# Endpoint Operations Management Agent Plug-in Development Kit

vRealize Operations Manager 6.1

vmware®

You can find the most up-to-date technical documentation on the VMware Web site at:

https://docs.vmware.com/

The VMware Web site also provides the latest product updates.

If you have comments about this documentation, submit your feedback to:

docfeedback@vmware.com



**VMware, Inc.**
3401 Hillview Ave.
Palo Alto, CA 94304
www.vmware.com

# Contents

# About the Endpoint Operations Management Agent Plug-in Development Kit

The *Endpoint Operations Management Agent Plug-in Development Kit* documents the XML plug-in descriptor that is the basis of every plug-in. Endpoint Operations Management supports classes for auto-discovery, measurement, control, and other management functions. It provides information about developing VMware vRealize Operations Manager product plug-ins to manage new object types.

## Intended Audience

This information is intended for developers who build or customize plug-ins.

## VMware Technical Publications Glossary

VMware Technical Publications provides a glossary of terms that might be unfamiliar to you. For definitions of terms as they are used in VMware technical documentation, go to http://www.vmware.com/support/pubs.

# 1 Introduction to Plug-in Development

Plug-ins are the interface between vRealize Operations Manager and products on the network you want to manage.

You can develop your own plug-ins to extend the functionality of vRealize Operations Manager coverage to products or parts of products not yet covered. For information about the Endpoint Operations Management source code and the plug-ins provided by VMware, see https://github.com/vmware/ep-ops-management.

## What Plug-ins Do

Plug-in development requires an understanding of the vRealize Operations Manager inventory model and of the management functions that plug-ins implement. Management functions can include:

**Auto-Discovery** Plug-ins can implement auto-discovery of server resources, services resources, and application resources. Custom plug-ins usually just call the vRealize Operations Manager built-in ServerDetector class.

**Monitoring** Plug-ins can implement metric collection, defining and collecting metrics and configuring them for display in the vRealize Operations Manager user interface. Measurement plug-ins implement monitoring.

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

This chapter includes the following topics:



## The Role of the Server and Agent in Plug-ins

Plug-ins exist on the server and you can download them automatically to the agents.

The agent gathers all the data from resources and generally communicates with the resource. Using the plug-in, the agent can:

- Auto-discover resources
- Collect resource metrics

The server manages metadata, including:

- Platform, server, and service resource types and how the plug-in's targeted resources map to the inventory model.
- The configuration schema for each resource.

## Technical Overview

vRealize Operations Manager plug-ins are self-contained .jar or .xml files that are deployed on both the server and every agent that you want to run the plug-in. Every plug-in contains, at a minimum, an XML descriptor, which is either a standalone .xml file or embedded in the .jar file.

## Plug-in Implementations

Consider measurement, control, and so on, as types of plug-ins. These types of plug-ins can be created for any type of object.

You write different implementations of plug-in types, depending on the type of object and how it communicates and presents its data. The different implementations are:

- Script
- JMX
- SQL
- SNMP

## Using Support Classes to Simplify a Plug-in

vRealize Operations Manager includes a number of support classes that you can invoke in your own plug-ins to abstract and simplify its construction.

vRealize Operations Manager provides the following support classes:

**Table 1-1.** vRealize Operations Manager Support Classes

| Category | Support Classes | When to Invoke the 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, and so on | 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 | |

# Writing Plug-ins

While the interface with vRealize Operations Manager plug-ins is straightforward, you also must determine how to retrieve data from a managed resources and how it should appear in the inventory model, and at what level.

## Plug-in Naming

Plug-in names must be in the following formats, where `PluginName` is the name of the plug-in, as specified in the root `plugin` element of the plug-in descriptor:

- `PluginName-plugin.jar` for a plug-in that contains program or script files in addition to the plug-in XML descriptor.
- `PluginName-plugin.xml` for a plug-in that consists only of the plug-in XML descriptor.

## JMX Plug-in

Auto-discovery (called "auto-inventory" within plug-ins) is easily implemented by implementing a vRealize Operations Manager-provided `autoinventory` plug-in.

To implement auto-discovery at the server level, you must invoke an autoinventory plug-in with the `MxServerDetector` class 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, driven by the `OBJECT_NAME` property. To implement auto-discovery at the service level, invoke the autoinventory plug-in, leaving out the class attribute, within a service tag:

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

<plugin type="autoinventory"/>

...
</service>
```

The JMX plug-in 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 are auto-configured.

By default, auto-discovered service names are 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"
```

You can override the naming using the `AUTOINVENTORY_NAME` property:

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

You can use the configuration properties from the platform, hosting server, and the service itself in the `%replacement%` strings, resulting in a name such as:

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

## Discovering Custom Properties

Discovery of Custom Properties is supported using the `OBJECT_NAME` and `MBeanServer.getAttribute`.

You 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>
```

that maps to the following MBean interface method:

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

## Custom MBean Plug-in Examples

Here are examples of MBean plugins that you can use to assist you in creating your plug-ins.

**`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>
```

**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"/>

    <!-- 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>
```

## Script Plug-ins

A script plug-in is a plug-in that runs one or more scripts that return process metrics.

A script plug-in uses the `org.hyperic.hq.product.DaemonDetector` support to discover resources from the process table — `DaemonDetector` runs a PTQL `process.query`.

### Script Plug-in Contents and Packaging

A script plug-in comprises the following components.

- An XML plug-in descriptor that defines the monitored process and its properties, along with the metrics that the script reports.
- A script that returns metric name: value pairs.

  You can embed your script in the XML plug-in descriptor, in which case you deploy only the XML file. If your script is in its own file, you reference it in the descriptor, and deploy an archive containing the script and the descriptor.

Save the script in `AgentHome/bundles/AgentBundleDir/pdk/scripts/`, or in the XML descriptor.

### Requirements for Script

Note the following requirements for writing a script plug-in.

- The script can be written in any required language. For example, Batch for Windows and Shell for Linux.
- A measurement script must report metrics as name-value pairs. For example,

  ```
  % ./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
  ```

- Unicode characters must be escaped.

  Unicode characters in a script are decoded during script processing. For example, the string `%3D` is decoded to an equals sign (=). To preserve the value of s string that might be interpreted as a unicode character, escape the string with a double backslash, for example: `\\%3D`.

### Defining the Proxy Resources in the Plug-in Descriptor

If the plug-in manages a single process, model the monitored process as a platform service. Specify it in a `<service>` element in the root `<plugin>` element of the descriptor.

If the plug-in manages a server and its component services, script reports on a multiple services, create a server-service hierarchy. Specify the parent <server> element in the root <plugin> element of the descriptor, and specify each of the component services as a child <service> element.

## Defining Management Functions in a Script Plug-in

A script plug-in can perform various management functions, including auto-discovery, measurement and control.

### Auto-Discovery

Script plug-ins use the org.hyperic.hq.product.DaemonDetector auto-discovery support class to discover a process. DaemonDetector requires a PTQL process query.

Determine the PTQL statement that identifies the target process. The most common query types for discovering a process are:

| Query Type | Description |
|---|---|
| State.Name.eq=BASENAME | Where BASENAME is the base name of the process executable (or regex ) and uniquely identifies it. For example, State.Name.eq=crond . |
| Pid.PidFile.eq=PIDFILE | Where PIDFILE is the path and name of the process PID file. For example, Pid.PidFile.eq=/var/run/sshd.pid.<br>This query is useful if the process base name does not uniquely identify the process. |
| Pid.Service.eq=SERVICENAME | Where SERVICENAME is the name of the process.<br>This query is useful in Windows environments. For example, Pid.Service.eq=Eventlog. |

You can supply multiple, comma-separated PTQL queries, if necessary.

For a Java process, you typically must specify command line arguments for the process to identify it.

To define auto-discovery in the plug-in descriptor, if you have defined a server-service hierarchy, in the <server> define a <property> element whose name is HAS_BUILTIN_SERVICES and value="true", so that component services are discovered.

When you define the auto-discovery function, identify org.hyperic.hq.product.DaemonDetector as the class that performs it in a <plugin> element whose type is "autoinventory". If you defined a server-service hierarchy, put the <plugin> element in the <server> element. If the plug-in manages a single service, put it in the <service> element that models the process to discover.

You also need to define the process query in an <option> element whose name is process.query and whose default is the PTQL query in the same resource element that contains the <plugin> element.

### Measurement

Script plug-ins use the org.hyperic.hq.product.MeasurementPlugin class to report the metrics returned by the scripts. MeasurementPlugin accepts metric name:value pairs.

You define the measurement function and identify MeasurementPlugin as the class that performs it in a <plugin> element whose type is measurement. If you have defined a server-service hierarchy, put the <plugin> element in the <server> element. If your resource "hierarchy" is simply a single platform service, put <plugin> element in the <service> element that models the process.

You must also define a <metric> element for each metric reported by the script. You must define at least the name, indicator, and template attributes.

The form of a metric template for a metric collected by a script is:

```
exec:timeout=TIMEOUT,exec=PREFIX,file=FILENAME,exec=MODE,args=ARGUMENTS:METRIC
```

where

| | |
|---|---|
| TIMEOUT | The time in seconds to wait for a response when the script runs. Optional, but recommended. |
| PREFIX | The script prefix, for example, sudo. |
| FILENAME | The path and name of script that returns the metric. Mandatory. |
| ARGUMENTS | A space-separated list of argument values to pass to the script. |
| METRIC | The name of the metric. |

For example, `exec:timeout=10,exec=sudo,file=pdk/scripts/metric_script.pl,args=sda:w/s`.

## Script Plug-in Examples

Use these examples to help you create your own script plug-ins.

### Control Script Example

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin name="hqcont-1-script-solution">

  <script name="controlscript.bat">
<![CDATA[
echo controlscript called
]]>
  </script>

  <script name="controlscript.sh">
<![CDATA[
#!sh
echo controlscript called
]]>
  </script>

  <server name="HQCONT-1 My Control Server">

    <property name="PROC_QUERY"
              value="State.Name.eq=firefox"/>

    <config>
      <option default="State.Name.eq=firefox"
              name="process.query"
              description="Process Query for singleprocess"/>
    </config>

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

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

    <config>
      <option name="program"
              description="control program"
              default="controlscript.bat"/>
    </config>

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

```
    <property name="DEFAULT_PROGRAM" value="controlscript.bat"/>

    <actions include="start"/>

  </server>

</plugin>
```

### iostat Script Example

```
<pluginname="IoDevice">
    <property name="version"
                 value="1.0"/>
  <service name="I/O Device">

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

      <option name="device"
              description="Device name"
              default="sda"/>
    </config>

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

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

    <metric name="Read Requests Merged per Second"
            category="THROUGHPUT"
            template="${template}:rrqm/s"/>

    <metric name="Write Requests Merged per Second"
            category="THROUGHPUT"
            template="${template}:wrqm/s"/>

    <metric name="Read Requests per Second"
            category="THROUGHPUT"
            indicator="true"
            template="${template}:r/s"/>

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

    <metric name="Sectors Read per Second"
            category="THROUGHPUT"
            template="${template}:rsec/s"/>

    <metric name="Sectors Writen per Second"
```

```
            category="THROUGHPUT"
            template="${template}:wsec/s"/>

    <metric name="Average Sector Request Size"
            category="THROUGHPUT"
            template="${template}:avgrq-sz"/>

    <metric name="Average Queue Length"
            category="PERFORMANCE"
            template="${template}:avgqu-sz"/>

    <metric name="Average Wait Time"
            category="PERFORMANCE"
            indicator="true"
            units="ms"
            template="${template}:await"/>

    <metric name="Average Service Time"
            category="PERFORMANCE"
            units="ms"
            template="${template}:svctm"/>

    <metric name="CPU Usage"
            category="PERFORMANCE"
            units="percent"
            template="${template}:%util"/>

  </service>

</plugin>
```

### sendmail Plug-in Descriptor

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

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

<!--
  NOTE: This copyright does *not* cover user programs that use HQ
  program services by normal system calls through the application
  program interfaces provided as part of the Hyperic Plug-in Development
  Kit or the Hyperic Client Development Kit - this is merely considered
  normal use of the program, and does *not* fall under the heading of
  "derived work".

  Copyright (C) [2004-2008], Hyperic, Inc.
  This file is part of HQ.

  HQ is free software; you can redistribute it and/or modify
  it under the terms version 2 of the GNU General Public License as
  published by the Free Software Foundation. This program is distributed
  in the hope that it will be useful, but WITHOUT ANY WARRANTY; without
  even the implied warranty of MERCHANTABILITY or FITNESS FOR A
  PARTICULAR PURPOSE. See the GNU General Public License for more
```

```
  details.

  You should have received a copy of the GNU General Public License
  along with this program; if not, write to the Free Software
  Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307
  USA.
 -->

<plugin>
  <!-- extracted to: pdk/work/scripts/sendmail/hq-sendmail-stat -->
  <script name="hq-sendmail-stat">
  <![CDATA[
#!/bin/sh

# linux / aix
[ -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"

# If the mqueue dir doesn't exist, exit 1
[ -z "$msgdir" -o ! -r "$msgdir" ] &&
    exit 1

# May not have permission to cd to the mqueue, make sure
# stdout/err don't get echo'd
cd $msgdir > /dev/null 2>&1
[ "$?" != "0" ] &&
    exit 1

# count msgs in sendmail mqueue dir.  DO NOT use find since it
# may fail when there are lots of files
messfiles=`ls 2>/dev/null | wc -w`

premessfiles=0

if [ ! -z "$premsgdir" ] && [ -d "$premsgdir" ] && [ -r "$premsgdir" ]
then
    [ `cd $premsgdir > /dev/null 2>&1` ] && [ "$?" = "0" ] &&
        premessfiles=`ls 2>/dev/null | wc -w`
fi

echo MessagesInQueue=$messfiles
echo MessagesAwaitingPreprocessing=$premessfiles

exit 0
  ]]>
  </script>

  <server name="Sendmail"
          version="8.x">
```

```
<property name="INVENTORY_ID" value="sendmail"/>
<!-- hardwire this cosmetic to universal location -->
<property name="INSTALLPATH" value="/usr/sbin/sendmail"/>

<config>
  <option name="process.query"
          description="Process Query"
          default="State.Name.eq=sendmail,State.Name.Pne=$1,CredName.User.eq=root"/>
  <option name="exec"
          description="Type &quot;sudo&quot; To Avoid Having Agent As Root"
          default=""/>
</config>

<!--notifies the plugin to auto-discover one instance of each service-->
<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="Message Submission Process">
  <config>
    <option name="user"
            default="smmsp"
            description="Sendmail Message Submission Process User"/>

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

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

<service name="Root Daemon Process">
  <plugin type="autoinventory"/>
  <config>
```

```
      <option name="process.query"
              default="State.Name.eq=sendmail,State.Name.Pne=$1,CredName.User.eq=root"
              description="PTQL for Sendmail Root Daemon Process"/>
    </config>
    <metric name="Availability"
            template="sigar:Type=MultiProcCpu,Arg=%process.query%:Processes"
            indicator="true"/>
    &multi-process-metrics;
  </service>

  <!-- sendmail-stat metrics -->
  <filter name="template"
          value="exec: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"/>

  <!-- protocol services+metrics -->
  <service name="SMTP">
    <config>
      <option name="port"
              description="SMTP Post"
              default="25"/>
      <option name="hostname"
              description="SMTP Hostname"
              default="localhost"/>
    </config>
    <filter name="template"
            value="SMTP:hostname=%hostname%,port=%port%:${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">
<![CDATA[
<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>
  <help name="Sendmail 8.x" include="Sendmail"/>
</plugin>
```

## SNMP Plug-in

SNMP is the standard protocol for monitoring network-attached devices, which is leveraged by several bundled plug-ins and made easy by the Plug-in Development Kit.

The bundled netdevice plug-in provides a generic network device platform type that 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 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 vRealize Operations Manager plug-in, there are two main concepts to understand:

**The Inventory Model**

Resource types define where things live in the hierarchy along with supported metrics, control actions, log message sources, and so on, 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, and so on.

**The Metric Template Attribute**

The metric template attribute which is a string containing all the information required to collect a specific data point. In an SNMP plug-in, each of the metrics correlate to an SNMP OID. Although the object names are frequently used to gather the required data points in the plug-ins, you can also use the numeric OID. This has the added benefit of negating the need for ready access to the MIB file anywhere that the plug-in is used.

Implementing a new SNMP-based plug-in for vRealize Operations Manager starts with locating the device vendor's MIB files and selecting which OIDs to collect as metrics in vRealize Operations Manager.

## JMX-Based Management

provides support for managing and monitoring JMX-enabled applications.

vRealize Operations Manager has a number of built-in plug-ins that monitor specific JMX products, including:

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

vRealize Operations Manager uses the remote API (http://www.jcp.org/en/jsr/detail?id=160) specified by JSR-160 to manage products that support JMX 1.2/JSR-160, including the ones listed above. For JMX-enabled servers that do not support JSR-160, vRealize Operations Manager uses vendor-specific connectors.

vRealize Operations Manager JMX support classes enable auto-discovery of MBean servers and MBeans, collection of MBean attributes, and execution of MBean operations.

To enable monitoring, you must configure the JMX-enabled target to accept remote connections. In many cases, the remote connector is enabled by default, otherwise, you must configure it for remote access.

| JMX Product | Product Information Link |
|---|---|
| J2SE 1.5 | http://download.oracle.com/javase/1.5.0/docs/api/javax/management/remote/package-summary.html |
| MX4J | http://mx4j.sourceforge.net/docs/ch03.html |
| ActiveMQ | http://activemq.apache.org/jmx.html |
| JOnAS | http://jonas.objectweb.org/current/doc/doc-en/integrated/howto/JSR160_support.html#JSR160_support |
| ServiceMix | http://servicemix.apache.org/jmx-console.html |

## Auto-Discovery of JMX Resources

vRealize Operations Manager discovers a JMX application or server using a Sigar process query.

MBeans are discovered by querying the MBean server for MBeans whose names match those configured in the plug-in descriptor. Sigar is used to discover servers. Services are discovered via MBean Server queries (`MBeanServer.queryMBeans()`).

**Measurement**

`MxMeasurement` uses Sigar queries for process metrics. Metrics that map to MBean attributes are obtained via an MBean query (`MBeanServer.getAttribute()`).

**Note** Sun JVM 1.5 type applies to any of the above and any other JMX-enabled 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 auto-discovered.

## Configuration Properties for JMX Monitoring

vRealize Operations Manager JMX support classes require the JMX URL and JMX user credentials so that they can connect to a remote MBean server.

The required credentials are as follows:

| | |
|---|---|
| jmx.url | The JMX service URL. See http://download.oracle.com/javase/1.5.0/docs/api/javax/management/remote/JMXServiceURL.html |
| jmx.username | Username, if authentication is required |
| jmx.password | Password, if authentication is required |

Configuration options that a user can configure are defined in a `<config>` element in a plug-in descriptor. The Plug-in Development Kit includes a global configuration schema named `jmx` that contains the required configuration option definitions, as follows.

```
<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"/>
```

You can use the following to reference the jmx schema in a plug-in descriptor.

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

## Creating a Custom JMX Plug-in

A JMX plug-in consists solely of an XML descriptor. You can include various components in the descriptor.

### 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 plug-ins 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 the version attribute must be 1.5, or any of the other supported server/version combinations. The name attribute is the choice of the plug-in implementor.

These services will become part of the inventory model, displayed together with the built-in server service types in the user interface and the shell. Service extensions also inherit the server configuration properties that are used to connect to the MBeanServer, jmx.url, jmx.username and jmx.password.

### Defining an ObjectName to Access Custom MBeans

To access custom MBeans, the plug-in 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"/>
```

### Defining Configuration Properties to Appear in the User Interface

All the configuration properties for a JMX plug-in, appear in the user interface for the object. The default values for each of these properties can be specified in the plug-in, but users can change the values by editing the resource identifiers in the vRealize Operations Manager user interface.

If there is only one instance of the String Cache, you can hard-code a property. If you are using multiple instances that follow the OBJECT_NAME pattern, you use configuration properties to support them.

For example, the WebApp Cache plug-in 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 differ for each instance of the MBean.

The WebApp Cache plug-in defines configuration 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 plug-in. For example,

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

### Defining and Gathering Metrics

Metrics are defined as for other plug-ins but, in the case of custom MBean services, the `OBJECT_NAME` property is used to compose the metric template attribute.

Use the `OBJECT_NAME` property as follows:

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

This results in the template being expanded. For example,

```
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. For example,

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

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

The MBean interface attributes collected by `tomcat-webapp-cache-plugin.xml` as metrics are as follows.

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

### Specifying the Availability Metric for MBeans

vRealize Operations Manager JMX plug-ins typically query for an MBean's "Availability" attribute to determine whether the MBean is available.

If the MBean server returns `1`, the MBean is considered available. If the return value is `0`, the MBean is considered unavailable. Other values cause availability to display incorrectly.

Many MBeans do not have an `Availability` attribute, therefore vRealize Operations Manager JMX plug-ins treat an Mbean to as available if the query returns an `AttributeNotFoundException` exception, assuming that the MBean is available to report that the attribute does not exist. If the MBean server returns any exception other than `AttributeNotFoundException`, the MBean is considered to be unavailable.

### Implementing Control Actions

After the `OBJECT_NAME` property is defined, MBean operations can be exposed as vRealize Operations Manager control actions by adding the list of method names to the plug-in.

Add the list of method names as follows.

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

The plug-in must also define the control implementation class, which resides in the `hq-jmx.jar` file.

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

The control actions are invoked as MBean operations by the plug-in, as follows

```
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();
}
```

### Example: WebApp Cache Control Actions

The `WebApp Cache` plug-in example provides the following control actions:

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

which maps to the following MBean operations:

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

### Defining the Server Auto-Inventory Element

To implement auto-discovery at the server level, you must invoke an autoinventory plug-in with the `MxServerDetectorclass` within the server tag.

Implement auto-discovery as follows,

```
<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, driven by the `OBJECT_NAME` property. To implement auto-discovery at the service level, invoke the autoinventory plug-in, leaving out the class attribute, within a service tag.

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

<plugin type="autoinventory"/>

...
</service>
```

The JMX plug-in uses the `MBeanServer.queryNames` method to discover a service for each MBean instance. In the case in which the `OBJECT_NAME` contains configuration properties, the properties are auto-configured.

By default, auto-discovered service names are composed using the hosting-server name, configuration properties, and service type name.

```
"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 `%eplacement%` strings, resulting in a name such as follows,

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

### Discovering Custom Properties

Discovery of custom properties is supported using OBJECT_NAME and `MBeanServer.getAttribute`.

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();
}
```

## Running and Testing Plug-ins from the Command Line

You can run plug-ins from a command line prompt, which you might find useful when documenting or testing your plug-in.

You can test the syntax of a plug-in and invoke any management function that the plug-in supports.

Management functions that can be invoked include the following:

| Function | Description |
|---|---|
| Auto-discovery | Run the discovery function for one or all plug-ins in the agent's plug-in directory. |
| Control | Run a plug-in 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 to obtain CPU, filesystem, and other system data. |

Documentation generation functions that can be invoked include the following:

| Function | Description |
|---|---|
| Help | Output the configuration help specified in the plug-in `<help>`descriptor element for each resource type, for one or all plug-ins. |
| Metric documentation | Output metric documentation for each resource type, for one or all plug-ins. |

## dcs-tools-pdk.jar Command Syntax

To run a plug-in from the command line, it is important that you understand the syntax and the functions of each of the methods.

The command for running a plug-in from the command line is structured as follows:

`java –jar AgentVersion/bundles/AgentBundle/pdk/lib/dcs–tools–pdk–VERSION.jar –m Method –a MethodAction –p PluginName –t ResourceType –Doption=value`

You can use the following information to guide you in your choices.

#### `–m Method`

The `–m Method` command specifies the method to run.

The *Method* can be one of the following:

| | |
|---|---|
| `lifecycle` | For details and functionality, see "lifecycle Method," on page 30. |
| `discover` | For details and functionality, see "discover Method," on page 31. |
| `metric` | For details and functionality, see "metric Method," on page 31. |
| `control` | For details and functionality, see GUID-8713429D-A37D-42B9-BEE3-67B8EBDAA6C0#GUID-8713429D-A37D-42B9-BEE3-67B8EBDAA6C0. |
| `track` | For details and functionality, see "track Method," on page 35. |
| `generate` | For details and functionality, see "generate Method," on page 35. |

#### `–p PluginName`

The `–p PluginName` command is the product portion of the plug-in name, without the `–plugin.jar` or `–plugin.xml` portion. For example, to run `jboss–plugin.jar`, you specify `–p jboss`.

If you use a generated properties file to supply resource properties, you do not have to specify the plug-in to run on the command line, because the resource properties file identifies the plug-in.

The command is required for the following methods.

- `lifecycle`
- `metric`
- `control`
- `track`

The command is optional for the following methods.

- `discover`
- `generate`

The command is not supported for `livedata`.

#### `–t ResourceType`

The `–t ResourceType` command specifies the name of a resource type managed by the plug-in you are running.

If the name includes spaces, you must enclose it in quotes, for example, `"JBoss 4.2"`.

If you use a generated properties file to supply resource properties, you do not have to specify the resource type on the command line, because the resource properties file identifies the resource type name for the resource.

The command is required for the following methods.

- `metric`
- `control`
- `track`
- `lifedata`

The command is not supported for the following methods.

- `discover`
- `generate`

***-a MethodAction***

The *MethodAction* argument is either supported or required by the method that is 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.

***-DOption=Value***

`-D`*`Option=Value`* sets a property value, where `Option=Value` specifies the property name and the value that you assign it.

You must include a `-DOption=Value` in the command line for every property that you specify. In addition, you must supply

- The value of a resource property that is required by the method called.

  You can reference a generated properties file, rather than supplying each resource property on the command line.

- The value of an agent or system property that manages agent behavior or plug-in execution.

## Generating and Using Resource Properties Files

You can create resource properties files to use when you run plug-ins from the command line. Using a resource properties file removes the need for you to specify individual property values multiple times in the command line.


- Resource Properties Files on page 28

  Generally, plug-ins require the values of one or more resource properties to run. To simplify the process of testing a plug-in, you can supply the properties in a file instead of the command line.

- Names and Locations of Properties Files on page 28

  The `discover` method's properties action writes configuration data for each discovered object in a directory tree whose root directory, `plug-in-properties`, is in your current working directory.

- Content of Properties Files on page 28

  When you run the `metric`, `control`, or `track` methods on an object you must supply resource configuration data, either explicitly on the command line, or using the properties file for the resource.

- Inherited Resource Properties on page 29

  Some resource properties might be inherited from a parent resource.

## Resource Properties Files

Generally, plug-ins require the values of one or more resource properties to run. To simplify the process of testing a plug-in, you can supply the properties in a file instead of the command line.

For example, to fetch metrics for a PostgreSQL table, the `metric` method must know the URL and database user credentials for the parent PostresSQL server, and the name of the table. The required properties are `jdbcUser`, `jdbcPassword`, `table`, and `jdbcUrl`.

Each property that a method requires for a resource type is defined in an `<option>` element in the XML descriptor for the plug-in that manages it.

When you run the discover method with the `properties` method argument, the agent creates a properties file for each resource instance it discovers. The properties file for a resource contains a name-value pair for each resource property that is required to run plug-in methods.

The configurable properties that you 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 must set the value of `server.log_track.exclude` which is `null` by default.

The following command supplies some command options and resource properties using the `melba_HQ_jBoss_4.x.properties` file and sets the value of `server.log_track.exclude` on the command line.

```
java –jar java –jar AgentVersion/bundles/AgentBundle/pdk/lib/dcs–tools–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 Locations of Properties Files

The `discover` method's properties action writes configuration data for each discovered object in a directory tree whose root directory, `plug–in–properties`, is in your current working directory.

The `plugin–properties` folder contains a subdirectory for each object type discovered. The folder name is the object type name, with spaces replaced by dashes, for example, `Tomcat–6.0–Connector`.

Each object type folder contains a file for each instance of that type discovered. The file name is the full name of the object 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` methods on an object you must supply resource configuration data, either explicitly on the command line, or using the properties file for the resource.

The properties file simplifies the command by defining the values that you would otherwise set with the `–p` and `–t` options.

The following example of discovery results for a JBoss 4.2 server is used to explain the properties file content.

```
# 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
```

The properties file contains:

- The object's resource type name and the product portion of the name of the plug-in that manages it:

| | |
|---|---|
| `dumper.plugin` | Specifies the product portion of the plug-in name. This is equivalent to setting the plug-in name in the command line with `–p`. |
| `dumper.type` | Specifies the resource type name. This is 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 an resource. The sample JBoss properties file above supplies values for `java.naming.provider.url`, `program`, and `server.log_track.files`.

### Inherited Resource Properties

Some resource properties might 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. The only service-level property in this file in `Application`.

```
# 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
```

## Properties for Controlling Agent Behavior and Plug-in Execution

You can use `–D`*`Option=Value`* to set any agent or system property.

This table lists some properties that are useful when you run a plug-in from the command line.

| Property | Description |
|---|---|
| `log` | Use this property to set the log level. log=debug. |
| `output.dir` | Use this property to override the default output directory default. |
| `plugins.include` | This agent property tells theEndpoint Operations Management agent to load a specific plug-in, and only that agent before executing the method. Otherwise, when you run `dcs–tools–pdk–shared–VERSION.jar` the Endpoint Operations Management agent loads all the plug-ins in the plug-in directory. |
| `plugins.exclude` | This agent property gives the agent a list of plug-ins that must not be loaded before executing the method. The Endpoint Operations Management agent loads all other plug-ins in the plug-in directory. |
| `exec.sleep` | Use this system property to override its default value when you are testing a script plug-in. 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 plug-in out. |

## Methods and Functions of the dcs-tools-pdk.jar File

You can use the methods and functions that are specified in the `dcs–tools–pdk.jar` file when you create your plug-ins.

## lifecycle Method

You use the `lifecycle` method to load a plug-in and report any errors found in the plug-in.

### Syntax

The syntax for the `lifecycle` method is as follows.

```
$ java –jar bundles/agent–VERSION/pdk/lib/dcs–tools–pdk–VERSION.jar –p PluginName –m lifecycle –
Dplugins.include=PluginName
```

| Argument | Description |
|---|---|
| `–p PluginName` | Identifies the plug-in to run by the product portion of the plug-in name. For example, to run `jboss–plugin.jar`, you specify `–p jboss`. |
| `–Dplugins.include=PluginName` | Ensures that only the specified plug-in is loaded. When this is not included, all plug-ins are loaded. |

### Example: Results of Running the lifecycle Method on a Plug-in Without Errors

This command runs the `lifecycle` method for the jboss plug-in. In the example, no errors are found.

```
$ java –jar bundles/agent–VERSION/pdk/lib/dcs–tools–pdk–VERSION.jar –m lifecycle –p jboss –
Dplugins.include=jboss
```

### Example: Results of Running the lifecycle Method on a Plug-in Containing Errors

This command runs the `lifecycle` method for the websphere plug-in. In the example, errors are found.

```
$ java –jar bundles/agent–VERSION/pdk/lib/dcs–tools–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 Method

You use the `discover` method to return key attributes for each discovered object to the terminal window or to a properties file.

The method can be run for one or all plug-ins. The returned attributes include the values of the resource's configuration options. If you save discovery results to a file, you can use that file to supply the required resource configuration data when you run another method that requires the resource's configuration data.

### Syntax

The syntax for the `discover` method is as follows.

```
{{java –jar bundles/agent–VERSION/pdk/lib/dcs–tools–pdk–VERSION.jar –m discover –p PluginName –a properties
```

| Argument | Description |
|---|---|
| `–p PluginName` | Identifies the plug-in to run by the product portion of the plug-in name. For example, to run `jboss–plugin.jar`, you specify `–p jboss`. If you do not specify the plug-in name, discovery is performed for all the plug-ins in the agent's `plugin` directory. |
| `–a properties` | Writes the discovery results to files. If you do not use this option, the results are returned only to the terminal window. |

You can use any of the following `discovery` method options.

| Command | Purpose | Comments |
|---|---|---|
| `–m discover` | To run discovery for all plug-ins. | Results are returned to the terminal window. |
| `–m discover –p jboss` | To run discovery for one plug-in, (in this case, JBoss). | Results are returned to the terminal window. |
| `–m discover –p jboss –a properties` | To run discovery for a plug-in (JBoss) and save results to files. | Results are written to files, and to the terminal window. |
| `–m discover –a properties` | To run discovery for all plug-ins and save results to files | Results are written to files, and to the terminal window. |

## metric Method

You use the `metric` method to fetch the metric template and the metric value for each metric for objects that are managed by the plug-in.

You can also use the method for the following:

- Metrics that are collected by default
- Metrics belonging to a specific metric category
- Metrics that are indicator metrics
- To return only the metric template, without the metric values, for the metrics
- Fetch metrics repeatedly for a specified number of times, and return the time that it took to perform the fetch action for each metric.

### Syntax for the metric Method When Using a Resource Properties File

The syntax for running the `metric` method using a properties file to supply resource configuration data is as follows.

```
java –jar bundles/agent–VERSION/pdk/lib/dcs–tools–pdk–VERSION.jar –m metric plugin–
properties/ResourceTypeDirectory/ResourceName.properties –a translate –Dmetric–collect=default –
Dmetric–indicator=true –Dmetric–cat=CATEGORY –Dmetric–iter=ITERATIONS
```

| Argument | Description |
|---|---|
| `plugin–properties/ResourceTypeDirectory/ResourceName.properties` | The path to the file generated when the resource was discovered using the `properties` action of the discover method.<br>The properties file provides the values for:<br>■ `–p PluginName`: the product portion of the plug-in name.<br>■ `–t ResourceType`: the object type name<br>■ The value of the configuration options for the object |
| `–a translate` | Causes metric templates, but not metric values, to be returned.<br>If `–a translate` is not specified, both metric templates and metric values are returned |

You can also use one of the following arguments to limit the metrics that are returned. If you do not use one of these arguments, all metrics are returned.

| Argument | Description |
|---|---|
| `–Dmetric–collect=default` | This option limits the results to metrics that have the `defaultOn` attribute set to `true`. |
| `–Dmetric–indicator=true` | This option limits the results to metrics that have the `indicator` attribute set to `true` |
| `–Dmetric–cat=`*`CATEGORY`* | This option limits the results to metrics of a specific category.<br>The categories are `AVAILABILITY`, `UTILIZATION`, `THROUGHPUT`, and `PERFORMANCE.` |
| `–Dmetric–iter=`*`ITERATIONS`* | This option causes the time (in milliseconds) to collect a metric repeatedly to be reported, rather than the metric value.. |

### Syntax for the metric Method that Specifies Configuration Data on Command Line

The syntax for running the `metric` method that supplies resource configuration data on the command line data is as follows.

```
java –jar bundles/agent–VERSION/pdk/lib/dcs–tools–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
```

| Argument | Description |
|---|---|
| `–p PluginName` | Specifies the product portion of the plug-in name. |
| `–t ResourceType` | Specifies the resource type name. |
| `–a translate` | Causes metric templates, but not metric values, to be returned.<br>If `–a translate` is not specified, both metric templates and metric values are returned |

Optionally, one of the following options is also specified, to limit the metrics that are returned. If you do not specify one of these options, all metrics are returned.

| Argument | Description |
|---|---|
| –Dmetric–collect=default | Limits the results to metrics that have the defaultOn option set to true. |
| –Dmetric–indicator=true | Limits the results to metrics that have the indicator option set to true. |
| –Dmetric–cat=*CATEGORY* | Limits the results to metrics of a specific category. The categories that are available are AVAILABILITY , UTILIZATION, THROUGHPUT, or PERFORMANCE. |
| –Dmetric–iter=*ITERATIONS* | Causes the time, in milliseconds, to collect a metric repeatedly |
| –DOption=*Value* | Specifies the value of a resource configuration option.<br>The command line must include a -DOption=*Value* for each resource configuration option. |

### Example Invocations

In the following examples, only the method invocation and command options are shown. The java –jar AgentHome/bundles/AgentBundle/pdk/lib/dcs–tools–pdk–VERSION.jar portion of the command is not shown.

| Command | Purpose | Comments |
|---|---|---|
| –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 | To fetch metrics for a JBoss server. | Resource configuration data is supplied on the command line |
| –m metric plugin–properties/jboss–4.2/melba_HQ_jBoss_4.x.properties | To fetch metrics for the jboss server supplying the configuration data using a properties file. | Resource configuration data is supplied by a properties file. |
| Add –Dmetric–collect=default to the command line. | To limit the results to indicator metrics. | If you use this option, do not use –Dmetric–cat=*CATEGORY* or –Dmetric–indicator=true . |
| Add –Dmetric–cat=*CATEGORY* to the command line, where *CATEGORY* is AVAILABILITY, UTILIZATION, THROUGHPUT, or PERFORMANCE. | To limit the results to metrics of a specific category. | If you use this option, do not use –Dmetric–collect=default or –Dmetric–indicator=true . |
| Add –Dmetric–indicator=true to the command line. | To limit the results to indicator metrics. | If you use this option, do not use –Dmetric–collect=default or –Dmetric–cat=*CATEGORY*. |
| Add –Dmetric–iter=*ITERATIONS* to the command line where *ITERATIONS* is the number of times to run getValue for each metric . | To collect each metric multiple times and report how long it took to do so (in milliseconds), instead reporting the metric value. | You can use this option in conjunction with one of the following:<br>■ –Dmetric–collect=default<br>■ –Dmetric–cat=*CATEGORY*<br>■ –Dmetric–indicator=true |
| Add –m metric plugin–properties/jboss–4.2/melba_HQ_jBoss_4.x.properties –a translate to the command line. | To fetch the metric template, but not the metrics, for the JBoss server. | |

#### Example: Results Returned by the -metric Method default Action

This example is an excerpt from the results of running the default action of the metric method. Both metric templates and metric values are returned.

**Note** Colons In `metric` templates appear as "%3A" in the results.

```
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%<=
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=
=>125.0<=
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=
=>15.0<=
JBoss 4.2 JVM Free Memory:
jboss.system:type=ServerInfo:FreeMemory:java.naming.provider.url=jnp
%3A//0.0.0.0%3A2099,java.naming.security.principal=%java.naming.security.principal
%,java.naming.security.credentials=
=>365.9 MB<=
```

#### Example: Results Returned by the metric Method translate Action

This example is an excerpt from the results of running the translate action of the metric method. Metric templates are returned but metric values are not returned.

**Note** Colons In `metric` templates appear as "%3A" in the results.

```
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:
```

## track Method

You use the `track` method to track log or configuration events.

### Syntax

The syntax for the `track` method is as follows.

```
ava –jar /bundles/agent–VERSION/pdk/lib/dcs–tools–pdk–VERSION.jar –p PluginName –t
"ResourceType" –m track –a TrackAction –Dserver.config_track.files=TrackFiles
```

| Argument | Description |
|---|---|
| *PluginName* | Identifies the plug-in to run. |
| `ResourceType` | Specifies a resource type managed by the plug-in. |
| `TrackAction` | Specifies whether to track log events or configuration events. Use with the `log` or `track` options. |

**Note** You can use a properties file instead of specifying

```
–pPluginName
–t ResourceType
–Dserver.config_track.files=TrackFiles
```

### Example: Various Usage Options

In the following examples, only the method invocation and command options are shown. The `java –jar AgentHome/bundles/AgentBundle/pdk/lib/dcs–tools–pdk–VERSION.jar` portion of the command is not shown.

| Command | Purpose |
|---|---|
| `–p apache –t "Apache 2.0" –m track –a config –Dserver.config_track.files=/etc/httpd/httpd.conf` | To 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 log –Dserver.log_track.files=/var/log/httpd/error_log` | To track log entries to `/var/log/httpd/error_log` for an "Apache 2.0" server . |
| Add the required property definition to the command line, for example, `–Dserver.log_track.exclude=String` | To implement other configurable tracking behaviors. For example, 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. |

## generate Method

You use the `generate` method to generate documentation from the plug-in descriptor.

### Syntax

The syntax for the `generate` method is as follows.

```
java –jar bundles/agent–VERSION/pdk/lib/dcs–tools–pdk–VERSION.jar –p PluginName –m generate –a
GenerateAction
```

| Argument | Description |
|---|---|
| *PluginName* | Identifies the plug-in to document.<br>If not specified, the action is applied to all plug-ins. |
| `GenerateAction` | Specifies a type of document to generate.<br>Available options are:<br>■ `metrics-wiki` Writes a Confluence Wiki-formatted summary of supported metrics to a file.<br>■ `metrics-xml` Outputs an XML-formatted summary of supported metrics for an object type to `stdout`.<br>■ `metrics-txt` Outputs a text-formatted summary of supported metrics for an object type to `stdout`.<br>■ `help`<br>Outputs the contents of the `<help>` element for each object type in the plug-in descriptors to HTML files in the `./plugin-help` directory. |
| `TrackAction` | Specifies whether to track log events or configuration events.<br>Use with the `log` or `track` options. |

### Example: Various Usage Options

In the following examples, only the method invocation and command options are shown. The `java -jar AgentHome/bundles/AgentBundle/pdk/lib/dcs-tools-pdk-VERSION.jar` portion of the command is not shown.

| Command | Purpose |
|---|---|
| `-m generate -a metrics-wiki` | To document metrics in Confluence Wiki format for all object types in all plug-ins. |
| `-m generate -a metrics-txt` | To document metrics in text format for all object types in all plug-ins. |
| `-m generate -a metrics-xml` | To document metrics in XML format for all object types in all plug-ins. |
| `-m generate -a help` | To generate a help page for all object types in all plug-ins. |
| Append `-p` *`jboss`* to the command line. | To limit results to the object types managed by a single plug-in, in this example, JBoss. |

## Running Protocol Checks from the Command Line

In addition to running a plug-in from the command line to test or document the plug-in, you can use the command line option to quickly retrieve metrics on-demand.

For example, you can run the netservices plug-in from the command line to check the availability of a variety of network service types.

To monitor an object of one of the types listed below on an on-going basis, you configure it as a platform service object on the required platform. The Endpoint Operations Management agent performs remote availability checks and metric collection.

To enable monitoring, you supply object configuration data, at a minimum, the hostname of the service object.

To run the plug-in from the command line, you must supply the required configuration data on the command line.

The netservices plug-in can monitor remote objects of the following types.

- HTTP
- POP3
- IMAP

- SMTP
- FTP
- LDAP
- DNS
- SSH
- NTP
- DHCP
- SNMP
- RPC
- InetAddress Ping
- TCP Socket

### Example: Run the netservices Plug-in metric Method for a Remote LDAP Server

You use the following command to run the netservices plug-in `metric` method for a remote LDAP server.

The value of each configuration option for the LDAP service object is supplied using a `-D` argument.

```
java -jar bundles/agent-VERSION/pdk/lib/dcs-tools-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
```

# Using Auto-Discovery Support Classes in Plug-ins 2

You can use the vRealize Operations Manager auto-discovery functionality in a custom plug-in. As most platform types are discovered by the system plug-in, custom plug-ins discover server and service resource types.

## Auto-Discovery Implementation

The auto-discovery class performs the discovery process. For many resource types, you can reference one of the vRealize Operations Manager built-in auto-discovery classes. If necessary, you can write a custom auto-discovery class that extends a vRealize Operations Manager auto-discovery class. Most of the vRealize Operations Manager auto-discovery implementations discover two levels of resources - servers, and the services that run in them, so typically you only specify a single implementation in the descriptor.

## Parameters Required by the Implementation

In addition to specifying the auto-discovery class, the plug-in descriptor must define the parameters that the class requires.

This chapter includes the following topics:



## Auto-Discovery Classes

Auto-discovery rules for a resource type are defined in the XML descriptor of the plug-in that manages the type.

### Auto-Discovery Class Hierarchy

VMware vRealize Operations Manager auto-discovery class is as follows.

```
org.hyperic.hq.product.GenericPlugin
          org.hyperic.hq.product.ServerDetector
               org.hyperic.hq.product.PlatformServiceDetector
               org.hyperic.hq.product.DaemonDetector
                    org.hyperic.hq.product.MxServerDetector
                         org.hyperic.hq.product.SunMxServerDetector
                    org.hyperic.hq.product.SNMPDetector
```

## Overview

The table below describes each of the classes in the auto-discovery class hierarchy.

**Table 2-1.** Auto-Discovery Classes

| Class | Description | When to Use |
|---|---|---|
| `ServerDetector` | Abstract class.<br>`ServerDetector` is the base auto-discovery class. | `ServerDetector` is an abstract class and must be inherited, rather than used directly. It may be extended by a custom auto-discovery class. |
| `PlatformServiceDetector` | Abstract class.<br>Intended for use by platform types with service types, but no server types. | |
| `DaemonDetector` | Discovers server types via a Sigar query of system process table. | |
| `MxServerDetector` | Discovers JMX server types via a Sigar query of system process table. Discovers JMX services by MBean query. | |
| `SunMxServerDetector` | Detector for Sun 1.5+ JVMs with remote JMX enabled. Note, JVM resource must be explicitly configured. | |
| `SNMPDetector` | Discovers SNMP server types via a Sigar query of system process table. Discovers SNMP services view SNMP request. | |

# Auto-Discovery Interfaces

The built-in auto-discovery classes in VMware vRealize Operations Manager each implement one or more interfaces.

The interfaces that are implemented are listed below.

**`org.hyperic.hq.product.AutoServerDetector`**
This interface is used by the default scan, which discovers servers by scanning the process table or Windows registry.

**`org.hyperic.hq.product.FileServerDetector`**
This interface is used by the file system scan. Plug-ins specify file patterns to match in `etc/hq-server-sigs.properties`.

When a file or directory matches one of these patterns, the method is invoked. The plug-in uses the matched file or directory as a hint to find server installations.

**`org.hyperic.hq.product.RuntimeDiscoverer`**
This interface is used by the run-time scan. This differs from the default and filesystem scan, which do not necessarily require a server to be running before it can be detected. Classes that implement the `RuntimeDiscoverer` interface communicate directly with a running target server to discover resources.

# Specifying Auto-Discovery Implementation for a Resource Type

You specify the class that performs auto-discovery for a resource type in a <plugin> element of type `autoinventory`.

The following is an example of an auto-discovery Implementation for a resource type.

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

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

    ...

</server>
```

## Supplying Parameters for Auto-Discovery Implementation

All auto-discovery implementations discover server types by scanning the process table or Windows registry for processes that match a Sigar process query.

You specify the process query in an `option` element named `process.query` (inside a `<config>` element) in the `<server>` element for a server type. Data that you define in an `option` element appears as part of a resource, and can be edited as required.

Data defined in a property element cannot be edited for a resource in the VMware vRealize Operations Manager user interface. The parameters that are required to discover services vary by plug-in implementation. Discovery of JMX services requires `ObjectNames`, discovery of SNMP services requires an OID.

# Measurement Plug-ins

A measurement plug-in is the part of a resource plug-in that implements metric collection. It can identify a resource instance and collect metrics for it.

A custom plug-in that leverages the vRealize Operations Manager base measurement classes consists of only the plug-in XML descriptor. For example, most custom JMX measurement plug-ins use the vRealize Operations Manager JMX measurement classes. To write a custom JMX plug-in you typically just define the resources and the required metrics in an XML file, which is the complete plug-in.

## Writing the XML Descriptor

Each plug-in requires an XML descriptor that specifies the type of plug-in, the resources to look for and the metrics to collect from them. The topic describes the major elements that you must include in that descriptor file.

## Measurement Support Classes

Here are the vRealize Operations Manager classes for metric collection.

**`org.hyperic.hq.product.MeasurementPlugin`**

This is the vRealize Operations Manager base measurement class. The `getValue()` method is called when a plug-in is asked for a metric value. This class is extended by the following classes, each of which specifies a `getValue()` method for a specific type of metric collection.

- JMX
- JDBC

| | |
|---|---|
| | ■ Sigar<br>■ ... |
| **org.hyperic.hq.product.JDBCMeasurementPlugin** | Obtains database server and database metrics using JDBC. Measurement classes in Hyperic plug-ins that monitor database servers extend this class. Such plug-ins include:<br>■ Mysql<br>■ PostgreSQL<br>■ Oracle<br>■ Sybase |
| **org.hyperic.hq.product.jmx.MxMeasurementPlugin** | Obtains MBean attribute values. Measurement classes in Hyperic plug-ins that monitor application servers extend this class. Plug-ins include:<br>■ JBoss<br>■ WebLogic<br>■ WebSphere<br>■ Tomcat<br>■ Resin |
| **org.hyperic.hq.product.SigarMeasurementPlugin** | Uses SIGAR API to obtain system and process data. The vRealize Operations Manager system plug-in uses this class to monitor system and process information for operating system platform types such as Linux, Win32, and so on. |
| **org.hyperic.hq.product.SNMPMeasurementPlugin** | Obtains metrics from SNMP-enabled resources. Measurement classes in vRealize Operations Manager plug-ins that use this class include Apache. |
| **org.hyperic.hq.product.Win32MeasurementPlugin** | Collects Windows Perflib data. |

### Specifying the Measurement Plug-in in the Plug-in Descriptor

You identify the measurement class for a resource type in the plug-in descriptor, in a `<plugin>` element in the resource element that defines the resource type.

The resource element could be a `<platform>`, `<server>`, or `<service>` element. For example, a plug-in that uses `org.hyperic.hq.product.MeasurementPlugin` to collect metrics for a server type resource includes a `<plugin>` element like the one in the following code snippet.

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

## Defining Measurements Using the metric Tag

A measurement plug-in collects metrics. In the plug-in descriptor, you define a `<metric>` element for each metric to be collected for an object type.

You must always collect the availability metric.

The availability metric indicates whether an object is up or down.

A metrics-gathering plug-in must determine availability for every object that it monitors. A single plug-in gathers availability for multiple objects. If availability is not gathered for an object, the object is determined to be unavailable.

A plug-in sets the value of availability to 100 if the object is up, and 0 if it is down. These values are displayed in the user interface as available or unavailable.

Verifying the existence of an object's process is a common technique for determining its availability. However, the method a plug-in uses to determine availability can vary depending on the object type and a plug-in developer's judgment. There might be alternative techniques for determining availability of an object. For example, a plug-in 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 table describes each metric attribute, most of which, are intended for use by the server to control display of the metric data.

**Table 2-2.** `metric` Tag Attributes

| Metric Attribute | Description | Required Yes/No | Possible Values |
|---|---|---|---|
| `name` | The name that appears for the metric in the vRealize Operations Manager user interface. | Yes | |
| `alias` | The abbreviated name of the metric, displayed in the plug-in's output (name-value pairs).<br>If not specified, `alias` defaults to the value of name, with any white space and any non-alphanumeric characters removed. | No | In the case of a JMX metric, alias exactly matches the name of the MBean attribute that supplies the metric value. |
| `category` | The category of metric. In the vRealize Operations Manager user interface, a user can group metrics by category on the Metric Data tab for the object. | No | ■ `AVAILABILITY`<br>This is the default category for a metric whose name attribute is "Availability".<br>■ THROUGHPUT<br>■ PERFORMANCE<br>■ UTILIZATION<br>This is the default category for a metric, except for a metric whose name is "Availability". |

**Table 2-2.** `metric` Tag Attributes (Continued)

| Metric Attribute | Description | Required Yes/No | Possible Values |
|---|---|---|---|
| `units` | The units of measurement for the metric, which affects how metric values are displayed and labelled in the vRealize Operations Manager user interface | No | None<br>Is not formatted.<br>Uses the following abbreviations:<br>■ percentage<br>■ B: Bytes<br>■ KB: Kilobytes<br>■ MB: Megabytes<br>■ GB: Gigabytes<br>■ TB: Terabytes<br>■ epoch-millis: Time since January 1, 1970 in milliseconds<br>■ epoch-seconds: Time since January 1, 1970 in seconds<br>■ ns: Nanoseconds<br>■ mu: Microseconds<br>■ ms: Milliseconds<br>■ sec: Seconds<br>If the name attribute is Availability, defaults to percentage, otherwise defaults to none. |
| `indicator` | Whether the metric is an indicator metric in vRealize Operations Manager. Indicator metrics are marked as KPI in the vRealize Operations Manager user interface. | No | ■ true<br>■ false |
| `template` | Expresses a request for a specific metric, for a specific object, in a format that the Endpoint Operations Management agent recognizes. It identifies the object instance, a specific metric, and where to obtain the metric value. A metric template takes the form `Domain:Properties:Metric:Connection` | No | The content of each segment of the metric template depends on how the metric is obtained: from an MBean server, SIGAR, a measurement class, through SNMP, and so on. |
| `defaultOn` | When `true`, this measurement is scheduled by default. | No | If `indicator` is true defaults to `true`, otherwise defaults to `false`. |

## Example: Simple `metric` Tag

```
<metric name="Availability"
            category="AVAILABILITY"
            units="percentage"
            indicator="true"/>
```

### Example: Complex `metric` Tag

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

## Using Templates to Collect Metric Data

Metric templates enable plug-ins 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`.

```
boss.system:Type=ServerInfo:FreeMemory:naming.url=%naming.url%
```

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 are 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. You can see this use of templates in many of the plug-in examples.

A template must be written in a way that the underlying support class can recognize, specifically, the order and kinds of values being passed to it.

In script plug-ins, the `exec` domain, in the script support class, is common. It is invoked using the arguments `file` (the file to execute) and possibly `timeout` (to make the timeout value explicit, for easier troubleshooting, instead of using the default value) 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 the Endpoint Operations Management agent.

## Setting a Plug-in to Auto-discover Resources

The Endpoint Operations Management agent has already defined an autoinventory plug-in for several collection methods. Generally, you only need to call it in your own plug-in.

### Auto-discovering a Server Resource

You can specify auto-discovery of a server by adding the following to line to a `<server>` tag.

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

The class name varies by type of plug-in. The class in the code snippet is for a JMX plug-in. For a script plug-in you use the following.

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

## Auto-discovering Services Resources

You can specify the auto-discovery of a services running on the server by adding another line so that the plug-in recognizes that the server is hosting services that it must discover.

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

For each hosted service enumerated in the plug-in, within the `<service>` tag, you again call the autoinventory plug-in, but without a class argument.

```
<plugin type="autoinventory"/>
```

# Working with Plug-in Descriptors 3

A plug-in descriptor is an XML file that defines what a plug-in does and how. It defines the object types it manages and, for each object type, specifies the management functions it performs, the resource data it requires and discovers, and the metrics it returns.

Every plug-in has a descriptor file. If a plug-in uses Endpoint Operations Management plug-in support classes or a script to perform management functions, the descriptor is the component to develop and deploy. The descriptor for a plug-in that uses custom management classes is packaged with the classes in a JAR for deployment.

This chapter includes the following topics:



## Hierarchy of Managed Object Types

A plug-in descriptor defines each object type that the plug-in manages.

In some cases there is only a single type, but more typically the descriptor defines a hierarchy of types, for example a server object (for example, Tomcat 6.0) and its service objects (for example, Vhosts).

A plug-in can manage multiple object type hierarchies. The descriptor for such plug-ins defines an object hierarchy for each version.

Although a plug-in can manage a platform object and one or more levels of dependent objects, in practice virtually all platform-level objects are managed by a single Endpoint Operations Management `system-plug-in.jar` plug-in. The system plug-in discovers and manages all supported OS platform objects and platform services objects, such as the network interface, CPU, and file server mount service objects for each platform object.

The only other Endpoint Operations Management plug-ins that manage objects that are determined by Endpoint Operations Management to be platform objects are those that manage virtual or network hosts.

## Management Functions and Classes for Object Types

A plug-in can perform one or more management function for each object type that it manages.

For example, the Tomcat plug-in 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.

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

The available management functions include the following.

| Plug-in Management Function | Description |
|---|---|
| Discover resources and resource data | Discovers running instances of and object type and collects object data. For example, an Apache server's build date and the path to its executable. |
| Obtain metrics | Measures or collects metrics that reflect the availability, throughput, and utilization of an object instance. |
| Monitor log files | Monitors log files for messages that match specified filter criteria, such as severity level or text that the message includes (or does not include). |
| Monitor configuration files | Monitors specific files for changes. |

## Inventory and Configuration Data for Object Types

For each object type that the plug-in manages, the descriptor defines the resource data that the plug-in uses, including data that the plug-in requires so that it can discover a resource, such as the address of an MBean server, or object attributes that the plug-in discovers.

## Metrics to Collect for Each Object Type

The plug-in descriptor specifies each metric that the plug-in obtains for each object type it manages.

For example, the Tomcat plug-in obtains "Availability", "Current Thread Count" and "Current Threads Busy" metrics 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).

## Structure of a Plug-in Descriptor

The structure of a plug-in descriptor is the same as the hierarchy of the object types that the plug-in manages, expressed in terms of the inventory model.

A plug-in descriptor contains a `<platform>` , `<server>` , or `<service>` object element for each object type that must be managed. The object element hierarchy in the descriptor must reflect relationships between the managed object types. For example, a `<server>` element for a Tomcat type contains (is the parent of) the `<service>` element for the Vhost type.

In the following table, the left column illustrates all of the object element relationships that are valid in a plug-in descriptor. Elements that map to object types are shown in bold. No element attributes are shown, and some lower level elements are excluded. The child elements below each object type element are used to define the object data, plug-in functions, and metrics for that object type. The right column illustrates the descriptor structures for resource hierarchies of varying depth.

| Supported Element Relationships | Element Structures for Various Object Hierarchies |
|---|---|
| <plugin><br><filter><br><property><br><config><br><option><br><properties><br><help><br><metrics><br><script><br><classpath><br><platform><br><filter><br><property><br><config><br><properties><br><plugin><br><help><br><metrics><br><metric><br><actions><br><classpath><br><script><br><server><br><filter><br><property><br><config><br><option><br><properties><br><plugin><br><help><br><metrics><br><metric><br><actions><br><scan><br><service><br><filter><br><property><br><config><br><option><br><properties><br><plugin><br><help><br><metrics><br><metric><br><actions><br><server><br><service><br><service><br><br>..... | **Note** Server Object - Service Object<br>Most Endpoint Operations Management plug-ins manage a server object and the service objects that it contains. The <server> element is the root of the plug-in and contains a <service> element for each of the service objects that the plug-in manages. The descriptor for a plug-in that manages multiple versions of a server object, for example the plug-in for Tomcat 5.5 and 6.0, defines a <server> - <service> hierarchy for each.<br><plugin> (root)<br><server><br><service><br><server><br><service><br>**Note** Platform Object - Platform Service Object<br>Non-typical<br>The system plug-in manages all of the Endpoint Operations Management supported OS platform objects, and the service objects that run on each platform. The plug-in descriptor defines a <platform> - <service> hierarchy for each OS platform object.<br><plugin><br><platform><br><service><br>.....<br><platform><br><service><br>.....<br>**Note** Platform Object - Server Object - Service Object<br>Non-typical<br>This structure is valid but uncommon. The only Endpoint Operations Management plug-ins that manage platform-server-service objects are plug-ins that manage virtual platform objects.<br><plugin><br><platform><br><server><br><service><br>**Note** Platform Service Object<br>If a plug-in manages only a platform service object, the <service> element appears in the root of the plug-in.<br><plugin><br><service> |

# Functionality of Plug-in Descriptor Elements

The hq-plugin.xml defines the plugin. It defines what the plug-in does and how, including which metrics to collect, the units of measurement, the type of metric, and other attributes that characterize metric nature and behavior.

In addition, the xml defines one or more management functions and the class or script that runs each of them. It also defines the resource data that the plug-in uses, and related user interface behaviors including whether and where resource properties are displayed in the user interface, defaults and permissible values for configurable data, and so on.

You can use the following table to determine the elements that you can define for each object that a plug-in manages.

| Element | Description and Usage |
|---|---|
| Configurable resource data<br>■ <config><br>■ <option> | <config> is a mandatory container element for <option> elements. A named <config> element is a reusable building block that can be included by reference in other <config> elements. This is useful when you define a group of options that apply to multiple managed objects in the object plug-in. You can designate a <config> element as global, in which case <config> elements in other plug-in descriptors can also reference it.<br><option> specifies a resource attribute with a value that must be supplied by the user. It can be supplied in the descriptor or by a plug-in class, but it must be editable. You can define the permissible values for a selector list, whether they are optional or mandatory, and so on. |
| Non-configurable resource data<br>■ <properties><br>■ <property> | <properties> is a container element for one or more <property> elements.<br><property> specifies a non-configurable resource attribute. Its value might be discovered (for example RAM, or CPU speed), returned by a plug-in class, or defined in the descriptor.<br>Resource data defined as a <property> cannot be entered or edited in the user interface. |
| Management functions for a resource type<br>■ <plugin><br>■ <actions> | <plugin> specifies a management function (auto-discovery, measurement, control, log tracking, and so on) for a resource type, and the Java class that performs that function.<br>Each management function for a resource type is specified in a separate <plugin> element.<br><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 control-type <plugin> element. |
| Metrics for a resource type<br>■ <metrics><br>■ <metric> | <metrics> is container for one or more <metric> elements. A named <metrics> element in the root of the plug-in is a reusable building block that 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 object types that are managed by the plug-in.<metric> specifies a measurement that the plug-in obtains for an object type. The attributes in the <metric> element define the type of metric (availability, throughput, utilization), units of measure, whether the metric is an indicator, and so on. |
| Metrics for use within the descriptor<br>■ filter | The filter element defines a name and value pair variable that you can use in the descriptor. filter is meaningful only within the descriptor.<br>The filter element make a descriptor easier to write, understand, debug, and maintain. It makes it easier to define the template for each metric. |

# Plug-In Support Classes 4

There are a number of Endpoint Operations Management plug-in support classes for use when writing your plug-ins.

Use the following classes, as required.

- Auto-Discovery Support Classes on page 51

  These support classes are relevant to Endpoint Operations Management auto-discovery functions.

- Measurement Support Classes on page 64

  There are a number of Endpoint Operations Management measurement support classes for use when writing your plug-ins.

- ProductPlugin Class on page 66

  The `ProductPlugin` class provides the deployment entry point on both the vRealize Operations Manager server and Endpoint Operations Management agent. It defines the object types and the plug-in implementations for measurement, control, and autoinventory.

- ServerResource Class on page 67

  The `ServerResource` class stores resource data for a newly discovered server objects during auto-discovery. `ServerResource` contains the data that is reported for a server object in the auto-inventory report that the Endpoint Operations Management agent sends to vRealize Operations Manager.

- ServiceResource Class on page 72

  The `ServiceResource` class is used to store information for newly discovered services during the autodiscovery methods. This class contains everything that is included in a runtime autoinventory report.

- ConfigResponse Class on page 72

  The `ConfigResponse` class is used throughout Endpoint Operations Management source code to store and transfer configuration data. From an end user perspective 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.

## Auto-Discovery Support Classes

These support classes are relevant to Endpoint Operations Management auto-discovery functions.

- DaemonDetector Class on page 52

  The `DaemonDetector` class auto-discovers a single process and adds the related PTQL query to the resource configuration.

- FileServerDetector Class on page 53

  The `FileServerDetector` interface is used to discover server resources based on a file system scan. This interface is used when a user manually invokes new autodiscovery on platform object level.

- MxServerDetector Class on page 57

  The `MxServerDetector` class auto-discovers JMX servers.

- RegistryServerDetector Class on page 58

  The `RegistryServerDetector` interface is used to discover server objects that are found by scanning the Windows registry.

- ServerDetector Class on page 62

  The `ServerDetector` class is the base implementation for autodiscovery. ServerDetector is an abstract class, so cannot be directly used for auto-discovery. An auto-discovery implementation must inherit ServerDetector.

- SNMPDectector Class on page 64

  You can use the `SNMPDetector` class in XML-only plug-ins that extend the Network Device plug-in, or SNMP-enabled servers, such as Squid.

## DaemonDetector Class

The `DaemonDetector` class auto-discovers a single process and adds the related PTQL query to the resource configuration.

### Class Hierarchy

```
java.lang.Object
     org.hyperic.hq.product.GenericPlugin
          [org.hyperic.hq.product.ServerDetector|ServerDetector]
               org.hyperic.hq.product.DaemonDetector
```

### Resource Properties

This table describes the resource data that you can define in the plug-in descriptor for a plug-in that uses `DaemonDetector`.

| Property | Description | Usage |
|---|---|---|
| `PROC_QUERY` | Initiates a PTQL query to identify the server object. | Mandatory |
| `AUTOINVENTORY_NAME` | Formats the auto-inventory name as defined by the plug-in. | Optional |
| `INSTALLPATH_MATCH` | Returns `true` if the installation path matches the specified substring. | Optional |
| `INSTALLPATH_NOMATCH` | Returns `false` if the installation path matches the specified substring. | Optional |
| `INVENTORY_ID` | The installation path parameter. | Optional |
| `PROC_QUERY` | The default PTQL query to scan for. | |
| `HAS_BUILTIN_SERVICES` | Scans for built-in service objects. The default is `false`. | Optional |
| `VERSION_FILE` | Returns `true` if the specified file exists in the installation path. | Optional |

### Example: Usage

This example defines a new server object that is auto-discovered using a 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>
```

## FileServerDetector Class

The `FileServerDetector` interface is used to discover server resources based on a file system scan. This interface is used when a user manually invokes new autodiscovery on platform object level.

A background scan receives hints from the `<scan>` tag to match the 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 the following methods.

**`getServerResources(ConfigResponse,String):List`**

```
public List getServerResources(ConfigResponse platformConfig, String
path)
throws PluginException;
```

This method is called if the associated autodiscovery implementation is implementing this interface. The method is called with every successfully matched result.

The method must return a list of `ServerResource` objects. See "ServerResource Class," on page 67 for more information.

| | | |
|---|---|---|
| Parameters: | platformConfig | Configuration for the underlying platform object. |
| | path | Matched path. |
| Returns: | | A list of `ServerResource` objects. |
| Exceptions: | | `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

The standalone plug-in invocation differs slightly from how the `FileServerDetector` and `AutoServerDetector` classes are executed compared to a real agent. If a real agent is going to use the `FileServerDetector` class, it executes that before the `AutoServerDetector` class. This standalone invocation executes either one of these, but not both.

To test the `FileServerDetector` interface verify that at least one of the following parameters exist.

| Property Key | Description | Values | Defaults |
|---|---|---|---|
| `fileScan.scanDirs` | The directories to scan. | A list of comma-separated directories. | Windows: `"C:\"`<br>Unix: `"/usr" , "/opt"` |
| `fileScan.excludeDirs` | The directories to exclude from a scan. | A list of comma-separated directories. | Windows: `"\WINNT" , "\TEMP", "\TMP", "\Documents and Settings", "\Recycled"`<br>Linux: `"/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` | The file system types to scan. | One of the following:<br>■ `All disks`<br>■ `Local disks`<br>■ `Network-mounted disks` | `All disks` |

| Property Key | Description | Values | Defaults |
|---|---|---|---|
| fileScan.depth | The depth of directory levels to scan. | 1, or higher.<br>Use –1 to indicate unlimited depth. | 6 |
| fileScan.followSymlinks | Whether symlinks are followed. | true or false | false |

A standalone invocation is implemented using –m discover and –p <server object name> options.

```
# java –jar dcs–tools–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
```

## MxServerDetector Class

The MxServerDetector class auto-discovers JMX servers.

### Class Hierarchy

```
java.lang.Object
     org.hyperic.hq.product.GenericPlugin
          [org.hyperic.hq.product.ServerDetector|ServerDetector]
               org.hyperic.hq.product.MxServerDetector
```

### Resource Properties

This table describes the resource data that you can define in the plug-in descriptor for a plug-in that uses MxServerDetector.

| Property | Description | Usage |
|---|---|---|
| DEFAULT_CONFIG_FILE | The default configuration file to track. | |
| PROC_MAIN_CLASS | | |
| PROC_HOME_PROPERTY | | |
| PROC_HOME_ENV | | |

### Example: Usage

jonas-plugin.xml

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

...
```

## RegistryServerDetector Class

The `RegistryServerDetector` interface is used to discover server objects that are found by scanning the Windows registry.

Scan criteria are specified in a `<scan>` element in the plug-in descriptor.

The following element attributes define where to search in the registry and what registry key to look for.

- `registry` - This attribute specifies a registry path in the Windows registry. Subkeys of the specified path are scanned. You can designate several search roots by appending the path with an asterisk.
- `include` - This attribute specifies the name of a key in the Windows registry. A `<scan>` element can contain multiple `include` attributes. The registry scan does not support wildcards in the registry key name.

The following `<scan>` element results in a scan for a subkey of "SOFTWARE\Microsoft\Internet Explorer.

```
<scan registry="SOFTWARE\Microsoft\Internet Explorer">
  <include name="AppName"/>
</scan>
```

`RegistryServerDetector` is called for each resource in the Windows registry that matches the scan criteria.

You can extend the search to multiple root keys by ending the key name with "". For example, `SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall\MySQL"` only searches for subkeys of `...\Uninstall` that start with `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 the following methods.

**`getServerResources(ConfigResponse,String):List`**

```
public List getServerResources(ConfigResponse platformConfig, String
path, RegistryKey current)
throws PluginException;
```

This method is called if the associated autodiscovery implementation is implementing this interface.

| | | |
|---|---|---|
| Parameters: | platformConfig | Configuration for the underlying platform object. |
| | path | Value of the matched key. |
| | current | Current registry object. |
| Returns: | | A list of `ServerResource` objects. |
| Exceptions: | | `org.hyperic.hq.product.PluginException` |

**`getRegistryScanKeys():List`**

```
public List getRegistryScanKeys();
```

This method returns a list of registry keys to scan. The `ServerDetector` class contains default implementation for this function, which requests keys from the plug-in descriptor file. The example XML content that is used in this document results in the single list member `SOFTWARE\Microsoft\Internet Explorer`.

A user can implement/overwrite this method to return a list of keys directly.

Returns: A 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

A standalone invocation is implemented using `-m discover` and `-p <server object name>` options.

```
# java -jar dcs-tools-pdk.jar
-Dplugins.include=registryscan-example
-Dlog=info
-m discover
-p registryscanserver
```

## ServerDetector Class

The `ServerDetector` class is the base implementation for autodiscovery. ServerDetector is an abstract class, so cannot be directly used for auto-discovery. An auto-discovery implementation must inherit ServerDetector.

### Class Hierarchy

```
java.lang.Object
     org.hyperic.hq.product.GenericPlugin
          org.hyperic.hq.product.ServerDetector
```

### Resource Properties

The table below describes the resource data that you can define in the plug-in descriptor for a plug-in that uses an auto-discovery implementation based on `ServerDetector`.

Resource properties that are not user-configurable are defined in `<property>` elements in the descriptor.

| Property | Description | Usage |
|---|---|---|
| `INSTALLPATH` | Overwrites the installation path. | Optional |
| `INSTALLPATH_MATCH` | See *Using Extra Filters* below. | Optional |
| `INSTALLPATH_NOMATCH` | See *Using Extra Filters* below. | Optional |
| `VERSION_FILE` | See *Using Extra Filters* below. | Optional |
| `INVENTORY_ID` | Overwrites the autoinventory ID (AIID). | Optional |
| `AUTOINVENTORY_NAME` | Formats the discovered object name. | Optional |

### Using Extra Filters

You can use the additional filters `INSTALLPATH_MATCH`, `INSTALLPATH_NOMATCH` and `VERSION_FILE` to filter discovered resources based on the discovered installation path.

The filters are used in the following order.

1 If `VERSION_FILE` is not detected, the resource is skipped.

2 If `INSTALLPATH_MATCH` is not detected from the installation path, the resource is skipped.

3 If `INSTALLPATH_NOMATCH` is detected from installation path, the resource is skipped.

### Using INSTALLPATH

Every server type object must have a value for the installation path property. When you create a server object manually from the user interface, this property is required.

For server types that are auto-discovered, the installation path is resolved automatically. It is usually either the server home directory or the process working directory. You can use `INSTALLPATH` to overwrite a resource's discovered installation path.

## Using AUTOINVENTORY_NAME

You can overwrite a discovered object name by defining a new qualifier. The format of this name is a single string containing variables (`%variable1%`) that map to configuration options. Three types of properties are passed to formatting functions as `ConfigResponse` objects, the parent resource, the resource itself, and custom resource properties.

**Caution** The `AUTOINVENTORY_NAME` property is used only if the auto-discovery implementation calls the appropriate formatting functions.

## Using INVENTORY_ID

The `INVENTORY_ID` property, sometimes referred to as the auto-inventory ID, is used to identify unique objects within discovered object types. The vRealize Operations Manager server verifies whether an object in an auto-discovery report is already in the inventory by checking to see if an object with that `INVENTORY_ID` already exists.

## Implementing Methods

This interface implements the following methods.

**setDescription(String):void**

```
protected void setDescription(String description)
```

This method sets the server description. It allows you to set the description outside of the `ServerResource` object. This enables you to update the server description while discovering new services, However some rules apply. If `discoverServers()` discovers something or `discoverServices()` does not discover anything, this field is ignored.

| Parameters: | description | The server description. |
|---|---|---|

**setCustomProperties(ConfigResponse):void**

```
protected void setCustomProperties(ConfigResponse cprops)
```

This method sets the custom properties for the server. It allows you to set custom properties outside of the `ServerResource` object. This enables you to update server custom properties while discovering new service objects. Some rules apply. If `discoverServers()` discovers something, or `discoverServices()` does not discover anything, this field is ignored.

| Parameters: | cprops | Server custom properties |
|---|---|---|

**discoverServers(ConfigResponse):List**

```
protected List discoverServers(ConfigResponse config)
```

This is a runtime method for discovering new servers. Override this method to discover servers for the server object of the plug-in instance. Most plug-ins override `discoverServices()`, rather than `{{discoverServers()}`.

`discoverServers()}` is typically used in the case in which a plug-in interface, `FileServerDetector` or `AutoServerDetector`, finds an administration server object, then `discoverServers()` discovers managed server nodes. Examples of this usage are found in vRealize Operations Manager WebLogic, WebSphere, and iPlanet plug-ins.

This method returns `NULL` if it is not overwritten.

| Parameters: | Parent configuration |
|---|---|
| Returns: | A list of `ServerResource` objects. |

**discoverServices(ConfigResponse):List**

```
protected List discoverServices(ConfigResponse config)
```

This runtime method discovers new services. Override this method to discover service objects for the server object of the plug-in instance.

This method returns `NULL` if not overwritten.

| | |
|---|---|
| Parameters: | Parent configuration |
| Returns: | A list of `ServerResource` objects. |

**createServerResource(String):ServerResource**

```
protected ServerResource createServerResource(String installpath)
```

This is a helper method to initialize a `ServerResource` with default values.

| | | |
|---|---|---|
| Parameters: | installpath | The object installation path. |

## SNMPDectector Class

You can use the `SNMPDetector` class in XML-only plug-ins that extend the Network Device plug-in, or SNMP-enabled servers, such as Squid.

### Class Hierarchy

```
java.lang.Object
     org.hyperic.hq.product.GenericPlugin
          [org.hyperic.hq.product.ServerDetector|ServerDetector]
               org.hyperic.hq.product.DaemonDetector
                    org.hyperic.hq.product.SNMPDetector
```

### Resource Properties

This table describes the resource data that you can define in the plug-in descriptor for a plug-in that uses `SNMPDetector`.

| Property | Usage |
|---|---|
| `SNMP_INDEX_NAME` | foo |
| `SNMP_DESCRIPTION` | foo |

### Example: Usage

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

# Measurement Support Classes

There are a number of Endpoint Operations Management measurement support classes for use when writing your plug-ins.

Use the following classes, as required.

- MeasurementPlugin Class on page 65

  The `MeasurementPlugin` class is a base implementation for measurement operations.

- SNMPMeasurementPlugin Class on page 66

  Use the `SNMPMeasurementPlugin` class to collect metrics from SNMP devices.

- Win32MeasurementPlugin Class on page 66

  Use the `Win32MeasurementPlugin` class to collect metrics from Windows service objects.

## MeasurementPlugin Class

The `MeasurementPlugin` class is a base implementation for measurement operations.

### Class Hierarchy

```
java.lang.Object
     org.hyperic.hq.product.GenericPlugin
          org.hyperic.hq.product.MeasurementPlugin
```

### Implementing Methods

This class implements the following methods.

**`init`**

`init(PluginManager):void`

`public void init(PluginManager manager)`

**Table 4-1.**

| Parameter | Description |
|---|---|
| `manager` | The plug-in manager. |

**`getManager`**

`getManager():MeasurementPluginManager`

`protected MeasurementPluginManager getManager()`

**`getMeasurementProperties`**

`getMeasurementProperties():Map`

`protected Map getMeasurementProperties()`

**`getMeasurements`**

`getMeasurements(TypeInfo):MeasurementInfo[]`

`public MeasurementInfo getMeasurements(TypeInfo info)`

**`getPlatformHelpProperties`**

`getPlatformHelpProperties():String[][]`

`protected String getPlatformHelpProperties()`

**`getPluginXMLHelp`**

`getPluginXMLHelp(TypeInfo, String, Map):String`

`protected String getPluginXMLHelp(TypeInfo info, String name, Map props)`

**`getHelp`**

`getHelp(TypeInfo, Map):String`

`public String getHelp(TypeInfo info, Map props)`

**`getValue`**

`getValue(Metric):MetricValue`

`public MetricValue getValue(Metric metric)`

**`getNewCollector`**

`getNewCollector():Collector`

`public Collector getNewCollector()`

**`getCollectorProperties`**

`getCollectorProperties(Metric):Properties`

`public Properties getCollectorProperties(Metric metric)`

| | |
|---|---|
| **translate** | translate(String, ConfigResponse):String |
| | public String translate(String template, ConfigResponse config) |
| **getConfigSchema** | getConfigSchema(TypeInfo, ConfigResponse):ConfigSchema |
| | public ConfigSchema getConfigSchema(TypeInfo info, ConfigResponse config) |

### SNMPMeasurementPlugin Class

Use the `SNMPMeasurementPlugin` class to collect metrics from SNMP devices.

#### Property Usage

| Property | Usage |
|---|---|
| `MIBS` | Specifies the MIB files that are necessary to run the plug-in. |

#### Example: Usage

```
<plugin>
  <property name="MIBS"
            value="/etc/squid/mib.txt"/>
...
<plugin type="measurement"
            class="org.hyperic.hq.product.SNMPMeasurementPlugin"/>
...
</plugin>
```

### Win32MeasurementPlugin Class

Use the `Win32MeasurementPlugin` class to collect metrics from Windows service objects.

#### Example: Usage

```
<plugin>

<filter name="store" value="win32:Object=MSExchangeIS"/>
...
<plugin type="measurement"
            class="org.hyperic.hq.product.Win32MeasurementPlugin"/>
...
</plugin>
```

## ProductPlugin Class

The `ProductPlugin` class provides the deployment entry point on both the vRealize Operations Manager server and Endpoint Operations Management agent. It defines the object types and the plug-in implementations for measurement, control, and autoinventory.

Most `ProductPlugin` classes are implemented using the plug-in XML descriptor. However, in order to dynamically generate the classpath, plug-ins can override `ProductPlugin`. For example, the JBoss plug-in uses SIGAR to find the installation path of a JBoss server running on the machine, which it uses to set the classpath.

# ServerResource Class

The `ServerResource` class stores resource data for a newly discovered server objects during auto-discovery. `ServerResource` contains the data that is reported for a server object in the auto-inventory report that the Endpoint Operations Management agent sends to vRealize Operations Manager.

This class stores the following information.

- `resource` This represents the object itself. Most `ServerResource` methods modify a modifying resource. The default constructor creates an empty resource object.
- `fqdn` The fully qualified domain name for an object. `fqdn` is not used unless the object is on a different platform than the Endpoint Operations Management agent that manages it.
- `productConfig` Contains the product configuration properties for an object.
- `metricConfig` Contains the metric configuration properties for an object.
- `controlConfig` Contains the control configuration properties for an object.
- `cprops` The object custom properties.

## Class Hierarchy

```
java.lang.Object
     org.hyperic.hq.product.ServerResource
```

## Implementing Methods

This class implements the following methods.

**`setInstallPath`**

`setInstallPath(String):void`

```
public void setInstallPath(String name)
```

This method sets the resource installation path.

| Parameter | Description |
|---|---|
| `name` | The path to the installation directory. |

**`setPlatformFqdn`**

`setPlatformFqdn(String):void`

```
public void setPlatformFqdn(String name)
```

This method sets the object's fully qualified domain name. This attribute should be set only if the discovered server object runs on a different platform object that on which the Endpoint Operations Management agent that performed the auto-discover runs. For example, the Endpoint Operations Management agent that manages a WebLogic Server cluster runs on the platform on which the administration server runs, and discovers the managed servers that are running on other platforms. Note that If you set this attribute, the platform object that you specify must exist in inventory.

| Parameter | Description |
|---|---|
| `name` | The name of the FQDN. |

**getPlatformFqdn**

`getPlatformFqdn():String`

`public String getPlatformFqdn()`

This method the object FQDN. It returns `NULL` if the field has not been specified.

**addService**

`addService(ServiceResource):void`

`public void addService(ServiceResource service)`

This method adds a new service resource to this server.

| Parameter | Description |
|---|---|
| `service` | The new service resource to add to this server. |

**addServiceType**

`addServiceType(ServiceType):void`

`public void addServiceType(ServiceType serviceType)`

This method adds a new service type to this server.

| Parameter | Description |
|---|---|
| `serviceType` | The new service type to add to this server. |

**setIdentifier**

`setIdentifier(String):void`

`public void setIdentifier(String name)`

This method sets the autoinventory identifier (AIID) for this resource.

| Parameter | Description |
|---|---|
| `name` | The autoinventory identifier. |

**getIdentifier**

`getIdentifier():String`

`public String getIdentifier()`

This method returns the resource autoinventory identifier.

**setType**

`setType(String):void`

`public void setIdentifier(String name)`

This method sets the resource type for the server as defined in the plug-in descriptor. Pass this method the name of the resource type as defined in the plug-in descriptor. For example, if the plug-in descriptor specifies `<server name="My Server">` set the resource type to `My Server`.

If the `<server>` element defines the version attribute, append the value of version to the value of the name attribute to create the resource type name. For example, if the server is defined as `<server name="My Server" version="1.x">`, set Type to `My Server 1.x`.

| Parameter | Description |
|---|---|
| `name` | The resource type as a string. |

`setType(GenericPlugin):void`

`public void setType(GenericPlugin plugin)`

This method derives the resource type to set for the server from the implementing auto-discovery plug-in, as opposed to the plug-in descriptor.

| Parameter | Description |
|---|---|
| plugin | The plug-in that is handling the discovery operation. |

**getType**

```
getType():String
```

```
public String getType()
```

This method returns the current resource type name.

**setName**

```
setName(String):void
```

```
public void setName(String name)
```

This method sets the name of this resource..

| Parameter | Description |
|---|---|
| name | The name of the resource. |

**setDescription**

```
setDescription(String):void
```

```
public void setDescription(String description)
```

This method sets the description of this resource.

| Parameter | Description |
|---|---|
| description | The description of the resource. |

**getDescription**

```
getDescription():String
```

```
public String getDescription()
```

This method returns the description of the resource.

**setProductConfig**

```
setProductConfig(ConfigResponse):void
```

```
public void setProductConfig(ConfigResponse config)
```

This method sets the shared configuration properties for the resource. The configuration is passed as a `ConfigResponse` object.

| Parameter | Description |
|---|---|
| config | The resource shared configuration. |

```
setProductConfig(Map):void
```

```
public void setProductConfig(Map config)
```

This method sets the shared configuration properties for the resource. The configuration is passed as a Map object. Internally, `ConfigResponse` uses Map to store its keys and values.

| Parameter | Description |
|---|---|
| config | The map of the resource configuration. |

```
setProductConfig():void
```

```
public void setProductConfig()
```

This method sets and initializes an empty product config.

**getProductConfig**

`getProductConfig():ConfigResponse`

```
public ConfigResponse getProductConfig()
```

This method returns the shared configuration properties for the resource.

**setMeasurementConfig**

`setMeasurementConfig(ConfigResponse):void`

```
public void setMeasurementConfig(ConfigResponse config)
```

This method sets the monitoring configuration properties for the resource. The configuration is passed as a `ConfigResponse` object.

| Parameter | Description |
|---|---|
| `description` | The description of the resource. |

`setMeasurementConfig(Map):void`

```
public void setMeasurementConfig(Map config)
```

This function sets the monitoring configuration properties for the resource. The configuration is passed as a Map object. Internally, `ConfigResponse` uses Map to store its keys and values.

| Parameter | Description |
|---|---|
| `config` | The map of the resource measurement configuration. |

`setMeasurementConfig():void`

```
public void setMeasurementConfig()
```

This function sets and initializes an empty measurement configuration.

`setMeasurementConfig(ConfigResponse, int, boolean):void`

```
public void setMeasurementConfig(ConfigResponse config,
                                 int logTrackLevel,
                                 boolean enableConfigTrack)
```

This function sets the monitoring configuration properties for the resource. The configuration is passed as a Map object. Internally, `ConfigResponse` uses Map to store its keys and values.

This function can be used to enable log and configuration tracking at the same time. `LogTrackPlugin` defines the following log levels.

```
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;
```

| Parameter | Description |
|---|---|
| `config` | The resource measurement configuration. |
| `logTrackLevel` | The log tracking level in internal type of int. |
| `enableConfigTrack` | Enables config tracking if `TRUE` otherwise, use `FALSE`. |

**getMeasurementConfig**

`getMeasurementConfig():ConfigResponse`

`public ConfigResponse getMeasurementConfig()`

This method returns the monitoring configuration properties for the resource.

**setControlConfig**

`setControlConfig(ConfigResponse):void`

`public void setControlConfig(ConfigResponse config)`

This method sets the control configuration properties for the resource. The configuration is passed as a `ConfigResponse` object.

| Parameter | Description |
|---|---|
| `config` | The resource control configuration. |

`setControlConfig(Map):void`

`public void setControlConfig(Map config)`

This function sets the control configuration properties for the resource. The configuration is passed as a Map object. Internally, `ConfigResponse` uses Map to store its keys and values.

| Parameter | Description |
|---|---|
| `config` | The map of the resource control configuration. |

`setControlConfig():void`

`public void setControlConfig()`

This function sets and initializes an empty control configuration.

**getControlConfig**

`getControlConfig():ConfigResponse`

`public ConfigResponse getControlConfig()`

This function returns the resource control configuration.

**setCustomProperties**

`setCustomProperties(ConfigResponse):void`

`public void setCustomProperties(ConfigResponse config)`

This method sets custom properties for the resource. These are the resource attributes that are defined using the `<property>` elements in the plug-in descriptor. The configuration is passed as a `ConfigResponse` object.

| Parameter | Description |
|---|---|
| `config` | The resource custom properties. |

`setCustomProperties(Map):void`

`public void setCustomProperties(Map props`

This function sets custom properties for the resource. These are the resource attributes that are defined using the `<property>` elements in the plug-in descriptor. The properties are passed using Map object. Internally, `ConfigResponse` uses Map to store its keys and values.

| Parameter | Description |
|---|---|
| config | The resource custom properties. |

**getCustomProperties**

```
getCustomProperties():ConfigResponse
```

```
public ConfigResponse getCustomProperties()
```

This method returns custom properties for the resource. These are the resource attributes that are defined using the <property> elements in the plug-in descriptor.

# ServiceResource Class

The ServiceResource class is used to store information for newly discovered services during the autodiscovery methods. This class contains everything that is included in a runtime autoinventory report.

## Class Hierarchy

```
java.lang.Object
    org.hyperic.hq.product.ServiceResource
```

## Implementing Methods

This class implements the following method.

**setName**

```
setName(String):void
```

```
public void setName(String name)
```

This method sets the resource name.

| Parameter | Description |
|---|---|
| name | The name of the resource. |

# ConfigResponse Class

The ConfigResponse class is used throughout Endpoint Operations Management source code to store and transfer configuration data. From an end user perspective 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.

## Class Hierarchy

```
java.lang.Object
    org.hyperic.util.config.ConfigResponse
```

## Implementing Methods

This class implements the following methods.

**setValue**

```
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. |
| Exceptions: | `InvalidOptionException` | Returned when the `ConfigResponse` does not support the specified option. |
| | `nvalidOptionValueException` | Returned when the supplied value is not valid for the specified option. |

## 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;
}
```

# Index