Oracle® GlassFish Server 3.0.1 Monitoring Scripting Client Installation and Quick Start Guide

Part No: 821–1765–11
July 2010

100720@24378

Contents

CHAPTER 1

Oracle GlassFish Server Monitoring Scripting Client Installation and Quick Start Guide

Monitoring is the process of reviewing the statistics of a system to improve performance or solve problems. By monitoring the state of components and services that are deployed in Oracle GlassFish Server, system administrators can identify performance bottlenecks, predict failures, perform root cause analysis, and ensure that everything is functioning as expected. Monitoring data can also be useful in performance tuning and capacity planning.

Oracle GlassFish Server 3.0.1 Monitoring Scripting Client Installation and Quick Start Guide explains how to obtain Monitoring Scripting Client and how to use Monitoring Scripting Client to write clients in the JavaScript programming language to provide monitoring data about Oracle GlassFish Server. The ability to program in the JavaScript language is assumed.

The following topics are addressed here:

- "Obtaining Monitoring Scripting Client" on page 5
- "Configuring Monitoring Scripting Client" on page 6
- "Running a Script for Monitoring GlassFish Server" on page 7
- "Writing Scripts in the JavaScript Language for Monitoring GlassFish Server" on page 7
- "Sample JavaScript Programs for Monitoring GlassFish Server" on page 15

Obtaining Monitoring Scripting Client

If GlassFish Server was installed from Oracle GlassFish Server, ignore this section. Monitoring Scripting Client is integrated with Oracle GlassFish Server and is installed when Oracle GlassFish Server is installed.

If you are using GlassFish Server Open Source Edition, you can obtain this feature by purchasing a right-to-use and upgrading to Oracle GlassFish Server. For more information, see "Upgrading to Oracle GlassFish Server From GlassFish Server Open Source Edition" in *Oracle GlassFish Server 3.0.1 Administration Guide*.

Configuring Monitoring Scripting Client

Before using Monitoring Scripting Client, you must configure it. Configuration of Monitoring Scripting Client is required regardless of whether Oracle GlassFish Server was initially installed or acquired through an upgrade.

Configuring Monitoring Scripting Client involves enabling Comet support for an HTTP listener of GlassFish Server and deploying the Monitoring Scripting Client web application.

▼ To Configure Monitoring Scripting Client

1 Start or restart an administrative domain.

- **If no administrative domain is running, start a domain.**

  `asadmin start-domain` *domain*

- **If an administrative domain is running, restart the domain.**

  `asadmin restart-domain` *domain*

domain is the name of the administrative domain to start or restart.

2 Enable Comet support for an HTTP listener of GlassFish Server.

`asadmin set configs.config.server-config.network-config.`
`protocols.protocol.`*listener-name***`.http.comet-support-enabled=true`**

listener-name is the name of the HTTP listener for which to enable Comet support.

For example, to enable Comet support for the HTTP listener `http-listener-1`, type:

```
asadmin set configs.config.server-config.network-config.
protocols.protocol.http-listener-1.http.comet-support-enabled=true
```

3 Deploy the application in the `monitoring-scripting-client.war` file.

`asadmin deploy` *as-install***`/monitoring-scripting-client/war/monitoring-scripting-client.war`**

as-install is the directory in which GlassFish Server is installed.

Running a Script for Monitoring GlassFish Server

Monitoring Scripting Client provides an `asadmin` subcommand to run scripts for monitoring GlassFish Server. To ensure that scripts can receive and process events correctly, you must use the subcommand that is provided to run these scripts.

▼ To Run a Script for Monitoring GlassFish Server

1 **Ensure that the server is running.**

Remote subcommands require a running server.

2 **Ensure that monitoring is enabled for GlassFish Server.**

If monitoring for GlassFish Server is disabled, no events that your script is listening for are sent.

For information about how to enable monitoring for GlassFish Server, see "To Enable Monitoring" in *Oracle GlassFish Server 3.0.1 Administration Guide*.

3 **Run the `run-script` subcommand.**

Example 1–1 Running a Script for Monitoring GlassFish Server

This example runs the script `/tools/mon/modulestarted.js`.

```
asadmin> run-script /tools/mon/modulestarted.js
```

See Also You can also view the full syntax and options of the subcommand by typing `asadmin help run-script` at the command line.

Writing Scripts in the JavaScript Language for Monitoring GlassFish Server

Monitoring Scripting Client enables you to write clients in the JavaScript programming language to provide monitoring data about Oracle GlassFish Server.

The following topics are addressed here:

- "Obtaining Information About Events That Provide Monitoring Data" on page 8
- "To Register a Script as a Listener for an Event" on page 10
- "To Display Information From a Script" on page 11
- "Writing an Event Callback Function" on page 12

Obtaining Information About Events That Provide Monitoring Data

Components and services that are deployed in the GlassFish Server typically generate statistics that the GlassFish Server can gather at run time. To provide statistics to GlassFish Server, components define events for the operations that generate these statistics. At runtime, components send these events when performing the operations for which the events are defined. For example, to enable the number of received requests to be monitored, a component sends a "request received" event each time that the component receives a request.

Monitoring Scripting Client enables you to list all events that are provided for monitoring GlassFish Server. Detailed information about each of these events is provided to enable you to identify which events provide the statistics that you want to monitor.

Use this information to process appropriately the events of interest in JavaScript programs that you write for monitoring GlassFish Server.

▼ To Obtain a List of Events That Provide Monitoring Data

1. **Ensure that the server is running.**

   Remote subcommands require a running server.

2. **Ensure that monitoring is enabled for GlassFish Server.**

   If monitoring for GlassFish Server is disabled, no events are listed.

   For information about how to enable monitoring for GlassFish Server, see "To Enable Monitoring" in *Oracle GlassFish Server 3.0.1 Administration Guide*.

3. **To include in the list events that are related to a container, ensure that the container is loaded.**

   Events that are related to a container are listed *only* if the container is loaded. For example, to list events that are related to the JRuby container, you must ensure that the JRuby container is loaded by deploying a JRuby application in GlassFish Server.

4. **Run the `list-probes` subcommand.**

   The signatures of all events for all installed components of GlassFish Server are displayed.

   An event signature consists of the event identifier (ID) followed in parentheses by a comma-separated list of the event's parameters. Each parameter is listed as its type followed by its name.

   For detailed information about the format of an event signature, see the help page for the `list-probes` subcommand.

Example 1–2 Listing All Events

This command lists all events for monitoring GlassFish Server. For better readability, some events that would listed by this example are not shown.

```
asadmin> list-probes
glassfish:jdbc:connection-pool:connectionRequestDequeuedEvent (java.lang.String
poolName)
glassfish:jca:connection-pool:connectionsFreedEvent (java.lang.String poolName,
int count)
glassfish:transaction:transaction-service:deactivated ()
glassfish:kernel:connections-keep-alive:incrementCountFlushesEvent (java.lang.String
listenerName)
glassfish:kernel:file-cache:countInfoMissEvent (java.lang.String fileCacheName)
glassfish:ejb:timers:timerRemovedEvent ()
glassfish:jdbc:connection-pool:decrementNumConnFreeEvent (java.lang.String poolName)

...
glassfish:kernel:thread-pool:threadAllocatedEvent (java.lang.String monitoringId,
java.lang.String threadPoolName, java.lang.String threadId)
glassfish:jca:connection-pool:connectionCreatedEvent (java.lang.String poolName)
glassfish:kernel:connection-queue:connectionAcceptedEvent (java.lang.String
listenerName, int connection)

Command list-probes executed successfully.
```

See Also You can also view the full syntax and options of the subcommand by typing `asadmin help list-probes` at the command line.

▼ To Obtain Detailed Information About an Event That Provides Monitoring Data

The following detailed information is available about events for monitoring GlassFish Server:

- The event's signature
- A description of the event, including an indication of what the event signifies and an explanation of what causes the event to be sent
- A description of each parameter in the event

1 **Ensure that the server is running.**

Remote subcommands require a running server.

2 **If necessary, obtain the event ID of the event for which you want detailed information.**

For details, see "To Obtain a List of Events That Provide Monitoring Data" on page 8.

3 **Specify the `--details` option of the `list-probes` subcommand and the ID of the event as the operand of the subcommand.**

Example 1–3 Displaying Detailed Information About an Event

This example displays detailed information about the `glassfish:web:web-module:webModuleStartedEvent` event.

```
asadmin list-probes --details glassfish:web:web-module:webModuleStartedEvent
```

Information similar to the following is displayed.

```
Events       glassfish:web:web-module:webModuleStartedEvent(5GFP)

NAME
     glassfish:web:web-module:webModuleStartedEvent - web  module
     started event

SYNOPSIS
     glassfish:web:web-module:webModuleStartedEvent(
     java.lang.String appName,
     java.lang.String hostName)

DESCRIPTION
     This event is sent whenever an application has been  started
     (for example, as part of its deployment).

PARAMETERS
     appName

         The name of the web application that has been started.

     hostName
         The name of the virtual server on which the  application
         has been deployed.

Java EE 6           Last change: 19 Nov 2009                   1


Command list-probes executed successfully.
```

▼ To Register a Script as a Listener for an Event

Registering a script as listener for an event enables the script to listen for the event and to receive callbacks from the Monitoring Scripting Client when the script receives the event. The script can then collect data from the event. Registering a script as listener for an event also specifies the event callback function that is to be called when the event is received. For information about writing an event callback function, see "Writing an Event Callback Function" on page 12.

1 **Create an array of the event parameters to pass to the event callback function.**

This array may contain any number of the event's parameters in any order.

2 **Invoke the `scriptContainer.registerListener` method.**

In the invocation of the `scriptContainer.registerListener` method, pass the following information as parameters to the method:

- The event ID of the event
- The array of event parameters that you created in the previous step
- The name of the event callback function that is to be called when the event is received

Example 1–4 Registering a Script as a Listener for an Event

This example registers a script as a listener for the event `glassfish:web:jsp:jspLoadedEvent`. When this event is received, the event parameter `hostName` is passed to the `jspLoaded()` event callback function. For clarity, the declaration of the event callback function `jspLoaded()` is also shown in this example.

```
function jspLoaded(hostName) {
    ...
}

params = java.lang.reflect.Array.newInstance(java.lang.String, 1);
params[0]="hostName";

scriptContainer.registerListener('glassfish:web:jsp:jspLoadedEvent',
    params, 'jspLoaded');
```

▼ To Display Information From a Script

To provide statistics to system administrators, a script must display information when the script is run. Monitoring Scripting Client provides a preinstantiated object that has a method for displaying information from scripts. You must use this method to display updated information on standard output on the client system where the script is run. You cannot use the standard printing mechanisms of the JavaScript language because they write information to the server log.

● **Invoke the `client.print` method.**

In the invocation of the `client.print` method, pass the text string to display as the parameter to the method.

Example 1–5 Displaying Information From a Script

This example displays a string similar to the following in standard output each time the function `jspLoaded()` is called.

```
js> jsp loaded event called on host = server and count = 1

var njspLoaded=0;

function jspLoaded(hostName) {
    njspLoaded = njspLoaded + 1;
    client.print( '\n js> jsp loaded event called on ' +
            'host = ' + hostName +
            ' and count = ' + njspLoaded);
}
...
```

Writing an Event Callback Function

An event callback function is a function in a script that Monitoring Scripting Client calls in response to an event.

In your event callback functions, provide code to generate statistics from the data in events. Typically, the following types of statistics can be generated from the data in events:

- **Counter statistics.** These types of statistics typically correspond to a single event. For example, to calculate the number of received requests, only one event is required, for example, a “request received” event. Every time that a “request received” event is sent, the number of received requests is increased by 1.
- **Timer statistics.** These types of statistics typically correspond to multiple events. For example, to calculate the time to process a request, two requests are required, for example, a “request received” event and a “request completed” event.

▼ To Generate Counter Statistics

Counter statistics typically correspond to a single event. For example, to calculate the number of received requests, only one event is required, for example, a “request received” event. Every time that a “request received” event is sent, the number of received requests is increased by 1.

1 **Declare and initialize a variable.**

2 **Increase or decrease the variable each time the appropriate event is received.**

Example 1–6 Generating a Counter Statistic

This example declares and initializes to zero the variable `njspLoaded`. Each time the callback function `jspLoaded()` is invoked, the value of this counter is increased by 1.

For the complete listing of the script from which this example is extracted, see Example 1–8.

```
var njspLoaded=0;

function jspLoaded(hostName) {
    njspLoaded = njspLoaded + 1;
    ...
}
...
```

▼ To Generate a Timer Statistic

Timer statistics typically correspond to multiple events. For example, to calculate the time to process a request, two events are required, for example, a “request received” event and a “request completed” event.

For operations that have a measurable duration, Monitoring Scripting Client provides pairs of events to indicate the start and the end of the operations. For example, to indicate the initiation and completion of an HTTP request that has been received by the web container, Monitoring Scripting Client provides the following pair of events:

- `glassfish:web:http-service:requestStartEvent`
- `glassfish:web:http-service:requestEndEvent`

Use pairs of events that indicate the start and end of an operation to generate a timer statistic.

1. **Write an event callback function to calculate the start time.**

2. **Ensure that the function to calculate the start time is called when the “operation started” event is received.**

   For details, see “To Register a Script as a Listener for an Event” on page 10.

3. **Write an event callback function to calculate the end time.**

4. **Ensure that the function to calculate the end time is called when the “operation ended” event is received.**

   For details, see “To Register a Script as a Listener for an Event” on page 10.

Example 1–7 Generating a Timer Statistic

This example uses the following events to measure the time to process web service requests:

- `glassfish:web:http-service:requestStartEvent`
- `glassfish:web:http-service:requestEndEvent`

The events for a single request are sent in the same thread of control. Therefore, the identity of the thread can be used as a key to associate the start event and the end event for the request.

For the complete listing of the script from which this example is extracted, see Example 1–9.

```
...
var startTime;
var object = new Object();
...

function requestStartEvent(appName,hostName,serverName,serverPort,contextPath,
    servletPath){

    ...
        startTime = (new Date()).getTime();
    //insert the request time in Map
        key = java.lang.Thread.currentThread().getId();
        object[key] = startTime;
        ...
}
scriptContainer.registerListener('glassfish:web:http-service:requestStartEvent',
    request_params , 'requestStartEvent');
...
function requestEndEvent(appName,hostName,serverName,serverPort,contextPath,
    servletPath,statusCode){

...
        key = java.lang.Thread.currentThread().getId();
        startTime = object[key];
        if (startTime == null)
            client.print("Error getting the startTime for thread = " + key);
        else
            delete[key];
        totalTime = (new Date()).getTime() - startTime;
        ...
}
scriptContainer.registerListener('glassfish:web:http-service:requestEndEvent',
    request1_params, 'requestEndEvent');
```

Sample JavaScript Programs for Monitoring GlassFish Server

The sample JavaScript programs in this section show how to use GlassFish Server events to generate and present statistics for system administrators who are monitoring GlassFish Server.

EXAMPLE 1–8 Counting the Number of Loaded JSP Technology Pages

This example uses the `glassfish:web:jsp:jspLoadedEvent` event to count the number of JavaServer Pages (JSP) technology pages that GlassFish Server has loaded.

```
var njspLoaded=0;

function jspLoaded(hostName) {
    njspLoaded = njspLoaded + 1;
    client.print( ’\n js> jsp loaded event called on ’ +
            ’host = ’ + hostName +
            ’ and count = ’ + njspLoaded);
}

params = java.lang.reflect.Array.newInstance(java.lang.String, 1);
params[0]="hostName";

scriptContainer.registerListener(’glassfish:web:jsp:jspLoadedEvent’,
    params, ’jspLoaded’);
```

This script can be run with a command similar to the following:

```
asadmin run-script jsp-loaded-count.js
```

Information similar to the following is displayed each time that GlassFish Server loads a JSP technology page:

```
 js> jsp loaded event called on host = server and count = 1
```

The script runs until a user types Ctrl-C to stop the script.

EXAMPLE 1–9 Measuring the Time to Process Web Service Requests

This example uses the following events to measure the time to process web service requests:

- `glassfish:web:http-service:requestStartEvent`
- `glassfish:web:http-service:requestEndEvent`

The script also displays the information that is contained in the parameters of these events.

```
// http request related probes

// glassfish:web:http-service:requestStartEvent requestStartEvent(
```

EXAMPLE 1–9 Measuring the Time to Process Web Service Requests *(Continued)*

```
// java.lang.String appName,
// java.lang.String hostName,
// java.lang.String serverName,
// int serverPort,
// java.lang.String contextPath,
// java.lang.String servletPath)


request_params = java.lang.reflect.Array.newInstance(java.lang.String, 6);
request_params[0]="appName";
request_params[1]="hostName";
request_params[2]="serverName";
request_params[3]="serverPort";
request_params[4]="contextPath";
request_params[5]="servletPath";

var startTime;
var object = new Object();
var nrequestStartEvent=0;

function requestStartEvent(appName,hostName,serverName,serverPort,contextPath,
    servletPath){

    nrequestStartEvent=nrequestStartEvent+1;
        startTime = (new Date()).getTime();
    //insert the request time in Map
        key = java.lang.Thread.currentThread().getId();
        object[key] = startTime;

        client.print(
            'Count: '+ nrequestStartEvent +'\n'+
            'Event: glassfish:web:http-service:requestStartEvent' +'\n'+
            'Application: '+appName+'\n'+
            'Host: ' + hostName +'\n'+
            'Server: ' + serverName +'\n'+
            'HTTP Port: ' + serverPort +'\n'+
            'Context Path: ' + contextPath +'\n'+
            'Servlet Path: ' + servletPath + '\n' +
            'Current Thread: ' + java.lang.Thread.currentThread().getId() +
        '\n\n');
}

scriptContainer.registerListener('glassfish:web:http-service:requestStartEvent',
    request_params , 'requestStartEvent');

// glassfish:web:http-service:requestEndEvent requestEndEvent(
```

EXAMPLE 1–9 Measuring the Time to Process Web Service Requests *(Continued)*

```
// java.lang.String appName,
// java.lang.String hostName,
// java.lang.String serverName,
// int server Port,
// java.lang.String contextPath,
// java.lang.String servletPath,
// int statusCode)

request1_params = java.lang.reflect.Array.newInstance(java.lang.String, 7);
request1_params[0]="appName";
request1_params[1]="hostName";
request1_params[2]="serverName";
request1_params[3]="serverPort";
request1_params[4]="contextPath";
request1_params[5]="servletPath";
request1_params[6]="statusCode";


var nrequestEndEvent=0;

function requestEndEvent(appName,hostName,serverName,serverPort,contextPath,
    servletPath,statusCode){

    nrequestEndEvent=nrequestEndEvent+1;
        key = java.lang.Thread.currentThread().getId();
        startTime = object[key];
        if (startTime == null)
            client.print("Error getting the startTime for thread = " + key);
        else
            delete[key];
        totalTime = (new Date()).getTime() - startTime;

        client.print(
            'Time Taken: ' + ((new Date()).getTime()-startTime) + ' ms\n' +
            'Count: '+nrequestEndEvent+'\n'+
            'Event: glassfish:web:http-service:requestEndEvent' +'\n'+
            'Application: '+appName+'\n'+
            'Host: ' + hostName +'\n'+
            'Server: ' + serverName +'\n'+
            'HTTP Port: ' + serverPort +'\n'+
            'Context Path: ' + contextPath +'\n'+
            'Servlet Path: ' + servletPath +'\n'+
            'Status Code: ' + statusCode + '\n' +
            'Current Thread: ' + java.lang.Thread.currentThread().getId() + '\n' +
         '\n\n');
}
```

EXAMPLE 1–9 Measuring the Time to Process Web Service Requests *(Continued)*

```
scriptContainer.registerListener('glassfish:web:http-service:requestEndEvent',
    request1_params, 'requestEndEvent');
```

This script can be run with a command similar to the following:

```
asadmin run-script web-service-request-timer.js
```

Information similar to the following is displayed each time that a web service request is initiated:

```
Count: 2
Event: glassfish:web:http-service:requestStartEvent
Application: __admingui
Host: __asadmin
Server: localhost
HTTP Port: 4848
Context Path:
Servlet Path: /common/commonTask.jsf
Current Thread: 98
```

Information similar to the following is displayed each time that a web service request is completed:

```
Time Taken: 1704 ms
Count: 2
Event: glassfish:web:http-service:requestEndEvent
Application: __admingui
Host: __asadmin
Server: localhost
HTTP Port: 4848
Context Path:
Servlet Path: /common/commonTask.jsf
Status Code: 200
Current Thread: 98
```

The script runs until a user types Ctrl-C to stop the script.

APPENDIX A

Monitoring Scripting Client API Reference

The Monitoring Scripting Client API is a set of preinstantiated objects that enable scripts to interact with the Monitoring Scripting Client framework.

The following topics are addressed here:

Object `client`

Method Summary

`void print(String` *string*`)`
Prints a string to the standard output on the system where the script is running.

Method Detail

print

```
void print(
    String string)
```

Prints a string to the standard output on the system where the script is running.

Parameters

string
The string to be printed.

Object `scriptContainer`

Method Summary

`void registerListener (String` *event-id*`, String[]` *params*`, String` *callback*`)`

Registers a script as a listener for a specific event.

Method Detail

`registerListener`

```
void registerListener (
    String event-id,
    String[] params,
    String callback)
```

Registers a script as a listener for a specific event.

Parameters

event-id
: The event identifier (ID) of the event for which the script is to listen.

params
: An array of the event parameters to pass to the event callback function that is called when the event is received.

callback
: The event callback function that is called when the event is received.