Using Go SDK

On this page:

Before you can start instrumenting your applications with the Go SDK, you need to get the current version of the Go SDK. The Go Language Agent and Download AppDynamics Software topic describes how to get and install the SDK.

Once you have the SDK, you can instrument your application. The SDK provides routines for creating and managing Business Transactions, transaction snapshots, backends, and exit calls. This topic provides an overview of these concepts.

About Go Application Instrumentation

The following topics provide a general outline of how to implement instrumentation in your Go application:

- Import the AppDynamics API File to the Application
- Initialize the Controller Configuration
- Initialize the Agent
- Create Business Transactions
- Manage Business Transaction Snapshots
- Create Backends
- Manage Exit Calls
- Correlate with Other Business Transactions

- Terminate the Agent

Import the AppDynamics Package to the Application

After downloading the SDK, you are ready to add AppDynamics instrumentation to your Go application. The first step is to import the AppDynamics package:

```
import appd "appdynamics"
```

Initialize the Controller Configuration

Controller information settings permit the SDK to connect to the Controller. Some settings are required for all applications, while others are required only for certain types of application environments. For example, if the SDK needs to connect to the Controller via an internal proxy in your network, set up the connection settings for the proxy.

See 'Agent Configuration Settings' in Go SDK Reference for a complete list of the settings and information about which ones are required.

In your application, assign values to the required settings. For example, to set the Controller connection information:

```
cfg := appd.Config{}

cfg.AppName = "exampleapp"
cfg.TierName = "orderproc"
cfg.NodeName = "orderproc01"
cfg.Controller.Host = "my-appd-controller.example.org"
cfg.Controller.Port = 8080
cfg.Controller.UseSSL = true
cfg.Controller.Account = "customer1"
cfg.Controller.AccessKey = "secret"
cfg.InitTimeoutMs = 1000  // Wait up to 1s for initialization to finish
```

Notice the `InitTimeoutMs` field. Once you initialize the configuration, you pass the configuration object to the call to initialize the agent via `InitSDK()`. The `InitTimeoutMs` field can have these values:

- When set to 0, the default, the `InitSDK()` function operates as an asynchronous action, so that the initialization call does not block your program.
- Setting the value to -1 instructs the program to wait indefinitely until the controller configuration is received by the agent, that is, the `InitSDK()` method returns control. This is useful when you want to capture short-running Business Transactions that occur at application startup and you do not mind the delay of waiting for the Controller to send the configuration.
- Alternatively, set it to a specific number of milliseconds to wait.

If you use a multi-tenant Controller (SaaS or on-premises multi-tenant), you need to create the context for the multi-tenant environments using the `AddAppContextToConfig()` method. You can then pass the context as a parameter to methods for performing particular operations, such as starting a BT or adding custom metrics. See Go SDK Reference for more about `AddAppContextToConfig()` and related methods.

Initialize the Agent

In your main function, initialize the agent by passing the configuration structure to `InitSDK()`.

If `InitSDK()` returns nil, the agent is initialized successfully. If an error returns, it is likely because the agent could not reach the Controller.

The following example illustrates how to initialize the SDK:

```
if err := appd.InitSDK(&cfg); err != nil {
        fmt.Printf("Error initializing the AppDynamics SDK\n")
} else {
        fmt.Printf("Initialized AppDynamics SDK successfully\n")
}
```

Create Business Transactions

Define a Business Transaction by enclosing the code that constitutes the request that you want to monitor between `StartBT()` and `EndBT()`calls. `StartBT()` returns a handle to use in subsequent routines that affect that Business Transaction.

If you are creating a Business Transaction that correlates with an upstream Business Transaction, pass the correlation header of the upstream transaction so the new transaction that you are creating can correlate with it. See `GetExitcallCorrelationHeader()`in the Go SDK Reference. If the transaction does not need to correlate with another transaction, pass an empty string for the correlation header parameter.

You can optionally store the Business Transaction handle in the global handle registry with a guid for easy retrieval later using `StoreBT()`. Retrieve the handle from the global handle registry `using GetBT()`.

The following shows an example of setting a Business Transaction:

```
// start the "Checkout" transaction
btHandle := appd.StartBT("Checkout", "")

// Optionally store the handle in the global registry
appd.StoreBT(btHandle, my_bt_guid)
...

// Retrieve a stored handle from the global registry
myBtHandle = appd.GetBT(my_bt_guid)

// end the transaction
appd.EndBT(btHandle)
```

Between starting and ending the transaction, you can perform operations such as adding errors to the Business Transaction, defining the transaction snapshot attributes, adding backends and exit calls, and so on.

When the Business Transaction ends, via a call to `EndBT()`, the agent stops reporting metrics for the Business Transaction.

Add Business Transaction Errors

Use `AddBTError()` to report Business Transaction errors. If you set this function's `markBTAsError` parameter, the transaction is reported as an error transaction when the error occurs.

The SDK provides constants classifying the error level as APPD_LEVEL_NOTICE, APPD_LEVEL_WARNING or APPD_LEVEL_ERROR.

Manage Business Transaction Snapshots

When the agent is monitoring a Business Transaction, it automatically creates Transaction Snapshots, which describe instances of the Business Transaction at certain points in time. Transaction snapshots are extremely useful for troubleshooting poor performance because they contain a lot of detail.

You do not have to modify anything to create these snapshots, other than creating the Business Transaction, but you can add calls to:

- Find out if a snapshot is being taken
- Provide additional data in a snapshot
- Set the URL for a snapshot

Determine if the Agent is Taking a Snapshot Now

The agent is not always collecting snapshots, because that would be expensive. By default, it collects a snapshot every ten minutes, but this schedule is configurable. See 'Configure Snapshot Periodic Collection Frequency' in Transaction Snapshots for information about the frequency.

You can determine if a snapshot is happening using `IsBTSnapshotting()`, which returns `true` if the agent is collecting a snapshot. The main reason to do this is to avoid the wasted overhead for collecting user data for a snapshot or setting the snapshot URL if no snapshot is currently being collected.

Add Business Transaction User Data

You can optionally add data to transaction snapshots. For example, you might want to know which users are getting a lot of errors, or from which regions users are experiencing slow response times, or which methods are slow. In the AppDynamics UI, the data appears in the 'USER DATA' tab of the transaction snapshot.

If a snapshot is occurring, use `AddUserDataToBT()` passing a key and value for the data that you want the snapshot to collect.

Add Snapshot URL

If a snapshot is occurring, you can set a URL for the current snapshot using `SetBTURL()`.

Set Snapshot Example

```
func setSnapshotAttributes(bt appd.BtHandle, key, value string) {
        if appd.IsBTSnapshotting(bt) {
                appd.AddUserDataToBT(bt, key, value)
                appd.SetBTURL(bt, "user/login")
        }
}
```

Create Backends

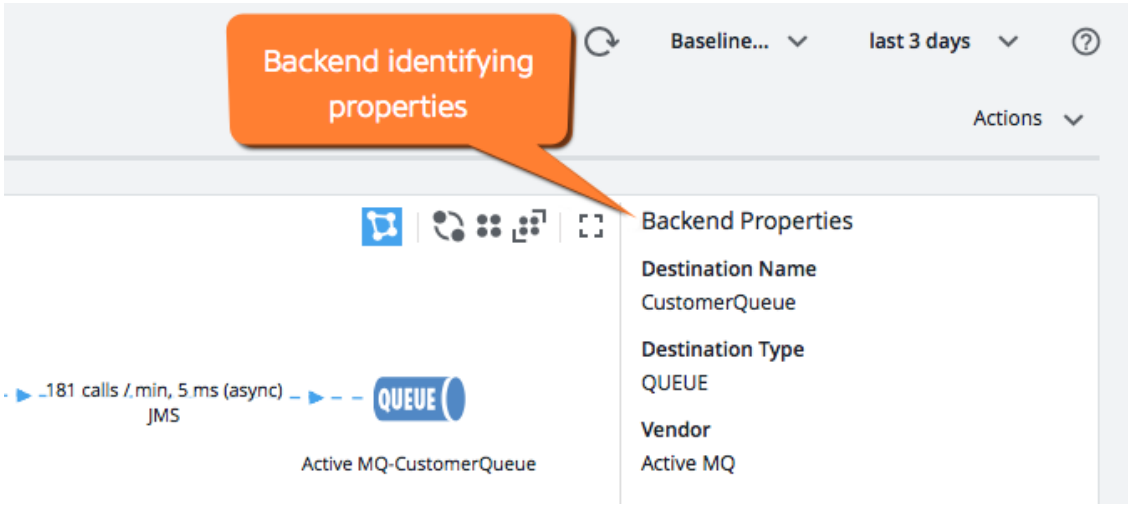
A backend is a database or a remote service such as a message queue, HTTP service or cache service that your application uses. A backend component is not itself monitored by the application agent, but the agent monitors calls to it from instrumented servers. You need to create backends in the instrumented environment so that the agent registers them. Creating and adding the backend to the instrumented application involves:

- Naming the backend
- Setting its identifying properties
- Optionally configuring how the backend is presented in the AppDynamics UI

Identifying Properties

A backend has identifying properties. These vary depending on the type of the backend and the types of information that you want to display. The Controller displays identifying properties in backend dashboards. You must set at least one identifying property for the type of any backend that you plan to add.

The following shows the backend properties in the Controller UI for an Oracle database:



Resolve to a Tier

By default the Controller doesn't display a detected backend as a separate entity in flow maps, but the agent reports its metrics as part of the downstream tier. If you want to display the backend as a separate component and not *resolved to the tier* set `resolve` to `false`. See Resolve Remote Services to Tiers for more information.

Backend Example

The following listing shows an example of setting up a database backend.

```
backendName := "Cart Product Database"
backendType := "DB"
backendProperties := map[string]string {
        "DATABASE": "sqlite3",
}
resolveBackend := false

appd.AddBackend(backendName, backendType, backendProperties, resolveBackend)
```

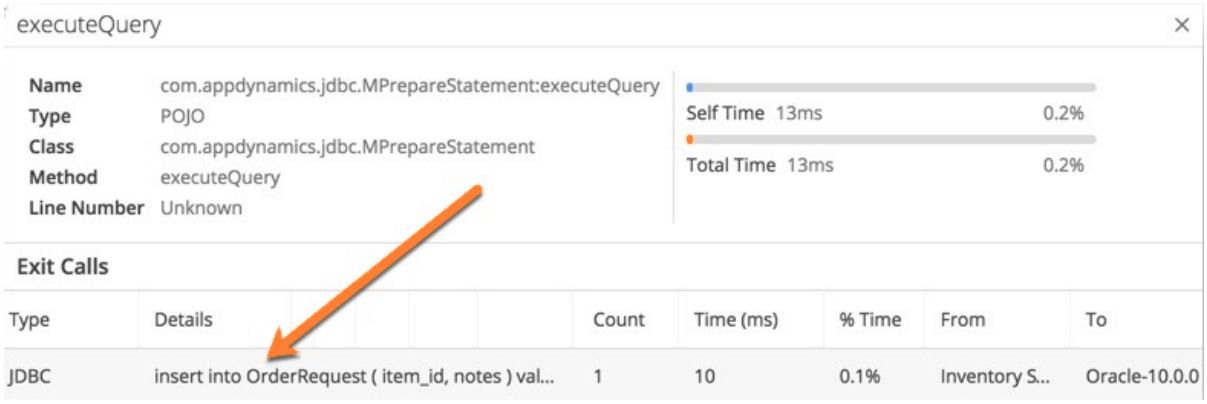
Manage Exit Calls

When an application makes a call to another component, such as a detected backend or another application server, the agent reports metrics on those calls.

Define an exit call by enclosing the code that constitutes the exit call between `StartExitCall()` and `EndExitcall()` calls. `StartExitcall()` returns a handle to use in subsequent routines that affect that exit call. An exit call occurs in the context of a Business Transaction.

You can optionally store the exit call handle in the global handle registry with a guid for easy retrieval later using `StoreExitcall()`. Retrieve the handle from the global handle registry `using GetExitcall()`.

You can optionally add details to an exit call as any arbitrary string with `SetExitcallDetails()`. The details are reported in the exit call details in the transaction snapshots in the Controller UI:



You can also add errors to an exit call using `AddExitcallError()`. Use the enum for the error levels. You can also add an error message.

Simple Exit Call Example

```
// start the exit call to backendName
ecHandle := appd.StartExitcall(btHandle, backendName)

...

// optionally store the handle in the global registry
appd.StoreExitcall(ecHandle, my_ec_guid)

...
// retrieve a stored handle from the global registry
myEcHandle := appd.GetExitcall(my_ec_guid)

// set the exit call details
if err := appd.SetExitcallDetails(myEcHandle, "Exitcall Detail String"); err != nil {
        log.Print(err)
}

// add an error to the exit call
appd.AddExitcallError(myEcHandle, appd.APPD_LEVEL_ERROR, "exitcall error!", true)

// end the exit call
appd.EndExitcall(myEcHandle)
```

Correlate with Other Business Transactions

A correlation header contains the information that enables the agents to continue the flow of a Business Transaction across multiple tiers.

An SDK agent can correlate with other SDK agents as well as other AppDynamics agents, such as Java, .NET, or PHP, that perform automatic correlation for certain types of entry and exit points.

Correlate with an Upstream Tier

When your instrumented process receives a continuing transaction from an upstream agent that supports automatic correlation:

1. Using a third-party `Header.Get()` function, extract the header named `APPD_CORRELATION_HEADER_NAME` from the incoming HTTP payload.
2. Pass the header to `StartBT()`. For example:

```
hdr := req.Header.Get(appd.APPD_CORRELATION_HEADER_NAME)
bt := appd.StartBT("Fraud Detection", hdr)
```

If the header retrieved by the `Header.Get()` function is valid, the business transaction started by the `StartBT()` call will be a continuation of the Business Transaction started by the upstream service.

Correlate with a Downstream Tier

The downstream agent is watching for a correlation header named `singularityheader` in the HTTP payload.

If your SDK agent is making an exit call to a downstream agent that supports automatic correlation:

1. Set the name of the correlation header using `APPD_CORRELATION_HEADER_NAME`.
2. Begin the exit call.
3. Retrieve the correlation header from the exit call using the `GetExitcallCorrelationHeader()` function.
4. Using third-party HTTP functions, prepare an HTTP POST request.
5. Inject a header named `APPD_CORRELATION_HEADER_NAME` with the value of the correlation header retrieved in step 3 into the outgoing payload of the HTTP request.
6. Make the request. For example:

```
inventoryEcHandle := appd.StartExitcall(btHandle, "Inventory DB")
hdr := appd.GetExitcallCorrelationHeader(inventoryEcHandle)

client := &http.Client{
        CheckRedirect: redirectPolicyFunc,
}

req, err := http.NewRequest("POST", "https://inventory/holds/xyz", nil)
// ...
req.Header.Add(appd.APPD_CORRELATION_HEADER_NAME, hdr)
resp, err := client.Do(req)
// ...

...
```

Terminate the Agent

Just before the application exits, terminate the agent.

```
appd.TerminateSDK()
```