown or HTTP Errors are returned. See [Configure Error Detection](#).
- **Code Problem Events** occur when a code deadlock is detected or a resource pool is

utilized beyond the specified threshold. For example this event occurs when a JDBC connection pool is above 80% utilized or when a java.lang.Thread is deadlocked.

- **Application Change Events** occur when administrative changes are made to an application tier. For example, this event occurs when an application server instance is restarted or when a Java VM option is modified on an application server. See [Monitor Application Change Events](#).
- **AppDynamics Config Warnings** occur when the AppDynamics infrastructure needs attention. For example, when the Controller detects low disk space conditions on its host the policy associated with this event type occurs. See [AppDynamics Administration](#).
- **Custom** events are user-defined events. See [Events](#).

2. To get details click an event in the list. For example, click a slow request event to see the details of the request in the transaction flow map so you can start troubleshooting the slow request.



For more information on resolving issues related to slow transactions, see [Troubleshoot Slow Response Time for Java](#).

Filtering Events

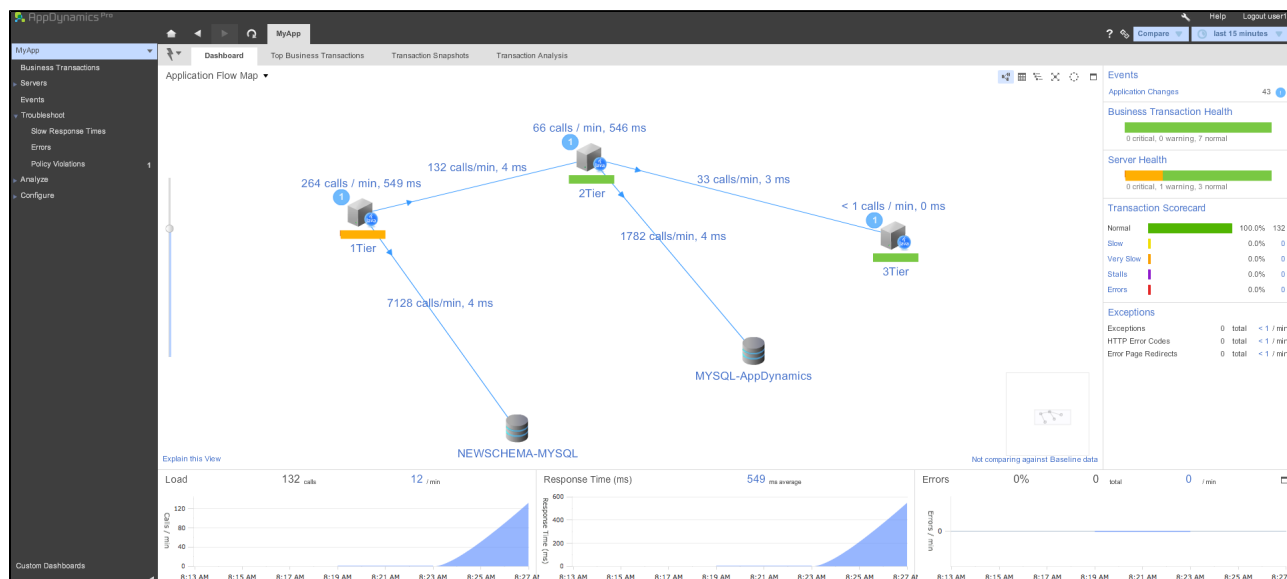
You can filter events by type in the **Events** panel. Click the type of event you want to see and click **Search**. For example, to see only **Deadlocks** and **Resource Pool Exhaustion** events select the **Code Problem Event** and click **Search**.

The screenshot shows the AppDynamics Events page. The left sidebar has a 'Troubleshoot' section with 'Events' selected. Under 'Events', 'Code Problems' is checked. A red arrow points to the 'Search' button, and another red arrow points to the 'Code Problems' filter. A green callout bubble points to the event list with the text 'The list just shows selected event types'.

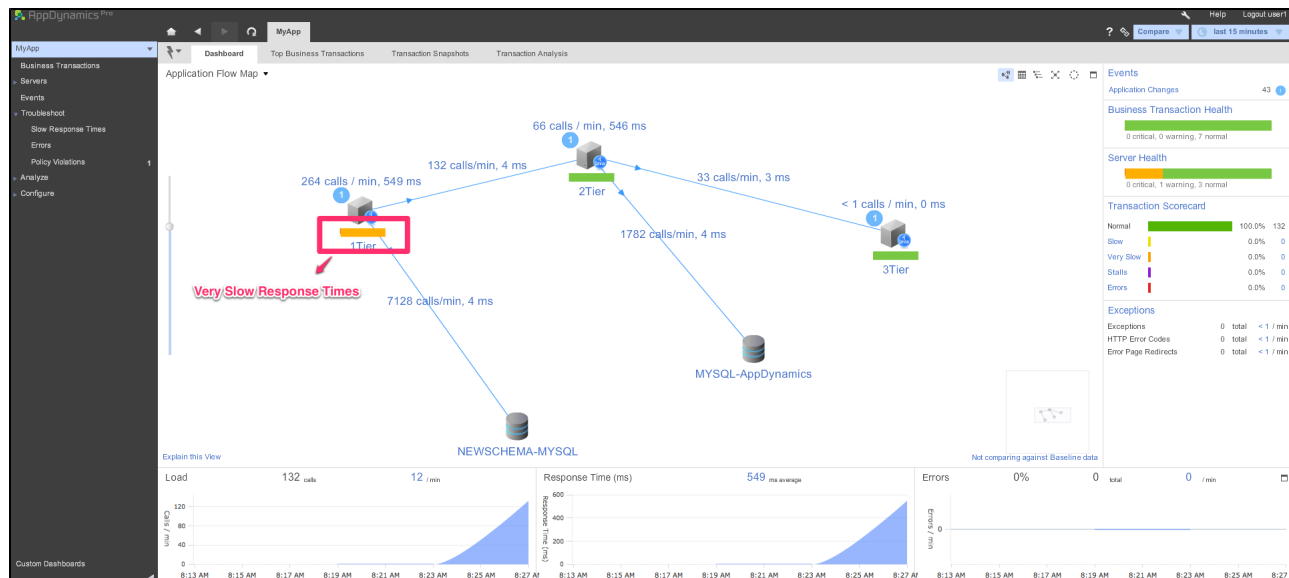
Type	Summary	Time	Business Transaction	Tier	Node
Code Deadlock	JVM deadlock det...	09/24/12 9:54:36 AM		2Tier	node2
Code Deadlock	JVM deadlock det...	09/24/12 9:49:36 AM		2Tier	node2
Code Deadlock	JVM deadlock det...	09/24/12 9:44:36 AM		2Tier	node2
Code Deadlock	JVM deadlock det...	09/24/12 9:39:36 AM		2Tier	node2
Code Deadlock	JVM deadlock det...	09/24/12 9:34:36 AM		2Tier	node2
Code Deadlock	JVM deadlock det...	09/24/12 9:29:36 AM		2Tier	node2

Tutorial for Java - Flow Maps

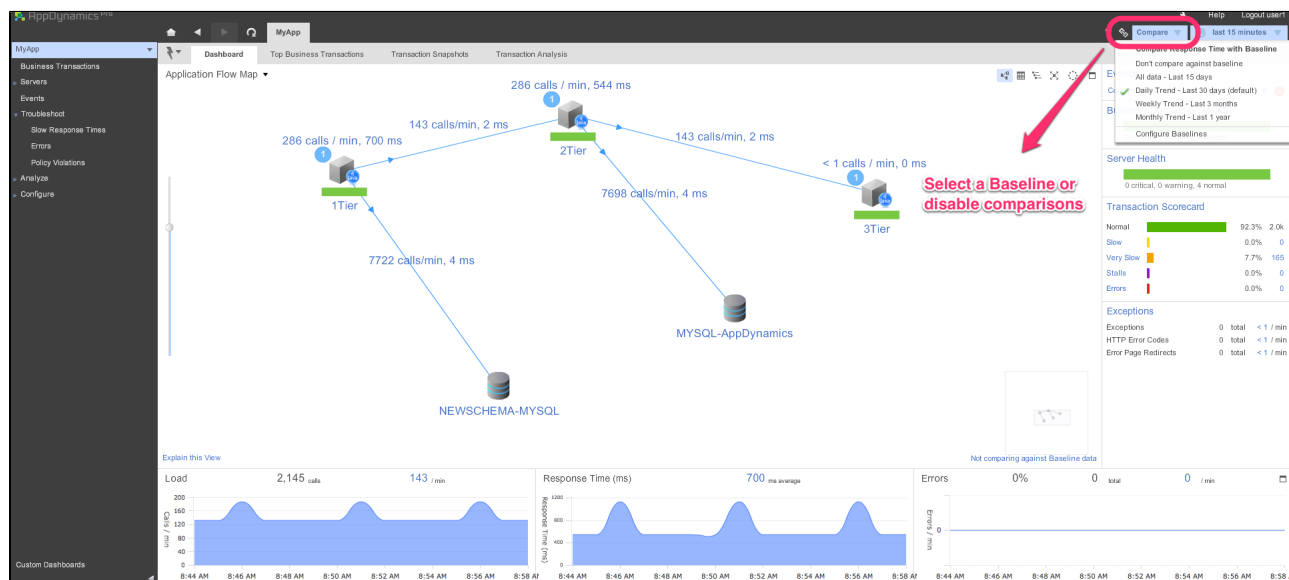
Flow maps show the health of your application, all tiers, and the communication between the tiers. It includes summary indicators such as call rates and error rates:



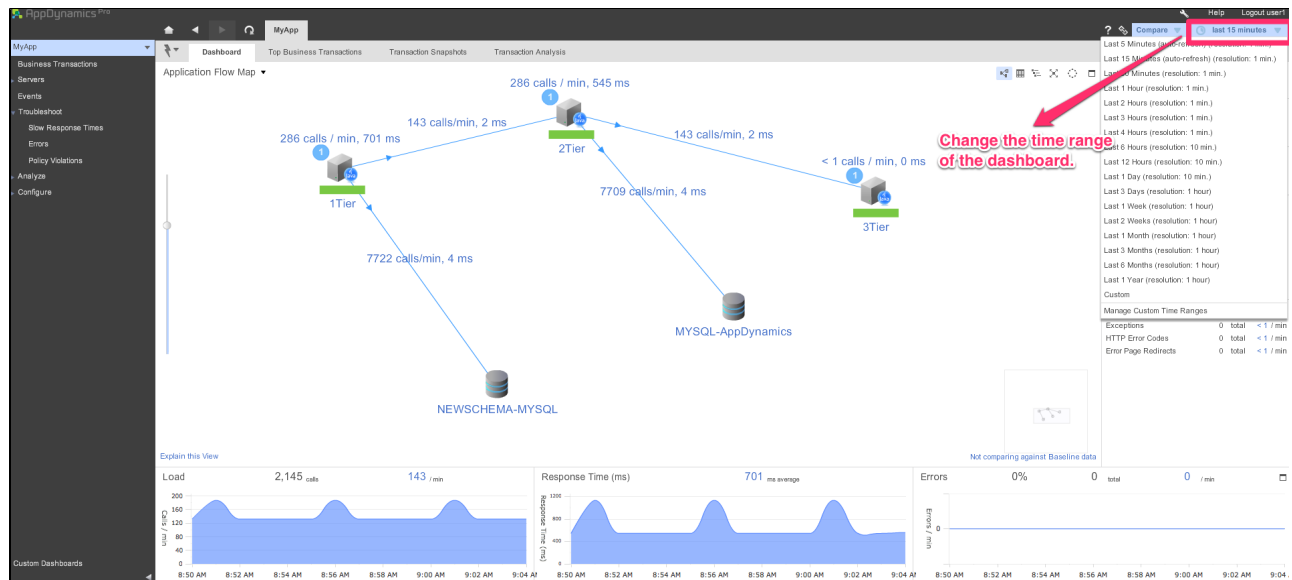
Visual indicators quickly show you problem tiers and healthy tiers. Below you can see tier 1 is experiencing very slow response times, while tier 2 and tier 3 are healthy:



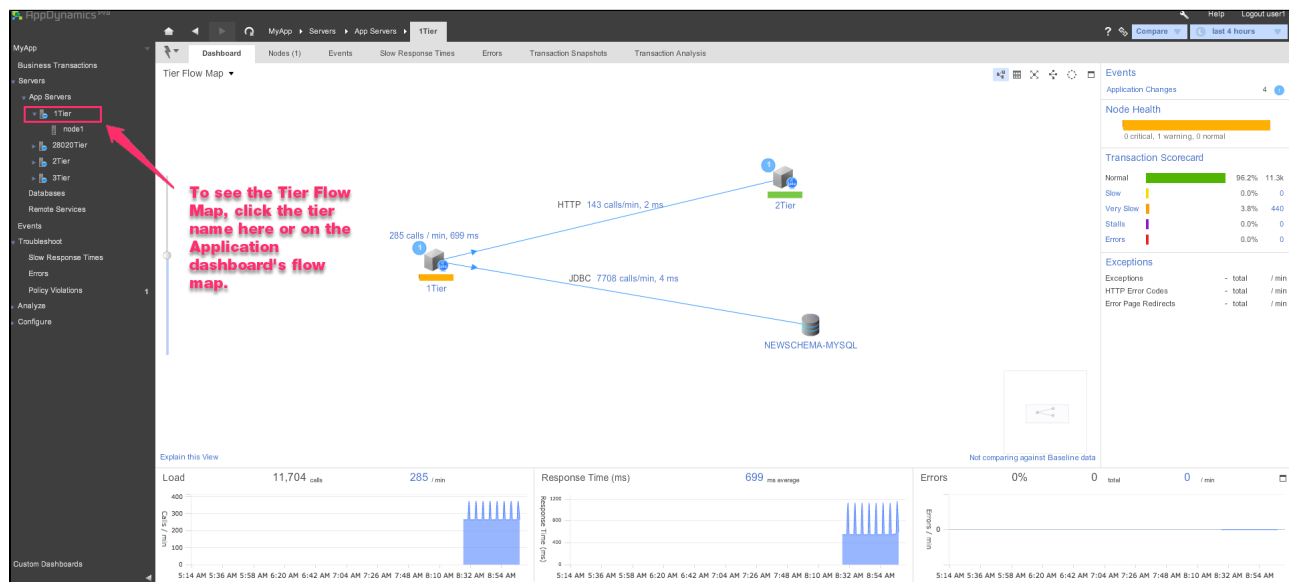
By default, the flow map computes tier health by comparing the state of the tier averaged over the last 15 minutes against the daily trend (the 30 day rolling average). You can change the time window for baseline comparison using the time window pull down menu. You can also disable baseline comparisons:



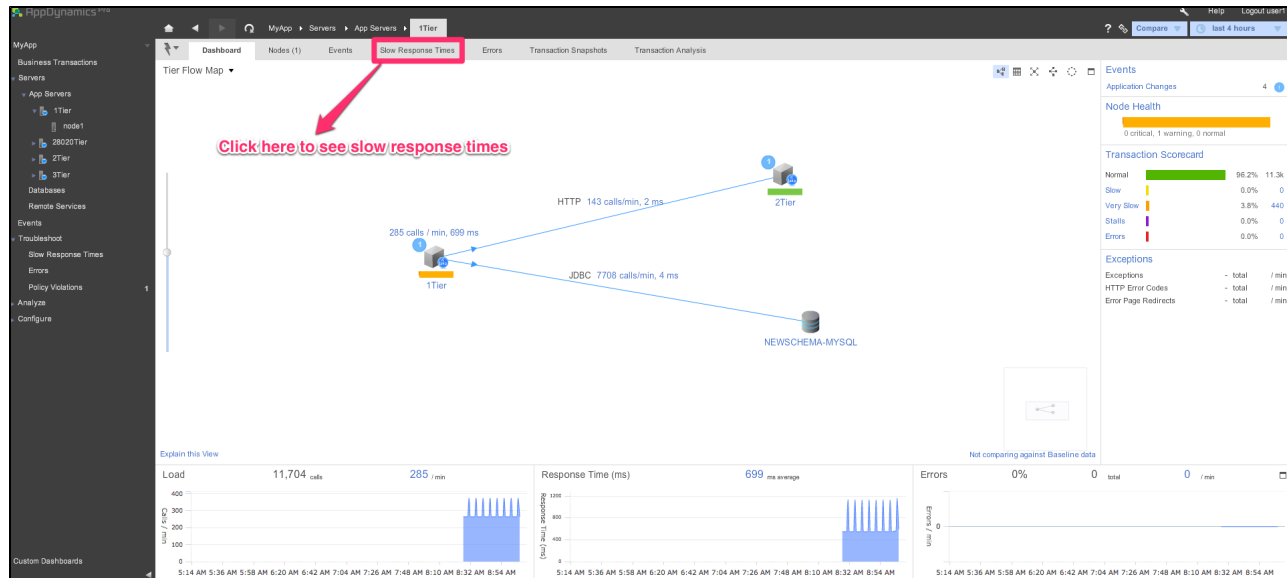
You can change the time range displayed in the flow map by changing the time window using the time window pull down menu. Changing the time range effects the entire dashboard:



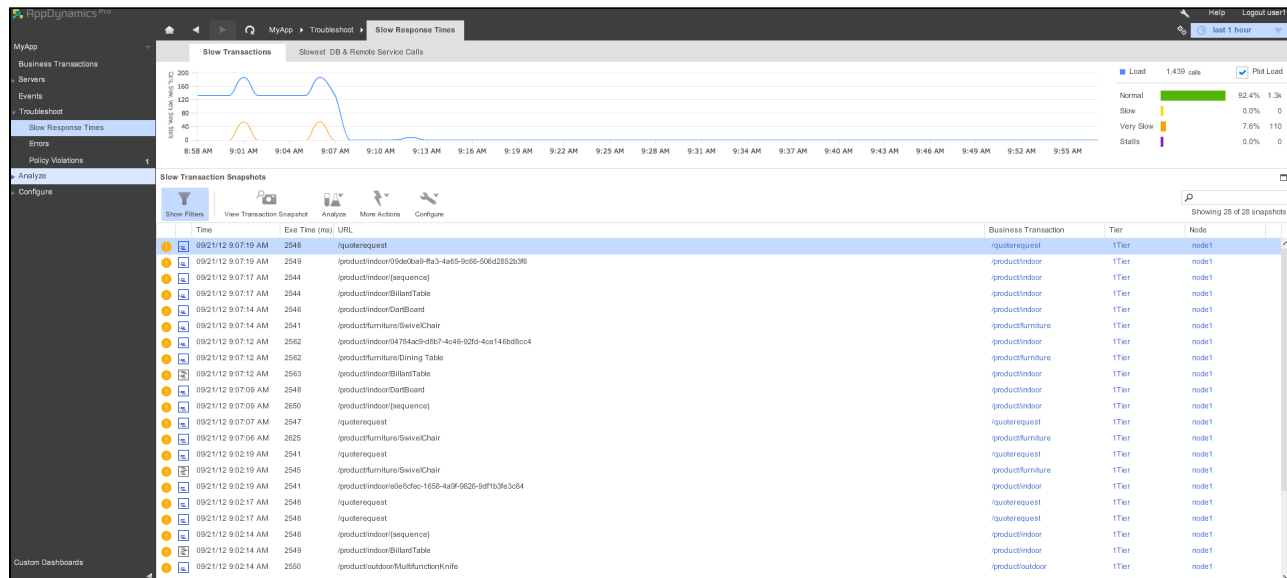
You can troubleshoot a problem system call by clicking on a the tier's name to drill down into a subset of the system involving the tier:



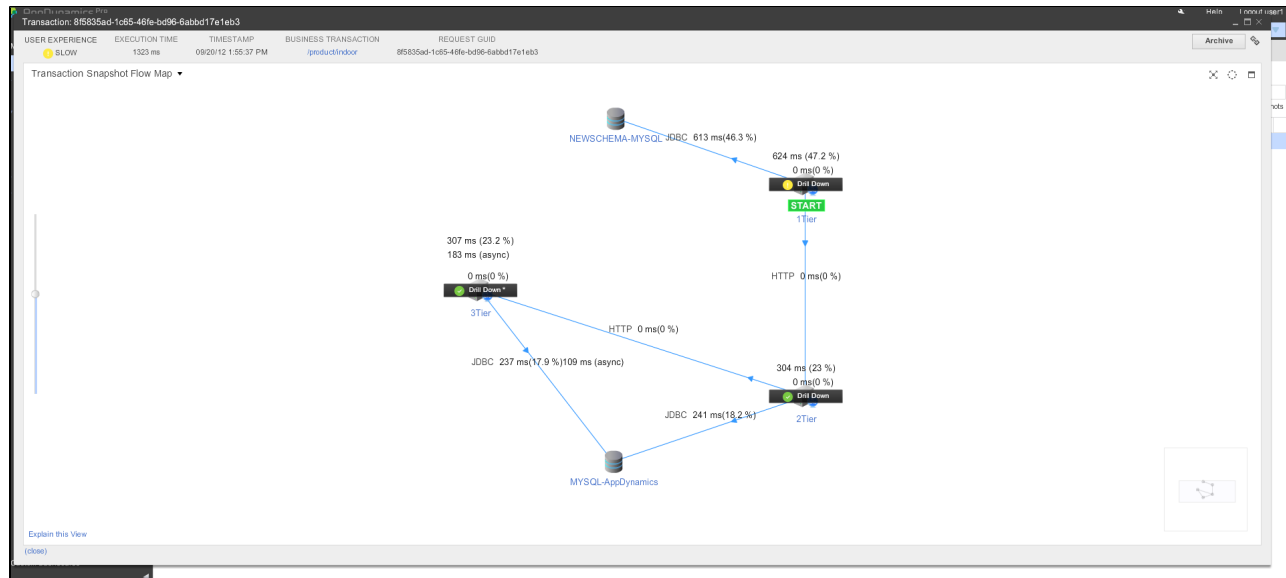
To view the slow response times in detail click on the slow response time menu:



From the slow response time pick a transaction to see a snapshot of the slow transaction:



From the transaction snapshot you can troubleshoot the slow transaction:



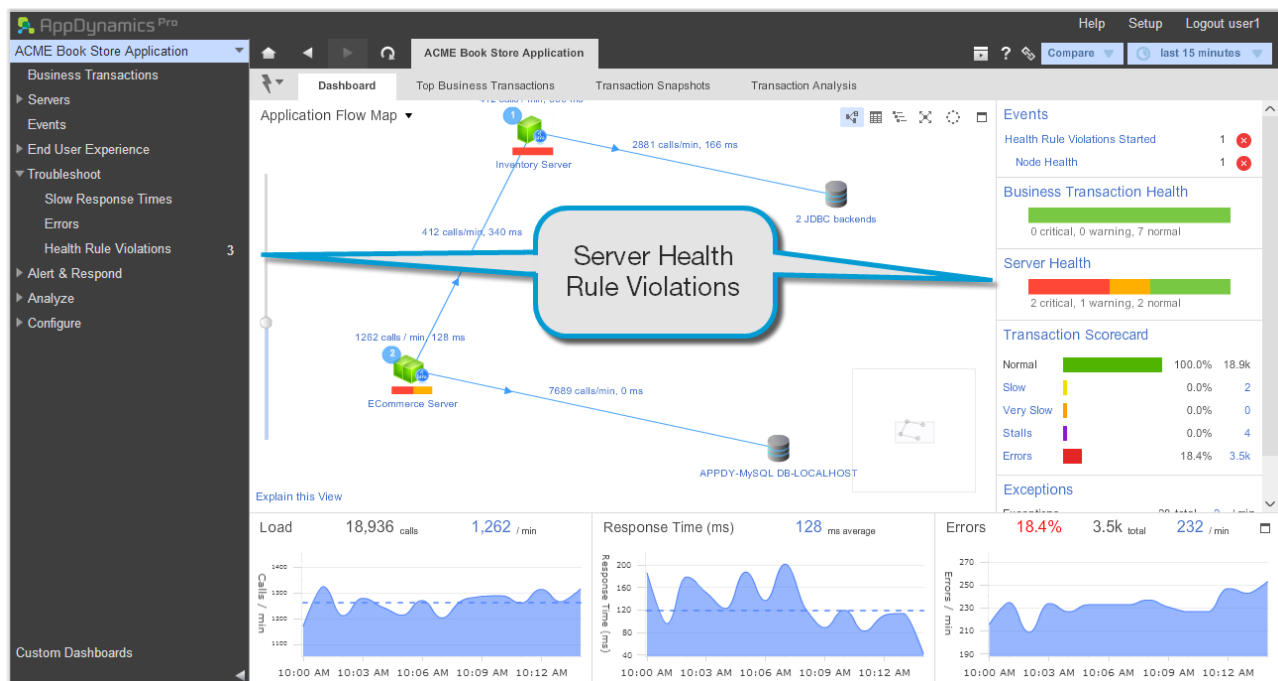
For more information on resolving issues related to slow transactions, see [Troubleshoot Slow Response Time for Java](#).

Tutorial for Java - Server Health

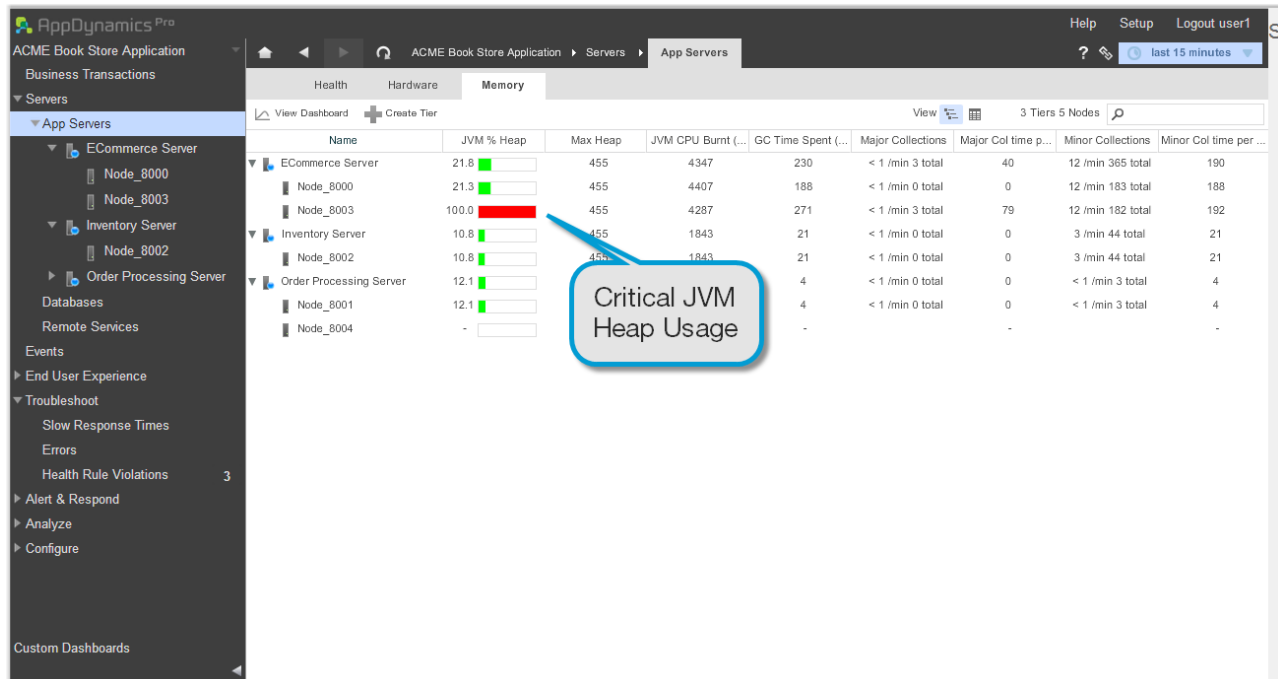
- [About Java Server Health](#)
- [Learn More](#)

About Java Server Health

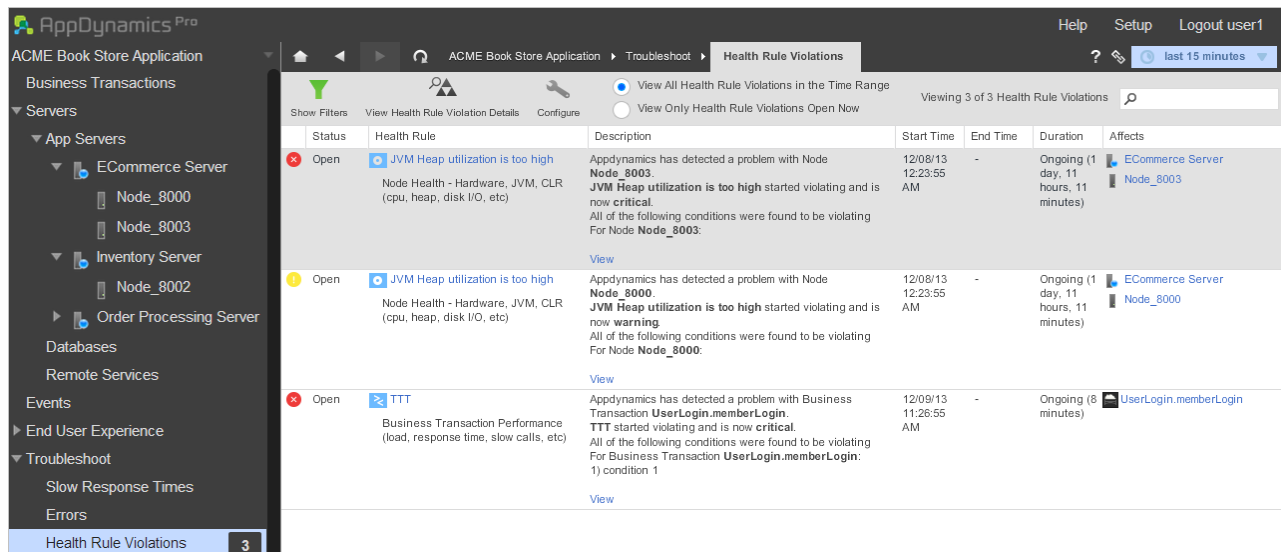
By default, AppDynamics provides predefined rules for CPU utilization, physical memory utilization, JVM heap utilization, and CLR heap utilization. For example the default health rule for CPU utilization triggers a warning when a node exceeds 75% CPU utilization and triggers a critical event when CPU utilization is 90% or above.



Node health is driven by node health rules. The following example shows that Node_8003 is experiencing a JVM heap health rule violation; all of the heap is consumed.

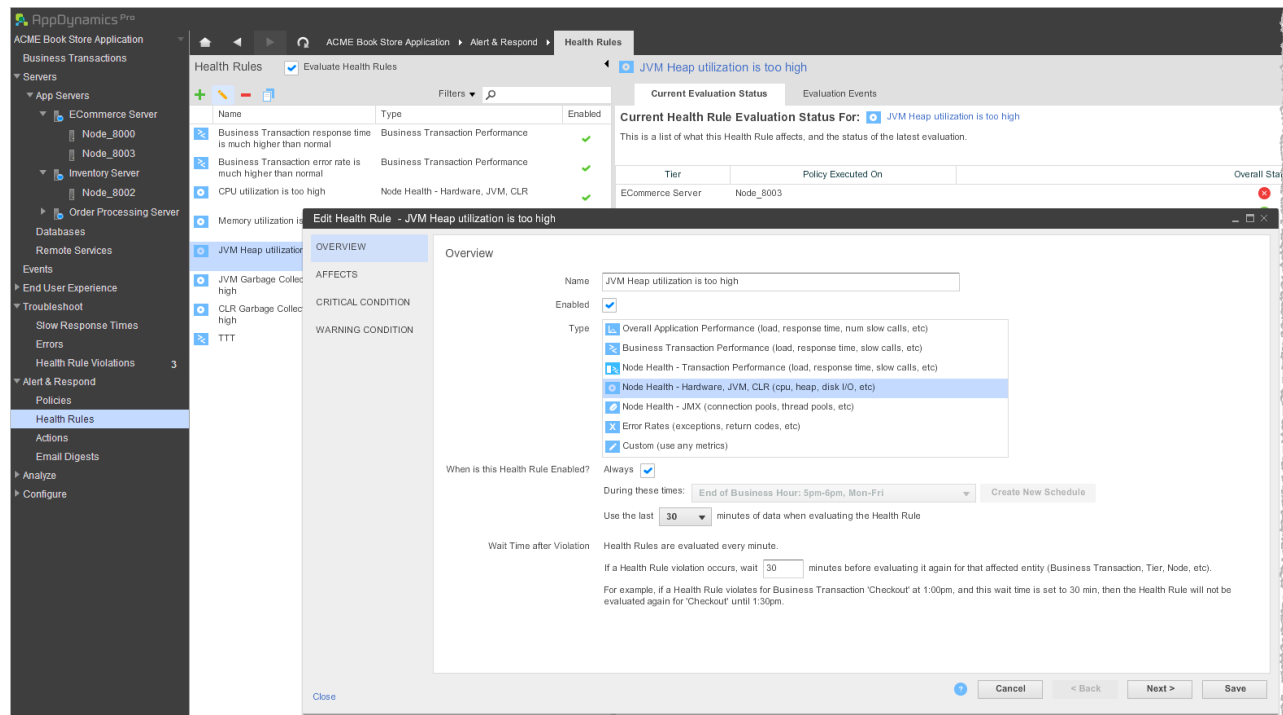


You can view the health rule violation details and the status of the violation:

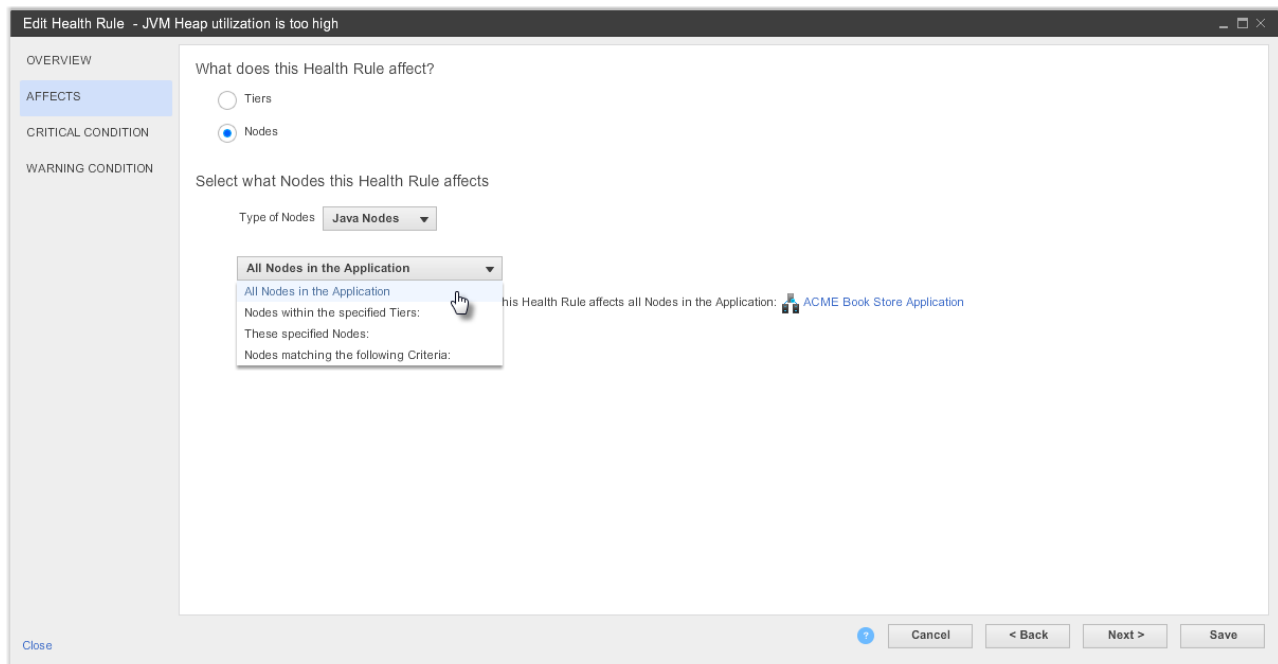


There are many types of health rules and each defines a condition or set of conditions in terms of a certain set of metrics that serve as health indicators of your application component. For example, the Business Transaction Performance health rule defines a set of conditions in terms of business transaction metrics, while the Node Health-Hardware,JVM,CLR health rule defines a set of conditions in terms of hardware metrics.

To change or add a new health rule, see [Configure Health Rules](#).

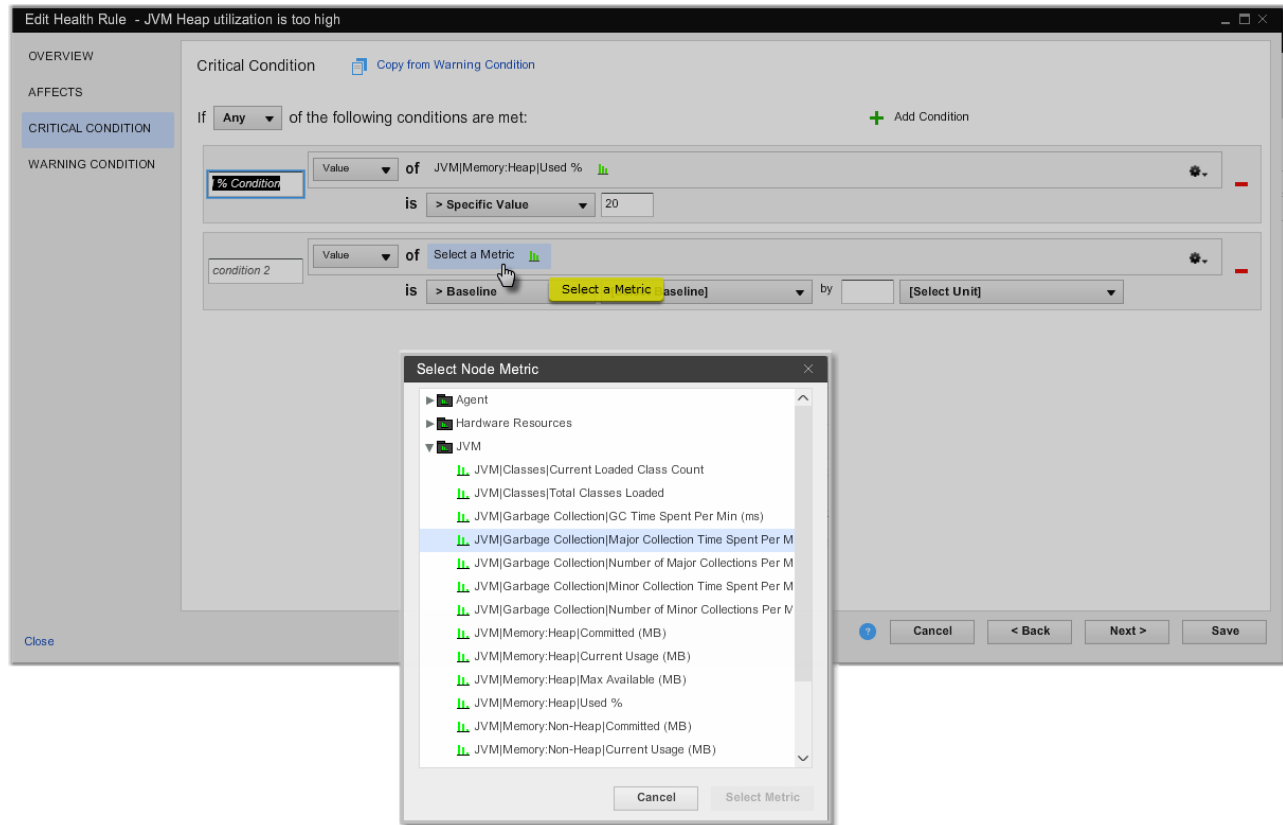


You can control the scope of a health rule to a specific application component. For example, for Node Health, you can choose between scoping to tiers or nodes. For tiers, you can apply the health rule to all tiers or to specific tiers only. For nodes, you can apply the health rule to all nodes. If you have a large cluster, you can choose specific nodes as well.

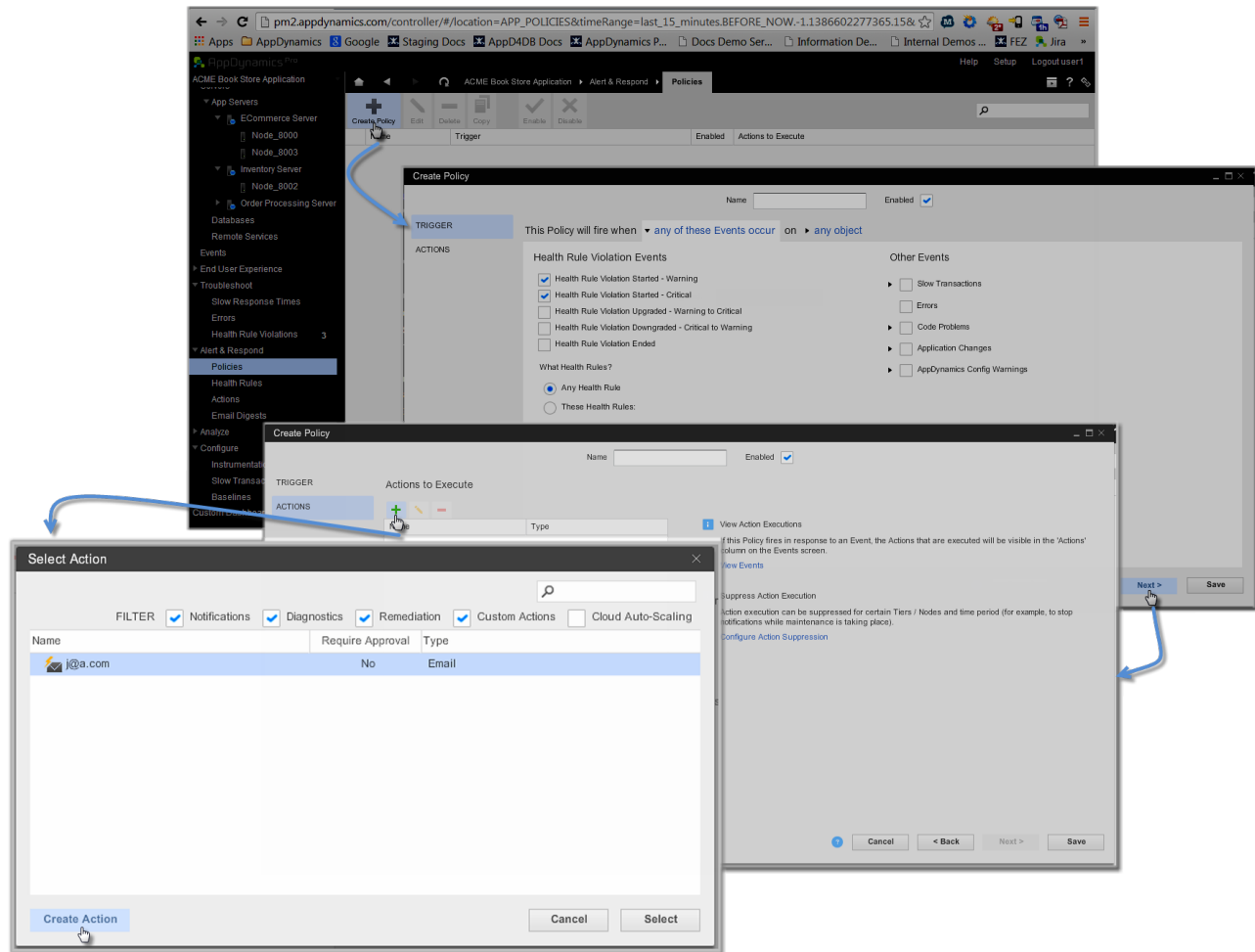


Once you scope the health rule, you can define the triggering conditions of the health rule. There are two status condition sets, warning and critical, which can be defined independently of each other. Each status condition set consists of atomic conditions that must either be ALL met or have ANY that are met in order to trigger the status condition. Atomic conditions are based on

predefined metrics that serve as health indicators of your application component. If you cannot find a Health Rule type that has a predefined metric that meets your needs, you can [add a metric using custom monitors](#) or [create a JMX metric from MBeans](#) and use a Custom Health Rule.



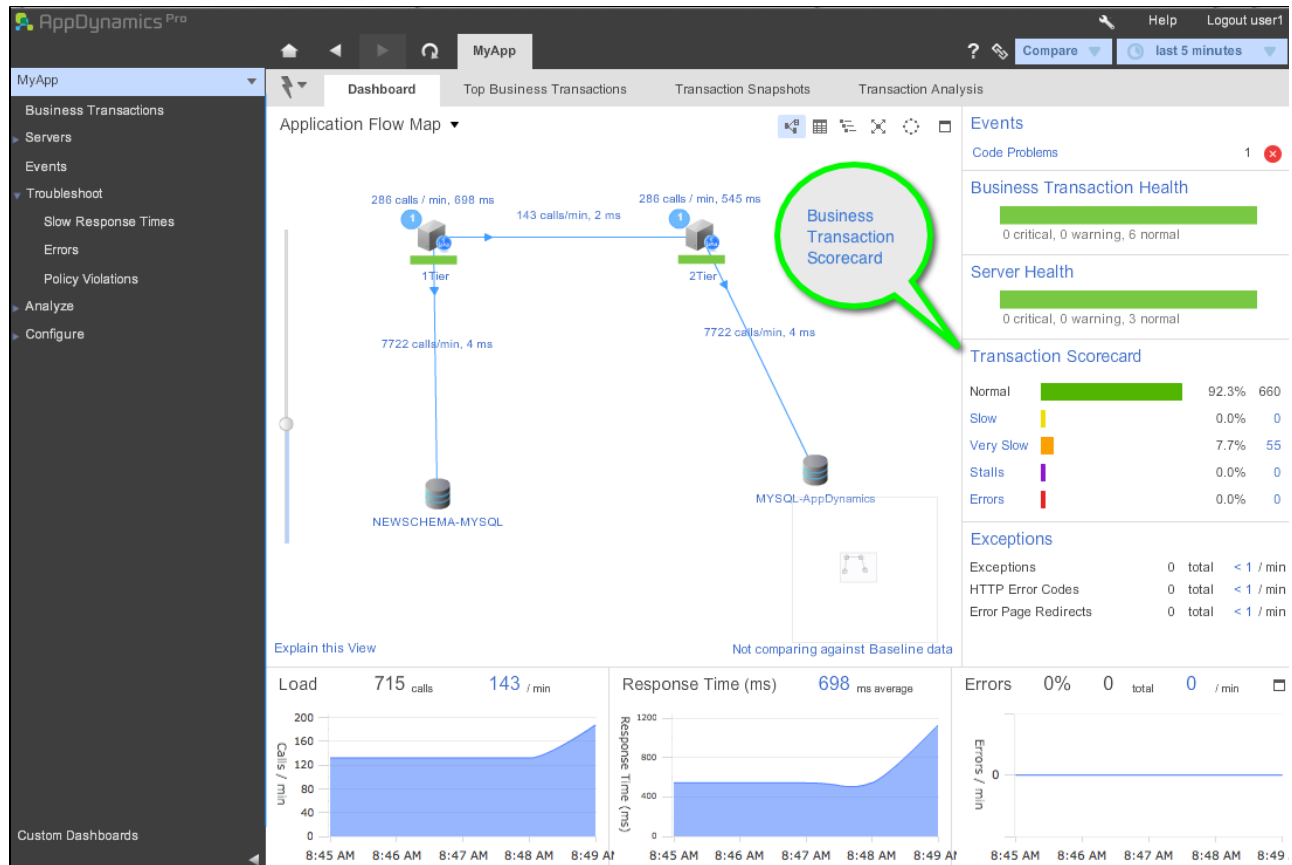
After defining Health Rules, you can use the Health Rule Violation Events in policies to trigger [actions](#) that can notify Administrators via email or SMS systems or perform some other action.



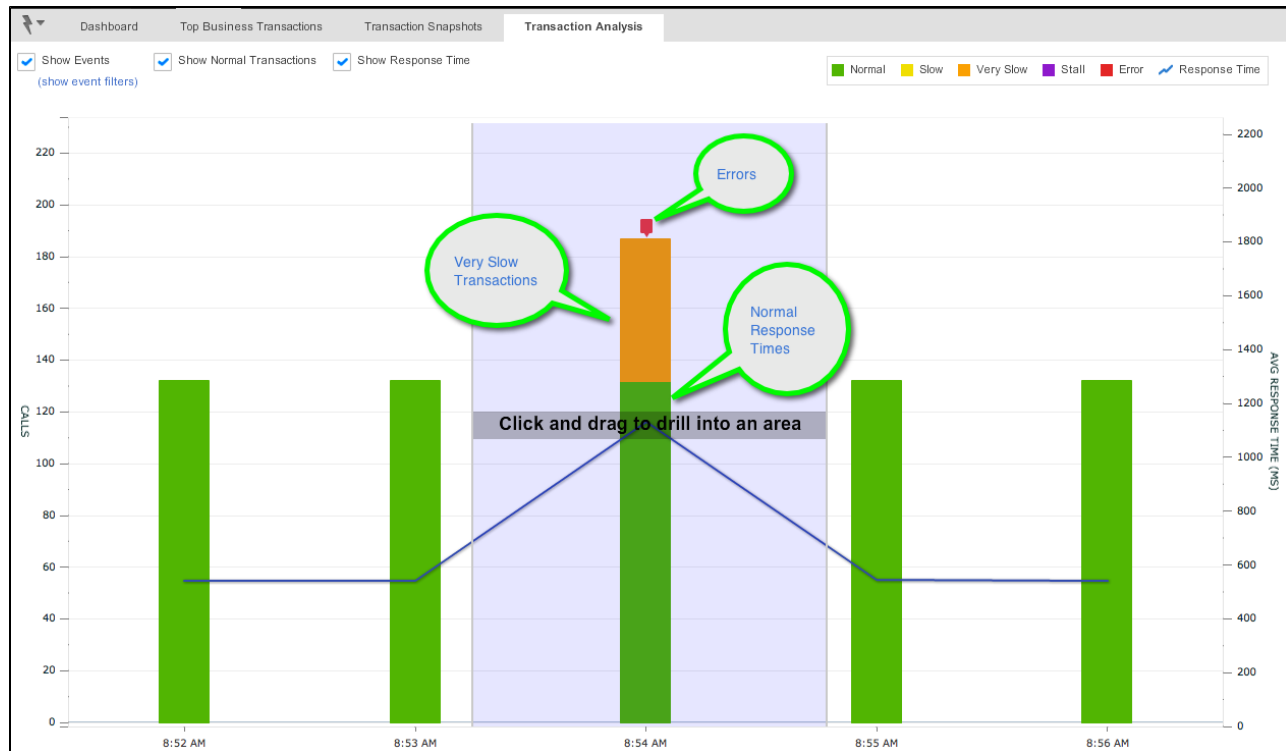
Learn More

- [Troubleshooting Server Health, AppDynamics in Action video](#) 

Tutorial for Java - Transaction Scorecards



Transactions are categorized as Normal, Slow, Very Slow, Stalled, or Errors, which are determined by thresholds and the AppDynamics error detection subsystem. Thresholds can be static or dynamic; dynamic thresholds are based on historical data. The Transaction Analysis Histogram shows the distribution over time.



Default Transaction Thresholds can be viewed here:

AppDynamics ^{PM}

MyApp > Configure > Slow Transaction Thresholds

MyApp

- Business Transactions
- Servers
- Events
 - Troubleshoot
 - Slow Response Times
 - Errors
 - Policy Violations
- Analyze
- Configure
 - Instrumentation
 - Policies
 - Alerts
 - Slow Transaction Thresholds
 - Baselines
- Custom Dashboards

Default Thresholds

Individual Transaction Thresholds

Hide Tree

Navigate Here

User Transaction Thresholds

Background Tasks Thresholds

User Transaction Thresholds

This section lets you configure Slow Transaction Thresholds, and when to trigger Diagnostic Sessions.

▼ Slow Transactions Thresholds

Every Transaction that is processed by the Application will be categorized as normal, slow, very slow, or stalled based on these thresholds.

1 Slow Transaction Threshold

☐ More than % slower than the average of the last 2 hours

☐ Greater than 5 Milliseconds

☒ Greater than 3 Standard Deviations for the last 2 hours

2 Very Slow Transaction Threshold

☐ More than % slower than the average of the last 2 hours

☐ Greater than 0 Milliseconds

☒ Greater than 4 Standard Deviations for the last 2 hours

3 Stall Threshold

☐ Disable Stall detection

☒ Stall occurs when a transaction takes more than 45 seconds.

☐ Stall occurs when a transaction's response time is deviations above the average for the last 2 hours.

☐ Apply to all Existing Business Transactions

▼ Diagnostic Session Settings

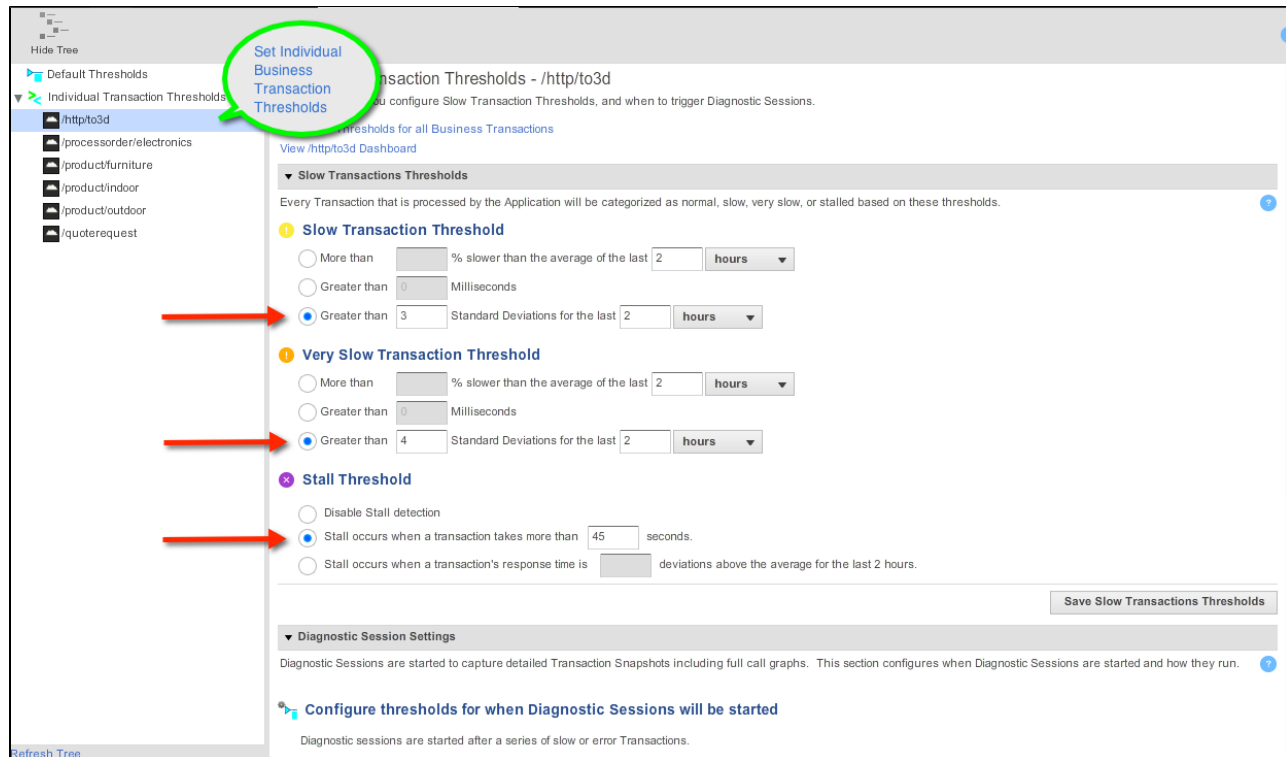
Diagnostic Sessions are started to capture detailed Transaction Snapshots including full call graphs. This section configures when Diagnostic Sessions are started and how they run.

Configure thresholds for when Diagnostic Sessions will be started

Diagnostic sessions are started after a series of slow or error Transactions.

Start Diagnostic Session if more than 10 % of the requests in a minute are slower than the Slow Transaction Threshold (configured above).

If you want to change the thresholds for all or individual transactions see the transaction threshold policy configurations (see below). See [Thresholds](#).



To troubleshoot slow transactions, see [Troubleshoot Slow Response Times for Java](#).

By Default, errors are determined when HTTP Error Codes are returned and by default AppDynamics instruments Java error and warning methods such as `logger.warn` and `logger.error`. AppDynamics captures the exception stack trace and automatically correlates it with the request. To learn how to change this, such as to reduce the number of errors reported by AppDynamics by default or to add redirect error pages, see [Configure Error Detection](#).

Troubleshooting Tutorials for Java

Tutorial for Java - Business Transaction Health Drilldown

Business Transaction Drilldown



Download MP4 version: [BTHealthDrilldown.mp4](#)

Download QuickTime version: [BTHealthDrilldown.mov](#)

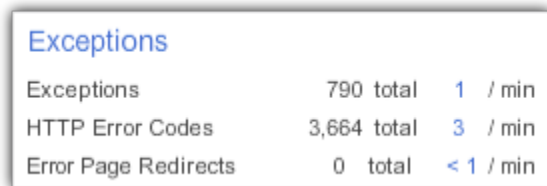
Tutorial for Java - Exceptions

- [The Exceptions](#)
- [Drill Down into the HTTP Error Code Exception](#)
- [Drill Down into the AxisFault Exception](#)
- [Drill Down into the Logger Exception](#)
- [See How Exceptions are Configured](#)
- [Learn More](#)

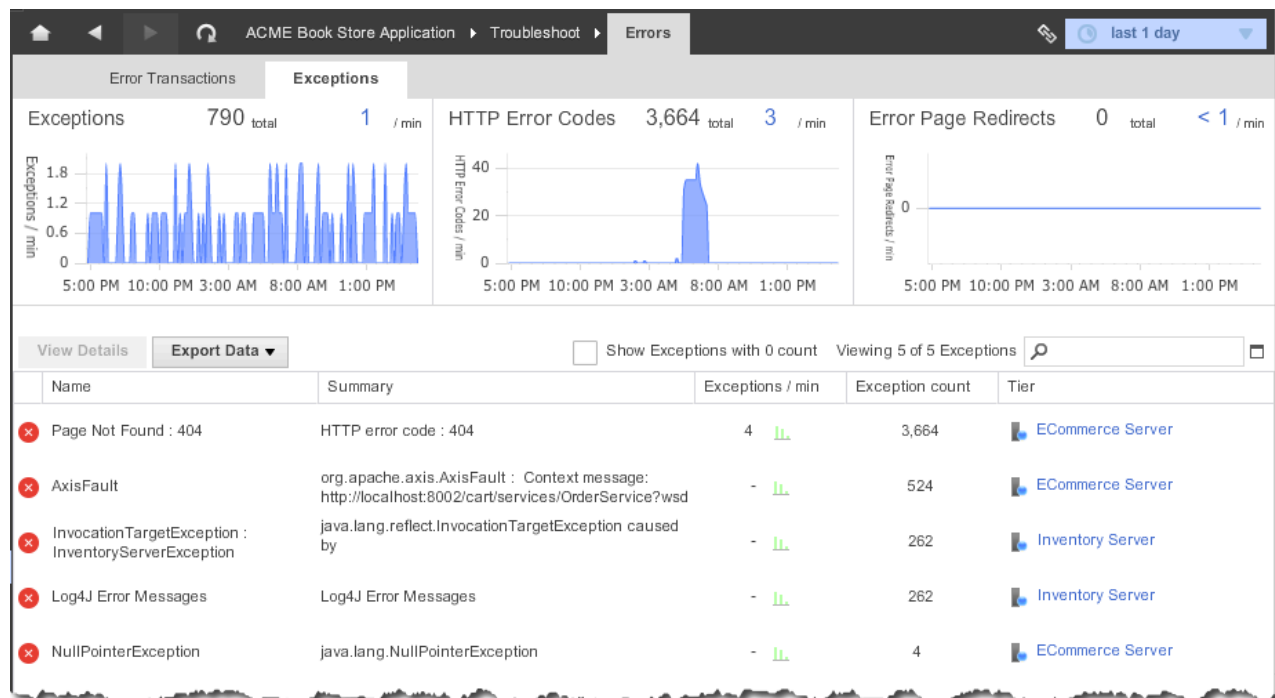
The Exceptions

An exception is a code-logged message outside the context of a business transaction. Common exceptions include code exceptions or logged errors, HTTP error codes, and error page redirects.

Exceptions display in the Exceptions pane of many dashboards.



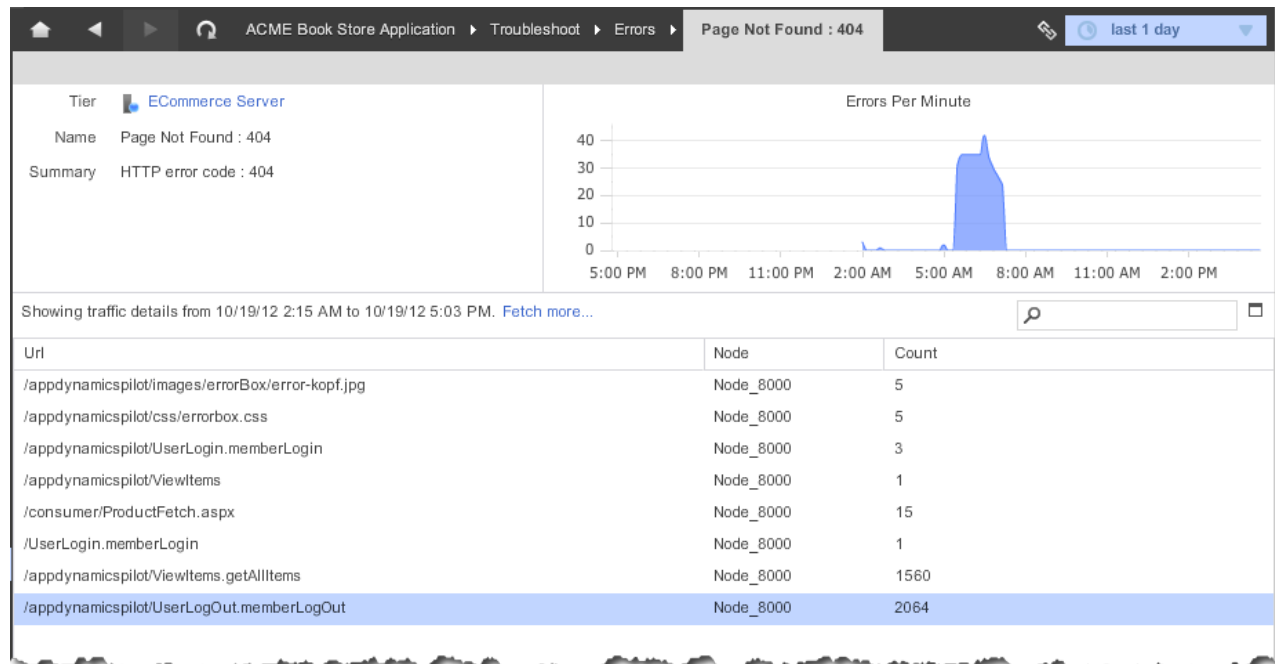
Click Exceptions to quickly see a list, ordered by frequency.



Drill Down into the HTTP Error Code Exception

Notice the spike in the HTTP Error Codes graph, and that the "Page Not Found: 404 error" is the most frequent.

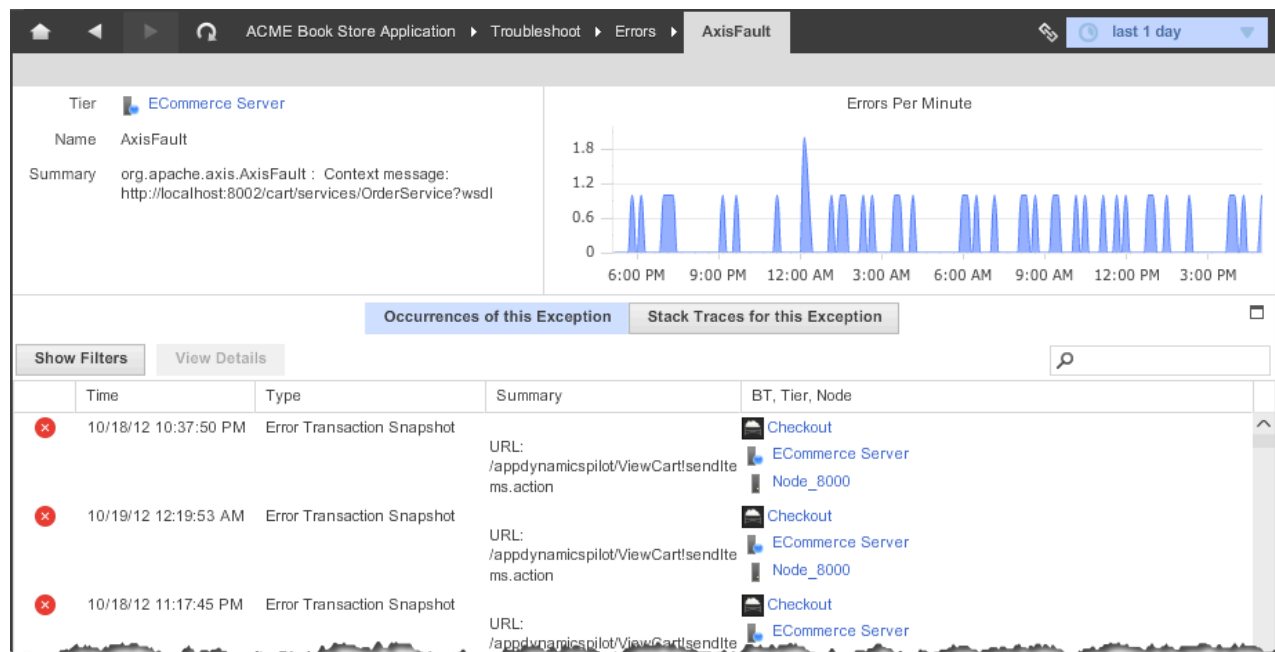
To find out more about the 404 error, click the row.



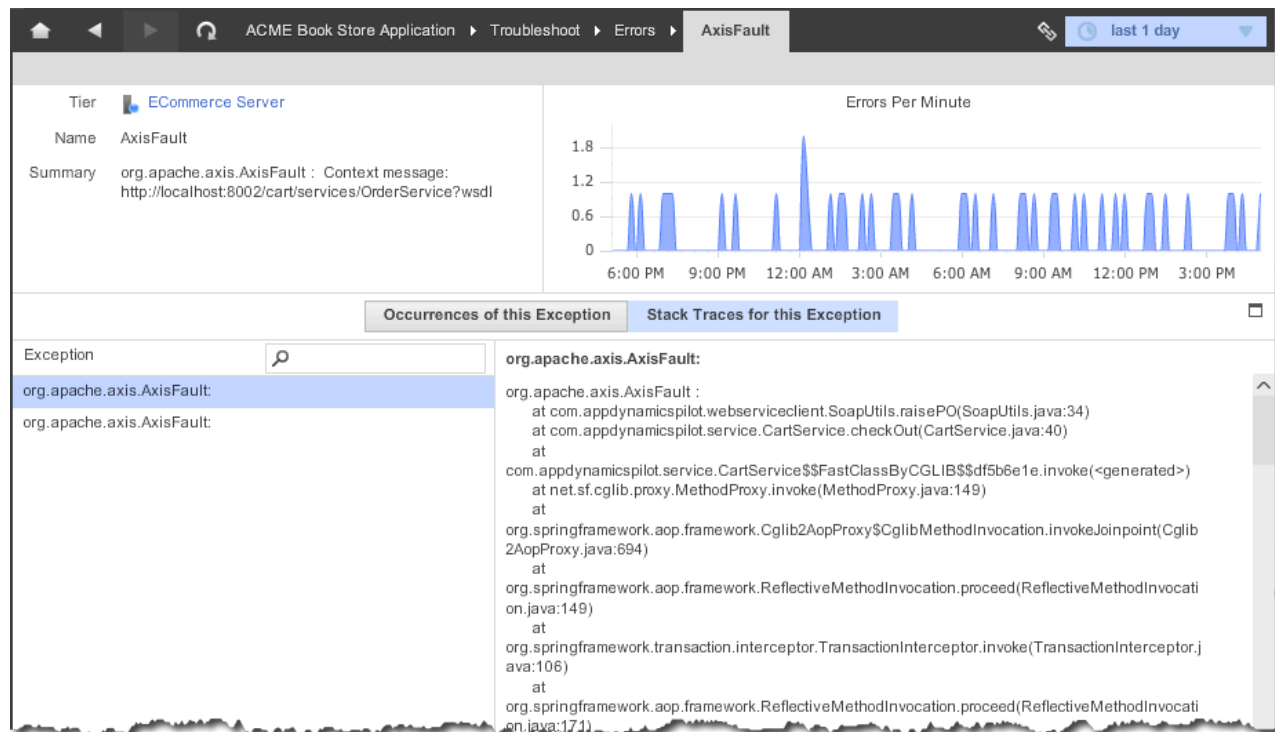
The list of URLs shows pages that have 404 errors. The memberLogOut and getAllItems URLs have the most 404 errors. You can provide this information to the web team to determine why those pages have so many 404 errors.

Drill Down into the AxisFault Exception

In the Exceptions tab, click the AxisFault row. A list of error snapshots shows the affected URL, tier, and node.



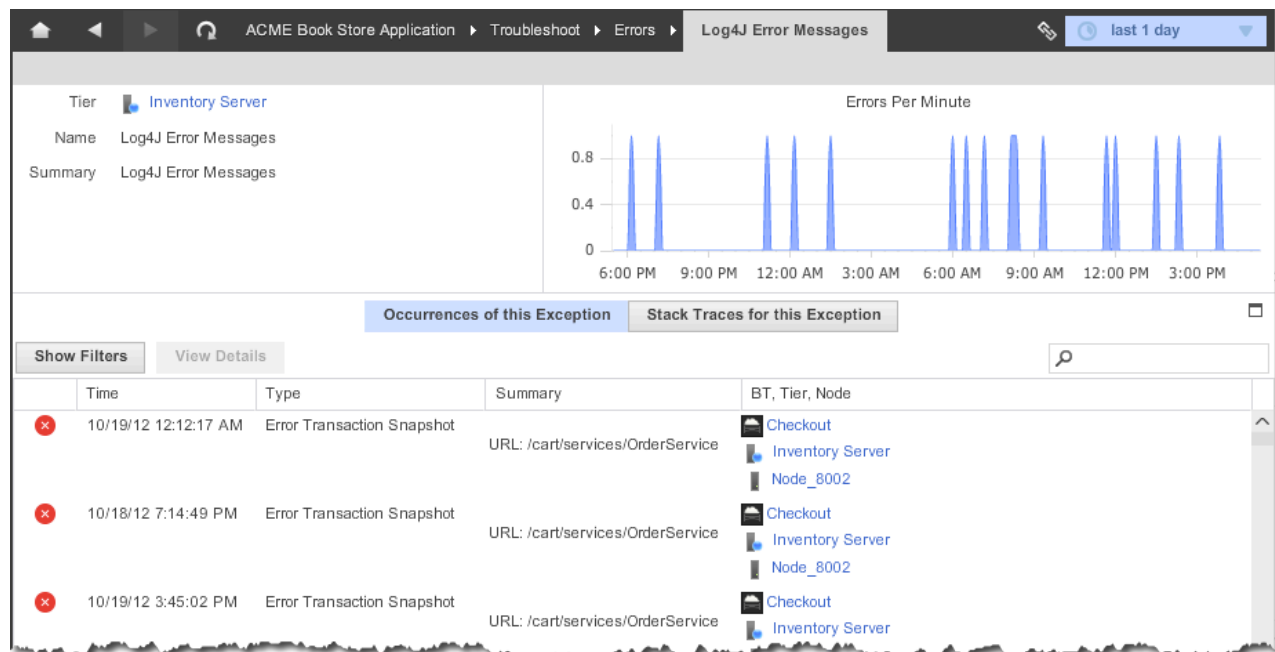
Click a row and then click the Stack Traces for This Exception tab to drill down further. Then click on one of the exceptions to see the stack trace.



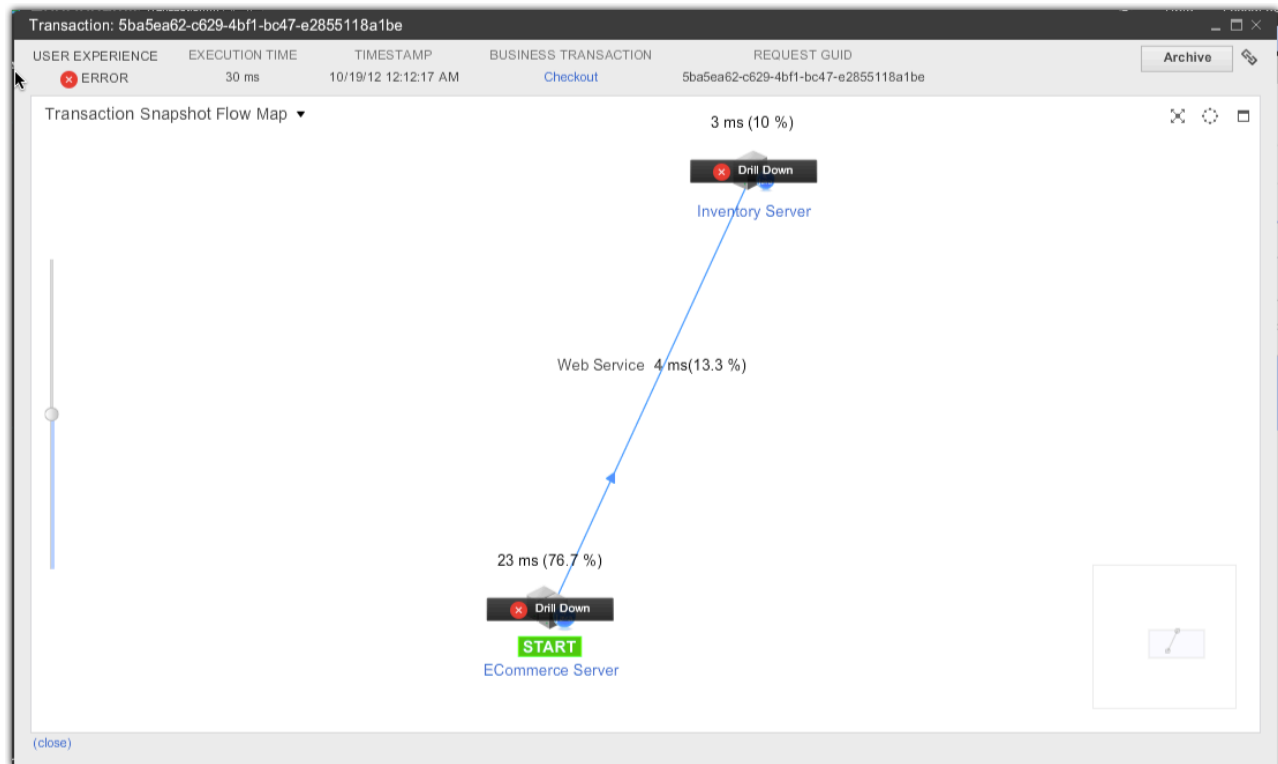
Share the stack trace with the development team to solve the problem.

Drill Down into the Logger Exception

In the Exceptions tab, click the Log4J Error Messages row. A list of error transaction snapshots shows the affected URL, business transaction, tier, and node. You can see a graph of the errors-per-minute data.

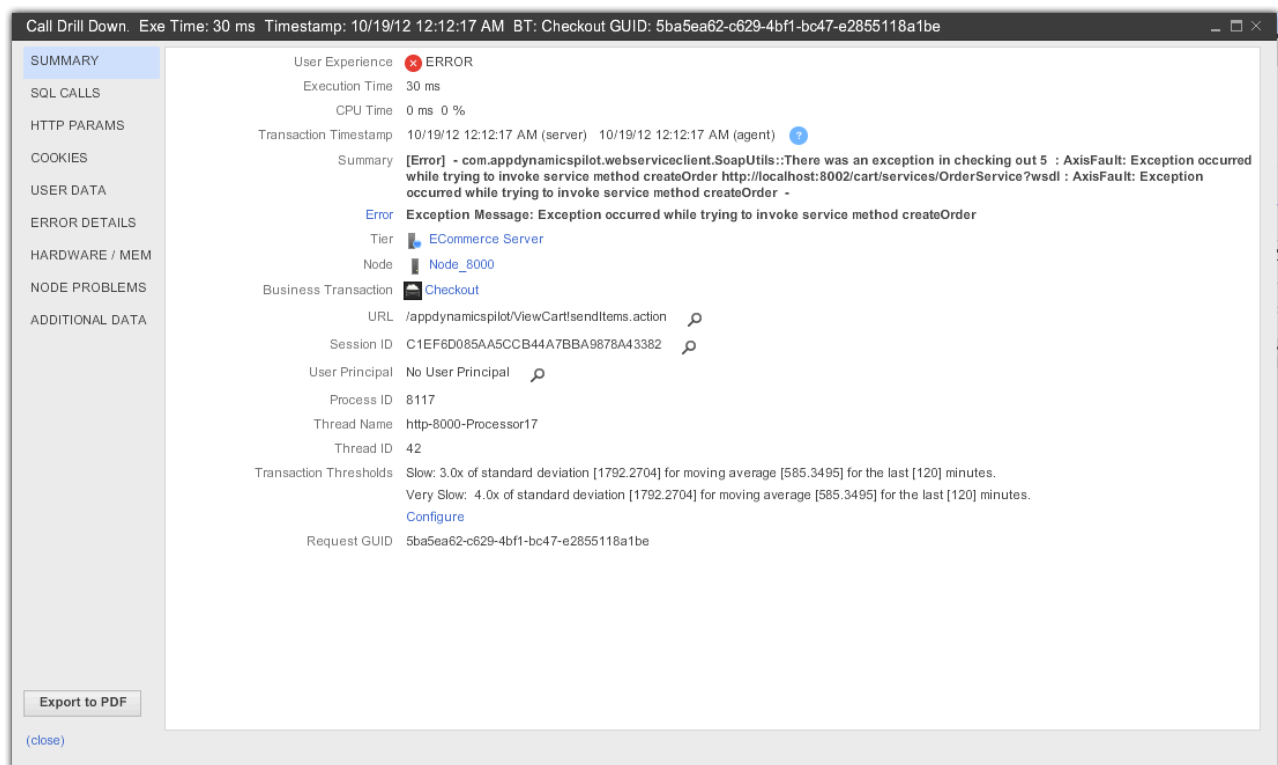


Click on a row to see the flow map for the error transaction snapshot.



The icons for both tiers have a Drill Down button. Click the Drill Down button on Ecommerce tier; it also says "Start", indicating that the transaction started on this tier.

The Call Drill Down shows the summary of the error message.



You can use the Export to PDF button at the lower left to send this information to your colleagues.

Go back to the flow map and click the Inventory tier **Drill Down** button. You see the Call Drill Down of the Inventory tier error message.

The screenshot shows the 'Call Drill Down' window in AppDynamics. The title bar indicates the transaction is 'Checkout' with a GUID of 5ba5ea62-c629-4bf1-bc47-e2855118a1be. The left sidebar contains a navigation menu with options: SUMMARY, SQL CALLS, HTTP PARAMS, COOKIES, USER DATA, ERROR DETAILS, HARDWARE / MEM, NODE PROBLEMS, and ADDITIONAL DATA. The main content area displays the following details:

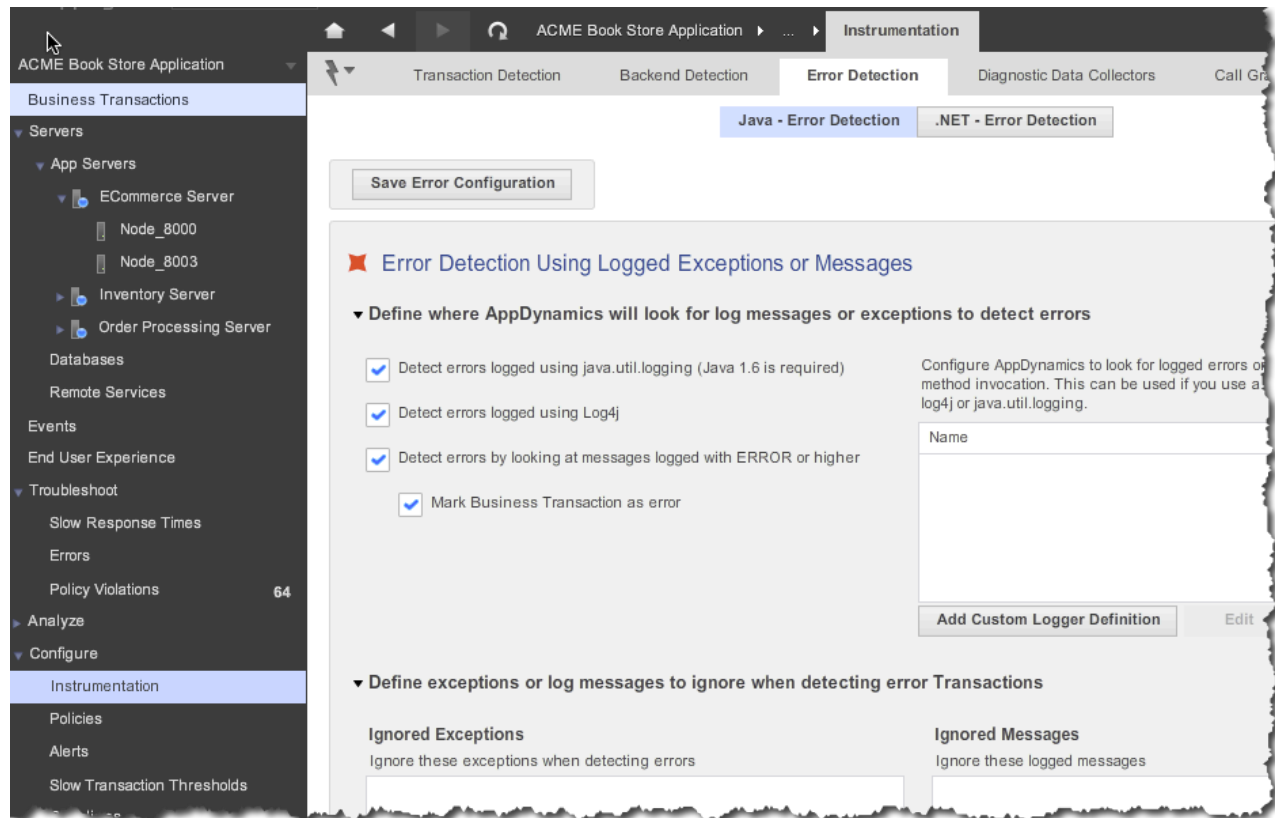
- User Experience:** ERROR (indicated by a red icon)
- Execution Time:** 3 ms
- CPU Time:** 0 ms 0 %
- Transaction Timestamp:** 10/19/12 12:12:17 AM (server) 10/19/12 12:12:17 AM (agent)
- Summary:** [Error] - org.apache.axis2.rpc.receivers.RPCMessageReceiver::Exception occurred while trying to invoke service method createOrder : InvocationTargetException com.appdynamics.inventory.OrderService : Error in creating order5 -
- Error:** Exception Message: null
- Tier:** Inventory Server
- Node:** Node_8002
- Business Transaction:** Checkout
- URL:** /cart/services/OrderService
- Session ID:** (not found)
- User Principal:** No User Principal
- Process ID:** 8153
- Thread Name:** http-8002-Processor18
- Thread ID:** 46
- Transaction Thresholds:** Slow: 3.0x of standard deviation [1392.9971] for moving average [296.60364] for the last [120] minutes. Very Slow: 4.0x of standard deviation [1392.9971] for moving average [296.60364] for the last [120] minutes. (with a 'Configure' link)
- Request GUID:** 5ba5ea62-c629-4bf1-bc47-e2855118a1be

At the bottom left, there is an 'Export to PDF' button and a '(close)' link.

Compare the two error messages.

See How Exceptions are Configured

AppDynamics provides application-level default configurations for detecting exceptions. In the left navigation pane click **Configure -> Instrumentation -> Error Detection**.

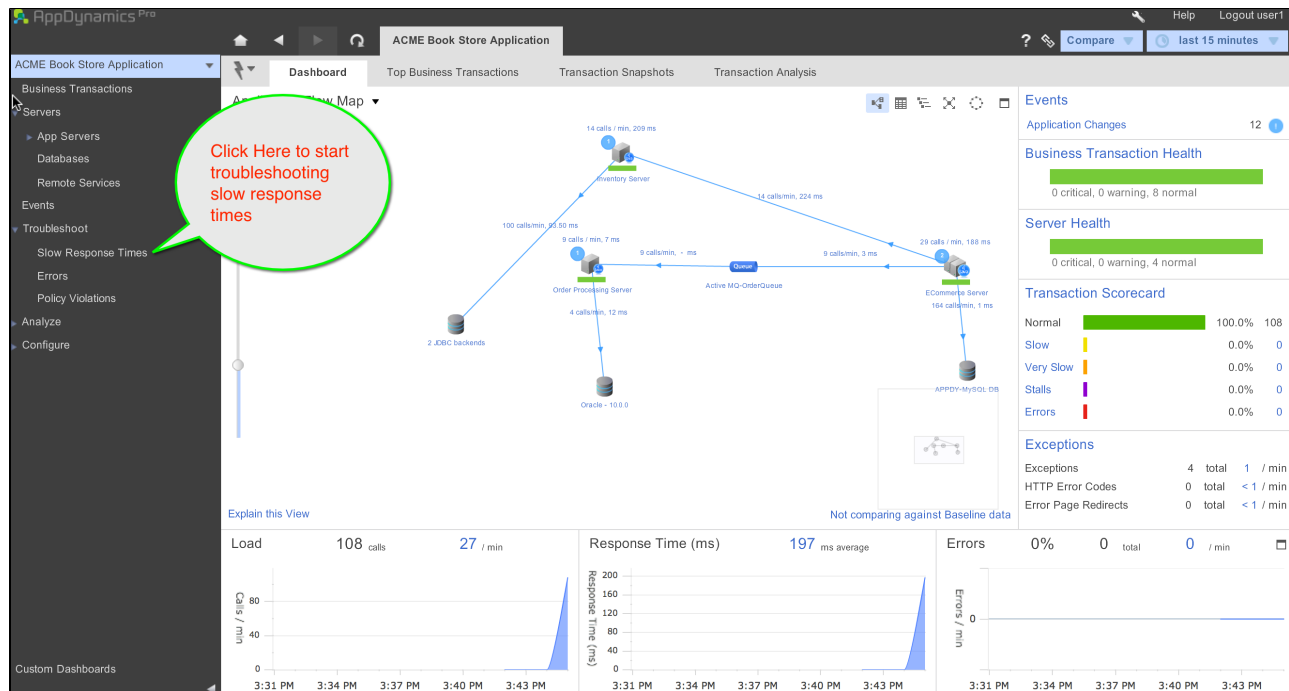


Learn More

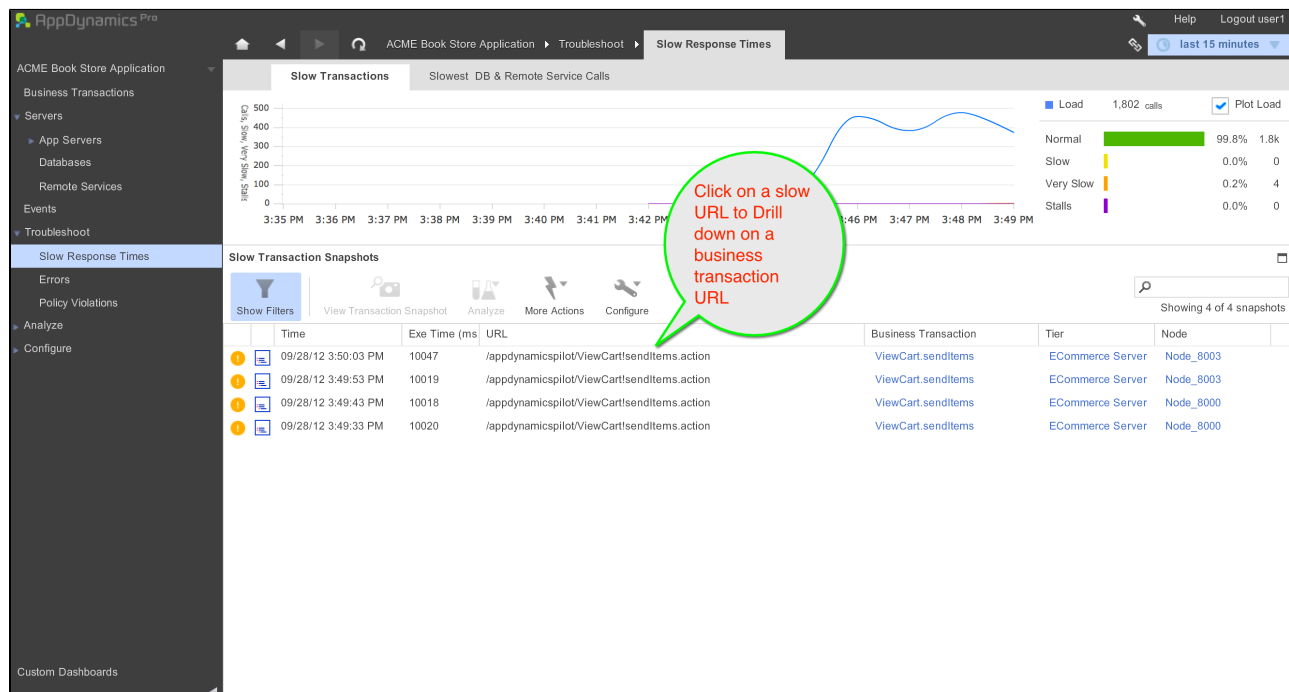
- [Troubleshoot Errors](#)
- [Configure Error Detection](#)

Tutorial for Java - Slow Transactions

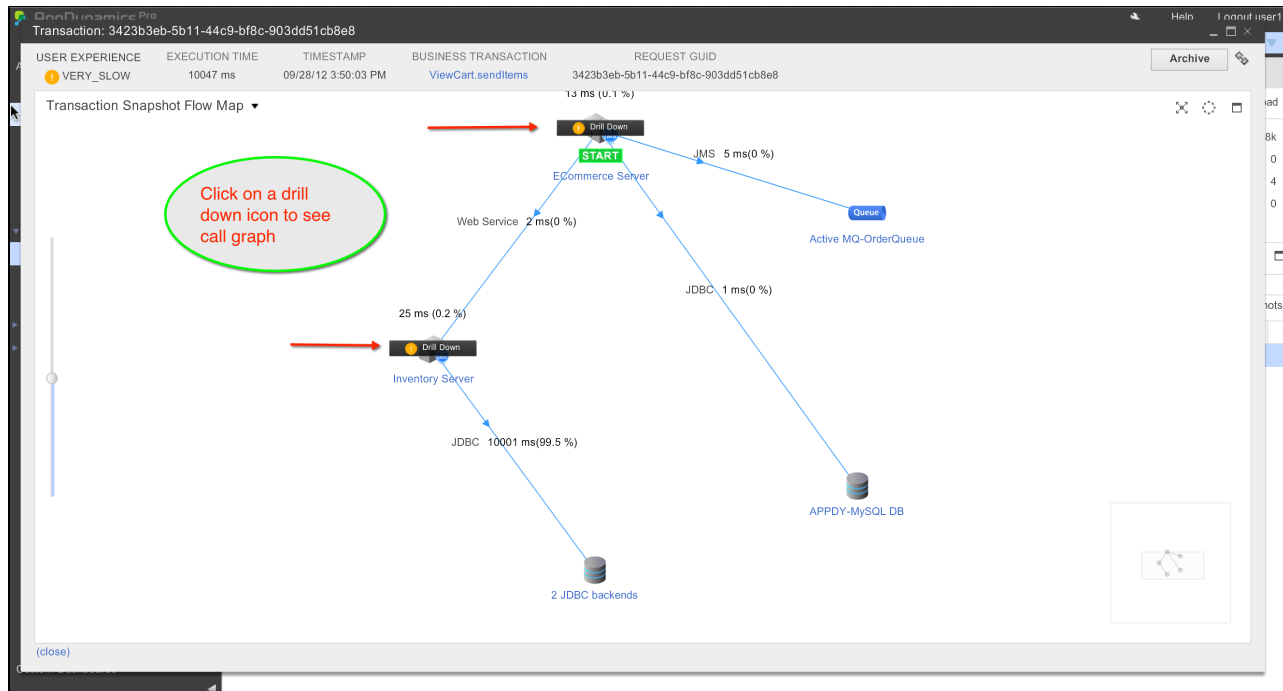
To begin troubleshooting, click **Troubleshoot -> Slow Response Times**:



To troubleshoot slow business transaction URLs, select the Slow Transactions tab and use the Transaction Snapshots pane:



Once you select the URL you will see a visualization of the transaction. You can drill into a call graph by clicking the drill-down icon.



Once you are in the call graph you can look for methods that have a significant response time. For example, the executeQuery method is responsible for 99% of response time:

Partial Call Graph

Execution Time: 10026 ms. Node Node_8002. Timestamp: 09/28/12 3:50:03 PM.

Call Drill Down. Exe Time: 10026 ms. Timestamp: 09/28/12 3:50:03 PM. BT: ViewCart.sendItems GUID: 3423b3eb-5b11-44c9-bf8c-903dd51cb8e8

Set as Root. Reset Root (?)

Show Filters

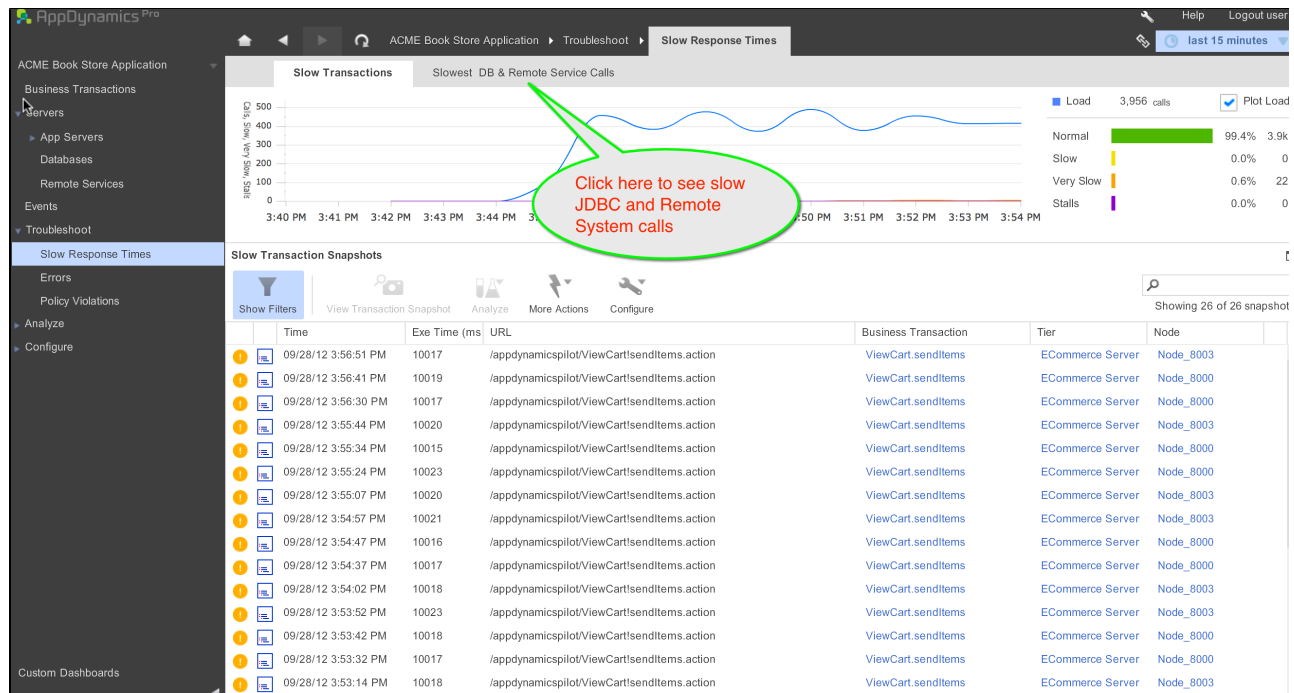
Name	Time (ms)	Exit Calls / Threads
java.lang.Thread.run:680	0 ms (self) 0 %	
HTTPServlet.service:729	0 ms (self) 0 %	
HTTPServlet.service:647	0 ms (self) 0 %	
Axis2 Webservice Servlet.doPost:116	0 ms (self) 0 %	
Web Service - org.apache.axis2.receivers.AbstractInOutSyncMessageReceiver.receive:39	0 ms (self) 0 %	
Web Service - org.apache.axis2.rpc.receivers.RPCMessageReceiver.invokeBusinessLogic:116	0 ms (self) 0 %	
Spring Bean - orderService.createOrder:16	0 ms (self) 0 %	
Proxy For Spring Bean - orderServiceTarget.createOrder	0 ms (self) 0 %	
Proxy For Spring Bean - orderServiceTarget.invoke	0 ms (self) 0 %	
Spring Bean - orderServiceTarget.createOrder:22	0 ms (self) 0 %	
Spring Bean - orderDao.createOrder:33	18 ms (self) 0.2 %	
com.appdynamics.inventory.QueryExecutor.executeSimplePS:61	0 ms (self) 0 %	
com.appdynamics.jdbc.MPreparedStatement.executeQuery:41	10005 ms (total) 99.8 %	JDBC

We find a very slow JDBC query

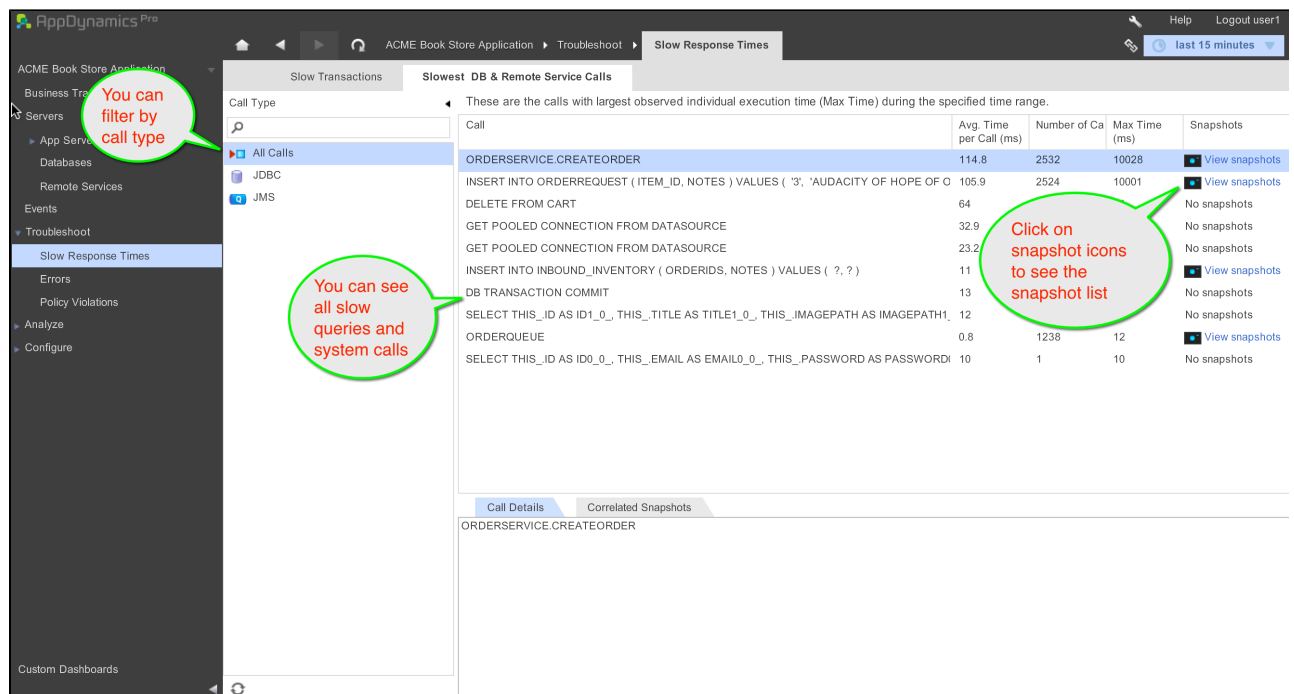
Export to PDF

Some packages have been excluded from this Call Graph

From the **Troubleshoot -> Slow Response Times** page, you can also select the **Slowest DB & Remote Service Calls** tab:



You can drill into the transaction snapshots from this tab to see the snapshot view:



For more information on resolving issues related to slow transactions, see [Troubleshoot Slow Response Times for Java](#).

Tutorial for Java - Troubleshooting using Events

- Troubleshooting with Events
 - How to Set up the Events List
 - How to Know Something is Not Quite Right
 - How to Investigate

- Investigating Errors
- Investigating Stalled Business Transactions
- Investigating Slow Business Transactions
- Investigating Application Server Exceptions
- Investigating Code Deadlocks
- Investigating Application Change Events

Troubleshooting with Events

How to Set up the Events List

1. From the left navigation pane, click an application and then click **Events**.

✔ You can also access the **Events** window by clicking **Events** on the right side of the application dashboard.

2. In the **Events** window, use the filter criteria to pick which events you want to monitor. Click **Search**.
3. Set the time range.
4. Look for issues and anomalies.

How to Know Something is Not Quite Right

You see:

- Red (critical, policy violation)
- Purple (warning, stall)
- Orange (warning, very slow)
- Yellow (warning, slow)

How to Investigate

You drill down to the root cause of the problem in different ways depending on the type of event.

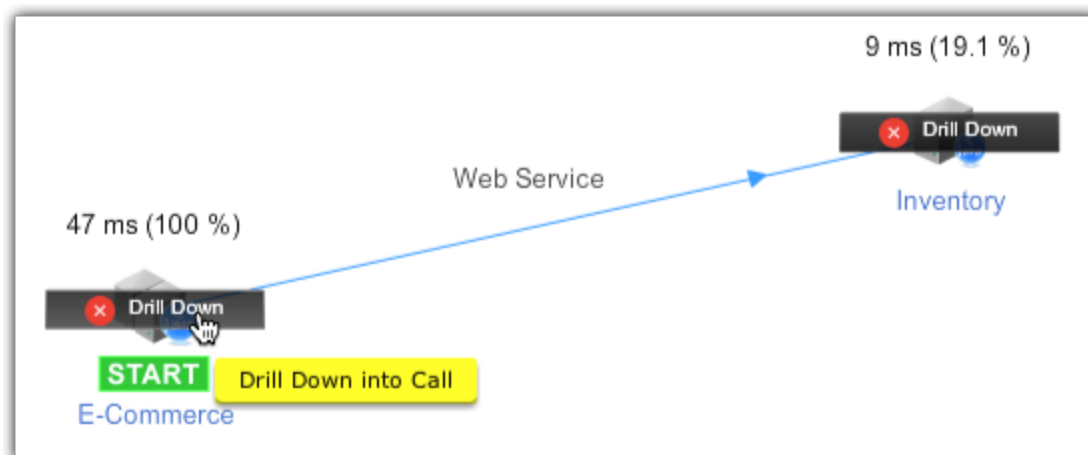
Investigating Errors

You can troubleshoot application issues by drilling down into errors.

1. In the **Events** window click an **Error**.



2. In the **Transaction Flow Map** click the Drill Down icon. If there are multiple drill down icons, select the one with the transaction that takes the most time.



3. In the **Call Drill Down** window click the **Summary** tab.

Call Drill Down. Exe Time: 47 ms Timestamp: 01/07/13 1:58:32 PM BT: Checkout GUID: 0cdcf690-e6a1-4c4d-8d2f-e53b693d0556

SUMMARY	User Experience	Execution Time	CPU Time	Transaction Timestamp	Summary	Error	Tier	Node	Business Transaction	URL	Session ID	User Principal	Process ID	Thread Name	Thread ID	Transaction Thresholds	Request GUID
	ERROR	47 ms	0 ms 0 %	01/07/13 1:58:32 PM (server) 01/07/13 1:58:32 PM (agent)	[Error] - com.appdynamicspilot.webserviceclient.SoapUtils::There was an exception while trying to invoke service method createOrder http://localhost:8002/cart/services/OrderService?wsdl	Exception Message: Exception occurred while trying to invoke service method createOrder	E-Commerce	E-Commerce-Node-8000	Checkout	/appdynamicspilot/ViewCart/sendItems.action	6F9E4F18355CB2B4958E27CBF5A87DE0	No User Principal	7366	http-8000-Processor19	46	Slow: 350 ms. Very Slow: 700 ms.	0cdcf690-e6a1-4c4d-8d2f-e53b693d0556

4. Use the **Summary** information to troubleshoot issues. This information can also be exported to PDF.

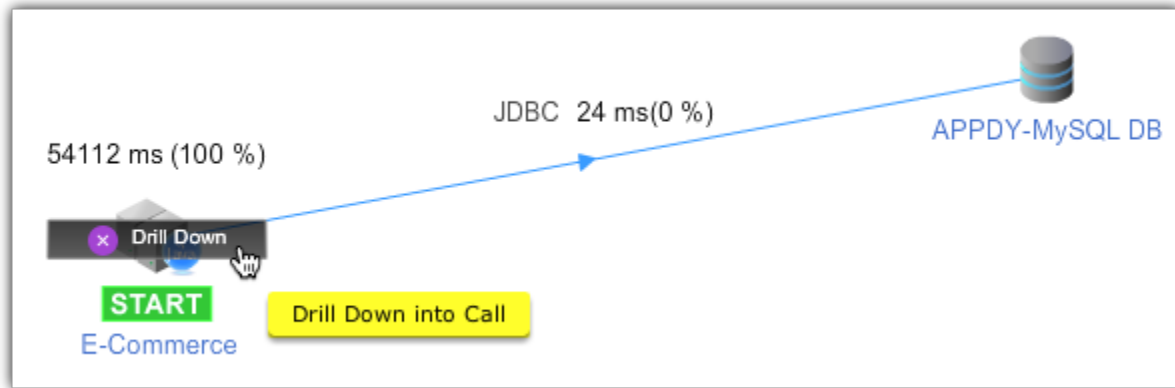
Investigating Stalled Business Transactions

You can troubleshoot business transactions by drilling down into stalled business transactions.

1. In the **Events** window click a **Slow Requests - Stalled** row.



2. In the **Transaction Flow Map** click the Drill Down icon.



3. In the **Call Drill Down** window click the **Summary** tab.

User Experience ✖ STALL

Execution Time 54136 ms

CPU Time 0 ms 0 %

Transaction Timestamp 01/07/13 1:57:41 PM (server) 01/07/13 1:57:41 PM (agent) ?

Summary **Request took higher than the stall threshold of [45000] ms -**

Tier E-Commerce

Node E-Commerce-Node-8000

Business Transaction Add to cart

Stack Dump

Thread Name:http-8000-Processor24
ID:51
Time:Mon Jan 07 21:58:26 UTC 2013
State:TIMED_WAITING
Priority:5

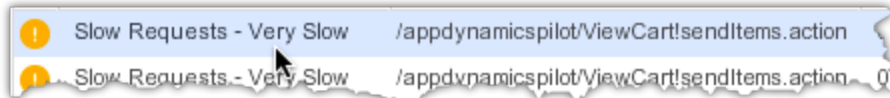
```
java.lang.Thread.sleep(Native Method)
com.appdynamics.pilot.action.CartAction.addToCart(CartAction.java:109)
sun.reflect.GeneratedMethodAccessor163.invoke(Unknown Source)
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
java.lang.reflect.Method.invoke(Method.java:597)
com.opensymphony.xwork2.DefaultActionInvocation.invokeAction(DefaultActionInvocation.java:40)
com.opensymphony.xwork2.DefaultActionInvocation.invokeActionOnly(DefaultActionInvocation.java:44)
com.opensymphony.xwork2.DefaultActionInvocation.invoke(DefaultActionInvocation.java:229)
com.opensymphony.xwork2.interceptor.DefaultWorkflowInterceptor.doIntercept(DefaultWorkflowInterceptor.java:150)
com.opensymphony.xwork2.interceptor.MethodFilterInterceptor.intercept(MethodFilterInterceptor.java:106)
```

4. Use the **Summary** information to troubleshoot business transaction issues. This information can also be exported to PDF.

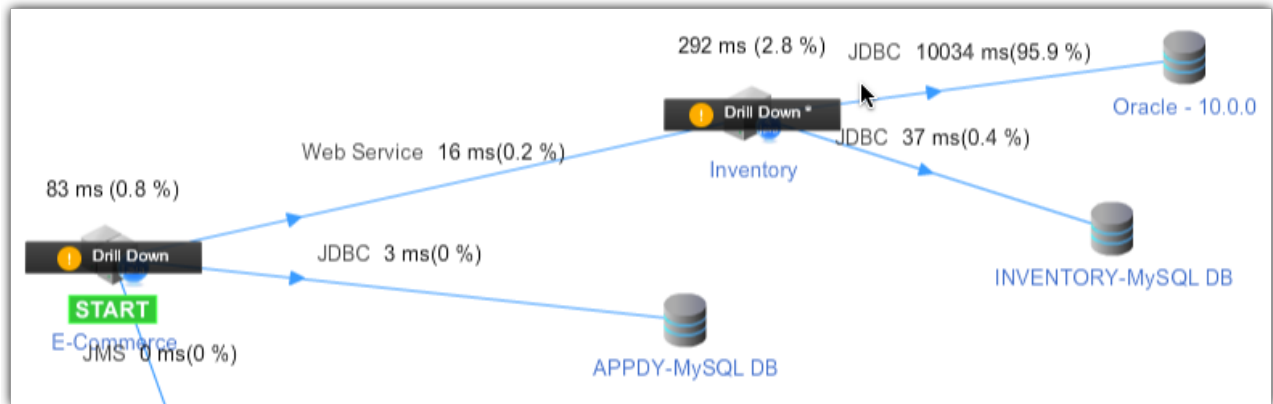
Investigating Slow Business Transactions

You can troubleshoot business transactions by drilling down into slow or very slow business transactions.

1. In the **Events** window click a **Slow Requests - Very Slow** or **Slow** row.



2. In the **Transaction Flow Map** click the **Drill Down** icon. If there are multiple drill down icons, select the one with the transaction that takes the most time.



If there is more than one call from the originating Tier, you will see the **Select a Call** window. In this case, proceed to step 3. Otherwise, skip to step 4.

3. In the **Select a Call** window click the slowest call.

Select a Call to Drill Down into

Multiple calls were made to this Tier as part of this Transaction.

Drill Down into Call

Show: All Calls

	Exe Time (ms)	Summary	Exit Calls
!	10032 ms	[Web Service] call from E-Commerce	7 JDBC calls (10003 ms. max, 1429.0 ms. avg.)
✓	299 ms	[Web Service] call from E-Commerce	7 JDBC calls (17 ms. max, 2.4 ms. avg.)
✓	32 ms	[Web Service] call from E-Commerce	7 JDBC calls (14 ms. max, 2.0 ms. avg.)

4. In the **Call Drill Down** window click the **Hot Spots** tab to see the slowest methods.

This screen displays all of the method calls in the call graph sorted by time

Name	Method Time (ms)	External Calls
com.appdynamics.jdbc.MPreparedStatement.executeQuery:45	10005 ms (self) 99.7 %	JDBC
Spring Bean - transactionManager.doCommit:578	23 ms (self) 0.2 %	JDBC

Invocation Trace

```

AxisServlet.doPost:unknown (3ms self time, 10031 ms total time)
AbstractInOutSyncMessageReceiver.receive:39 (0ms self time, 10028 ms total time)
RPCMessageReceiver.invokeBusinessLogic:116 (0ms self time, 10028 ms total time)
OrderWebservices.createOrder:16 (0ms self time, 10028 ms total time)
OrderService$$EnhancerByCGLIB$$1ee8c32e.createOrder:unknown (0ms self time, 10028 ms total time)
OrderService$$FastClassByCGLIB$$e49d675f.invoke:unknown (0ms self time, 10005 ms total time)
OrderService.createOrder:22 (0ms self time, 10005 ms total time)
OrderDaoImpl.createOrder:33 (0ms self time, 10005 ms total time)
QueryExecutor.executeSimplePS:61 (0ms self time, 10005 ms total time)
MPreparedStatement.executeQuery:45 (10005ms self time, 10005 ms total time)
    
```

5. In this example, since the slow call is a database call you know you can click the **SQL Calls** tab to see the slowest SQL statement.

Query Type	Query	Avg. Time	Count
Insert	insert into OrderRequest (item_id, notes) values (?, ?)	10003	1
COMMIT	DB Transaction Commit	22	1

6. Use this information to diagnose transaction issues. This information can also be exported to PDF.

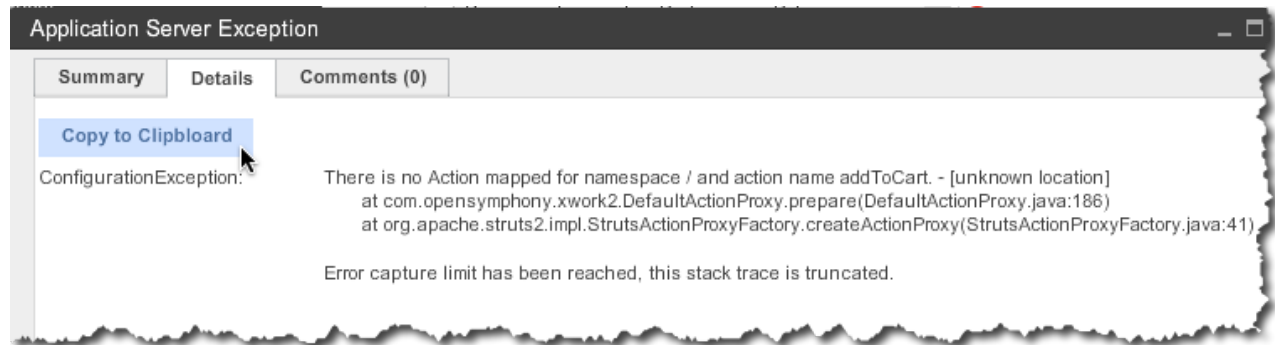
Investigating Application Server Exceptions

You can troubleshoot application server issues by drilling down into application server exceptions.

1. In the **Events** window click an **Application Server Exception**.

Application Server Exception	org.apache.struts2.dispatcher.Dispatch.
Slow Requests - Stalled	/appdynamicspilot/ViewCart!addToCart.a

2. In the **Application Server Exception** window, click the **Details** tab.



3. Use this information to troubleshoot application server issues. Use the **Copy to Clipboard** button to save the exception details.

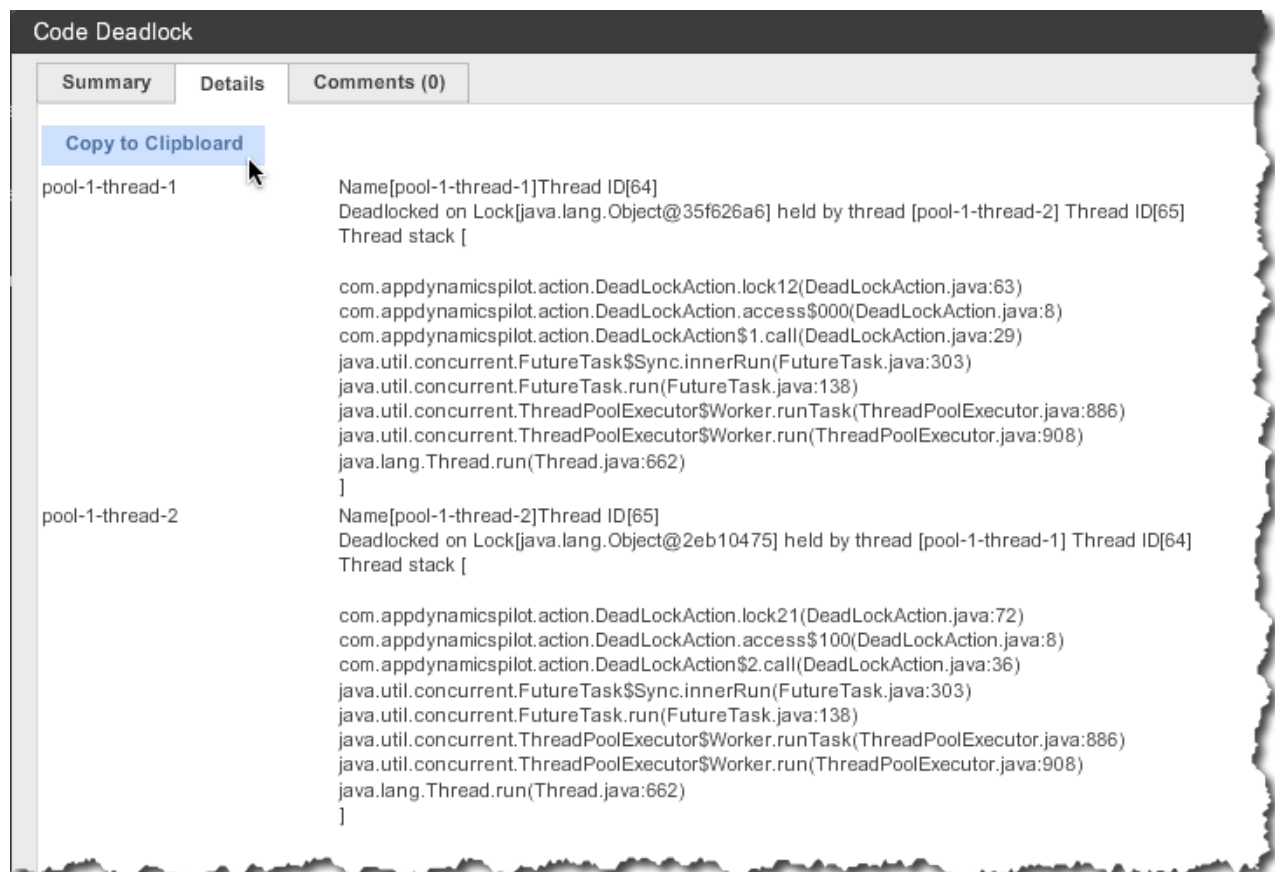
Investigating Code Deadlocks

You can troubleshoot code deadlocks by drilling down into a code deadlocks.

1. In the **Events** window click a **Code Deadlock**.



2. In the **Code Deadlock** window click the **Details** tab.



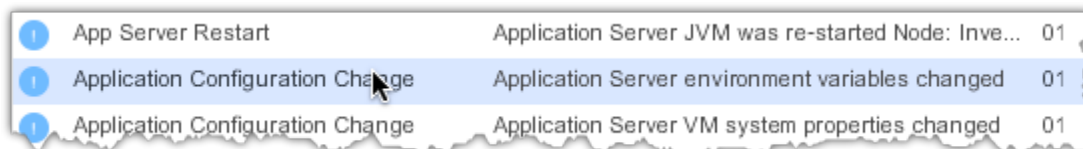
3. Use this information to troubleshoot code deadlock issues. Use the **Copy to Clipboard** button to save the deadlock details.

Investigating Application Change Events

You can view application changes by drilling down into application change events.

- ✓ By default AppDynamics reports events when applications are deployed, app servers are restarted, and configuration parameters are changed. Since these are not problems, they are indicated by a blue icon.

1. Click a change event to see a summary and details, for example:



2. Use this information to view application changes. Use the **Copy to Clipboard** button to save the change details.



Administer App Agents for Java

Resolving Configuration Issues App Agent for Java

- [Resolving App Agent for Java Startup Issues](#)
 - [Locating the App Agent for Java Log Files](#)
 - [Resolving Incomplete Agent Configuration Issues](#)
 - [Unblocking the Controller Port](#)
 - [Correcting File Permission Issues](#)
- [Learn More](#)

This topic discusses techniques for finding and interpreting the information in the App Agent for Java log files.

Resolving App Agent for Java Startup Issues

After sending a request to your web application, data should appear in the UI. If no data appears, check the following:

1. You have re-started 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 [App Agent for Java Configuration Properties](#).

4. Check the Agent logs directory located at <Agent_Installation_Directory>/logs/<Node_Name> for the agent.log file.

5. Verify that the Agent is compatible with the Controller. For details see [Agent - Controller Compatibility Matrix](#).

Locating the App Agent for Java Log Files

Agent log files are located in the <Agent_Installation_Directory>/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 App Agent for Java 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:

Error Message	Solution
Cannot connect to the Agent - ERROR com.singularity.XMLConfigManager - Incomplete Agent Identity data, Invalid Controller Port Value []	<p>This indicates that the value for the controller port in controller-info.xml is missing. Add the port value, along with the host value (<your-host-name>), to fix this error.</p>  <ul style="list-style-type: none"> • For on-premise Controller installations: Default port value is 8090 for HTTP and 8181 for HTTPS. • For Controller SaaS service: Default port value is 80 for HTTP and 443 for HTTPS.
Caused by: com.singularity.ee.agent.configuration.a: Could not resolve agent-controller basic configuration	<p>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.</p>

For more information about agent properties see [App Agent for Java Configuration Properties](#).

Unblocking the Controller Port

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

Error Message	Solution
<p>ERROR com.singularity.CONFIG.ConfigurationChannel - Fatal transport error: Connection refused WARN com.singularity.CONFIG.ConfigurationChannel - Could not connect to the controller/invalid response from controller, cannot get initialization information, controller host \x.x.x.x\, port 8090, exception Fatal transport error: Connection refused</p>	<p>Try to ping <your-host-name> from the machine where you have configured the Application Server Agent. If it works, then confirm if the Controller port is not blocked in your network.  To check if a port was blocked in the network; use command: netstat -an for Windows and nmap for Linux.  * For on-premise Controller installations: Default port value is 8090 for HTTP and 8181 for HTTPS. • For Controller SaaS service: Default port value is 80 for HTTP and 443 for HTTPS.</p>

Correcting File Permission Issues

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

Error Message	Solution
<p>ERROR com.singularity.JavaAgent - Could Not Start Java Agent com.singularity.ee.agent.appagent.kernel.spi.c : Could not start services"</p>	<p>This is usually caused because of incorrect permissions for log files. Confirm if the user who is running the server, has read and write permission on the agent directories. If the user has chmod a-r equivalent permission, change the permission to chmod a+r "<agent_directory>"</p>

Learn More

- [Install the App Agent for Java](#)
- [App Agent for Java Configuration Properties](#)

App Agent for Java Configuration Properties

- [Where to Configure App Agent Properties](#)
 - [Creating and Registering Tiers](#)
- [Example Java App Agent controller-info.xml File](#)
- [Example Startup-up Using System Properties](#)
- [Java App Server Agent Properties](#)
 - [Agent-Controller Communication Properties](#)
 - [Controller Host Property](#)

- Controller Port Property
- SSL Configuration Properties
 - Controller SSL Enabled Property
 - Controller Keystore Password Property
 - Controller Keystore Filename Property
 - Force Default SSL Certificate Validation Property
- Agent Identification Properties
 - Application Name Property
 - Tier Name Property
 - Node Name Property
 - Reuse Node Name Property
 - Reuse Node Name Prefix Property
- Multi-Tenant Mode Properties
 - Account Name Property
 - Account Access Key Property
- Proxy Properties for the Controller
 - Proxy Host Property
 - Proxy Port Property
 - Proxy User Name Property
 - Proxy Password Property
- Other Properties
 - Enable Orchestration Property
 - Agent Runtime Directory Property
 - Redirect Logfiles Property
 - Force Agent Registration Property
 - Reuse Node Name Property
 - Auto Node Name Prefix Property
 - Cron/Batch JVM Property
 - Unique Host ID Property
- [Learn More](#)

Where to Configure App Agent Properties

You can configure the App Server Agent properties:

- in the controller-info.xml file in the <Agent_Installation_Directory>/conf directory
- in the system properties (-D options) in the JVM startup script

The system properties override the settings in the controller-info.xml file.

For shared binaries among multiple JVM instances, AppDynamics recommends using a combination of the xml file and the start-up properties to configure the app agent. Configure all the properties common to all the JVMs in the controller-info.xml file. Configure the properties unique to a JVM using the system properties in the start-up script.

For example:

- For multiple JVMs belonging to the same application serving different tiers, configure the application name in the controller-info.xml file and the tier name and node name using the system properties.
- For multiple JVMs belonging to the same application and the same tier, configure the

application name and the tier name in the controller-info.xml file and the node name using the system properties.

After you configure agent properties, confirm that the javaagent argument has been added to the JVM startup script. For more information, see [Java Server-Specific Installation Settings](#).

For some properties, you can use system properties already defined in the start-up script as the App Server Agent property values. For more information, see [Configure App Agent for Java to Use Existing System Properties](#).

Creating and Registering Tiers

You can create a tier in the Controller prior to setting up any agents. Alternatively, an agent can register its tier with the Controller the first time, and only the first time, that it connects with the Controller. If a tier with the name used to connect already exists, the agent is associated with the existing tier.

Example Java App Agent controller-info.xml File

```
<?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></account-name>
  <account-access-key></account-access-key>

  <force-agent-registration>false</force-agent-registration>

</controller-info>
```

Example Startup-up Using System Properties

The following command uses the system properties to start the agent that monitors the ACME Online sample application's Inventory tier. Note that the system properties are case-sensitive.

```
java -javaagent:/home/appdynamics/AppServerAgent/javaagent.jar
-Dappdynamics.controller.hostName=192.168.1.20
-Dappdynamics.controller.port=8090
-Dappdynamics.agent.applicationName=ACMEOnline
-Dappdynamics.agent.tierName=Inventory
-Dappdynamics.agent.nodeName=Inventory1 SampleApplication
```

Java App Server Agent Properties

This section describes the Java App Agent configuration properties, including their controller-info.xml elements and their system property options.

Agent-Controller Communication Properties

Controller Host Property

Description: This is the host name or the IP address of the AppDynamics Controller. Example values are 192.168.1.22 or myhost or myhost.abc.com. This is the same host that you use to access the AppDynamics browser-based user interface. For an on-premise 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

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 Property

Description: 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 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

Type: Positive Integer

Default: For On-premise installations, port 8090 for HTTP and port 8181 for HTTPS are the defaults.

For the SaaS Controller Service, port 80 for HTTP and port 443 for HTTPS are the defaults.

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 Property

Description: When set to true, this property 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

Type: Boolean

Default: False

Required: No

Controller Keystore Password Property

Description: This is an encrypted value of the Controller certificate password. See [Password Encryption Utility](#).

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

System Property: Not applicable

Type: Boolean

Default: None

Required: No

Controller Keystore Filename Property

Description: By default the agent looks for a Java truststore file named cacerts.jks in the configuration directory: <agent install directory>/conf. Use this property to enable full validation of Controller SSL certificates with a different Java truststore file. See [Enable SSL for Java](#).

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

System Property: Not applicable

Type: String

Default: None

Required: No

Force Default SSL Certificate Validation Property

Description: This property allows you 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-Premise 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 SaaS controller, full validation is performed.
 - If the agent is connecting to an on-premise 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-premise controller, and there is no cacerts.jks file, then minimal validation is performed

Element in controller-info.xml: Not applicable

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

Type: Boolean

Default: None

Required: No

Agent Identification Properties

Application Name Property

Description: This is 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

Type: String

Default: None

Required: Yes

Tier Name Property

Description: This is the name of the logical 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 / AppServer start-up script already has a system property that references the tier, such as -Dserver.tier, you could use \${server.tier} as the tier name. For more information, see [Configure App Agent for Java to Use Existing System Properties](#).

See [Name Business Applications, Tiers, and Nodes](#).

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

System Property: -Dappdynamics.agent.tierName

Type: String

Default: None

Required: Yes

Node Name Property

Description: This is the name of the JVM node.

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

If your JVM / AppServer start-up 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}.MyNode to define the node name. See [Configure App Agent for Java to Use Existing System Properties](#) 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 Property](#).

See [Name Business Applications, Tiers, and Nodes](#).

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

System Property: -Dappdynamics.agent.nodeName

Type: String

Default: None

Required: Yes

New

Reuse Node Name Property

Description: This system property enables the reuse of node names.

This property is useful in zOS Dynamic Workload Manager based-environments where new JVMs are launched and shutdown based on actual work load. 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).

Use this option in environments where the node name can't be specified in the server startup script, and therefore needs to be auto-generated. Every node creates its own metrics and the names are pooled to make sure the Controller isn't overloaded with too many metric names. The node name is generated by the Controller and sent back to the agent. The Controller recycles node names based on the node retention period property.

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

System Property: -Dappdynamics.agent.reuse.nodeName

Type: Boolean - valid values are "true" or "false"

Default: String.

Required: No

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 etc. depending on how many nodes are running in parallel. Later, the node names are reused by the controller.

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

Reuse Node Name Prefix Property

Description: This property directs the Controller to generate node names dynamically with the prefix specified.

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

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

Type: Boolean - valid values are "true" or "false"

Default: false, when set to "true", you do not need to specify a node name.

Required: No

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 etc. depending on how many nodes are running in parallel. Later, the node names are reused by the controller.

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

Multi-Tenant Mode Properties

Description: If the AppDynamics Controller is running in multi-tenant mode or if you are using the AppDynamics SaaS Controller, specify the account name and account access key for this 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-install>/initial_account_access_info.txt file.

If the Controller is running in single-tenant mode (the default) there is no need to configure these values.

Account Name Property

Description: This is the account name used to authenticate with the Controller.

Element in controller-info.xml: <account-name>

System Properties: -Dappdynamics.agent.accountName

Type: String

Default: None

Required: Yes for AppDynamics SaaS Controller and other multi-tenant users; no for single-tenant users.

Account Access Key Property

Description: This is the account access key used to authenticate with the Controller.

Element in controller-info.xml: <account-access-key>

System Properties: -Dappdynamics.agent.accountAccessKey


Type: String

Default: None

Required: Yes for AppDynamics SaaS Controller and other multi-tenant users; no for single-tenant users.

Proxy Properties for the Controller

These properties route data to the Controller through a proxy.

 Proxy authentication cannot be used in conjunction with SSL.

Proxy Host Property

Description: This is the proxy host name or IP address.

Element in controller-info.xml: Not applicable

System Property: -Dappdynamics.http.proxyHost

Type: String

Default: None

Required: No

Proxy Port Property

Description: This is the proxy HTTP(S) port.

Element in controller-info.xml: Not applicable

System Property: -Dappdynamics.http.proxyPort

Type: Positive Integer

Default: None

Required: No

Proxy User Name Property

Description: *New for 3.8.1* This is the name of the user that is authenticated by the proxy host.

Element in controller-info.xml: Not applicable

System Property: -Dappdynamics.http.proxyUser

Type: String

Default: None

Required: No

Proxy Password Property

Description: *New for 3.8.1* This is 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 and must be in clear, unencrypted text.

Element in controller-info.xml: Not applicable

System Property: -Dappdynamics.http.proxyPasswordFile

Type: String

Default: None

Required: No

Example: -Dappdynamics.http.proxyPasswordFile=/path/to/file-with-first-line-containing-password-in-clear-text

Other Properties

Enable Orchestration Property

Description: 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>

System Property: Not applicable

Type: Boolean

Default: False

Required: No

Agent Runtime Directory Property

Description: This property sets the runtime directory for all runtime files (logs, transaction configuration) for nodes that use this agent installation. 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.

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

System Property: -Dappdynamics.agent.runtime.dir

Type: String

Default: <Agent_Installation_Directory>/nodes

Required: No

Redirect Logfiles Property

Description: This property sets the destination directory to which to redirect log files for a node.

Element in controller-info.xml: Not applicable

System Property: -Dappdynamics.agent.logs.dir

Type: String

Default: <Agent_Installation_Directory>/logs/<Node_Name>

Required: No

Force Agent Registration Property

Description: Set to true only under the following conditions:

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

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

System Property: Not applicable

Type: Boolean

Default: False

Required: No

Reuse Node Name Property

Description: Set this property if you want the Controller to generate unique node names automatically using a prefix.

You can specify the prefix in the [Auto Node Name Prefix Property](#). If you do not provide a prefix but set the reuse.nodeName property to true, the Controller uses the tier name as a prefix.

This property is useful for dynamic multi-tier clustered applications with many JVMs that have short life spans. It allows AppDynamics to reuse node names and to capture historical data for these short-lived nodes after they become historical or are deleted.

Element in controller-info.xml: Not applicable

System Property: -Dappdynamics.agent.reuse.nodeName

Type: Boolean

Default: None

Required: No

Auto Node Name Prefix Property

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

The Controller generates node names based on the prefix concatenated with a number, which is incremented sequentially. For example, if you assign a value of "mynode" to this property, the Controller generates node names "mynode-1", "mynode-2" and so on.

If one of the nodes is deleted and the [Reuse Node Name Property](#) is true, the Controller will re-use the deleted node name.

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 Property

Description: Set this property to true if the JVM is a batch/cron process or if you are instrumenting the main() method. 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 Property

Description: UniqueHostId 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 this property is set on the app agent, it must be set on the machine agent as well.

See [Configure Multiple Standalone Machine Agents for One Machine for Java](#).

System Property: -Dappdynamics.agent.uniqueHostId

Type: String

Default: None

Required: No

Learn More

- [Name Business Applications, Tiers, and Nodes](#)

- [Configure App Agent for Java for JVMs that are Dynamically Identified](#)
- [Configure App Agent for Java to Use Existing System Properties](#)
- [Java Agent on z-OS or Mainframe Environments Configuration](#)

Configure and Start an Agent Logging Session

1. In the **Agents** tab of the node dashboard, scroll down to the Agent Logs panel.
2. Click **Start Agent Logging Session**.

The Agent Logging Session window opens.

Agent Logging Session

Duration of logging session to run: 5 minutes

Type of logging to run:

- ☐ Application Wide Configuration
- ☐ BT Registration
- ☐ Current Time
- ☒ Events
- ☒ Metric Data
- ☐ Metric Registration
- ☐ One Way Agent
- ☐ Request Segment data
- ☐ System Agent Registration
- ☐ System Agent Reregistration
- ☐ System Agent Polling Handler
- ☐ Task Execution
- ☐ Top Summary Stats
- ☐ Application Configuration
- ☐ Transient Channel

Cancel Start Agent Logging Session

3. From the drop-down menu select the duration for which you want to log.
4. Check the check boxes for the types of requests that you want to log.
5. Click **Start Agent Logging Session**.

The selected logging sessions appear in the logging list.

The logged request and response output appears in the Controller and the agent log.

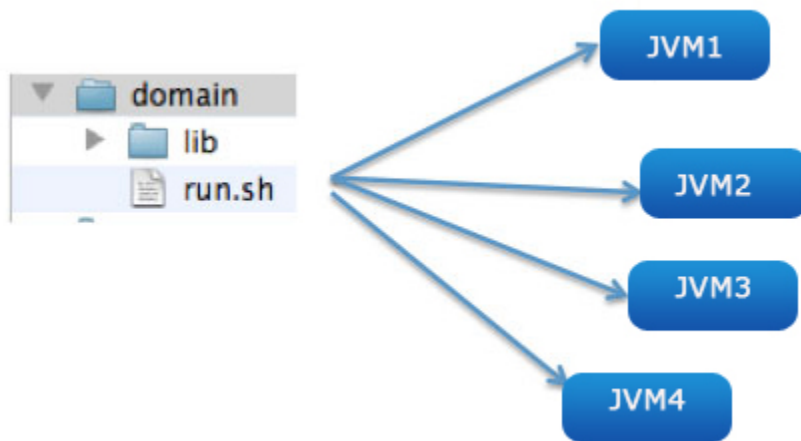


For more information, see [Request Agent Log Files](#).

Configure App Agent for Java in z-OS or Mainframe Environments

- [To name nodes automatically](#)
- [Learn More](#)

In some environments JVMs have transient identity, such as when a single script spawns multiple JVMs.



For example, an environment may consist of WebSphere on IBM Mainframes, using a dynamic workload management feature that spawns new JVMs for an existing application server (called a servant). These JVMs are exact clones of an existing JVM, but each of them has a different process ID. Based on load, any number of additional JVMs may be created.

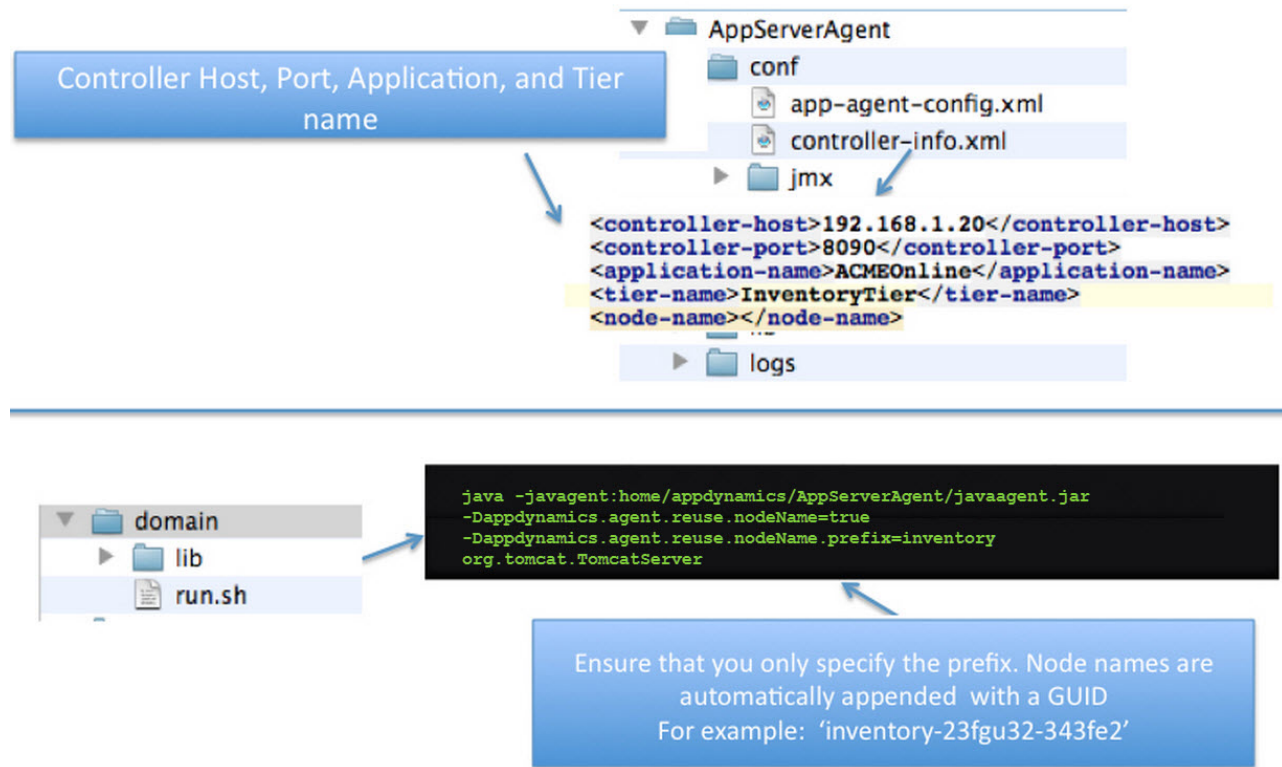
To name nodes automatically

The App Server Agent can automatically name the dynamically generated JVMs using the `appdynamics.agent.reuse.nodeName` property. See [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. You specify these properties in your startup script, using the following format:

```
-Dappdynamics.agent.reuse.nodeName=true
-Dappdynamics.agent.reuse.nodeName.prefix=<node name prefix>
```


If you are using these properties, ensure that you have not specified the node name anywhere (controller-info.xml file or as a system property) in your JVMs start-up script.

The following illustration shows the sample configuration for ACME Bookstore. This configuration will create unique node names for every instance of the virtual machine starting up in the ACME Bookstore environment.



The name of a node is available for reuse if the node has been out of contact with the Controller for a period of time that exceeds the node retention setting. For more about historical nodes, node retention and deletion, see [Remove Unused Nodes](#).

Learn More

- [App Agent for Java Configuration Properties](#)
- [Remove Unused Nodes](#)

App Agent for Java Performance Tuning

- [Business Transaction Thresholds](#)
- [Snapshot Collection Thresholds](#)
 - [Disable Scheduled Snapshots](#)
 - [Suggested Scheduling Settings](#)
 - [Suggested Diagnostic Session Settings](#)
- [Tuning Call Graph Settings](#)
 - [Suggested SQL Query Time and Parameters](#)
- [Memory Monitoring](#)

- [Learn More](#)

This topic discusses how to get the best performance from App Agents for Java.

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 [Configure 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 [Configure Thresholds](#).

Disable Scheduled Snapshots

For more information see [Transaction Snapshots](#).

Suggested Scheduling Settings

- 10 minutes for small deployments < 10 BTs
- 20 minutes for medium deployments < 10 - 50 BTs
- 30 minutes for > 50 - 100 BTs
- 60 minutes > 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

To configure call graph settings, click **Configure -> Instrumentation** and the **Call Graph Settings** tab. See [Configure Call Graphs](#).

Suggested SQL Query Time and Parameters

- Minimum SQL Query Time : 100 ms (Default is 10)
- 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 [Troubleshoot Java Memory Leaks](#).

Learn More

- [Configure Thresholds](#)
- [Configure Call Graphs](#)
- [Troubleshoot Java Memory Leaks](#)

Move an App Agent for Java Node to a New Application or Tier

- [Moving a Node to a New Application or Tier](#)
 - [To change the Java Agent associations from the UI](#)
 - [Optionally update the controller-info.xml file](#)
 - [Forcing node re-registration using the controller-info.xml file](#)
 - [Learn More](#)

If your JVM machine has both an App Agent for Java 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 App Agent for Java controller-info.xml file.

Moving a Node to a New Application or Tier

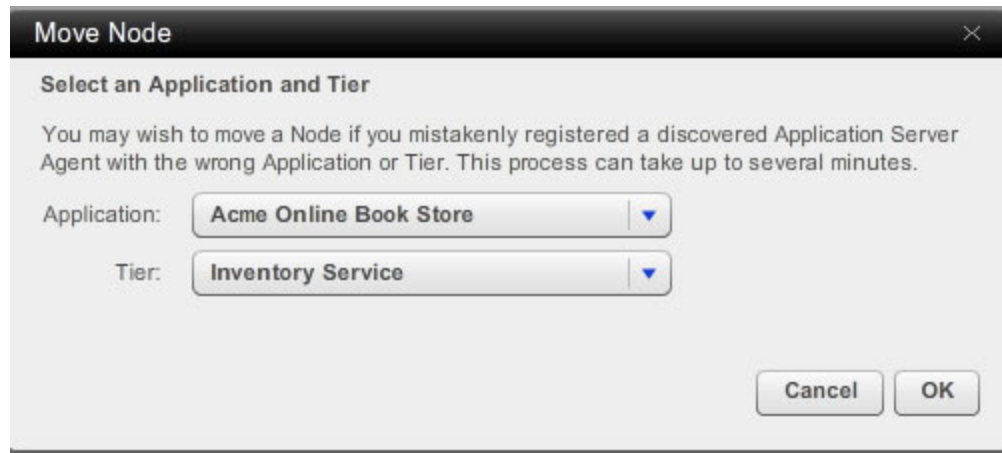
You can assign an App Agent for Java node to a new application or tier using the AppDynamics UI without restarting the JVM.

You can optionally update the controller-info.xml file and restart the JVM. You must restart the JVM when you change the associations in the controller-info.xml file.

Moving the node using the AppDynamics UI cannot be overridden by an agent configuration unless you set the force-agent-registration flag to true in the controller-info.xml file.

To change the Java Agent associations from the UI

1. Select the node that you want to move by clicking **Servers -> App Servers -> <Tier> -> <Node>**.
2. Click **Actions -> Move Node**.
4. In the Move Node window select the new application and/or tier from the drop-down menus.
5. Click **OK** to confirm the reassignment. It may take several minutes to complete.



i When you change the associations for an App Agent for Java, AppDynamics registers an Application Changes event. You can review the details of the event in the [Events](#) view.

Optionally update the controller-info.xml file

The UI does not update the controller-info.xml file. However if you restart the JVM the Controller will remember and keep the change you made from the UI.

You may want to maintain consistency with the controller-info.xml file, just for neatness' sake. Perform these two additional steps:

6. Update these configuration changes in the App Agent for Java controller-info.xml file.
7. Restart the JVM.

If it is not feasible to restart the JVM at this time, remember the change and update the file the next time you restart the JVM.

Forcing node re-registration using the controller-info.xml file

If you've moved a node in the UI and you want to move it again elsewhere using controller-info.xml, then when you restart the JVM you set the force-agent-registration property to 'true'. See [Force Agent Registration Property](#).

Learn More

- [Logical Model](#)

App Agent for Java Diagnostic Data

- [To view agent diagnostic data](#)
- [To view agent diagnostic stats](#)
- [Learn More](#)

To view agent diagnostic data

1. From an Application Dashboard, click **Actions -> View Agent Diagnostics**.

The Agent Diagnostic Data window opens and displays various metrics related to the application, tiers, and nodes.

Agent Diagnostic Data

View

last 1 day

View Agent Diagnostic Events for selected Node

Name	Overflow Calls	Transactions Successfully Registered	Transactions Registration Failed	Discovered Backends Success	Discovered Backends Registration	Metrics Upload Requests Time Skew	Metrics Upload Invalid Metrics	Metrics Uploads	Metrics Upload Requests Exceeded	Metrics Upload Requests License	Snapshots Time Skew Errors	Snapshots With Invalid Data	Snapshots Uploads	Snapshots Exceeding Limit	Events Time Skew Errors	Events Uploads	Events Exceeding Limit	Application Infrastructure Changes
<div><div></div><div></div><div></div></div> ACME Book Store Application	-	9	-	3	-	-	-	572	-	-	-	-	46	-	-	5	-	6
<div><div></div><div></div><div></div></div> ECommerce Server	-	9	-	3	-	-	-	333	-	-	-	-	15	-	-	2	-	3
<div><div></div><div></div><div></div></div> Node_8000	-	5	-	2	-	-	-	174	-	-	-	-	11	-	-	1	-	2
<div><div></div><div></div><div></div></div> Node_8003	-	3	-	1	-	-	-	159	-	-	-	-	4	-	-	1	-	2
<div><div></div><div></div><div></div></div> Inventory Server	-	-	-	-	-	-	-	121	-	-	-	-	20	-	-	1	-	2
<div><div></div><div></div><div></div></div> Order Processing Server	-	1	-	-	-	-	-	118	-	-	-	-	11	-	-	1	-	2

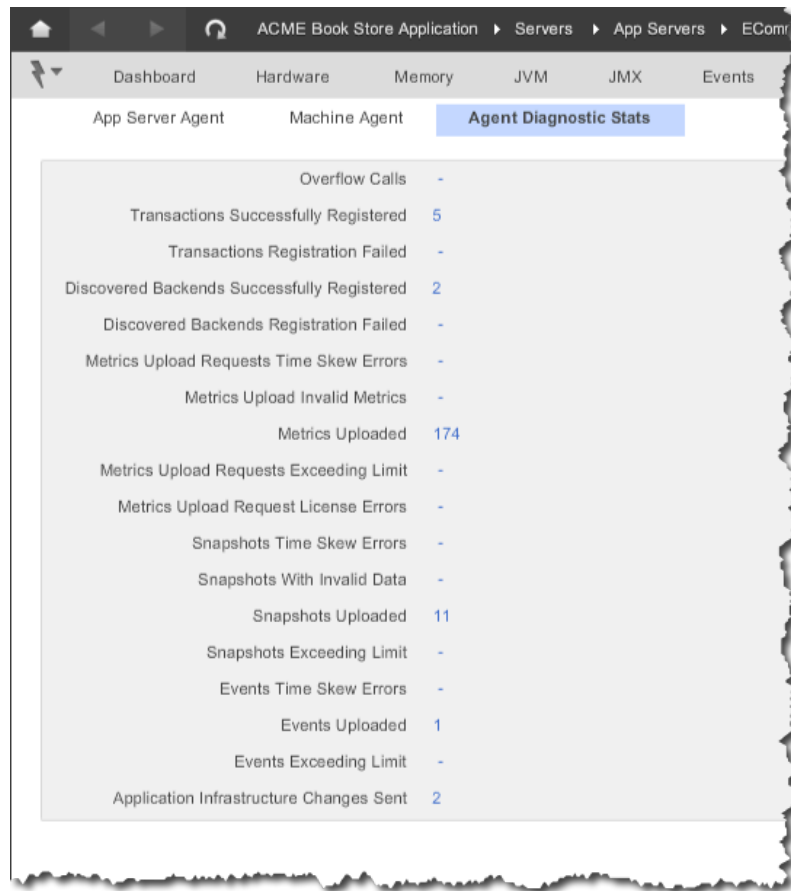
(close)

2. Select a node and click **View Agent Diagnostic Events for selected Node**.

The Agent Diagnostics Event window opens.

To view agent diagnostic stats

1. From the left navigation pane, click *Servers -> App Servers -> <Tier> -> <Node>.
2. In the Node Dashboard, click the Agents tab.
3. Click the Agent Diagnostic Stats subtab.



[Learn More](#)

- [Troubleshoot Node Problems](#)

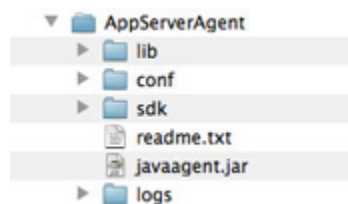
App Agent for Java Directory Structure

- [App Agent for Java Directory Structure](#)
- [The conf Directory](#)

App Agent for Java Directory Structure

The AppServerAgent.zip folder contains files for installing the App Agent for Java.

The directory structure for the agent is illustrated in the following screen shot.

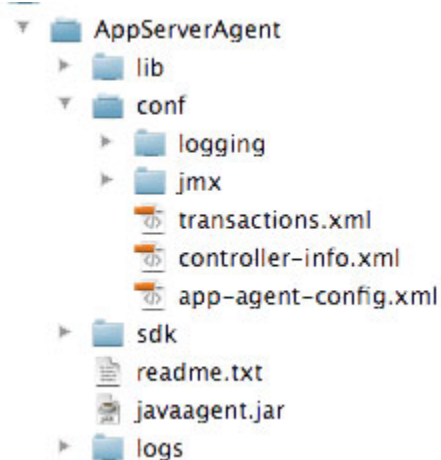


- **lib:** The lib folder has the core agent libraries and the third-party libraries.
- **conf:** The conf folder is the configuration directory that has local configuration backup.
- **sdk:** This folder contains the Javadocs and APIs to extend AppDynamics' monitoring capabilities.

- **javaagent.jar:** This JAR file has the argument to bootstrap the App Agent for Java. To enable the agent, the `--javaagent` argument for JVM startup is required. See [Install the App Agent for Java](#).
- **logs:** All logs written by the agent in this directory.

The conf Directory

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



- **transactions.xml:** This XML file has configurations for business transaction identified by the agent.
- **controller-info.xml:** This XML file has controller connectivity and identification.

The following agent settings are configured using this XML file:

- Agent communication settings (specified using `<controller-host>` and `<controller-port>` properties).
- Agent identification settings (specified using `<application-name>`, `<tier-name>`, and `<node-name>` properties).
- **app-agent-config.xml:** This XML file contains agent configuration. Any changes to the agent's local settings are specified in this file, and these changes override the global configuration. This file is typically used for short-term properties settings or for debugging agent issues.
- **jmx:** The jmx folder contains files for configuring the JMX and Websphere PMI metrics.
- **logging/log4j.xml:** This file contains flags to control logging levels for the agent. It is highly recommended not to change the default logging levels.

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

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

IBM App Agent for Java

- [Supported JVMs](#)

- [Instrumenting the IBM App Agent for Java](#)
- [Running the App Agent for Java in a WebSphere/InfoSphere Environment with WebLogic Security Enabled](#)

Under most circumstances, the IBM App Agent for Java works the same as the App Agent for Java.

This topic gathers information specific to the IBM App Agent for Java.

Supported JVMs

IBM JVM 1.5.x, 1.6.x

Instrumenting the IBM App Agent for Java

To change instrumentation for the IBM App Agent for Java, the IBM JVM must be restarted. By default the IBM App Agent for Java 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
- Aggressive snapshot policy (also called hotspot instrumentation)
- 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

Running the App Agent for Java in a WebSphere/InfoSphere Environment with WebLogic Security Enabled

If your WebSphere/InfoSphere environment includes a security-enabled WebLogic Application Server, several InfoSphere Master Data Management (MDM) Server clients require security configuration.

For more information, see [Configuring Java clients to work with WebLogic security in the IBM documentation](#).

Configure App Agent for Java for Batch Processes

- [To configure the App Agent for Java](#)
- [To use the script name as the node name](#)

[Learn More](#)

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

To configure the App Agent for Java

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

The `agent_install_dir` is the full path of the App Agent for Java installation directory.

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.

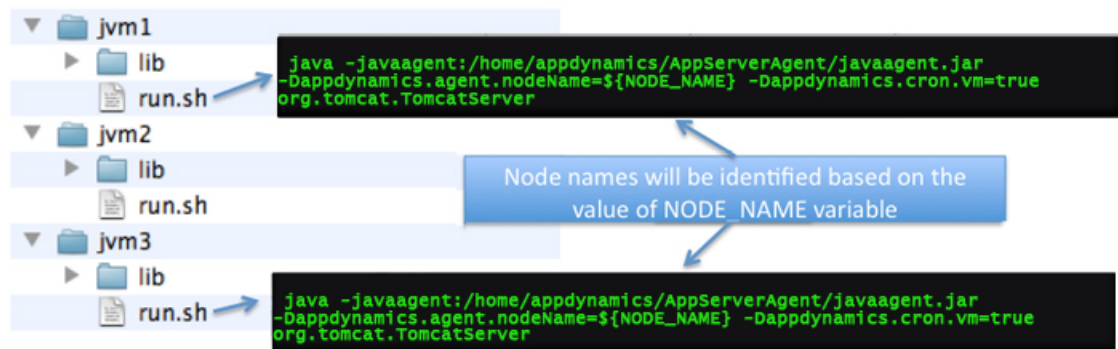
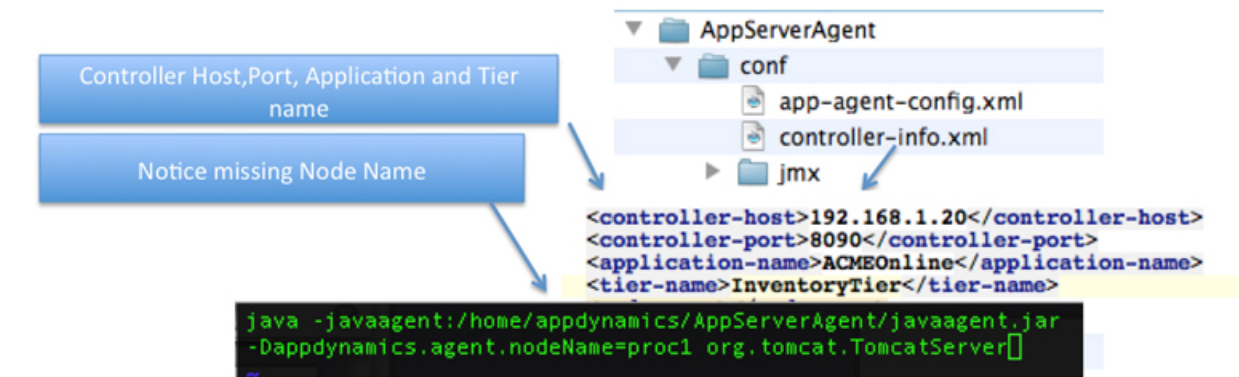
To use the script name as the node name

You can use the script name that executes the cron or batch job as the node name.

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.

```
# 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"
```

The following illustration shows the sample configuration for `controller-info.xml` and the startup script of the JVM.



Learn More

- [Configure Background Tasks for Java](#)

Configure App Agent for Java in Restricted Environments

- [To write the "startup hook" agent program](#)

Some restricted environments do not allow any changes to the JVM startup script. For these environments AppDynamics provides the `appdynamics.agent.startup.hook` property. This "startup hook" allows a single point of deployment for the agent. You create a Java main method that is invoked programmatically, before your startup script is executed.

To write the "startup hook" agent program

1. Implement a class with Java main method.
2. Create a JAR file for this class.
3. In the manifest of the JAR file, specify the class created in step 1.
4. Add the following javaagent argument and system properties (-D options) to your startup script:

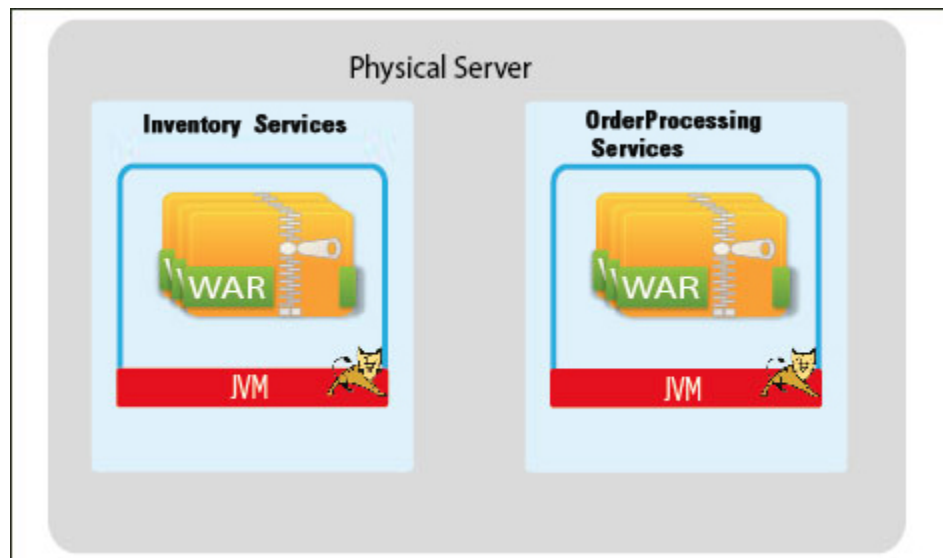
```
-javaagent:<agent_install_dir>/javaagent.jar  
-Dappdynamics.agent.startup.hook=<JAR-file>
```

Configure App Agent for Java on Multiple JVMs on the Same Machine that Serve Different Tiers

- To configure the App Agent for Java

This section describes how you can configure the App Agent for Java for multiple JVMs that are located on a single machine and are serving two tiers.

For example, the ACME Bookstore has two JVMs on the same physical server. These two JVMs are bound to two different virtual IP (one JVM is used for Order Processing Services and the other JVM is used for Inventory Services).



For such cases, follow these rules:

- All of the common information should be configured using controller-info.xml file.
- All of the information unique to a JVM should be configured using the system properties in the JVM startup script.
- Information in the startup scripts always overrides the information in the controller-info.xml file.

To configure the App Agent for Java

1. Add **application name** to **controller-info.xml** file.
2. Add **javaagent** argument and following system properties (-D options) to the start-up script to each of your JVM:

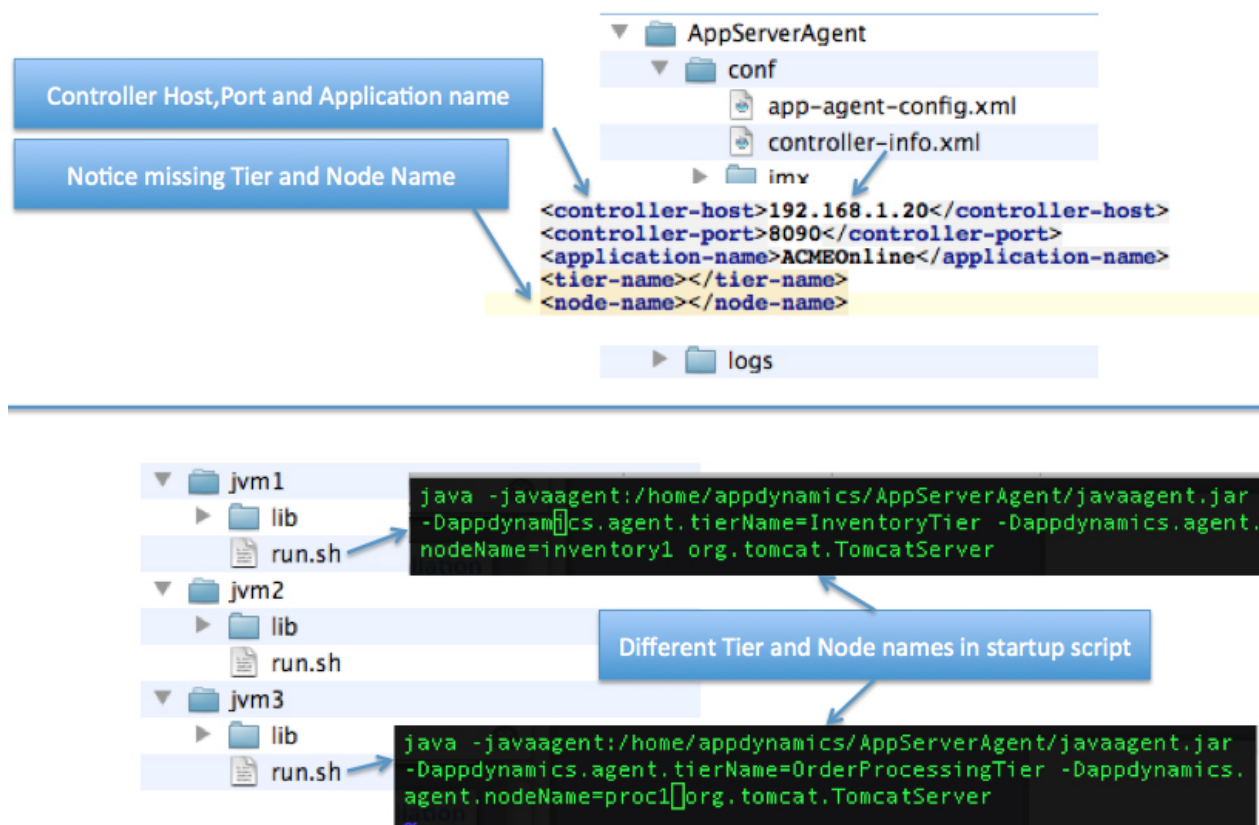
```
java -javaagent:<Agent-Installation-Directory>/javaagent.jar
-Dappdynamics.agent.tierName=$tierName -Dappdynamics.agent.nodeName=$nodeName
```

Separate the system properties with a white space character.

All agents in shared mode need a unique node name so that they can be differentiated from one another. See [Configure App Agent for Java to Use Existing System Properties](#).

The following illustration displays how this configuration is applied to ACME Bookstore:

**Add Controller Host, Port, and Application Name in controller-info.xml file
and add rest of the properties in the start-up script.**



i Tips:

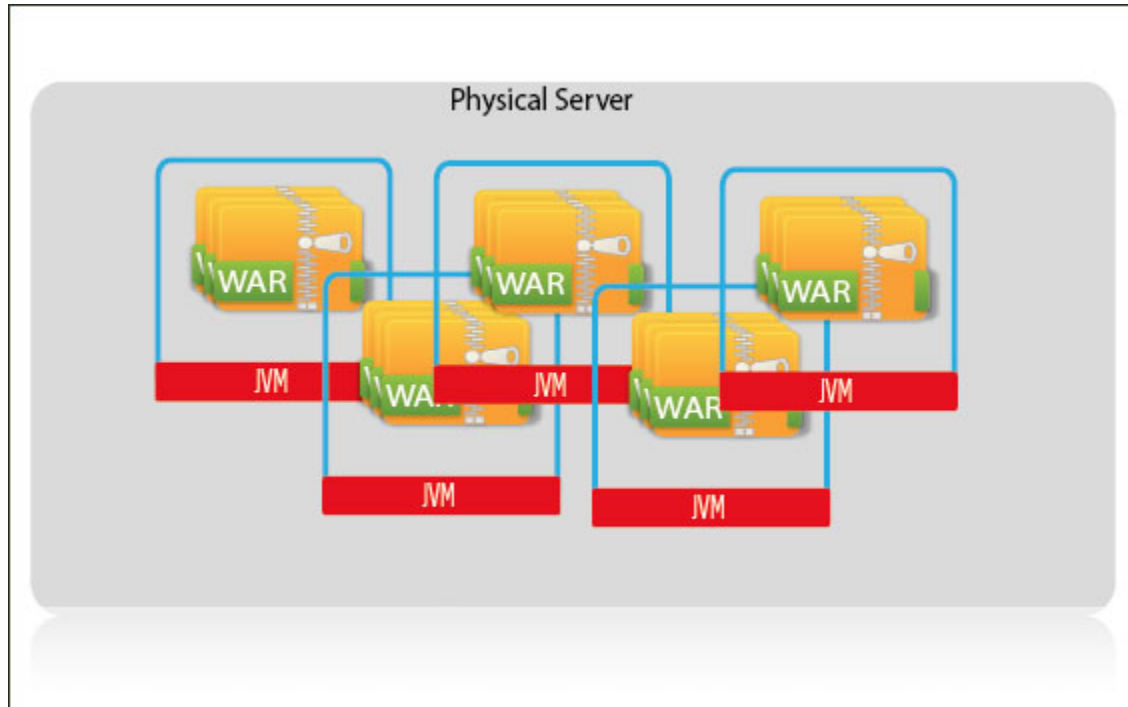
- The application and tier names can also be configured using the system properties. For more details see [App Agent for Java Configuration Properties](#).
- Some application server management consoles allow you to specify startup arguments using a web interface. See [Java Server-Specific Installation Settings](#).

Configure App Agent for Java on Multiple JVMs on the Same Machine that Serves the Same Tier

- To configure the App Agent for Java properties

This topic describes how to configure the App Agent for Java for multiple JVMs that are located on a single machine and are serving the same tier.

For example, ACME Bookstore has a physical server with five JVMs installed on it. All of these JVMs are used for the Inventory Services.



For such cases, follow these rules:

- 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 in the start-up script.
- Information in the startup scripts always overrides the information in the controller-info.xml file.

To configure the App Agent for Java properties

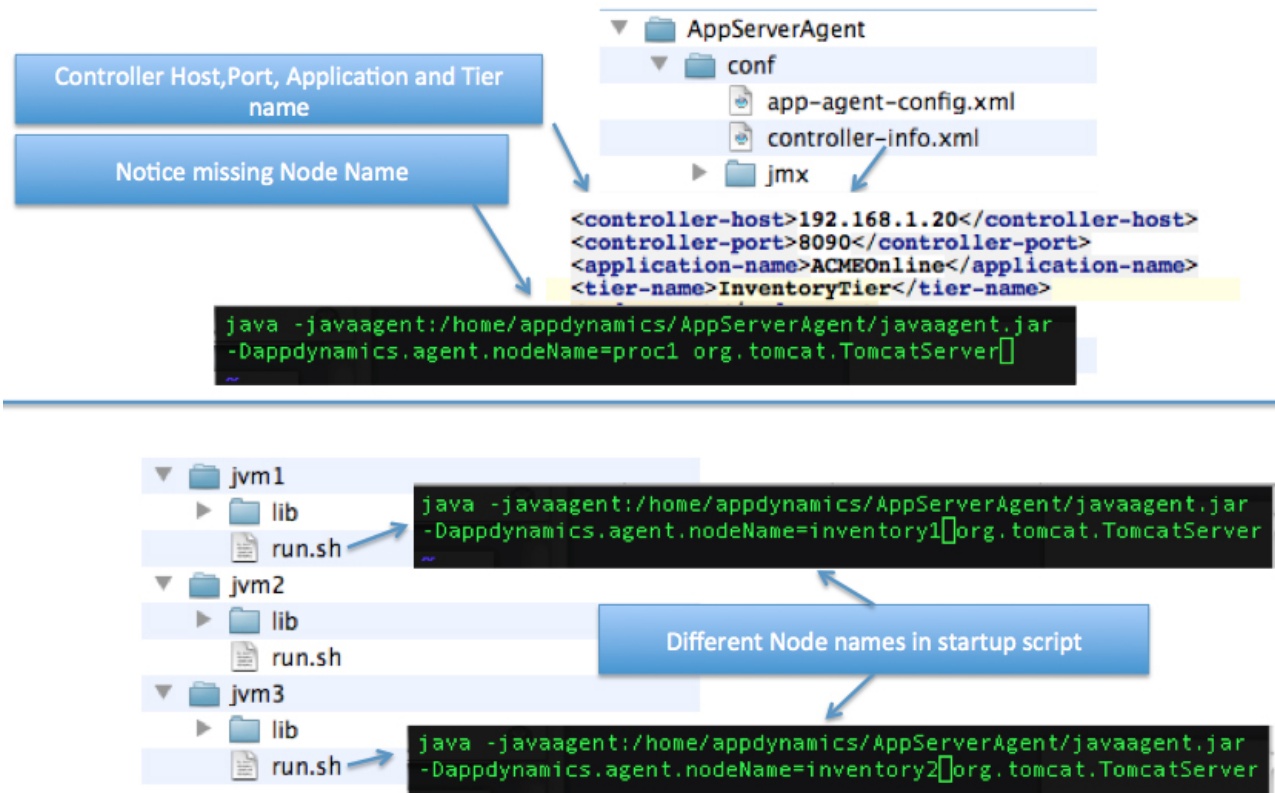
1. Provide the application and tier name in the controller-info.xml file.
2. Add the javaagent argument and system property (-D option) for the node name to the batch file or startup script of each JVM.

```
java -javaagent:<Agent-Installation-Directory>/javaagent.jar  
-Dappdynamics.agent.nodeName=$nodeName
```

Separate the system properties with a white space character.

The following illustration displays how this configuration is applied to the ACME Bookstore.

Add Controller Host, Port, Application, and Tier Name in controller-info.xml file and add Node Name in the start-up script.



The application and tier names can also be configured using the system properties. See [App Agent for Java Configuration Properties](#).

Some application server management consoles allow you to specify start-up arguments using a web interface. See [Java Server-Specific Installation Settings](#).

Configure App Agent for Java to Use Existing System Properties

- [System Properties](#)
 - [Using System Properties](#)
 - [To identify nodes](#)
 - [To identify tiers](#)
 - [Sample Agent Configuration Using System Properties](#)
- [Learn More](#)

This topic explains how to configure the App Agent for Java using the existing system property values.

System Properties

AppDynamics recommends that you use the existing system properties to configure the Agent when your environment consists of multiple JVMs on the same machine. Once you have these

variables configured, you can complete Agent installation for all JVMs by simply adding the `javaagent` argument to each JVM startup script. Then add the rest of the information to the `controller-info.xml` file.

AppDynamics recommends that you use the system properties if the same startup script is starting all the JVMs in your environment.

You can identify the node name based on the value of `-Dserver.name` and the tier name based on the value of `-Dcluster.name`.

Also, you can combine two or more system properties to identify the node or tier name. You can use `-Dhost.name` and `-Dserver.name` to identify similarly named nodes on different machines even when they belong to the same tier.

You can use existing system properties for the controller host and port, however combining is not supported in this case.

Using System Properties

Use the following syntax to represent the value of the system property in the `controller-info.xml` file.

```
${system property name}
```

You can combine multiple system properties.

```
${host.name}${server.name}
```

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

```
${host.name}_${server.name}.inventory
```

To identify nodes

```
${host.name}
```

or

```
${server.name}
```

or

```
${host.name}${server.name}
```

To identify tiers

```
${cluster.name}
```

Sample Agent Configuration Using System Properties

Consider a JVM with a script file named startserver.sh. This script file has following system property:

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

If you execute:

```
startserver.sh ecommerce01
```

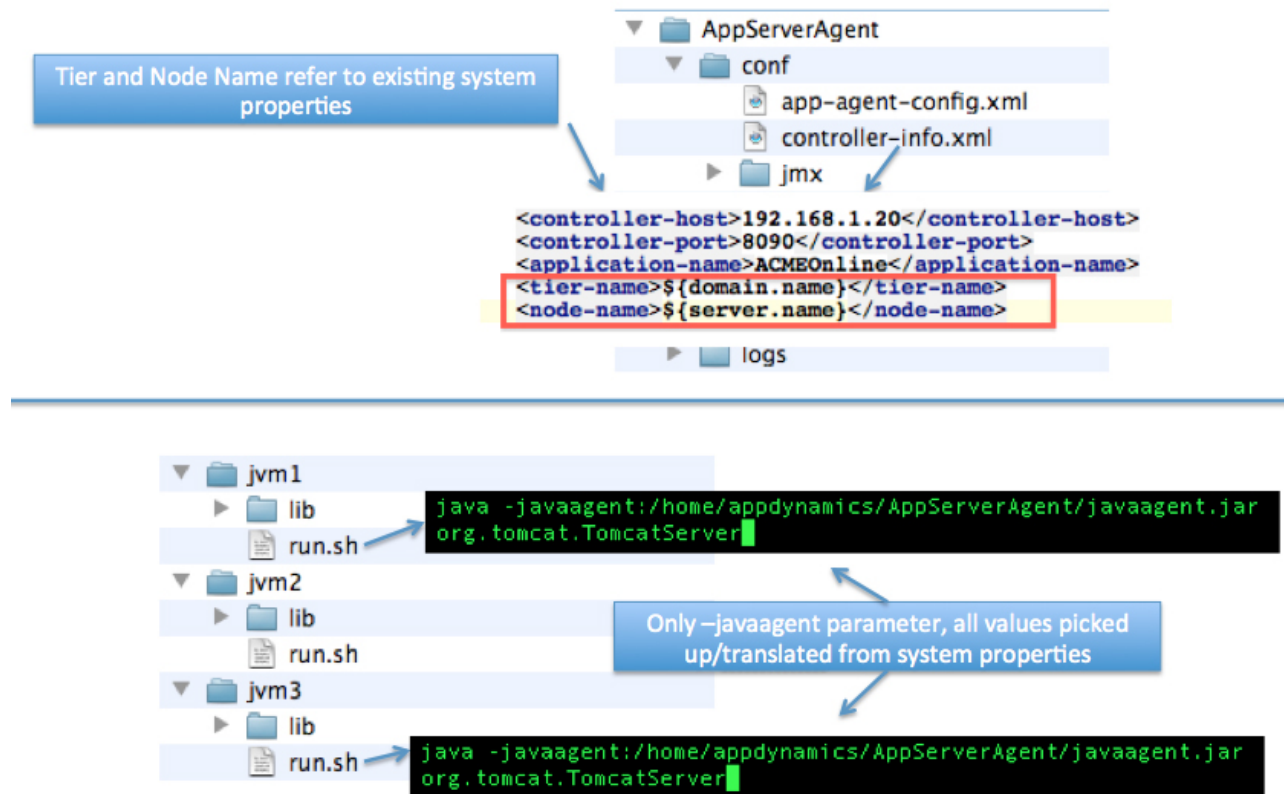
The script creates a new server named ecommerce01.

To use this system property for Agent configuration, add the appdynamics.agent.nodeName property to startserver.sh file.

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

When the script creates the new server named ecommerce01, it will be identified by AppDynamics as both a server and as a node.

The following screenshot shows a sample configuration for the controller-info.xml file and the startup script.



Learn More

- [Logical Model](#)
- [Install the App Agent for Java](#)

Administer App Agent for Java FAQ

- [App Agent for Java Administration FAQ](#)
 - Q. Why am I seeing "WARN AsyncHandOffIdentificationInterceptor - Reached maximum limit 500 of async handoff call graph samples. No more samples will be taken" message in the agent log files?

App Agent for Java Administration FAQ

Q. Why am I seeing "WARN AsyncHandOffIdentificationInterceptor - Reached maximum limit 500 of async handoff call graph samples. No more samples will be taken" message in the agent log files?

In AppDynamics 3.6 and 3.7, all Runnable, Callable and Thread are instrumented by default except for the ones that are excluded by the agent configuration in app-agent-config.xml. In some environments, this could 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 application using Closures.

To debug the cause of the message, check the call graph to see if so many asynchronous activities are legitimate. If they are not, then exclude the packages that are not really asynchronous activities. See [Configure Multi-Threaded Transactions for Java](#).

Configure App Agent for Java for JVMs that are Dynamically Identified

- [To configure the node name of the App Agent for Java](#)
- [Configuration notes](#)

This topic describes how to configure the App Agent for Java in environments where the JVMs are dynamic.

To configure the node name of the App Agent for Java

1. Add the **application** and **tier name** to the controller-info.xml file.
2. Add the javaagent argument and the following system properties (-D options) to the startup script of the JVMs:

```
java -javaagent:<agent-install-dir>/javaagent.jar  
-Dappdynamics.agent.nodeName=${NODE_NAME}
```

Configuration notes

The system properties are separated by a white space character.

The <agent-install-dir> references the full path of the App Agent for Java installation directory.

The token \${NODE_NAME} identifies the JVMs dynamically and names these JVMs based on the parameter value passed during the execution of the startup script for your JVM process.

The application and tier names can also be configured using the system properties. For example, you can configure the agent to direct the Controller to create node names using a prefix and to reuse node names so the Controller is not overloaded, using the `-Dappdynamics.agent.reuse.nodeName.prefix` and `=Dappdynamics.agent.reuse.nodeName` options respectively. For more details see [App Agent for Java Configuration Properties](#).

Some application server management consoles allow you to specify startup arguments using a web interface.

For details see [Java Server-Specific Installation Settings](#).

Add the Agent into an Embedded JVM

In order to add the agent into an embedded JVM you must first identify the java process in which you want to embed the agent and then apply the agent to that java process.

1. From the command line, use the JPS tool to list the process id and the fully-qualified java main class, as follows:

```
C:\>jps -l
```

The system returns information such as,

```
2160 sun.tools.jps.Jps
2020 org.apache.catalina.startup.Bootstrap
```

2. Apply the agent to the relevant process id as follows:

```
C:\>java -jar javaagent.jar <java>process_id>
```

Note: Ensure the AppDynamics Pro Controller has all the required information for the JVM, including controller-host, port, tier, and node name.