Resource Plugin Development Guide

Version: 3
Date: 11/19/10
Creator: vFabric Documentation Team

Table of Contents

1. Introduction to Plugin Development

*Topics marked with***relate to features available only in vFabric Hyperic.*

Plugins are the interface between Hyperic HQ and products on the network you want to manage. HQ can detect hundreds of products thanks to its standard plugins, but you can extend HQ's functionality to products (or parts of products) not yet covered by HQ by developing your own custom plugins. This page gives you an overview of plugins and points you to detailed instructions for specific plugin types.

All HQ bundled plugins are available in both the Enterprise and Open Source versions of HQ. The only difference is that, as reflected in the HQ feature list, plugins in the Enterprise version enable more management features.

For a soup-to-nuts lesson on writing your own custom plugins, please see our Section 2.3, “Plugins Tutorial”.

1.1. What Do Plugins Do? An Overview

Proper development and functioning of HQ plugins rely on a thorough understanding of the HQ inventory model and of the four HQ functions that plugins implement:

- *Auto-Discovery*: Plugins can implement HQ auto-discovery of servers and services (*not* platforms, which are discovered via the Product plugin). Custom plugins will usually just call on the existing, HQ-provided auto-inventory plugin.
- *Monitoring*: Plugins can implement HQ monitoring, defining and collecting metrics and configuring them for display in the UI. Measurement plugins implement monitoring.
- *Control*: Plugins can implement the HQ control feature, defining control actions used to control resources. Control plugins implement control.
- *Event Management*

In addition, the Section 2.8, “Product Plugin” provides the deployment entry point on both the Server and Agent. It defines the resource types and plugin implementations for measurement, control, and auto-discovery (auto-inventory).

You can use plugins discover, collect data from, and control resources; plugins cannot be used to change alerting, reporting, or similar, Server-side functionality.

1.1.1. How Plugins Use the HQ Inventory Model

Unable to render {include} Couldn't find a page to include called: Plugin_InvModel

1.1.2. What Role do the Server and Agent Play in Plugins

Plugins must be deployed on both the Server and Agent. The Server and Agent each play different roles vis-a-vis plugins:

The Agent does the work of gathering all the data from resources and generally communicating with the resource. Using the plugin, the Agent can:

- *Auto-discover a resource.* In the case of a server, say, the plugin can also auto-discover all the server's hosted services. (Learn more about auto-discovery.)
- *Gather metrics from a resource*. (Learn more about monitoring and measuring resource behavior.) See the XML descriptor syntax for defining and gathering metrics.
- *Control a resource.* (Learn more about controlling resources.)

The Server deals in meta-data, which is to say, it knows about:

- *Platform, server, and service resource types* and how the plugin's targeted resources map to the inventory model. See the XML descriptor syntax for specifying servers and services.
- *The configuration schema* for each resource type (the same data you see displayed on the "Resource Configuration" screen). See the XML descriptor syntax for specifying configuring properties.
- *Data Display* (Learn more about viewing metrics)
- *Resource-type-specific help text* that instructs users how to set up their environment to properly run the plugin. See the XML descriptor syntax for specifying help text.
- *Definitions of control actions*. (Learn more about controlling resources.)

*_back to top_*

1.2. Technical Overview

Hyperic HQ plugins are self-contained .jar or .xml files which are deployed on both the Server and every Agent that you want to run the plugin. Every plugin contains, at a minimum, an XML descriptor, which is either a standalone .xml file or embedded in the .jar file.

Learn more about the XML descriptor syntax.

1.2.1. Plugin Implementations

For the sake of discussion, we'll call measurement, control, etc. the "types" of plugins. They can be created for any kind of resource. Depending on the kind of resource and how it communicates and surfaces its data, however, you will write different "implementations" of those plugin types. The different implementations are:

- Script
- JMX
- SQL
- SNMP

- ...

1.2.2. Using Support Classes to Simplify Your Plugin

HQ provides a bunch of "support classes" (that is, plugins) that you can invoke in your own plugins to abstract and simplify its construction. HQ provides the following support classes:

Category	Support Classes	When You Would Invoke This Support Class
Scripting	qmail, Sendmail, Sybase	
SNMP	Squid, Cisco IOS	
JMX	JBoss, WLS, WAS, ActiveMQ, Jetty	
JDBC	MySQL, PostgreSQL, Oracle	To gather database system tables metrics
Win-Perf Counters	IIS, Exchange, DS, .NET	To gather metrics from an application that surfaces perf counters
SIGAR	System, Process, Netstat	To communicate with an operating system. SIGAR is HQ's proprietary OS-independent API.
Net Protocols	HTTP, FTP, SMTP, etc. (A full list can be found on the "Add New Platform Service" screen, in Service Type.)	To communicate with platform services that HQ already has built-in, but you might want to gather additional metrics from it
Vendor	Citrix, DB2, VMware	

*_back to top_*

1.3. How Do I Write a Plugin?

The interface with HQ plugins is simple. For example, the HQ-provided Measurement plugin has only one method (getValue); the Autoinventory plugin has only one method for each inventory level. The hard part of writing a plugin is figuring out:

- How do you get the data out of the managed resource?

- Where should this data "live" in the inventory model? At what inventory level (platform, server, or service)?

The easiest way to write your first plugin is to start with an existing plugin that does *almost* what you want and tweak it to fit your needs. You can start with these example plugins\[7] or with our Section 2.3, “Plugins Tutorial”.

*_back to top_*

1.4. Testing Your Plugin

This page is replaced by Section 2.10, “Testing Plugins”.

_back to top_\

1.5. Deploying Your Plugin

You must deploy your custom plugin on the HQ Server and on all the Agents you want to run the plugin.

1.5.1. Deploying the Plugin in .xml or .jar Format

You can name your plugin as you please, but the name must end with either `-plugin.xml` or `-plugin.jar`, depending on the plugin format.

Plugin Format	**Description and Contents**
`-plugin.jar`	A standard jar-formatted file, containing `etc/hq-plugin.xml` — the plugin's XML descriptor — and any java class files, scripts, and MIB files.
`-plugin.xml`	For plugins that consist solely of an XML descriptor. It is combined with HQ's plugin support classes (either in the hq-pdk.jar or other *-plugin.jar plugins). In the case of a script plugin, the script can also be included in a separate file (as opposed to embedded in the XML descriptor).

1.5.2. Where to Place Custom-Plugin Files

Usually you will place your custom-plugin files in a subdirectory of the Server's and Agent's parent directories. By doing this, your plugin files will persist through upgrades. The subdirectory is called `hq-plugins`. For example:

Non-Windows Installation

- If the Server is installed at `/usr/local/hyperic/<Server directory>`, then add the plugin in `/usr/local/hyperic/hq-plugins`
- If the Agent is installed at `/usr/local/hyperic/<Agent directory>`, then add the plugin in `/usr/local/hyperic/hq-plugins`

Windows Installation

- If the Server is installed at `C:\Program Files\Hyperic HQ\<Server directory>`, then add the plugin in `C:\Program Files\Hyperic HQ\hq-plugins`
- If the Agent is installed at `C:\Program Files\Hyperic HQ\<Agent directory>`, then add the plugin in `C:\Program Files\Hyperic HQ\hq-plugins`

hq-plugins/ must exist prior to starting the Server

Hot-deployment on the Server side using the custom `hq-plugins/` directory requires that the directory exist prior to starting the Server. So, after creating `/usr/local/hyperic/hq-plugins/`, you must restart the Server.

Alternate File Placement

You can also place the files under the Server and Agent directories:

- On the Server: `<Server directory>/hq-engine/hq-server/webapps/ROOT/WEB_INF/hq-plugins/`
- On the Agent: `<Agent directory>/bundles/agent-x.y.z/pdk/plugins/`

You should do this when, for example, you are installing a patch to a plugin, or if you want to keep all your Agent-deployment files together.

1.5.3. Hot Deployment of Plugins

The Server supports hot-deployment: plugins can be updated or added without restarting the Server.

The Agent does *not* support hot-deployment: it must be restarted after a plugin is updated or added.

1.5.4. Invoking Plugins Outside the Server or Agent

When developing plugins, it's helpful to be able to invoke plugin methods directly for quick testing. The Invoking Plugins Standalone document describes this process.

*_back to top_*

1.6. Plugin Examples

- Protocol Monitor Plugin
- Process Monitor Plugin
- Section 2.1, “Script Plugin”
- qmail Plugin (combines Protocol, Process and Script)
- Section 2.2, “SQL Query Plugin”
- Windows Plugin
- Section 2.4, “JMX Plugin”
- JBoss Custom MBean Plugin

*_back to top_*

Related Information

SIGAR

PTQL

1.6.1. Return to Development Resources .

2. Introductory Topics on Resource Plugins

2.1. Script Plugin

*Topics marked with***relate to features available only in vFabric Hyperic.*

This page explains everything at your disposal when crafting your own custom script plugin. Please also consult the Section 2.5, “Script Plugin Tutorial” to be guided through writing a relatively simple custom script measurement plugin.

2.1.1. HQ Script Service

The **Script** service available underneath all platform resources can be used to invoke Nagios style plugins in the language of your choice.

Each instance of a **Script** service provides the following metrics:

- **Availability** - Uses process exit code:
- 0 == ✓ OK
- 1 == ⚠ Warning
- 2 == ❗ Critical
- 3 == ❓ Unknown
- 4 == ⏸ Paused

- **Execution Time** - Real time (in milliseconds) taken for each execution

- **Result Value** - First number (if any) seen in the output stream, example:

"Number of Transactions: 234"

Script services are configured using the following properties:

Name	**Description**	**Required**	**Example**	**Notes**
prefix	Space delimited prefix argument(s)	No	sudo	
path	Path to the script or program	Yes	/usr/local/nagios/libexec/check_http	HQ will check that the path exists
arguments	Space delimited script arguments	No	-H 209.237.227.36	Use quotes for arguments with spaces
timeout	Timeout in seconds	Yes	120	Execution is aborted if > timeout

To create a **Script** service, click on the "New Platform Service" link from the platform view, select **Script** from the drop-down and enter a name of your choice. Edit the Configuration Properties to configure the fields listed above.

Another option is to click the "New Server" link and choose **Nagios** from the drop-down. You can optionally import an existing nagios configuration file by checking the **Auto-Discover Plugins** check box or create them by clicking the "New Service" link and selecting **Nagios Plugin** from the drop-down. The **Nagios Plugin** service type has the same functionality as the platform **Script** service. Note that HQ has several built-in services for network protocols which do not require any Nagios plugins to use.

2.1.2. Script-Based Plugins

While the **Script** service can be useful, the metrics are very limited. The Script concept can be expanded to include any number of metrics by implementing an XML plugin.

A script plugin includes an XML descriptor and at least one script, which can be defined externally or embedded in the XML file. The plugin can use multiple scripts or a single script; the script must output *key=value* pairs. These pairs are parsed by the script executor and are collected as metrics using the keys within the metric templates. The scripts can be written in any language. You can even use Nagios plugins out of the box.

Learn more about the XML descriptor.

As an example, we will create an *I/O Device* service which uses an iostat script wrapper to format the data. Most Linux admins are familar with the **iostat** command which reports CPU and I/O stats for devices and partitions:

```
% iostat -d -x
Linux 2.6.9-22.EL (hammer)      06/23/2006

Device:    rrqm/s wrqm/s   r/s   w/s   rsec/s   wsec/s avgrq-sz avgqu-sz   await  svctm
 %util
hdc          0.00   0.00  0.00  0.00     0.00     0.00    96.94     0.00  129.82 113.47
 0.00
sda          0.02   0.59  0.07  0.54     2.00     9.06    17.95     0.00    4.21   1.75
 0.11
```

Each device (*hdc*, *sda*) will be an instance of the *I/O Device* service, so we want the wrapper script to only collect metrics for a given device:

```
% iostat -d -x sda

Linux 2.6.9-22.EL (hammer)      06/23/2006

Device:    rrqm/s wrqm/s   r/s   w/s   rsec/s   wsec/s avgrq-sz avgqu-sz   await  svctm
 %util
sda          0.02   0.59  0.07  0.54     2.00     9.06    17.95     0.00    4.21   1.75
 0.11
```

The wrapper script will invoke **iostat** with the arguments given above and parse the tabular output into key value pairs like so:

```
% ./pdk/scripts/device_iostat.pl sda
rrqm/s=0.02
wrqm/s=0.59
r/s=0.07
w/s=0.54
rsec/s=2.00
wsec/s=9.06
avgrq-sz=17.95
avgqu-sz=0.00
await=4.21
svctm=1.75
%util=0.11
```

The plugin defines configuration properties for the script path and the device name:

```
<config>
  <option name="script"
          description="Collector script"
          default="pdk/scripts/device_iostat.pl"/>
  <option name="device"
          description="Device name"
          default="sda"/>
</config>
```

The properties are then applied to a filter template:

```
<filter name="template"
        value="exec:file=%script%,args=%device%"/>
```

Where **exec** will route collection of the metric to the script executor plugin and the properties will be expanded to:

exec:file=pdk/scripts/device_iostat.pl,args=sda.

The filter is then used in each **metrictemplate** with the key it is to collect:

```
<metric name="Write Requests per Second"
        category="PERFORMANCE"
        indicator="true"
        template="${template}:w/s"/>
```

Which is expanded to:

```
template="exec:file=pdk/scripts/device_iostat.pl,args=sda:w/s"
```

Command Line Test

```
% java -jar pdk/lib/hq-pdk.jar -Dplugins.include=io-device -Ddevice=sda -t "I/O Device"
```

```
I/O Device Availability:
   I/O Device:exec:file=pdk/scripts/device_iostat.pl,args=sda:Availability
   =>100.0%<=

I/O Device Read Requests Merged per Second:
   I/O Device:exec:file=pdk/scripts/device_iostat.pl,args=sda:rrqm/s
   =>0.0<=

I/O Device Write Requests Merged per Second:
   I/O Device:exec:file=pdk/scripts/device_iostat.pl,args=sda:wrqm/s
   =>0.6<=

I/O Device Read Requests per Second:
   I/O Device:exec:file=pdk/scripts/device_iostat.pl,args=sda:r/s
   =>0.1<=

I/O Device Write Requests per Second:
   I/O Device:exec:file=pdk/scripts/device_iostat.pl,args=sda:w/s
   =>0.6<=

I/O Device Sectors Read per Second:
   I/O Device:exec:file=pdk/scripts/device_iostat.pl,args=sda:rsec/s
   =>2.0<=

I/O Device Sectors Writen per Second:
   I/O Device:exec:file=pdk/scripts/device_iostat.pl,args=sda:wsec/s
   =>9.1<=

I/O Device Average Sector Request Size:
   I/O Device:exec:file=pdk/scripts/device_iostat.pl,args=sda:avgrq-sz
   =>18.0<=

I/O Device Average Queue Length:
   I/O Device:exec:file=pdk/scripts/device_iostat.pl,args=sda:avgqu-sz
   =>0.0<=

I/O Device Average Wait Time:
   I/O Device:exec:file=pdk/scripts/device_iostat.pl,args=sda:await
   =>0.004s<=

I/O Device Average Service Time:
   I/O Device:exec:file=pdk/scripts/device_iostat.pl,args=sda:svctm
   =>0.001s<=

I/O Device CPU Usage:
   I/O Device:exec:file=pdk/scripts/device_iostat.pl,args=sda:%util
   =>0.1%<=
```

Create an I/O Device service

After you have deployed the iodevice plugin, instances of the service must be created manually. Using the HQ GUI:

- Navigate to the platform view of choice and click the "New Platform Service" link
- Enter a **Name** of your choice
- Select **I/O Device** from the drop-down list
- Click the **OK** button
- From the Inventory tab, click the **Edit** button

- Change any properties if needed
- Click the **OK** button

Your service and now configured and metric data will be viewable from the **Monitor** tab.

I/O Device plugin sources

- The io-device-plugin.xml HQ plugin descriptor (Download)
- The iostat wrapper script (Download)

2.1.3. How often are my scripts executed?

The script collector caches the results to avoid executing the script for each individual metric. The cache key is properties of the metric template.

In the iostat example, where the template is:

```
exec:file=pdk/scripts/device_iostat.pl,args=sda:Availability
```

The properties/cache-key would be:

```
file=pdk/scripts/device_iostat.pl,args=sda
```

If you wanted the script to collect data for all resources in a single round, you could just change the template like so:

```
exec:file=pdk/scripts/device_iostat.pl:sda_Availability
```

Which makes the properties/cache-key the same for all resources:

```
file=pdk/scripts/device_iostat.pl
```

The io-device-plugin.xml would change to:

```
<filter name="template"
             value="exec:file=%script%:%device%"/>

     <metric name="Availability"
             template="${template}:Availability"
             indicator="true"/>
```

And **device_iostat.pl** would just format the output keys with device name in the key:

```
print "${device}_$labels[$i]=$values[$i]\n";
```

The lifetime of the cache is defined by the metric intervals, whose defaults are defined by the plugin and can be changed later per-resource or globally per-type in the UI. So, if your metric intervals were configured to collect every 5 minutes, the script would only be run once every 5 minutes regardless of how many resources the script output applies to.

2.2. SQL Query Plugin

The majority of HQ's built-in database plugins use SQL queries to collect the metrics provided by each server and service type. The support classes for collecting these metrics can be used to create custom plugins that collect metrics using SQL queries defined by the plugin author.

- Connection configuration [13]
- SQL plugin example [14]
- Testing the plugin at the command line [16]
- HQ inventory plugin sources [16]

2.2.1. Connection Configuration

All SQL query plugins require configuration properties for connecting to the database. These properties are:

- Database driver
- JDBC URL
- Username
- Password (optional in some cases)

Supported drivers and corresponding JDBC URL syntax:

Driver	**JDBC URL Syntax**
Oracle	jdbc:oracle:thin:@localhost:1521:TEST
MySQL	jdbc:mysql://localhost/test
PostgreSQL	jdbc:postgresql://localhost:5432/test
MS SQL Server	jdbc:microsoft:sqlserver://localhost:1433;databasename=test

The SQL configuration schema can be imported by any plugin using the following snippet:

```
<config include="sql"/>
```

This config is defined by the sqlquery plugin as follows:

```
<config>
  <option name="jdbcDriver" type="enum"
          description="JDBC Driver Class Name">

    <include name="org.postgresql.Driver"/>
    <include name="oracle.jdbc.driver.OracleDriver"/>
    <include name="com.microsoft.jdbc.sqlserver.SQLServerDriver"/>
    <include name="com.mysql.jdbc.Driver"/>
  </option>

  <option name="jdbcUrl"
          description="JDBC Connection URL"/>
          default="jdbc:postgresql://localhost:9432/hqdb"/>
  <option name="jdbcUser"
          description="JDBC User"/>


  <option name="jdbcPassword"
          type="secret"
          optional="true"
          description="JDBC Password"/>
</config>
```

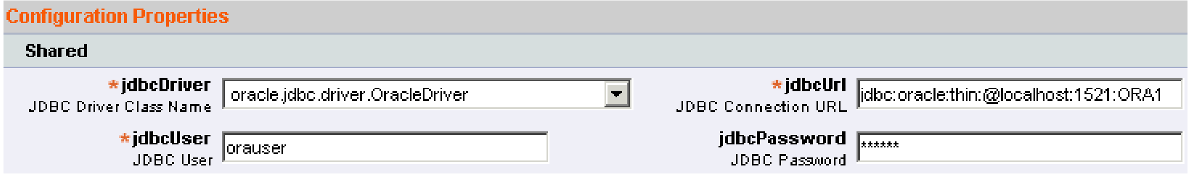
Your custom plugin is free to define this config schema rather than including it, should you wish to change the descriptions or defaults.

Only the **name** attributes must remain the same for the SQL Query plugin to create the database connection.

A custom plugin can also override the default values using the **property** tag and any of the **name** attributes defined in the config schema:

```
<property name="jdbcUrl"
    value="jdbc:oracle:thin:@localhost:1521:ORA1"/>

<property name="jdbcDriver"
          value="oracle.jdbc.driver.OracleDriver"/>

<property name="jdbcUser"
          value="orauser"/>

<property name="jdbcPassword"
          value="mypass"/>

<config include="sql"/>
```

Changing the default configuration properties to:



Configuration Properties

Shared

*jdbcDriver JDBC Driver Class Name	oracle.jdbc.driver.OracleDriver	*jdbcUrl JDBC Connection URL	jdbc:oracle:thin:@localhost:1521:ORA1
*jdbcUser JDBC User	orauser	jdbcPassword JDBC Password	******

Overriding the defaults will make it easier to configure your custom service and in the future could allow for auto-discovery.

back to top

2.2.2. SQL Plugin Example

For the example, we'll use a plugin included in the HQ agent distribution which you'll find at: **pdk/examples/hq-inventory-plugin.xml**.

This plugin uses simple SQL queries to collect Availability and counters for the number of Platforms, Servers and Services currently in the server's inventory. First, the plugin defines the service type:

```
<plugin>

  <service name="HQ Inventory">
  ...
  </service>
</plugin>
```

Next, the required SQL configuration schema:

```
<config include="sql"/>
```

A **filter** is used to define the metric **template**:

```
<filter name="template" value="sql:${query}:${name}"/>
```

- **sql** - Metric domain routes collection to the SQL query plugin
- **${query}** - The query to execute, defined by each **metric** tag
- **${name}** - The name of the metric being collected, defined by each **metric** tag

Another **filter** is used to for part of the syntax common to each query of this plugin:

```
<filter name="count" value="SELECT COUNT(*) FROM"/>
```

Now the plugin can start defining queries to be collected. The **Availability** metric is a special case where we want **1** to be returned as the value,

appending *WHERE 1=1* to accomplish that:

```
<metric name="Availability"
        query="${count} EAM_CONFIG_PROPS WHERE 1=1"
        indicator="true"/>
```

The rest of the metrics use queries that just count the number of rows in each table:

```
<metric name="Number of Platforms"
        query="${count} EAM_PLATFORM"
        indicator="true"/>

<metric name="Number of Servers"
        query="${count} EAM_SERVER"
        indicator="true"/>

<metric name="Number of Services"
        query="${count} EAM_SERVICE"
        indicator="true"/>
```

back to top

2.2.3. Testing the SQL Plugin at the Command Line

```
% java -jar pdk/lib/hq-pdk.jar -Dplugins.include=hq-inventory -
DjdbcDriver=org.postgresql.Driver -DjdbcUrl=jdbc:postgresql://localhost:9432/hqdb -
DjdbcUser=hqadmin -DjdbcPassword=hqadmin -t "HQ Inventory"

HQ Inventory Availability:

   HQ Inventory:sql:SELECT COUNT(*) FROM EAM_CONFIG_PROPS WHERE
 1%3D1:Availability:jdbcDriver=org.postgresql.Driver,jdbcUrl=jdbc%3Apostgresql%3A//localhost
%3A9432/hqdb,jdbcUser=hqadmin,jdbcPassword=hqadmin

   =>100.0%<=

HQ Inventory Number of Platforms:

   HQ Inventory:sql:SELECT COUNT(*) FROM EAM_PLATFORM:Number of
 Platforms:jdbcDriver=org.postgresql.Driver,jdbcUrl=jdbc%3Apostgresql%3A//localhost%3A9432/
hqdb,jdbcUser=hqadmin,jdbcPassword=hqadmin

   =>17.0<=

HQ Inventory Number of Servers:

   HQ Inventory:sql:SELECT COUNT(*) FROM EAM_SERVER:Number of
 Servers:jdbcDriver=org.postgresql.Driver,jdbcUrl=jdbc%3Apostgresql%3A//localhost%3A9432/
hqdb,jdbcUser=hqadmin,jdbcPassword=hqadmin

   =>172.0<=

HQ Inventory Number of Services:

   HQ Inventory:sql:SELECT COUNT(*) FROM EAM_SERVICE:Number of
 Services:jdbcDriver=org.postgresql.Driver,jdbcUrl=jdbc%3Apostgresql%3A//localhost%3A9432/
hqdb,jdbcUser=hqadmin,jdbcPassword=hqadmin

   =>1,289.0<=
```

back to top

2.2.4. HQ Inventory Plugin Sources

- The hq-inventory-plugin.xml plugin (Download)

Deploy following these steps

back to top

2.3. Plugins Tutorial

Plugins are the interface between Hyperic HQ and products on the network you want to manage. HQ can detect hundreds of products thanks to its standard plugins, but if HQ dose not yet detect and manage products (or parts of products) you want it to, we encourage you to develop your own custom plugins.

Here we provide tutorials for a couple implementations of a Measurement plugin, with the hope that leading you through the construction of enough different plugins will enable you to understand the plugin gestalt and then to write your own.

Consult the Plugin Development Center for a full and detailed explanation of all things plugin.

- Tutorials by plugin type [17]
- How plugins and the HQ inventory model play nicely together [17]
- Every plugin needs an XML descriptor [17]
- Troubleshooting
- Other sources of help [18]

2.3.1. Tutorials by Plugin Type and Implementation

This tutorial encompasses a variety of smaller tutorials, each dedicated to a specific type of plugin.

Plugin Type	Implementation	When You Should Write a Plugin of This Type and Implementation	Tutorial
Measurement	Script	You want to discover and collect metrics from a script	Section 2.5, "Script Plug-in Tutorial"
Measurement	JMX	You want to discover and collect metrics from a JMX-enabled application	Section 2.6, "JMX Plugin Tutorial"

back to top

2.3.2. How Plugins and the HQ Inventory Model Play Nicely Together

Unable to render {include} Couldn't find a page to include called: Plugin_InvModel

back to top

2.3.3. Every Plugin Needs an XML Descriptor

For every plugin, you must write an XML descriptor. In many cases, that is all you must write, as the plugin implementation has been templatized.

Learn more about the Plugin XML Descriptor and its components.

XML Descriptor for Measurement Plugins

Auto-Discovery

Every Measurement plugin must implement its own auto-discovery (called "autoinventory" within plugins) so that, once the plugin is deployed, HQ can actually find the product for which the plugin is being written and inventory it.

In almost all cases, you will implement auto-discovery in your plugins for servers or services, not platforms.

Availability Metric

The Availability metric indicates whether a Resource is up or down.

A metrics-gathering plugin must determine Availability for *every* server and *every* service it monitors. A single plugin will likely gather Availability for multiple Resources. If Availability is not gathered for a Resource, HQ will consider the Resource to be unavailable, and will not show any metrics for it in the Portal.

A plugin sets the value of Availability to 1 if the Resource is up, and 0 if it is down. These values are displayed in the Portal as "available" or "not available".

Verifying the existence of a Resource's process is a common technique for determining its Availability. However, the method a plugin uses to determine Availability can vary depending on the Resource Type and the plugin developer's judgment. There might be alternative techniques for determining the Availability of a Resource. For instance, a plugin might determine the Availability of a web server based on whether its process is up, its port is listening, it is responsive to a request, or by some combination of these conditions.

back to top

2.3.4. Troubleshooting

This section describes several things you can do to figure out problems with your plugin.

- Are you having problems with the HQ Server after the plugin is deployed? Turn on DEBUG mode on the Agent.
- What user is running the Agent? It should be the same user that is running the tests of the plugin.
- Do you have the proper permissions for everything (executing scripts, accessing files, etc.)?
- Are your environment variables set up correctly?

Testing the Plugin

This page is replaced by Section 2.10, “Testing Plugins”.

back to top

2.3.5. Other Sources of Help

If this tutorial leaves you with questions about how to write your plugin, you can find additional help in the following places:

- Plugin Development Center
- HQ forums
- Contact HQ Support

back to top

Related Information

Plugin Development Center: All things plugin

Overview of measurement plugins

Overview of JMX plugins

Overview of script plugins

Learn more about the plugin XML descriptor's syntax

2.4. JMX Plugin

Topics marked with*relate to features available only in vFabric Hyperic.

JMX 1.2 provides a standard remote API to manage remote JMX-enabled applications. Prior to JMX 1.2/JSR-160 it was up to the server vendor to implement a remote MBeanServer connector. Several Open Source projects use the MX4J implementation, others such as WebSphere, WebLogic and JBoss rolled their own. Because of this HQ plugins had required connector code specific to each product and still do for those products which do not support JSR-160. With the adoption of JMX 1.2 in newer versions of products (ActiveMQ 4.0, Geronimo 1.0, WebLogic 9.1, Resin 3.0, etc) and the built-in support with J2SE 1.5, HQ is able to provide JMX plugin support classes without having to use a vendor specific connector.

This page explains everything at your disposal when crafting your own custom JMX plugin. Please also consult the Section 2.6, “JMX Plugin Tutorial” to be guided through writing a relatively simple custom JMX measurement plugin.

- How JMX plugins implement HQ features
- Configuring JMX-enabled applications for remote connections [20]
- Finding the right MBeans [20]
- Getting started with JMX in HQ [21]
- Creating custom JMX plugins [22]

2.4.1. How JMX Plugins Implement HQ Features

JMX features map well to HQ features. For the most part, this implementation has been templatized so all you will need to do is invoke the appropriate bundled (HQ-provided) plugin (for example, the `autoinventory` the `measurement` plugins):

HQ Feature	How a JMX Plugin Implements It	Learn More
Server auto-discovery	SIGAR PTQL, File scanning	Auto-discovery [25]
Service auto-discovery	`MBeanServer.queryMBeans()`	Auto-discovery [25]
Monitoring	`MBeanServer.getAttribute()`, SIGAR	Gathering metrics [23]
Control	`MBeanServer.invoke()`, `MBeanServer.setAttribute()`, scripts, SCM	Control [24]
Event Management	`MBeanServer.handleNotification()`, log4j files	Log and config tracking

back to top

2.4.2. Configuring JMX-Enabled Applications for Remote Connections

The JMX-enabled application to be monitored must configured to accept remote connections. In many cases, the remote connector enabled by default, otherwise, you must configure it for remote access.

- J2SE 1.5
- MX4J
- ActiveMQ
- JOnAS
- ServiceMix
- Many more...

back to top

2.4.3. Finding the Right MBeans

There are several tools available for browsing MBeans in a remote application which can be useful for HQ plugin development:

- JConsole
- MC4J

The HQ JMX support classes also provide a command line tool to dump MBeans in text format:

```
java -Duser=system -Dpass=manager -Dplugins.include=jmx -jar pdk/lib/hq-pdk.jar \
jmx MBeanDumper service:jmx:rmi:///jndi/rmi://localhost:1099/jmxrmi
```

Example output snippet:

```
...

MBean: org.apache.commons.modeler.BaseModelMBean
Name:  Catalina:type=StringCache
        0. Attribute: modelerType = org.apache.tomcat.util.buf.StringCache (rw)
        1. Attribute: trainThreshold = 20000 (rw)
        2. Attribute: byteEnabled = true (rw)
        3. Attribute: hitCount = 0 (r)
        4. Attribute: accessCount = 0 (r)
        5. Attribute: charEnabled = false (rw)
        6. Attribute: cacheSize = 200 (rw)

         Operation: void reset []

MBean: org.apache.commons.modeler.BaseModelMBean
Name:  Catalina:type=Cache,host=localhost,path=/jsp-examples
        0. Attribute: modelerType = org.apache.naming.resources.ResourceCache (rw)
        1. Attribute: accessCount = 20 (r)
        2. Attribute: cacheMaxSize = 10240 (rw)
        3. Attribute: hitsCount = 8 (r)
        4. Attribute: maxAllocateIterations = 20 (rw)
        5. Attribute: spareNotFoundEntries = 500 (rw)
        6. Attribute: cacheSize = 36 (r)
        7. Attribute: desiredEntryAccessRatio = 3 (rw)
         Operation: boolean unload [java.lang.String]

         Operation: void load [org.apache.naming.resources.CacheEntry]
         Operation: org.apache.naming.resources.CacheEntry lookup [java.lang.String]
...
```

Each of the attribute names given above (for example, modelerType or accessCount) can be used as the metric `alias` in the plugin when defining the metrics [23] to be collected.

back to top

2.4.4. Getting Started with JMX in HQ

The following server types are implemented using HQ's JSR-160 based support classes:

- Sun JVM 1.5
- ActiveMQ 4.0
- Geronimo 1.0
- Resin 3.0
- JOnAS 4.7

Other server types continue to use vendor specific connectors for backward compatibility. The **Sun JVM 1.5** type applies to any of the above and any other JMX-enable server running under a Sun 1.5 JVM but has its own set of metrics and control actions. Unlike the other server types, **Sun JVM 1.5** instances are not currently auto-discovered.All of the server types have the following configuration properties:

- jmx.url - The JMX Service URL
- jmx.username - Username if authentication is required
- jmx.password - Password if authentication is required

For example:

```
<config>
<option name="jmx.url" description="JMX URL to MBeanServer" default="service:jmx:rmi:///
jndi/rmi://localhost:6969/jmxrmi"/>
<option name="jmx.username" description="JMX username" optional="true" default=""/>
<option name="jmx.password" description="JMX password" optional="true" default=""
 type="secret"/>
```

Instead of specifying all three properties, you can use a shortcut:

```
<config include="jmx"/>
```

This includes login configuration properties from the JMX plugin.

back to top

2.4.5. Creating Custom JMX Plugins

A JMX plugin consists solely of an XML descriptor. This section explains the components you can include in the descriptor.

back to top

Defining Service Types to Provide Management via Custom MBeans

Each server type defines several service types such as EJBs, Connection Pools and JMS Queues. Custom plugins define additional service types to provide management via custom MBeans. The `service` element defines a service type, for example:

```
<service name="String Cache"
         server="Sun JVM"
         version="1.5">
</service>
```

The **server** attribute must be *Sun JVM* and **version** attribute must be *1.5*, or any of the other supported server+version combinations. The **name** attribute is the choice of the plugin implementor. These services will become part of the HQ inventory model, displayed along with the built-in **server** service types throughout the UI and shell. Service extensions will also inherit the server's configuration properties used to connect to the `MBeanServer`:

- jmx.url
- jmx.username
- jmx.password

Defining an ObjectName to Access Custom MBeans

In order to access custom MBeans, the plugin must define its JMX ObjectName to be used with various `MBeanServer` interface methods. Only one `ObjectName` is defined per-service type using the **property** tag within the **service** tag:

```
<property name="OBJECT_NAME"
```

```
          value="Catalina:type=StringCache"/>
```

back to list of plugin components *[22]*

Defining Configuration Properties to Be Displayed in the UI

All the configuration properties for a JMX plugin, as for all other plugins, are displayed in the "Resource Configuration" screen for the resource. The default values for each of these options can be specified in the plugin, but users can change the values on that screen.

In the example above, the `OBJECT_NAME` is hard-coded since there is only one instance of the String Cache. Configuration Properties are used to support multiple instances that follow the same `ObjectName` pattern. For example, the *WebApp Cache* plugin uses an `ObjectName` with the following pattern:

```
<property name="OBJECT_NAME"
          value="Catalina:type=Cache,host=*,path=*"/>
```

Where the `ObjectName`**Domain** is always *Catalina* and **type** attribute value is always *Cache*, but the **host** and **path** attributes will be different for each instance of the MBean. The *WebApp Cache* plugin defines config options for each of the instance properties:

```
<config>
  <option name="host"
          description="Host name"
          default="localhost"/>

  <option name="path"
          description="Path"
          default="/jsp-examples"/>

</config>
```

The values of the instance attributes within the `OBJECT_NAME` is replaced with the value of the configuration property when used by the plugin, for example:

```
"Catalina:type=Cache,host=localhost,path=/jsp-examples"
```

back to list of plugin components *[22]*

Defining and Gathering Metrics

Metrics are defined just as they are with other plugins, but in the case of custom MBean services the **OBJECT_NAME** property is used to compose the metric **template** attribute:

```
<metric name="Access Count"
        template="${OBJECT_NAME}:accessCount"
        category="THROUGHPUT"
        indicator="true"
        collectionType="trendsup"/>
```

This results in the **template** being expanded, for example, to:

```
template="Catalina:type=Cache,host=localhost,path=/jsp-examples:accessCount"
```

Where *accessCount* is an attribute of the MBean and can be collected internally using the `MBeanServer` interface like this:

```
ObjectName name = new ObjectName("Catalina:type=Cache,host=localhost,path=/jsp-examples");

return MBeanServer.getAttribute(name, "accessCount");
```

The MBean interface attributes collected by Section 2.4.8, “tomcat-webapp-cache-plugin.xml” as metrics are as follows:

```
public interface WebAppCacheMBean {
    public int getAccessCount();
    public int getHitCount();
    public int getCacheSize();
}
```

Learn more about measurement plugins.

back to list of plugin components *[22]*

Implementing Control Actions

With the **OBJECT_NAME** property defined, MBean operations can be exposed as HQ control actions simply by adding the list of method names:

```
<actions include="reset"/>
```

The plugin must also define the control implementation class (resides in *hq-jmx.jar*):

```
<plugin type="control"
        class="org.hyperic.hq.product.jmx.MxControlPlugin"/>
```

The control actions will then be invoked as MBean operations by the plugin like so:

```
ObjectName name = new ObjectName("Catalina:type=StringCache");

return MbeanServer.invoke(name, "reset", new Object[0], new String[0]);
```

Which maps to the following MBean operation:

```
public interface StringCacheMBean {

    public void reset();
}
```

If an MBean operation requires arguments, they can be passed in using the HQ control UI.

The *WebApp Cache* example provides the following control actions:

```
<actions include="unload,lookup,allocate"/>
```

Which map to the following MBean operations:

```
public interface WebAppCacheMBean {
    public boolean unload(String name);
    public CacheEntry lookup(String name);
    public boolean allocate(int value);
}
```

Learn more about control plugins.

back to list of plugin components *[22]*

Implementing Auto-Discovery

Auto-discovery (called "auto inventory" within plugins) is easily implemented by taking advantage of an HQ-provided `autoinventory` plugin.

To implement auto-discovery at the server level, you must invoke an autoinventory plugin with a specific class — `MxServerDetector` — within the server tag:

```
<server name="Java Server Name" version ="version #">
...

<plugin type="autoinventory" class="org.hyperic.hq.product.jmx.MxServerDetector"/>
...

</server>
```

In the case of service, auto-discovery is supported for custom MBean services, again driven by the `OBJECT_NAME` property. To implement auto-discovery at the service level, invoke the autoinventory plugin, leaving out the class attribute, within a service tag:

```
<service name="Java Service Name">
...

<plugin type="autoinventory"/>

...
</service>
```

The JMX plugin uses the `MBeanServer.queryNames` method to discover a service for each MBean instance. In the case where the `OBJECT_NAME` contains configuration properties, the properties will be auto-configured.

By default, auto-discovered service names will be composed using the hosting-server name, configuration properties, and service type name. For example:

```
"myhost Sun JVM 1.5 localhost /jsp-examples WebApp String Cache"
```

The naming can be overridden using the `AUTOINVENTORY_NAME` property:

```
<property name="AUTOINVENTORY_NAME"
             value="%platform.name% %path% Tomcat WebApp String Cache"/>
```

Configuration properties from the platform, hosting server, and the service itself can be used in the `%replacement%` strings, resulting in a name like so:

```
"myhost /jsp-examples Tomcat WebApp String Cache"
```

Learn more about auto-inventory plugins.

back to list of plugin components [22]

Discovering Custom Properties

Discovery of Custom Properties is supported, again using the **OBJECT_NAME** and `MBeanServer.getAttribute`. Simply define a **properties** tag with any number of **property** tags where the **name** attribute value is that of an MBean attribute:

```
<properties>
  <property name="cacheMaxSize"
            description="Maximum Cache Size"/>
</properties>
```

Which maps to the following MBean interface method:

```
public interface WebAppCacheMBean {
    public int getCacheMaxSize();
}
```

back to list of plugin components [22]

Implementing Log and Config Tracking

All log and config tracking data is displayed in the HQ UI on the "Current Health" screen.

Should your plugin wish to track log and/or config files, simply use the generic classes which are included in **pdk/lib/hq-pdk.jar** and available for use by all plugins. As you can see in the following code, these classes require that files be in Log4J format (which most will be).

```
<property name="DEFAULT_LOG_FILE"
          value="log/mybean.log"/>

<plugin type="log_track"
        class="org.hyperic.hq.product.Log4JLogTrackPlugin"/>

<property name="DEFAULT_CONFIG_FILE"
          value="conf/mybean-service.xml,conf/mybean.policy"/>

<plugin type="config_track"
        class="org.hyperic.hq.product.ConfigFileTrackPlugin"/>
```

Tracking an MBeanLog

You can also easily implement log tracking for a specific MBean. Invoke the `log_track` plugin with the class `MxNotificationPlugin` before declaring the metric for the desired MBean (in this example, Threading MBeans, which we'll just pretend were enumerated earlier).

```
<plugin type="log_track"
     class="org.hyperic.hq.product.jmx.MxNotificationPlugin"/>
```

```
<property name="OBJECT_NAME"
     value="java.lang:type=Threading"/>

<metrics
     include="Threading"/>
```

2.4.6. Example Custom MBean Plugins

- Section 2.4.7, “tomcat-string-cache-plugin.xml”
- Section 2.4.8, “tomcat-webapp-cache-plugin.xml”

2.4.7. tomcat-string-cache-plugin.xml

```
<plugin>
  <service name="String Cache"
           server="Sun JVM" version="1.5">

    <property name="OBJECT_NAME"
              value="Catalina:type=StringCache"/>

    <property name="AUTOINVENTORY_NAME"
              value="%platform.name% Tomcat String Cache"/>

    <plugin type="autoinventory"/>

    <plugin type="measurement"
            class="org.hyperic.hq.product.jmx.MxMeasurementPlugin"/>

    <plugin type="control"
            class="org.hyperic.hq.product.jmx.MxControlPlugin"/>

    <!-- reset is an MBean operation, set* are attribute setters -->
    <actions include="reset,setcacheSize,settrainThreshold"/>

    <properties>
      <property name="cacheSize" description="Cache Size"/>
      <property name="trainThreshold" description="TrainThreshold"/>
    </properties>

    <filter name="template"
            value="${OBJECT_NAME}:${alias}"/>

    <metric name="Availability"
            template="${OBJECT_NAME}:Availability"
            indicator="true"/>

    <metric name="Cache Hits"
            alias="hitCount"
            collectionType="trendsup"
            indicator="true"/>
  </service>
</plugin>
```

Download this plugin.

2.4.8. tomcat-webapp-cache-plugin.xml

```
<plugin>
```

```
  <service name="WebApp Cache"
           server="Sun JVM" version="1.5">

    <property name="OBJECT_NAME"
              value="Catalina:type=Cache,host=*,path=*"/>

    <property name="AUTOINVENTORY_NAME"
              value="%platform.name% %path% Tomcat WebApp Cache"/>

    <plugin type="autoinventory"/>

    <plugin type="measurement"
            class="org.hyperic.hq.product.jmx.MxMeasurementPlugin"/>

    <plugin type="control"
            class="org.hyperic.hq.product.jmx.MxControlPlugin"/>

    <!-- set* are attribute setters, the rest are MBean operations-->
    <actions include="setscacheMaxSize,unload,lookup,allocate"/>

    <config>
      <option name="host"
              description="Host name"
              default="localhost"/>

      <option name="path"
              description="Path"
              default="/jsp-examples"/>
    </config>

    <properties>
      <property name="cacheMaxSize" description="Maximum Cache Size"/>
    </properties>

    <filter name="template"
            value="${OBJECT_NAME}:${alias}"/>

    <metric name="Availability"
            template="${OBJECT_NAME}:Availability"
            indicator="true"/>

    <metric name="Access Count"
            alias="accessCount"
            collectionType="trendsup"
            indicator="true"/>

    <metric name="Hit Count"
            alias="hitsCount"
            collectionType="trendsup"
            indicator="true"/>

    <metric name="Size"
            alias="cacheSize"/>
  </service>
</plugin>
```

Download this plugin.

2.5. Script Plugin Tutorial

This tutorial will lead you through writing, testing, and deploying a simple script measurement plugin that auto-discovers and collects several metrics from Sendmail application and its processes. The Sendmail plugin is one of the simplest script measurement plugins in HQ. We hope that you will be able to extrapolate from this simple example to your own needs for discovering and monitoring a script service.

Learn more about script plugins and measurement plugins.

See the Section 2.3, “Plugins Tutorial” overview for general plugin information.

Procedural Overview for Writing a Script Measurement Plugin

1. Write a script. [29]

2. Review the sample script measurement plugin descriptor and copy it for your own use.

3. See what this plugin will show in the HQ UI [31].

4. Understand each component of the descriptor and tailor the XML descriptor [32] to reflect your own plugin.

5. Test the plugin. [41]

6. Deploy the plugin. [41]

2.5.1. How the Plugin Fits Into HQ's Inventory Model

This plugin fits into the HQ inventory model by starting at the (Sendmail) server level and organizing several (Sendmail) services under it. Other plugins targeting only a service could bypass the server and be organized directly under the platform on which it is run.

back to top

2.5.2. Step 1. Write a script.

This you can do on your own, but keep the following things in mind:

- The script must retrieve the metrics you want to see in HQ, and it must output them in a specific format: `name=value` pairs. These name-value pairs will also be the output of the plugin. For a simple example:

```
$ ${HQ_HOME}/plugins/scripts/script.pl
Availability=0
Metric=8192
```

  Example 1. Sample Script Output

- Place the script in `<Agent dir>/pdk/scripts/`, for reference from within the plugin's XML descriptor. You can also place the script directly in the XML descriptor.

- Note the name of your script.

back to top

2.5.3. Step 2. Review the Sample Script Measurement Plugin Descriptor

The sample script plugin XML descriptor below — Sendmail Script Meaurement Plugin — autodiscovers and collects several metrics from a Sendmail server and its hosted services. The sample provides the structure of the plugin, and you should start with it when writing your own plugin. You will need to change certain values — which we'll point out — within the descriptor.

```
<plugin>

<script name="hq-sendmail-stat">
<![CDATA[
#!/bin/sh
# linux
[ -d "/var/spool/mqueue" ] &&
    msgdir="/var/spool/mqueue" &&
    premsgdir="/var/spool/clientmqueue"
# solaris
[ -d "/usr/spool/mqueue" ] &&
    msgdir="/usr/spool/mqueue" &&
    premsgdir="/usr/spool/clientmqueue"
[ -z "$msgdir" ] &&
    exit 1
cd $msgdir
[ $? != 0 ] &&
    exit 1
messfiles=`find * -print 2>/dev/null | wc -w`
cd $premsgdir
[ $? != 0 ] &&
    exit 1
premessfiles=`find * -print 2>/dev/null | wc -w`
echo MessagesInQueue=$messfiles
echo MessagesAwaitingPreprocessing=$premessfiles
]]>
</script>

<server name="SendMail">

<config>
<option name="process.query" description="Process Query"
default="State.Name.eq=sendmail,CredName.User.eq=root"/>
<option name="exec"
description="Type &quot;sudo$quot; To Avoid Having Agent As Root"
default=""/>
</config>

<property name="HAS_BUILTIN_SERVICES" value="true"/>
<property name="PROC_QUERY" value="State.Name.eq=sendmail"/>
<plugin type="autoinventory" class="org.hyperic.hq.product.DaemonDetector"/>
<plugin type="measurement" class="org.hyperic.hq.product.MeasurementPlugin"/>

<metric name="Availability"
alias="Availability"
template="sigar:Type=ProcState,Arg=%process.query%:State"
category="AVAILABILITY" indicator="true" units="percentage" collectionType="dynamic"/>

<service name="SendMail Message Submission Process">

<config>
<option name="process.query"
default="State.Name.eq=sendmail,CredName.User.ne=root"
description="PTQL for SendMail Message Submission Process"/>
</config>

<metric name="Availability"
template="sigar:Type=ProcState,Arg=%process.query%:State" indicator="true"/>

</service>

<service name="SendMail Root Daemon Process">
```

```
<config>
<option name="process.query"
default="State.Name.eq=sendmail,CredName.User.eq=root"
description="PTQL for SendMail Root Daemon Process"/>
</config>

<metric name="Availability"
template="sigar:Type=ProcState,Arg=%process.query%:State"
indicator="true"/>
</service>

<filter name="template"
value="exec:timeout=10,file=pdk/work/scripts/sendmail/hq-sendmail-stat,exec=%exec%:
${alias}"/>

<metric name="Messages In Queue" indicator="true"/>

<metric name="Messages Awaiting Preprocessing" indicator="true"/>

<service name="SMTP">

<config>
<option name="hostname" description="SMTP Hostname" default="localhost"/>
</config>

<filter name="template" value="SMTP:hostname=%hostname%:${alias}"/>

<metric name="Availability" indicator="true"/>

<metric name="Inbound Connections" indicator="true"/>

<metric name="Outbound Connections" indicator="true"/>

</service>
</server>

<!--
 ==================== Plugin Help ===========================
-->
<help name="SendMail">
  <p>
  <h3>Configure HQ for monitoring SendMail</h3>
  </p>
  <p>
  This plugin needs sudo access as root in order to access the appropriate
  <br>
  SendMail dirs.
  <br>
  To configure sudo (in /etc/sudoers):
  <br>
  Cmnd_Alias HQ_SENDMAIL_STAT = &lt;hqdir&gt;/agent/pdk/work/scripts/sendmail/hq-sendmail-
stat
  <br>
  &lt;agentuser&gt; ALL = NOPASSWD: HQ_SENDMAIL_STAT
  </p>
  </help>
</plugin>
```

Example 2. Sendmail Script Measurement Plugin

back to top

2.5.4. Step 3. What This Plugin Will Show in the HQ UI

A plugin with the XML descriptor above will show in the HQ UI:

- The existence of the Sendmail server and the three component processes ("services") — SendMail Message Submission Process, SendMail Root Daemon Process, and SMTP — that it hosts
- For each of these four resources (one server and three services), its availability
- For the Sendmail server, these metrics:
  - Messages In Queue
  - Messages Awaiting Preprocessing
- For the SMTP service, these metrics:
  - Inbound Connections
  - Outbound Connections
- Configuration instructions for the Sendmail server, on the server's "Resource Configuration" screen

Learn more about viewing resources and metrics in HQ.

back to top

2.5.5. Step 4. Understand and Modify Each Piece of the Sample Plugin XML Descriptor

The sample XML descriptor above is divided into parts, each of which is repeated and explained for your particular use below. Some of the more important parts of the XML descriptor are called out.

Learn more about the plugin XML descriptor.

Include or Reference the Script

1. This is the actual script — `hq-sendmail-stat` — to be executed by the plugin.

```
<plugin>

<script name="hq-sendmail-stat">
#!/bin/sh
# linux
[ -d "/var/spool/mqueue" ] &&
    msgdir="/var/spool/mqueue" &&
    premsgdir="/var/spool/clientmqueue"
# solaris
[ -d "/usr/spool/mqueue" ] &&
    msgdir="/usr/spool/mqueue" &&
    premsgdir="/usr/spool/clientmqueue"
[ -z "$msgdir" ] &&
    exit 1
cd $msgdir
[ $? != 0 ] &&
    exit 1
messfiles=`find * -print 2>/dev/null | wc -w`
cd $premsgdir
[ $? != 0 ] &&
    exit 1
premessfiles=`find * -print 2>/dev/null | wc -w`
echo MessagesInQueue=$messfiles
echo MessagesAwaitingPreprocessing=$premessfiles
</script>
```

You can embed your script in the XML descriptor as is done here, or you can keep it in a separate file and reference it by putting this line before the config options, like so:

```
<plugin>

<server name="SendMail">
<filter name="script" value="pdk/scripts/hq-sendmail-stat.pl"/>
<config>
<option name="process.query"
...
```

where you would replace "hq-sendmail-stat.pl" with the name of your script. It must always be in the `pdk/scripts` directory, and HQ assumes that that is in the Agent directory.

2. This code identifies the server being monitored by the plugin.

```
<server name="SendMail">
```

This value will be displayed in the UI for the server, so while you *can* name it anything you want, we recommend naming it something meaningful. If you want to monitor only a service, change `server` to `service` and name the service.

Identify the Exact Processes You Want to Monitor

3. This code specifically identifies the exact process(es) of interest.

```
<config>
<option name="process.query" description="Process Query"
default="State.Name.eq=sendmail,CredName.User.eq=root"/>
```

`State.Name.eq=sendmail` makes sure the query returns only sendmail processes, and `CredName.User.eq=root` makes sure the query returns only those Sendmail processes with root access. In this case, we need root access to access the (Sendmail) server. The query is written in PTQL, which isn't necessary to understand for this tutorial, but would be useful to understand for future, custom plugins.
If you are interested in a different application or different Sendmail processes, replace `sendmail` with a different server name or `CredName.User.eq=root` with a different filter that will get you the desired set of processes.
Learn more about the config tags and option tags.

4. This code isn't strictly necessary for the operation of the plugin. It is used in conjunction with later lines to use sudo instead of root access in order to tidy the console display while the plugin is running.

```
<option name="exec"
description="Type &quot;sudo$quot; To Avoid Having Agent As Root"
default=""/>
</config>
```

You can remove it or keep as-is.

Auto-Discover the Server and Services

5. This code implements auto-discovery (called "autoinventory" in plugins).

```
<property name="HAS_BUILTIN_SERVICES" value="true"/>
<property name="PROC_QUERY" value="State.Name.eq=sendmail"/>
<plugin type="autoinventory" class="org.hyperic.hq.product.DaemonDetector"/>
```

   The first line tells HQ that the Sendmail server has component services, so please auto-discover them, too. Keep it as-is if your server contains services you want to discover and manage. The second line tells HQ to discovery if the Sendmail process is running or not. In a different plugin, to determine the existence of a different product, replace `sendmail` with the name of the application you're writing the plugin for (and the one you just queried for). The third line invokes HQ's already defined auto-discovery process; Keep it as is. In a plugin targeted at a service (not at the server level), there does not exist a similarly simple way to conduct auto-discovery. A simple workaround is to write the plugin at the server level anyway and then include that one service you're interested in. That way, you can use this code as-is.

My Services Aren't Being Auto-Discovered

If your services are not being auto-discovered via your plugin (and you are relying on HQ's built-in auto-discovery process, as above), the most likely culprit is that you left out this line:

```
<property name="HAS_BUILTIN_SERVICES" value="true"/>
```

6. This code simply tells HQ what type of plugin this is — Measurement — so that HQ will know how to execute the plugin.

```
<plugin type="measurement" class="org.hyperic.hq.product.MeasurementPlugin"/>
```

   For all Measurement plugins, keep it as-is. (Learn about other types of plugins.)

Define and Gather Server Metrics

7. This code collects the requisite Availability metric, this time at the Sendmail-server level. (It is later collected for all Sendmail's component services.)

```
<metric name="Availability"
alias="Availability"
template="sigar:Type=ProcState,Arg=%process.query%:State"
category="AVAILABILITY" indicator="true" units="percentage" collectionType="dynamic"/>
```

   The variable `%process.query%` gets assigned the value from the above process.query (which determined whether or not the server exists), and so then the plugin can determine whether or not the service is available. Keep it as-is.

Use of Variables in XML Descriptors

In plugin XML descriptors, variables are indicated by "%" on either side of the variable name (for example, `%process.query%`). The variables are assigned the value that was most recently determined. The value of the variable must be determined *before* the variable is used.

You Must Always Collect the Availability Metric

The Availability metric indicates whether a Resource is up or down.

A metrics-gathering plugin must determine Availability for *every* server and *every* service it monitors. A single plugin will likely gather Availability for multiple Resources. If Availability is not gathered

> for a Resource, HQ will consider the Resource to be unavailable, and will not show any metrics for it in the Portal.
>
> A plugin sets the value of Availability to 1 if the Resource is up, and 0 if it is down. These values are displayed in the Portal as "available" or "not available".
>
> Verifying the existence of a Resource's process is a common technique for determining its Availability. However, the method a plugin uses to determine Availability can vary depending on the Resource Type and the plugin developer's judgment. There might be alternative techniques for determining the Availability of a Resource. For instance, a plugin might determine the Availability of a web server based on whether its process is up, its port is listening, it is responsive to a request, or by some combination of these conditions.

For each metric you collect, you must specify at least the following attributes:

Metric Attribute	**Description**	**Req'd**	**Possible Values**
`name`	Name of the metric to be displayed in the GUI	Y	
`indicator`	Whether or not this metric should be treated as an indicator metric in HQ	Y	true, false
`template`	Unable to render {include} Couldn't find a page to include called: TemplateLearn more [217].	Y	See examples [217]

See all the possible attributes [214].
Identify and Gather Metrics from the Server's Hosted Services

8. Now that we've established the minimum requirements at the Sendmail server level (auto-discovery and the Availability metric), we can proceed to the component Sendmail processes. The next many lines of code deal with each individual Sendmail service (process) under the Sendmail server: SendMail Message Submission Process, SendMail Root Daemon Process, and SMTP. If your server contains services, break them out here just as this code does.

```
<service name="SendMail Message Submission Process">
<config>
<option name="process.query"
default="State.Name.eq=sendmail,CredName.User.ne=root"
description="PTQL for SendMail Message Submission Process"/>
</config>
<metric name="Availability"
template="sigar:Type=ProcState,Arg=%process.query%:State"
indicator="true"/>
</service>
```

First the existence of the exact Sendmail process — SendMail Message Submission Process — is determined with a `process.query`. In this case, this process can be uniquely identified by asking for a Sendmail process that does *not* have root access (`CredName.User.ne=root`). If you are working with another process, then replace `sendmail` with the appropriate server name or `CredName.User.ne` with the appropriate filtering criterion. Again, this query is written in PTQL, which is unnecessary for this tutorial, but will enable you to write a variety of queries. Then the requisite Availability metric (of the specific process we just identified) is gathered. Keep it as-is.

Finding Process Names with SIGAR

To find the exact names of the processes you are interested in so that you can name them in your plugin:

a. i. In the HQ Agent directory, start SIGAR:

```
java -jar <Agent dir>/pdfk/lib/sigar.jar.
```

ii. At the SIGAR prompt, run:

```
sigar> ps State.Name.eq=sendmail
```

where, instead of `sendmail` you type the name of the application you're running the script against. The results of this command are equivalent to those returned by `ps`: a list of Sendmail processes running.

```
In the XML descriptor, you would use one of these process names like this:
<service name="process_name_here">
```

where `process_name_here` can be, for example, `SendMail Message Submission Process`.

Learn more about using SIGAR for process identification.

9. Now the second Sendmail process — SendMail Root Daemon Process — is identified with another `process.query`.

```
<service name="SendMail Root Daemon Process">

<config>
<option name="process.query"
default="State.Name.eq=sendmail,CredName.User.eq=root"
description="PTQL for SendMail Root Daemon Process"/>
</config>

<metric name="Availability"
template="sigar:Type=ProcState,Arg=%process.query%:State"
indicator="true"/>
</service>
```

Unlike with the first process, this time we want a Sendmail process *with* root access (`CredName.User.eq=root`). If you are working with another process (service), then replace `sendmail` with the appropriate server name and `CredName.User.eq` with the appropriate filtering criterion. Then the requisite Availability metric (of the specific process we just identified) is gathered. Keep it as-is.

10. This code collects two metrics for the Sendmail server.

```
<filter name="template"
value="exec:timeout=10,file=pdk/work/scripts/sendmail/hq-sendmail-stat,exec=%exec%:
${alias}"/>

<metric name="Messages In Queue"
indicator="true"/>

<metric name="Messages Awaiting Preprocessing"
indicator="true"/>
```

The code is not defined within a service object, therefore it applies to the server.

How Do Filters and Templates Work?

*Topics marked with*relate to features available only in vFabric Hyperic.*

Element Overview

The `<filter>` element declares a variable that can be referenced by descendent elements in the descriptor to obtain resource-type specific values for other expressions. The most common use for the `<filter>` element is to define a string that forms all or a portion of the `template` attribute for `<metric>` elements, instead of explicitly defining the `template` for each metric.

A `<filter>` element is available to all descendants of its parent. For example, a `<filter>` definition in the plugin root applies to all `<platform>`, `<server>` and `<service>` elements that follow. You override the filter value a child inherits from a parent by defining a filter of the same name in the child element.

Parent Elements

- `<plugin>` (root) - A `<filter>` element here is available to all resource elements in the descriptor
- `<platform>` - A `<filter>` element here is available within the element and its descendants.
- `<server>` - A `<filter>` element here is available within the element and its descendants.
- `<service>` - A `<filter>` element here is available only within the element.

Child Elements and Attributes

- `name` - (Required) Name by which the variable is referenced. The name can be an arbitrary value, or exactly match the name of an attribute the filter defines. Typically the name attribute defines a filter named "template" which defines the template attribute forelements using other variables whose values are assigned at the resource and metric level.
- `value` - (Required) Value of the variable, expressed explicitly or using other variables.

Examples<filter> defines metric template components using configuration and metric alias

This excerpt from the HQ Zimbra plugin, is part of a `<server>` element that defines:

- a `<filter` named "template" that specifies a building block in terms of the `installpath` configuration option and the `metric` element's `alias` attribute.
- an `<option>` element that defines the `installpath` configuration option.

```
<server name="Zimbra"
     version="4.5.x">

        <plugin type="autoinventory"
         class="ZimbraServerDetector"/>
        <plugin type="measurement"
         class="org.hyperic.hq.product.MeasurementPlugin"/>
        <plugin type="collector"
         class="org.hyperic.hq.plugin.zimbra.ZimbraCollector"/>

        <filter name="template"
         value="zimbra-stats:installpath=%installpath%:${alias}"/>

        <properties>
            <property name="version"
```

```
             description="Zimbra Version"/>
        </properties>

        <config>
            <option name="installpath"
             default="/opt/zimbra"
             description="Zimbra Install Path"/>
            <option name="zimbra-ptql"
             default="Pid.PidFile.eq=%installpath%/log/zmtomcatmgr.pid"
             description="Sigar PTQL Process Query"/>
        </config>
```

<filter> defines metric template using a values supplied by a <property> and a <metric> alias attribute

This element from the HQ Resin plugin defines a filter named template that references the values of a resource named OBJECT_NAME and a element's alias attribute. This reference provides two components of the metric template:

```
<filter name="template"
value="${OBJECT_NAME}:${alias}"
/>
```

Because the filter is defined in the root `<plugin>` element, it is available to all resource elements in the descriptor.

The value of the OBJECT_NAME property is defined in each resource element in the descriptor, a `<server>` and two{{ <service>}} elements. For example:

```
<service name="Port">
<property name="OBJECT_NAME"
value="resin:type=Port,name=*"/>
```

The template filter is expanded for each server and service metric, using the value of the owning resource's OBJECT_NAME property and the metric's alias attribute.

<filter> references globally available configuration schema

This excerpt shows the use of a globally available configuration schema in a filter:

```
<filter name="iplanet.snmp"
value="iplanet:$\{snmp.config\}"/>
```

The "snmp" schema is defined in the HQ Netservices plugin descriptor, as shown in this excerpt:

```
description="SNMP agent IP address"
default="127.0.0.1"/>
description="SNMP agent port"
type="port"
default="161"/>
description="SNMP Version"
type="enum">
```

In the filter definition,

`${snmp.config`} is replaced by:

snmpIP=,
`snmpPort=*,*snmpVersion=*`

resulting in:

```
iplanet:snmpIP=,snmpPort=,snmpVersion=*
```

In the same plugin descriptor, the filter is referenced in a template definition:

```
template="${iplanet.snmp}:httpStatisticsRequests:
${server.config.v4}"
```

<filter> uses value of config <option> to provide "domain" portion of metric template

The HQ mssql-plugin descriptor defines several filters in the root, based on configuration options:

```
<plugin name="mssql">
   ......
 ......
  <filter name="db.domain"
          value="Databases(${db.name})"/>
  <filter name="lock.domain"
          value="Locks(${lock.name})"/>
  <filter name="cache.domain"
          value="Cache Manager(${cache.name})"/>
  .....
  .....
```

Also in the root, `<metric>` elements, define the `template` in terms of the filter:

```
<metrics name="mssql-avail">
    <metric name="Availability"
            alias="Availability"
            template="${db.domain}:Type=Availability:Active Transactions"
            category="AVAILABILITY"
            group="Reliability"
            indicator="true"
            collectionType="dynamic"
            units="percentage"/>
  </metrics>
```

Later in the descriptor, each resource element, like this `<service>` element, defines the `<option>`:

```
<service name="Database">
        ......
        ......
      <config>
        <option name="db.name"
                description="Database name"
                default="Northwind"/>
      </config>
    </service>
```

In this case, the filter applies the template to all the following metrics. Learn more about using filters to apply a template to metrics [218].

Unable to render {include} Couldn't find a page to include called: Template

Learn more about templates [217].

To get each of the two metrics, HQ will run the `hq-sendmail-stat` script, with a 10-second timeout, using the `exec` value previously established, which is prepended to whatever it is you're executing (here, instructions to use "sudo" are prepended to the script). The use of `$(alias)` means that the `alias` value will assume the metric `name` specified just below. Learn more about the alias attribute [214].

It's a Good Habit to Specify a Timeout Value

When HQ runs a script, it will use a default timeout value. However, it is a good practice to make the timeout value explicit (implicit unit of measure is seconds) when using the `exec` command so that, if the script fails, that is one fewer variable you have to figure out in your troubleshooting.

11. Now we address the last Sendmail process of interest: SMTP.

```
<service name="SMTP">

<config>
<option name="hostname"
description="SMTP Hostname"
default="localhost"/>
</config>

<filter name="template"
value="SMTP:hostname=%hostname%:${alias}"/>

<metric name="Availability"
indicator="true"/>

<metric name="Inbound Connections"
indicator="true"/>

<metric name="Outbound Connections"
indicator="true"/>

</service>
</server>
```

This code takes advantage of a pre-existing "protocol checker" for SMTP, which knows how to get SMTP metrics. You must first declare SMTP as a service and then simply pass to the checker the hostname and the metric name, as is done here. If you are interested in additional SMTP metrics, simply add another line specifying the metric name. Learn more about protocol checkers [218].

Provide Configuration Instructions

12. This code includes the configuration instructions that show up at the bottom of the resource "Resource Configuration" screen.

```
<!--
 =================== Plugin Help ==========================
-->
<help name="SendMail">
  <p>
  <h3>Configure HQ for monitoring SendMail</h3>
  </p>
  <p>
  This plugin needs sudo access as root in order to access the appropriate SendMail dirs.
  <br>
  To configure sudo (in /etc/sudoers):
  <br>
  Cmnd_Alias HQ_SENDMAIL_STAT = &lt;hqdir&gt;/agent/pdk/work/scripts/sendmail/hq-
sendmail-stat
  <br>
  &lt;agentuser&gt; ALL = NOPASSWD: HQ_SENDMAIL_STAT
  </p>
  </help>
</plugin>
```

For your plugin, replace this help with whatever configuration instructions are necessary for getting the plugin running in your environment (prerequisites, etc.). Learn more about using the help tag.

back to top

2.5.6. Step 5. Test the Plugin

This page is replaced by Section 2.10, “Testing Plugins”.

back to top

2.5.7. Step 6. Deploy the Plugin.

Before deploying your complete and successfully tested plugin, if your script is not already embedded in the plugin, make sure the script is in `<Agent dir>/pdk/scripts` and that it's executable.
You must deploy your custom plugin on the HQ Server and on all the Agents you want to run the plugin.

Deploying the Plugin in .xml or .jar Format

You can name your plugin as you please, but the name must end with either `-plugin.xml` or `-plugin.jar`, depending on the plugin format.

Plugin Format	**Description and Contents**
`-plugin.jar`	A standard jar-formatted file, containing `etc/hq-plugin.xml` — the plugin's XML descriptor — and any java class files, scripts, and MIB files.
`-plugin.xml`	For plugins that consist solely of an XML descriptor. It is combined with HQ's plugin support classes (either in the hq-pdk.jar or other *-plugin.jar plugins). In the case of a script plugin, the script can also be included in a separate file (as opposed to embedded in the XML descriptor).

Where to Place Custom-Plugin Files

Usually you will place your custom-plugin files in a subdirectory of the Server's and Agent's parent directories. By doing this, your plugin files will persist through upgrades. The subdirectory is called `hq-plugins`. For example:

Non-Windows Installation

- If the Server is installed at `/usr/local/hyperic/<Server directory>`, then add the plugin in `/usr/local/hyperic/hq-plugins`
- If the Agent is installed at `/usr/local/hyperic/<Agent directory>`, then add the plugin in `/usr/local/hyperic/hq-plugins`

Windows Installation

- If the Server is installed at `C:\Program Files\Hyperic HQ\<Server directory>`, then add the plugin in `C:\Program Files\Hyperic HQ\hq-plugins`

- If the Agent is installed at `C:\Program Files\Hyperic HQ\<Agent directory>`, then add the plugin in `C:\Program Files\Hyperic HQ\hq-plugins`

> **hq-plugins/ must exist prior to starting the Server**
>
> Hot-deployment on the Server side using the custom `hq-plugins/` directory requires that the directory exist prior to starting the Server. So, after creating `/usr/local/hyperic/hq-plugins/`, you must restart the Server.

Alternate File Placement
You can also place the files under the Server and Agent directories:

- On the Server: `<Server directory>/hq-engine/hq-server/webapps/ROOT/WEB_INF/hq-plugins/`
- On the Agent: `<Agent directory>/bundles/agent-x.y.z/pdk/plugins/`

You should do this when, for example, you are installing a patch to a plugin, or if you want to keep all your Agent-deployment files together.

Hot Deployment of Plugins

The Server supports hot-deployment: plugins can be updated or added without restarting the Server.

The Agent does *not* support hot-deployment: it must be restarted after a plugin is updated or added.
back to top

2.5.8. Done!

Your plugin should now kick into action the next time the Agent runs (assuming it's been rebooted since the plugin was added to it), and you should see all the desired metrics in the HQ UI.

back to top

2.6. JMX Plugin Tutorial

This tutorial will lead you through writing, testing, and deploying a simple JMX measurement plugin that auto-discovers and collects several metrics from Sun JVM. We hope that you will be able to extrapolate from this simple example to your own needs for discovering and monitoring a remote JMX-enabled application.

JMX plugins target remote JMX-enabled applications. They extract metrics from the Java services via MBeans. One of the main tasks of writing a JMX plugin is determining which metrics to monitor via those MBeans. JMX plugins are templatized and so you will not need to write any Java code. All you need to do is write an XML descriptor.

Learn more about JMX plugins and measurement plugins.

See the Section 2.3, “Plugins Tutorial” overview for general plugin information.

Procedural Overview for Writing a JMX Measurement Plugin

1. Configure the JMX-enabled application for remote connections [43].
2. Determine which metrics [43] to collect in the plugin.
3. Review the sample JMX measurement Plugin descriptor [44] and copy it for your own use.

4. See what this plugin will show in the HQ UI [45].
5. Understand each component of the descriptor and tailor the XML descriptor [46] to reflect your own plugin.
6. Test the plugin. [56]
7. Deploy the plugin. [56]

2.6.1. How the Plugin Fits Into HQ's Inventory Model

Unable to render {include} Couldn't find a page to include called: Plugin_InvModel

This plugin fits into the HQ inventory model by starting at the (Sun JVM) server level and organizing services (in this case, only one: Garbage Collector) under it. Other plugins targeting only a service could bypass the server and be organized directly under the platform on which it is run.

back to top

2.6.2. Step 1. Configure the JMX-Enabled Application for Remote Connection

In order to manage a JMX-enabled application from HQ, it must be configured to accept remote connections. Many such applications have the remote connector enabled by default; those which do not often require changing a line or two of their configuration.

Learn more about configuring for remote connection [20].

back to top

2.6.3. Step 2. Determine Which Metrics to Collect in the Plugin

You can find appropriate metrics to collect from MBeans in several ways:

- JConsole
- MC4J
- Command line tool [20] provided by HQ JMX support classes. It dumps MBeans in text format.

To find metrics using JConsole:

1. Run JConsole.
   JConsole will discover all Java processes on a host that has JMX enabled and will enumerate all the available MBeans.
2. Find the MBean attribute of interest and its value.
   A numeric value indicates that the attribute is capable of being returned as a metric. This includes two statistic types:
   - javax.management.j2ee.statistics.CountStatistic (gets the count)
   - javax.management.j2ee.statistics.RangeStatistic (gets the current value)
     A non-numeric value indicates that the attribute is not appropriate for collection by the plugin.
3. Record the MBean name, for use in the XML descriptor.

back to top

2.6.4. Step 3. Review the Sample JMX Measurement Plugin Descriptor

The sample JMX plugin XML descriptor below collects several metrics and auto-discovers a Sun JVM server and a single component service. The sample provides the structure of the plugin, and you should start with it when writing your own plugin. You will need to change certain values — which we'll point out — within the descriptor. That's it. All you need for a JMX measurement plugin is an XML descriptor.

```
<?xml version="1.0"?>

<!DOCTYPE plugin [<!ENTITY process-metrics SYSTEM "/pdk/plugins/process-metrics.xml">]>

<plugin package="org.hyperic.hq.plugin.java">

<classpath>
<include name="pdk/lib/mx4j"/>
</classpath>

<filter name="template" value="${OBJECT_NAME}:${alias}"/>

<metrics name="Class Loading Metrics">
<metric name="Loaded Class Count" indicator="false" category="THROUGHPUT"/>
<metric name="Total Loaded Class Count" indicator="false" category="THROUGHPUT"/>
<metric name="Unloaded Class Count" indicator="false" category="THROUGHPUT"/>
</metrics>

<metrics name="Compilation">
<metric name="Total Compilation Time" indicator="false" category="THROUGHPUT"
 collectionType="trendsup" units="ms"/>
</metrics>

<metrics name="Garbage Collector">
<metric name="Collection Count" indicator="false" category="THROUGHPUT"
 collectionType="trendsup"/>
<metric name="Collection Time" indicator="false" category="THROUGHPUT"
 collectionType="trendsup"/>
</metrics>

<metrics name="Memory">
<metric name="Object Pending Finalization Count" category="THROUGHPUT" indicator="false"/>
</metrics>

<metrics name="Threading">
<metric name="Thread Count" category="UTILIZATION" indicator="false"/>
<metric name="Daemon Thread Count" category="UTILIZATION" indicator="false"/>
</metrics>

<server name="Java" version="1.5.x">
<property name="HAS_BUILTIN_SERVICES" value="true"/>
<property name="VERSION_FILE" value="jre/lib/fontconfig.Sun.2003.bfc"/>
<property name="DEFAULT_PROGRAM" value="bin/java"/>
<property name="domain" value="Java"/>

<config>
<option name="jmx.url" description="JMX URL to MBeanServer"  default="service:jmx:rmi:///
jndi/rmi://localhost:6969/jmxrmi"/>
<option name="jmx.username" description="JMX username" optional="true" default=""/>
<option name="jmx.password" description="JMX password" optional="true" default=""
 type="secret"/>
<option name="process.query" description="PTQL for Java Process"
 default="State.Name.eq=java,Args.*.ct=proc.java.home"/>
```

```
</config>

<metric name="Availability" template="sigar:Type=ProcState,Arg=%process.query%:State"
 indicator="true"/>
&process-metrics;

<property name="OBJECT_NAME" value="java.lang:type=ClassLoading"/>

<metrics include="Class Loading Metrics"/>
<property name="OBJECT_NAME" value="java.lang:type=Compilation"/>

<metrics include="Compilation"/>
<property name="OBJECT_NAME" value="java.lang:type=Memory"/>

<plugin type="log_track" class="org.hyperic.hq.product.jmx.MxNotificationPlugin"/>

<property name="OBJECT_NAME" value="java.lang:type=Threading"/>
<metrics include="Threading"/>

<!-- derive installpath from JAVA_HOME env prop... -->
<property name="PROC_HOME_ENV" value="JAVA_HOME"/>

<!-- derive installpath from -Dproc.java.home=... -->
<property name="PROC_HOME_PROPERTY" value="proc.java.home"/>
<plugin type="autoinventory" class="org.hyperic.hq.product.jmx.MxServerDetector"/>
<plugin type="measurement" class="org.hyperic.hq.product.jmx.MxMeasurementPlugin"/>

<service name="Java GC">
<plugin type="autoinventory"/>
<property name="OBJECT_NAME" value="java.lang:type=GarbageCollector,name=*"/>
<metrics include="Garbage Collector"/>
</service>
</server>

<server name="Java" version="1.6.x" include="1.5.x">
<property name="VERSION_FILE" value="jre/lib/management-agent.jar"/>
</server>

<!--
 ==================== Plugin Help ==========================
-->
<help name="Java">
<![CDATA[
  <p>
  <h3>Configure HQ for monitoring Java</h3>
  </p>
  <p>
  1) Add this line to the java options when executing the binary.
  <br>
  "-Dcom.sun.management.jmxremote \
  <br>
  -Dcom.sun.management.jmxremote.port=6969 \
  <br>
  -Dcom.sun.management.jmxremote.ssl=false \
  <br>
  -Dcom.sun.management.jmxremote.authenticate=false"
  <br>
  </p>
]]>
</help>
<help name="Java 1.5.x" include="Java"/>
<help name="Java 1.6.x" include="Java"/>
</plugin>
```

Example 3. JMX Measurement Plugin

2.6.5. Step 4. What This Plugin Will Show in the HQ UI

After this plugin is successfully run by an Agent, the HQ UI will show:

- The existence of the JMX-enabled application (Sun JVM server) and its one hosted service: Java GC (Garbage Collector)
- For that server, these metrics:
  - Availability of the server
  - Loaded Class Count
  - Total Loaded Class Count
  - Unloaded Class Count
  - Total Compilation Time
  - Object Pending Finalization Count
  - Thread Count
  - Daemon Thread Count
- For that server, log-tracking data for the Threading MBean (displayed on the "Current Health" screen)
- For the service, these metrics:
  - Collection Count
  - Collection Time
- Configuration instructions for the JMX-enabled server, on the server's "Resource Configuration" screen

Learn more about viewing resources and metrics in HQ.

back to top

2.6.6. Step 5. Understand and Modify Each Piece of the Sample Plugin XML Descriptor

The sample XML descriptor above is divided into parts, each of which is repeated and explained for your particular use below. Some of the more important parts of the XML descriptor are called out.

Learn more about the plugin XML descriptor.

> **Order Matters**
>
> If a part of the XML descriptor references or includes another part of the XML Descriptor, the *referenced* part must come before the part doing the referencing. This can be seen in this tutorial with the inclusion of metrics parameters, for example.

1. This code, in conjunction with one line later, retrieves the standard process metrics for the process obtained by a later process query [52].

```
<!DOCTYPE plugin [<!ENTITY process-metrics SYSTEM "/pdk/plugins/process-metrics.xml">]>
```

It is a very simple way to retrieve such useful metrics as StartTime, Memory Utilization, and CPU System Time. Keep it as-is.
This code works when a *single* process is retrieved by the query; if the query retrieves multiple processes, then simply add `multi-` just in front of the two occurrences of `process-metrics` here, and there will be an equivalent change in the later line:

```
<!DOCTYPE plugin [<!ENTITY multi-process-metrics SYSTEM "/pdk/plugins/multi-process-
metrics.xml">]>
# This code provides the path for the Java package.
{code:xml}
<plugin package="org.hyperic.hq.plugin.java">
```

Replace it with the path in your environment.

2. This code enables JMX classes for which HQ has already set up libraries.

```
<classpath>
<include name="pdk/lib/mx4j"/>
</classpath>
```

Keep it as-is.
Define Metrics

3. This code assigns a `template` value to each of the metrics defined just after it.

```
<filter name="template" value="${OBJECT_NAME}:${alias}"/>
```

> ### How Do Filters and Templates Work?
>
> *Topics marked with***relate to features available only in vFabric Hyperic.*
>
> Element Overview
>
> The `<filter>` element declares a variable that can be referenced by descendent elements in the descriptor to obtain resource-type specific values for other expressions. The most common use for the `<filter>` element is to define a string that forms all or a portion of the `template` attribute for `<metric>` elements, instead of explicitly defining the `template` for each metric.
>
> A `<filter>` element is available to all descendants of its parent. For example, a `<filter>` definition in the plugin root applies to all `<platform>`, `<server>` and `<service>` elements that follow. You override the filter value a child inherits from a parent by defining a filter of the same name in the child element.
> Parent Elements
>
> - `<plugin>` (root) - A `<filter>` element here is available to all resource elements in the descriptor
> - `<platform>` - A `<filter>` element here is available within the element and its descendants.
> - `<server>` - A `<filter>` element here is available within the element and its descendants.
> - `<service>` - A `<filter>` element here is available only within the element.
>
> Child Elements and Attributes

- `name` - (Required) Name by which the variable is referenced. The name can be an arbitrary value, or exactly match the name of an attribute the filter defines. Typically the name attribute defines a filter named "template" which defines the template attribute forelements using other variables whose values are assigned at the resource and metric level.

- `value` - (Required) Value of the variable, expressed explicitly or using other variables.

Examples<filter> defines metric template components using configuration and metric alias

This excerpt from the HQ Zimbra plugin, is part of a `<server>` element that defines:

- a `<filter` named "template" that specifies a building block in terms of the `installpath` configuration option and the `metric` element's `alias` attribute.

- an `<option>` element that defines the `installpath` configuration option.

```
<server name="Zimbra"
     version="4.5.x">

        <plugin type="autoinventory"
         class="ZimbraServerDetector"/>
        <plugin type="measurement"
         class="org.hyperic.hq.product.MeasurementPlugin"/>
        <plugin type="collector"
         class="org.hyperic.hq.plugin.zimbra.ZimbraCollector"/>

        <filter name="template"
         value="zimbra-stats:installpath=%installpath%:${alias}"/>

        <properties>
            <property name="version"
             description="Zimbra Version"/>
        </properties>

        <config>
            <option name="installpath"
             default="/opt/zimbra"
             description="Zimbra Install Path"/>
            <option name="zimbra-ptql"
             default="Pid.PidFile.eq=%installpath%/log/zmtomcatmgr.pid"
             description="Sigar PTQL Process Query"/>
        </config>
```

<filter> defines metric template using a values supplied by a <property> and a <metric> alias attribute

This element from the HQ Resin plugin defines a filter named template that references the values of a resource named OBJECT_NAME and a element's alias attribute. This reference provides two components of the metric template:

```
<filter name="template"
value="${OBJECT_NAME}:${alias}"
/>
```

Because the filter is defined in the root `<plugin>` element, it is available to all resource elements in the descriptor.

The value of the OBJECT_NAME property is defined in each resource element in the descriptor, a `<server>` and two{{ <service>}} elements. For example:

```
<service name="Port">
<property name="OBJECT_NAME"
```

```
value="resin:type=Port,name=*"/>
```

The template filter is expanded for each server and service metric, using the value of the owning resource's OBJECT_NAME property and the metric's alias attribute.

<filter> references globally available configuration schema

This excerpt shows the use of a globally available configuration schema in a filter:

```
<filter name="iplanet.snmp"
value="iplanet:$\{snmp.config\}"/>
```

The "snmp" schema is defined in the HQ Netservices plugin descriptor, as shown in this excerpt:

```
description="SNMP agent IP address"
default="127.0.0.1"/>
description="SNMP agent port"
type="port"
default="161"/>
description="SNMP Version"
type="enum">
```

In the filter definition,

`${snmp.config`} is replaced by:

snmpIP=,
`snmpPort=*,*snmpVersion=*`

resulting in:

`iplanet:snmpIP=,snmpPort=,snmpVersion=*`

In the same plugin descriptor, the filter is referenced in a template definition:

`template="${iplanet.snmp}:httpStatisticsRequests:`
`${server.config.v4}"`

<filter> uses value of config <option> to provide "domain" portion of metric template

The HQ mssql-plugin descriptor defines several filters in the root, based on configuration options:

```
<plugin name="mssql">
   ......
 ......
  <filter name="db.domain"
          value="Databases(${db.name})"/>
  <filter name="lock.domain"
          value="Locks(${lock.name})"/>
  <filter name="cache.domain"
          value="Cache Manager(${cache.name})"/>
  .....
  .....
```

Also in the root, `<metric>` elements, define the `template` in terms of the filter:

```
<metrics name="mssql-avail">
    <metric name="Availability"
            alias="Availability"
            template="${db.domain}:Type=Availability:Active Transactions"
            category="AVAILABILITY"
            group="Reliability"
            indicator="true"
            collectionType="dynamic"
            units="percentage"/>
  </metrics>
```

Later in the descriptor, each resource element, like this `<service>` element, defines the `<option>`:

```
<service name="Database">
        ......
        ......
      <config>
        <option name="db.name"
                description="Database name"
                default="Northwind"/>
      </config>
    </service>
```

In this case, the filter applies the template to all the following metrics. Learn more about using filters to apply a template to metrics [217].

Unable to render {include} Couldn't find a page to include called: Template

Learn more about templates [217].

In this case, the Loaded Class Count, Total Loaded Class Count, Unloaded Class Count, Total Compilation Time, Collection Count, Collection Time, Object Pending Finalization Count, Thread Count, and Daemon Thread Count metrics all have a template, with the value specified, applied to it, without having to repeat the template parameter for each metric. Keep it as-is.

1. This code collects in one place, for readability, all the metrics that will be gathered, later [53], from MBeans.

```
<metrics name="Class Loading Metrics">
<metric name="Loaded Class Count" indicator="false" category="THROUGHPUT"/>
<metric name="Total Loaded Class Count" indicator="false" category="THROUGHPUT"/>
<metric name="Unloaded Class Count" indicator="false" category="THROUGHPUT"/>
</metrics>

<metrics name="Compilation">
<metric name="Total Compilation Time" indicator="false" category="THROUGHPUT"
 collectionType="trendsup" units="ms"/>
</metrics>

<metrics name="Garbage Collector">
<metric name="Collection Count" indicator="false" category="THROUGHPUT"
 collectionType="trendsup"/>
<metric name="Collection Time" indicator="false" category="THROUGHPUT"
 collectionType="trendsup"/>
</metrics>

<metrics name="Memory">
<metric name="Object Pending Finalization Count" category="THROUGHPUT" indicator="false"/
>
</metrics>

<metrics name="Threading">
<metric name="Thread Count" category="UTILIZATION" indicator="false"/>
<metric name="Daemon Thread Count" category="UTILIZATION" indicator="false"/>
</metrics>
```

The Garbage Collector metrics are handled separately [54] from the other metrics. Replace these metrics with the metrics you want to collect. However you name the group of metrics here (Memory, Compilation, etc.), that's the name you will explicitly "include" later.

For each metric you collect, you can must specify at least the following attributes:

Metric Attribute	Description	Req'd	Possible Values
`name`	Name of the metric to be displayed in the GUI	Y	
`indicator`	Whether or not this metric should be treated as an indicator metric in HQ	Y	true, false
`template`	Unable to render {include} Couldn't find a page to include called: TemplateLearn more [217].	Y	See examples [217]

See all the possible attributes [214].

1. This code specifies the name and version of the process against which the plugin will be run.

```
<server name="Java" version="1.5.x">
```

These values will be displayed in the UI for the server, so while you *can* name it anything you want, we recommend naming it something meaningful.
Tell the Plugin About Services the Server Hosts

1. This code tells HQ that the server has component services, which tells the later-invoked auto-discovery function to please auto-discover the services, too.

```
<property name="HAS_BUILTIN_SERVICES" value="true"/>
```

Keep it as-is if your server contains services you want to discover and manage. Otherwise, replace `true` with `false`.

2. This code enables the plugin to identify different versions of Java.

```
<property name="VERSION_FILE" value="jre/lib/fontconfig.Sun.2003.bfc"/>
```

The file `jre/lib/fontconfig.Sun.2003.bfc` exists only in version 1.5, not in 1.6, and therefore enables the plugin to identify a version as 1.5, which is necessary to determine what parts of the overall plugin to run against the server. If necessary, replace this with a file that uniquely identifies the version of your server.

3. This code specifies the actual program name being run from the default path.

```
<property name="DEFAULT_PROGRAM" value="bin/java"/>
```

If Java is being run from `/usr/jdk/latest/bin/java`, then `/bin/java` is the value you should put here.

4. This code uniquely identifies the plugin's domain.

```
<property name="domain" value="Java"/>
```

It it typically the name of the server, specified earlier [51], but can be anything. If desired, replaced "Java" with your unique identifier.

Enable the Plugin to Connect to the MBean Server

1. This code provides the configuration properties HQ needs to connect to the MBean server.

```
<config>
<option name="jmx.url" description="JMX URL to MBeanServer" default="service:jmx:rmi:///
jndi/rmi://localhost:6969/jmxrmi"/>
<option name="jmx.username" description="JMX username" optional="true" default=""/>
<option name="jmx.password" description="JMX password" optional="true" default=""
 type="secret"/>
```

All the configuration options specified here are the ones that show up in the server's "Resource Configuration" screen. The default values for each of these options can be specified here, but users can change the default values on that screen. You can leave most of the default values in this code as-is but should change the default username and password.
For a JMX plugin, this list of configuration options suffices, but you can add other options. Learn more about how to write an option Learn more about login configuration properties [21] or configuration properties [23] in general.

Identify the Exact Processes You Want to Collect Metrics From

1. This last of configuration options specifically identifies the exact process(es) of interest.

```
<option name="process.query" description="PTQL for Java Process"
 default="State.Name.eq=java,Args.*.ct=proc.java.home"/>
</config>
```

`ct.=proc.java.home` narrows the query to only the salient processes (those with "proc.java.home" in the names). The query is written in PTQL, which isn't necessary to understand for this tutorial, but would be useful to understand for future, custom plugins.
If you are interested in different processes, replace `proc.java.home` with another partial name that will narrow the search. The `process.query` value is used later on when gathering metrics.
Gather Metrics

2. This code collects the requisite Availability metric.

```
<metric name="Availability" template="sigar:Type=ProcState,Arg=%process.query%:State"
 indicator="true"/>
```

The variable `%process.query%` gets assigned the value from the above process query (which determined whether or not the Java server exists), and so then the plugin can determine whether or not the server is available. Keep it as-is.

Use of Variables in XML Descriptors

In plugin XML descriptors, variables are indicated by "%" on either side of the variable name (for example, `%process.query%`). The variables are assigned the value that was most recently determined. The value of the variable must be determined *before* the variable is used.

You Must Always Collect the Availability Metric

The Availability metric indicates whether a Resource is up or down.

> A metrics-gathering plugin must determine Availability for *every* server and *every* service it monitors. A single plugin will likely gather Availability for multiple Resources. If Availability is not gathered for a Resource, HQ will consider the Resource to be unavailable, and will not show any metrics for it in the Portal.
>
> A plugin sets the value of Availability to 1 if the Resource is up, and 0 if it is down. These values are displayed in the Portal as "available" or "not available".
>
> Verifying the existence of a Resource's process is a common technique for determining its Availability. However, the method a plugin uses to determine Availability can vary depending on the Resource Type and the plugin developer's judgment. There might be alternative techniques for determining the Availability of a Resource. For instance, a plugin might determine the Availability of a web server based on whether its process is up, its port is listening, it is responsive to a request, or by some combination of these conditions.
> Some mBeans do not have an "Availability" attribute. Even if this is the case, you must still specify an "Availability" metric as this sample plugin does.

3. This line completes the earlier [46] code that retrieves the standard process metrics for the process just obtained by the above `process.query` [52].

```
&process-metrics;
```

Keep it as-is.
This code works when a *single* process is retrieved by the query; if the query retrieves multiple processes, then simply add `multi-` just in front of `process-metrics` here, as there was an equivalent change in the earlier code.

1. This code gathers the metric values.

```
<property name="OBJECT_NAME" value="java.lang:type=ClassLoading"/>
<metrics include="Class Loading Metrics"/>
<property name="OBJECT_NAME" value="java.lang:type=Compilation"/>
<metrics include="Compilation"/>
<property name="OBJECT_NAME" value="java.lang:type=Memory"/>
<metrics include="Memory"/>
...
<property name="OBJECT_NAME" value="java.lang:type=Threading"/>
<metrics include="Threading"/>
```

It references the previously defined metrics [50], which are grouped earlier for easier reference. If you want to gather other metrics, you should:

- Replace the `java.lang:type` value (for example, ClassLoading) with the associated MBean "objectName" (which you can find, for example, in JConsole).

- `Include` a section that you would define earlier [50]. If you `include="Foobar Metrics"`here, then you would define `metrics name="Foobar Metrics"` earlier, with all its component metrics. Learn more about defining metrics [23] including the use of `OBJECT_NAME` [22] and about defining properties [26].

2. This code implements log tracking of the Threading MBeans (the metrics declared just after invoking log tracking).

```
<plugin type="log_track" class="org.hyperic.hq.product.jmx.MxNotificationPlugin"/>
```

Keep it as-is. Learn more about log tracking in JMX plugins [26].

1. This code provides two methods — of which you must use one — for defining the install (base) path for the process.

```
<!-- derive installpath from JAVA_HOME env prop... -->
<property name="PROC_HOME_ENV" value="JAVA_HOME"/>
<!-- derive installpath from -Dproc.java.home=... -->
<property name="PROC_HOME_PROPERTY" value="proc.java.home"/>
```

If using the first method, replace `JAVA_HOME` with the environmental property of the process. If using the second method, replace `proc.java.home` with the defined name-value pair argument of the process; the MxServer Detector class uses this variable to perform auto-discovery.
You could instead use a third method:

```
<property name="PROC_MAIN_CLASS" value="proc.java.home"/>
```

As with `PROC_HOME_PROPERTY`, this variable tells the MxServerDetector class to autodiscover the process with this PTQL: `State.Name.eq=java,Args.*.sw=-D<value of proc main class>`.
Any of these methods will set the installpath config variable to be the current working directory of the process in question. And while this is particularly useful for log-tracking and control plugins (which are outside the scope of this tutorial), this value is also used later [55] in this XML descriptor.

Auto-Discover the Server

1. This one line of code, in conjunction with the results of the earlier `process.query` [52], implements auto-discovery in this plugin.

```
<plugin type="autoinventory" class="org.hyperic.hq.product.jmx.MxServerDetector"/>
```

Keep it as-is. Autoinventory for a server must also include this class. Learn more about auto-discovery in JMX plugins [25].

2. This one line of code tells HQ what type of plugin this is — Measurement — and therefore how it should be run.

```
<plugin type="measurement" class="org.hyperic.hq.product.jmx.MxMeasurementPlugin"/>
```

Keep it as-is.

Identify and Gather Metrics from the Server's Hosted Services

1. This code defines the Garbage Collector as a service and collects its metrics.

```
<service name="Java GC">
<plugin type="autoinventory"/>
<property name="OBJECT_NAME" value="java.lang:type=GarbageCollector,name=*"/>
<metrics include="Garbage Collector"/>
</service>
</server>
```

If you want to collect GarbageCollector metrics, then keep as-is. Autoinventory for a service does not need a class (whereas a server does).

This code looks slightly different than the earlier metrics-collection code [53] because there are multiple types of Garbage Collection, each of which collects the same metrics. In cases where there are numerous types, calling out each type and its respective metrics (which are identical to the other types'), is onerous and unnecessary. Using

```
name=*
```

makes sure that every type of GarbageCollector gets measured. The `<plugin type="autoinventory"/>` line tells HQ to resolve the `*`. If you want to collect metrics of another objectName that similarly has multiple types, replace `Java GC` with a meaningful name and `GarbageCollector` with the objectName, and accordingly modify the earlier metrics enumeration [50].

> ### Declaring Services as Metrics for Convenient Viewing
>
> This plugin declares the Garbage Collector as a service of the Java server. It is also possible to treat it simply as a metric of the server. Why would you do that? Then all the metrics of the would be displayed in the UI on one level — the server level — and you wouldn't have to drill down from the server to the service to access the metrics.

2. This code determines which part of the plugin to use depending on the version of the Java process. Just as we did before [51], we provide a file — `jre/lib/management-agent.jar` — that only exists in Java v1.6 so that HQ can distinguish between v1.5 and v1.6. For version 1.6, the plugin includes everything that's been written for v1.5 plus anything additional we write specifically for v1.6 (in this sample, we haven't written anything additional). The install path that we specified above [54] is used here: the file name is resolved within that path (`JAVA_HOME/jre/lib/management-agent.jar`).

```
<server name="Java" version="1.6.x" include="1.5.x">
<property name="VERSION_FILE" value="jre/lib/management-agent.jar"/>
</server>
```

Provide Configuration Instructions

1. This code includes the configuration instructions that show up at the bottom of the "Resource Configuration" screen.

```
<!--
 ==================== Plugin Help ==========================
-->
<help name="Java">

  <p>
  <h3>Configure HQ for monitoring Java</h3>
  </p>
  <p>
  1) Add this line to the java options when executing the binary.
  <br>
  "-Dcom.sun.management.jmxremote \
  <br>
  -Dcom.sun.management.jmxremote.port=6969 \
  <br>
  -Dcom.sun.management.jmxremote.ssl=false \
  <br>
  -Dcom.sun.management.jmxremote.authenticate=false"
  <br>
  </p>

</help>
```

```
<help name="Java 1.5.x" include="Java"/>
<help name="Java 1.6.x" include="Java"/>
</plugin>
```

This help text — which are instructions for exposing MBeans — should suffice for any plugins gathering JMX metrics. If you are dealing with more than two versions, the `<help name="Java 1.5.x" include="Java"/>` lines can be augmented to include other versions. Additional lines should have the same content but with the server and version replaced with the format: `server version`.
If necessary, replace this help with whatever configuration instructions are necessary for getting the plugin running in your environment (prerequisites, etc.). Learn more about using the help tag.

back to top

2.6.7. Step 6. Test the Plugin.

This page is replaced by Section 2.10, “Testing Plugins”.
For this JMX plugin, you need to include in these commands your JMX credentials: URL, username, and password, like this:

```
... -Dplugins.include=yourPluginName -Djmx.url=<URL> -Djmx.username=<username> -
Djmx.password=<password> ...
```

back to top

2.6.8. Step 7. Deploy the Plugin.

You must deploy your custom plugin on the HQ Server and on all the Agents you want to run the plugin.

Deploying the Plugin in .xml or .jar Format

You can name your plugin as you please, but the name must end with either `-plugin.xml` or `-plugin.jar`, depending on the plugin format.

Plugin Format	**Description and Contents**
`-plugin.jar`	A standard jar-formatted file, containing `etc/hq-plugin.xml` — the plugin's XML descriptor — and any java class files, scripts, and MIB files.
`-plugin.xml`	For plugins that consist solely of an XML descriptor. It is combined with HQ's plugin support classes (either in the hq-pdk.jar or other *-plugin.jar plugins). In the case of a script plugin, the script can also be included in a separate file (as opposed to embedded in the XML descriptor).

Where to Place Custom-Plugin Files

Usually you will place your custom-plugin files in a subdirectory of the Server's and Agent's parent directories. By doing this, your plugin files will persist through upgrades. The subdirectory is called `hq-plugins`. For example:

Non-Windows Installation

- If the Server is installed at `/usr/local/hyperic/<Server directory>`, then add the plugin in `/usr/local/hyperic/hq-plugins`
- If the Agent is installed at `/usr/local/hyperic/<Agent directory>`, then add the plugin in `/usr/local/hyperic/hq-plugins`

Windows Installation

- If the Server is installed at `C:\Program Files\Hyperic HQ\<Server directory>`, then add the plugin in `C:\Program Files\Hyperic HQ\hq-plugins`
- If the Agent is installed at `C:\Program Files\Hyperic HQ\<Agent directory>`, then add the plugin in `C:\Program Files\Hyperic HQ\hq-plugins`

> **hq-plugins/ must exist prior to starting the Server**
>
> Hot-deployment on the Server side using the custom `hq-plugins/` directory requires that the directory exist prior to starting the Server. So, after creating `/usr/local/hyperic/hq-plugins/`, you must restart the Server.

Alternate File Placement
You can also place the files under the Server and Agent directories:

- On the Server: `<Server directory>/hq-engine/hq-server/webapps/ROOT/WEB_INF/hq-plugins/`
- On the Agent: `<Agent directory>/bundles/agent-x.y.z/pdk/plugins/`

You should do this when, for example, you are installing a patch to a plugin, or if you want to keep all your Agent-deployment files together.

Hot Deployment of Plugins

The Server supports hot-deployment: plugins can be updated or added without restarting the Server.

The Agent does *not* support hot-deployment: it must be restarted after a plugin is updated or added.
back to top

2.6.9. Done!

Your plugin should now kick into action the next time the Agent runs (assuming it's been rebooted since the plugin was added to it), and you should see all the desired metrics in the HQ UI.

back to top

Related Information

Plugin Development Center: All things plugin
Tutorial for a Script Measurement Plugin
Overview of all plugin tutorials
Overview of measurement plugins

Overview of JMX plugins
Learn more about the plugin XML descriptor's syntax
Overview of MBeans

2.7. SNMP Plugin

*Topics marked with*relate to features available only in vFabric Hyperic.*

2.7.1. Writing an SNMP Plugin

SNMP is the standard protocol for monitoring network-attached devices, which is leveraged by several bundled HQ plugins and made easy by the PDK.

- The bundled netdevice plugin provides a generic Network Device platform type which can be used to monitor any device that implements IF-MIB (rfc2863) and IP-MIB (rfc4293).
- The Network Host platform type extends Network Device with support for HOST-RESOURCES-MIB (rfc2790).
- The Cisco IOS platform also extends Network Device, adding metrics from CISCO-PROCESS-MIB and CISCO-MEMORY-POOL-MIB.
- The Cisco PIXOS platform extends Cisco IOS, adding metrics from CISCO-FIREWALL-MIB.

In any HQ plug-in, there are two main concepts to understand:

First is the inventory model. Resource types define where things live in the hierarchy along with supported metrics, control actions, log message sources, etc., as well as the configuration properties used by each feature. In the case of implementing a custom SNMP plug-in for a network device, you are typically defining a platform type that collects any scalar variables that apply to the device and one or more service types to collect table data such as interfaces, power supplies, fans, etc.

Second is the metric template attribute which is a string containing all info required to collect a particular data point. In an SNMP plug-in, each of the metrics correlate to an SNMP OID. While we tend to use the object names to gather the desired data points in the plugins, you can also use the numeric OID. This has the added benefit of avoiding having to worry about ready access to the MIB file anywhere the plug-in is used.

The process of implementing a new SNMP based plugin for HQ starts with locating the device vendor's MIB file(s) and choosing which OIDs you wish to collect as metrics in HQ.

2.7.2. Getting Started

We'll be implementing a plugin for NetScreen, an SSL VPN gateway. The first step is to verify basic connectivity to the device using the snmpwalk command:

```
$ snmpwalk -Os -v2c -c netscreen 10.2.0.140 system
sysDescr.0 = STRING: NetScreen-5GT version 5.0.0r11.1 (SN: 0064062006000809, Firewall+VPN)
sysObjectID.0 = OID: enterprises.3224.1.14
sysUpTimeInstance = Timeticks: (186554200) 21 days, 14:12:22.00
sysContact.0 = STRING: ops@hyperic.com
sysName.0 = STRING: ns5gt
sysLocation.0 = STRING: SF Office
sysServices.0 = INTEGER: 72
```

Next, having downloaded and unarchived the MIB packages, we install the NetScreen MIB files in the appropriate location for our machine's SNMP installation:

```
$ sudo cp NS-SMI.mib NS-RES.mib NS-INTERFACE.mib /usr/share/snmp/mibs
```

Now, verify we can view OIDs defined in NS-RES.mib:

```
$ snmpwalk -Os -M /usr/share/snmp/mibs -m all -v2c -c netscreen 10.2.0.140 netscreenResource
nsResCpuAvg.0 = INTEGER: 2
nsResCpuLast1Min.0 = INTEGER: 2
nsResCpuLast5Min.0 = INTEGER: 2
nsResCpuLast15Min.0 = INTEGER: 2
nsResMemAllocate.0 = INTEGER: 47310400
nsResMemLeft.0 = INTEGER: 60884848
nsResMemFrag.0 = INTEGER: 2537
nsResSessAllocate.0 = INTEGER: 19
nsResSessMaxium.0 = INTEGER: 2000
nsResSessFailed.0 = INTEGER: 0
```

And the tabular OIDs defined in NS-INTERACE.mib:

```
$ snmpwalk -Os -M /usr/share/snmp/mibs -m all -v2c -c netscreen 10.2.0.140
 netscreenInterface | more
nsIfIndex.0 = INTEGER: 0
nsIfIndex.1 = INTEGER: 1
nsIfIndex.2 = INTEGER: 2
nsIfIndex.3 = INTEGER: 3
nsIfName.0 = STRING: "trust"
nsIfName.1 = STRING: "untrust"
nsIfName.2 = STRING: "serial"
nsIfName.3 = STRING: "vlan1"
...
```

2.7.3. Iteration 1: A Very Basic Plug-in

Once the MIBs are sorted out, you can begin with a very simple plug-in that might look something like this (line numbers added for instructional purposes):

```
1  <plugin>
2    <property name="MIBDIR" value="/usr/share/snmp/mibs"/>
3
4    <property name="MIBS"
5              value="${MIBDIR}/NS-SMI.mib,${MIBDIR}/NS-RES.mib,${MIBDIR}/NS-INTERFACE.mib"/
>
6
7    <platform name="NetScreen">
8
9      <config include="snmp"/>
10
11     <plugin type="measurement"
12             class="org.hyperic.hq.product.SNMPMeasurementPlugin"/>
13
14     <property name="template" value="${snmp.template}:${alias}"/>
15
16     <metric name="Availability"
17             template="${snmp.template},Avail=true:sysUpTime"
18             indicator="true"/>
19
20     <metric name="Uptime"
21             alias="sysUpTime"
22             category="AVAILABILITY"
```

```
23             units="jiffys"
24             defaultOn="true"
25             collectionType="static"/>
26
27   </platform>
28 </plugin>
```

Let's dissect this to better understand what is going on:

1. The first and last lines enclose the plug-in contents within the tags <plugin> and </plugin>

2. Line 2 defines where the MIB files can be found on the system that will be collecting the SNMP data from the Netscreen device

3. Line 4 and 5 define which specific MIBs our plug-in will use

4. Line 7 begins the Platform definition, and provides the type name that will appear in HQ

5. Line 9 specifies that we want to include the HQ default SNMP information and templates available in the Network Host and Network Device specifications

6. Lines 11 and 12 specify that we are defining a measurement plug-in using the SNMPMeasurementPlugin class we've imported

7. Line 14 declares the template we will use for the measurement data we collect

8. Lines 16 through 18 define how the Availability metric will be collected. The name set for the metric is how it will show up in the HQ UI. Note also that we change the template to denote that Availability is true if we can get the sysUpTime OID data, and that we set this as an indicator value that is turned on (provides the green light / red light information for the Platform)

9. Lines 20 through 25 define our Uptime metric. Note the clarification of the metric alias that will be substituted in for the template's ${alias} (line 14) as data is collected. We also specify the category, units, defaultOn, and collectionType values as per the measurement plug-in documentation

10. Line 27 closes out the platform definition

This gets us going, but does not yet provide us with a lot of useful information about the platform. Before diving in to gather more information, let's take another look at line 21. Instead of using the alias parameter, we could also have defined that line like this:

```
template="${snmp.template}:sysUpTime"
```

This explicitly defines a template for this metric rather than relying on the alias value and the default measurement template we set.

2.7.4. Iteration 2: Additional Platform Metrics

OK. Let's gather some more scalar Platform metrics that might prove interesting:

```
...

26    <metric name="Average CPU Utilization"
27            alias="nsResCpuAvg"
28            units="percent"/>
29
```

```
<metric name="Average CPU Utilization (Last 1 min)"
        alias="nsResCpuLast1Min"
        units="percent"/>

<metric name="Average CPU Utilization (Last 5 min)"
        alias="nsResCpuLast5Min"
        units="percent"/>

<metric name="Average CPU Utilization (Last 15 min)"
        alias="nsResCpuLast15Min"
        indicator="true"
        units="percent"/>

<metric name="Memory Allocated"
        alias="nsResMemAllocate"
        units="B"/>

<metric name="Memory Left"
        alias="nsResMemLeft"
        indicator="true"
        units="B"/>

<metric name="Memory Memory Fragment"
        alias="nsResMemFrag"
        units="B"/>

<metric name="Sessions Allocated"
        alias="nsResSessAllocate"/>

<metric name="Sessions Maximum"
        alias="nsResSessMaxium"/>

<metric name="Sessions Failed"
        alias="nsResSessFailed"
        collectionType="trendsup"/>

...
```

Again, we provide a name value for how the metric will appear in HQ, use the alias to specify the OID name to be used with the template, and where necessary, specify units, whether or not this will be a default indicator, and the collectionType. This gets us good, basic system information for the platform.

2.7.5. Iteration 3: Pulling in Network Interfaces as Platform Services

Now, we want to get information about the device network interfaces. To do this, we must query the SNMP table data from the device, and put them in proper context as Service definitions within HQ. We add the following to the plug-in:

```
...

<!-- index to get table data -->
<filter name="index"
        value="snmpIndexName=${snmpIndexName},snmpIndexValue=%snmpIndexValue%"/>

<filter name="template"
        value="${snmp.template}:${alias}:${index}"/>

<server>
  <service name="Interface">
    <config>
      <option name="snmpIndexValue"
              description="Interface name"/>
    </config>
```

```
80
81        <property name="snmpIndexName" value="nsIfName"/>
82
83        <metric name="Availability"
84                template="${snmp.template},Avail=true:nsIfStatus:${index}"
85                indicator="true"/>
86
87      </service>
88    </server>

...
```

Breaking the collection of this table data down:

1. In lines 68 and 69 we define an index filter to correlate name and value pairs from the SNMP table data

2. In lines 71 and 72 we define a new template that takes into account the OID and its associated index

3. In line 74 we start a Server definition. In this case, the Server's only attributes are the Platform Services we are defining in lines 75 through 87: the network interfaces for the device

4. In lines 75 through 81 he Service is given a name, and the individual interface name is derived by assoicating the snmpIndexValue with the nsIfName (through the snmpIndexNmae association) defined by the OID

5. In lines 83 through 85, like we did at the top, Platform-level, we define our Availability metric, defining availability as true if we can gather nsIfStatus value for the inteface, and setting it as a default indicator.

6. In line 87 we close the Service definition with the </service> tag

2.7.6. Iteration 4: Collecting Network Interface Service Metrics

Collecting the metric data for each interface is very similar to what we did to collect the scalar data for the Platform. The difference is that it is contained within the Service definition. Here's what that looks like:

```
...

87        <!-- nsIfFlow* metrics -->
88        <metric name="Bytes Received"
89                alias="nsIfFlowInByte"
90                indicator="true"
91                collectionType="trendsup"
92                category="THROUGHPUT"
93                units="B"/>
94
95        <metric name="Bytes Sent"
96                alias="nsIfFlowOutByte"
97                indicator="true"
98                collectionType="trendsup"
99                category="THROUGHPUT"
100               units="B"/>
101
102       <metric name="Packets Received"
103               alias="nsIfFlowInPacket"
104               collectionType="trendsup"
105               category="THROUGHPUT"/>
106
107       <metric name="Packets Sent"
108               alias="nsIfFlowOutPacket"
109               collectionType="trendsup"
110               category="THROUGHPUT"/>
111
```

```
    <!-- nsIfMon* metrics -->
    <metric name="Auth Failures"
            alias="nsIfMonAuthFail"
            collectionType="trendsup"
            category="AVAILABILITY"/>

...
```

Iteration 5: Adding Auto-Discovery Components for the Platform

The final touch is to add the necessary pieces for auto-discovery to work. This makes it nice when you use the plug-in, since inventory information for the Platform, and any discoverable services that are defined are automatically pulled into HQ. The additions are:

```
...

7    <!-- for autoinventory plugin -->
8    <classpath>
9      <include name="pdk/plugins/netdevice-plugin.jar"/>
10   </classpath>

...

11     <properties>
12       <property name="sysContact"
13                 description="Contact Name"/>
14       <property name="sysName"
15                 description="Name"/>
16       <property name="sysLocation"
17                 description="Location"/>
18       <property name="Version"
19                 description="Version"/>
20     </properties>
21
22     <plugin type="autoinventory"
23             class="org.hyperic.hq.plugin.netdevice.NetworkDevicePlatformDetector"/>

...

94      <plugin type="autoinventory"
95              class="org.hyperic.hq.plugin.netdevice.NetworkDeviceDetector"/>

...

105       <plugin type="autoinventory"/>
106
107       <properties>
108         <property name="nsIfIp"
109                   description="IP Address"/>
110         <property name="nsIfNetmask"
111                   description="Netmask"/>
112         <property name="nsIfGateway"
113                   description="Gateway"/>
114       </properties>

...
```

1. In lines 7 through 10, we import the netdevice-plugin to enable auto-discovery

2. In lines 11 through 20, we add some inventory properties that will show-up on the Inventory tab

3. In lines 22 and 23, we call-out the NetworkDevicePlatformDetector for auto-inventory of the Platform scalar values (enabled through the inclusion we did in lines 7 through 10)

4. In lines 94 and 95, we call-out the NetworkDeviceDetector for auto-inventory of the Platform table values (also enabled through the inclusion we did in lines 7 through 10)

5. In lines 105 thorugh 114, we insure that the network data is incorporated into the Platform inventory as part of the auto-discovery process

2.7.7. The Final Product

The final plug-in in its entirety is here in netscreen-plugin.xml.

2.7.8. Additional SNMP Plugin Examples

- netscaler
- zxtm
- wxgoos
- squid

2.7.9. Resources

- SNMP Tutorial
- SNMP Glossary
- Cisco MIB browser
- OidView MIB browser
- Net-SNMP
- SNMP4J

2.8. Product Plugin

*Topics marked with*relate to features available only in vFabric Hyperic.*

The ProductPlugin is the deployment entry point on both the Server and Agent. It defines the resource types and plugin implementations for measurement, control, and autoinventory. Most ProductPlugin implementations are done using the Plugin XML Descriptor. However, in order to dynamically generate the classpath, plugins can override ProductPlugin. For example, the JBoss plugin uses SIGAR to find the installpath of a JBoss server running on the machine, which it uses to set the classpath.

Next Steps

TBD

Related Information

TBD

2.8.1. Return to the Plugin Development Center .

2.9. Deploy Plugin

You must deploy your custom plugin on the HQ Server and on all the Agents you want to run the plugin.

2.9.1. Deploying the Plugin in .xml or .jar Format

You can name your plugin as you please, but the name must end with either `-plugin.xml` or `-plugin.jar`, depending on the plugin format.

Plugin Format	Description and Contents
`-plugin.jar`	A standard jar-formatted file, containing `etc/hq-plugin.xml` — the plugin's XML descriptor — and any java class files, scripts, and MIB files.
`-plugin.xml`	For plugins that consist solely of an XML descriptor. It is combined with HQ's plugin support classes (either in the hq-pdk.jar or other *-plugin.jar plugins). In the case of a script plugin, the script can also be included in a separate file (as opposed to embedded in the XML descriptor).

2.9.2. Where to Place Custom-Plugin Files

Usually you will place your custom-plugin files in a subdirectory of the Server's and Agent's parent directories. By doing this, your plugin files will persist through upgrades. The subdirectory is called `hq-plugins`. For example:

Non-Windows Installation

- If the Server is installed at `/usr/local/hyperic/<Server directory>`, then add the plugin in `/usr/local/hyperic/hq-plugins`
- If the Agent is installed at `/usr/local/hyperic/<Agent directory>`, then add the plugin in `/usr/local/hyperic/hq-plugins`

Windows Installation

- If the Server is installed at `C:\Program Files\Hyperic HQ\<Server directory>`, then add the plugin in `C:\Program Files\Hyperic HQ\hq-plugins`
- If the Agent is installed at `C:\Program Files\Hyperic HQ\<Agent directory>`, then add the plugin in `C:\Program Files\Hyperic HQ\hq-plugins`

> **hq-plugins/ must exist prior to starting the Server**
>
> Hot-deployment on the Server side using the custom `hq-plugins/` directory requires that the directory exist prior to starting the Server. So, after creating `/usr/local/hyperic/hq-plugins/`, you must restart the Server.

Alternate File Placement

You can also place the files under the Server and Agent directories:

- On the Server: `<Server directory>/hq-engine/hq-server/webapps/ROOT/WEB_INF/hq-plugins/`
- On the Agent: `<Agent directory>/bundles/agent-x.y.z/pdk/plugins/`

You should do this when, for example, you are installing a patch to a plugin, or if you want to keep all your Agent-deployment files together.

2.9.3. Hot Deployment of Plugins

The Server supports hot-deployment: plugins can be updated or added without restarting the Server.

The Agent does *not* support hot-deployment: it must be restarted after a plugin is updated or added.

2.10. Testing Plugins

The PDK allows you to invoke your plugin methods from the command line for quick testing. To perform any of the following tasks (targeted mainly at measurement plugins), run the associated commands from the command line in the HQ Agent directory:

Check the syntax of the plugin: (simply initializes the plugin)

```
java -jar <Agent Directory>/pdk/lib/hq-pdk-VERSION.jar -
Dplugins.include=yourPluginName -m lifecycle -Dlog=debug
```

Make sure the plugin can indeed collect the metrics: (runs the plugin and outputs the metric values)

```
java -jar <Agent Directory>/pdk/lib/hq-pdk-VERSION.jar -
Dplugins.include=yourPluginName -m metric
```

Make sure the plugin can perform auto-discovery: (enables auto-discovery in addition to metric collection)

```
java -jar <Agent Directory>/pdk/lib/hq-pdk-VERSION.jar -
Dplugins.include=yourPluginName -m discover -a metric
```

Enable debug for troubleshooting:

```
java -jar <Agent Directory>/pdk/lib/hq-pdk-VERSION.jar -
Dplugins.include=yourPluginName -m discover -a metric -Dlog=debug
```

Passing in Config Options

When testing your plugin, you need to also pass in values (using the `-D` argument) for all the config options that are required to make the plugin run, for example, config options that specify credentials. For example, `-Djmx.password=<password>`. Otherwise you will not be able to run your plugin.

It's a Good Habit to Specify an exec.sleep Value

When running the script plugin from the command line, it is best to assign an explicit value to `exec.sleep` just in case the script takes longer to run than the default `exec.sleep` value (30 sec).

```
java -jar <Agent Directory>/pdk/lib/hq-pdk-VERSION.jar -
Dexec.sleep=number_of_seconds ...
```

This doesn't apply to plugins deployed in HQ because there the `timeout` variable kicks in.

Learn more about invoking plugins outside of the Server and Agent, including more testing commands.

3. Plugin Descriptors

3.1. Key Facts About HQ Plugin Descriptors

*Topics marked with***relate to features available only in vFabric Hyperic.*

This page contains information that has not been reviewed. Click **Add Comment** at the bottom of the page to enter suggestions or corrections.

This page is a broad overview of the descriptor component of an HQ resource plugin. High level view of the XML schema, descriptions of the key XML elements, and highlights key concepts and behaviors a plugin developer should understand.

3.1.1. Descriptor Defines Resource Types, Management Functions, and Metrics

A plugin descriptor is an XML file that defines what a plugin does and how - the resource types it manages and for each type, the management funtions it performs, the resource data it requires and discovers, and the metrics it returns.

Every plugin has a descriptor file. If a plugin that uses HQ plugin support classes or a script to perform management functions, the descriptor is the component to develope and deploy. The descriptor for a plugin that uses custom management classes is packaged with the classes in a JAR for deployment

Managed Resource Type Hierarchy

A plugin descriptor defines each resource type the plugin will manage, in some cases a single type, more typically a hierarchy of types, for instance a server (e.g., Tomcat 6.0) and its services (e.g. Vhosts). A plugin can manage multiple resource type hierarchies, for instance, HQ's Tomcat plugin manages Tomcat 5.5 as well as Tomcat 6. The descriptor for such plugins defines a resource hierarchy for *each* version.

Although a plugin can manage a platform and one or more levels of dependent resources, in practice virtually all platform-level resources are managed by a single HQ plugin - the system plugin (system-plugin.jar). The system plugin discovers and manages *all* supported operating system platforms (Unix, Linux, Win32, Solaris, MacOSX, AIX, HPUX , and FreeBSD) and platform services (such as network interface, CPU, file server mount services) for each.

The only other HQ plugins that manage resources that HQ considers platforms are those that manage virtual or network hosts.

Management Functions and Classes for each Resource Type

A plugin can perform one or more management function for each resource type it manages.

For example, the Tomcat plugin enables autodiscovery, metric collection, log tracking, control operations for Tomcat 5.5 and 6.0 servers, and one or more management functions for Tomcat connectors and web applications.

The available management functions include:

Plugin Management Function	**Description**
Discover resources and resource data.	Discover running instances of a resource type and collect resource data (e.g., an Apache server's build date and the path to its executable).
Obtain metrics.	Measure or collect metrics that reflect the availability, throughput, and utilization of a resource instance.
Monitor log files.	Monitor log files for messages that match specified filter criteria, such as severity level or text the message contains (or does not contain).
Monitor configuration files.	Monitor specific files for changes.
Perform resource control actions.	Perform a control action supported by the resource type on a resource instance. For instance, stop a server instance, or run a database housekeeping function.

For each management function, the descriptor specifies the class, support libraries, or external JAR the plugin uses to perform that function. For example, the Tomcat plugin uses `org.hyperic.hq.product.jmx.MxServerDetector` to discover Tomcat instances.

Inventory and Configuration Data for Each Resource Type

For each resource type the plugin manages, the descriptor defines the resource data the plugin uses, including data the plugin needs to discover a resource (e.g. the address of an MBean server, or resource attributes that the plugin discovers. The descriptor defines any plugin the uses or presents whether its value is defined in the descriptor, configured by a user, or returned by a plugin class.

Metrics to Collect for Each Resource Type

The descriptor specifies each metric that the plugin obtains for each resource type it manages. For example, the Tomcat plugin obtains "Availability", "Current Thread Count" and "Current Threads Busy" for a "Thread Pools" service. The rules for obtaining a metric are defined in a structured expression referred to as a *metric template*. A metric template identifies the target metric by the name the relevant measurement class returns it, and provides the data the class requires to obtain the metric (e.g., the resource's JMX ObjectName).

3.1.2. Resource Hierarchy is the Skeleton of a Descriptor

The structure of a plugin descriptor is the same as the hierarchy of resource types the plugin manages, expressed in terms of the HQ inventory model. A plugin descriptor contains a resource type element - `<platform>`, `<server>`, or `<service>` - for each resource type to be managed. Note that the resource element hierarchy in the descriptor must reflect relationships between the managed resource types. For example, a `<server>` element for a Tomcat type contains (is the parent of) the `<service>` element for the Vhost type.

The left column below illustrates *all* of the resource element relationships that are valid in a plugin descriptor. Elements that map to resource types shown in bold. (No element attributes are shown, and some lower level elements are excluded. The child elements below each resource type element are used to define the resource data, plugin functions, and metrics for that resource type.

The right column illustrates the descriptor structures for resource hierarchies of varying depth.

Supported Element Relationships	**Element Structures for Various Resource Hierarchies**

<plugin>
   <filter>
   <property>
   <config>
      <option>
   <properties>
   <help>
   <metrics>
   <script>
   <classpath>
   <platform>
      <filter>
      <property>
      <config>
      <properties>
      <plugin>
      <help>
      <metrics>
      <metric>
      <actions>
      <classpath>
      <script>
         <server>
         <filter>
         <property>
         <config>
            <option>
         <properties>
         <plugin>
         <help>
         <metrics>
         <metric>
         <actions>
         <scan>
         <service>
            <filter>
            <property>
            <config>
               <option>
            <properties>
            <plugin>
            <help>
            <metrics>
            <metric>
            <actions>
   <server>
      <service>
   <service>
 	
.....
	Platform-Platform Service
	(NOT TYPICAL)
	HQ's system-plugin manages all of HQ-supported operating system platforms, and services that run on each, usually referred to as *platform services*. The plugin descriptor defines a <platform>-<service> hierarchy for each operating system platform.

	<plugin>
	   <platform>
	     <service>
	        .....
	   <platform>
	      <service>
	         .....

	Platform-Server-Service (NOT TYPICAL) This structure is valid but uncommon. The only HQ plugins that manage a platform-server-service are plugins that manage virtual platforms.

	<plugin>
	   <platform>
	      <server>
	         <service>

	Server-Service (TYPICAL) Most HQ plugins manage a server type and the service types that it contains; the <server> element is the root of the plugin, and contains a <service> element for each of the services the plugin manages. The descriptor for a plugin that manages multiple versions of a server type (e.g the plugin for Tomcat 5.5 and 6.0) defines a <server>-<service> hierarchy for each.

	<plugin> (root)
	   <server>
	      <service>
	   <server>
	   <service>

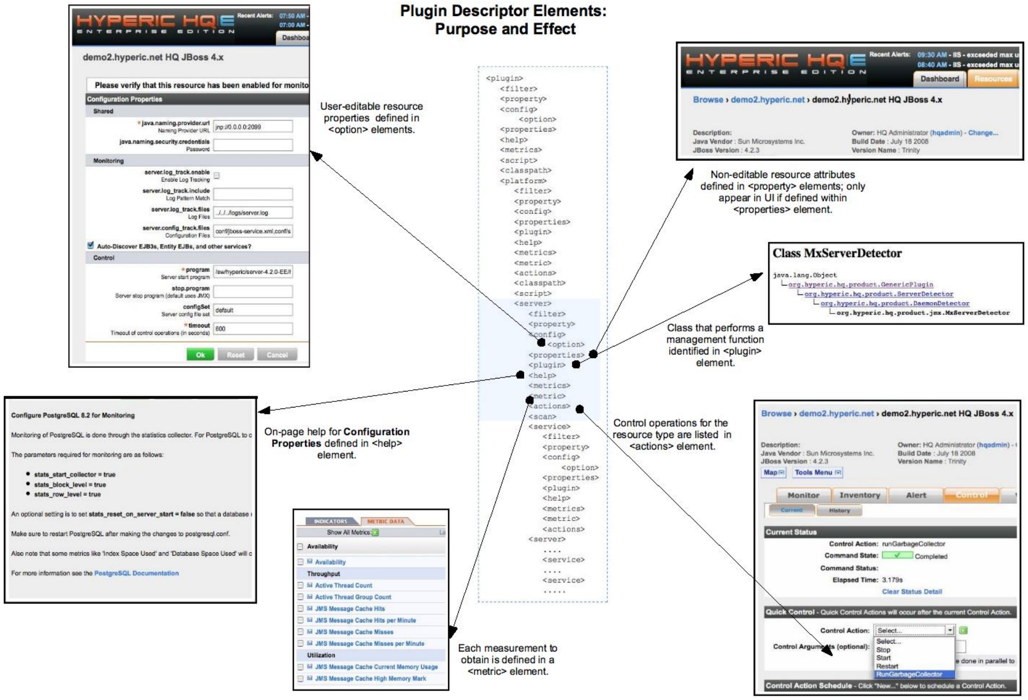
	Platform Service If a plugin manages only a platform service the <service> element appears in the root of the plugin.
	<plugin> <service>

3.1.3. A Functional View of Common Plugin Descriptor Elements

The children of a resource element are the meat of a plugin - they define what the plugin does and how, including:

- Metrics to collect, along with units of measure, the type of the metric, and other attributes that characterize metric nature and behavior.
- One or more management functions, and the class or script that performs each.
- Resource data the plugin uses, and related user interface behaviors, whether and where resource properties are presented in the HQ user interface, defaults and allowable values for configurable data, and so on.

The following table below introduces elements you can define for each resource type a plugin will manage. Click the thumbnail to display a diagram that visualizes the purpose and effect of key descriptor elements.


Plugin Descriptor Elements:
Purpose and Effect
User-editable resource properties defined in <option> elements.
Non-editable resource attributes defined in <property> elements; only appear in UI if defined within <properties> element.
Class MxServerDetector
Class that performs a management function identified in <plugin> element.
On-page help for Configuration Properties defined in <help> element.
Control operations for the resource type are listed in <actions> element.
Each measurement to obtain is defined in a <metric> element.
<plugin>
<filter>
<property>
<config>
<option>
<properties>
<help>
<metrics>
<script>
<classpath>
<platform>
<filter>
<property>
<config>
<properties>
<plugin>
<help>
<metrics>
<metric>
<actions>
<classpath>
<script>
<server>
<filter>
<property>
<config>
<option>
<properties>
<plugin>
<help>
<metrics>
<metric>
<actions>
<scan>
<service>
<filter>
<property>
<config>
<option>
<properties>
<plugin>
<help>
<metrics>
<metric>
<actions>
<server>
....
<service>
....
<service>
.....

Elements by "Purpose"	Element Description and Usage
User-configurable resource data • `<option>` • `<config>`	`<option>` specifies a resource attribute whose value must be supplied by the user, or is otherwise supplied (in the descriptor or by a plugin class), but must be editable by a user. Resource data defined as an `<option>` is presented in the **Configuration Properties** page for a resource instance. You can define the allowable values for a selector list, whether it is optional or required, and so on. `<config>` is a (required) container element for `<option>` elements. An `<option>` element must be the child of a `<config>` element. A named `<config>` element can is a reusable building block - it can be included by reference in other `<config>` elements. This is useful when you define a group of

Elements by "Purpose"	Element Description and Usage
	options that apply to multiple managed resources in the resource plugin. A `<config>` element can be designated as "global", in which case `<config>` elements in other plugin descriptors can references it as well.
Non-configurable resource data • `<property>` • `<properties>`	`<property>` specifies a non-configurable resource attribute, whose value might be discovered (e.g., RAM, or CPU speed for a platform), returned by a plugin class, or defined in the descriptor. Resource data defined as a `<property>` cannot be entered or edited in the HQ user interface. `<properties>` is a container element for one or more `<property>` elements. `<property>` elements contained by a `<properties>` element are presented at the top of the **Browse** page for a resource instance.
Management functions for a resource type: • `<plugin>` • `<actions>`	`<plugin>` specifies a management function (auto-discovery, measurement, control, log tracking, and so on) for a resource type, and the Java class - either a custom class or an HQ support class - that performs that function. Each management function for a resource type is specified in a separate `<plugin>` element. `<actions>` specifies a list of control operations, supported by the resource type, that the plugin can perform. The `<actions>` element is required (as a sibling) for a `<plugin>` element of type "control".
Metrics for a resource type: • `<metric>` • `<metrics>`	`<metric>` specifies a measurement the plugin obtains for a resource type; the attributes in `<metric>` define the type of metric (availability, throughput, utiiization), units of measure, whether the metric is an "Indicator", and so on. `<metrics>` is container for one or more `<metric>` elements. A named `<metrics>` element in the root of the plugin is a reusable building block - it can be included by reference in `<metrics>` elements in multiple resource elements within the descriptor. This is useful when you define a set of metrics that apply to multiple resource types managed by the plugin.
Variable for use within descriptor. • `<filter>`	`<filter>` element defines a variable - a name and value pair - that can be used in the descriptor. `<filter>` is meaningful only within the descriptor - not for data that a plugin class needs or provides in a `<filter>` element. The purpose of the `<filter>` element is to make a descriptor easier to write, understand, debug, and maintain. Typically, `<filter>` is used to make it easier to define the template for each metric.

3.1.4. First Facts About Metric Templates

Every metric a plugin collects has a *metric template* that expresses a request for a specific metric, for a specific resource, in a format that the HQ Agent understands. A metric template takes this form:

Domain
:
Properties
:
Metric:Connection

A metric template provides the information a measurement class needs to obtain a metric. That information includes resource attributes (connection data, resource type and name, and so on) and metric attributes (category, units of measure, whether it is an indicator, and so on). Those resource and metric attributes are defined in <option>, <property>, and <metric> elements for a resource type. A metric template codifies this data for a particular metric as structured "metric request" that the HQ Agent can fulfill.

An HQ plugin that manages a server-service hierarchy often collects over a hundred unique metrics. To ease the process of defining metric templates, plugin developers typically define the template in terms of variables that return the values of relevant <option>, <property> and <metric> attributes.

For more information see Section 3.2.2, “Understanding Metric Templates”.

3.1.5. Inheritance and Reuse in Plugin Descriptors

In an HQ plugin descriptor, variable values can be inherited, and certain element types can be defined at one level and inherited or included by reference in lower level resource elements.

For example, the root of an HQ plugin descriptor may contain a `<metrics>` element that defines metrics common to multiple resource types managed by plugin. Then, the `<metrics>` element in each resource to which those common metrics apply, can include them by reference to `<metrics>` element in the desscriptor root.

Inheritance and reuse behaviors are useful to the plugin developer but may confuse the new plugin developer taking a first look at the descriptors for HQ's built-in plugins.

For readers new to HQ plugins, the thing to understand is that a lot of information that applies to a particular resource type in a descriptor file can be defined in the root of the file, and included by reference in multiple resource elements.

3.2. Plugin Descriptor Tips and Techniques

The value defined in a plugin descriptor for a `<property>` element can be overridden for resource instances of that type on a particular platform by adding a property of this form to the `agent.properties` file for the HQ Agent on the platform:

ResourceTypeName.PropertyName=Value

where:

- ResourceTypeName
  - is the `name` attribute and the `version` attribute (if defined) for the resource type in the plugin descriptor, with each space escaped with a backslash character.

- PropertyName
  - is the `name` attribute from a `<property>` defined for
  ResourceTypeName

.

- Value
  - is the value that will override the value specified in the descriptor for the `<property>` element's `value` attribute.

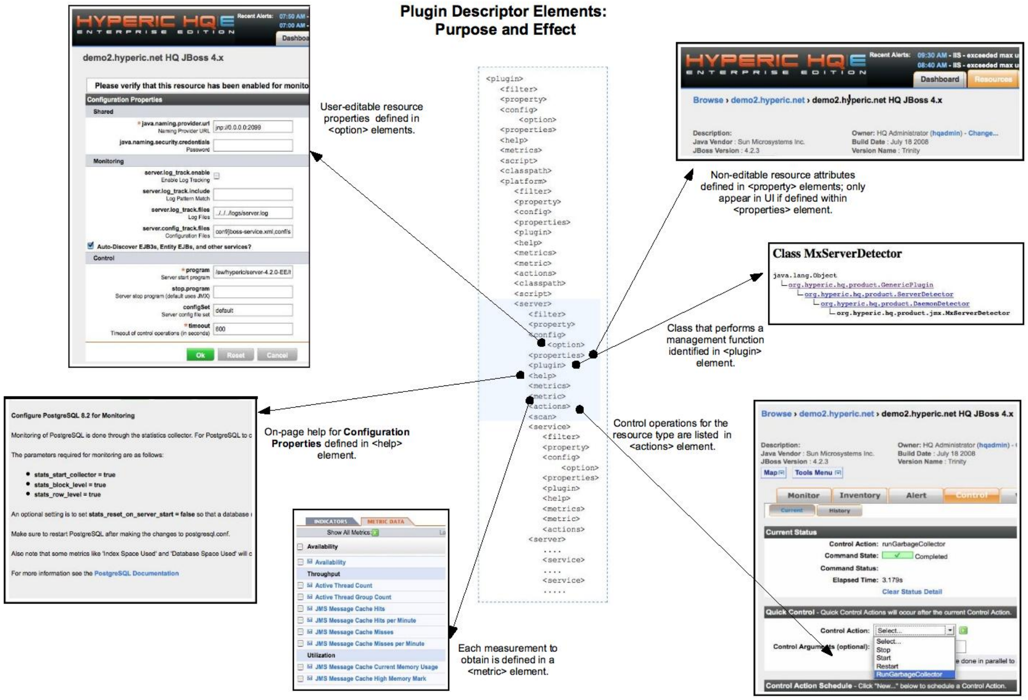
For example, adding this line to `agent.properties`:

```
Apache\ Tomcat\ 5.5.PROC_HOME_PROPERTY=catalina.base
```

overrides, for Apache Tomcat 5.5 instances on that platform, the `PROC_HOME_PROPERTY` value set in this excerpt for the Apache Tomcat plugin descriptor:

```
<server name="Apache Tomcat"
          version="5.5">
           .......
           .......

           <property name="PROC_HOME_PROPERTY"
               value="catalina.base-DISABLED"/>
.....
```

3.2.1. A Graphical View of Descriptor Elements and Functions


Plugin Descriptor Elements:
Purpose and Effect
User-editable resource properties defined in <option> elements.
Non-editable resource attributes defined in <property> elements; only appear in UI if defined within <properties> element.
Class MxServerDetector
Class that performs a management function identified in <plugin> element.
On-page help for Configuration Properties defined in <help> element.
Control operations for the resource type are listed in <actions> element.
Each measurement to obtain is defined in a <metric> element.
<plugin>
<filter>
<property>
<config>
<option>
<properties>
<help>
<metrics>
<script>
<classpath>
<platform>
<filter>
<property>
<config>
<properties>
<plugin>
<help>
<metrics>
<metric>
<actions>
<classpath>
<script>
<server>
<filter>
<property>
<config>
<option>
<properties>
<plugin>
<help>
<metrics>
<metric>
<actions>
<scan>
<service>
<filter>
<property>
<config>
<option>
<properties>
<plugin>
<help>
<metrics>
<metric>
<actions>
<server>
....
<service>
....
<service>
.....

3.2.2. Understanding Metric Templates

*Topics marked with***relate to features available only in vFabric Hyperic.*

This page contains information that has not been reviewed. Click **Add Comment** at the bottom of the page to enter suggestions or corrections.

Introduction to Metric Templates

This page is a high level introduction to metric templates, their purpose, and how they are defined. Because the content of a metric template varies considerably depending on how a metric is collected, this introduction is general. For information about metric templates for each collection method, see Metric Template Rules for Each Collection Method.

Anatomy of a Metric Template

A *metric template* expresses a request for a specific metric, for a specific resource, in a format that the HQ Agent understands. It identifies the resource instance, a particular metric, and where to get the metric value. A metric template takes this form:

Domain
:
Properties
:
Metric:Connection

The content of each segment of the metric template depends on how the metric is obtained - from an MBean server, SIGAR, an HQ measurement class, via SNMP, and so on.

- **Domain** - Specifies the management facility that collects the metric. For a JMX metric,
  Domain
  is a JMX domain.
  Domain
  may also take values that specify that the metric is collected via SIGAR, SNMP, or the script processing service. For metrics collected by a Java measurement class using a vendor-specific API,
  Domain
  is not a meaningful segment of a metric template - it has an arbitrary value.

- **Properties** - Specifies properties that identify the resource for which to collect the metric. For a JMX metric, *Properties* contains one or more key=value pairs that identify an Mbean instance of the type specified by the
  Domain
  . For other collection methods, *Properties* might identify a specific resource by its type and name, or a SIGAR query to run.

- **Metric** - Specifies the metric to collect. For a JMX metric, *Metric* is an attribute of the Mbean specified by the *Domain* and *Properties* segments of the metric template. For other collection methods, *Metric* is the name by which the measurement class or script makes the metric available.

- **Connection** - Specifies connection properties for the managed resource, for example, JNDI naming properties or JDBC connection properties.

What Does a Metric Template Look Like?

Here is an actual metric template, for the "Sequential Scans" metric for a PostgreSQL database table:
postgresql:Type=Table,table=eam_action:seq_scan:jdbcUrl= jdbc:postgresql://localhost:9432/hqdb,jdbcUser=hqadmin,jdbcPassword=******
The table below breaks down the metric template and describes each segment.

Segment	Value	What it Does
Domain	postgresql	Nothing - in HQ database plugins the Domain segment is a dummy string.

Segment	Value	What it Does
Properties	Type=Table,table=eam_action	The resource for which to obtain the metric - the "eam_action" database table.
Metric	seq_scan	Name of the metric, as the measurement class makes it available - "seq_scan".
Connection	jdbcUrl=jdbc:postgresql://localhost:9432/hqdb,jdbcUser=hqadmin,jdbcPassword=*****	The database URL and database user credentials the HQ Agent uses to connect to PostgreSQL.

Where the Data for a Metric Template Comes From

When you design a plugin, you determine the appropriate collection method for the resource types it manages, the requirements for accessing or connecting to each type, and the measurements you want to collect. In the plugin descriptor you define these resource and measurement attributes using <option>, <property> and <metric> elements. These data items map to fields that a plugin class either needs (the value is set in descriptor or supplied by user) or discovers.

A metric template codifies connection data, resource properties, and metric attributes as a "metric request" that the HQ Agent can fulfill.

How a Metric's template Attribute is Specified

A <metric> element's `template` attribute specifies the first three segments of a metric template -
Domain
:
Properties
:
Metric
. (The properties for the
Connection
segment are defined in the plugin descriptor, but do not form a portion of the `template` attribute; the
Connection
segment is appended to the `template` attribute when a metric value is requested.)

HQ plugin descriptors typically use variables for resource properties and options in the `template` attribute to make the descriptor easier to write, maintain, and read. Strictly speaking, the `template` attribute, or certainly parts of it, can be defined explicitly, as in this <metric> element for the "Sequential Scans" metric:

```
<metric name="Sequential Scans" alias="seq_scan"
template="postgresql:Type=Table,table=eam_action:seq_scan" category="UTILIZATION" indicator="true"
units="none" collectionType="trendsup"/>
```

More typically, `template` is defined as a variable expression - this is easy to do, as the value of each <option> and <property> is available from a variable of the same name. As useful, a plugin developer can also use <filter> elements to define local variables that are meaningful only within the descriptor.

Below, the `template` attribute is simplified by expressing
Domain
by reference to an local variable, and the `table` property by reference to the `%table%` variable that returns the name attribute for the <option> element that defines it, "table".

```
template="${domain}:Type=Table,table=%table%:seq_scan
```

The use of variables in HQ plugin descriptors is widespread, and, well - variable. Peruse a few HQ plugin descriptors, and you will note <metric> elements that do not contain a `template` attribute at all; instead, a local variable named `template` defines the whole `template` attribute in terms of variables that return attributes from resource-specific elements. Instead of defining `template` explicitly in each <metric> element, a variable is defined once, outside and before the <metric> elements for the resource type.

For example, the `template` variable below defines the `template` attribute for all metrics for all tables. This is similar to the previous definition, but in this case, the *Metric* segment of the `template` attribute for a metric is supplied by the <metric> element's `alias` attribute.

```
<filter name="template" value="template="${domain}:Type=Table,table=%table%:${alias}"/>
```

which expands to:

```
template=postgresql:Type=Table,table=eam_action:seq_scan
```

When the *Connection* segment is appended to the `template` attribute value, the metric template is complete:

```
template=postgresql:Type=Table,table=eam_action:seq_scan:jdbcUrl= jdbc:postgresql://localhost:9432/hqdb,jdbcUser=hqadmin,jdbcPassword=******
```

Along with other <metric> attributes (`category` and `indicator`, for instance), the metric template supplies the values of fields in the `Metric` object that is passed to the measurement class when the metric value is requested.

Learn More About Variables in Plugin Descriptors

This page briefly introduced the use of variables to define metric templates. Using variables is not required, but it simplifies the process of writing a plugin descriptor. For more information, see Section 3.2.5, “Variables in Plugin Descriptors”.

Metric Template Rules for Each Collection Method

Because the the rules for for constructing a metric's `template` attribute vary by collection method, see the instructions and examples for the metric collection method you are using.

- Section 3.2.3, “JMX Metric Templates”
- SIGAR (links TBS)
- JDBC (links TBS)
- script (links TBS)
- SQL (links TBS)
- JMX (links TBS)
- network collectors (links TBS)
- SNMP
- vendor API

3.2.3. JMX Metric Templates

*Topics marked with*relate to features available only in vFabric Hyperic.*

This page contains information that has not been reviewed. Click **Add Comment** at the bottom of the page to enter suggestions or corrections.

Components of the Template for a JMX Metric

A metric template is a structured expression with four segments:

Domain
:
Properties
:
Metric:Connection

Each <metric> element's `template` attribute specifies the first three segments of a metric template - Domain
:
Properties
:
Metric
. (The properties for the
Connection
segment are defined in the plugin descriptor, but do not form a portion of the `template` attribute; the
Connection
segment is appended to the `template` attribute when a metric value is requested.)

The table below describes the content of a metric template for a JMX metric.

Template Component	**Description**	**Notes**
Domain	JMX domain	In a metric template for a JMX metric, the Domain segment is the JMX domain that contains that MBean from which the metric is obtained. Given that the JMX domain is the same for a server type and its child service types, you can define a "Domain" `<property>` in the `<server>` element and reference it in the `template` attribute for metrics. `${Domain}` returns the value of a "Domain" property.
Properties	Comma-separated list of name-value pairs that provide the MBean properties that uniquely identify an Mbean in the Domain. `property1=value,propertyN=value`	The property=value pairs supplied here, appended to the Domain value, form the JMX ObjectName for the MBean from which the metric is obtained. In most HQ plugins, the key properties for a JMX metric are defined as a `<property>` ("OBJECT_NAME" in the example below) which each resource element defines uniquely. Key property values can be defined explicitly in the `template` at-

Template Component	Description	Notes
		tribute, supplied by a plugin class, or as resource-specific configuration `<option>`.
Metric	The MBean attribute that supplies the metric value.	Typically, the the attribute name is defined in each `<metric>` element's `alias` attribute, and is referenced in the `template` as `${alias}`.
Connection	JMX URL and credentials.	Defined in `<option>` elements to appear in the "Shared" section of a resource's **Configuration Properties** page. Appended by plugin class to Domain:Properties:Metric when the value of the metric is requested.

template Attribute for a JMX Metric

Defined explicitly (without variables), the `template` attribute for a JMX metric is formed like this:

```
template=JmxDomain:property1=value,propertyN=value:MbeanAttribute
```

In practice, the the `template` attribute for a JMX metric is expressed using variables, for example:

```
template="${OBJECT_NAME}:${alias}"
```

Example Definition of JMX Metric Template

In HQ's apache tomcat plugin descriptor, portions of which are excerpted below, metric templates are defined in this fashion:

- **JMX Domain** - The JMX domain is defined in line 94 in a `<property>` element named "domain" below the `<server>` element for Tomcat 5.5. This value is inherited by each child `<service>`.
- **Key Properties** - Each `<service>` defines a `<property>` named "OBJECT_NAME", like the one on line 365 for the service type "Cache". "OBJECT_NAME" defines the JMX ObjectName for the target MBean. The value of the "domain" property is returned by the `${domain}` variable; the value of the "type" property (the Mbean type) is explicitly defined as "Cache"; the values of "host" and "path", provided by the `MxServerDetector` class, are referenced in this form: `host=*,path=*`.
- **MBean attribute** - Each `metric` element's `alias` attribute is set to the name of an MBean attribute.
- **Complete `template` attribute** - Each `metric` element's `template` is defined as `template="${OBJECT_NAME}:${alias}"`

```
86 <server name="Apache Tomcat"
87 version="5.5">
88
89 <property name="VERSION_FILE"
90 value="server/lib/catalina-storeconfig.jar"/>
91
92 <plugin type="autoinventory"
93 class="org.hyperic.hq.product.jmx.MxServerDetector"/>
94 <property name="domain"
95 value="Catalina"/>
```

```
...
...
...
<service name="Cache">
<property name="OBJECT_NAME"
value="${domain}:type=Cache,host=*,path=*"/>

<plugin type="autoinventory"/>

<!-- listen for JMX notifications -->
<plugin type="log_track"
class="org.hyperic.hq.product.jmx.MxNotificationPlugin"/>

<config>
<option name="path"
description="Context Path of Deployed Application"
default=""/>
<option name="host"
description="Hostname"
default=""/>
description="Associated Java Class"
default=""/>
</config>
...
...
...
<metric name="Access Count"
alias="accessCount"
indicator="true"
template="${OBJECT_NAME}:${alias}"
collectionType="trendsup"
units="none"/>
```

3.2.4. Using Global Configuration Schemas

*Topics marked with*relate to features available only in vFabric Hyperic.*

Available only in vFabric Hyperic

Globally Available Resource Configurations

Configuration data about resources are defined using `<option>` elements contained by a `<config>` element. If a `<config>` element's type attribute is "global", and its name attribute is defined, other plugins can reference it.

This excerpt from HQ's netservices-plugin descriptor defines a globally available configuration schema named "snmp":
<config name="snmp" type="global"> <option name="snmpIp" description="SNMP agent IP address" default="127.0.0.1"/> <option name="snmpPort" description="SNMP agent port" type="port" default="161"/> <option name="snmpVersion" description="SNMP Version" type="enum"> <... </option>

Include a Global Configuration Schema by Reference

This excerpt from HQ's netscaler-plugin, the <platform> element includes the globally available configuration schema named "snmp" by referencing it.
<platform name="NetScaler"> <config include="snmp"/>

Variables for Returning Formatted Config Schemas

See Variables and Global Configs.

Including a Descriptor in Another

If you have a set of <config>, <property>, or <metric> elements that are common to multiple managed resources, you can simplify your plugin descriptor by specifying the common elements in a external entity and include the fragment in the descriptor by reference.
For example, process-metrics.xml defines these metrics:
<metric name="Process Virtual Memory Size" template="sigar:Type=ProcMem,Arg=%process.query%:Size" units="B"/> <metric name="Process Resident Memory Size" template="sigar:Type=ProcMem,Arg=%process.query%:Resident" indicator="true" units="B"/> <metric name="Process Page Faults" template="sigar:Type=ProcMem,Arg=%process.query%:PageFaults" collectionType="trendsup"/> <metric name="Process Cpu System Time" template="sigar:Type=ProcCpu,Arg=%process.query%:Sys" units="ms" collectionType="trendsup"/> <metric name="Process Cpu User Time" template="sigar:Type=ProcCpu,Arg=%process.query%:User" units="ms" collectionType="trendsup"/> <metric name="Process Cpu Total Time" template="sigar:Type=ProcCpu,Arg=%process.query%:Total" units="ms" collectionType="trendsup"/> <metric name="Process Cpu Usage" template="sigar:Type=ProcCpu,Arg=%process.query%:Percent" indicator="true" units="percentage"/> <metric name="Process Start Time" template="sigar:Type=ProcTime,Arg=%process.query%:StartTime" category="AVAILABILITY" units="epoch-millis" collectionType="static"/> <metric name="Process Open File Descriptors" template="sigar:Type=ProcFd,Arg=%process.query%:Total"/> <metric name="Process Threads" template="sigar:Type=ProcState,Arg=%process.query%:Threads"/>
process-metrics.xml is included by reference in a number of HQ plugin descriptors, for instance the tomcat-plugin and the active-mq-plugin.

Declare external entities in an <!ENTITY> element below the!DOCTYPE declaration at the the beginning of the descriptor. This excerpt from the tomcat-plugin declares the process-metrics.xml entity:
<!DOCTYPE plugin [<!ENTITY process-metrics SYSTEM "/pdk/plugins/process-metrics.xml">]>
This <service> element includes the entity
<service name="Java Process Metrics"> <config> <option name="process.query" default="%ptql%" description="PTQL for Tomcat Java Process"/> </config> <metric name="Availability" template="sigar:Type=ProcState,Arg=%process.query%:State" indicator="true"/> &process-metrics; </service>
This <service> element in the tomcat-plugin descriptor includes the process-metrics entity:
<service name="Java Process Metrics"> <config> <option name="process.query" default="%ptql%" description="PTQL for Tomcat Java Process"/> </config> <metric name="Availability" template="sigar:Type=ProcState,Arg=%process.query%:State" indicator="true"/> &process-metrics; </service>

3.2.5. Variables in Plugin Descriptors

*Topics marked with***relate to features available only in vFabric Hyperic.*

Variables Simplify Metric Template

Typically, the `template` attribute for a metric is derived from a variable expression rather than explicitly specified.

*Note: * For more information about metric templates and the components thereof, see the Section 3.3.6, “metric” page, and the examples in the template section on that page.

A metric expression can be defined in terms of elements and attributes that are available as variables:

- `<option>`

- `<property>`

- `<metric>` attributes
- `<filter>`

The scope of a variable and the syntax for referencing is vary by type.

<option>

Any `<option>` element is available as a variable. Template expressions usually reference one or more `option` elements to supply resource-specific connection or location information for a resource.

Scope - You can reference an `<option>` as a variable within the resource element that defines or references it. Data that is defined as a `<option>` is available to plugin classes, whether its value is discovered by the plugin, specified within the plugin descriptor, or defined by a user on a resource's **Configuration Properties** page.

Syntax - The syntax for referencing an `<option>` variable is:

```
%OptionName%
```

For example, the value of an `<option>` whose `name` attribute is "naming.url" can be obtained with `%naming.url%`.

Examples - For an example of a template expression using `<option>` variables, see template for a metric obtained via a vendor APIon the Section 3.3.6, “metric” page.

<metric> Attributes

Any `<metric>` attribute is available as a variable; in practice, `alias` is the one most commonly used - to express the "metric" portion of a metric template. The `alias` attribute for a metric matches the name by which the class that collects the metric makes it available.

Scope - You can reference an attribute of a `<metric>` element as a variable within the `<metric>` element that defines it.

Syntax - The syntax for referencing an `<metric>` attribute variable is:

```
%attribute%
```

For example:

```
%alias%
```

<property>

Any `<property>` element is available as a variable.

> The `<property>` element is analogous to the `<option>` element - each define a resource attribute. The main difference is: data defined as a `<property>` is not configurable by the user, whereas data defined as an `<option>` appears, and can be edited by an authorized user, on the **Configuration Properties** page for a resource.
>
> Note also that you can override the descriptor-defined value for a `<property>` for resource instances on a particular platform. See Overriding the Value of a Resource `property` at Platform Level [74].

Scope - You can reference a `<property>` as variable in the resource element that defines it, and in descendant resource elements. A lower level element may override an inherited property value by locally defining a `<property>` element of the same name. Data that is defined as a `<property>` is available to plugin classes, whether its value is discovered by the plugin or specified within the plugin descriptor.

Syntax - To reference a `<property>`, use

${PropertyName
}, where
PropertyName
is the name of the property.

Examples - For more information, see the Section 3.3.13, “property” page and the <property> defines OBJECT_NAME for use in plugin descriptor and class example on that page.

<filter>

The `<filter>` element defines a variable that descendant resource elements can reference. For example, a `<filter>` declared in a `<platform>` element is inherited by the `<platform>`'s child `<server>` elements, unless you define the same filter with a different value at the server level.

A `<filter>` element can specify a value explicitly, or in terms of another variable, specifically:

- other `<filter>` elements
- `<option>` element
- `<metric>` attribute

The `<filter>` element is often used to define the `template` attribute as an expression that uses other variables. A `<filter>` element might also define a component of a metric template.

Scope - The value of a `<filter>` variable is available only within the plugin descriptor. A filter is inherited by all of the elements below the one that defines it. A lower level element may override an inherited filter value by defining a `<filter>` element of the same name.

Syntax - The syntax for referencing an `<filter>` variable is:

%FilterName%

For example, the value of an `<filter>` whose `name` attribute is "template" can be obtained with `%template%`.

Examples - For more information see the Section 3.3.2, “filter” page, which provides usage examples including:

- <filter> defines metric template using a values supplied by a <property> and a <metric> alias attribute
- <filter> uses value of config <option> to provide "domain" portion of metric template

Variables and Global Configs

> This advanced technique is used in some HQ plugin descriptors and is documented here for the curious reader who has encountered the usage. The technique is non-essential, may not be applicable to most custom plugin developers.

You can use special variables to return all of the options defined in a globally available `<config>` in format that is useful for constructing a metric's `template` attribute.

${ configname .config} Returns Option Name:Value Pairs

HQ's netservices plugin defines a globally available configuration schema named "url" that includes these options:

```
<option name="hostname" description="Hostname" default="localhost"/> <option name="port"
description="Port" type="port"/> <option name="sotimeout" description="Socket Timeout (in seconds)"
default="10" type="int"/> <option name="ssl" description="Use SSL" type="boolean" optional="true"
default="false"/>
```

This excerpt from the HQ's iplanet plugin references `${url.config}` which returns the options in format that is useful the as the middle component of a metric template.

```
<metric name="Availability" alias="Availability" template="%protocol%:${url.config}:${alias}"
category="AVAILABILITY" group="Reliability" indicator="true" units="percentage"
collectionType="dynamic"/>
```

The variable ${url.config} expands to:

```
hostname=,port=,sotimeout=,ssl=
```

Resulting in:

```
template="%protocol%:hostname=,port=,sotimeout=,ssl=:${alias}"
```

${ configname .template} Includes Domain for Metric Template

The variable

```
${configname.template
```

} returns the same results as

```
${configname.config
```

}, prepended by the name of the `<config>` and a colon.

For example, the variable `${url.template}` expands to:

```
url:hostname=,port=,sotimeout=,ssl=
```

3.3. Plugin Descriptor Element and Attribute Reference

- Section 3.3.1, “actions”
- Section 3.3.3, “classpath”
- Section 3.3.18, “config”
- Section 3.3.2, “filter”
- Section 3.3.5, “include”
- Section 3.3.6, “metric”
- Section 3.3.7, “metrics”
- Section 3.3.8, “option”
- Section 3.3.9, “platform”
- Section 3.3.10, “plugin (Platform, Server and Service)”
- Section 3.3.11, “plugin (Root)”
- Section 3.3.12, “properties”
- Section 3.3.13, “property”

3.3.1. actions

*Topics marked with*relate to features available only in vFabric Hyperic.*

This page contains information that has not been reviewed. Click **Add Comment** at the bottom of the page to enter suggestions or corrections.

Element Overview

The `<actions>` element lists the available control actions for a server type or a service type.

An action corresponds to a remote operation that the resource type supports, for example, a JMX operation, a JDBC operation, a Windows Service Manager command, or an action performed by a custom script.

The `<actions>` element must have as siblings:

- `<plugin>` element of type "control"
- `<config>` element that defines the resource data needed to perform the action.

Parent Elements

An `<actions>` element can be a child of the following element types:

- `<server>` - Defined actions apply to instances of the server type.
- `<service>` - Defined actions apply to instances of the service type.

Child Elements and Attributes

An `<actions>` element contains these attributes and child elements:

- `include` - (Optional) Value is a comma-separated list of actions, each of which must be supported by the parent resource type.
- `<include>` - (Optional) This element can be used as an alternative way to list supported actions. That is, you can list actions in the include attribute or the `<include>` element.
- `platform` - (Optional) Value is a comma-separated list of the platform types upon which the control options are supported, each of which must be a valid HQ operating system platform type: Unix, Linux, Solaris, HPUX, AIX, MacOSX, FreeBSD, OpenBSD, NetBSD, Win32. If platform is not specified, the control actions are supported on all platforms on which the server or service runs.

<action> Examples

Specifying actions in the include attribute

This excerpt from the Glassfish plugin descriptor defines the JCA Connection Factory service type:

```
<service name="JCA Connection Factory">
    ....
    <plugin type="control"
              class="org.hyperic.hq.product.jmx.MxControlPlugin"/>
    <actions include="setConfigProperty,getConfigProperty"/>
    <config>
        <option name="J2EEApplication"
                description="J2EE Application"
                default=""/>
        <option name="JCAResource"
                description="JCA Resource"
                default=""/>
        <option name="name"
                description="Name"
                default=""/>
      </config>
```

Notes:

- The `<actions>` element specifies the supported actions using the include attribute. The actions are Mbean operations supported for the Mbean type.
- A sibling `<plugin>` element of type control specifies the class that will perform the control actions.
- A sibling `<config>` element defines the configuration options that the control class needs to communicate with a "JCA Connection Factory" instance.

Specifying actions in the <include> element

This excerpt from the VMware plugin illustrates the use of an `<include>` element to list each action, in contrast to the previous example, in which actions are specified with the include attribute.

```
<service name="VM">
    ....
    <plugin type="control"
          class="VMwareControlPlugin"/>
    <actions>
        <include name="start"/>
        <include name="stop"/>
        <include name="reset"/>
        <include name="suspend"/>
        <include name="resume"/>
        <include name="createSnapshot"/>
        <include name="revertSnapshot"/>
        <include name="deleteSnapshot"/>
        <include name="saveScreenshot"/>
    </actions>
    <config include="vm"/>
```

Limiting actions to a platform type

This excerpt from a `<server>` element in the Tomcat plugin descriptor illustrates the use of the platform attribute to limit the actions to servers running on Windows.

```
<server name="Apache Tomcat"
          version="5.5">
     .....
     <plugin type="control"
            platform="Win32"
            class="org.hyperic.hq.product.Win32ControlPlugin"/>
    <actions platform="Win32"
             include="start,stop,restart"/>
    <config type="control" platform="Win32">
          <option name="service_name"
              default="Apache Tomcat"
              description="Tomcat Service Name"/>
    </config>
```

3.3.2. filter

*Topics marked with*relate to features available only in vFabric Hyperic.*

Element Overview

The `<filter>` element declares a variable that can be referenced by descendent elements in the descriptor to obtain resource-type specific values for other expressions. The most common use for the `<filter>` element is to define a string that forms all or a portion of the `template` attribute for `<metric>` elements, instead of explicitly defining the `template` for each metric.

A `<filter>` element is available to all descendants of its parent. For example, a `<filter>` definition in the plugin root applies to all `<platform>`, `<server>` and `<service>` elements that follow. You override the filter value a child inherits from a parent by defining a filter of the same name in the child element.

Parent Elements

- `<plugin>` (root) - A `<filter>` element here is available to all resource elements in the descriptor
- `<platform>` - A `<filter>` element here is available within the element and its descendants.
- `<server>` - A `<filter>` element here is available within the element and its descendants.
- `<service>` - A `<filter>` element here is available only within the element.

Child Elements and Attributes

- `name` - (Required) Name by which the variable is referenced. The name can be an arbitrary value, or exactly match the name of an attribute the filter defines. Typically the name attribute defines a filter named "template" which defines the template attribute forelements using other variables whose values are assigned at the resource and metric level.
- `value` - (Required) Value of the variable, expressed explicitly or using other variables.

Examples

<filter> defines metric template components using configuration and metric alias

This excerpt from the HQ Zimbra plugin, is part of a `<server>` element that defines:

- a `<filter` named "template" that specifies a building block in terms of the `installpath` configuration option and the `metric` element's `alias` attribute.
- an `<option>` element that defines the `installpath` configuration option.

```
<server name="Zimbra"
     version="4.5.x">

        <plugin type="autoinventory"
         class="ZimbraServerDetector"/>
        <plugin type="measurement"
         class="org.hyperic.hq.product.MeasurementPlugin"/>
        <plugin type="collector"
         class="org.hyperic.hq.plugin.zimbra.ZimbraCollector"/>

        <filter name="template"
         value="zimbra-stats:installpath=%installpath%:${alias}"/>

        <properties>
            <property name="version"
             description="Zimbra Version"/>
        </properties>

        <config>
            <option name="installpath"
             default="/opt/zimbra"
             description="Zimbra Install Path"/>
            <option name="zimbra-ptql"
             default="Pid.PidFile.eq=%installpath%/log/zmtomcatmgr.pid"
             description="Sigar PTQL Process Query"/>
        </config>
```

<filter> defines metric template using a values supplied by a <property> and a <metric> alias attribute

This element from the HQ Resin plugin defines a filter named template that references the values of a resource named OBJECT_NAME and a element's alias attribute. This reference provides two components of the metric template:

```
<filter name="template"
value="${OBJECT_NAME}:${alias}"
/>
```

Because the filter is defined in the root `<plugin>` element, it is available to all resource elements in the descriptor.

The value of the OBJECT_NAME property is defined in each resource element in the descriptor, a `<server>` and two{{ <service>}} elements. For example:

```
<service name="Port">
<property name="OBJECT_NAME"
value="resin:type=Port,name=*"/>
```

The template filter is expanded for each server and service metric, using the value of the owning resource's OBJECT_NAME property and the metric's alias attribute.

<filter> references globally available configuration schema

This excerpt shows the use of a globally available configuration schema in a filter:

```
<filter name="iplanet.snmp"
value="iplanet:$\{snmp.config\}"/>
```

The "snmp" schema is defined in the HQ Netservices plugin descriptor, as shown in this excerpt:

```
description="SNMP agent IP address"
default="127.0.0.1"/>
description="SNMP agent port"
type="port"
default="161"/>
description="SNMP Version"
type="enum">
```

In the filter definition,

`${snmp.config}` is replaced by:

snmpIP=,
`snmpPort=*,*snmpVersion=*`

resulting in:

`iplanet:snmpIP=,snmpPort=,snmpVersion=*`

In the same plugin descriptor, the filter is referenced in a template definition:

`template="${iplanet.snmp}:httpStatisticsRequests:${server.config.v4}"`

<filter> uses value of config <option> to provide "domain" portion of metric template

The HQ mssql-plugin descriptor defines several filters in the root, based on configuration options:

```
<plugin name="mssql">
   ......
 ......
  <filter name="db.domain"
          value="Databases(${db.name})"/>
  <filter name="lock.domain"
          value="Locks(${lock.name})"/>
  <filter name="cache.domain"
          value="Cache Manager(${cache.name})"/>
  .....
  .....
```

Also in the root, `<metric>` elements, define the `template` in terms of the filter:

```
<metrics name="mssql-avail">
    <metric name="Availability"
            alias="Availability"
            template="${db.domain}:Type=Availability:Active Transactions"
            category="AVAILABILITY"
            group="Reliability"
            indicator="true"
            collectionType="dynamic"
            units="percentage"/>
  </metrics>
```

Later in the descriptor, each resource element, like this `<service>` element, defines the `<option>`:

```
<service name="Database">
        ......
        ......
      <config>
        <option name="db.name"
                description="Database name"
                default="Northwind"/>
      </config>
    </service>
```

3.3.3. classpath

*Topics marked with*relate to features available only in vFabric Hyperic.*

This page contains information that has not been reviewed. Click **Add Comment** at the bottom of the page to enter suggestions or corrections.

Element Overview

The <classpath> element identifies HQ libraries or external JARs that a plugin uses to perform auto-discovery or other plugin functions. You use this element to specify the path to JDBC drivers or JMX libraries in the HQ PDK. As appropriate, <classpath> can also specify vendor API JARs, another HQ plugin, or a particular class external to the plugin.

If a plugin does not rely on any external libraries or JARs, its descriptor does not include a <classpath> element.

You do not need to specify plugin support classes that a plugin uses in a <classpath> element. The classes in org.hyperic.hq.product are available to plugins.

Parent Elements

The <classpath> element may be the child of:

- the root <plugin> element

Child Elements and Attributes

The <classpath> element has one child element:

- <include> (Required) Specifies the path to a JAR or directory, relative to the AgentHome/bundles/AgentBundleDir in HQ 4.0 or later. Exception: If the path starts with server/default or server/all, the path is resolved relative to ServerHome/hq-engine on the HQ server.

<classpath> Examples

<classpath> specifies path to JDBC drivers

HQ plugins that manage databases use the <classpath> element to specify the path to the appropriate database drivers to the classpath, as shown in this example:

```
<plugin>
  <!-- include all drivers from this directory -->
  <classpath>
    <include name="pdk/lib/jdbc"/>
  </classpath>
```

<classpath> specifies path to a JMX library

HQ plugins that obtain metrics from an MBean server use the <classpath> element to specify the path to the JMX utilities:

```
<classpath>
        <include name="pdk/lib/mx4j"/>
</classpath>
```

<classpath> specifies path to another plugin JAR file

This excerpt from the HQ Drupal plugin specifies the path and name to the HQ sqlquery plugin JAR file:

```
<plugin>
  <classpath>
    <!-- for SQLQueryDetector --> ConfigName
    <include name="pdk/plugins/sqlquery-plugin.jar"/>
  </classpath>
```

<classpath> specifies path to multiple vendor jars

This excerpt from the WebLogic Server plugin specifies the WLS jars that the plugin uses.

```
<plugin name="weblogic" class="WeblogicProductPlugin">
  <classpath>
    <include name="pdk/lib/mx4j/hq-jmx.jar"/>
    <include name="server/lib/weblogic_sp.jar"/>
    <include name="server/lib/weblogic.jar"/>
    <include name="server/lib/wlcipher.jar"/>
    <include name="server/lib/webservices.jar"/>
    <\!-\- 9.x+ssl \-->
    <include name="server/lib/jsafe.jar"/>
    <\!-\- 6.1 \-->
    <include name="lib/weblogic.jar"/>
  </classpath>
```

<classpath> specifies jars in HQ Server installation

In this `<classpath>` element from the JBoss plugin descriptor, the paths to the JARs on the HQ Server (shown in bold) will be resolved relative to `ServerHome/hq-engine` on the HQ server. The other paths specified are resolved related to `AgentHome/bundles/AgentBundleDir`.

```
<classpath>
  <include name="pdk/lib/mx4j/hq-jmx.jar"/>
  <include name="client/jnp-client.jar"/>
  <include name="client/jboss-common-client.jar"/>
  <include name="client/jboss-jsr77-client.jar"/>
  <include name="client/jbossall-client.jar"/>
  <include name="client/log4j.jar"/>
  <include name="client/jmx-rmi-connector-client.jar"/>
  <include name="lib/jboss-system.jar"/>
  <include name="lib/jboss-jmx.jar"/>
  <include name="lib/jboss-management.jar"/>
  <include name="lib/dom4j.jar"/>
  <\!-\- used underneath javax.management.Query.match() \-->
  <include name="lib/gnu-regexp.jar"/>
  <\!-\- required for MainDeployer.listDeployed() \-->
  <include name="lib/endorsed/xercesImpl.jar"/>
  <include name="lib/endorsed/xml-apis.jar"/>
  <include name="server/default/lib/jboss-management.jar"/>
  <include name="server/all/lib/jboss-management.jar"/>
  <\!-\- for jndi authentication \-->
  <include name="server/default/lib/jbosssx.jar"/>
  <include name="server/all/lib/jbosssx.jar"/>
  < \!-\- relative to $installpath incase server/{default,all}do not exist \-->
  <include name="lib/jboss-management.jar"/>
  <include name="lib/jbosssx.jar"/>
</classpath>
```

3.3.4. help

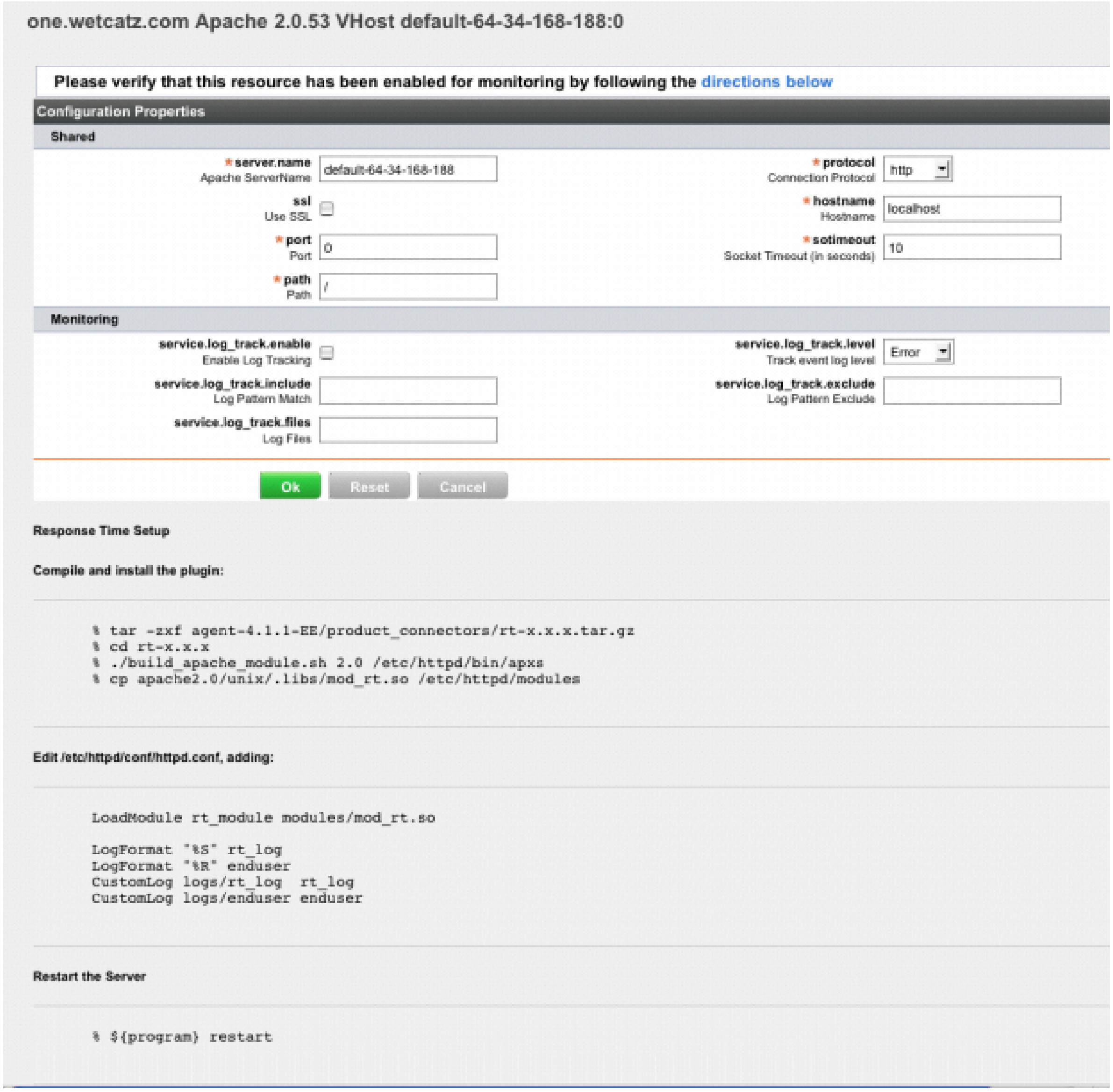
*Topics marked with***relate to features available only in vFabric Hyperic.*

Element Overview

The `<help>` element is used to:

- to define a named block of HTML, and

- to associate a named `<help>` element by reference to its name with a specific resource type. The referenced help content will be included on the **Configuration Properties** page for instances of that resource type, as shown in the bottom portion of the page below.



one.wetcatz.com Apache 2.0.53 VHost default-64-34-168-188:0

Please verify that this resource has been enabled for monitoring by following the directions below

Configuration Properties

Shared

* server.name Apache ServerName	default-64-34-168-188	* protocol Connection Protocol	http
ssl Use SSL	□	* hostname Hostname	localhost
* port Port	0	* sotimeout Socket Timeout (in seconds)	10
* path Path	/		

Monitoring

service.log_track.enable Enable Log Tracking	□	service.log_track.level Track event log level	Error
service.log_track.include Log Pattern Match		service.log_track.exclude Log Pattern Exclude	
service.log_track.files Log Files			

Response Time Setup

Compile and install the plugin:

```
% tar -zxf agent-4.1.1-EE/product_connectors/rt-x.x.x.tar.gz
% cd rt-x.x.x
% ./build_apache_module.sh 2.0 /etc/httpd/bin/apxs
% cp apache2.0/unix/.libs/mod_rt.so /etc/httpd/modules
```

Edit /etc/httpd/conf/httpd.conf, adding:

```
LoadModule rt_module modules/mod_rt.so

LogFormat "%S" rt_log
LogFormat "%R" enduser
CustomLog logs/rt_log  rt_log
CustomLog logs/enduser enduser
```

Restart the Server

```
% ${program} restart
```

Parentage

The `<help>` element can be the child of:

- <plugin> (root) - In the root of a plugin a `<help>` element can:
  - Defines a named block of HTML that can be associated by reference with instances of the server types or service types that the plugin manages. Note that the content defined in `<help>` element in the descriptor root will only appear as help if another `<help>` element associates with a resource type.

  - Associate a named block of HTML with instances specific resource type the plugin manages.

- <server> or <service> - Within a resource element, a `<help>` element define an unnamed block of HTML that will be presented as help on the Configuration Properties page for all instances of the resource type.

Contents

- name - This attribute assigns a name to the `<help>` element, by which it can be referenced to associate it with a resource type. In HQ plugins, named `<help>` element are defined in the the root of the plugin descriptor.
- HTML content - The text formatted as HTML within a `<help>` element will be presented on **Configuration Properties** page for instances of resources types the element is associated with.
- include - Used to specify the name attribute of a `<help>` element to include in the **Configuration Properties** page for instances of resources types the element is associated with. Help content pointed to by the include attribute will appear before any HTML content contained in the body of the `<help>` element.
- append - Similar to the include attribute, append is used to specify the name attribute of a `<help>` element to include in the **Configuration Properties** page for instances of resources types the element is associated with. Help content pointed to by the attribute attribute will appear after any HTML content contained in the body of the `<help>` element.

Examples

<help> element defines a named block of HTML

This `<help>` element in the root of HQ's apache-plugin specifies a block of HTML and names it "restart-server". Another `<help>` element in the descriptor can include this content by setting its include or append attribute to "restart-server". (The only functional difference between include or append is relates to sequence in which the referenced content is presented.)

```
<help name="restart-server">
<![CDATA[
<p><h4>Restart the Server</h4></p>
<pre>
% $\{program\} restart
</pre>
]]>
</help>
```

Note that because the `<help>` element is defined in the root of the plugin, it must be referenced in a resource element later in the plugin. Otherwise the help content will not appear on any Configuration Properties page.

<help> element defines help for a service type explicitly and by reference with append attribute

This `<help>` element in the root of HQ's apache-plugin defines the help for services whose type is "Apache 1.3 VHost". The HTML within the body of the element will be presented first. The content of the `<help>` element named "restart-server" will appear after the HTML defined in the body, because it is referenced with the append attribute. (See the screen shot on the previous page.)

```
<help name="Apache 1.3 VHost" append="restart-server">
<![CDATA[
<p>
<h3>Response Time Setup</h3>
<h4>Compile and install the plugin:</h4>
</p>
<pre>
% tar zxf agent$\{HQVersion\}/product_connectors/rt-$\{rtVersion\}.tar.gz
% cd rt-$\{rtVersion\}
% ./build_apache_module.sh 1.3 $\{installpath\}/bin/apxs
% cp apache1.3/unix/mod_rt.so $\{installpath\}/libexec
</pre>
<p><h4>Edit $\{installpath\}/conf/httpd.conf, adding:</h4></p>
<pre>
LoadModule rt_module libexec/mod_rt.so
RtLog logs/rt_log
EndUserLog logs/enduser
</pre>
<p>
<h4>If the ClearModuleList directive is in your config file,
you will need to add:</h4>
</p>
<pre>
AddModule mod_rt.c
</pre>
]]>
</help>
```

<help> element defines help for a service type by reference with append and include attributes

This `<help>` element defines the help for services whose type is "MySQL Process 5.x". The several bits of HTML will be ordered in this way:

- The content of the `<help>` element named "MySQL 5.x" will appear prior to any HTML in the body, because it is referenced using the include attribute.
- The HTML included in the body of the element will appear next.
- The content of the `<help>` element named "mysql-process" will appear after any HTML in the body, because it is referenced using the append attribute.

```
<help name="MySQL Process 5.x" include="MySQL 5.x" append="mysql-process"/>
<\!\[CDATA\[
<p><h4>If you want something done right</h4></p>
<pre>
Do it yourself\!
</pre>
\]\]>
```

<help> element references a named block of HTML, by virtue of element location

This excerpt from the bind-plugin illustrates a different method of associating a named block of HTML to a specific version of a resource type. In this example, the referencing `<help>` element is located in the <server> element that defines the defines Bind 9.x server type. In this usage you do not need to use the name attribute to define the resource type version explicitly.

```
<server name="Bind"
        version="9.x"
        platforms="Unix">
  <plugin type="measurement"
          class="BindMeasurementPlugin"/>
  <plugin type="autoinventory"
          class="BindServerDetector"/>
  <config>
    <option name="rndc"
            description="Path to rndc"
            default="/usr/sbin/rndc"/>
    <option name="named.stats"
            description="Path to named.stats"
            default="/var/named/named.stats"/>
    <option name="process.query"
            description="Process query for named"
            default="State.Name.eq=named"/>
  </config>
  <metrics include="process,rndc"/>
  <help include="rndc"/>
</server>
```

<help> element references 2 named blocks of HTML using include and append attributes

```
<help name="MySQL Process 3.x"
      include="MySQL 3.x"
      append="mysql-process"/>
```

The content in the `<help>` element named "MySQL 3.x" will precede the content in the `<help>` element named "mysql-process".

3.3.5. include

*Topics marked with*relate to features available only in vFabric Hyperic.*

Element Overview

The `<include>` element is used within a number of other elements. Its effect varies depending on its parent element.

Parentage

- `<actions>` - Within an `<actions>` element the `<include>` element specifies a list of actions. See Actions specified in <include>.
- `<option>` - Within an `<option>` element the `<include>` element specifies a _value to be presented in a selector list of values for the option on the Configuration Properties page for a resource. See <option> element defines selector list values.
- `<classpath>` - Within a `<classpath>` element the `<include>` element specifies an external class, a plugin, or path that a plugin needs access to in order to perform a plugin function. See Section 3.3.3, “classpath”.
- `<metrics>` - Within a `<metrics>` element the `<include>` element specifies the name of another `<metrics>` element , defined in the root of the plugin, to be included in the the current `<metrics>` element. See <metrics> elements included using include element.
- `<scan>` - Within a `<scan>` element the `<include>` element specifies a filename pattern or a registry key to use in an auto-discovery process.

3.3.6. metric

*Topics marked with*relate to features available only in vFabric Hyperic.*

Element Overview

The `<metric>` element defines a metric to be obtained for a resource type.

Any attribute of a `<metric>` element is available as a variable, as described in _Variables in Plugin Descriptors.

Parentage

- `<metrics>` - A <metric> element can be included in a named <metrics> element that can be included in resource elements by reference. For more information, see *<metrics>* on page .
- `<platform>` - A `<metric>` at this level is associated with the platform type.
- `<server>` - A `<metric>` at this level is associated with the server type.
- `<service>` - A `<metric>` at this level is associated with the service type.

Contents

```
<metric>
    name
    alias
    category
    defaultOn
    indicator
    collectionType
    units
    interval
    group
    rate
    template
```

name

Required:	y
Description:	Name of the metric shown in the HQ UI.
Default:	None

alias

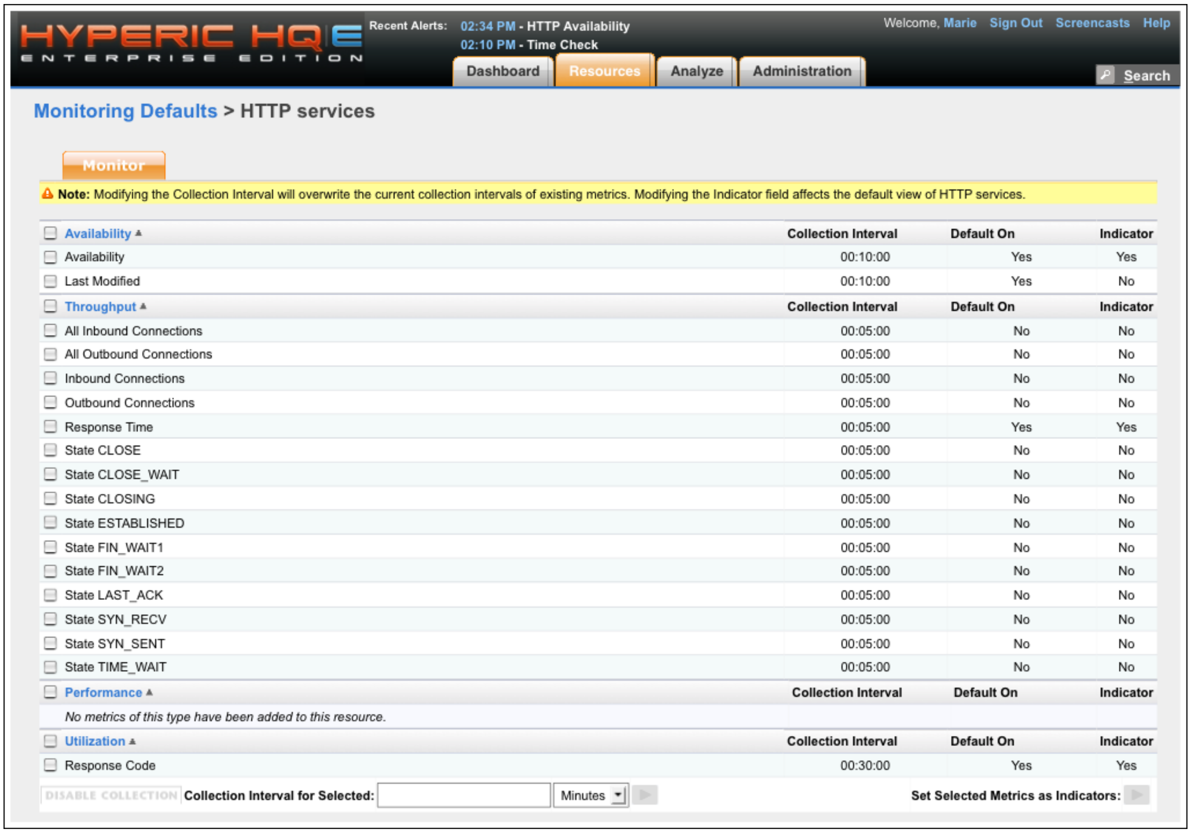
Required:	n
Description:	Abbreviated name of the metric, displayed in the plugin output (name-value pairs). In the case of a JMX measurement plugin, `alias` must exactly match the mbean attribute name.
Default:	Value of name, stripped of spaces and non-alphanumeric content.

category

Required:	n
Description:	The category of metric; determines where a metric appears on HQ pages that organize metrics by category, like the example shown in the screenshot below.
Default:	If the `name` attribute is set to "Availability", `category` defaults to AVAILABILITY, otherwise the default is UTILIZATION.
Values:	AVAILABILITY

	THROUGHPUT PERFORMANCE UTILIZATION

The screenshot is the **Monitoring Defaults** page for the HTTP service type. Note that the metrics are listed by category.



Monitoring Defaults > HTTP services

⚠ **Note:** Modifying the Collection Interval will overwrite the current collection intervals of existing metrics. Modifying the Indicator field affects the default view of HTTP services.

		Collection Interval	Default On	Indicator
□	Availability ▲	Collection Interval	Default On	Indicator
□	Availability	00:10:00	Yes	Yes
□	Last Modified	00:10:00	Yes	No
□	Throughput ▲	Collection Interval	Default On	Indicator
□	All Inbound Connections	00:05:00	No	No
□	All Outbound Connections	00:05:00	No	No
□	Inbound Connections	00:05:00	No	No
□	Outbound Connections	00:05:00	No	No
□	Response Time	00:05:00	Yes	Yes
□	State CLOSE	00:05:00	No	No
□	State CLOSE_WAIT	00:05:00	No	No
□	State CLOSING	00:05:00	No	No
□	State ESTABLISHED	00:05:00	No	No
□	State FIN_WAIT1	00:05:00	No	No
□	State FIN_WAIT2	00:05:00	No	No
□	State LAST_ACK	00:05:00	No	No
□	State SYN_RECV	00:05:00	No	No
□	State SYN_SENT	00:05:00	No	No
□	State TIME_WAIT	00:05:00	No	No
□	Performance ▲	Collection Interval	Default On	Indicator
	No metrics of this type have been added to this resource.			
□	Utilization ▲	Collection Interval	Default On	Indicator
□	Response Code	00:30:00	Yes	Yes

DISABLE COLLECTION **Collection Interval for Selected:** Minutes ▾ ▶ **Set Selected Metrics as Indicators:** ▶

units

Required:	n
Description:	The units of measurement for the metric's value, which affects the formatting applied in the HQ user interface.
Values:	none - *No formatting will be applied to metric values in the HQ user interface.*
	percentage -*Metric values will be shown as a percentage, for instance "100.0%".*
	B -*Metric values are in bytes, for instance "12.5 B".*
	KB -*Metric values are in kilobytes.*
	MB

	-Metric values are in megabytes, for instance "14.5 KB".
	GB *-Metric values are in gigabytes, for instance "102.8 GB".*
	TB *-Metric values are in terabytes, for instance "100.5 TB".*
	epoch-millis *-Metric values are reported in number of milliseconds since Jan 1, 1970.*
	epoch-seconds *- Metric values will be shown in number of seconds since Jan 1, 1970.*
	ns *- Metric values will be shown in nanoseconds.*
	mu *- Metric values will be shown in microseconds.*
	ms *- Metric values will be shown in_milliseconds.*
	jiffys *- Metric values will be shown in Jiffies (1/100 of a second).*
	sec *- Metric values will be shown in seconds.*
	cents *-Metric values will be shown in cents (1/100 of 1 US dollar).*
Default	If `units` is not specified, default is "none", unless the `name` attribute is "Availability", in which case the default for `units` is "percentage".

indicator

Required:	y
Description:	Whether or not this metric is an indicator metric in HQ. Indicator metrics are charted on a resources **Indicators** tab in the HQ UI.
Values:	true
	false
Default:	None

collectionType

Required:	n
Description:	Describes the behavior of a metric's values over time. For example, the value of "Requests Served" will

	trend up as more and more requests are counted over time.
Values:	dynamic - Metric value may go either up or down over time. "Availability" is an example of a metric whose collectionType is "dynamic".
	static - Metric value does not change. For example, a value that takes the form of a timestamp.
	trendsup - Metric value will increase, but not decrease. For metrics whose collectionType is "trendsup", HQ automatically creates a derived metric that reports the rate at which the metric value increases per minute. If an automatically created rate metric's defaultOn attribute is "true", the defaultOn attribute for metric it is based upon will be set to "false" - so that you'll collect and display the only the rate metric, not the original metric). If you do not want HQ to generate a rate metric for a metrics whose collectionType is "trendsup", set rate to "none".
	trendsdown - Value changes will always decrease. Is above info for trendsup also valid for this section?
Default:	dynamic

defaultOn

Required:	n
Description:	Controls whether or not the metric is collected by default.
Values:	true
	false
Default:	*"*true", if indicator=true. Otherwise, *"*false".

interval

Required:	n
Description:	Default collection interval.
Values:	A numeric value, which is interpreted as milliseconds (ms).
Default:	If name attribute is Availability, defaults to 1, 5, and 10 minutes, for resources of types Platform, Server, or Service, respectively.
	Otherwise, the default depends on the value of collectionType:
	If collectionType is dynamic, default is 5 minutes
	If collectionType is trendsup or trendsdown, default is 10 minutes
	If collectionType is static, 30 minutes

group

Required:	n
Description:	
Values:	
Default:	

rate

Required:	n	
Description:	Specifies the time period for a rate measurement. Valid only for metrics whose collectionType is trendsup.	
Values:	1s	1 second
	1m	1 minute
	1h	1 hour
	disable automatically generated rate metric	
Default:		

template

Required:	y
Description:	The template attribute specifies the information required to obtain a specific metric. Every metric must have a template. However, template attribute may not always be defined within theelement. Often, template is defined in the root of a descriptor, using variables whose values are assigned in resource elements. See Variables Simplify Metric Template Definitions **on page** and the examples below for more information.
Default:	none

Examples

template for a metric obtained via a vendor API

This excerpt from the coldfusion-plugin shows a <metric> element that does not include an explicitly defined template attribute. The <filter> element defines the template based on variables provided by config <option> elements (%installpath% and %logfile%) and the metric's alias attribute.

```
<filter name="template"
        value="coldfusion-stats:installpath=%installpath%,logfile=%logfile%:$\{alias\}"/>
<metric name="Threads Listening For A New Connection"
        alias="listenTh"
        category="THROUGHPUT"
        indicator="false"
        collectionType="dynamic"
        units="none"/>
```

template for metric obtained with network collector

The structure of a template for a network collector is:

```
Protocol:hostname=HostName,port=PortNumber,ssl=yes:MetricName
```

For example:

HTTP:hostname=www.hyperic.com,port=443,ssl=true:Availability

template for metric obtained with SIGAR

For example:

sigar:*Type=_QueryType*:Arg=*PtqlQuery*:_MetricName

Where PtqlQuery is in this form:

Class.Attribute.operator=value

Where:

- Class is the name of the Sigar class minus the Proc prefix.
- Attribute is an attribute of the given Class, index into an array or key in a Map class.
- Operator=value is a string operation, e.g. eq, ne, ew, etc.

Pid.PidFile.eq=%installpath%/log/zmconvertdmon.pid

template for metric obtained with JDBC

DummyDomain:KeyProperties:MetricName

For example:

sybase:Type=Service,instance=Geniousity:Number of Indexes

template for metric obtained with SQL Query

sql:Query:MetricName

For example:

sql:SELECT COUNT⭐ FROM MAILHOSTS:Number of Servers

template for metric obtained with JMX (JSR 160 and connector method)

Domain_:*KeyProperties*:_MbeanAttribute

For example

org.apache.activemq:Type=Connector,BrokerName=broker1,ConnectorName=winConn:EnqueueCount

template for metric obtained with script

exec:timeout=value,file=filename,args=ArgumentList:MetricName

For example:

exec:file=pdk/scripts/device_iostat.pl,args=sda:w/s

template for metric obtained with SNMP, constructed with globally available <config>

The template below is constructed using a globally available configuration schema.
template="${snmp.template}:cacheUptime"
The "snmp" schema is defined in HQ's netservices-plugin descriptor, as shown in this excerpt.
<config name="snmp" type="global"> <option name="snmpIp" description="SNMP agent IP address" default="127.0.0.1"/> <option name="snmpPort" description="SNMP agent port" type="port" default="161"/> <option name="snmpVersion" description="SNMP Version" type="enum"> <include name="v2c"/> <include name="v1"/> <include name="v3"/> </option>
In the template definition, ${snmp.template} is replaced by:

snmp:snmpIP=**,snmpPort=**,snmpVersion=*

resulting in this template:

snmp:snmpIP=**,snmpPort=**,snmpVersion=*:cacheUptime

${snmp.template},Avail=true:wsIfMedia:${if.config}

Metric Parameters

Metric Attribute	Description	Req'd	Possible Values
`name`	Name of the metric to be displayed in the GUI	Y	
`alias`	Abbreviated name of the metric, displayed in the plugin's output (name-value pairs). If not specified, the alias defaults to the metric-parameter `name`, stripped of any non-letter and non-digit characters and of whitespace.	N	In the case of a JMX measurement plugin, the alias must match exactly the name of the mbean attribute.
`category`	The category of metric	N	AVAILABILITY, THROUGHPUT, PERFORMANCE, UTILIZATION If the name attribute is Availability, defaults to AVAILABILITY, otherwise defaults to UTILIZATION.
`units`	The units of measurement of the metric, which determines its display in the UI	N	• none: Will not be formatted. • percentage • B: Bytes • KB: Kilobytes • MB: Megabytes • GB: Gigabytes

Metric Attribute	Description	Req'd	Possible Values
			- TB: Terabytes - epoch-millis: Time since January 1, 1970 in milliseconds. - epoch-seconds: Time since January 1, 1970 in seconds. - ns: Nanoseconds - mu: Microseconds - ms: Milliseconds - jiffys: Jiffies (1/100 sec) - sec: Seconds - cents: Cents (1/100 of 1 US Dollar)

If the name attribute is Availability, defaults to percentage, otherwise defaults to none. |

`indicator`	Whether or not this metric should be treated as an indicator metric in HQ	N	true, false
`collectionType`	A description of how the metric's data will behave, for purposes of display in HQ. For example, the metric "Requests Served" will trend up as more and more requests are counted over time.	N	- dynamic: Value may go up or down. - static: Value will not change or not graph. For example, a date stamp. - trendsup: Values will always increase. Because of that, the rate of change becomes more important, so HQ automatically creates a secondary metric: a per-minute rate measurement. If this rate metric has a defaultOn attribute set to true, the defaultOn attribute for the original metric is set to false (therefore only the rate metric will

			be displayed, not the original metric). To disable the automatically generated rate metric, set its rate attribute to none. • trendsdown: Value changes will always decrease. Defaults to dynamic.
`template`	Unable to render {include} Couldn't find a page to include called: TemplateLearn more.	N	See examples
`defaultOn`	If true, this measurement will be scheduled by default.	N	If `indicator` is true defaults to true. Otherwise defaults to false.
`interval`	Default collection interval (in milliseconds)	N	If the name attribute is Availability, defaults are: • Platforms, 1 minute • Servers, 5 minute • Services, 10 minutes Otherwise, defaults are: • collectionType dynamic, 5 minutes • collectionType trendsup,trendsdown,10 minutes • collectionType static, 30 minutes
`rate`	Specifies the time period for a rate measurement. Valid only for metrics of collectionType trendsup.	N	Possible values: • 1s (1 second) • 1m (1 minute) • 1h (1 hour) • <none> (disable automatically generated rate metric)

Plugin UOM

- none: Will not be formatted.
- percentage

- B: Bytes
- KB: Kilobytes
- MB: Megabytes
- GB: Gigabytes
- TB: Terabytes
- epoch-millis: Time since January 1, 1970 in milliseconds.
- epoch-seconds: Time since January 1, 1970 in seconds.
- ns: Nanoseconds
- mu: Microseconds
- ms: Milliseconds
- jiffys: Jiffies (1/100 sec)
- sec: Seconds
- cents: Cents (1/100 of 1 US Dollar)

If the name attribute is Availability, defaults to percentage, otherwise defaults to none.

3.3.7. metrics

*Topics marked with*relate to features available only in vFabric Hyperic.*

The `<metrics>` element defines a set of metrics. A `<metrics>` element can containing one or more `<metric>` elements that define an individual metric and its attributes. `<metrics>` elements can serve as building blocks - one `<metrics>` element can include one or more other named `<metrics>` elements by reference.

Creating a named `<metrics>` element is useful if you will to collect the same set of metrics for multiple resources. You can define the set of metrics once, and then include it in multiple places in the descriptor. There are several ways to use a `<metrics>` element defined in the root of a plugin in resource elements that follow in the descriptor. Each method is shown in the Examples section below.

Element Structure

```
<metrics>
    name
    include
    <include>
    <metric>
```

Parentage

The `<metrics>` element can be a child of:

- `<plugin>` (root) - A `<metrics>` element in the plugin root is not associated with a resource type unless it (1) is included explicitly in a resource element later in the descriptor, or (2) has a `name` attribute value that matches the string that results from concatenating a resource element's `name` and `version` attributes.
- `<platform>` - A `<metrics>` element at this level is associated with the platform type.

- `<server>` - A `<metrics>` element at this level is associated with the server type.
- `<service>` - A `<metrics>` element at this level is associated with the service type.

Contents

- `name` - Optionally, specifies the name `<metrics>` element. The `name` is required if you wish to associate the element with `<platform>`, `<server>`, and `<service>` elements later in the the descriptor. There are several ways that a `<metrics>` element defined in the root of a plugin can be used in resource elements that follow in the descriptor:
  - Use the `<include>` element to reference the `<metrics>` element by its `name` .
  - Use the `include` attribute, to reference the `<metrics>` element by its `name` .
  - Set the `<metrics>` element's `name` attribute to a string that matches the string made up by concatenating a resource element's `name` and `version` attributes. See <metrics> name matches resource's NameAttribute string.
- `include` - Can be used to include one or more `<metrics>` elements in the descriptor by reference to the target `name` attribute(s). If you use include attribute to include multiple `<metrics>` elements, specify the element names as a comma-separated list. For usage, see <metrics> element is referenced using the include attribute.
- `<include>` - Has similar functionality to the include attribute, can be used to include another `<metrics>` element in the descriptor by reference to the target's name attribute. For usage, see <metrics> element is referenced using the include element.
- `<metric>` - A `<metrics>` element can contain multiple `<metric>` elements that define the name and other attributes of a particular metric. see unnamed <metrics> element within a <service> element. For information about the contents of the `<metric>` element, see Section 3.3.6, “metric”.

Examples

unnamed <metrics> element within a <service> element

The excerpt below from the oc4j-plugin is a `<service>` element that contains an unnamed `<metrics>` element that defines two metrics. Because the `<metrics>` element is not referenced elsewhere in the descriptor, the `name` attribute is optional.

```
<service name="Application">
   <config>
      <option name="name" description="Name" default="" />
   </config>
   <property name="OBJECT_NAME"
      value="${domain}:j2eeType=J2EEApplication,name=*,J2EEServer=standalone" />
   <metrics>
      <metric name="Availability"
         template="${OBJECT_NAME}:state:jmx.url=%jmx.url%,jmx.username=%jmx.username
%,jmx.password=%jmx.password%,jmx.provider.pkgs=%jmx.provider.pkgs%"
         indicator="true" />
      <metric name="Start Time" alias="startTime"
         template="${OBJECT_NAME}:${alias}:jmx.url=%jmx.url%,jmx.username=%jmx.username
%,jmx.password=%jmx.password%,jmx.provider.pkgs=%jmx.provider.pkgs%"
         indicator="true" units="epoch-millis" collectionType="static" />
   </metrics>
   <plugin type="autoinventory" />
</service>
```

<metrics> element defined

This excerpt from the Jetty plugin defines a `<metrics>` element named "Class Loading Metrics" in the root `<plugin>` element.

```
<metrics name="Class Loading Metrics">
  <metric name="Loaded Class Count"
          indicator="false"
          category="THROUGHPUT"/>
  <metric name="Total Loaded Class Count"
          indicator="false"
          category="THROUGHPUT"/>
  <metric name="Unloaded Class Count"
          indicator="false"
          category="THROUGHPUT"/>
</metrics>
```

<metrics> element is referenced using include attribute

This `<metrics>` element in the Jetty plugin includes the group of metrics defined in the previous example in a `<server>` element.

```
<server name="Jetty"
     <metrics include="Class Loading Metrics"/>
```

<metrics> elements included using include element

In this excerpt from the mssql-plugin, the `<metrics>` element includes 8 sets of metrics, defined in the root of the plugin. This is an example of using the `<include>` element, instead the `include` attribute to point to a named set of metrics. Both forms of inclusion are supported, and have the same effect.

```
<server name="MsSQL"
  ...
  ...
  <metrics>
    <include name="mssql-avail"/>
    <include name="mssql-database"/>
    <include name="mssql-access"/>
    <include name="mssql-general"/>
    <include name="mssql-memory"/>
    <include name="mssql-locks"/>
    <include name="mssql-latches"/>
    <include name="mssql-cache"/>
  </metrics>
```

<metrics> element includes several metric groups, plus a metric definition

This `<metrics>` element from the weblogic-plugin includes other metric groups, and defines an Availability `<metric>` specifically.

```
<service name="Entity EJB">
  <metrics include="ejb-tx-runtime,ejb-pool-runtime,ejb-cache-runtime">
    <metric name="Availability"
            template="${EntityEJBRuntime}:Name"
            indicator="true"/>
  </metrics>
```

<metrics> name matches resource's NameAttribute string

The `metrics` element named "iPlanet Admin 4.1" is defined in the root of the HQ's iplanet-plugin descriptor.

```
<metrics name="iPlanet Admin 4.1">
   <metric name="Availability"
   alias="Availability"
   template="%protocol%:$\{url.config\}:$\{alias\}"
   category="AVAILABILITY"
   group="Reliability"
   indicator="true"
   units="percentage"
   collectionType="dynamic"/>
</metrics>
```

The `server` element later in the same descriptor does not contain any `<metric>` or `<metrics>` elements. Instead, the metrics in `<metrics>` element named "iPlanet Admin 4.1" are mapped to the resource element whose `name` attribute is "iPlanet Admin" and `version` attribute is "4.1"

```
<server name="iPlanet Admin"
   description="Admin Server"
   version="4.1"
   platforms="Unix,Win32">

   <plugin type="measurement"
      class="iPlanetMeasurementPlugin"/>

   <plugin type="control"
            class="iPlanetControlPlugin"/>

   <actions include="start,stop,restart"/>

   <plugin type="autoinventory"
            class="iPlanet4Detector"/>

    <config include="url,protocol">
      <option name="server.id"
              description="Server instance id"
              default="https-admserv"/>
    </config>

    <config include="snmp" type="measurement"/>

    <scan>
      <include name="/**/https-admserv/config/obj.conf"/>
    </scan>

</server>
```

3.3.8. option

*Topics marked with***relate to features available only in vFabric Hyperic.*

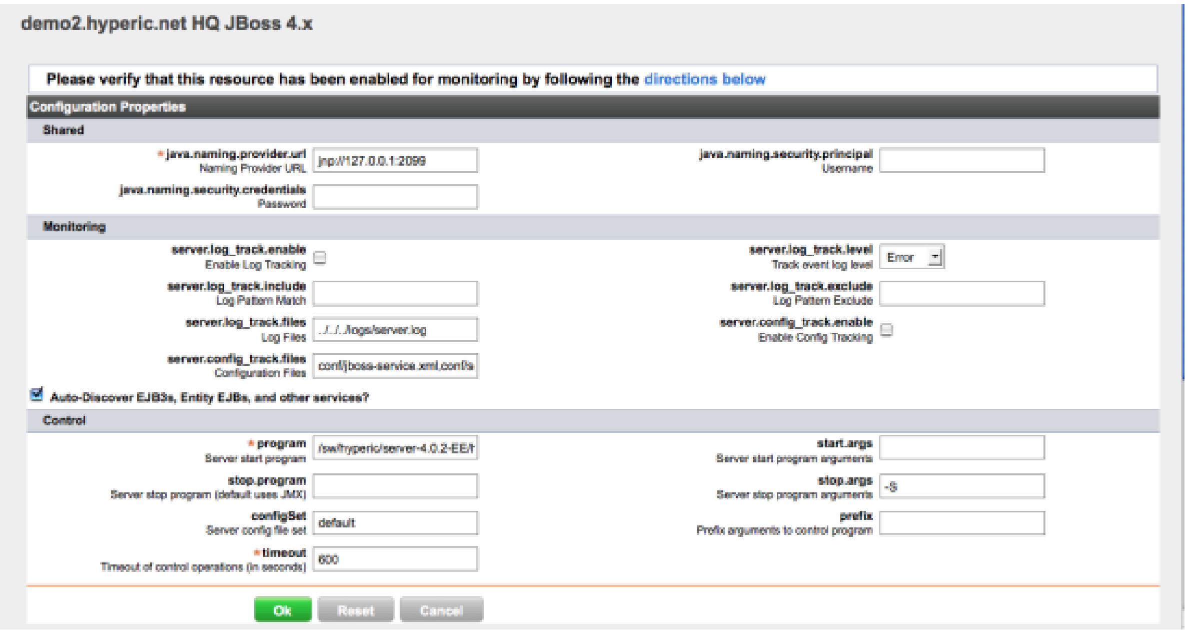
This page contains information that has not been reviewed. Click **Add Comment** at the bottom of the page to enter suggestions or corrections.

Element Overview

The `<option>` element defines an attribute of a platform, server, or service type. Some resource attributes are auto-discovered, others must be supplied by a user explicitly to enable monitoring and management, others may contain information that is useful but optional.

Attributes you specify in `<option>` elements appear in the **Configuration Properties** page for a resource and may be edited there. The type attribute of the parent `<config>` element controls whether the option is displayed in the "Shared", "Monitoring", or "Control" section of the page.

Data defined as an `<option>` is often used to in `<filter>` element to supply one or more portions of a `<metric>` elements template attribute.



demo2.hyperic.net HQ JBoss 4.x

Please verify that this resource has been enabled for monitoring by following the directions below

Configuration Properties

Shared

*java.naming.provider.url Naming Provider URL	jnp://127.0.0.1:2099	java.naming.security.principal Username	
java.naming.security.credentials Password			

Monitoring

server.log_track.enable Enable Log Tracking	☐	server.log_track.level Track event log level	Error
server.log_track.include Log Pattern Match		server.log_track.exclude Log Pattern Exclude	
server.log_track.files Log Files	../../../logs/server.log	server.config_track.enable Enable Config Tracking	☐
server.config_track.files Configuration Files	conf/jboss-service.xml,conf/s		

☑ Auto-Discover EJB3s, Entity EJBs, and other services?

Control

*program Server start program	/sw/hyperic/server-4.0.2-EE/t	start.args Server start program arguments	
stop.program Server stop program (default uses JMX)		stop.args Server stop program arguments	-S
configSet Server config file set	default	prefix Prefix arguments to control program	
*timeout Timeout of control operations (in seconds)	600		

Regardless of whether the resource attribute will be auto-discovered by the plugin or supplied by the user, if you want it to appear on the **Configuration Properties** page, you must specify it in an `<option>` element.

Parent Elements

Child of a `<config>` element.

Child Elements and Attributes

- name - Required, specifies the option name, used by plugins to auto-configure during auto-discovery.
- description - Text description, shown in the **Configuration Properties** page.
- default - Optional, specifies the default value for the option, shown in the **Configuration Properties** page
- type - Optionally can be used to specify rules for the option value. You can specify a data type, and exclude it from the Configuration **Properties** page. Allowable values include:
  - string - Default type, arbitrary string value.
  - int - Value validated using Integer.parseInt
  - double - Value validated using Integer.parseDouble
  - secret - Value is not displayed in the UI.
  - hidden - Option is not displayed in the UI.
  - yesno - Boolean option.
  - ipaddress - Value must be a valid IP address.

  - enum - Drop-down list displayed in the UI.

- optional - May be used to specify that the option is not required.

- `<include>` - May be used to supply an value to be presented in a selector list of values for the option on the **Configuration Properties** page for a resource.

<option> Examples

<option> element defines selector list values

```
<option name="method" type="enum" description="Request Method"> <include name="HEAD"/> <include name="GET"/> </option>
```

<option> element for an optional option

```
<option name="hostheader" description="Host Header" optional="true"/>
```

<option> element defines default value

```
<option name="follow" description="Follow Redirects" type="boolean" default="false"/>
```

3.3.9. platform

*Topics marked with*relate to features available only in vFabric Hyperic.*

This page contains information that has not been reviewed. Click **Add Comment** at the bottom of the page to enter suggestions or corrections.

Element Overview

The `<platform>` element supplies the information or rules that govern the functions - which may include autodiscovery, metric collection, log/configuration tracking, and control - the plugin performs for a resource of the type "platform", in terms of the HQ inventory model.

A plugin that only manages lower level resources - servers or services - will not include a `<platform>` element.

Parent Elements

The `<platform>` element may only be a child of:

- root `<plugin>`

platform Element Structure

```
<platform> <filter> <property> <config> <properties> <plugin> <help> <metrics> <metric> <actions> <classpath> <script>
```

Child Elements

- `<filter>` - Defines variables that specify a pattern for the metric templates for a resource type (in this usage, a platform type) and elements of the metric template, as desired. Strictly speaking, use of `<filter>`

elements is optional, but most plugins use them because it is more efficient and less error-prone than defining the template for each metric explicitly. For more information, see `Section 3.3.2, “filter”`.

- `<property>` - Defines variables that can be referenced within the plugin descriptor and also by plugin classes. The use of `<property>` element is optional. For more information see `Section 3.3.13, “property”`.
- `<config>` - Defines a configuration schema, which consists of a set of `<option>` elements for a resource type (in this usage, a platform type). One or more `<config>` elements can be defined.
- `<properties>` - A container for one or more `<property>` elements. For more information, see `Section 3.3.12, “properties”`.
- `<plugin>` - Defines a plugin function - auto-discovery, measurement, etc - supported for the resource type, and the class to use for the function. For more information, see `plugin` (Platform, Server and Service).
- `<help>` - Defines the help text that is displayed on the Configuration Properties for the resource type. For more information, see `Section 3.3.4, “help”`.
- `<metrics>` - A container for one or more `<metric>` elements. You can assign a name to `<metric>` elements, and reference it by name in other parts of the plugin descriptor. The use of the `<metrics>` element is optional. For more information, see `Section 3.3.7, “metrics”`.
- `<metric>` - Defines the attributes of a metric, for the resource type, including its name, category, whether it is an indicator metric, how to collect it (metric template) etc. If the plugin performs measurement for the platform type, the plugin descriptor will contain a `metric` element for each metric it collects. For more information, see `Section 3.3.6, “metric”`.
- `<script>` - May be used to include a script a plugin uses in the XML descriptor. Use of the `<script>` element is optional. Scripts used by a plugin can reside within the descriptor or in the filesystem. For more information see `Section 3.3.15, “script”`.
- `<server>` - Defines a server type that runs on the platform type. The `<server>` element is analogous to the `<platform>` and `<service>` elements - it define the information or rules that govern the functions - which may include autodiscovery, metric collection, log/configuration tracking, and control - the plugin performs for a specific server type, in terms of the HQ inventory model. For more information see `Section 3.3.16, “server”`.

platform Examples

Structure of a simple <platform> element

This `<platform>` element from the NetScaler plugin descriptor has this structure:

```
<platform> <config> <plugin> <metric> (multiple elements) <server> <config> <plugin> <metric> (multiple elements)
```

Simple <platform> element in full

This is the complete `<platform>` element from the NetScaler plugin descriptor:

```
<platform name="NetScaler"> <config> <option name="snmpIp" description="NetScaler IP address" type="ipaddress" default="10.0.0.11"/> <option name="snmpPort" description="NetScaler SNMP port" type="port" default="161"/> <option name="snmpVersion" description="SNMP Version" default="v1" type="enum"> <include name="v1"/> <include name="v2c"/> </option> <option name="snmpCommunity" description="SNMP Community" default="public"/> </config> <plugin type="measurement" class="net.hyperic.hq.product.SNMPMeasurementPlugin"/> <server name="NetScaler
```

```
Interface"> <config> <option name="if.name" description="Interface name" default="www"/> </config> <plugin type="measurement" class="net.hyperic.hq.product.SNMPMeasurementPlugin"/> <metric name="Availability" template="$\{snmp.template\},Avail=true:wsIfMedia:$\{if.config\}" indicator="true"/> <metric name="Bits Received" template="$\{snmp.template\}:rxRawBandwidthUsage:$\{if.config\}" collectionType="trendsup" units="b" rate="1s" indicator="true"/> <metric name="Packets Received" template="$\{snmp.template\}:rxCurrentPacketRate:$\{if.config\}" collectionType="trendsup" rate="1s" indicator="true"/> <metric name="Bits Transmitted" template="$\{snmp.template\}:txRawBandwidthUsage:$\{if.config\}" collectionType="trendsup" units="b" rate="1s" indicator="true"/> <metric name="Packets Transmitted" template="$\{snmp.template\}:txCurrentPacketRate:$\{if.config\}" collectionType="trendsup" rate="1s" indicator="true"/> </server> <metric name="Availability" template="$\{snmp.template\},Avail=true:totalClientConnections" indicator="true"/> <metric name="Current Client Connections" template="$\{snmp.template\}:curClientConnections" indicator="true"/> <metric name="Total Client Connections" template="$\{snmp.template\}:totalClientConnections" indicator="true" collectionType="trendsup"/> <metric name="Current Server Connections" template="$\{snmp.template\}:curServerConnections" indicator="true"/> <metric name="Total Server Connections" template="$\{snmp.template\}:totalServerConnections" indicator="true" collectionType="trendsup"/> </platform>
```

3.3.10. plugin (Platform, Server and Service)

*Topics marked with*relate to features available only in vFabric Hyperic.*

This page contains information that has not been reviewed. Click **Add Comment** at the bottom of the page to enter suggestions or corrections.

Element Overview

A `<plugin>` element element declares a management function — such as metric collection or log tracking — that the plugin performs for a resource type, and the class performs that function.

Parent Elements

- `<server>` - A `<plugin>` element in a `<server>` element specifies a function the plugin performs for that server type.
- `<service>` - A `<plugin>` element in a `<service>` element specifies a function the plugin performs for that service type.

Element Structure

```
<plugin> type class
```

Child Attributes

- `type` - This required attribute specifies a plugin function. Allowable values:
  - collector - obtain metrics for a remote resource on the network - for example an HTTP, FTP, or DNS service.
  - log_track - tracks messages written to resource log files.
  - configtrack - tracks changes made to resource configuration files.
  - control - performs supported control actions on a resource, for example, shutdown, restart, run garbage collection, and so on. When `type` is set to "control", an `<action>` element must exist, as a sibling of the

  `<plugin>` element. The `<action>` element specifies the supported control actions. For more information, see Section 3.3.1, “actions”.

  - measurement - collect metrics for a resource
  - autoinventory - discover new resource instances or changes to existing resource data.
  - livedata - Obtain live system data and metrics, for instance CPU information and utilization, network interface configuration and metrics, and so on.
  - responsetime - No longer used. This value will not prevent a plugin from loading but it is not supported.

- `class` - Specifies the class that will perform the function, which could be a class in the plugin JAR, or a support class in `org.hyperic.hq.product`. Required for `<plugin>` elements within a `<server>` element. If the `class` attribute is omitted from a `<plugin>` element within a `<service>` element, its value is inherited from the class specified in the parent `<server>` element's `<plugin>` element.

Examples

<plugin> elements for a server type

This `<server>` element excerpted from HQ'x xen-plugin contains four `<plugin>` elements. The `<plugin>` element whose type is "measurement" specifies a plugin support class. The other `<plugin>` elements specify classes within the plugin.

```
<server name="Xen VM"> <config include="server.uuid"/> <properties> <prop-
erty name="os" description="OS"/> </properties> <plugin type="autoinventory"
class="org.hyperic.hq.plugin.xen.XenVmDetector"/> <plugin type="measurement"
class="org.hyperic.hq.product.MeasurementPlugin"/> <plugin type="collector"
class="org.hyperic.hq.plugin.xen.XenVmCollector"/> <plugin type="control"
class="org.hyperic.hq.plugin.xen.XenVmControlPlugin"/> <actions
include="start,shutdown,forceShutdown,suspend,resume,reboot,forceReboot"/>
```

3.3.11. plugin (Root)

*Topics marked with*relate to features available only in vFabric Hyperic.*

This page contains information that has not been reviewed. Click **Add Comment** at the bottom of the page to enter suggestions or corrections.

Element Overview

The top level `<plugin>` element defines the plugin's name, package, and class.

Element Structure

```
<plugin> name class package <config> <metrics> <script> <filter> <classpath> <help> <platform> <server>
<service>
```

Child Elements and Attributes

- `name` - Name of the plugin. Defaults to the name of the plugin jar or xml file, stripped of -plugin.xml or -plugin.jar.

- `package` - The default value is:
  net.hyperic.hq.product.name
  where
  name
  is the value of the `name` attribute. If the `class` attribute is specified, the value of `package` is used to prefix it.

- `class` - Name of the ProductPlugin implementation. This is required only if it is necessary to override methods in the ProductPlugin.

- `<config>` - A `config` element in the plugin root can be included by reference by name in resource elements below it in the descriptor. For more information, see Section 3.3.18, “config”.

- `<metrics>` - A `metrics` element in the plugin root can be included by reference by name in resource elements below it in the descriptor. For more information, see Section 3.3.7, “metrics”.

- `<script>` - Used in a script plugin to include the script body in the plugin desscriptor. For more information, see Section 3.3.15, “script”.

- `<filter>` - A `<filter>` element in the plugin root defines a variable, typically used to in metric expressions, that is available to resource elements below it in the descriptor. For more information, see Section 3.3.2, “filter”.

- `<classpath>` - Identifies HQ libraries or external JARs that the plugin uses to perform auto-discovery or other plugin functions. For more information, see Section 3.3.3, “classpath”.

- `<help>` - A `<help>` element in the plugin root can be included by reference by name in resource elements below it in the descriptor. For more information, see Section 3.3.4, “help”.

- `<platform>` - Defines a platform type managed by the plugin. For more information, see Section 3.3.9, “platform”.

- `<server>` - A `server` element in the plugin root defines a server type managed by the plugin, that runs on a platform type managed by HQ. For more information, see Section 3.3.16, “server”.

- `<service>` - A `service` element in the plugin root defines a platform service type managed by the plugin, that runs on a platform type managed by HQ. For more information, see Section 3.3.17, “service”.

Examples

plugin name and package specified in plugin root

In HQ's geronimo plugin, the name of the plugin and its package are specified explicitly.

```
<plugin package="org.hyperic.hq.plugin.geronimo" name="geronimo">
```

plugin class specified in root

In HQ's vmware plugin, the `class` attribute specifies the ProductPlugin implementation.

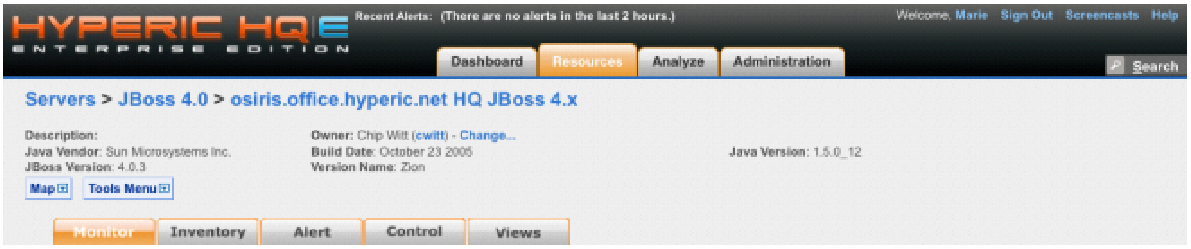
```
<plugin name="vmware" class="VMwareProductPlugin"
```

3.3.12. properties

*Topics marked with***relate to features available only in vFabric Hyperic.*

Element Overview

The <properties> element is a container for <property> elements. Properties whose definitions are within a <properties> will appear at the top of a resource page (for a resource of the current type) in the HQ user interrface. Otherwise, the properties will not appear in the user interface.



Recent Alerts: (There are no alerts in the last 2 hours.)

Servers > JBoss 4.0 > osiris.office.hyperic.net HQ JBoss 4.x

Description:
Java Vendor: Sun Microsystems Inc.
JBoss Version: 4.0.3

Owner: Chip Witt (cwitt) - Change...
Build Date: October 23 2005
Version Name: Zion

Java Version: 1.5.0_12

Parentage

The <properties> element can be a child of these elements:

- <platform> - The properties defined in a <properties> element below a <platform> element will displayed in the HQ user interface for resources of the platform type.
- <server> - The properties defined in a <properties> element below a <server> element will be displayed in the HQ user interface for resources of the server type.
- <service> - The properties defined in a <properties> element below a <service> element will be displayed in the HQ user interface for resources of the service type.

Child Elements

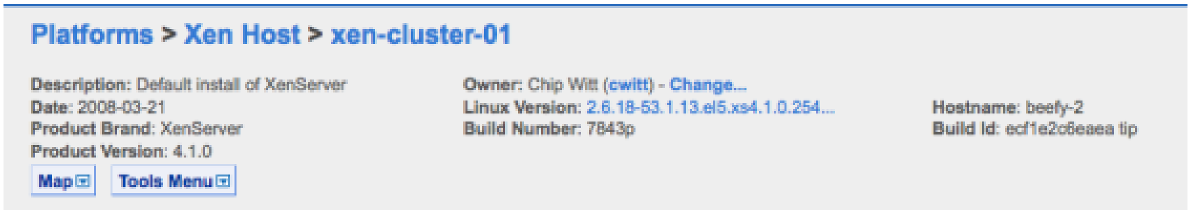
A <properties> element contains one or more <property> elements. For more information, see Section 3.3.13, "property".

Examples

<properties> element for a platform type

The properties specified in this <properties> element from HQ's xen plugin appear in the HQ user interface for resources whose type is "Xen Host".

<platform name="Xen Host"> <properties> <property name="version" description="Product Version"/> <property name="brand" description="Product Brand"/> <property name="build_id" description="Build Id"/> <property name="hostname" description="Hostname"/> <property name="date" description="Date"/> <property name="build_number" description="Build Number"/> <property name="linux" description="Linux Version"/> </properties>



Platforms > Xen Host > xen-cluster-01

Description: Default install of XenServer
Date: 2008-03-21
Product Brand: XenServer
Product Version: 4.1.0

Owner: Chip Witt (cwitt) - Change...
Linux Version: 2.6.18-53.1.13.el5.xs4.1.0.254...
Build Number: 7843p

Hostname: beefy-2
Build Id: ecf1e2c6eaea tip

3.3.13. property

*Topics marked with*relate to features available only in vFabric Hyperic.*

Element Overview

The `<property>` element is used to:

- Define an attribute of a resource type whose value for a particular resource instance is defined or discovered by a plugin class or support class, so that it can be referenced in other parts of the descriptor.
- Define an a resource type attribute value that a plugin class or support class needs to perform autodiscovery or another plugin function. Values can be hard-coded or defined as a string expression that uses values supplied by a configuration option (in an `<option>` element).

Within a descriptor, the `<property>` element is similar in functionality to the the `<filter>` element, but data specified in a `<filter>` element is available only within the descriptor - it is neither provided by nor available to plugin classes.

`<property>` elements may be included in a `<properties>` element to effect their presentation in the HQ UI. `<property>` elements are often used to supply some portion of a metric template.

> **Overriding <property> value with an agent property**
>
> You can override the descriptor-defined value for a `<property>` for resource instances on a particular platform. See Overriding the Value of a Resource `property` at Platform Level [74].

Parent Elements

- `<platform>` - Use to specify properties associated with the platform type.
- `<server>` - Use to specify properties associated with the server type.
- `<service>` - Use to specify properties associated with the service type.
- `<properties>` - Enclosing `<property>` elements in a `<properties>` element causes them to be displayed in the the resource page, for the instances of the owning platform, server, or server type. For more information, see `Section 3.3.12, “properties”`.

Child Attributes

- `name` - Name of the property
- `value` - Value of the property

Examples

<property> defines connection string that a support class appends to metric templates

The `<property>` element in this excerpt from the db2 plugin defines a property named `template-config`. The value of `template-config` is expressed in terms of variables that return the values of several configuration options for the resource type. The options - nodename, user, and password - are defined in `<option>` elements earlier in the descriptor; their values are user-supplied on the **Configuration Properties** page for a DB2 server.

A number of HQ plugins use `template-config` to define connection-related properties that are necessary to access metrics. HQ's `MeasurementInfo` support class, available to all plugins, has a method that appends the value of `template-config` to the `template` attribute for a metric.

<plugin name="db2" class="DB2ProductPlugin"> <\!-\- appended to each template by MeasurementInfoXML \--> <property name="template-config" value="nodename=%nodename%,user=%user%,password=%password%"/>
The plugin class DB2MeasurementPlugin obtains the the values for nodename, user, and password.

Later in the plugin, the `template` attribute in a `<metric>` element is defined like this:

<metric name="Local Databases with Connects" alias="ConLocalDbases" template="db2:type=Server:${alias}" category="THROUGHPUT" units="none" collectionType="dynamic"/>
The `MeasurementInfo` class appends the value of `template-config` to the value of the `template` attribute for each metric, as defined above, resulting in:

```
template="db2:type=Server:${alias}:nodename=${nodename},user=
${user},password=${password}"
```

<property> defines OBJECT_NAME for use in plugin descriptor and class

This `<service>` element from the jboss-entity-container-plugin descriptor defines the `OBJECT_NAME` property, and references it in the `template` attribute for a metric. Note that the

%name%

portion of the value supplies the value of the the option's `name` attribute.

<service name="JCA Data Source"> <config> <option name="name" description="JNDI Name" default="DefaultDS"/> </config> <property name="OBJECT_NAME" value="jboss.jca:name=%name%,service=DataSourceBinding"/> <metric name="Availability" template="$\{OBJECT_NAME\}:StateString" indicator="true"/> <plugin type="autoinventory"/> <plugin type="control" class="JBossStateServiceControlPlugin"/> <actions include="stop,start"/> </service>
The `OBJECT_NAME` property is also used in the plugin's JBossServiceControlPlugin class:
protected String getObjectName() \{ //defined in hq-plugin.xml within the <service> tag String objectName = getTypeProperty(JBossQuery.PROP_OBJECT_NAME);

<property> element overrides the default value of an <option>

In this excerpt from the example drupal-plugin, the `<property>` element is used to set the default values for several configuration options, overriding the defaults defined in their `<option>` elements.

The last line of the excerpt includes the globally available `<config>` element named "sql", which is defined in the sql-query plugin descriptor. A listing of the "sql" `<config>` is shown in the second example on the config page.

The `<property>` elements shown in bold overrides the default values defined in the the sql-query plugin descriptor, with values appropriate for connecting to drupal.

<server name="Drupal"> <property name="INVENTORY_ID" value="drupal"/> <property name="PROC_QUERY" value="State.Name.re=post(gres\|master)"/> <\!-\- default properties \--> <property name="jdbcUrl" value="jdbc:postgresql://localhost/drupal?protocolVersion=2"/> <property name="jdbcDriver" value="org.postgresql.Driver"/> <property name="jdbcUser" value="drupal"/> <property name="jdbcPassword" value="drupal"/> <\!-\- config defined by the sqlquery plugin \--> <config include="sql,http"/>

3.3.14. scan

*Topics marked with*relate to features available only in vFabric Hyperic.*

This page contains information that has not been reviewed. Click **Add Comment** at the bottom of the page to enter suggestions or corrections.

Element Overview

The `<scan>` element specifies where the plugin looks, and what it looks for, when scanning for new and changed server instances.

The HQ Agent automatically scans for new or changed server instances upon startup, and every 15 minutes thereafter. This automatic scan, referred to as an *autoscan*, is performed differently on Unix-like and Windows systems. On Unix-like system, the autoscan process searches for a pattern in the process table. On Windows, autoscan search the system registry for a keys registered by products at installation time. HQ also supports another type of scan, referred to as a *file system* scan, which target the whole file system.

Although not typical, several HQ plugins automatically perform a file system scan, rather than an process table, to discover new server instances on Unix-like systems.

Element Structure

```
<scan> registry type <include
```

Parent Elements

The `<scan>` element can be a child of:

- `<server>`

Child Attributes and Elements

- `registry` - For a server type running on Windows, specifies a Windows registry path expression, for the key specified by the `<include>` element.
- `type` - In HQ's oracle, postgresql, and mysql plugins, the `type` attribute is set to the value "file" to cause the automatic discovery process on Windows platforms - performed at agent startup and periodically thereafter - to scan the full file system for the pattern specified using the `<include>` element.
- `<include>` - specifies a pattern or, if the `registry` attribute is present, a Windows registry key for which to scan.

Examples

Scan process table for Apache servers on Unix-like operating system

This `<scan>` element specifies multiple file path expressions for which to scan.

```
<scan> <include name="/**/bin/httpd"/> <include name="/**/bin/httpsd"/> <include name="/**/bin/apache"/> <include name="/**/bin/apache2"/> <include name="/**/sbin/httpd"/> <include name="/**/sbin/httpsd"/> <include name="/**/sbin/apache"/> <include name="/**/sbin/apache2"/> </scan>
```

Scan Windows registry for Apache servers

This `<scan>` element specifies a Windows registry path expression ("SOFTWARE\Apache Group\Apache\2.*") to scan, and a registry key ("ServerRoot") for which to scan.

```
<scan registry="SOFTWARE\Apache Group\Apache\2.*"> <include name="ServerRoot"/>
```

Scan Unix filesystem for MySQL servers

This `<scan>` element specifies a file path expression for which to scan the entire file system.

```
<scan type="file"> <include name="/**/bin/safe_mysqld"/> </scan>
```

3.3.15. script

*Topics marked with***relate to features available only in vFabric Hyperic.*

This page contains information that has not been reviewed. Click **Add Comment** at the bottom of the page to enter suggestions or corrections.

Element Overview

A `<script>` element is used to embed a the body of a script that the plugin uses in the plugin descriptor. (Scripts used by a plugin may instead be in a separate file, in the `\etc` directory of the plugin ja.

Parent Elements

A `<script>` element cam be a child of:

- `<plugin>` (root)

Child Attributes and Content

- `name` - Specifies the name of the script, without a file extension.
- The body of the script.

Example

The excerpt below from HQ's oracle plugin uses a `<script>` element to embed a script the plugin uses obtains availability and response time metrics for a tns ping service.

```
<plugin package="org.hyperic.hq.plugin.oracle"> <!-- extracted to: pdk/work/scripts/oracle/hq-tns-ping --> <script name="hq-tns-ping"> #!/bin/sh this_script=`basename $0` usage() { printf "usage: %s -p &lt;installpath&gt; -n &lt;tnsname&gt;\n" $this_script exit 4 } while getopts "p:n:" opt do case $opt in n) nflag=1 tnsname="$OPTARG";; p) pflag=1 path="$OPTARG";; ?) usage;; esac done if [ -z "$nflag" ] then usage elif [ -z "$pflag" ] then usage fi tnschk=`$path/bin/tnsping $tnsname` tnschk2=`echo $tnschk | grep -c OK` if [ "${tnschk2}" -eq 1 ] ; then tnschk3=`echo $tnschk | sed -e 's/.*(//' -e 's/).*//' -e 's/ msec//'` echo "$tnsname.TNSResponseTime=${tnschk3}" echo "$tnsname.Availability=1" exit 0 else echo "No TNS Listener on $tnsname" echo "$tnsname.Availability=0" exit 3 fi </script> ...... ...... ......
```

3.3.16. server

*Topics marked with***relate to features available only in vFabric Hyperic.*

This page contains information that has not been reviewed. Click **Add Comment** at the bottom of the page to enter suggestions or corrections.

Element Overview

```
<server> <filter> <property> <properties> <config> <metric> <metrics> <plugin> <actions> <scan> <service> <help>
```

Parent Elements

A `<server>` element can be a child of:

- `<platform>` - If a plugin manages a platform type in addition to its child server types, the `<server>` elements will be contained by the `<platform>` element.
- `<plugin>` root - If a plugin manages a server type, but not the server type's parent platform type, the `<server>` element will be in the descriptor root.

Child Attributes and Elements

- `name` - (Required) The name of the server type. Typically matches the name of the product the plugin manages. For example, "MySQL".
- `description` - (Optional) A brief description of the server type. If value is supplied it is presented in the header on resource instance pages. And in the general properties section of the Inventory tab for an instance.
- `version` - (Optional) Version of the server type. Optional. When supplied, this value is appended to the value of the name attribute to provide the server type. For example in the MySQL plugin, the version attribution value "4.x" is appended to the name attribute value "MySQL" to create the server type "MySQL 4.x".
- `platforms` - (Optional) Platforms types to which a user may add instances of the server type. Values include:
  - "Win32" - Instances of the server type may be added to platforms of type Win32.
  - "Linux" - Instances of the server type may be added to platforms of type Linux and Unix.
  - "Unix" - Instances of the server type may be added to platforms of type Linux and Unix.
- include - (Optional) Used to include by reference the contents of another `<server>` element whose version attribute matches the value of the include attribute. Optional.
- `virtual` - (Optional) This attribute is used only in the hq-netservices plugin. It is used to indicate that the server type is virtual - that is, an instance is explicitly configured simply to the parent of remote services that the agent monitors over the network. Optional. Values include "true" and "false". Defaults to false.
- `<filter>` - Defines variables that specify a pattern for the metric templates for a resource type (in this usage, a server type) and elements of the metric template, as desired. Strictly speaking, use of `<filter>` elements is optional, but most plugins use them because it is more efficient and less error-prone than defining the template for each metric explicitly. For more information, see `Section 3.3.2, “filter”` and Section 3.2.5, “Variables in Plugin Descriptors”.
- `<property>` - Defines variables that can be referenced within the plugin descriptor and also by plugin classes. The use of `<property>` element is optional. For more information see `Section 3.3.13, “property”`.
- `<properties>` - A container for one or more `<property>` elements. For more information, see `Section 3.3.12, “properties”`.
- `<config>` - Defines a configuration schema, which consists of a set of `<option>` elements for a resource type (in this usage, a platform type). One or more `<config>` elements can be defined.
- `<plugin>` - Defines a plugin function - auto-discovery, measurement, etc - supported for the resource type, and the class to use for the function. For more information, see `plugin` (Platform, Server and Service).

- `<metric>` - Defines the attributes of a metric, for the resource type, including its name, category, whether it is an indicator metric, how to collect it (metric template) etc. If the plugin performs measurement for the platform type, the plugin descriptor will contain a `metric` element for each metric it collects. For more information, see Section 3.3.6, “metric”.
- `<metrics>` - A container for one or more `<metric>` elements. You can assign a name to `<metric>` elements, and reference it by name in other parts of the plugin descriptor. The use of the `<metrics>` element is optional. For more information, see Section 3.3.7, “metrics”.
- `<help>` - Defines the help text that is displayed on the Configuration Properties for the resource type. For more information, see Section 3.3.4, “help”.
- `<service>` - Defines a service type. Like the `<server>` and `<platform>` elements - it define the information or rules that govern the functions - which may include autodiscovery, metric collection, log/configuration tracking, and control - the plugin performs for a specific service type, in terms of the HQ inventory model. For more information see Section 3.3.17, “service”.

Examples

<server> element example

Appendix B contains the complete descriptor file for HQ's tomcat-plugin. For an example of a complete `<server>` element, see the one that defines an Apache Tomcat 5.5 server, starting on page .

<server> element contents are defined by reference using include attribute

This excerpt from the descriptor for HQ's tomcat-plugin illustrates how you can include, by reference, the elements defined for one server type in another server. This is useful when the configuration options, resource properties, metrics, and so one are the same for two different versions of a managed resource. The tomcat-plugin manages two versions of Tomcat: 5.5 and 6.0. The two versions are identical from a management point of view. It makes for a simpler and shorter descriptor to define the 6.0 version by including the 5.5 definition.

```
<server name="Apache Tomcat" version="6.0" include="5.5">
```

server type is limited to specific platforms types using platforms attribute

This excerpt from HQ's mssql-plugin defines a server type that is valid only on Windows 32. The server type MsSQL 2000 will only be available in the **New Server** dialogs server type selector list when the currently selected platform is of type "Win32".

```
<server name="MsSQL" version="2000" platforms="Win32"> ........
```

Server type is defined as virtual

Note: This example documents the only usage of the virtual attribute in any HQ plugin. This rarely used attribute is unlikely to be used by most plugin developers.

This excerpt from HQ's netservices-plugin descriptor defines a server type that is virtual - there are no physical instances of the server type. Rather, an instance of a "Net Services" server is a user-configured element that exists only to be a parent of the network services monitored by the plugin.

```
<server name="Net Services" description="Network Services" virtual="true"> <plugin type="autoinventory" class="NetServicesDetector"/> <service name="InetAddress Ping" description="Java InetAddress Monitor">
```

3.3.17. service

*Topics marked with***relate to features available only in vFabric Hyperic.*

This page contains information that has not been reviewed. Click **Add Comment** at the bottom of the page to enter suggestions or corrections.

Element Overview

The `<service>` element defines a service type. Like the `<platform>` and `<server>` elements - it defines the information and rules enable the plugin to manage a particular resource type.

Element Structure

<service> name description internal <filter> <property> <properties> <config> <plugin> <metric> <metrics> <actions> <help>

Parent Elements

- `<plugin>` root - a `<service>` element in the descriptor root usually represents a platform service.
- `<server>` - For regular services (not platform services) the `<service>` element is contained by a `<server>` element.

Child Attributes and Elements

- `name` - The name of the service type. Required.
- `description` - Description of service type.
- `internal` - This attribute was used in early versions of HQ. Not currently used.
- `<filter>` - Defines variables that specify a pattern for the metric templates for a resource type (in this usage, a service type) and elements of the metric template, as desired. Strictly speaking, use of `<filter>` elements is optional, but most plugins use them because it is more efficient and less error-prone than defining the template for each metric explicitly. For more information, see `Section 3.3.2, “filter”` and Section 3.2.5, “Variables in Plugin Descriptors”.
- `<property>` - Defines variables that can be referenced within the plugin descriptor and also by plugin classes. The use of `<property>` element is optional. For more information see `Section 3.3.13, “property”`.
- `<properties>` - A container for one or more `<property>` elements. For more information, see `Section 3.3.12, “properties”`.
- `<config>` - Defines a configuration schema, which consists of a set of `<option>` elements for a resource type (in this usage, a platform type). One or more `<config>` elements can be defined.
- `<plugin>` - Defines a plugin function - auto-discovery, measurement, etc - supported for the resource type, and the class to use for the function. For more information, see `plugin` (Platform, Server and Service).
- `<metric>` - Defines the attributes of a metric, for the resource type, including its name, category, whether it is an indicator metric, how to collect it (metric template) etc. If the plugin performs measurement for the platform type, the plugin descriptor will contain a `metric` element for each metric it collects. For more information, see `Section 3.3.6, “metric”`.
- `<metrics>` - A container for one or more `<metric>` elements. You can assign a name to `<metric>` elements, and reference it by name in other parts of the plugin descriptor. The use of the `<metrics>` element is optional. For more information, see `Section 3.3.7, “metrics”`.

- `<help>` - Defines the help text that is displayed on the Configuration Properties for the resource type. For more information, see Section 3.3.4, “help”.

Examples

<service> defines service that runs on a server

The `<service>` element from HQ's tomcat plugin descriptor below specifies a service that runs on a Tomcat server. Note that the `<service>` element is a child of a he `<server>` element.

```
<server name="Apache Tomcat" version="5.5"> ...... ...... <service name="Web Module Stats"> <property
name="OBJECT_NAME"
value="${domain}:j2eeType=WebModule,name=*,J2EEApplication=*,J2EEServer=*"/> <plugin
type="autoinventory"/> <plugin type="control" class="org.hyperic.hq.product.jmx.MxControlPlugin"/
> <actions include="stop,start,reload"/> <!-- listen for JMX notifications --> <plugin
type="log_track" class="org.hyperic.hq.product.jmx.MxNotificationPlugin"/> <config> <option name="name"
description="Name Of Web Module" default=""/> <option name="J2EEApplication" description="J2EE
Application" default=""/> <option name="J2EEServer" description="J2EE Server" default=""/> </config>
<metric name="Availability" indicator="true"/> <metric name="Processing Time" alias="processingTime"
indicator="true" template="${OBJECT_NAME}:${alias}" units="sec"/> </service>
```

<service> defines platform service

The `<service>` element from HQ's netservices plugin descriptor below specifies a platform service. Note that the `<service>` element is in the root of the plugin descriptor.

```
<plugin name="netservices">

   ......
   ......

 <service name="HTTP"
            description="HTTP/S Monitor">

     <property name="DOMAIN" value="http,url.availability"/>

     <property name="port" value="80"/>
     <property name="sslport" value="443"/>

     <config include="http"/>

     <plugin type="collector"
             class="HTTPCollector"/>

     <plugin type="log_track"/>

     <metric name="Availability"
             indicator="true"/>

     <metric name="Response Code"
             indicator="true"
             template="${http.template}:${alias}"
             collectionType="static"/>

     <metric name="Response Time"
             indicator="true"
             category="Throughput"
             units="ms"/>

     <metric name="Last Modified"
             defaultOn="true"
```

```
              category="Availability"
              units="epoch-millis"
              collectionType="static"/>

      <metrics include="sockaddr-netstat"/>
    </service>
```

3.3.18. config

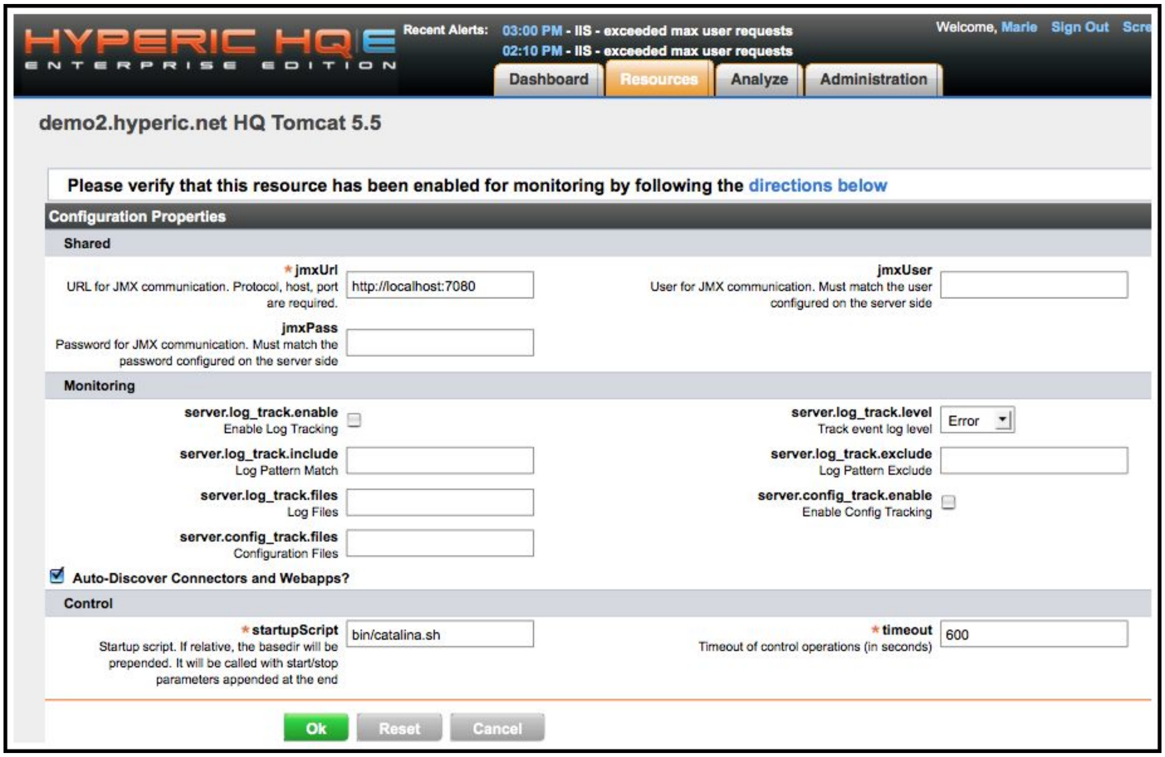
*Topics marked with*relate to features available only in vFabric Hyperic.*

Element Overview

Sometimes referred to as configuration schema, a `<config>` element contains a set of `<option>` elements for a resource type.

Key facts about about the options defined in a `<config>` element:

- The data defined as `<option>` elements within a `<config>` element appear in the HQ user interface and can be input or edited by an authorized user. The options are displayed on the **Configuration Properties** page for instances of the resource types. You specify the section of the **Configuration Properties** page where the options will appear with the type attribute. Click the thumbnail below to see an example:



Recent Alerts: 03:00 PM - IIS - exceeded max user requests
02:10 PM - IIS - exceeded max user requests

demo2.hyperic.net HQ Tomcat 5.5

Please verify that this resource has been enabled for monitoring by following the directions below

Configuration Properties

Shared

*jmxUrl — URL for JMX communication. Protocol, host, port are required. — http://localhost:7080

jmxUser — User for JMX communication. Must match the user configured on the server side

jmxPass — Password for JMX communication. Must match the password configured on the server side

Monitoring

server.log_track.enable — Enable Log Tracking □

server.log_track.level — Track event log level — Error

server.log_track.include — Log Pattern Match

server.log_track.exclude — Log Pattern Exclude

server.log_track.files — Log Files

server.config_track.enable — Enable Config Tracking □

server.config_track.files — Configuration Files

☑ Auto-Discover Connectors and Webapps?

Control

*startupScript — Startup script. If relative, the basedir will be prepended. It will be called with start/stop parameters appended at the end — bin/catalina.sh

*timeout — Timeout of control operations (in seconds) — 600

- A `<config>` element can be defined once in a descriptor, and be referenced by its name attribute in multiple resource elements with the descriptor. For example, if a plugin manages several different service types that have configuration options in common, you can define the `<config>` in the root of the plugin, and reference it in each <service> element to which the options apply.

- You can make a `<config>` element available to other plugin descriptors by setting its global attribute. HQ plugins define a number of useful global configuration schemas. They are documented in XREF.

- A <config> element whose global attribute is set to "true" can be accessed in a descriptor using special variables that are useful in defining a <metric> element's template attribute. (Configuration option values frequently form a portion of the metric template expression). This usage is described in XREF.

Parent Elements

- <plugin> (root) - Defines a set of configuration options that can by referenced by <config> elements within resource elements. The <config> must have a name attribute by which it can be referenced. A schema defined in the plugin root does not apply to any resource type unless it is explicitly referenced by a resource element (<platform>, <server>, or <service>).
- <platform> - Defines configuration options for the platform type.
- <server> - Defines configuration options for the server type.
- <service> - Defines configuration options for the service type specified.

Child Elements and Attributes

A <config> element contains these attributes an d elements:

- name - (Optional) Assigns a name to a <config> element by which the schema can be referenced from <config> elements in multiple resource elements (<platform>, <server>, or <service>). If the type attribute for the schema, described below, is set to global, the schema can also be referenced by <config> elements in resource elements in other plugin descriptors.
- include - (Optional) Includes one or more named configuration schemas defined in the current descriptor, or globally available schemas defined in other plugin descriptors.
- platform - (Optional) Specifies the platform type to which the configuration applies, for instance, win32, unix, or linux.
- type - (Optional) Affects two aspects of a configuration schema:
  - Where the options it defines appear on a resource's **Configuration Properties** page.
  - Whether the schema is globally available.

    Values for type include:
    No value specified - If you do not include the type attribute in a <config> element, the options it defines appear in the "Shared" section of the Configuration Properties page. The schema is not globally available.measurement - The options appear in the "Monitoring" section of the Configuration Properties page. The schema is not globally available.control - The options appear in the "Control" section of the Configuration Properties page. The schema is not globally available.global - Makes the schema globally available, so that other plugin descriptors can include it by reference. For the resource types that the schema applies to in the plugin that defines it, the options appear in the "Shared" section of the Configuration Properties page. The location of the options on the Configuration Properties page of resources defined by other plugins that reference the schema is controlled by the type attribute of the referencing <config> element.
- <option> - Defines a configurable resource attribute. A <config> element can contain one or more <option> elements. For details, see Section 3.3.8, “option”.

    Note: Optional if the the <config> element's include attribute (defined above)includes by reference the options specified in another <config> element.

<config> Examples

<config> element defines a globally available configuration schema

This excerpt from the jmx plugin, which monitors Sun JVMs, contains is a `<config>` element that defines a global configuration schema named "jmx":

```
<config name="jmx" type="global">
    <option name="jmx.url"
            description="JMX URL to MBeanServer"
            default="service:jmx:rmi:///jndi/rmi://localhost:1099/jmxrmi"/>

    <option name="jmx.username"
            description="JMX username"
            optional="true"/>

    <option name="jmx.password"
            description="JMX password"
            optional="true"
            type="secret"/>
  </config>
```

The The options in the jmx schema will appear in the "Shared" section of a Sun JVM's **Configuration Properties** page.

<config> elements define and reference a globally available configuration schema

This excerpt from the sql-query plugin contains two `<config>` elements:

```
<server name="SQL Query">
 .....
 .....
 <config name="sql" type="global">
        <option name="jdbcDriver" type="enum"
               description="JDBC Driver Class Name"
               default="org.postgresql.Driver"/>
        <option name="jdbcUrl"
               description="JDBC Connection URL"/>
               default="jdbc:postgresql://localhost:9432/hqdb"/>
        <option name="jdbcUser"
               description="JDBC User"/>
        <option name="jdbcPassword" type="secret"
               optional="true"
               description="JDBC Password"/>
     </config>

 <config type="measurement" include="sql">
  <option name="jdbcQuery"
   description="SQL query to run"/>
     </config>
</server>
```

Notes:

- The first `<config>` element defines a globally available configuration schema named sql.
- The second `<config>` element includes the options defined in the sql schema and defines an additional option explicitly. Because `type` attribute of the referencing `config` is set to "measurement", the options

appear in the "Monitoring" section of the **Configuration Properties** page for instances of the SQL Query server type.

<config> element references a globally available configuration schema

This `<config>` element from the db-mailhost plugin includes the globally available configuration schema named "sql". The options will appear in the "Shared" section of the **Configuration Properties** page for instances of the Mail Host server type. If the referencing `<config>` element set the `type` attribute to the value "measurement" or "control", the options would appear in the "Monitoring" or "Control" section instead.

```
<server name="Mail Host">
 .....
     <config include="sql"/>
```

<config> element includes two schemas by reference

This `<config>` element from the IIS plugin references two globally available configurations, protocol and url, and defines two options explicitly. Because the referencing `<config>` element's `type` attribute is set to "measurement", all of the options (those included by reference and those defined explicitly) appear in the "Monitoring" section of the **Configuration Properties** page for instances of the VHost service type.

```
<service name="VHost">
       .....
 .....
      <config type="measurement" include="protocol,url">
        <option name="hostheader"
                description="Host Header Name"/>
        <option name="iishost"
                description="Web Site Description"
                default="Default Web Site"/>
      </config>
```

3.4. Global Configuration Schemas Reference

*Topics marked with*relate to features available only in vFabric Hyperic.*

3.4.1. basicauth config

name	description	default	optional	type	Notes	Parent Schema
realm	Realm		true		Supply security realm if target site is password-protected.	basicauth
user	Username		true		Supply if target site is password-protected.	credentials
pass	Password		true	secret	Supply if target site is password-protected.	credentials

3.4.2. credentials config

name	description	default	optional	type	Notes	Parent Schema
user	Username		true		Supply if target site is password-protected.	credentials
pass	Password		true	secret	Supply if target site is password-protected.	credentials

3.4.3. dhcp config

<option> name	description	default	optional	type	Notes	Parent Schema
hwaddr	Hardware (MAC) Address		true		If MAC address is supplied, the plugin will issue a request for the IP address mapped to the MAC address on the DHCP Server. If the DHCP client request packet will be forwarded by a router to a DHCP server in a different subnet, the router must support static allocation.	dhcp
hostname	Hostname	localhost			Hostname of system that hosts the service to monitor. For example: mysite.com	sockaddr
port	Port	A default value for port is usually set for each type of network service by prop-			Port where service listens.	sockaddr

<option> name	description	default	optional	type	Notes	Parent Schema
		erties in the netservices plugin descriptor.				
sotimeout	Socket Timeout (in seconds)	10		int	The maximum amount of time the agent will wait for a response to a request to the remote service.	sockaddr

3.4.4. dns config

name	description	default	optional	type	Notes	Parent Schema
lookupname	Lookup Name	www.hyperic.com			Hostname to use in queries to the DNS service.	dns
pattern	Answer Match		true		This setting affects how the agent will determine availability of the DNS service. If you do not enter a value, if the agent can connect to the DNS service, the agent will report it to be available, even if no Answers are returned. If you enter an asterisk, the agent will report the DNS service to be available if the service returns an Answer to a query.	dns

name	description	default	optional	type	Notes	Parent Schema
					If you enter a regular expression or substring, the agent will report the DNS service to be available if it returns an Answer that matches the pattern. For information on DNS answers, see Information about DNS Answers	
type	Record type			enum	The DNS resource record type to use in queries to the DNS service. Selector list values are: A, ANY, CNAME, MX, NS, TXT. For record type definitions, see Wikipedia definition of DNS query types	dns
hostname	Hostname	localhost			Hostname of system that hosts the service to monitor. For example: mysite.com	sockaddr
port	Port	A default value for port is usually set for each type of network service by properties in the netservices			Port where service listens.	sockaddr

name	description	default	optional	type	Notes	Parent Schema
		plugin descriptor.				
sotimeout	Socket Timeout (in seconds)	10		int	The maximum amount of time the agent will wait for a response to a request to the remote service.	sockaddr

3.4.5. ftp config

name	description	default	optional	type	Notes	Parent Schema
hostname	Hostname	localhost			Hostname of system that hosts the service to monitor. For example: mysite.com	sockaddr
port	Port	A default value for port is usually set for each type of network service by properties in the netservices plugin descriptor.			Port where service listens.	sockaddr
sotimeout	Socket Timeout (in seconds)	10		int	The maximum amount of time the agent will wait for a response to a request to the remote service.	sockaddr
user	Username		true		Supply if target site is password-protected.	credentials
pass	Password		true	secret	Supply if target site is password-protected.	credentials

3.4.6. http config

name	description	default	optional	type	Notes	Parent Schema
method	Request Method	HEAD	false	enum	Allowable values: HEAD, GET. Method for checking availability. HEAD results in less network traffic. Use GET to return the body of the request response if you wish to specify a pattern to match in the response.	http
hostheader	Host Header	none	true		Use this option to set a "Host" HTTP header in the request, useful if you use name-based virtual hosting. Specify the host name of the Vhost's host, for example, `blog.hyperic.com`	http
follow	Follow Redirects	enabled	true	boolean	Enable if the HTTP request that HQ generates will be re-directed. This is important, because an HTTP server returns a different code for a redirect and HQ will assume that the HTTP service check is	http

name	description	default	optional	type	Notes	Parent Schema
					not available if it is a redirect, unless this redirect configuration is set.	
pattern	Response Match (substring or regex)	none	true		Specify a pattern or substring that HQ will attempt to match against the content in the HTTP response. This allows you to check that in addition to being available, the service is serving the content you expect.	http
proxy	Proxy Connection	none	true	If connection to the HTTP service will go through a proxy server, supply the hostname and port for the proxy server. For example, proxy.myco.com:3128		http
path	Path	/			Enter a value to monitor a specific page or file on the site. for example: /Support.html	url
ssl	Use SSL	false	true	boolean		ssl
hostname	Hostname	localhost			Hostname of system that hosts the service to monitor. For example: mysite.com	sockaddr

name	description	default	optional	type	Notes	Parent Schema
port	Port	A default value for port is usually set for each type of network service by properties in the netservices plugin descriptor.			Port where service listens.	sockaddr
sotimeout	Socket Timeout (in seconds)	10		int	The maximum amount of time the agent will wait for a response to a request to the remote service.	sockaddr
realm	Realm		true		Supply security realm if target site is password-protected.	basicauth
user	Username		true		Supply if target site is password-protected.	credentials
pass	Password		true	secret	Supply if target site is password-protected.	credentials

3.4.7. InetAddress Ping config

name	description	default	optional	type	Notes	Parent Schema
hostname	Hostname	localhost			Hostname of system that hosts the service to monitor. For example: mysite.com	netservices plugin descriptor
sotimeout	Socket Timeout (in seconds)	10		int	The maximum amount of time the agent will wait for a response to a	netservices plugin descriptor

name	description	default	optional	type	Notes	Parent Schema
					request to the remote service.	

3.4.8. jmx config

The options in the `jmx` global configuration schema configure the location and user credentials for connecting to an MBean server.

name	description	default	optional	type	Notes	Parent Schema
jmx.url	JMX URL to MBeanServer	service:jmx:rmi:///jndi/rmi://localhost:1099/jmxrmi	true			jmx
jmx.username	JMX username		true			jmx
jmx.password	JMX password			secret		jmx

3.4.9. ldap config

name	description	default	optional	type	Notes	Parent Schema
baseDN	Search Base				The top level of the LDAP directory tree, in X.500 format, for example: `o="hyperic",c=US`	ldap
bindDN	Bind DN		true		The user on the external LDAP server permitted to search the LDAP directory within the defined search base. Supply if directory requires authentication prior to search. Not necessary if anonymous	ldap

name	description	default	optional	type	Notes	Parent Schema
					searches are allowed.	
bindPW	Bind Password		true	secret	Password for user permitted to search the LDAP directory. Supply if directory requires authentication prior to search. Not necessary if anonymous searches are allowed.	ldap

filter	Search Filter		true		Use to specify entries to search using one or more boolean expressions, based on LDAP attributes. If you specify multiple expressions, prefix them with a logical operator. Here are two example filters:
					`(!(location=SFO*))`
					- matches if value of `location` attribute does not begin with "SFO"
					`||((facility=Mission)(facility=Financial))`
					- matches if value of `location` attribute is either "Mission" or "Financial"
						ldap

ssl	Use SSL	false	true	boolean		ssl

name	description	default	optional	type	Notes	Parent Schema
hostname	Hostname	localhost			Hostname of system that hosts the service to monitor. For example: mysite.com	sockaddr
port	Port	A default value for port is usually set for each type of network service by properties in the netservices plugin descriptor.			Port where service listens.	sockaddr
sotimeout	Socket Timeout (in seconds)	10		int	The maximum amount of time the agent will wait for a response to a request to the remote service.	sockaddr

3.4.10. ntp config

<option> name	description	default	optional	type	Notes	Parent Schema
hostname	Hostname	localhost			Hostname of system that hosts the service to monitor. For example: mysite.com	sockaddr
port	Port	A default value for port is usually set for each type of network service by properties in the netservices plugin descriptor.			Port where service listens.	sockaddr

<option> name	description	default	optional	type	Notes	Parent Schema
sotimeout	Socket Timeout (in seconds)	10		int	The maximum amount of time the agent will wait for a response to a request to the remote service.	sockaddr

3.4.11. pop3 config

<option> name	description	default	optional	type	Notes	Parent Schema
ssl	Use SSL	false	true	boolean		ssl
hostname	Hostname	localhost			Hostname of system that hosts the service to monitor. For example: mysite.com	sockaddr
port	Port	A default value for port is usually set for each type of network service by properties in the netservices plugin descriptor.			Port where service listens.	sockaddr
sotimeout	Socket Timeout (in seconds)	10		int	The maximum amount of time the agent will wait for a response to a request to the remote service.	sockaddr

name	description	default	optional	type	Notes	Parent Schema
user	Username		true		Supply if target site is password-protected.	credentials

name	description	default	optional	type	Notes	Parent Schema
pass	Password		true	secret	Supply if target site is password-protected.	credentials

3.4.12. protocol config

<option> name	description	default	optional	type	<include name (Selector Values)	Notes	Parent Schema
protocol	Connection Protocol			enum	http ftp socket		protocol

3.4.13. rpc config

<option> name in plugin	description in HQ UI	default	optional	type	Notes	Parent Schema
hostname	Hostname	localhost	no	string	IP address or domain name of the RPC host.	rpc
program	RPC program	nfs	no	string	The name by which the program is registered with its host portmapper.	rpc
version	RPC version	2	no	int		rpc
protocol	RPC protocol	any	no	enum	Specifies transport protocol to use to ping the PRC service. Values are "any", "tcp", "udp". If set to "any", Agent will try try TCP first and then UDP, if necessary.	rpc

3.4.14. smtp config

<option> name	description	default	optional	type	Notes	Parent Schema
ssl	Use SSL	false	true	boolean		ssl
hostname	Hostname	localhost			Hostname of system that hosts the service to monitor. For example: mysite.com	sockaddr
port	Port	A default value for port is usually set for each type of network service by properties in the netservices plugin descriptor.			Port where service listens.	sockaddr
sotimeout	Socket Timeout (in seconds)	10		int	The maximum amount of time the agent will wait for a response to a request to the remote service.	sockaddr

3.4.15. snmp config

name	description	default	optional	type	Notes	Parent Schema
snmpIp	SNMP agent IP address	127.0.0.1	no		The IP address of the SNMP agent for the network device.	snmp
snmpPort	SNMP agent port	161	no	port	The port the SNMP agent uses.	snmp
snmpTransport	SNMP Transport		no	enum	Values: • udp • tcp	snmp
snmpVersion	SNMP Version		no	enum	Values:	snmp

name	description	default	optional	type	Notes	Parent Schema
					• v2c • v1 • v3	
snmp Community (v1 and v2c only)	SNMP Community	public				snmp
snmpUser (v3 only)	SNMP Security Name	username	true		The SNMP security name to use when communicating with the remote SNMP agent.	snmp
snmpSecurity Context (v3 only)	SNMP Context Name	hqadmin	true		The name of the SNMP context that provides access to the remote management information.	snmp
snmpAuthType (v3 only)	SNMP Authentication Protocol	none	true	enum	The SNMP authentication protocol to use for communicating with the remote SNMP agent. Values: • none • MD5 • SHA	snmp
snmpPassword (v3 only)	SNMP Authentication Passphrase		true	secret	The SNMP authorization passphrase to use for communicating with the remote SNMP agent.	snmp
snmpPrivacy Type	SNMP Privacy Protocol		true	enum	The SNMP Privacy Proto-	snmp

name	description	default	optional	type	Notes	Parent Schema
(v3 only)					col HQ Server should use for communicating with the remote SNMP agent. Values: • none, • DES, • 3DES • AES-128 • AES-192 • AES-256	
snmpPrivacy PassPhrase (v3 only)	SNMP Privacy Passphrase		true	secret	The SNMP privacy passphrase configured for use when communicating with the remote SNMP agent.	snmp

3.4.16. sockaddr config

<option> name	description	default	optional	type	Notes	Parent Schema
hostname	Hostname	localhost			Hostname of system that hosts the service to monitor. For example: mysite.com	sockaddr
port	Port	A default value for port is usually set for each type of network service by properties in the netservices plugin descriptor.			Port where service listens.	sockaddr

<option> name	description	default	optional	type	Notes	Parent Schema
sotimeout	Socket Timeout (in seconds)	10		int	The maximum amount of time the agent will wait for a response to a request to the remote service.	sockaddr

3.4.17. sql config

<option> name	description	default	optional	type	Notes	Parent Schema
jdbcDriver	JDBC Driver Class Name	org.postgresql.Driver		enum		sql
jdbcUrl	JDBC Connection URL	jdbc:postgresql://localhost:9432/hqdb			<!-- jdbcUrl option overridden in SQLMeasurementPlugin -->	sql
jdbcUser	JDBC User			secret		sql
jdbcPassword	JDBC Password		true			sql

3.4.18. ssh config

name	description	default	optional	type	Notes	Parent Schema
hostname	Hostname	localhost			Hostname of system that hosts the service to monitor. For example: mysite.com	sockaddr
port	Port	A default value for port is usually set for each type of network service by properties in the netservices plugin descriptor.			Port where service listens.	sockaddr

name	description	default	optional	type	Notes	Parent Schema
sotimeout	Socket Timeout (in seconds)	10		int	The maximum amount of time the agent will wait for a response to a request to the remote service.	sockaddr
user	Username		true		Supply if target site is password-protected.	credentials
pass	Password		true	secret	Supply if target site is password-protected.	credentials

3.4.19. ssl config

<option> name	description	default	optional	type	Notes	Parent Schema
ssl	Use SSL	false	true	boolean		ssl

3.4.20. sslprotocol config

<option> name	description	default	optional	type	Notes	Parent Schema
sslprotocol	SSL Protocol			enum	Allowable values: SSL, TLS	sslprotocol

3.4.21. sslsockaddr config

<option> name	description	default	optional	type	Notes	Parent Schema
ssl	Use SSL	false	true	boolean		ssl
hostname	Hostname	localhost			Hostname of system that hosts the service to monitor. For example: mysite.com	sockaddr

<option> name	description	default	optional	type	Notes	Parent Schema
port	Port	A default value for port is usually set for each type of network service by properties in the netservices plugin descriptor.			Port where service listens.	sockaddr
sotimeout	Socket Timeout (in seconds)	10		int	The maximum amount of time the agent will wait for a response to a request to the remote service.	sockaddr

3.4.22. tcp config

name	description	default	optional	type	Notes	Parent Schema
hostname	Hostname	localhost			Hostname of system that hosts the service to monitor. For example: mysite.com	sockaddr
port	Port	A default value for port is usually set for each type of network service by properties in the netservices plugin descriptor.			Port where service listens.	sockaddr
sotimeout	Socket Timeout (in seconds)	10		int	The maximum amount of time the agent will wait for a response to a request to the remote service.	sockaddr

3.4.23. url config

<option> name	description	default	optional	type	Notes	Parent Schema
path	Path	/			Enter a value to monitor a specific page or file on the site. for example: /Support.html	url
ssl	Use SSL	false	true	boolean		ssl
hostname	Hostname	localhost			Hostname of system that hosts the service to monitor. For example: mysite.com	sockaddr
port	Port	A default value for port is usually set for each type of network service by properties in the netservices plugin descriptor.			Port where service listens.	sockaddr
sotimeout	Socket Timeout (in seconds)	10		int	The maximum amount of time the agent will wait for a response to a request to the remote service.	sockaddr

3.5. Example Plugin Descriptor

1 <?xml version="1.0"?> 2 3 <!DOCTYPE plugin [4 <!ENTITY process-metrics SYSTEM "/pdk/plugins/process-metrics.xml"> 5]> 6 7 <plugin package="org.hyperic.hq.plugin.tomcat" name="tomcat"> 8 <classpath> 9 <include name="pdk/lib/mx4j"/> 10 <!-- relative to auto-discovered installpath (see PROC_HOME_PROPERTY) --> 11 <include name="server/lib/commons-modeler-*.jar"/> 12 </classpath> 13 14 <filter name="template" 15 value="${OBJECT_NAME}:${alias}"/> 16 17 <metrics name="Thread Metrics"> 18 <metric name="Thread Count" 19 alias="ThreadCount" 20 indicator="false" 21 template="${OBJECT_NAME}:${alias}" 22 units="none" 23 collectionType="trendsup"/> 24 <metric name="Current Thread Cpu Time" 25 alias="CurrentThreadCpuTime" 26 indicator="false" 27 template="${OBJECT_NAME}:${alias}" 28 units="ms" 29 collectionType="trendsup"/> 30 <metric name="Current Thread User Time" 31 alias="CurrentThreadUserTime" 32 indicator="false" 33 template="${OBJECT_NAME}:${alias}" 34 units="ms" 35 collectionType="trendsup"/> 36

<metric name="Daemon Thread Count" 37 alias="DaemonThreadCount" 38 indicator="false" 39 template="${OBJECT_NAME}:${alias}" 40 units="none" 41 collectionType="dynamic"/> 42 <metric name="Peak Thread Count" 43 alias="PeakThreadCount" 44 indicator="false" 45 template="${OBJECT_NAME}:${alias}" 46 units="none" 47 collectionType="static"/> 48 </metrics> 49 50 <metrics name="OS Metrics"> 51 <metric name="Free Swap Space Size" 52 alias="FreeSwapSpaceSize" 53 indicator="true" 54 template="${OBJECT_NAME}:${alias}" 55 units="B" 56 collectionType="dynamic"/> 57 <metric name="Free Physical Memory Size" 58 alias="FreePhysicalMemorySize" 59 indicator="true" 60 template="${OBJECT_NAME}:${alias}" 61 units="B" 62 collectionType="dynamic"/> 63 <metric name="Process Cpu Time" 64 alias="ProcessCpuTime" 65 indicator="true" 66 template="${OBJECT_NAME}:${alias}" 67 units="ms" 68 collectionType="trendsup"/> 69 <metric name="Open File Descriptor Count" 70 alias="OpenFileDescriptorCount" 71 indicator="false" 72 template="${OBJECT_NAME}:${alias}" 73 units="none" 74 collectionType="dynamic"/> 75 </metrics> 76 77 <metrics name="Runtime Metrics"> 78 <metric name="UpTime" 79 alias="Uptime" 80 indicator="true" 81 template="${OBJECT_NAME}:${alias}" 82 units="ms" 83 collectionType="static"/> 84 </metrics> 85 86 <server name="Apache Tomcat" 87 version="5.5"> 88 89 <property name="VERSION_FILE" 90 value="server/lib/catalina-storeconfig.jar"/> 91 92 <plugin type="autoinventory" 93 class="org.hyperic.hq.product.jmx.MxServerDetector"/> 94 <property name="domain" 95 value="Catalina"/> 96 97 <property name="OBJECT_NAME" 98 value="java.lang:type=Runtime"/> 99 <metrics include="Runtime Metrics"/> 100 101 <property name="OBJECT_NAME" 102 value="java.lang:type=OperatingSystem"/> 103 <metrics include="OS Metrics"/> 104 105 <property name="OBJECT_NAME" 106 value="java.lang:type=Threading"/> 107 <metrics include="Thread Metrics"/> 108 109 <!-- remove -DISABLED to enable server discovery for 5.5 --> 110 <property name="PROC_HOME_PROPERTY" 111 value="catalina.base-DISABLED"/> 112 113 <property name="DEFAULT_CONF" 114 value="conf/server.xml"/> 115 116 <property name="DEFAULT_LOG_FILE" 117 value="logs/catalina.out"/> 118 119 <plugin type="log_track" 120 class="org.hyperic.hq.product.Log4JLogTrackPlugin"/> 121 122 <property name="DEFAULT_PROGRAM" 123 value="bin/catalina.sh"/> 124 125 <plugin type="control" 126 class="org.hyperic.hq.product.jmx.MxServerControlPlugin"/> 127 128 <property name="start.args" 129 value="start"/> 130 131 <property name="stop.args" 132 value="stop"/> 133 134 <config> 135 <option name="jmx.url" 136 description="JMX URL to MBeanServer" 137 default="service:jmx:rmi:///jndi/rmi://localhost:6969/jmxrmi"/> 138 <option name="jmx.username" 139 description="JMX username" 140 optional="true" 141 default="system"/> 142 <option name="jmx.password" 143 description="JMX password" 144 optional="true" 145 default="manager" 146 type="secret"/> 147 <option name="ptql" 148 description="PTQL for Tomcat Process" 149 default="State.Name.eq=java,Args.*.ct=catalina.home"/> 150 </config> 151 152 <plugin type="measurement" 153 class="org.hyperic.hq.product.jmx.MxMeasurementPlugin"/> 154 155 <metric name="Availability" 156 template="sigar:Type=ProcState,Arg=%ptql%:State" 157 indicator="true"/> 158 159 <service name="Web Module Stats"> 160 <property name="OBJECT_NAME" 161 value="${domain}:j2eeType=WebModule,name=*,J2EEApplication=*,J2EEServer=*"/> 162 <plugin type="autoinventory"/> 163 <plugin type="control" 164 class="org.hyperic.hq.product.jmx.MxControlPlugin"/> 165 <actions include="stop,start,reload"/> 166 <!-- listen for JMX notifications --> 167 <plugin type="log_track" 168 class="org.hyperic.hq.product.jmx.MxNotificationPlugin"/> 169 170 <config> 171 <option name="name" 172 description="Name Of Web Module" 173 default=""/> 174 <option name="J2EEApplication" 175 description="J2EE Application" 176 default=""/> 177 <option name="J2EEServer" 178 description="J2EE Server" 179 default=""/> 180 </config> 181 <metric name="Availability" 182 indicator="true"/> 183 <metric name="Processing Time" 184 alias="processingTime" 185 indicator="true" 186 template="${OBJECT_NAME}:${alias}" 187 units="sec"/> 188 </service> 189 190 <service name="Thread Pools"> 191 <property name="OBJECT_NAME" 192 value="${domain}:type=ThreadPool,name=*"/> 193 194 <plugin type="autoinventory"/> 195 196 <plugin type="control" 197 class="org.hyperic.hq.product.jmx.MxControlPlugin"/> 198 <actions include="start,shutdown"/> 199 200 <!-- listen for JMX notifications --> 201 <plugin type="log_track" 202 class="org.hyperic.hq.product.jmx.MxNotificationPlugin"/> 203 204 <config> 205 <option name="name" 206 description="Listener Name" 207 default=""/> 208 </con-

fig> 209 <metric name="Availability" 210 indicator="true"/> 211 <metric name="Current Thread Count" 212 alias="currentThreadCount" 213 indicator="true" 214 template="${OBJECT_NAME}:${alias}" 215 units="none"/> 216 <metric name="Current Thread Busy" 217 alias="currentThreadsBusy" 218 indicator="true" 219 template="${OBJECT_NAME}:${alias}" 220 units="none"/> 221 </service> 222 223 <service name="Servlet Monitor"> 224 <property name="OBJECT_NAME" 225 value="${domain}:j2eeType=Servlet,name=*,WebModule=*,J2EEApplication=*,J2EEServer=*"/> 226 227 <plugin type="autoinventory"/> 228 229 <!-- listen for JMX notifications --> 230 <plugin type="log_track" 231 class="org.hyperic.hq.product.jmx.MxNotificationPlugin"/> 232 233 <config> 234 <option name="WebModule" 235 description="Deployed Module" 236 default=""/> 237 <option name="J2EEApplication" 238 description="J2EE Application" 239 default=""/> 240 <option name="J2EEServer" 241 description="J2EE Server" 242 default=""/> 243 </config> 244 245 <metric name="Availability" 246 indicator="true"/> 247 <metric name="Class Load Time" 248 alias="classLoadTime" 249 indicator="false" 250 template="${OBJECT_NAME}:${alias}" 251 units="none"/> 252 <metric name="Error Count" 253 alias="errorCount" 254 indicator="true" 255 template="${OBJECT_NAME}:${alias}" 256 collectionType="trendsup" 257 units="none"/> 258 <metric name="Load Time" 259 alias="loadTime" 260 indicator="false" 261 template="${OBJECT_NAME}:${alias}" 262 units="none"/> 263 <metric name="Processing Time" 264 alias="processingTime" 265 indicator="false" 266 template="${OBJECT_NAME}:${alias}" 267 collectionType="trendsup" 268 units="none"/> 269 <metric name="Request Count" 270 alias="requestCount" 271 indicator="true" 272 template="${OBJECT_NAME}:${alias}" 273 collectionType="trendsup" 274 units="none"/> 275 </service> 276 277 <service name="JSP Monitor"> 278 <property name="OBJECT_NAME" 279 value="${domain}:type=JspMonitor,name=jsp,WebModule=*,J2EEApplication=*,J2EEServer=*"/> 280 <plugin type="autoinventory"/> 281 282 <!-- listen for JMX notifications --> 283 <plugin type="log_track" 284 class="org.hyperic.hq.product.jmx.MxNotificationPlugin"/> 285 286 <config> 287 <option name="WebModule" 288 description="Deployed Module" 289 default=""/> 290 <option name="J2EEApplication" 291 description="J2EE Application" 292 default=""/> 293 <option name="J2EEServer" 294 description="J2EE Server" 295 default=""/> 296 </config> 297 <metric name="Availability" 298 indicator="true"/> 299 <metric name="JSP Count" 300 alias="jspCount" 301 indicator="true" 302 template="${OBJECT_NAME}:${alias}" 303 collectionType="trendsup" 304 units="none"/> 305 <metric name="JSP Reload Count" 306 alias="jspReloadCount" 307 indicator="true" 308 template="${OBJECT_NAME}:${alias}" 309 collectionType="trendsup" 310 units="none"/> 311 </service> 312 313 <service name="Global Request Processor"> 314 <property name="OBJECT_NAME" 315 value="${domain}:type=GlobalRequestProcessor,name=*"/> 316 317 <plugin type="autoinventory"/> 318 319 <!-- listen for JMX notifications --> 320 <plugin type="log_track" 321 class="org.hyperic.hq.product.jmx.MxNotificationPlugin"/> 322 323 <config> 324 <option name="name" 325 description="Listener Name" 326 default=""/> 327 </config> 328 329 <metric name="Availability" 330 indicator="true"/> 331 <metric name="Bytes Sent" 332 alias="bytesSent" 333 indicator="false" 334 template="${OBJECT_NAME}:${alias}" 335 collectionType="trendsup" 336 units="none"/> 337 <metric name="Bytes Received" 338 alias="bytesReceived" 339 indicator="false" 340 template="${OBJECT_NAME}:${alias}" 341 collectionType="trendsup" 342 units="none"/> 343 <metric name="Error Count" 344 alias="errorCount" 345 indicator="true" 346 template="${OBJECT_NAME}:${alias}" 347 collectionType="trendsup" 348 units="none"/> 349 <metric name="Processing Time" 350 alias="processingTime" 351 indicator="true" 352 template="${OBJECT_NAME}:${alias}" 353 collectionType="trendsup" 354 units="none"/> 355 <metric name="Request Count" 356 alias="requestCount" 357 indicator="true" 358 template="${OBJECT_NAME}:${alias}" 359 collectionType="trendsup" 360 units="none"/> 361 </service> 362 363 <service name="Cache"> 364 <property name="OBJECT_NAME" 365 value="${domain}:type=Cache,host=*,path=*"/> 366 367 <plugin type="autoinventory"/> 368 369 <!-- listen for JMX notifications --> 370 <plugin type="log_track" 371 class="org.hyperic.hq.product.jmx.MxNotificationPlugin"/> 372 373 <config> 374 <option name="path" 375 description="Context Path of Deployed Application" 376 default=""/> 377 <option name="host" 378 description="Hostname" 379 default=""/> 380 description="Associated Java Class" 381 default=""/> 382 </config> 383 384 <metric name="Availability" 385 indicator="true"/> 386 <metric name="Access Count" 387 alias="accessCount" 388 indicator="true" 389 template="${OBJECT_NAME}:${alias}" 390

```
collectionType="trendsup" 391 units="none"/> 392 <metric name="Hits Count" 393 alias="hitsCount" 394 indicator="true" 395 template="${OBJECT_NAME}:${alias}" 396 collectionType="trendsup" 397 units="none"/> 398 </service> 399 400 <service name="DataSource Pool"> 401 <property name="OBJECT_NAME" 402 value="${domain}:type=DataSource,path=*,host=*,class=*,name=*"/> 403 404 <plugin type="autoinventory"/> 405 406 <!-- listen for JMX notifications --> 407 <plugin type="log_track" 408 class="org.hyperic.hq.product.jmx.MxNotificationPlugin"/> 409 410 <config> 411 <option name="path" 412 description="Context Path of Deployed Application" 413 default=""/> 414 <option name="host" 415 description="Hostname" 416 default=""/> 417 <option name="class" 418 description="Associated Java Class" 419 default=""/> 420 <option name="name" 421 description="Name of Attribute" 422 default=""/> 423 </config> 424 425 <metric name="Availability" 426 indicator="true"/> 427 <metric name="Idle DataSource Connections" 428 alias="numIdle" 429 indicator="true" 430 template="${OBJECT_NAME}:${alias}" 431 units="none"/> 432 <metric name="Active DataSource Connections" 433 alias="numActive" 434 indicator="true" 435 template="${OBJECT_NAME}:${alias}" 436 units="none"/> 437 </service> 438 439 <service name="Java Process Metrics"> 440 <config> 441 <option name="process.query" 442 default="%ptql%" 443 description="PTQL for Tomcat Java Process"/> 444 </config> 445 <metric name="Availability" 446 template="sigar:Type=ProcState,Arg=%process.query%:State" 447 indicator="true"/> 448 &process-metrics; 449 </service> 450 451 <service name="HTTP"> 452 <config include="http"/> 453 <filter name="template" 454 value="${http.template}:${alias}"/> 455 456 <metric name="Availability" 457 indicator="true"/> 458 459 <metric name="Inbound Connections" 460 indicator="true"/> 461 462 <metric name="Outbound Connections" 463 indicator="true"/> 464 </service> 465 </server> 466 467 <server name="Apache Tomcat" 468 version="6.0" 469 include="5.5"> 470 <property name="VERSION_FILE" 471 value="lib/catalina-ha.jar"/> 472 <!-- derive installpath from -Dcatalina.base=... --> 473 <property name="PROC_HOME_PROPERTY" 474 value="catalina.base"/> 475 <plugin type="autoinventory" 476 class="org.hyperic.hq.product.jmx.MxServerDetector"/> 477 </server> 478 479 <!-- ==================== Plugin Help =========================== --> 480 <help name="Apache Tomcat"> 481 <![CDATA[ 482 <p> 483 <h3>Configure Apache Tomcat for JMX</h3> 484 </p> 485 To configure Tomcat for JMX monitoring see http://tomcat.apache.org/tomcat-${product.version}-doc/monitoring.html. 486 <br> 487 For a quick down and dirty method follow these instructions, 488 <br> 489 in <installpath>/bin/catalina.sh add: 490 <br> 491 [ $1 != "stop" ] &amp;&amp; 492 JAVA_OPTS="-Dcom.sun.management.jmxremote \ 493 <br> 494 -Dcom.sun.management.jmxremote.port=6969 \ 495 <br> 496 -Dcom.sun.management.jmxremote.ssl=false \ 497 <br> 498 -Dcom.sun.management.jmxremote.authenticate=false $JAVA_OPTS" 499 <br> 500 export JAVA_OPTS 501 <br> 502 From there restart Tomcat and that is it. 503 </p> 504 ]]> 505 </help> 506 <help name="Apache Tomcat 5.5" include="Apache Tomcat"/> 507 <help name="Apache Tomcat 6.0" include="Apache Tomcat"/>
```

3.6. Running and Testing Plugins from Command Line

*Topics marked with***relate to features available only in vFabric Hyperic.*

3.6.1. Invoking Plugins from the Command Line

This page has instructions, with examples, for running HQ resource plugins from the command line - useful for both testing and documenting plugins.

You can test a plugin's syntax and invoke any management function the plugin supports:

- **Auto-discovery** - Run the discovery function for one or all plugins in the HQ Agent's `plugin` directory.
- **Control** - Run a plugin control action on a resource.
- **Metric collection** - Collect metrics for a resource.
- **Event Tracking** - Watch for log or configuration change events for a resource.

- **Fetch live system data** - Run supported system commands - the same commands available in the Live Exec view for platforms - to obtain CPU, filesystem, and other system data.

You can also generate documentation for a plugin:

- **Help** - Output the configuration help specified in plugin descriptor `<help>` element for each resource type, for one or all plugins.

- **Metric documentation** - Output metric documentation for each resource type, for one or all plugins.

3.6.2. hq-pdk.jar Command Syntax

The command for running a plugin from the command line is formed like this:

```
java -jar AgentVersion/bundles/AgentBundle/pdk/lib/hq-pdk-VERSION.jar -m Method -a
 MethodAction -p PluginName -t ResourceType -Doption=value
```

The subsections below the command arguments.

-m Method specifies the method to run

- One of:
  - `lifecycle`
  - `discover`
  - `metric`
  - `control`
  - `track`
  - `livedata`
  - `generate`

For details and functionality and usage of each method, see hq-pdk.jar Methods and Functionality.

-p PluginName specifies the plugin to run

Supply the product portion of the plugin name, without the "-plugin.jar" or "-plugin.xml" portion. For example, to run `jboss-plugin.jar`, you specify `-p jboss`.

- Required by these methods:
  - `lifecycle`
  - `metric`
  - `control`
  - `track`
- Optional for:
  - `discover`

    - `generate`
- Not supported for:
    - `livedata`

Note: If you use a generated properties file to supply resource properties, you do not have to specify the plugin to run on the command line, because a resource properties identifies the plugin. See Generating and Using Resource Properties Files.

-t ResourceType specifies the target resource type

Supply the name of a resource type managed by the plugin you are running enclosed in quotes if the the name includes spaces, for instance, "JBoss 4.2".

- Required by these methods:
    - `metric`
    - `control`
    - `track`
    - `livedata`
- Not supported for:
    - `discover`
    - `generate`

Note: If you use a generated properties file to supply resource properties, you do not have to specify resource type on the command line, because a resource properties specifies the resource type name for the resource. See Generating and Using Resource Properties Files .

-a MethodAction specifies a method argument

Supply an argument supported or required by the the method called. For example, when you run the `track` method, you specify whether you want to track log or configuration events by including either `-a log` or `-a config` in the command line. The arguments for each method are documented in hq-odj.jar Methods and Functionality.

-D Option = Value sets a property value

Include a `-D` Option = Value in the command line for each property you wish to set.

Supply:

- The value of a resource property required by the method called.
    - **Note:** Rather than supply each resource property on the command line, you can reference a generated properties file. For more information see Generating and Using Resource Properties Files.

- The value of an agent or system property that governs agent behavior or plugin execution. For more information, see Properties for Controlling Agent Behavior and Plugin Execution.

3.6.3. Generating and Using Resource Properties Files

Must plugin methods require the values of one or more resource properties to run. For instance, to fetch metrics for a PostgreSQL table, the `metric` method requires to know the URL and database user credentials for the parent PostresSQL server, and the name of the table.

- jdbcUser
- jdbcPassword
- table
- jdbcUrl

Each property that a method requires for a resource type is defined in an `<option>` element in the XML descriptor for the plugin that manages it. To ease the process of testing a plugin, you can supply required properties in a file, instead of on the command line.

When you run the `discover` method with the `properties` method argument, the agent will create a properties file for each resource instance it discovers. The properties file for a resource contains a name-value pair for each resource property required to run plugin methods.

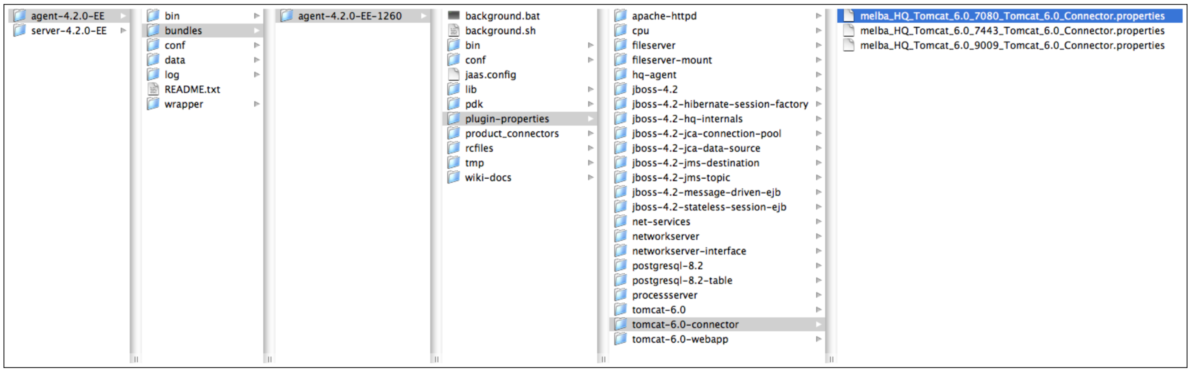
Configurable properties that the user must supply must be added to the properties file or supplied on the command line. For example, to check the results of tracking log messages that do not contain a particular string, you must supply the string on the command line. Specifically, you need to set the value of **server.log_track.exclude** which is null by default.

This command supplies some command options and resource properties using the `melba_HQ_jBoss_4.x.properties` file (for details, see Content of Properties Files), and sets the value of **server.log_track.exclude** on the command line

```
java -jar java -jar AgentVersion/bundles/AgentBundle/pdk/lib/hq-pdk-shared-VERSION.jar
-m track plugin-properties/jboss-4.2/melba_HQ_jBoss_4.x.properties
-Dserver.log_track.exclude=just kidding
```

Names and Location of Properties Files

The `discover` method's `properties` action writes configuration data for each discovered resource in a directory tree whose root directory - `plugin-properties` - is in your current working directory.



Note:

- The `plugin-properties` folder contains a subdirectory for each resource type discovered. The folder name is the resource type name, with spaces replaced by dashes, for example, "Tomcat-6.0-Connector"
- Each resource type folder contains a file for each instance of that type discovered. The file name is the full name of the resource instance, with spaces replaced by underscore characters,for example "melba_HQ_Tomcat_6.0_7080_Tomcat 6.0_Connector."

Content of Properties Files

When you run the `metric`, `control`, or `track` method on a resource you must supply resource configuration data - either explicitly on the command line, or using the properties file for the resource.

Use of a properties file far more convenient than defining the configuration data on the command line. The properties file also simplifies the command by defining the values that you would otherwise set with the `-p` and `-t` options.

The discovery results saved for a JBoss 4.2 server is shown below.

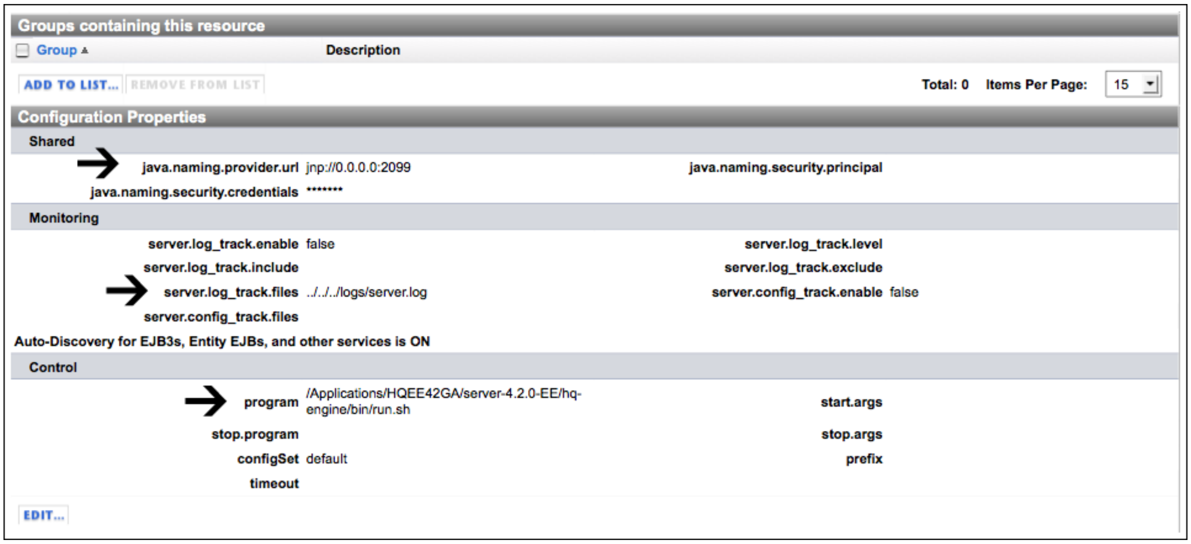
```
# same as '-p "jboss"'
dumper.plugin=jboss
# same as '-t "JBoss 4.2"'
dumper.type=JBoss 4.2
\#melba HQ JBoss 4.x
\#Fri Jan 22 10:38:10 PST 2010
java.naming.provider.url=jnp://0.0.0.0:2099
program=/Applications/HQEE42GA/server-4.2.0-EE/hq-engine/bin/run.sh
server.log_track.files=../../../logs/server.log
configSet=default
```

Note:

The properties file contains:

- The resource's resource type name and the product portion of the name of the plugin that manages it:
  - `dumper.plugin` - Specifies the product portion of the plugin name; equivalent to setting the plugin name in the command line with `-p`.
  - `dumper.type` - Specifies the resource type name; equivalent to setting the resource type in the command line with `-t`.
- Resource configuration data that is required to use the `metric`, `track`, or `control` methods on a resource - without the properties file, you must supply values for required configuration options in the command line when you run the method. The Jboss properties file above supplies values for:
  - `java.naming.provider.url`
  - `program`
  - `server.log_track.files`

In the HQ user interface, these options appear on the **Configuration Properties** page for a JBoss server that has been added to inventory. Click the thumbnail to see a screenshot.



Groups containing this resource

Group ▲ Description

Total: 0 Items Per Page: 15

Configuration Properties

Shared

java.naming.provider.url jnp://0.0.0.0:2099
java.naming.security.principal
java.naming.security.credentials *******

Monitoring

server.log_track.enable false
server.log_track.level
server.log_track.include
server.log_track.exclude
server.log_track.files ../../../logs/server.log
server.config_track.enable false
server.config_track.files

Auto-Discovery for EJB3s, Entity EJBs, and other services is ON

Control

program /Applications/HQEE42GA/server-4.2.0-EE/hq-engine/bin/run.sh
start.args
stop.program
stop.args
configSet default
prefix
timeout

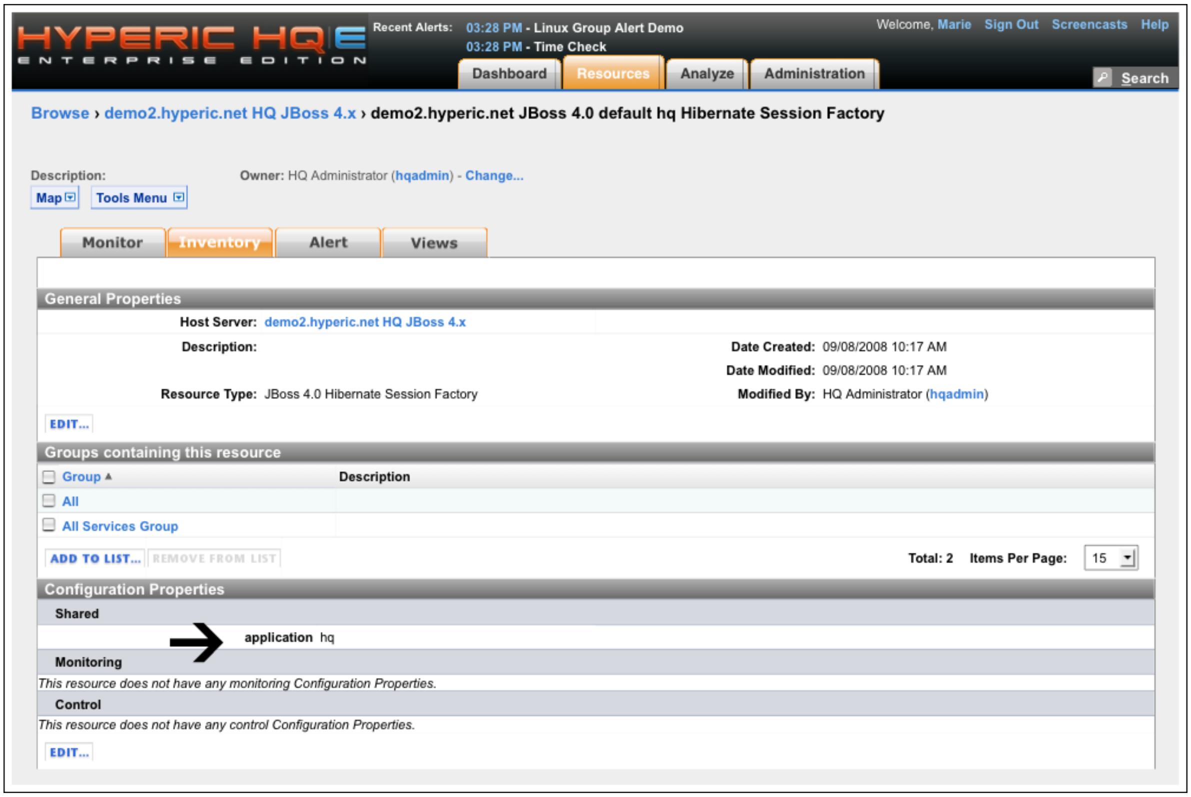
Inherited Resource Properties

Some resource properties may be inherited from a parent resource. For example, the properties file for a JBoss 4.2 Hibernate Session Factory service, shown below, includes all of the properties discovered for its parent - a JBoss 4.2 server.

```
# same as '-p "jboss"'
dumper.plugin=jboss
# same as '-t "JBoss 4.2 Hibernate Session Factory"'
dumper.type=JBoss 4.2 Hibernate Session Factory
#192.168.0.12 JBoss 4.2 default hq Hibernate Session Factory
#Fri Jan 22 12:56:05 PST 2010
java.naming.provider.url=jnp://0.0.0.0:2099
program=/Applications/HQEE42GA/server-4.2.0-EE/hq-engine/bin/run.sh
application=hq
server.log_track.files=../../../logs/server.log
configSet=default
```

The only service-level property in the properties file above is **Application**.

Click the thumbnail to see a screenshot of the service's **Configuration Properties**.



Recent Alerts: 03:28 PM - Linux Group Alert Demo
03:28 PM - Time Check

Browse › demo2.hyperic.net HQ JBoss 4.x › demo2.hyperic.net JBoss 4.0 default hq Hibernate Session Factory

Description:
Owner: HQ Administrator (hqadmin) - Change...

General Properties

Host Server: demo2.hyperic.net HQ JBoss 4.x
Description:
Resource Type: JBoss 4.0 Hibernate Session Factory
Date Created: 09/08/2008 10:17 AM
Date Modified: 09/08/2008 10:17 AM
Modified By: HQ Administrator (hqadmin)

Groups containing this resource

Group ▲	Description
All	
All Services Group	

Total: 2 Items Per Page: 15

Configuration Properties

Shared
application hq

Monitoring
This resource does not have any monitoring Configuration Properties.

Control
This resource does not have any control Configuration Properties.

3.6.4. Properties for Controlling Agent Behavior and Plugin Execution

You can use

-DOption=Value

to set any agent or system property. The table below lists some properties that are useful when you run a plugin from the command line.useful Properties for Command Line

Property	**Description**
`log`	Set the log level. log=debug
`output.dir`	Override default output directory default
`plugins.include`	This agent property tells the HQ Agent to load a specific plugin, and only that agent before executing the method. Otherwise, when you run `hq-pdk-shared-VERSION.jar` the HQ Agent will load all of the plugins in the plugin directory.
`plugins.exclude`	This agent property give the agent a list of plugins to not load before executing the method. The HQ Agent will load all other plugins in the plugin directory.
`exec.sleep`	You can use this system property to override its default value when you are testing a script plugin. By default, `exec.sleep` is 30 seconds. If your script might take longer than that to run, it is useful to increase the value while you check the plugin out.

3.6.5. hq-pdk.jar Methods and Functionality

lifecyle: Initialize and Check Syntax

The `lifecycle` method loads a plugin and reports errors found.

The syntax for the `lifecycle` method is:

```
$ java -jar bundles/agent-VERSION/pdk/lib/hq-pdk-VERSION.jar -p PluginName -m lifecycle -
Dplugins.include=PluginName
```

where:

- `-p`
  PluginName
  identifies the plugin to run, by the product portion of the plugin name, e.g. "jboss"; the plugin must reside in the agent's `plugin` directory.

- `-Dplugins.include=`
  PluginName
  ensures that only the specified plugin is loaded; otherwise all plugins are loaded. (Use of this command option is recommended, although not required.)

Example: Results of lifecycle Method on a Plugin with No Errors

This command runs the `lifecycle` method for the jboss plugin; no errors are found.

```
$ java -jar bundles/agent-VERSION/pdk/lib/hq-pdk-VERSION.jar -m lifecycle -p jboss -
Dplugins.include=jboss
```

Example: Results of lifecycle Method on a Plugin with Errors

This command runs the `lifecycle` method for the websphere plugin; errors are reported.

```
$ java -jar bundles/agent-VERSION/pdk/lib/hq-pdk-VERSION.jar -m lifecycle -p websphere -
Dplugins.include=websphere
WARN [main] [MetricsTag] MsSQL 2000 include not found: mssql-cache
WARN [main] [MetricsTag] WebSphere 6.1 include not found: WebSphere 6.0
WARN [main] [MetricsTag] WebSphere 6.1 Application include not found: WebSphere 6.0
 Application
WARN [main] [MetricsTag] WebSphere 6.1 EJB include not found: WebSphere 6.0 EJB
WARN [main] [MetricsTag] WebSphere 6.1 Webapp include not found: WebSphere 6.0 Webapp
WARN [main] [MetricsTag] WebSphere 6.1 Connection Pool include not found: WebSphere 6.0
 Connection Pool
WARN [main] [MetricsTag] WebSphere 6.1 Thread Pool include not found: WebSphere 6.0 Thread
 Pool
WARN [main] [MetricsTag] WebSphere Admin 6.1 include not found: WebSphere Admin 6.0
```

discover: Auto-Discover Resources and Configuration

The `discover` method can be run for one or all plugins. It returns key attributes for each resource discovered, including the values of the resource's configuration options, to the terminal window or to a properties files. If you save discovery results to a file, you can use the file to supply required resource configuration data when you run another method that requires the resource's configuration data.

The syntax for the `discover` option is:

```
{{java -jar bundles/agent-VERSION/pdk/lib/hq-pdk-VERSION.jar -m discover -p PluginName -a
 properties
```

where:

- `-p P`
  luginName
  specifies the plugin to run; if not specified, discovery is performed for all plugins in the agent's `plugin` directory.
- `-a properties` writes discovery results to files; otherwise results are returned only to the terminal window. See Names and Location of Properties Files and Content and Purpose of Properties Files for more information.

To...	Use...	Notes
run discovery for all plugins	`-m discover`	Results are returned to the terminal window.
run discovery for one plugin, in this case Jboss	`-m discover -p jboss`	Results are returned to the terminal window.
run discovery for JBoss and save results to files	`-m discover -p jboss -a properties`	Results are written to files, as well as to the terminal window.
run discovery for all plugins and save results to files	`-m discover -a properties`	Results are written to files, as well as to the terminal window.

metric: Fetch Metrics

The `metric` method can be used to:

- Fetch the metric template and metric value for each metric for a resource managed by the plugin, or just for those metrics that:
  - are collected by default,
  - are of a specified metric category, or
  - are indicator metrics
- Return the metric template only (and not the metric value) for the metrics.
- Fetch metrics repeatedly - as many times as you specify - and return the time it took to perform the fetches for each metric.

Syntax for metric Method Using Resource Properties File

The syntax for running the `metric` method using a properties file to supply resource configuration data is:

```
java -jar bundles/agent-VERSION/pdk/lib/hq-pdk-VERSION.jar -m metric plugin-properties/
ResourceTypeDirectory/ResourceName.properties -a translate -Dmetric-collect=default -
Dmetric-indicator=true -Dmetric-cat=CATEGORY -Dmetric-iter=ITERATIONS
```

where:

- `plugin-properties/`
  ResourceTypeDirectory
  `/`

ResourceName
`.properties` is the path to the file generated when the resource was discovered using the `properties` action of the `discover` method. The properties file supplies the values for:

  - `-p`
    PluginName
    specifies the product portion of the plugin name.
  - `-t`
    ResourceType
    is the resource type name.
  - the value of configuration options for the resource

- `-a translate` causes metric templates, but not metric values, to be returned. If `-a translate` is not specified both metric templates and metric values are returned.
- optionally, one (and only one) of these options is specified to limit the metrics returned (if none of these options is specified, all metrics are returned):
  - `-Dmetric-collect=default` limits results to metrics whose "defaultOn" attribute is set to "true".
  - `-Dmetric-indicator=true` limits results to metrics whose "indicator" attribute is set to "true".
  - `-Dmetric-cat=`
    CATEGORY
    limits results to metrics of the specified category - one of AVAILABILITY, UTILIZATION, THROUGHPUT, or PERFORMANCE.
- `-Dmetric-iter=`
  ITERATIONS
  causes a the time (in millis) to collect a metric repeatedly to be reported, rather than the metric value.

Syntax for metric Method Specifying Configuration Data on Command Line

The syntax for running the `metric` method suppling resource configuration data on the command line is:

```
java -jar bundles/agent-VERSION/pdk/lib/hq-pdk-VERSION.jar -m metric -p PluginName -t
 ResourceType -a translate -Dmetric-collect=default -Dmetric-indicator=true -Dmetric-
cat=CATEGORY -Dmetric-iter=ITERATIONS -DOption=Value
```

where:

- `-p`
  PluginName
  specifies the product portion of the plugin name.
- `-t`
  ResourceType
  is the resource type name.
- `-a translate` causes metric templates, but not metric values, to be returned. If `-a translate` is not specified both metric templates and metric values are returned.
- optionally, one (and only one) of these options is specified to limit the metrics returned (if none of these options is specified, all metrics are returned):

- `-Dmetric-collect=default` limits results to metrics whose "defaultOn" attribute is set to "true".
- `-Dmetric-indicator=true` limits results to metrics whose "indicator" attribute is set to "true".
- `-Dmetric-cat=`
  CATEGORY
  limits results to metrics of the specified category - one of AVAILABILITY, UTILIZATION, THROUGHPUT, or PERFORMANCE.

- `-Dmetric-iter=`
  ITERATIONS
  causes a the time (in millis) to collect a metric repeatedly to be reported, rather than the metric value.

- a `-D`
  Option
  =
  Value
  specifies the value of a resource configuration option; the command line must include a "`D`
  Option
  =
  Value
  " for *each* resource configuration option.

Example Invocations

In these examples, only the method invocation and command options are shown. The `java -jar bundles/agent-`
VERSION/pdk/lib/hq-pdk-VERSION.jar
portion of the command is not shown.

To...	Use...	Notes
fetch metrics for a jboss server	-m metric -p jboss -t "JBoss 4.2" -m metric -Djava.naming.provider.url=jnp://0.0.0.0:2099 -Dserver.log_track.files=../../../logs/server.log -Dprogram=/Applications/HQEE42GA/server-4.2.0-EE/hq-engine/bin/run.sh	In this example, resource configuration data is supplied on the command line.
fetch metrics for the jboss server supplying the configuration data using a properties file	-m metric plugin-properties/jboss-4.2/melba_HQ_jBoss_4.x.properties	In this example, resource configuration data is supplied by a properties file

limit the results to indicator metrics	add the following to the command line:
	`-Dmetric-collect=default`
		If you use this option, do not use either of these:
		`-Dmetric-cat=`
		CATEGORY
		`-Dmetric-indicator=true`

limit the results to metrics of a specific category	add the following to the command line:
	`-Dmetric-cat=`
	CATEGORY
	where:
	CATEGORY
		If you use this, do not use either of these:
		`-Dmetric-collect=default`
		`-Dmetric-indicator=true`

To...	Use...	Notes
	is one of AVAILABILITY, UTILIZATION, THROUGHPUT, or PERFORMANCE	

limit the results to indicator metrics	add the following to the command line:
	`-Dmetric-indicator=true`
		If you use this, do not use either of these:
		`-Dmetric-collect=default`
		`-Dmetric-cat=CATEGORY`

collect each metric multiple times and report how long it took to do so (in millis) instead reporting the metric value	add the following to the command line:
	`-Dmetric-iter=`
	ITERATIONS
	where
	ITERATIONS
	is the number of times to run `getValue` for each metric.
		May be used in conjunction with one of:
		`-Dmetric-collect=default`
		`-Dmetric-cat=CATEGORY`
		`-Dmetric-indicator=true`

to fetch the metric template (but not metrics) for the jboss server	-m metric plugin-properties/jboss-4.2/ melba_HQ_jBoss_4.x.properties -a translate	

Results returned by -metric method's default action

Below is an excerpt from the results of running the default action of the metric method. Both metric templates and metric values are returned.

```
JBoss 4.2 Availability:
jboss.system:service=MainDeployer:StateString:java.naming.provider.url=jnp
%3A//0.0.0.0%3A2099,java.naming.security.principal=%java.naming.security.principal
%,java.naming.security.credentials=
=>100.0%125.015.0365.9 MB
```

Note: Colons In metric templates appear as "%3A" in the results above. |

Results returned by metric method's translate action

This is an excerpt from the results of running the the metric method with the translate action. Metric templates, but not metric values, are returned.

```
JBoss 4.2 Availability:
jboss.system:service=MainDeployer:StateString:java.naming.provider.url=jnp
%3A//0.0.0.0%3A2099,java.naming.security.principal=%java.naming.security.principal
%,java.naming.security.credentials=%java.naming.security.credentials%
JBoss 4.2 Active Thread Count:
jboss.system:type=ServerInfo:ActiveThreadCount:java.naming.provider.url=jnp
%3A//0.0.0.0%3A2099,java.naming.security.principal=%java.naming.security.principal
%,java.naming.security.credentials=%java.naming.security.credentials%
JBoss 4.2 Active Thread Group Count:
jboss.system:type=ServerInfo:ActiveThreadGroupCount:java.naming.provider.url=jnp
%3A//0.0.0.0%3A2099,java.naming.security.principal=%java.naming.security.principal
%,java.naming.security.credentials=%java.naming.security.credentials%
JBoss 4.2 JVM Free Memory:
```

Note: Colons In metric templates appear as "%3A" in the results above.

track: Watch for Log or Configuration Events

The `track` method watches for a log or configuration event.

The syntax for the `track` method is:

```
java -jar /bundles/agent-VERSION/pdk/lib/hq-pdk-VERSION.jar -p PluginName -t "ResourceType"
 -m track -a TrackAction -Dserver.config_track.files=TrackFiles
```

where:

- PluginName
  identifies the plugin to run.
- ResourceType
  specifies a particular resource type managed by the plugin.
- TrackAction
  specifies whether to track log events or configuration events:
  - `log`
  - `track`

Note: You could use a generated properties file instead of explicitly specifying:

- `-p`
  PluginName
- `-t "`
  ResourceType
- `-Dserver.config_track.files=TrackFiles`

For an example of using a properties file see Syntax for metric Method Using Resource Properties File.

Examples

In these examples only the method invocation and command options are shown. The `java -jar bundles/agent-`
VERSION/pdk/lib/hq-pdk-VERSION.jar
portion of the command is not shown.

To...	**Use...**	**Notes**
track changes made to the `/etc/httpd/httpd.conf` file for an "Apache 2.0" server	-p apache -t "Apache 2.0" -m track -a config -Dserver.config_track.files=/etc/httpd/httpd.conf	
track log entries to `/var/log/httpd/error_log` for an "Apache 2.0" server	-p apache -t "Apache 2.0" -m track -a log -Dserver.log_track.files=/var/log/httpd/error_log	
To exercise other configurable tracking behaviors, for instance to check the results of tracking log	add the desired property definition to thecommand line, for example: -Dserver.log_track.exclude=String	

To...	Use...	Notes
messages that do or do not contain a particular string, you must set the value of **server.log_track.exclude** or **server.log_track.include** on the command line.		

control: Perform a Control Action on a Resource

The `control` method can be used to perform a supported control action on a resource instance.

The syntax for the `control` method is:

```
java -jar /bundles/agent-VERSION/pdk/lib/hq-pdk-VERSION.jar -m control -a ControlAction -p
 PluginName -t ResourceType -Doption=value
```

where:

- PluginName
  identifies the plugin to run.
- ResourceType
  specifies a particular resource type managed by the plugin.
- ControlAction
  specifies the action to perform.
- a `-D`
  option=value
  for each resource configuration option sets the option's value.

Note: You could use a generated properties file instead of explicitly specifying resource configuration options. For an example of using a properties file see Syntax for metric Method Using Resource Properties File.

Examples

In these examples, only the method invocation and command options are shown. . The `java -jar bundles/agent-`
VERSION/pdk/lib/hq-pdk-VERSION.jar
portion of the command is not shown.

To...	Use...	Notes
run the "removeAllMessages" control action on a JMS destination	-m control -p jboss -a removeAllMessages -t "JBoss 4.0 JMS Destination" -Djms.destination=DLQ -Djava.naming.naming.url=jnp://localhost:1099	In this example resource configuration data is supplied on the command line.
run the "start" control action on a JBoss server	-m control -a start plugin-properties/jboss-4.0/hammer_JBoss_4.0_all.properties	In this example resource configuration data is supplied from a generated properties file.
to exercise other configurable control behaviors, for instance to supply optional control action argu-	add desired property to the command line, for example:	

To...	Use...	Notes
ments, you must set value of the appropriate property, for instance **start.args** or **stop.args** on the command line.	-Dstart.args=SupportedStartArgument	

livedata: Fetch Live System Information

The `livedata` method obtains live system metrics and platform data. The data is returned live, rather than obtained from the HQ database, and is provided in XML format.

The syntax for the `livedata` method is:

```
java -jar bundles/agent-VERSION/pdk/lib/hq-pdk-VERSION.jar -m livedata -t ResourceType -a
 LiveDataAction
```

where:

- ResourceType
  is an operating system platform type name, one of:
  - `Linux`
  - `AIX`
  - `MacOSX`
  - `HPUX`
  - `NetBSD`
  - `Win32`
  - `OpenBSD`
  - `Solaris`
- LiveDataAction
  is a one of these SIGAR commands:
  - `cpuinfo` - CPU information for each CPU discovered.
  - `cpuperc` - CPU percentage usage.
  - `top` - Display system resource utilization summaries and process information. SIGAR's top command.requires a PTQL query to select which processes to display.
  - `netstat` return network connections information.
  - `ifconfig` - Display network interface configuration and metrics.
  - `who` - return information about all users currently on the local system.
  - `df` - Report filesytem disk space usage.

Results of cpu Command

This is an example of the results returned by the `cpu` command.

```
$java -jar bundles/agent-VERSION/pdk/lib/hq-pdk-VERSION.jar -m livedata -t MacOSX -a cpu


26101070
12908410
829010
236665670
0
0
0
0
276504160


31011100
8567260
929150
235996420
0
0
0
0
276503930
```

cpuperc Command Results

This is an example of the results returned by SIGAR's `cpuperc` command.

```
$java -jar bundles/agent-VERSION/pdk/lib/hq-pdk-VERSION.jar -m livedata -t MacOSX -a cpu


26101070
12908410
829010
236665670
0
0
0
0
276504160


31011100
8567260
929150
235996420
0
0
0
0
276503930
```

ifconfig Command Results

This is an excerpt of example results returned by SIGAR's `ifconfig` command.

```
$java -jar bundles/agent-VERSION/pdk/lib/hq-pdk-VERSION.jar -Dplugins.include=jboss -m
 livedata -t MacOSX -a ifconfig
```

```


lo0
00:00:00:00:00:00
Local Loopback
lo0
127.0.0.1
127.0.0.1
0.0.0.0
255.0.0.0
32841
16384
0


1718492625
6594769
0
0
-1
-1
1718494642
6594766
0
-1
-1
0
-1
0
```

Note: The output contains a `<org.hyperic.hq.plugin.system.NetInterfaceData>` element for each network interface discovered.

who Command Results

This is an excerpt of the results returned by SIGAR's `who` command.

```
$ java -jar bundles/agent-VERSION/pdk/lib/hq-pdk-VERSION.jar -Dplugins.include=jboss -m
 livedata -t MacOSX -a who


mmcgarry
console

1264178819


mmcgarry
ttyp1

1264185170


mmcgarry
ttyp2

1264181578
```

generate: Create Plugin Documentation

The `generate` method generates documentation from the plugin descriptor.

The syntax for the `generate` method is:

```
java -jar bundles/agent-VERSION/pdk/lib/hq-pdk-VERSION.jar -p PluginName -m generate -a
 GenerateAction
```

where:

- PluginName
  - identifies a plugin to document. If not specified, the action will be applied to all plugins.
- GenerateAction
  specifies the type of docuementation to generate, one of:
  - `metrics-wiki` - Write a Confluence Wiki formatted summary of supported metrics to a file.
  - `metrics-xml` - Output an XML-formatted summary of supported metrics for a resource type to stdout.
  - `metrics-txt` Output a text formatted summary of supported metrics to stdout.
  - `help` - Output the contents of the `<help>` element for each resource type in plugin descriptor(s) to HTML files in the `./plugin-help` directory.

Syntax

In these examples, only the method invocation and command options are shown. The `java -jar bundles/agent-`
VERSION/pdk/lib/hq-pdk-VERSION.jar
portion of the command is not shown.

To...	Use...	Notes
document metrics in Confluence wiki format for all resource types in all plugins	`-m generate -a metrics wiki`	
document metrics in text format for all resource types in all plugins	`-m generate -a metrics-txt`	
document metrics in XML format for all resource types in all plugins	`-m generate -a metrics-xml`	
generate a help page for all resource types in all plugins	`-m generate -a help`	
limit results to the resource types managed by a single plugin, in this example, jboss	add `-p jboss` to the command line.	

3.6.6. Running Protocol Checks from Command Line

Running a plugin from the command line is useful for testing and documenting plugins. Command line execution is also an expeditious way to fetch metrics on-demand.

For example, you can run HQ's netservices plugin from the command line to check the availability of a variety of network service types. The netservices plugin can monitor remote resources of the following types.

- HTTP
- POP3
- IMAP

- SMTP
- FTP
- LDAP
- DNS
- SSH
- NTP
- DHCP
- SNMP
- RPC
- InetAddress Ping
- TCP Socket

If you monitor a resource of one of the types shown above on an on-going basis, you configure it as a platform service on a platform of your choice, whose HQ Agent will perform the remote availability checks and metric collection. To enable monitoring, you supply resource configuration data - the hostname of the service, at a minimum. If you wish to run the plugin at the command line, you must supply the required configuration data on the command line.

For an overview of HQ's network service monitoring features and documentation on the configuration options for each network service type, see Monitoring Network Services.

The command below runs the netservices plugin's `metric` method for a remote LDAP server.

```
java -jar bundles/agent-VERSION/pdk/lib/hq-pdk-shared-VERSION.jar -m metric -p netservices
-t LDAP -Dplugins.include=netservices -Dhostname=192.168.1.1 -Dssl=false -Dport=389
-DbaseDN=dc=foobar,dc=co,dc=nz -DbindDN=cn=root,c=foobar,dc=co,dc=nz
-DbindPW=changeme -Dfilter=uidNumber
```

Note that the value of each configuration option for the LDAP service type is supply with a -D argument. The configuration options are defined on the LDAP Platform Service page.

3.6.7. Generating and Using Resource Properties Files

Must plugin methods require the values of one or more resource properties to run. For instance, to fetch metrics for a PostgreSQL table, the `metric` method requires to know the URL and database user credentials for the parent PostresSQL server, and the name of the table.

- jdbcUser
- jdbcPassword
- table
- jdbcUrl

Each property that a method requires for a resource type is defined in an `<option>` element in the XML descriptor for the plugin that manages it. To ease the process of testing a plugin, you can supply required properties in a file, instead of on the command line.

When you run the discover method with the properties method argument, the agent will create a properties file for each resource instance it discovers. The properties file for a resource contains a name-value pair for each resource property required to run plugin methods.

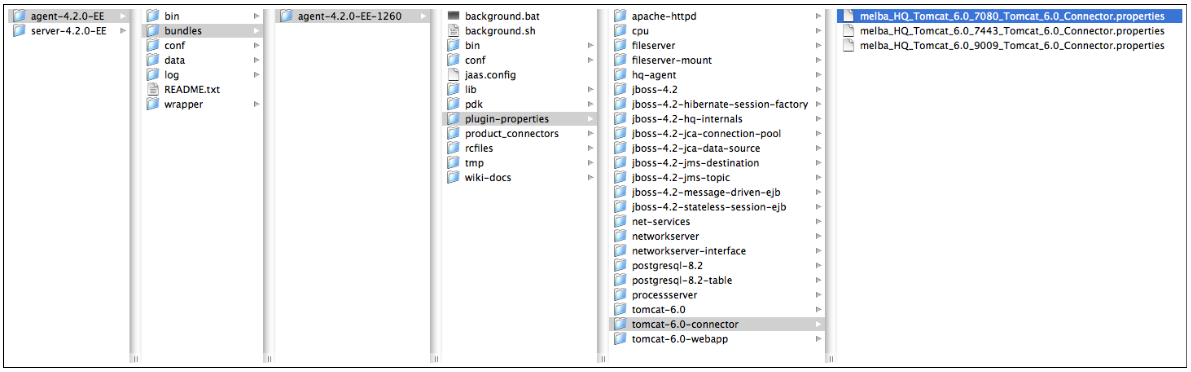
Configurable properties that the user must supply must be added to the properties file or supplied on the command line. For example, to check the results of tracking log messages that do not contain a particular string, you must supply the string on the command line. Specifically, you need to set the value of **server.log_track.exclude** which is null by default.

This command supplies some command options and resource properties using the melba_HQ_jBoss_4.x.properties file (for details, see Content of Properties Files), and sets the value of **server.log_track.exclude** on the command line

```
java -jar java -jar AgentVersion/bundles/AgentBundle/pdk/lib/hq-pdk-shared-VERSION.jar
-m track plugin-properties/jboss-4.2/melba_HQ_jBoss_4.x.properties
-Dserver.log_track.exclude=just kidding
```

Names and Location of Properties Files

The discover method's properties action writes configuration data for each discovered resource in a directory tree whose root directory - plugin-properties - is in your current working directory.



Note:

- The plugin-properties folder contains a subdirectory for each resource type discovered. The folder name is the resource type name, with spaces replaced by dashes, for example, "Tomcat-6.0-Connector"
- Each resource type folder contains a file for each instance of that type discovered. The file name is the full name of the resource instance, with spaces replaced by underscore characters,for example "melba_HQ_Tomcat_6.0_7080_Tomcat 6.0_Connector."

Content of Properties Files

When you run the metric, control, or track method on a resource you must supply resource configuration data - either explicitly on the command line, or using the properties file for the resource.

Use of a properties file far more convenient than defining the configuration data on the command line. The properties file also simplifies the command by defining the values that you would otherwise set with the -p and -t options.

The discovery results saved for a JBoss 4.2 server is shown below.

```
# same as '-p "jboss"'
dumper.plugin=jboss
```

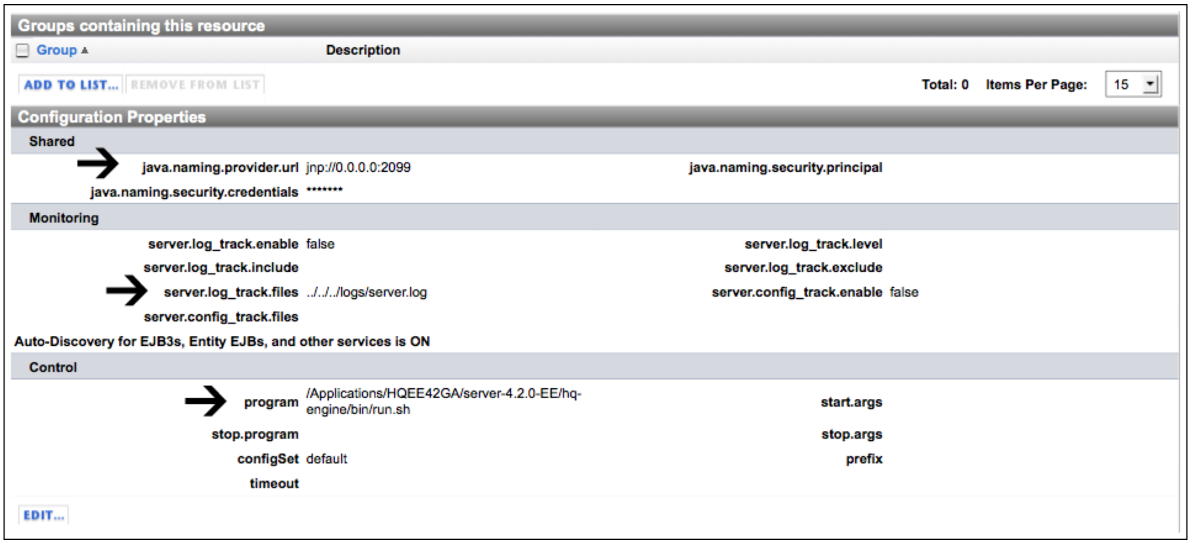
```
# same as '-t "JBoss 4.2"'
dumper.type=JBoss 4.2
\#melba HQ JBoss 4.x
\#Fri Jan 22 10:38:10 PST 2010
java.naming.provider.url=jnp://0.0.0.0:2099
program=/Applications/HQEE42GA/server-4.2.0-EE/hq-engine/bin/run.sh
server.log_track.files=../../../logs/server.log
configSet=default
```

Note:

The properties file contains:

- The resource's resource type name and the product portion of the name of the plugin that manages it:
  - `dumper.plugin` - Specifies the product portion of the plugin name; equivalent to setting the plugin name in the command line with `-p`.
  - `dumper.type` - Specifies the resource type name; equivalent to setting the resource type in the command line with `-t`.
- Resource configuration data that is required to use the `metric`, `track`, or `control` methods on a resource - without the properties file, you must supply values for required configuration options in the command line when you run the method. The Jboss properties file above supplies values for:
  - `java.naming.provider.url`
  - `program`
  - `server.log_track.files`

In the HQ user interface, these options appear on the **Configuration Properties** page for a JBoss server that has been added to inventory. Click the thumbnail to see a screenshot.



Shared

java.naming.provider.url	jnp://0.0.0.0:2099	java.naming.security.principal	
java.naming.security.credentials	*******		

Monitoring

server.log_track.enable	false	server.log_track.level	
server.log_track.include		server.log_track.exclude	
server.log_track.files	../../../logs/server.log	server.config_track.enable	false
server.config_track.files			

Auto-Discovery for EJB3s, Entity EJBs, and other services is ON

Control

program	/Applications/HQEE42GA/server-4.2.0-EE/hq-engine/bin/run.sh	start.args	
stop.program		stop.args	
configSet	default	prefix	
timeout			

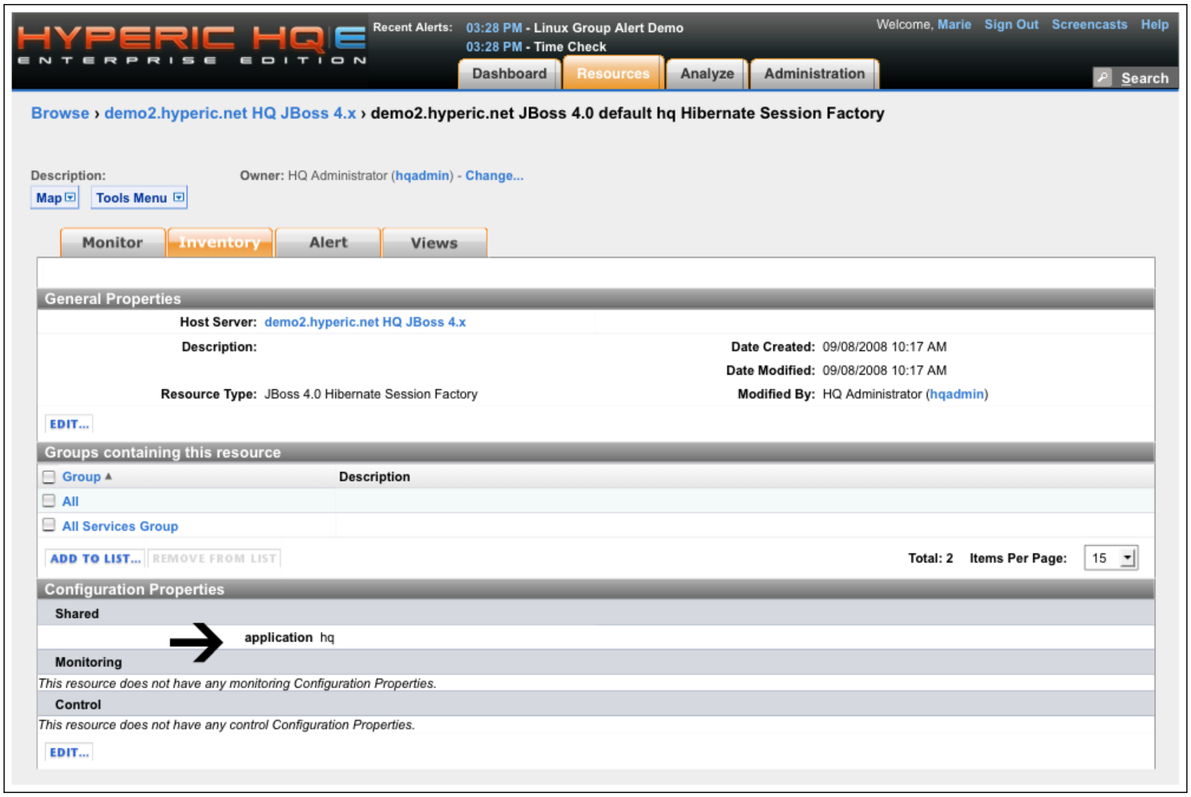
Inherited Resource Properties

Some resource properties may be inherited from a parent resource. For example, the properties file for a JBoss 4.2 Hibernate Session Factory service, shown below, includes all of the properties discovered for its parent - a JBoss 4.2 server.

```
# same as '-p "jboss"'
dumper.plugin=jboss
# same as '-t "JBoss 4.2 Hibernate Session Factory"'
dumper.type=JBoss 4.2 Hibernate Session Factory
#192.168.0.12 JBoss 4.2 default hq Hibernate Session Factory
#Fri Jan 22 12:56:05 PST 2010
java.naming.provider.url=jnp://0.0.0.0:2099
program=/Applications/HQEE42GA/server-4.2.0-EE/hq-engine/bin/run.sh
application=hq
server.log_track.files=../../../logs/server.log
configSet=default
```

The only service-level property in the properties file above is **Application**.

Click the thumbnail to see a screenshot of the service's **Configuration Properties**.



Recent Alerts: 03:28 PM - Linux Group Alert Demo
03:28 PM - Time Check

Browse › demo2.hyperic.net HQ JBoss 4.x › demo2.hyperic.net JBoss 4.0 default hq Hibernate Session Factory

Description: Owner: HQ Administrator (hqadmin) - Change...

General Properties

Host Server: demo2.hyperic.net HQ JBoss 4.x

Description:

Date Created: 09/08/2008 10:17 AM

Date Modified: 09/08/2008 10:17 AM

Resource Type: JBoss 4.0 Hibernate Session Factory

Modified By: HQ Administrator (hqadmin)

EDIT...

Groups containing this resource

Group ▲	Description
□ All	
□ All Services Group	

ADD TO LIST... REMOVE FROM LIST Total: 2 Items Per Page: 15

Configuration Properties

Shared

application hq

Monitoring

This resource does not have any monitoring Configuration Properties.

Control

This resource does not have any control Configuration Properties.

EDIT...

3.6.8. hq-pdk.jar Command Syntax

The command for running a plugin from the command line is formed like this:

```
java -jar AgentVersion/bundles/AgentBundle/pdk/lib/hq-pdk-VERSION.jar -m Method -a
 MethodAction -p PluginName -t ResourceType -Doption=value
```

The subsections below the command arguments.

-m Method specifies the method to run

- One of:

  - `lifecycle`
  - `discover`
  - `metric`
  - `control`
  - `track`
  - `livedata`
  - `generate`

For details and functionality and usage of each method, see hq-pdk.jar Methods and Functionality.

-p PluginName specifies the plugin to run

Supply the product portion of the plugin name, without the "-plugin.jar" or "-plugin.xml" portion. For example, to run `jboss-plugin.jar`, you specify `-p jboss`.

- Required by these methods:
  - `lifecycle`?
  - `metric`
  - `control`
  - `track`
- Optional for:
  - `discover`
  - `generate`
- Not supported for:
  - `livedata`

Note: If you use a generated properties file to supply resource properties, you do not have to specify the plugin to run on the command line, because a resource properties identifies the plugin. See Generating and Using Resource Properties Files.

-t ResourceType specifies the target resource type

Supply the name of a resource type managed by the plugin you are running enclosed in quotes if the the name includes spaces, for instance, "JBoss 4.2".

- Required by these methods:
  - `metric`
  - `control`

  - `track`
  - `livedata`
- Not supported for:
  - `discover`
  - `generate`

Note: If you use a generated properties file to supply resource properties, you do not have to specify resource type on the command line, because a resource properties specifies the resource type name for the resource. See Generating and Using Resource Properties Files .

-a MethodAction specifies a method argument

Supply an argument supported or required by the the method called. For example, when you run the `track` method, you specify whether you want to track log or configuration events by including either `-a log` or `-a config` in the command line. The arguments for each method are documented in hq-odj.jar Methods and Functionality.

-D Option = Value sets a property value

Include a `-D`
Option
`=`
Value
in the command line for each property you wish to set.

Supply:

- The value of a resource property required by the method called.
  - **Note:** Rather than supply each resource property on the command line, you can reference a generated properties file. For more information see Generating and Using Resource Properties Files.
- The value of an agent or system property that governs agent behavior or plugin execution. For more information, see Properties for Controlling Agent Behavior and Plugin Execution.

3.6.9. hq-pdk.jar Methods and Functionality

lifecyle: Initialize and Check Syntax

The `lifecycle` method loads a plugin and reports errors found.

The syntax for the `lifecycle` method is:

```
$ java -jar bundles/agent-VERSION/pdk/lib/hq-pdk-VERSION.jar -p PluginName -m lifecycle -
Dplugins.include=PluginName
```

where:

- `-p`

PluginName
identifies the plugin to run, by the product portion of the plugin name, e.g. "jboss"; the plugin must reside in the agent's `plugin` directory.

- `-Dplugins.include=`
  PluginName
  ensures that only the specified plugin is loaded; otherwise all plugins are loaded. (Use of this command option is recommended, although not required.)

Example: Results of lifecycle Method on a Plugin with No Errors

This command runs the `lifecycle` method for the jboss plugin; no errors are found.

```
$ java -jar bundles/agent-VERSION/pdk/lib/hq-pdk-VERSION.jar -m lifecycle -p jboss -
Dplugins.include=jboss
```

Example: Results of lifecycle Method on a Plugin with Errors

This command runs the `lifecycle` method for the websphere plugin; errors are reported.

```
$ java -jar bundles/agent-VERSION/pdk/lib/hq-pdk-VERSION.jar -m lifecycle -p websphere -
Dplugins.include=websphere
WARN [main] [MetricsTag] MsSQL 2000 include not found: mssql-cache
WARN [main] [MetricsTag] WebSphere 6.1 include not found: WebSphere 6.0
WARN [main] [MetricsTag] WebSphere 6.1 Application include not found: WebSphere 6.0
 Application
WARN [main] [MetricsTag] WebSphere 6.1 EJB include not found: WebSphere 6.0 EJB
WARN [main] [MetricsTag] WebSphere 6.1 Webapp include not found: WebSphere 6.0 Webapp
WARN [main] [MetricsTag] WebSphere 6.1 Connection Pool include not found: WebSphere 6.0
 Connection Pool
WARN [main] [MetricsTag] WebSphere 6.1 Thread Pool include not found: WebSphere 6.0 Thread
 Pool
WARN [main] [MetricsTag] WebSphere Admin 6.1 include not found: WebSphere Admin 6.0
```

discover: Auto-Discover Resources and Configuration

The `discover` method can be run for one or all plugins. It returns key attributes for each resource discovered, including the values of the resource's configuration options, to the terminal window or to a properties files. If you save discovery results to a file, you can use the file to supply required resource configuration data when you run another method that requires the resource's configuration data.

The syntax for the `discover` option is:

```
{{java -jar bundles/agent-VERSION/pdk/lib/hq-pdk-VERSION.jar -m discover -p PluginName -a
 properties
```

where:

- `-p P`
  luginName
  specifies the plugin to run; if not specified, discovery is performed for all plugins in the agent's `plugin` directory.

- `-a properties` writes discovery results to files; otherwise results are returned only to the terminal window. See Names and Location of Properties Files and Content and Purpose of Properties Files for more information.

To...	Use...	Notes
run discovery for all plugins	`-m discover`	Results are returned to the terminal window.
run discovery for one plugin, in this case Jboss	`-m discover -p jboss`	Results are returned to the terminal window.
run discovery for JBoss and save results to files	`-m discover -p jboss -a properties`	Results are written to files, as well as to the terminal window.
run discovery for all plugins and save results to files	`-m discover -a properties`	Results are written to files, as well as to the terminal window.

metric: Fetch Metrics

The `metric` method can be used to:

- Fetch the metric template and metric value for each metric for a resource managed by the plugin, or just for those metrics that:
  - are collected by default,
  - are of a specified metric category, or
  - are indicator metrics
- Return the metric template only (and not the metric value) for the metrics.
- Fetch metrics repeatedly - as many times as you specify - and return the time it took to perform the fetches for each metric.

Syntax for metric Method Using Resource Properties File

The syntax for running the `metric` method using a properties file to supply resource configuration data is:

```
java -jar bundles/agent-VERSION/pdk/lib/hq-pdk-VERSION.jar -m metric plugin-properties/
ResourceTypeDirectory/ResourceName.properties -a translate -Dmetric-collect=default -
Dmetric-indicator=true -Dmetric-cat=CATEGORY -Dmetric-iter=ITERATIONS
```

where:

- `plugin-properties/`
  ResourceTypeDirectory
  `/`
  ResourceName
  `.properties` is the path to the file generated when the resource was discovered using the `properties` action of the `discover` method. The properties file supplies the values for:
  - `-p`
    PluginName
    specifies the product portion of the plugin name.
  - `-t`
    ResourceType
    is the resource type name.
  - the value of configuration options for the resource

- `-a translate` causes metric templates, but not metric values, to be returned. If `-a translate` is not specified both metric templates and metric values are returned.
- optionally, one (and only one) of these options is specified to limit the metrics returned (if none of these options is specified, all metrics are returned):
  - `-Dmetric-collect=default` limits results to metrics whose "defaultOn" attribute is set to "true".
  - `-Dmetric-indicator=true` limits results to metrics whose "indicator" attribute is set to "true".
  - `-Dmetric-cat=`
    CATEGORY
    limits results to metrics of the specified category - one of AVAILABILITY, UTILIZATION, THROUGHPUT, or PERFORMANCE.
- `-Dmetric-iter=`
  ITERATIONS
  causes a the time (in millis) to collect a metric repeatedly to be reported, rather than the metric value.

Syntax for metric Method Specifying Configuration Data on Command Line

The syntax for running the `metric` method suppling resource configuration data on the command line is:

```
java -jar bundles/agent-VERSION/pdk/lib/hq-pdk-VERSION.jar -m metric -p PluginName -t
 ResourceType -a translate -Dmetric-collect=default -Dmetric-indicator=true -Dmetric-
cat=CATEGORY -Dmetric-iter=ITERATIONS -DOption=Value
```

where:

- `-p`
  PluginName
  specifies the product portion of the plugin name.
- `-t`
  ResourceType
  is the resource type name.
- `-a translate` causes metric templates, but not metric values, to be returned. If `-a translate` is not specified both metric templates and metric values are returned.
- optionally, one (and only one) of these options is specified to limit the metrics returned (if none of these options is specified, all metrics are returned):
  - `-Dmetric-collect=default` limits results to metrics whose "defaultOn" attribute is set to "true".
  - `-Dmetric-indicator=true` limits results to metrics whose "indicator" attribute is set to "true".
  - `-Dmetric-cat=`
    CATEGORY
    limits results to metrics of the specified category - one of AVAILABILITY, UTILIZATION, THROUGHPUT, or PERFORMANCE.
- `-Dmetric-iter=`
  ITERATIONS
  causes a the time (in millis) to collect a metric repeatedly to be reported, rather than the metric value.
- a `-D`

Option
=
Value
specifies the value of a resource configuration option; the command line must include a "`D`
Option
=
Value
" for *each* resource configuration option.

Example Invocations

In these examples, only the method invocation and command options are shown. The `java -jar bundles/agent-`
VERSION/pdk/lib/hq-pdk-VERSION.jar
portion of the command is not shown.

To...	Use...	Notes
fetch metrics for a jboss server	-m metric -p jboss -t "JBoss 4.2" -m metric -Djava.naming.provider.url=jnp://0.0.0.0:2099 -Dserver.log_track.files=../../../logs/server.log -Dprogram=/Applications/HQEE42GA/server-4.2.0-EE/hq-engine/bin/run.sh	In this example, resource configuration data is supplied on the command line.
fetch metrics for the jboss server supplying the configuration data using a properties file	-m metric plugin-properties/jboss-4.2/melba_HQ_jBoss_4.x.properties	In this example, resource configuration data is supplied by a properties file

limit the results to indicator metrics	add the following to the command line:
	`-Dmetric-collect=default`
		If you use this option, do not use either of these:
		`-Dmetric-cat=`
		CATEGORY
		`-Dmetric-indicator=true`

limit the results to metrics of a specific category	add the following to the command line:
	`-Dmetric-cat=`
	CATEGORY
	where:
	CATEGORY
	is one of AVAILABILITY, UTILIZATION, THROUGHPUT, or PERFORMANCE
		If you use this, do not use either of these:
		`-Dmetric-collect=default`
		`-Dmetric-indicator=true`

limit the results to indicator metrics	add the following to the command line:
	`-Dmetric-indicator=true`
		If you use this, do not use either of these:
		`-Dmetric-collect=default`
		`-Dmetric-cat=CATEGORY`

collect each metric multiple times and report how long it took to do so (in millis) instead reporting the metric value	add the following to the command line:
	`-Dmetric-iter=`
	ITERATIONS
	where
		May be used in conjunction with one of:
		`-Dmetric-collect=default`
		`-Dmetric-cat=CATEGORY`

To...	Use...	Notes
	ITERATIONS is the number of times to run getValue for each metric.	-Dmetric-indicator=true
to fetch the metric template (but not metrics) for the jboss server	-m metric plugin-properties/jboss-4.2/melba_HQ_jBoss_4.x.properties -a translate	

Results returned by -metric method's default action

Below is an excerpt from the results of running the default action of the metric method. Both metric templates and metric values are returned.

```
JBoss 4.2 Availability:
jboss.system:service=MainDeployer:StateString:java.naming.provider.url=jnp
%3A//0.0.0.0%3A2099,java.naming.security.principal=%java.naming.security.principal
%,java.naming.security.credentials=
=>100.0%125.015.0365.9 MB
```

Note: Colons In metric templates appear as "%3A" in the results above. |

Results returned by metric method's translate action

This is an excerpt from the results of running the the metric method with the translate action. Metric templates, but not metric values, are returned.

```
JBoss 4.2 Availability:
jboss.system:service=MainDeployer:StateString:java.naming.provider.url=jnp
%3A//0.0.0.0%3A2099,java.naming.security.principal=%java.naming.security.principal
%,java.naming.security.credentials=%java.naming.security.credentials%
JBoss 4.2 Active Thread Count:
jboss.system:type=ServerInfo:ActiveThreadCount:java.naming.provider.url=jnp
%3A//0.0.0.0%3A2099,java.naming.security.principal=%java.naming.security.principal
%,java.naming.security.credentials=%java.naming.security.credentials%
JBoss 4.2 Active Thread Group Count:
jboss.system:type=ServerInfo:ActiveThreadGroupCount:java.naming.provider.url=jnp
%3A//0.0.0.0%3A2099,java.naming.security.principal=%java.naming.security.principal
%,java.naming.security.credentials=%java.naming.security.credentials%
JBoss 4.2 JVM Free Memory:
```

Note: Colons In metric templates appear as "%3A" in the results above.

track: Watch for Log or Configuration Events

The `track` method watches for a log or configuration event.

The syntax for the `track` method is:

```
java -jar /bundles/agent-VERSION/pdk/lib/hq-pdk-VERSION.jar -p PluginName -t "ResourceType"
 -m track -a TrackAction -Dserver.config_track.files=TrackFiles
```

where:

- PluginName

identifies the plugin to run.

- ResourceType
  specifies a particular resource type managed by the plugin.

- TrackAction
  specifies whether to track log events or configuration events:

  - `log`
  - `track`

Note: You could use a generated properties file instead of explicitly specifying:

- `-p`
  PluginName
- `-t "`
  ResourceType
- `-Dserver.config_track.files=TrackFiles`

For an example of using a properties file see Syntax for metric Method Using Resource Properties File.

Examples

In these examples only the method invocation and command options are shown. The `java -jar bundles/agent-` VERSION/pdk/lib/hq-pdk-VERSION.jar portion of the command is not shown.

To...	**Use...**	**Notes**
track changes made to the `/etc/httpd/httpd.conf` file for an "Apache 2.0" server	-p apache -t "Apache 2.0" -m track -a config -Dserver.config_track.files=/etc/httpd/httpd.conf	
track log entries to `/var/log/httpd/error_log` for an "Apache 2.0" server	-p apache -t "Apache 2.0" -m track -a log -Dserver.log_track.files=/var/log/httpd/error_log	
To exercise other configurable tracking behaviors, for instance to check the results of tracking log messages that do or do not contain a particular string, you must set the value of **server.log_track.exclude** or **server.log_track.include** on the command line.	add the desired property definition to thecommand line, for example: -Dserver.log_track.exclude=String	

control: Perform a Control Action on a Resource

The `control` method can be used to perform a supported control action on a resource instance.

The syntax for the `control` method is:

```
java -jar /bundles/agent-VERSION/pdk/lib/hq-pdk-VERSION.jar -m control -a ControlAction -p
 PluginName -t ResourceType -Doption=value
```

where:

- PluginName
  identifies the plugin to run.
- ResourceType
  specifies a particular resource type managed by the plugin.
- ControlAction
  specifies the action to perform.
- a `-D`
  option=value
  for each resource configuration option sets the option's value.

Note: You could use a generated properties file instead of explicitly specifying resource configuration options. For an example of using a properties file see Syntax for metric Method Using Resource Properties File.

Examples

In these examples, only the method invocation and command options are shown. . The `java -jar bundles/agent-` VERSION/pdk/lib/hq-pdk-VERSION.jar portion of the command is not shown.

To...	Use...	Notes
run the "removeAllMessages" control action on a JMS destination	-m control -p jboss -a removeAllMessages -t "JBoss 4.0 JMS Destination" -Djms.destination=DLQ -Djava.naming.naming.url=jnp://localhost:1099	In this example resource configuration data is supplied on the command line.
run the "start" control action on a JBoss server	-m control -a start plugin-properties/jboss-4.0/hammer_JBoss_4.0_all.properties	In this example resource configuration data is supplied from a generated properties file.
to exercise other configurable control behaviors, for instance to supply optional control action arguments, you must set value of the appropriate property, for instance **start.args** or **stop.args** on the command line.	add desired property to the command line, for example: - Dstart.args=SupportedStartArgument	

livedata: Fetch Live System Information

The `livedata` method obtains live system metrics and platform data. The data is returned live, rather than obtained from the HQ database, and is provided in XML format.

The syntax for the `livedata` method is:

```
java -jar bundles/agent-VERSION/pdk/lib/hq-pdk-VERSION.jar -m livedata -t ResourceType -a
 LiveDataAction
```

|

where:

- ResourceType
  is an operating system platform type name, one of:
  - `Linux`
  - `AIX`
  - `MacOSX`
  - `HPUX`
  - `NetBSD`
  - `Win32`
  - `OpenBSD`
  - `Solaris`
- LiveDataAction
  is a one of these SIGAR commands:
  - `cpuinfo` - CPU information for each CPU discovered.
  - `cpuperc` - CPU percentage usage.
  - `top` - Display system resource utilization summaries and process information. SIGAR's top command.requires a PTQL query to select which processes to display.
  - `netstat` return network connections information.
  - `ifconfig` - Display network interface configuration and metrics.
  - `who` - return information about all users currently on the local system.
  - `df` - Report filesytem disk space usage.

Results of cpu Command

This is an example of the results returned by the `cpu` command.

```
$java -jar bundles/agent-VERSION/pdk/lib/hq-pdk-VERSION.jar -m livedata -t MacOSX -a cpu


26101070
12908410
829010
236665670
0
0
0
0
```

```
276504160


31011100
8567260
929150
235996420
0
0
0
0
276503930
```

cpuperc Command Results

This is an example of the results returned by SIGAR's `cpuperc` command.

```
$java -jar bundles/agent-VERSION/pdk/lib/hq-pdk-VERSION.jar -m livedata -t MacOSX -a cpu


26101070
12908410
829010
236665670
0
0
0
0
276504160


31011100
8567260
929150
235996420
0
0
0
0
276503930
```

ifconfig Command Results

This is an excerpt of example results returned by SIGAR's `ifconfig` command.

```
$java -jar bundles/agent-VERSION/pdk/lib/hq-pdk-VERSION.jar -Dplugins.include=jboss -m
 livedata -t MacOSX -a ifconfig


lo0
00:00:00:00:00:00
Local Loopback
lo0
127.0.0.1
127.0.0.1
0.0.0.0
255.0.0.0
32841
16384
0


1718492625
6594769
```

```
0
0
-1
-1
1718494642
6594766
0
-1
-1
0
-1
0
```

Note: The output contains a `<org.hyperic.hq.plugin.system.NetInterfaceData>` element for each network interface discovered.

who Command Results

This is an excerpt of the results returned by SIGAR's `who` command.

```
$ java -jar bundles/agent-VERSION/pdk/lib/hq-pdk-VERSION.jar -Dplugins.include=jboss -m
 livedata -t MacOSX -a who


mmcgarry
console

1264178819


mmcgarry
ttyp1

1264185170


mmcgarry
ttyp2

1264181578
```

generate: Create Plugin Documentation

The `generate` method generates documentation from the plugin descriptor.

The syntax for the `generate` method is:

```
java -jar bundles/agent-VERSION/pdk/lib/hq-pdk-VERSION.jar -p PluginName -m generate -a
 GenerateAction
```

where:

- PluginName
  - identifies a plugin to document. If not specified, the action will be applied to all plugins.
- GenerateAction
  specifies the type of docuementation to generate, one of:
  - `metrics-wiki` - Write a Confluence Wiki formatted summary of supported metrics to a file.
  - `metrics-xml` - Output an XML-formatted summary of supported metrics for a resource type to stdout.

- `metrics-txt` Output a text formatted summary of supported metrics to stdout.
- `help` - Output the contents of the `<help>` element for each resource type in plugin descriptor(s) to HTML files in the `./plugin-help` directory.

Syntax

In these examples, only the method invocation and command options are shown. The `java -jar bundles/agent-`
VERSION/pdk/lib/hq-pdk-VERSION.jar
portion of the command is not shown.

To...	Use...	Notes
document metrics in Confluence wiki format for all resource types in all plugins	`-m generate -a metrics wiki`	
document metrics in text format for all resource types in all plugins	`-m generate -a metrics-txt`	
document metrics in XML format for all resource types in all plugins	`-m generate -a metrics-xml`	
generate a help page for all resource types in all plugins	`-m generate -a help`	
limit results to the resource types managed by a single plugin, in this example, jboss	add `-p jboss` to the command line.	

3.6.10. Properties for Controlling Agent Behavior and Plugin Execution

You can use
-DOption=Value
to set any agent or system property. The table below lists some properties that are useful when you run a plugin from the command line.useful Properties for Command Line

Property	Description
`log`	Set the log level. log=debug
`output.dir`	Override default output directory default
`plugins.include`	This agent property tells the HQ Agent to load a specific plugin, and only that agent before executing the method. Otherwise, when you run `hq-pdk-shared-VERSION.jar` the HQ Agent will load all of the plugins in the plugin directory.
`plugins.exclude`	This agent property give the agent a list of plugins to not load before executing the method. The HQ Agent will load all other plugins in the plugin directory.
`exec.sleep`	You can use this system property to override its default value when you are testing a script plugin. By default, `exec.sleep` is 30 seconds. If your script

Property	Description
	might take longer than that to run, it is useful to increase the value while you check the plugin out.

3.6.11. Running Protocol Checks from Command Line

Running a plugin from the command line is useful for testing and documenting plugins. Command line execution is also an expeditious way to fetch metrics on-demand.

For example, you can run HQ's netservices plugin from the command line to check the availability of a variety of network service types. The netservices plugin can monitor remote resources of the following types.

- HTTP
- POP3
- IMAP
- SMTP
- FTP
- LDAP
- DNS
- SSH
- NTP
- DHCP
- SNMP
- RPC
- InetAddress Ping
- TCP Socket

If you monitor a resource of one of the types shown above on an on-going basis, you configure it as a platform service on a platform of your choice, whose HQ Agent will perform the remote availability checks and metric collection. To enable monitoring, you supply resource configuration data - the hostname of the service, at a minimum. If you wish to run the plugin at the command line, you must supply the required configuration data on the command line.

For an overview of HQ's network service monitoring features and documentation on the configuration options for each network service type, see Monitoring Network Services.

The command below runs the netservices plugin's `metric` method for a remote LDAP server.

```
java -jar bundles/agent-VERSION/pdk/lib/hq-pdk-shared-VERSION.jar -m metric -p netservices
-t LDAP -Dplugins.include=netservices -Dhostname=192.168.1.1 -Dssl=false -Dport=389
-DbaseDN=dc=foobar,dc=co,dc=nz -DbindDN=cn=root,c=foobar,dc=co,dc=nz
-DbindPW=changeme -Dfilter=uidNumber
```

Note that the value of each configuration option for the LDAP service type is supply with a -D argument. The configuration options are defined on the LDAP Platform Service page.

4. Plugin Reference Library

4.1. Auto-Discovery Topics

4.1.1. Autoinventory Overview

Discovering new resources automatically is a very powerfull functionality and helps users and administrations to limit what has to be done manually. Discovery operations are always done by the agent using resource structures defined by the plugins. On general level we are able to discover resources defined by the plugin descriptor.

Discovery Operations

When studying discovery operations we can think about two separate questions - when discovery happens and what happens during the discovery?

Autodiscovery is handled within two different threads or two different scan operations to being more presice. Result from both scans is an autoinventory report which is sent back to the HQ server. First discovery operation is called a default scan and usually happens right after agent has been started. Second discovery operation is called a runtime scan and usually happens when existing services are discovered from servers. While these scan operations are sharing overlapping functionalities which means autoinventory framework doesn't limit service detections only to happen during runtime scan usual scenario is that platforms and server resources are detected during the default scan and underlying services are detected afterwards when platform and server has been approved to the inventory model.

Discovery is initiated either automatically or manually. First automatic default scan operation happens right after agent startup and afterwards once per defined time frame. Runtime scan also happens automatically on predefined intervals. Both of the intervals on default and runtime scans can be configured or turned of by modifying agent properties.

Discover Resources

To understand the ball game let's see who are the stakeholders playing on the field.

Default Scan

Default scan happens right after agent startup and periodically after that or when user manually requests discovery on platform level through UI.

Download Error
(could not download image resource)

Runtime Scan

Runtime scan happens periodically when agent is running or based on request by the HQ server. Direct request for runtime scan from server happens when default scan discovers a server, it contains services and user approves server into the inventory. At this moment server sends a request to agent to discover runtime resources. This same discovery is also ran periodically to ensure that new services are discovered automatically.

Download Error
(could not download image resource)

Manually Invoked Scan

It's also possible to manually initiate the whole discovery process to happen and is always done through UI on platform level. This manual request adds a small twist to this discovery operation by adding possibility to detect resources by scanning filesystems.

Download Error
(could not download image resource)

Discover Resource Types

Usually all different resource types has to be hard coded using xml plugin descriptor. HQ4.2 added new functionality to dynamically discover new service resource types. See separate document Section 4.1.6, “Dynamic Service Type Detection”.

General Consepts

Installation Path

If resource type is a server, installation path is one of the mandatory fields what has to be entered. Sometimes there are no real meaning in terms of mapping spesific server to physical path on a hard drive, but even in that case something has to be entered. You've propably seen this happening when manually defining server using the HQ UI and installation path is asked and it you've noticed that it can't be plank.

Some default helper methods derives the server AIID directly from the server's installation path.

Autoinventory Identifier

AIID is pretty unknown but important concept to successfully discover individual servers from the same platform. This identifier is telling HQ server whether the particular server instance has been autodiscovered or not. HQ agent is using it to disregard dublicate server instances during the scan operation. Even if you define different server types within single plugin descriptor you're unable to discover both server types if AIID's are not unique.

4.1.2. SNMP Detector Plugin

*Topics marked with*relate to features available only in vFabric Hyperic.*

Warning

This document is still a work in progress.

Generic SNMP Detector intended for pure-xml plugins that extend the Network Device platform or servers with builtin SNMP management such as Squid.

Property	Use
SNMP_INDEX_NAME	foo
SNMP_DESCRIPTION	foo

```
<plugin>
<plugin type="autoinventory"
            class="org.hyperic.hq.product.SNMPDetector"/>
...
</plugin>
```

Return to the Plugin Development Center .

4.1.3. Mx Server Detector Plugin

*Topics marked with*relate to features available only in vFabric Hyperic.*

Autodiscover JMX servers

An example Plugin using Mx Server Detector Plugin is the jonas plugin

```
...
<plugin type="autoinventory"
            class="org.hyperic.hq.product.jmx.MxServerDetector"/>
```

```
...
```

Example 4. jonas-plugin.xml

Properties

Property	**Use**	
DEFAULT_CONFIG_FILE	default config file to track	
PROC_MAIN_CLASS		
PROC_HOME_PROPERTY		
PROC_HOME_ENV		

Examples Plugins:
jonas Plugin XML Descriptor

Next Steps

TBD

Related Information

Section 4.3.3, “Log File Tracking Plugin”
Log Tracking
Configuration Tracking

Return to the Plugin Development Center .

4.1.4. Auto Inventory Plugin

*Topics marked with***relate to features available only in vFabric Hyperic.*

The ServerDetector implements autoinventory (which is what auto-discovery is called in the plugin world) of servers and services.

```
<server name="xinetd">
    <property name="PROC_QUERY"
              value="State.Name.eq=xinetd"/>


<config>
      <option name="process.query"
              description="Process Query"
              default="${PROC_QUERY}"/>
</config>
 ...
<plugin type="autoinventory"
            class="org.hyperic.hq.product.DaemonDetector"/>

 ...
</server>
```

Example 5. xinetd-plugin.xml

Properties

Property	Use	
AUTOINVENTORY_NAME	Format the auto-inventory name as defined by the plugin	
INSTALLPATH_MATCH	Return true if installpath matches given substring	
INSTALLPATH_NOMATCH	Return false if installpath matches given substring	
INVENTORY_ID	The installpath param	
PROC_QUERY	default PTQL query to scan for	
HAS_BUILTIN_SERVICES	Scan for built in services, default false	
VERSION_FILE	Return true if given file exists within installpath	

Next Steps

TBD

Related Information

Section 4.1.11, “Daemon Detector Plugin”
PTQL

Return to the Plugin Development Center .

4.1.5. AutoServerDetector

*Topics marked with*relate to features available only in vFabric Hyperic.*

This page contains information that has not been reviewed. Click **Add Comment** at the bottom of the page to enter suggestions or corrections.

Overview

The AutoServerDetector interface (`org.hyperic.hq.product.AutoServerDetector`) is used to discover server resources during the basic system scan. Scan occurs after agent startup and periodically after that or during the manual autoinventory request on platform level. Interface is called with every existing server type definition found from the plugin xml descriptor.

Interface Hierarchy

- org.hyperic.hq.product.AutoServerDetector

Interface References

```
package org.hyperic.hq.product;
import java.util.List;
import org.hyperic.util.config.ConfigResponse;
```

Implementing Methods

This interface implements following methods:

getServerResources(ConfigResponse):List

```
public List getServerResources(ConfigResponse platformConfig)
throws PluginException;
```

This method is called if assosiated autodiscovery implementation is implementing this interface. If plugin defines more that one server type this interface is called against every server type respectively. Method has to return a List of ServerResource object. See Section 4.6, “ServerResource” for more information.

Parameters:

- platformConfig* Configuration for underlying platform.

Returns:
List of `ServerResource` objects.

Throws:
org.hyperic.hq.product.PluginException

Example Usage

```
package hq.example;

import java.io.File;
import java.util.ArrayList;
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.List;

import org.hyperic.hq.product.AutoServerDetector;
import org.hyperic.hq.product.PluginException;
import org.hyperic.hq.product.ServerDetector;
import org.hyperic.hq.product.ServerResource;
import org.hyperic.util.config.ConfigResponse;

public class CustomAutoScanDetector
extends ServerDetector
implements AutoServerDetector {

    /** PTQL query to find matching processes */
    private static final String PTQL_QUERY = "State.Name.ct=firefox";

    public List getServerResources(ConfigResponse config) throws PluginException {
        List servers = new ArrayList();
        List paths = getServerProcessList();

        // if no processes found, return empty list.
        // this means we couldn't discover anything.
        if(paths.size() < 1)
            return servers;

        // using empty installation path,
        // since we don't need it anywhere
        String installPath = "";

        ConfigResponse productConfig = new ConfigResponse();

        // need to save original query to config.
        // this can be later altered through hq gui.
        productConfig.setValue("process.query", PTQL_QUERY);
        ServerResource server = createServerResource(installPath);
        setProductConfig(server, productConfig);
        server.setMeasurementConfig();
        servers.add(server);

        return servers;
    }

    private List getServerProcessList() {
        List servers = new ArrayList();
        long[] pids = getPids(PTQL_QUERY);
        for (int i=0; i<pids.length; i++) {
            String exe = getProcExe(pids[i]);
            if (exe == null) {
                continue;
            }
            File binary = new File(exe);
            if (!binary.isAbsolute()) {
                continue;
            }
            servers.add(binary.getAbsolutePath());
        }
        return servers;
    }


}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
   name="autoscan-example"
   package="hq.example">

      <metrics
            name="basic-process-metrics">
         <metric
               indicator="true"
               units="percentage"
               name="Availability"
               collectionType="dynamic"
               template="sigar:Type=ProcState,Arg=%process.query%:State"
               category="AVAILABILITY">
         </metric>
         <metric
               indicator="true"
               units="B"
               name="Process Virtual Memory Size"
               collectionType="dynamic"
               template="sigar:Type=ProcMem,Arg=%process.query%:Size"
               category="UTILIZATION">
         </metric>
         <metric
               units="B"
               name="Process Resident Memory Size"
               template="sigar:Type=ProcMem,Arg=%process.query%:Resident">
         </metric>
         <metric
               name="Process Page Faults"
               collectionType="trendsup"
               template="sigar:Type=ProcMem,Arg=%process.query%:PageFaults">
         </metric>
         <metric
               units="ms"
               name="Process Cpu System Time"
               collectionType="trendsup"
               template="sigar:Type=ProcCpu,Arg=%process.query%:Sys">
         </metric>
         <metric
               units="ms"
               name="Process Cpu User Time"
               collectionType="trendsup"
               template="sigar:Type=ProcCpu,Arg=%process.query%:User">
         </metric>
         <metric
               units="ms"
               name="Process Cpu Total Time"
               collectionType="trendsup"
               template="sigar:Type=ProcCpu,Arg=%process.query%:Total">
         </metric>
         <metric
               indicator="true"
               units="percentage"
               name="Process Cpu Usage"
               template="sigar:Type=ProcCpu,Arg=%process.query%:Percent">
         </metric>
         <metric
               units="epoch-millis"
               name="Process Start Time"
               collectionType="static"
               template="sigar:Type=ProcTime,Arg=%process.query%:StartTime"
               category="AVAILABILITY">
         </metric>
         <metric
               name="Process Open File Descriptors"
               template="sigar:Type=ProcFd,Arg=%process.query%:Total">
         </metric>
         <metric
               name="Process Threads"
               template="sigar:Type=ProcState,Arg=%process.query%:Threads">
         </metric>
      </metrics>

      <server name="autoscanserver">
         <plugin
               type="autoinventory"
               class="CustomAutoScanDetector">
         </plugin>
         <plugin
               type="measurement"
               class="org.hyperic.hq.product.MeasurementPlugin">
         </plugin>
         <config>
            <option
                  default="State.Name.eq=firefox"
                  name="process.query"
                  description="Process Query for customserver">
            </option>
         </config>
         <metrics
               include="basic-process-metrics">
         </metrics>
      </server>

</plugin>
```

Standalone Invocation

Standalone invocation is done using -m discover and -p <server type name> options.

```
# java -jar hq-pdk.jar
```

```
-Dplugins.include=autoscan-example
-Dlog=info
-m discover
-p autoscanserver
```

4.1.6. Dynamic Service Type Detection

Normally service types are defined using plugin descriptor xml file. Using that approach all of the related information including service name, configuration options, related plugin implementations and metrics structures has to be known at plugin development time. Once plugin is deployed these structures can't be changed. Dynamic service type detection is a method to create new types during runtime.

> Currently dynamic service type creation doesn't allow to add Configuration Options. These are the ones to exist within the <config> tag in the XML plugin descriptor. This means that the used metric templates has to be fully created during the service type discovery.

Autodiscovery base class ServerDetector contains function discoverServiceTypes(ConfigResponse):Set which is called during runtime discovery operation. This method is called prior discoverServices(ConfigResponse):List which usually handles runtime service discovery. This means that from your custom discovery class you can first discover service types and then services. Information fo the service structure is then sent back to server. Structures on server are updated from multiple sources which means that same type service should always contain identical configuration.

Following sections contains information how this method is used. During this documentation we make some references to plugin which basic xml structure would look like this:

```
<plugin name="myplugin">
  ...
  <server name="MyServer">
    ...
</plugin>
```

As you can see there are no services defined under a server type. Since we are about to define these services dynamically, we expect similar result as if these services would be hard coded into the plugin descriptor:

```
<plugin name="myplugin">
  ...
  <server name="MyServer">
    ...
    <!-- -->
    <service name="Service A">
      ...
    </service>
    <service name="Service B">
      ...
    </service>
    <service name="Service C">
      ...
    </service>
</plugin>
```

Implement discoverServiceTypes method

This method has to return a Set containing ServiceType objects. Method skeleton would look something seeing below:

```
@Override
```

```
protected Set<ServiceType> discoverServiceTypes(ConfigResponse serverConfig)
    throws PluginException {
    Set<ServiceType> serviceTypes = new HashSet<ServiceType>();

    // Do your magic...

    return serviceTypes;
}
```

Within this function you should do all the operations to discover and build up your service types structures. Propably you will end up to create a separate factory class to handle these issues. Regardless of what you choose to do, there are definitely some things you need pass on to your factory methods unless you decide to hard code some of the service names. At least you need to know what is your product name which equals the plugin name and server name which is hosting your services. Technically you allready know these but it might be easier to query those during runtime.

Plugin name can be queried from ProductPlugin using function getName():String. To access the object within use function getProductPlugin():ProductPlugin.

Creation of service type structures needs access to hosting server type information which is available from method getTypeInfo():TypeInfo. Object type you actually need is ServerTypeInfo and casting is needed respectively. Your "magic" code segment from above would look something like this:

```
try {
    MyServiceTypeFactory serviceTypeFactory = new MyServiceTypeFactory();

    ProductPlugin pp = getProductPlugin();
    ServerTypeInfo sTypeInfo = (ServerTypeInfo)getTypeInfo();

    serviceTypes = serviceTypeFactory.createTypes(
        pp,
        sTypeInfo
        /*, myDiscoveredServices*/); // object with discovered services

    } catch (Exception e) {
        throw new PluginException(e.getMessage(), e);
    }
```

Using ServiceTypeInfo

Class constructor ServiceTypeInfo(String name, String description, ServerTypeInfo server) takes few arguments. Here's what those means:
name - Fully qualified service name. This is the one you will see under Monitoring Defaults.

description - Service description.

server - Parent server type in form of ServerTypeInfo.

Here is some example how you would construct the class. ServerTypeInfo is passed using variable sTypeInfo.

```
ServiceTypeInfo typeInfo = new ServiceTypeInfo(
    "MyServer Service A",
    "Autodetected Service A which MyServer",
    sTypeInfo);

ServiceTypeInfo typeInfo = new ServiceTypeInfo(
    sTypeInfo.getName() + ' ' + name,
    "Autodetected service type for " + name,
    sTypeInfo);
```

Using ServiceType

Class constructor ServiceType(String serviceName, String productName, ServiceTypeInfo info) takes few arguments. Here's what those means:

serviceName - The unique service type name (unique with respect to server type)

productName The name of the product containing this service

info The ServiceTypeInfo describing this service type

Constructing new ServiceType object you need to use the service name and plugin name as those would exist in XML. Last parameter typeInfo is the one you've just created.

```
ServiceType type = new ServiceType("Service A", "myplugin", typeInfo);

ServiceType type = new ServiceType(name, productName, typeInfo);
```

Handle service settings

After the basic ServiceType is created you are not yet done. At this point it only contain the service name, what is the hosting server type and to which plugin it belongs to. Now you need to add all the missing information related to properties, custom properties, plugins, control actions and metrics. As you remember config options can't be added. These all will be added to ServiceType object.

Even if you don't have anything to set (no properties or control action, etc) it's still adviced to call appropriate methods to initialize empty information. Fail to do this may result errors and exceptions.

Custom properties

Adding custom properties is done using method setCustomProperties(ConfigSchema):void. Always set atleast empty schema. If you want to add some properties, add StringConfigOption to the schema.

```
private void addCustomProperties(final ServiceType serviceType) {

    final ConfigSchema propertiesSchema = new ConfigSchema();
    propertiesSchema.addOption(new StringConfigOption("myopt", "myval"));

    serviceType.setCustomProperties(propertiesSchema);
}
```

> Setting custom properties differs slightly what we've used to see during the normal resource discovery methods. There properties are setted using ConfigResponse class.

Properties

Properties for the ServiceType are setted using method setProperties(ConfigResponse):void. Use method setValue(String, String):void from ConfigResponse to set values.

```
private void addProperties(final ServiceType serviceType) {

    final ConfigResponse properties = new ConfigResponse();
    properties.setValue("myprop", "myval");

    serviceType.setProperties(properties);
}
```

Plugins

If servicetype is to support any kind of measurement functions it need to know the plugin implementation. Same goes with autoinventory and other plugin types. These are setted using same method what we just saw by setting properties.

```
private void addPlugins(final ServiceType serviceType) {

    final ConfigResponse pluginClasses = new ConfigResponse();
    pluginClasses.setValue("autoinventory", "hq.training.MyDetector");
    pluginClasses.setValue("measurement", "hq.training.MyMeasurementPlugin");

    serviceType.setPluginClasses(pluginClasses);
}
```

Control actions

Supported control actions are setted using method setControlActions(Set):void. Set contains a list of actions as type of String.

```
private void addControlActions(final ServiceType serviceType) {

    Set<String> actions = new HashSet<String>();
    actions.add("start"):
    actions.add("stop"):

    serviceType.setControlActions(actions);
}
```

Measurements

Metrics are added using method setMeasurements(MeasurementInfos):void where MeasurementInfos contains the actual metrics. One metric entity is represented by class MeasurementInfo. These are tied together using method addMeasurementInfo(MeasurementInfo):void from class MeasurementInfos.

We can usually create metrics using same type of helper method, except the availability metric which usually have different type of layout. To simplify metric creation we use two methods, one to create measurement properties and second to add these properties to actual metric.

```
private MeasurementInfo createAvailabilityMeasurement(final ServiceType serviceType) {
    Properties measurementProperties = new Properties();

    measurementProperties.put(MeasurementInfo.ATTR_UNITS,
 MeasurementConstants.UNITS_PERCENTAGE);
    measurementProperties.put(MeasurementInfo.ATTR_NAME, Metric.ATTR_AVAIL);
    measurementProperties.put(MeasurementInfo.ATTR_ALIAS, Metric.ATTR_AVAIL);
    measurementProperties.put(MeasurementInfo.ATTR_COLLECTION_TYPE, "dynamic");
    measurementProperties.put(MeasurementInfo.ATTR_CATEGORY,
 MeasurementConstants.CAT_AVAILABILITY);
    measurementProperties.put(MeasurementInfo.ATTR_INDICATOR, "true");
    measurementProperties.put(MeasurementInfo.ATTR_DEFAULTON, "true");
    measurementProperties.put(MeasurementInfo.ATTR_INTERVAL, "600000");

    measurementProperties.put(MeasurementInfo.ATTR_TEMPLATE, "dummy-domain::Availability");

    return createMeasurementInfo(measurementProperties);
}
```

```
private MeasurementInfo createMeasurementInfo(Properties measurementProperties) {
    MeasurementInfo metric = new MeasurementInfo();

    try {
        metric.setAttributes(measurementProperties);
    } catch (Exception e) {
        log.warn("Error setting metric attributes.  Cause: " + e.getMessage());
    }

    // Make sure we're using upper case letters for category
    metric.setCategory(metric.getCategory().toUpperCase());

    return metric;
}
```

Next we can use similar type of template for other than availability metrics. If you need to modify metric parameters, create separate function respectively. To simplify this method example we use metric name which also qualify as metric alias. Usually metric name differs from alias by being more human readable.

```
private MeasurementInfo createMeasurementInfo(ServiceType serviceType,
                                              String metric) {

    Properties measurementProperties = new Properties();

    measurementProperties.put(MeasurementInfo.ATTR_UNITS, "none");
    measurementProperties.put(MeasurementInfo.ATTR_NAME, metric);
    measurementProperties.put(MeasurementInfo.ATTR_ALIAS, metric);
    measurementProperties.put(MeasurementInfo.ATTR_COLLECTION_TYPE,"dynamic");
    measurementProperties.put(MeasurementInfo.ATTR_RATE, "none");
    measurementProperties.put(MeasurementInfo.ATTR_INTERVAL, "300000");
    measurementProperties.put(MeasurementInfo.ATTR_CATEGORY,
 MeasurementConstants.CAT_UTILIZATION);
    measurementProperties.put(MeasurementInfo.ATTR_INDICATOR, "false");
    measurementProperties.put(MeasurementInfo.ATTR_DEFAULTON, "false");

    measurementProperties.put(MeasurementInfo.ATTR_TEMPLATE, "dummy-domain::"+metric);
    return createMeasurementInfo(measurementProperties);
}
```

We just used hard coded metric template which only added metric name as an attribute. If this is all you need to collect your metrics, you're good. However in real world there is propably need to pass some configuration parameters. Instead of just hardcode metric template, we could use helper function to add template to measurement properties.

```
private void addMeasurementTemplate(Properties measurementProperties,
                                    ProductPlugin productPlugin,
                                    ServiceType serviceType) {
    TokenReplacer replacer = new TokenReplacer();

    final String objectName =
 serviceType.getProperties().getValue(serviceType.getInfo().getName() + ".OBJECT");

    addFilter(MeasurementInfo.ATTR_ALIAS, measurementProperties, replacer);
    replacer.addFilter("OBJECT", objectName);
    final String template = filter(productPlugin.getPluginProperty("template"), replacer);

    measurementProperties.put(MeasurementInfo.ATTR_TEMPLATE, template);
}

private String filter(String val, TokenReplacer replacer) {
    return replacer.replaceTokens(val);
}
```

Above example is using same TokenReplacer used to translate variables during metric collection. We could Store the metric template to xml using property tag and write properties to there.

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin name="myplugin">

  <property name="template" value="service-domain:Object=${OBJECT}:${alias}" />
```

These translated properties can be stored to ServiceType using methods we saw in addProperties function. These properties goes to global set and has to be unique, that's wy we are using serviceType.getInfo().getName() + ".OBJECT" and not just OBJECT.

4.1.7. DaemonDetector

Topics marked with*relate to features available only in vFabric Hyperic.

This page contains information that has not been reviewed. Click **Add Comment** at the bottom of the page to enter suggestions or corrections.

Overview

The DaemonDetector class (`org.hyperic.hq.product.DaemonDetector`) autodiscovers a single process and adds related PTQL query to resource configuration.

Class Hierarchy

- java.lang.Object
  - org.hyperic.hq.product.GenericPlugin
    - org.hyperic.hq.product.ServerDetector
      - org.hyperic.hq.product.DaemonDetector

Configurable Options

A resource managed by a plugin that uses the `DaemonDetector` class has the configuration properties shown in the table below. The properties appear on a managed resource's **Configuration Properties** page, in the "Monitoring" section - you need to define them as `<options>` in a plugin descriptor.

Property	Usage
`process.query`	Placeholder for used PTQL process query.

Resource Properties

The table below defines properties you can define in `<property>` elements in the descriptor for a plugin that uses this class.

Property	Description	Usage
`PROC_QUERY`	Sets default value of PTQL query to find daemon process.	Required.

Example Usage

This example defines a new server resource which is autodiscovered based on results from PTQL process query.

```
<server name="My Single Process Server">

   <property name="PROC_QUERY" value="State.Name.eq=myprocess"/>

   <config>
     <option
       default="State.Name.eq=myprocess"
       name="process.query"
       description="Process Query for singleprocess">
     </option>
   </config>

  <plugin type="autoinventory" class="org.hyperic.hq.product.DaemonDetector" />

  ...

</server>
```

4.1.8. RegistryServerDetector

Topics marked with*relate to features available only in vFabric Hyperic.

This page contains information that has not been reviewed. Click **Add Comment** at the bottom of the page to enter suggestions or corrections.

Overview

The RegistryServerDetector interface (`org.hyperic.hq.product.RegistryServerDetector`) is used to discover server resources based on windows registry scan. Background scan is getting hints from <scan> tag to match correct file paths. Based on these results this interface is called with every matched result.

Scan hints are given using the <scan> tag. Registry attribute is the base key where recursive search will happen. <include> can be used to spesify registry keys to search for. Wildcards can't be used within the key name. Every found key results one call against this interface.

```
<scan registry="SOFTWARE\Microsoft\Internet Explorer">
  <include name="AppName"/>
</scan>
```

It's possible to extend search to multiple root keys by ending key name with "**". For example "SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall\MySQL**" only dig into subkeys of "...\Uninstall" that startWith "MySQL".

Interface Hierarchy

- org.hyperic.hq.product.RegistryServerDetector

Interface References

```
package org.hyperic.hq.product;
import java.util.List;
import org.hyperic.sigar.win32.RegistryKey;
import org.hyperic.util.config.ConfigResponse;
```

Implementing Methods

This interface implements following methods:

getServerResources(ConfigResponse):List

```
public List getServerResources(ConfigResponse platformConfig, String path, RegistryKey
 current)
throws PluginException;
```

This method is called if assosiated autodiscovery implementation is implementing this interface.

Parameters:

- platformConfig* Configuration for underlying platform.
- path* Value of the matched key
- current* Current registry object.

Returns:
List of `ServerResource` objects.

Throws:
org.hyperic.hq.product.PluginException

getRegistryScanKeys():List

```
public List getRegistryScanKeys();
```

This method returns list of registry keys to scan. ServerDetector contains default implementation for this function which is requesting keys from plugin descriptor file. Example xml used in this document would result single list member "SOFTWARE\Microsoft\Internet Explorer".

User can implement/overwrite this method and return list of keys directly from this method.

Returns:
List of registry keys.

Example Usage

```
package hq.example;

import java.util.ArrayList;
import java.util.List;

import org.hyperic.hq.product.PluginException;
import org.hyperic.hq.product.RegistryServerDetector;
import org.hyperic.hq.product.ServerDetector;
import org.hyperic.hq.product.ServerResource;
import org.hyperic.sigar.win32.RegistryKey;
import org.hyperic.util.config.ConfigResponse;

public class CustomRegistryScanDetector
extends ServerDetector
implements RegistryServerDetector {

    /** Base PTQL query to find matching processes by full path */
    private static final String PTQL_QUERY = "State.Name.eq=iexplore";

 public List getServerResources(ConfigResponse platformConfig, String path, RegistryKey
 current)
   throws PluginException {
        List servers = new ArrayList();

        ConfigResponse productConfig = new ConfigResponse();

        productConfig.setValue("process.query", PTQL_QUERY);
        ServerResource server = createServerResource(path);
        setProductConfig(server, productConfig);
        server.setMeasurementConfig();
        servers.add(server);

        return servers;

 }

}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
   name="registryscan-example"
   package="hq.training">

      <metrics
            name="multi-process-metrics">
         <metric
               indicator="true"
               units="percentage"
               name="Availability"
               collectionType="dynamic"
               template="sigar:Type=MultiProcCpu,Arg=%process.query%:Availability"
               category="AVAILABILITY">
         </metric>
         <metric
               units="none"
               name="Number of Processes"
               alias="NumProcesses"
               collectionType="dynamic"
               template="sigar:Type=MultiProcCpu,Arg=%process.query%:Processes"
               category="UTILIZATION">
         </metric>
         <metric
               units="B"
               name="Memory Size"
               alias="MemSize"
               collectionType="dynamic"
               template="sigar:Type=MultiProcMem,Arg=%process.query%:Size"
               category="UTILIZATION">
         </metric>
         <metric
               units="B"
               name="Resident Memory Size"
               alias="ResidentMemSize"
               collectionType="dynamic"
               template="sigar:Type=MultiProcMem,Arg=%process.query%:Resident"
               category="UTILIZATION">
         </metric>
         <metric
               units="ms"
               name="Cpu System Time"
               alias="SystemTime"
               collectionType="trendsup"
               template="sigar:Type=MultiProcCpu,Arg=%process.query%:Sys"
               category="UTILIZATION">
         </metric>
         <metric
               units="ms"
               name="Cpu User Time"
               alias="UserTime"
               collectionType="trendsup"
               template="sigar:Type=MultiProcCpu,Arg=%process.query%:User"
               category="UTILIZATION">
         </metric>
         <metric
               units="ms"
               name="Cpu Total Time"
               alias="TotalTime"
               collectionType="trendsup"
               template="sigar:Type=MultiProcCpu,Arg=%process.query%:Total"
               category="UTILIZATION">
         </metric>
         <metric
               indicator="true"
               units="percentage"
               name="Cpu Usage"
               alias="Usage"
               collectionType="dynamic"
               template="sigar:Type=MultiProcCpu,Arg=%process.query%:Percent"
               category="UTILIZATION">
         </metric>
      </metrics>


      <server name="registryscanserver">
         <plugin
               type="autoinventory"
               class="CustomRegistryScanDetector">
         </plugin>
         <plugin
               type="measurement"
               class="org.hyperic.hq.product.MeasurementPlugin">
         </plugin>
         <scan registry="SOFTWARE\Microsoft\Internet Explorer">
            <include name="AppName"/>
         </scan>
         <config>
            <option
                  default="State.Name.eq=iexplore"
                  name="process.query"
                  description="Process Query for customserver">
            </option>
         </config>
         <metrics
               include="multi-process-metrics">
         </metrics>
      </server>

</plugin>
```

Standalone Invocation

Standalone invocation is done using -m discover and -p <server type name> options.

```
# java -jar hq-pdk.jar
-Dplugins.include=registryscan-example
-Dlog=info
-m discover
-p registryscanserver
```

4.1.9. ServerDetector

*Topics marked with*relate to features available only in vFabric Hyperic.*

This page contains information that has not been reviewed. Click **Add Comment** at the bottom of the page to enter suggestions or corrections.

Overview

The ServerDetector class (`org.hyperic.hq.product.ServerDetector`) is a base implementation for autodiscovery. This is a abstract class and thus can't be directly used to handle autodiscovery operations. Actual autodiscovery implementation has to inherit this base class.

Class Hierarchy

- java.lang.Object
  - org.hyperic.hq.product.GenericPlugin
    - org.hyperic.hq.product.ServerDetector

Resource Properties

The table below defines properties you can define in `<property>` elements in the descriptor for a plugin that uses this class.

Property	**Description**	**Usage**
`INSTALLPATH`	Overwrites the installation path.	Not required.
`INSTALLPATH_MATCH`		Not required.
`INSTALLPATH_NOMATCH`		Not required.
`VERSION_FILE`		Not required.
`INVENTORY_ID`	Overwrites the autoinventory id (AIID).	Not required.
`AUTOINVENTORY_NAME`	Formats discovered resource name.	Not required.

Using extra filters

Extra filters `INSTALLPATH_MATCH`, `INSTALLPATH_NOMATCH` and `VERSION_FILE` can be used to filter discovered resources based on discovered intallpath. These filters are used in following order:

1. If `VERSION_FILE` is not found resource is skipped.
2. If `INSTALLPATH_MATCH` is not found from installpath resource is skipped.
3. If `INSTALLPATH_NOMATCH` is found from installpath resource is skipped.

Using INSTALLPATH

Every server type resource has to have setting relates to installation path. You can see that this information is asked if server is created manuall. If server is autodiscovered this setting is resolved automatically and usually is either server home directory or process working directory. This property can be used to overwrite discovered installation path.

Using AUTOINVENTORY_NAME

It's possible to overwrite discovered resource name by defining new qualifier. Format of this name is a single string containing variables (%variable1%) which are translated from configuration properties. Three different types of properties are passed to formatting functions as ConfigResponse objects. These properties includes parent resource, resource itself and custom properties.

> This property is only used if autodiscovery implementation spesifically calls appropriate formatting functions. Check from implementing class documentation if property is activated.

Using INVENTORY_ID

Autoinventory ID is used to identify unique resources within discovered resource types. When new autodiscovery report is sent from agent to server AIID is used to check if resource already exist in inventory model.

Implementing Methods

This class implements following methods:

setDescription(String):void

```
protected void setDescription(String description)
```

Sets the server description. This method allows to set description outside of ServerResource object. Techically this allows you to update server description while discovering new services but few rules apply. If discoverServers() discovers something or discoverServices() doesn't discover anything this field will be ignored.

Parameters:

- description* Server description

setCustomProperties(ConfigResponse):void

```
protected void setCustomProperties(ConfigResponse cprops)
```

Sets the server custom properties. This method allows to set custom properties outside of ServerResource object. Techically this allows you to update server custom properties while discovering new services but few rules apply. If discoverServers() discovers something or discoverServices() doesn't discover anything this field will be ignored.

Parameters:

- cprops* Server custom properties

discoverServers(ConfigResponse):List

```
protected List discoverServers(ConfigResponse config)
```

Runtime method to discover new servers. Override to discover servers for the server type of the plugin instance. In most cases plugins will override discoverServices() rather than this method. The general use of this method is where a plugin interface FileServerDetector or AutoServerDetector finds the Admin server and this method discovers the managed server nodes. Such plugins include WebLogic, WebSphere and iPlanet.

This method returns NULL if not overwritten.

Parameters:

- config* Parent configuration.

Returns:
List of type ServerResource.

discoverServices(ConfigResponse):List

```
protected List discoverServices(ConfigResponse config)
```

Runtime method to discover new services. Override to discover services for the server type of the plugin instance.

This method returns NULL if not overwritten.

Parameters:

- config* Parent configuration.

Returns:
List of type ServiceResource.

discoverServiceTypes(ConfigResponse):Set

```
protected Set discoverServiceTypes(ConfigResponse config)
```

Runtime method to discover new service types.

This method returns empty HashSet if not overwritten.

Parameters:

- config* Parent configuration.

Returns:
Set of ServiceType objects.

createServerResource(String):ServerResource

```
protected ServerResource createServerResource(String installpath)
```

This is a helper method to initialize an ServerResource with default values.

Parameters:

- installpath* Resource installation path.

4.1.10. FileServerDetector

Topics marked with*relate to features available only in vFabric Hyperic.

This page contains information that has not been reviewed. Click **Add Comment** at the bottom of the page to enter suggestions or corrections.

Overview

The FileServerDetector interface (`org.hyperic.hq.product.FileServerDetector`) is used to discover server resources based on file system scan. This interface is used if user manually invokes new autodiscovery on platform level. Background scan is getting hints from <scan> tag to match correct file paths. Based on these results this interface is called with every matched result.

Interface Hierarchy

- org.hyperic.hq.product.FileServerDetector

Interface References

```
package org.hyperic.hq.product;
import java.util.List;
import org.hyperic.util.config.ConfigResponse;
```

Implementing Methods

This interface implements following methods:

getServerResources(ConfigResponse,String):List

```
public List getServerResources(ConfigResponse platformConfig, String path)
throws PluginException;
```

This method is called if assosiated autodiscovery implementation is implementing this interface. Method is called with every successfully matched result. Return value has to be a List of ServerResource object. See Section 4.6, “ServerResource” for more information.

Parameters:

- platformConfig* Configuration for underlying platform.
- path* Matched path

Returns:
List of `ServerResource` objects.

Throws:
org.hyperic.hq.product.PluginException

Example Usage

```
package hq.example;

import java.util.ArrayList;
import java.util.List;

import org.hyperic.hq.product.FileServerDetector;
import org.hyperic.hq.product.PluginException;
import org.hyperic.hq.product.ServerDetector;
import org.hyperic.hq.product.ServerResource;
import org.hyperic.util.config.ConfigResponse;
```

```
public class CustomFileScanDetector
extends ServerDetector
implements FileServerDetector {

    /** Base PTQL query to find matching processes by full path */
    private static final String PTQL_QUERY = "Exe.Name.ct=";

    public List getServerResources(ConfigResponse platformConfig, String path)
    throws PluginException {
        List servers = new ArrayList();
        ConfigResponse productConfig = new ConfigResponse();

        // alter query to find discovered process
        // this can be later altered through hq gui.
        productConfig.setValue("process.query", PTQL_QUERY + path);
        ServerResource server = createServerResource(path);
        setProductConfig(server, productConfig);
        server.setMeasurementConfig();
        servers.add(server);

        return servers;
 }

}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
   name="filescan-example"
   package="hq.example">

      <metrics
            name="basic-process-metrics">
         <metric
               indicator="true"
               units="percentage"
               name="Availability"
               collectionType="dynamic"
               template="sigar:Type=ProcState,Arg=%process.query%:State"
               category="AVAILABILITY">
         </metric>
         <metric
               indicator="true"
               units="B"
               name="Process Virtual Memory Size"
               collectionType="dynamic"
               template="sigar:Type=ProcMem,Arg=%process.query%:Size"
               category="UTILIZATION">
         </metric>
         <metric
               units="B"
               name="Process Resident Memory Size"
               template="sigar:Type=ProcMem,Arg=%process.query%:Resident">
         </metric>
         <metric
               name="Process Page Faults"
               collectionType="trendsup"
               template="sigar:Type=ProcMem,Arg=%process.query%:PageFaults">
         </metric>
         <metric
               units="ms"
               name="Process Cpu System Time"
               collectionType="trendsup"
               template="sigar:Type=ProcCpu,Arg=%process.query%:Sys">
         </metric>
         <metric
```

```
               units="ms"
               name="Process Cpu User Time"
               collectionType="trendsup"
               template="sigar:Type=ProcCpu,Arg=%process.query%:User">
         </metric>
         <metric
               units="ms"
               name="Process Cpu Total Time"
               collectionType="trendsup"
               template="sigar:Type=ProcCpu,Arg=%process.query%:Total">
         </metric>
         <metric
               indicator="true"
               units="percentage"
               name="Process Cpu Usage"
               template="sigar:Type=ProcCpu,Arg=%process.query%:Percent">
         </metric>
         <metric
               units="epoch-millis"
               name="Process Start Time"
               collectionType="static"
               template="sigar:Type=ProcTime,Arg=%process.query%:StartTime"
               category="AVAILABILITY">
         </metric>
         <metric
               name="Process Open File Descriptors"
               template="sigar:Type=ProcFd,Arg=%process.query%:Total">
         </metric>
         <metric
               name="Process Threads"
               template="sigar:Type=ProcState,Arg=%process.query%:Threads">
         </metric>
      </metrics>

      <server name="filescanserver">
         <plugin
               type="autoinventory"
               class="CustomFileScanDetector">
         </plugin>
         <plugin
               type="measurement"
               class="org.hyperic.hq.product.MeasurementPlugin">
         </plugin>
         <scan>
            <include name="/**/firefox.exe"/>
         </scan>
         <config>
            <option
                  default="Exe.Name.eq=svc"
                  name="process.query"
                  description="Process Query for customserver">
            </option>
         </config>
         <metrics
               include="basic-process-metrics">
         </metrics>
      </server>

</plugin>
```

Standalone Invocation

Standalone plugin invocation differs slightly how FileServerDetector and AutoServerDetector are executed compared to real agent. If real agent is about to use FileServerDetector it executes that before AutoServerDetector. Standalone invocation executes either one of these but not both of them.

To test FileServerDetector interface make sure that at least one of the following parameters exist:

Property key	Description	Values	Defaults
fileScan.scanDirs	Directories to scan	List of directories separated by comma	Win:"C:\" Unix:"/usr", "/opt"
fileScan.excludeDirs	Directories to exclude from scan	List of directories separated by comma	Win:"\WINNT", "\TEMP", "\TMP","\Documents and Settings", "\Recycled" Unix:"/usr/doc", "/usr/dict", "/usr/lib", "/usr/libexec", "/usr/man", "/usr/tmp", "/usr/include", "/usr/share", "/usr/src","/usr/local/include", "/usr/local/share", "/usr/local/src"
fileScan.fsTypes	File system types to scan	One of "All disks", "Local disks", "Network-mounted disks"	All disks
fileScan.depth	How deep (in directory levels) to scan.	1 or above, use -1 to indicate unlimited depth.	6
fileScan.followSymlinks	Should symlinks be followed?	true or false	false

Standalone invocation is done using -m discover and -p <server type name> options.

```
# java -jar hq-pdk.jar
-Dplugins.include=filescan-example
-Dlog=info
-DfileScan.scanDirs="C:\\Program Files (x86)"
-DfileScan.excludeDirs="\\WINNT,\\TEMP,\\TMP,\\Documents and Settings,\\Recycled"
-DfileScan.fsTypes="Local disks"
-DfileScan.depth=2
-DfileScan.followSymlinks=false
-m discover
-p filescanserver
```

4.1.11. Daemon Detector Plugin

Topics marked with*relate to features available only in vFabric Hyperic.

The Daemon Detector Plugin implements a way to discover running processes automatically

An example Plugin using Daemon Detector Plugin is xinetd Plugin:

```
...
<server name="xinetd">
    <property name="PROC_QUERY"
              value="State.Name.eq=xinetd"/>


<config>
      <option name="process.query"
              description="Process Query"
              default="${PROC_QUERY}"/>
</config>
 ...
<plugin type="autoinventory"
            class="org.hyperic.hq.product.DaemonDetector"/>
...
```

Example 6. xinetd-plugin.xml

Properties

Property	Use	
PROC_QUERY	PTQL query to identify server	
HAS_BUILTIN_SERVICES	additional services	
AUTOINVENTORY_NAME	Format the auto-inventory name as defined by the plugin	
INSTALLPATH_MATCH	Return true if installpath matches given substring	
INSTALLPATH_NOMATCH	Return false if installpath matches given substring	
INVENTORY_ID	The installpath param	
PROC_QUERY	default PTQL query to scan for	
HAS_BUILTIN_SERVICES	Scan for built in services, default false	
VERSION_FILE	Return true if given file exists within installpath	

Examples Plugins:
xinetd Plugin XML Descriptor
sendmail Plugin XML Descriptor

Next Steps

TBD

Related Information

Section 4.1.4, “Auto Inventory Plugin”

Return to the Plugin Development Center .

4.2. Measurement

- Section 4.2.1, “Measurement Plugin”
  - Availability
  - Variables

- Section 4.2.2, “MeasurementPlugin”
- Section 4.2.3, “Measurement Support Class”
- Section 4.2.4, “SNMP Measurement Plugin”
- Section 4.2.5, “Win32 Measurement Plugin”

4.2.1. Measurement Plugin

*Topics marked with***relate to features available only in vFabric Hyperic.*

Measurement plugins implement HQ's monitoring function. They are one of the most common plugins, both for standard plugins and custom. They allow HQ to for identify resources and collect metrics for them. The main (and sometimes sole) task when writing a custom plugin is to write an XML descriptor file.

- Writing the XML descriptor [213]
- Using the MeasurementPlugin helper class [213]
- Defining measurements with the `<metric>` tag [214]
- Using Templates to collect metric data [217]
- Getting your plugin to auto-discover resources [219]

Writing the XML Descriptor

Each plugin requires an XML descriptor that specifies the type of plugin and, in this case, the resources to look for and the metrics to collect from them. The rest of this page describes the major elements that you need to include in that file.

Consult the Plugin XML Descriptor page to learn about the structure and syntax of this file and generally how to construct one for your plugin.

back to top

Using the MeasurementPlugin Support Class

When writing a measurement plugin, you always identify it as such. The exact syntax depends on the plugin implementation:

Implementation	**How to Identify It as a Measurement Plugin**

script
built-in protocol monitors
Windows
	`<plugin type="measurement"`
	` class="org.hyperic.hq.product.MeasurementPlugin"/`
	`>`

JMX	`<plugin type="measurement"`
	` class="org.hyperic.hq.product.jmx.MxMeasurementPlugin"/`
	`>`

SNMP	`<plugin type="measurement"`
	` class="org.hyperic.hq.product.SNMPMeasurementPlugin"/`
	`>`

This invokes the HQ "support class" MeasurementPlugin, which defines metrics and implements a single method: `getValue()`, which it uses to collect configured metrics. The method of collecting metrics is left entirely to the plugin.

Support classes are provided to assist with certain collection methods such as SNMP, URL, JDBC, Windows Perflib Data and SIGAR. Following are some examples of collection methods used by various plugins:

- **JMX** - JBoss, WebLogic, WebSphere, Tomcat, Resin
- **SNMP** - Apache, iPlanet/SunONE, Apple Airport
- **JDBC** - Mysql, PostgreSQL, Oracle, Sybase
- **Windows Perflib** - .NET, Exchange, Terminal Services, IIS
- **SIGAR** - System, Server process metrics (NTP, Mysql, VMware, etc)
- **Vendor API** - DB2, WebSphere 4.x, SilverStream, VMware

Learn more about support classes.

back to top

Defining Measurements with the <metric> Tag

The sine qua non of Measurement plugins is the collection of metrics. In the XML descriptor, you define the collection of a metric (a "measurement," if you will) using the `<metric>` tag.

> **You Must Always Collect the Availability Metric**
>
> The Availability metric indicates whether a Resource is up or down.
>
> A metrics-gathering plugin must determine Availability for *every* server and *every* service it monitors. A single plugin will likely gather Availability for multiple Resources. If Availability is not gathered for a Resource, HQ will consider the Resource to be unavailable, and will not show any metrics for it in the Portal.
>
> A plugin sets the value of Availability to 1 if the Resource is up, and 0 if it is down. These values are displayed in the Portal as "available" or "not available".
>
> Verifying the existence of a Resource's process is a common technique for determining its Availability. However, the method a plugin uses to determine Availability can vary depending on the Resource Type and the plugin developer's judgment. There might be alternative techniques for determining the Availability of a Resource. For instance, a plugin might determine the Availability of a web server based on whether its process is up, its port is listening, it is responsive to a request, or by some combination of these conditions.

The following explains each each metric attribute, most of which, in fact, are intended for use by th eServer to control display of the metric data.

Metric Attribute	Description	Req'd	Possible Values
`name`	Name of the metric to be displayed in the GUI	Y	
`alias`	Abbreviated name of the metric, displayed in the plugin's output (name-val-	N	In the case of a JMX measurement plugin, the alias must match exactly the

Metric Attribute	Description	Req'd	Possible Values
	ue pairs). If not specified, the alias defaults to the metric-parameter `name`, stripped of any non-letter and non-digit characters and of whitespace.		name of the mbean attribute.
`category`	The category of metric	N	AVAILABILITY, THROUGHPUT, PERFORMANCE, UTILIZATION If the name attribute is Availability, defaults to AVAILABILITY, otherwise defaults to UTILIZATION.
`units`	The units of measurement of the metric, which determines its display in the UI	N	• none: Will not be formatted. • percentage • B: Bytes • KB: Kilobytes • MB: Megabytes • GB: Gigabytes • TB: Terabytes • epoch-millis: Time since January 1, 1970 in milliseconds. • epoch-seconds: Time since January 1, 1970 in seconds. • ns: Nanoseconds • mu: Microseconds • ms: Milliseconds • jiffys: Jiffies (1/100 sec) • sec: Seconds • cents: Cents (1/100 of 1 US Dollar)

If the name attribute is Availability, defaults to percentage, otherwise defaults to none. |

`indicator`	Whether or not this metric should be treated as an indicator metric in HQ	N	true, false
`collectionType`	A description of how the metric's data will behave, for purposes of display in HQ. For example, the metric "Requests Served" will trend up as more and more requests are counted over time.	N	• dynamic: Value may go up or down. • static: Value will not change or not graph. For example, a date stamp. • trendsup: Values will always increase. Because of that, the rate of change becomes more important, so HQ automatically creates a secondary metric: a per-minute rate measurement. If this rate metric has a defaultOn attribute set to true, the defaultOn attribute for the original metric is set to false (therefore only the rate metric will be displayed, not the original metric). To disable the automatically generated rate metric, set its rate attribute to none. • trendsdown: Value changes will always decrease. Defaults to dynamic.
`template`	Unable to render {include} Couldn't find a page to include called: TemplateLearn more.	N	See examples
`defaultOn`	If true, this measurement will be scheduled by default.	N	If `indicator` is true defaults to true. Otherwise defaults to false.
`interval`	Default collection interval (in milliseconds)	N	If the name attribute is Availability, defaults are: • Platforms, 1 minute • Servers, 5 minute • Services, 10 minutes Otherwise, defaults are: • collectionType dynamic, 5 minutes • collectionType trendsup,trendsdown,10 minutes

Table 1. Example of a simple metric tag:

```
<metric name="Availability"
            category="AVAILABILITY"
            units="percentage"
            indicator="true"
            collectionType="dynamic"/>
```

Example of a more complicated one:

```
<metric name="Availability"
            alias="Availability"
            template="sigar:Type=ProcState,Arg=%process.query%:State"
            category="AVAILABILITY"
            indicator="true"
            units="percentage"
            collectionType="dynamic"/>
```

back to top

Using Templates to Collect Metric Data

Unable to render {include} Couldn't find a page to include called: Template

Templates allow plugins to "mix and match" sources for the data they collect.

The measurement template uses an extended form of a JMX ObjectName:

```
domain:properties:attribute:connection-properties
```

Example 7. Template...template

```
jboss.system:Type=ServerInfo:FreeMemory:naming.url=%naming.url%
```

Example 8. Example template

where:

- `domain` = jboss.system
- `properties` = Type=ServerInfo
- `attribute` = FreeMemory
- `connection-properties` = naming.url=%naming.url%

This is the extension to the JMX ObjectName format. Arbitrary properties generally used to connect to the managed server. In this example, JBoss JMX requires a JNP URL (specified here as a variable, indicated by "%": `%naming.url%`). The variable is given a value by the MeasurementPlugin.translate method, using the inventory property value for this server instance.

Using Support Classes to Simplify Metric Collection

In a template, the `domain` can be used to invoke an HQ-provided support class for handling common sources of metrics, such as Process Information, scripts, SQL Queries, and Network Services. (Specific domain names are "registered" with these support classes.) You can see this use of templates in many of the plugin examples.

A template needs to be written in a way that the underlying support class is expecting: the order and kinds of values being passed to it.

In script plugins, the `exec` domain, in the Script support class, is common. It is invoked with the arguments `file` (the file to execute) and possibly `timeout` (to make the timeout value explicit, instead of the default value, for easier troubleshooting) and `exec` (to specify permissions). For example:

```
template=exec:timeout=10,file=pdk/work/scripts/sendmail/hq-sendmail-stat,exec=sudo:${alias}
```

There is also a large class of "protocol checkers" that you can use in a template for easy collection of protocol metrics, for example, for HTTP or SMTP. You can use a protocol checker for any of the platform services that are defined in HQ; all of these platform services are listed on the "New Service" screen (where you add a new platform service to the HQ inventory), in Service Type. Furthermore, you can find all the arguments you can pass to these protocol checkers on the "Resource Configuration" screen; they are the configuration properties listed in bold type. For all of the protocol checkers, the hostname is passed in; for TCP, the port number can be, but is optional. For example:

```
template=SMTP:hostname=host.example.com,port=25
```

Using a Filter to Efficiently Apply a Template to Metrics

A filter with variables can be used to easily "macro-ize" templates. The `alias` variable is particularly useful. For example:

```
<filter name="template" value="jboss.system:Type=ServerInfo:${alias}:naming.url=%naming.url
%"/>

<metric name="Free Memory"
        indicator="true"
        units="B"/>

<metric name="Used Memory"
        indicator="true"
        units="B"/>
```

> **Using Variables**
>
> In plugin XML descriptors, variables are indicated by "%" on either side of the variable name (for example, `%process.query%`). The variables are assigned the value that was most recently determined. The value of the variable must be determined *before* the variable is used.

The variable `alias`, when the filter is applied to each metric, takes on the value of the metric's alias. Neither metrics have an explicit `alias` value, so it is taken from the metric's `name`: "FreeMemory" and "UsedMemory". So, the previous code expands to:

```
<metric name="Free Memory"
        alias="FreeMemory"
        template="jboss.system:Type=ServerInfo:FreeMemory:naming.url=%naming.url%"
        indicator="true"
        units="B"/>

<metric name="Used Memory"
        alias="UsedMemory"
        template="jboss.system:Type=ServerInfo:UsedMemory:naming.url=%naming.url%"
        indicator="true"
        units="B"/>
```

Learn more about filter tags.

back to top

Getting Your Plugin to Auto-Discover Resources

HQ has already defined an autoinventory plugin for several collection methods, and for the most part, all you need to do in your own plugin is call it.

To auto-discover a server:

- Add this one line within the <server> tag:

```
<plugin type="autoinventory" class="org.hyperic.hq.product.jmx.MxServerDetector"/>
```

  The class name varies by type of plugin. That class is for a JMX plugin. For a script plugin:

```
<plugin type="autoinventory" class="org.hyperic.hq.product.DaemonDetector"/>
```

To auto-discover services on a server:

1. Add one more line that tells the plugin that the server hosts services and please discover them, too:

```
<property name="HAS_BUILTIN_SERVICES" value="true"/>
```

2. For each hosted service enumerated in the plugin, within the <service> tag, you again call the autoinventory plugin, but *without* a class argument.

```
<plugin type="autoinventory"/>
```

Return to the Plugin Development Center .

Availability

The Availability metric indicates whether a Resource is up or down.

A metrics-gathering plugin must determine Availability for *every* server and *every* service it monitors. A single plugin will likely gather Availability for multiple Resources. If Availability is not gathered for a Resource, HQ will consider the Resource to be unavailable, and will not show any metrics for it in the Portal.

A plugin sets the value of Availability to 1 if the Resource is up, and 0 if it is down. These values are displayed in the Portal as "available" or "not available".

Verifying the existence of a Resource's process is a common technique for determining its Availability. However, the method a plugin uses to determine Availability can vary depending on the Resource Type and the plugin developer's judgment. There might be alternative techniques for determining the Availability of a Resource. For instance, a plugin might determine the Availability of a web server based on whether its process is up, its port is listening, it is responsive to a request, or by some combination of these conditions.

Variables

In plugin XML descriptors, variables are indicated by "%" on either side of the variable name (for example, `%process.query%`). The variables are assigned the value that was most recently determined. The value of the variable must be determined *before* the variable is used.

4.2.2. MeasurementPlugin

*Topics marked with*relate to features available only in vFabric Hyperic.*

This page contains information that has not been reviewed. Click **Add Comment** at the bottom of the page to enter suggestions or corrections.

Overview

The MeasurementPlugin class (`org.hyperic.hq.product.MeasurementPlugin`) is a base implementation for measurement operations.

Class Hierarchy

- java.lang.Object
  - org.hyperic.hq.product.GenericPlugin
    - org.hyperic.hq.product.MeasurementPlugin

Configurable Options

Resource Properties

The table below defines properties you can define in `<property>` elements in the descriptor for a plugin that uses this class.
|| Property || Description
|| Usage
||
| XXX | xxx. | Not required.
|

Implementing Methods

This class implements following methods:

init(PluginManager):void

```
public void init(PluginManager manager)
```

Parameters:

- manager* Plugin manager

getManager():MeasurementPluginManager

```
protected MeasurementPluginManager getManager()
```

getMeasurementProperties():Map

```
protected Map getMeasurementProperties()
```

getMeasurements(TypeInfo):MeasurementInfo[]

```
public MeasurementInfo[] getMeasurements(TypeInfo info)
```

getPlatformHelpProperties():String[][]

```
protected String[][] getPlatformHelpProperties()
```

getPluginXMLHelp(TypeInfo, String, Map):String

```
protected String getPluginXMLHelp(TypeInfo info, String name, Map props)
```

getHelp(TypeInfo, Map):String

```
public String getHelp(TypeInfo info, Map props)
```

getValue(Metric):MetricValue

```
public MetricValue getValue(Metric metric)
```

getNewCollector():Collector

```
public Collector getNewCollector()
```

getCollectorProperties(Metric):Properties

```
public Properties getCollectorProperties(Metric metric)
```

translate(String, ConfigResponse):String

```
public String translate(String template, ConfigResponse config)
```

getConfigSchema(TypeInfo, ConfigResponse):ConfigSchema

```
public ConfigSchema getConfigSchema(TypeInfo info, ConfigResponse config)
```

4.2.3. Measurement Support Class

*Topics marked with*relate to features available only in vFabric Hyperic.*

tdb

Property	Use	
DOMAIN	valid on platform/environment ???	

```
<plugin>ok
...
```

```
<server name="xinetd">

 <plugin type="measurement"
            class="org.hyperic.hq.product.MeasurementPlugin"/>
...
</plugin>
```

Example 9. xinetd-plugin.xml

Examples Plugins:

xinetd Plugin XML Descriptor

Next Steps

TBD

Related Information

docbox:Win32 Measurement Plugin

Return to the docbox:Plugin Development Center .

4.2.4. SNMP Measurement Plugin

*Topics marked with*relate to features available only in vFabric Hyperic.*

Warning

This document is still a work in progress.

Plugin to collect metrics from SNMP devices.

Property	Use	
MIBS	MIB files necessary to run the plugin	

```
<plugin>
  <property name="MIBS"
            value="/etc/squid/mib.txt"/>
...
<plugin type="measurement"
            class="org.hyperic.hq.product.SNMPMeasurementPlugin"/>
...
</plugin>
```

Return to the Plugin Development Center .

4.2.5. Win32 Measurement Plugin

*Topics marked with*relate to features available only in vFabric Hyperic.*

Warning

This document is still a work in progress.

Plugin to collect metrics from Windows Services

Property	Use
tbd	tbd

```
<plugin>

<filter name="store" value="win32:Object=MSExchangeIS"/>
...
<plugin type="measurement"
            class="org.hyperic.hq.product.Win32MeasurementPlugin"/>
...
</plugin>
```

Examples Plugins:
Exchange Plugin XML Descriptor

Next Steps

TBD

Related Information

Section 4.2.1, “Measurement Plugin”

Return to the Plugin Development Center .

4.3. Event Tracking Topics

- Section 4.3.1, “Configuration File Tracking Plugin”
- Section 4.3.2, “Log4JLogTrackPlugin”
- Section 4.3.3, “Log File Tracking Plugin”
- Section 4.3.4, “Win32 EventLogTrack Plugin”

4.3.1. Configuration File Tracking Plugin

*Topics marked with***relate to features available only in vFabric Hyperic.*

The Configuration Tracking Plugin implements tracking of configuration files.

An example Plugin using Configuration Tracking Plugin is xinetd Plugin:

```
...
<plugin type="config_track"
            class="org.hyperic.hq.product.ConfigFileTrackPlugin"/>
```

```
...
```

Example 10. xinetd-plugin.xml

Properties

Property	Use	
DEFAULT_CONFIG_FILE	default config file to track	
DEFAULT_CONFIG_TRACK_ENABLE	config tracking enabled by default	

Examples Plugins:
xinetd Plugin XML Descriptor

Next Steps

TBD

Related Information

Section 4.3.3, “Log File Tracking Plugin”
Log Tracking
Configuration Tracking

Return to the Plugin Development Center .

4.3.2. Log4JLogTrackPlugin

*Topics marked with*relate to features available only in vFabric Hyperic.*

Overview

The Log4JLogTrackPlugin class (`org.hyperic.hq.product.Log4JLogTrackPlugin`) monitors a log file and returns an event for messages that contain a log4j level that is equal to or higher than a specified level and match any filter critera specifed

The table below maps the log level specified in a message to the log level assigned to the log event in HQ.

A message with log4j level ...	is presented in HQ with the log level....
"FATAL" or "ERROR"	"Error"
"WARN"	"Warning"
"INFO"	"Info"
"DEBUG"	"Debug"

Class Hierarchy

```
java.lang.Object
org.hyperic.hq.product.GenericPlugin
org.hyperic.hq.product.LogTrackPlugin
org.hyperic.hq.product.LogFileTrackPlugin
org.hyperic.hq.product.LogFileTailPlugin
org.hyperic.hq.product.Log4JLogTrackPlugin
```

Configurable Options

A resource managed by a plugin that that uses the `Log4JLogTrackPlugin` class has the configuration properties shown in the table below. The properties appear on a managed resource's **Configuration Properties** page, in the "Monitoring" section - you do not need to define them as `<options>` in a plugin descriptor.

Property	**Usage**
InventoryType `.log_track.enable`	Whether or not log tracking is enabled.
InventoryType `.log_track.level`	Defines the lowest level of message to track.
InventoryType `.log_track.include`	Specifies a substring or expression a message must match for it to be tracked as an event.
InventoryType `.log_track.exclude`	Specifies a substring or expression a message must not match for it to be tracked as an event.
InventoryType `.log_track.files`	Log files to track for the resource.

Resource Properties

The table below defines properties you can define in `<property>` elements in the descriptor for a plugin that uses this class.

Property	**Description**	**Usage**
`DEFAULT_LOG_FILE`	Sets default value of InventoryType `.log_track.files`	Not required.
`DEFAULT_LOG_LEVEL`	Sets default value of InventoryType `.log_track.level`	Not required but **highly recommended**. Set a default value of "Error" or "Warning" to prevent inadvertent logging of too many messages, which can increase HQ overhead.
`DEFAULT_LOG_TRACK_ENABLE`	Sets default value of InventoryType `.log_track.enable`	Not required. If you do not define this property, log tracking will still be disabled by default.

Return Type

The class returns an `org.hyperic.hq.product.TrackEvent` object, which contains the following attributes.

- `time` - date and time that the the message was generated.
- `level` - message log level.
- `name` - name of the file, up to 100.
- `message` - content of the message, up to 4000.

`TrackEvent` sets the maximum length for `name` and `message`, which are also specified in the schema file `sql/events/EventLog.hbm.xml`.

Example Usage

Log4jLogTrackPlugin is used by a number of HQ plugins for server types, including JBoss, Tomcat, and Glassfish.

This excerpt from the JBoss plugin sets the value of `DEFAULT_LOG_FILE` and declares the fully-qualified plugin class name.

```
<server name="JBoss"
....

<property name="DEFAULT_LOG_FILE"
value="log/server.log"/>

<plugin type="log_track"
class="org.hyperic.hq.product.Log4JLogTrackPlugin"/>
```

4.3.3. Log File Tracking Plugin

Log File Tracking Plugin

Topics marked with*relate to features available only in vFabric Hyperic.

The LogFile Tail Plugin implements tracking of logfiles.

Property	**Use**
DEFAULT_LOG_FILE	"path to log file"
DEFAULT_LOG_INCLUDE	"Java regex string" - a substring or a regular expression that must be matched by a log message.
DEFAULT_LOG_EXCLUDE	"Java regex string" - a substring that, if found in a log message, will cause the message to be excluded.
DEFAULT_LOG_TRACK_ENABLE	"true" or "false"
DEFAULT_LOG_LEVEL	Only events of a level greater than or equal to this level will be tracked. The hierarchy of log levels, from greatest to least, is:

- "Error" - corresponds to error types "emerg", "alert", "crit", and "error".
- "Warn" - corresponds to the error type "warn".
- "Info" - corresponds to error types "info" and "notice".
- "Debug" - corresponds to error type "debug".

About Event Volume

Keep in mind that event tracking increases agent overhead. Set "DEFAULT_LOG_LEVEL" to ERROR to reduce the volume of tracked events.

Example

An example Plugin using Log File Tracking is xinetd Plugin:

```
...
<property name="DEFAULT_LOG_FILE"
              value="/var/log/xinetd.log"/>
...
<plugin type="log_track"
              class="org.hyperic.hq.product.LogFileTailPlugin"/>
...
```

Example 11. hq-plugin.xml

Examples Plugins:
xinetd Plugin XML Descriptor
Zimbra Plugin XML Descriptor

Next Steps

TBD

Related Information

Section 4.3.1, “Configuration File Tracking Plugin”
Log Tracking
Configuration Tracking

Return to the Plugin Development Center .

4.3.4. Win32 EventLogTrack Plugin

*Topics marked with***relate to features available only in vFabric Hyperic.*

Warning

This document is still a work in progress.

The Win32 EventLogTrack Plugin enables the tracking of the Event Log on Windows Platforms.

Property	Use
EVENT_LOG_SOURCE_FILTER	tbd

```
<plugin>
...
<property name="EVENT_LOG_SOURCE_FILTER"
              value="MSExchange"/>

...
<plugin type="log_track"
            class="org.hyperic.hq.product.Win32EventLogTrackPlugin"/>
 ...
```

Example 12. hq-plugin.xml

Examples Plugins:

Exchange Plugin XML Descriptor

4.4. Control Topics

4.4.1. Control Plugin

*Topics marked with*relate to features available only in vFabric Hyperic.*

The ControlPlugin defines control actions and implements the doAction() method used to control resources. Like the Section 4.2.1, “Measurement Plugin”, the method of control is left entirely to the plugin. Support classes are provided to assist with certain types of control:

- JDBC
- JMX
- Script Execution
- Windows Service Manager

Following are some examples of collection methods used by various plugins:

Collection Method	**Plugins that Use It**
JMX	JBoss, WebLogic, WebSphere
JDBC	Mysql, PostgreSQL
Script Execution	Apache, Tomcat
Windows Service Manager	IIS, Apache, Tomcat

Table 2.

Property	**Use**	
BACKGROUND_COMMAND	background.sh command silently fails when running	
DEFAULT_PROGRAM	default control program	

Script Execution

```
...
<server name="Zope"
          version="2.x">
...
<!-- Use this helper class -->
 <plugin type="control"
            class="org.hyperic.hq.product.ScriptControlPlugin"/>
...

<!-- default script, visible and configurable in UI -->
<property name="DEFAULT_PROGRAM" value="zopectl"/>
...

<!-- Actions visible in UI-->
```

```
 <actions include="start,restart,stop,kill,status,test"/>

...
</server>
```

Example 13. Script Execution Plugin

Windows Service Manager

Code snippet with the required code to run Control Actions on a Windows Service.
Always specify the platform (platform="Win32") if you use a common plugin for Unix/Windows platforms.

```
<!-- Use this helper class for your Control Actions -->
<plugin type="control"
platform="Win32"
        class="org.hyperic.hq.product.Win32ControlPlugin"/>

<!-- Actions are only valid on Windows platforms -->
<actions platform="Win32"
include="start,stop,restart"/>

<!-- Specify service name as a configurable option and setting the default value-->
<config type="control" platform="Win32">
<option name="service_name"
default="Apache2"
description="Apache Service Name"/>
```

Example 14. Windows Service Manager Control Actions

ControlPlugin.doAction Method

Control actions are defined in the Plugin XML Descriptor. Server and Service resources can include an **<actions>** tag that will define the control actions that resource supports. Multiple control actions can be defined by separating the actions with a comma. For example:

```
<actions include="start,stop,restart"/>
```

Example 15. Defining Control Actions

These actions are passed into doAction as a String argument. The plugin can then act accordingly. Each resource that supports control will have its own ControlPlugin instance. Configuration parameters defined within the Plugin XML Descriptor**<config>** tags can be retrieved using the ControlPlugin.getConfig method.

An example using a JBoss JMS Destination, which uses JMX for its control actions:

```
<!-- ObjectName properties used in the control plugins -->
  <property name="JMSQueue"
     value="jboss.mq.destination:service=Queue,name=%jms.destination%"/>

  ...
  <service name="JMS Destination">
    <config>
      <option name="jms.destination"
                description="JMS Destination"
                default=""/>
    </config>

    <actions include="removeAllMessages"/>
    ...
  </service>
  ...
```

Example 16. hq-plugin.xml

```
public class JBossJMSControlPlugin extends ControlPlugin {

    public void doAction(String action) throws PluginException {

        // ObjectName to invoke the method on
        String oName = getProperty("JMSQueue"); //From hq-plugin.xml

        try {
            ... // Get a reference to the MBeanServer
            mBeanServer.invoke(oName, action, new Object[0], new String[0]);
            ...
        } catch (Exception e) {
            throw new PluginException("Unable to invoke method '" +
                                      action + "'", e);
        }
    }
}
```

Example 17. JBossJMSControlPlugin.java

Return to the Plugin Development Center .

4.4.2. Mx Server Control Plugin

*Topics marked with*relate to features available only in vFabric Hyperic.*

Run Control Actions by JMX

An example Plugin using Mx Server Control Plugin is the jonas plugin

```
...
<plugin type="control"
            class="org.hyperic.hq.product.jmx.MxServerControlPlugin"/>
...
```

Example 18. jonas-plugin.xml

Properties

Property	Use

Examples Plugins:
jonas Plugin XML Descriptor
terracotta Plugin XML Descriptor

Next Steps

TBD

Related Information

Section 4.4.1, “Control Plugin”

Section 4.4.4, “Win32 Control Plugin”
Server Control Tracking

Return to the Plugin Development Center .

4.4.3. Script Control Plugin

*Topics marked with***relate to features available only in vFabric Hyperic.*

Run Control Actions with a script or binary

An example Plugin using Script Control Plugin is the

```
...
<plugin type="autoinventory"
class="org.hyperic.hq.product.ScriptControlPlugin"/>
...
```

Example 19. -plugin.xml

Properties

Property	**Use**	
BACKGROUND_COMMAND		

Examples Plugins:
foo Plugin XML Descriptor

Next Steps

TBD

Related Information

Section 4.4.4, “Win32 Control Plugin”

Return to the Plugin Development Center .

4.4.4. Win32 Control Plugin

*Topics marked with***relate to features available only in vFabric Hyperic.*

Warning

This document is still a work in progress.

Plugin to run Control Actions on Windows platforms.

Option	**Use**	
service_name	Windows Service Name	

```
<plugin>
...
<plugin type="control"
            platform="Win32"
            class="org.hyperic.hq.product.Win32ControlPlugin"/>

    <actions platform="Win32"
             include="start,stop,restart"/>

    <config type="control" platform="Win32">
      <option name="service_name"
              default="Apache2"
              description="Apache Service Name"/>
...
</plugin>
```

Example 20. Example

Example Plugins:

Apache Plugin XML Descriptor
IIS Plugin XML Descriptor

Return to the docbox:Plugin Development Center .

4.5. ConfigResponse

*Topics marked with***relate to features available only in vFabric Hyperic.*

This page contains information that has not been reviewed. Click **Add Comment** at the bottom of the page to enter suggestions or corrections.

4.5.1. Overview

The ConfigResponse class (`org.hyperic.util.config.ConfigResponse`) is used throughout HQ source code to store and transfer configuration data. From end user point of view this class acts as a key/value storage. Usually you use this class to add configuration properties to new resources created during auto discovery methods.

4.5.2. Class Hierarchy

- java.lang.Object
  - org.hyperic.util.config.ConfigResponse

4.5.3. Implementing Methods

This class implements following methods:

setValue(String, String):void

```
public void setValue(String key, String value)
throws InvalidOptionException, InvalidOptionValueException;
```

Set the value for an option.
Parameters:

- key* The name of the option to set
- value* The value to set the option to.

Throws:
InvalidOptionException - If this ConfigResponse does not support the specified option.
InvalidOptionValueException - If the value supplied is not a legal/valid value for the option.

4.5.4. Example Usage

```
private static final String PTQL_QUERY = "State.Name.ct=firefox";

public List getServerResources(ConfigResponse config) throws PluginException {
    List servers = new ArrayList();

    String installPath = "";

    ConfigResponse productConfig = new ConfigResponse();

    productConfig.setValue("process.query", PTQL_QUERY);
    ServerResource server = createServerResource(installPath);
    setProductConfig(server, productConfig);
    server.setMeasurementConfig();
    servers.add(server);

    return servers;
}
```

4.6. ServerResource

*Topics marked with***relate to features available only in vFabric Hyperic.*

This page contains information that has not been reviewed. Click **Add Comment** at the bottom of the page to enter suggestions or corrections.

4.6.1. Overview

The ServerResource class (`org.hyperic.hq.product.ServerResource`) is used to store information for newly discovered servers during the autodiscovery methods. This class contains everything what goes into an autoinventory report which is created by agent and later sent back to the HQ server.

This class stores the following information:

- resource - Resource itselves represented by this object. Most of the class methods are modifying this resource. Default constructor creates an empty resource object.
- fqdn - This is a resource fully qualified domain name. This field is not used unless resource should belong under a different platform than running agent is handling.
- productConfig - Main configuration properties for resource. This reflects to the shared section in resource inventory section within the UI.
- metricConfig - Configuration for measurement section.

- controlConfig - Configuration for control section.
- cprops - Resource custom properties.

4.6.2. Class Hierarchy

- java.lang.Object
  - org.hyperic.hq.product.ServerResource

4.6.3. Implementing Methods

This interface implements following methods:

setInstallPath(String):void

```
public void setInstallPath(String name)
```

Sets the resource installation path. More about installation path see Installpath.

Parameters:

- name* Path to a installation directory

getInstallPath():String

```
public String getInstallPath()
```

Returns the resource intallation path.

setPlatformFqdn(String):void

```
public void setPlatformFqdn(String name)
```

Sets the resource fully qualified domain name. This attribute should only be set if discovered server is hosted on a platform other than the platform which ran the auto discovery. The WebLogic plugin for example may find cluster nodes on other platforms. Note that setting this attribute will also require the given platform to exist in the HQ inventory.

Parameters:

- name* Name of the FQDN

getPlatformFqdn():String

```
public String getPlatformFqdn()
```

Gets the resource FQDN.

Returns:

NULL if field has not been set.

addService(ServiceResource):void

```
public void addService(ServiceResource service)
```

Adds new service resource to this server. More about discovering resources see Discovering resources.

Parameters:

- service* New service resource to be added to this server.

addServiceType(ServiceType):void

```
public void addServiceType(ServiceType serviceType)
```

Adds new service type to this server. More about discovering resource types see Discovering resource types.

Parameters:

- serviceType* New service type to be added. This type will eventually go back to server together with autoinventory report.

setIdentifier(String):void

```
public void setIdentifier(String name)
```

Sets the autoinventory identifier (AIID) for this resource. More about AIID see AIID.

Parameters:

- name* Autoinventory identifier.

getIdentifier():String

```
public String getIdentifier()
```

Returns the resource autoinventory identifier.

setType(String):void

```
public void setType(String name)
```

Set type of this resource. Type of the server resource is mapped with the names defined in plugin descriptor. If server type name is defined as `<server name="My Server">` in xml then name passed to this method should match the one from plugin descriptor. In this case you would set the type to be `"My Server"`. If server type definition contains references to versions (using version attribute), type has to be referenced respectively. For example if server is defined as `<server name="My Server" version="1.x">`, type would be set as `"My Server 1.x"`.

Parameters:

- name* Resource type as String.

setType(GenericPlugin):void

```
public void setType(GenericPlugin plugin)
```

Derive the resource type directly from the implementing plugin. Autodiscovery implementation knows what is the current resourcy type it is interset in.

Parameters:

- plugin* The plugin handling the discovery operation.

getType():String

```
public String getType()
```

Returns the current resource type name.

setName(String):void

```
public void setName(String name)
```

Sets the name of this resource.

Parameters:

- name* Name of the resource.

getName():String

```
public String getName()
```

Returns the name of the resource.

setDescription(String):void

```
public void setDescription(String description)
```

Sets the description of this resource.

Parameters:

- description* Description of the resource.

getDescription():String

```
public String getDescription()
```

Returns the description of the resource.

setProductConfig(ConfigResponse):void

```
public void setProductConfig(ConfigResponse config)
```

Setting product config for this resource. This config is mapping the resource shared section under configuration properties. More about ConfigResponse see Section 4.5, “ConfigResponse”.

Parameters:

- config* Resource shared configuration.

setProductConfig(Map):void

```
public void setProductConfig(Map config)
```

Setting product config for this resource. This config is mapping the resource shared section under configuration properties. Product configuration is passed to resourse using Map object. Internally ConfigResponse is using Map to store its keys and values.

Parameters:

- config* Map of resource configuration.

setProductConfig():void

```
public void setProductConfig()
```

Sets and initializes an empty product config.

getProductConfig():ConfigResponse

```
public ConfigResponse getProductConfig()
```

Returns the resource shared configuration.

setMeasurementConfig(ConfigResponse):void

```
public void setMeasurementConfig(ConfigResponse config)
```

Setting measurement config for this resource. This config is mapping the resource monitoring section under configuration properties. More about ConfigResponse see Section 4.5, “ConfigResponse”.

Parameters:

- config* Resource measurement configuration.

setMeasurementConfig(Map):void

```
public void setMeasurementConfig(Map config)
```

Setting measurement config for this resource. This config is mapping the resource monitoring section under configuration properties. Product configuration is passed to resourse using Map object. Internally ConfigResponse is using Map to store its keys and values.

Parameters:

- config* Map of resource measurement configuration.

setMeasurementConfig():void

```
public void setMeasurementConfig()
```

Sets and initializes an empty measurement config.

setMeasurementConfig(ConfigResponse, int, boolean):void

```
public void setMeasurementConfig(ConfigResponse config,
                                 int logTrackLevel,
                                 boolean enableConfigTrack)
```

Setting measurement config for this resource. This config is mapping the resource monitoring section under configuration properties. Product configuration is passed to resourse using Map object. Internally ConfigResponse is using Map to store its keys and values.

This function can be used to enable log and config tracking at the same time. Following log levels are defined by LogTrackPlugin:

public static final int LOGLEVEL_ANY = -1;
public static final int LOGLEVEL_ERROR = 3;
public static final int LOGLEVEL_WARN = 4;
public static final int LOGLEVEL_INFO = 6;
public static final int LOGLEVEL_DEBUG = 7;

Parameters:

- config* Resource measurement configuration.
- logTrackLevel* Log tracking level in internal type of int.
- enableConfigTrack* Enables config tracking if TRUE, use FALSE otherwise.

getMeasurementConfig():ConfigResponse

```
public ConfigResponse getMeasurementConfig()
```

Returns resource measurement config.

setControlConfig(ConfigResponse):void

```
public void setControlConfig(ConfigResponse config)
```

Setting control config for this resource. This config is mapping the resource control section under configuration properties. More about ConfigResponse see Section 4.5, “ConfigResponse”.

Parameters:

- config* Resource control control configuration.

setControlConfig(Map):void

```
public void setControlConfig(Map config)
```

Setting control config for this resource. This config is mapping the resource control section under configuration properties. Product configuration is passed to resourse using Map object. Internally ConfigResponse is using Map to store its keys and values.

Parameters:

- config* Map of resource control configuration.

setControlConfig():void

```
public void setControlConfig()
```

Sets and initializes an empty control config.

getControlConfig():ConfigResponse

```
public ConfigResponse getControlConfig()
```

Returns resource control configuration.

setCustomProperties(ConfigResponse):void

```
public void setCustomProperties(ConfigResponse config)
```

Setting resource custom properties. These properties are mapping to values seen in top of the resource page. Values which are unmodifiable through UI.

Parameters:

- config* Resource custom properties.

setCustomProperties(Map):void

```
public void setCustomProperties(Map props)
```

Setting resource custom properties. These properties are mapping to values seen in top of the resource page. Values which are unmodifiable through UI. Custom properties are passed to resourse using Map object. Internally ConfigResponse is using Map to store its keys and values.

Parameters:

- config* Resource custom properties.

getCustomProperties():ConfigResponse

```
public ConfigResponse getCustomProperties()
```

Returns resource custom properties.

4.7. ServiceResource

*Topics marked with*relate to features available only in vFabric Hyperic.*

This page contains information that has not been reviewed. Click **Add Comment** at the bottom of the page to enter suggestions or corrections.

4.7.1. Overview

The ServiceResource class (`org.hyperic.hq.product.ServiceResource`) is used to store information for newly discovered services during the autodiscovery methods. This class contains everything what goes into an runtime autoinventory report which is created by agent and later sent back to the HQ server.

4.7.2. Class Hierarchy

- java.lang.Object
  - org.hyperic.hq.product.ServiceResource

4.7.3. Implementing Methods

This interface implements following methods:

setName(String):void

```
public void setName(String name)
```

Sets the resource name.

Parameters:

- name* Name of the resource.