

HOW TO WRITE A PLUGIN FOR JBoss ON, JOPR AND RHQ

BY HEIKO W. RUPP <heiko.rupp@redhat.com>

PREFACE

This document builds on a series of blog postings I posted at
<http://pilhuhn.blogspot.com/search/label/RHQ>

The blog also features other JBoss ON, Jopr, RHQ, and plug-in
related content that is not contained here.

Please contact me or my colleagues if you have questions or suggestions.

TABLE OF CONTENTS

| | | | |
|----|---|----|--|
| 3 | INTRODUCTION | 24 | THINGS TO CONSIDER WHEN WRITING A PLUG-IN |
| 3 | GENERAL ARCHITECTURE OF RHQ | 24 | DECOMPOSING PLUG-INS |
| 4 | Server services | 25 | USING PROCESS SCANS FOR DISCOVERY |
| 4 | Agent architecture | 25 | Process-scans in the plug-in descriptor |
| 5 | Central functionality: Inventory | 26 | Discovery component revisited |
| 6 | CREATING A PLUG-IN | 26 | A FEW MORE FACETS |
| 6 | WHAT DO WE NEED ? | 26 | ConfigurationFacet |
| 7 | Plug-in descriptor | 26 | OperationFacet |
| 9 | Discovery component | 26 | ContentFacet |
| 9 | Plug-in component | 27 | EVENTS |
| 10 | THE RHQ PROJECT STRUCTURE | 27 | Plug-in Component |
| 10 | Directory layout | 27 | Event Poller |
| 11 | Maven pom | 28 | PLUG-IN COMMUNITY |
| 11 | Plug-in artifacts | 28 | ABOUT THE AUTHOR |
| 15 | READY, STEADY, GO ... | | |
| 17 | What do we have now? | | |
| 18 | ENHANCING THE PLUG-IN | | |
| 19 | Changed plug-in descriptor | | |
| 19 | A word about configuration and properties | | |
| 21 | Change in discovery components | | |
| 22 | Change in plug-in components | | |
| 23 | Building the enhanced plug-in | | |
| 23 | Summary | | |

INTRODUCTION

RHQ Project is the foundation of a powerful open source system management suite. It builds the framework for other management applications like JBoss ON.

Red Hat and Hyperic openly released Project RHQ in February 2008. At JavaOne 2008, we released the first GA version of RHQ together with JBossON 2.0, which is built on top of RHQ.

This paper will explain how to write your own plug-ins for RHQ. As an example, this plug-in will try to reach an HTTP server, see if the base URL is available, and return the status code with the time it took to reach it. We will first write a simple version and enhance it afterwards, as well as explore topics around plug-in development.

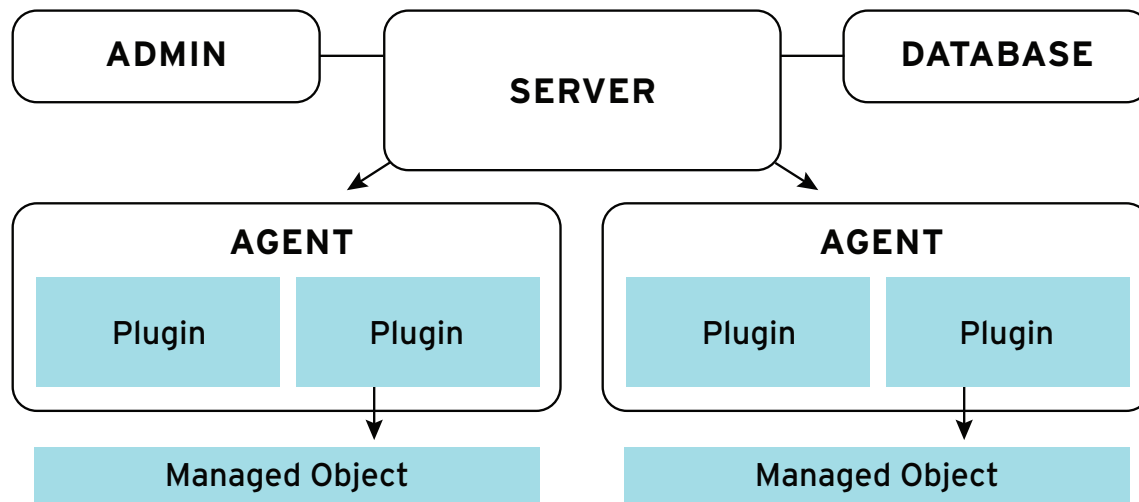
GENERAL ARCHITECTURE OF RHQ

Before we go into detailed plug-in writing, you should first understand the general architecture of RHQ and its plug-in system.

RHQ follows a hub and spoke approach: A central server (or cluster of servers) processes data coming in from agents. It stores the data in a database connected to the server(s). Users and administrators can look at the data and trigger operations through a web-based GUI on the server.

Agents don't have a fixed (as in compile-time) functionality, but they can be extended through plug-ins, which you'll see below. Usually there is one agent running per machine with resources to manage. The RHQ server itself is able to run an agent embedded in the server. This is mostly for demo and test scenarios, but it's also well able to monitor the server machine.

GENERAL ARCHITECTURE OF RHQ



SERVER SERVICES

The server hosts a number of services like:

- A view on the complete inventory
- Processing incoming measurement data
- Triggering alerts to be sent
- Triggering operations on managed resources
- Hosting graphical user interface
- Hosting the user management

Some of those services are reflected in the agent, like inventory syncing, gathering of measurement data, and running operations on a managed resource. In contrast, alert processing or hosting of the GUI is purely on the server.

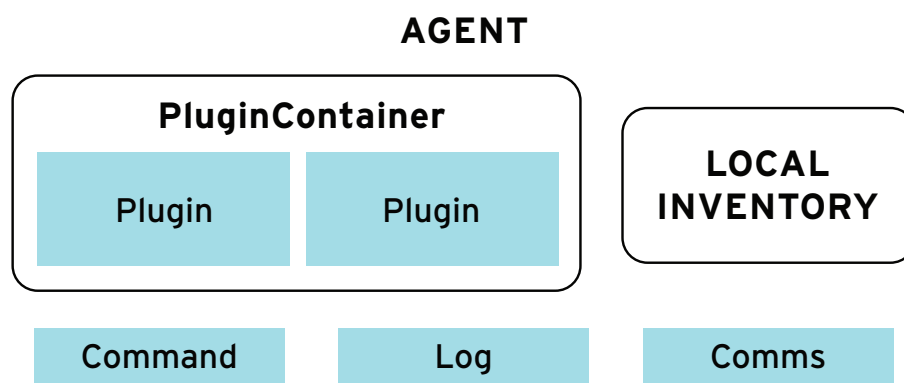
Note that an RHQ server never directly talks to a managed resource. Only agents (or better, their plug-ins) talk to managed resources.

AGENT ARCHITECTURE

The agent is a container that hosts some common functionality, like the communication interface with the server; logging, starting and stopping of plug-ins; reading configuration files; and spooling data in case the server is not reachable. It also handles the command line and interactive command prompt.

Most importantly for these instructions, the agent hosts the *plug-in container*, which hosts the actual plug-ins. When you write a plug-in, you talk to the plug-in container.

AGENT ARCHITECTURE



The agent also hosts a local view of the inventory for the resources it knows.

CENTRAL FUNCTIONALITY: INVENTORY

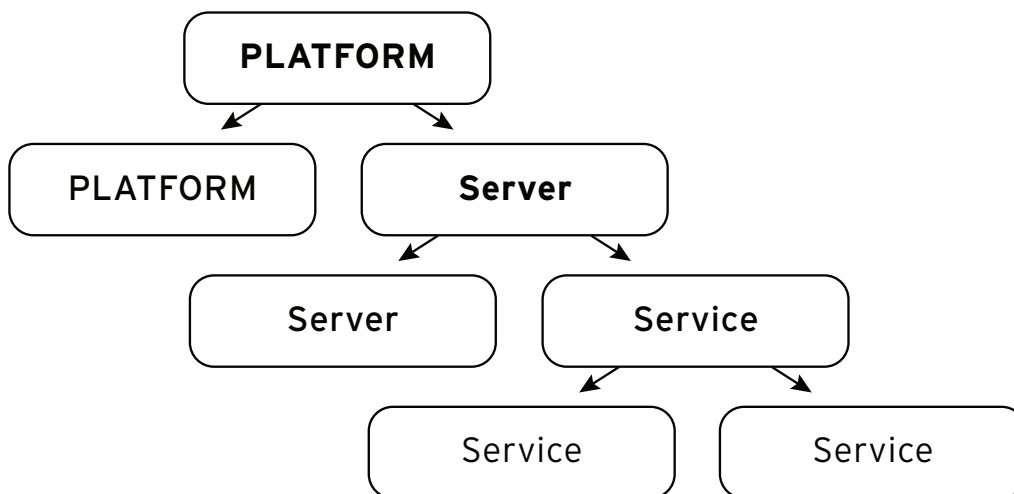
RHQ's central piece of functionality is the inventory. Each resource that you want to manage or monitor must be present in that inventory. RHQ has mechanisms to auto-detect and manually add resources. You'll learn more about that when you learn about implementing plug-ins.

Each `org.rhq.core.domain.resource.Resource` has a certain `org.rhq.core.domain.resource.ResourceCategory`:

- **Platform:** A host things run on
- **Server:** Things like the database server, JbossAS, or the RHQ agent
- **Service:** Fine-grained services offered by a server

The `ResourceCategory` is hierarchic:

RESOURCECATEGORY



A platform hosts servers. A server can host other servers and services. A service can host other services. In theory it is also possible that a platform is hosting other platforms.

An example: you have a Red Hat Linux platform, which hosts the RHQ Agent and JBossAS as a server. The AS itself is hosting a Tomcat server. Both JBossAS and Tomcat are hosting services like JMS or Connectors. The result is a tree of resources with the Linux platform as its root.

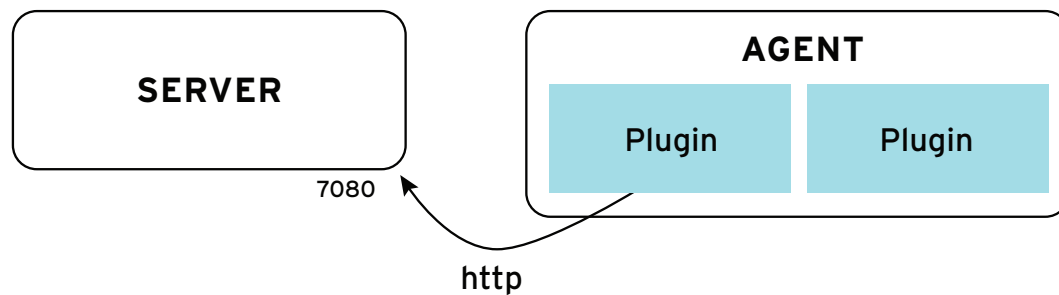
In addition to the category, each resource also is of a certain `org.rhq.core.domain.resource.ResourceType`. For a platform this might be Mac OS X, Red Hat Linux, Debian Linux, etc. The JBossAS and Tomcat would both have the server category, but different resource types.

CREATING A PLUG-IN

Our example plug-in should be able to:

- Connect to an HTTP server
- Issue a GET or HEAD request on the base url (e.g. `http://localhost/`)
- Return the HTTP return code as trait and the time it took as numeric data.

EXAMPLE PLUG-IN



To make things easier for this first implementation, we'll have the agent running on the machine that the RHQ server lives on. We'll get data from the Servers HTTP connector at port 7080 (the default port).

WHAT DO WE NEED?

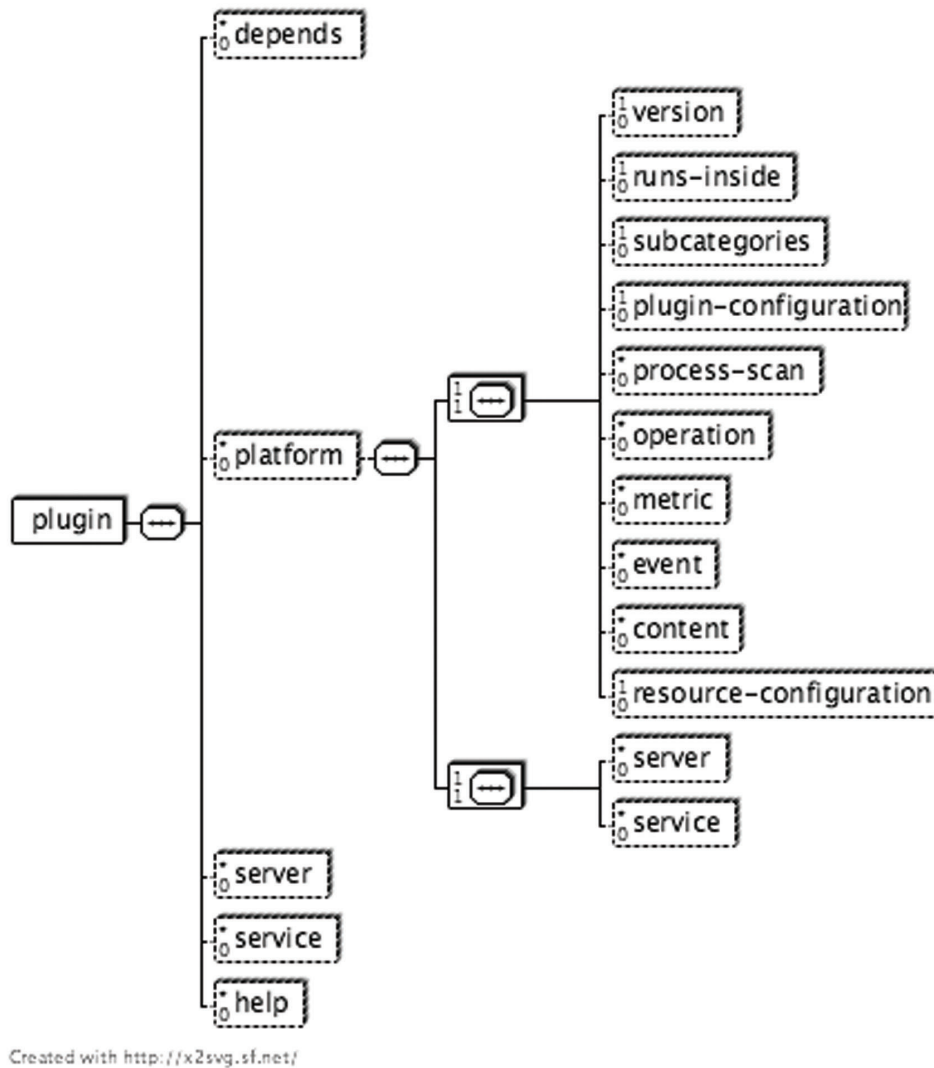
In order to write this plug-in, we need three things:

- A plug-in descriptor, which contains metadata about the plug-in—which metrics should be collected, what operations it supports, etc.
- A discovery component that discovers the actual resource(s) and delivers them to the Inventory.
- A plug-in component, which executes operations and gathers the measurement data.

PLUG-IN DESCRIPTOR

The plug-in descriptor is described by an XML schema that you can find in the Subversion repository.¹ The basic structure is as follows:

PLUG-IN DESCRIPTOR



The descriptor includes a few sections. First you can express dependencies to other plug-ins, which lets you reuse existing plug-ins and is useful when you want to write a plug-in that itself needs the JMX plug-in (also see “Decomposing Plug-ins”).

¹ <http://viewvc.rhq-project.org/cgi-bin/viewvc.cgi/rhq/trunk/modules/core/client-api/src/main/resources/rhq-plugin.xsd?view=log>

The next part is a row of platform/server/service sections. Each of those can have the same (XML-) content as the platform that is shown as an example. As each is a kind of resource type, they are all of the same (XML-) data type (as a platform/server/service).

Example:

```
<service name="CheckHttp">
  <metric property="responseTime"
    description="How long did it take to connect"
    displayType="Summary"
    displayName="Time to get the response"
    units="ms" />
</service>
```

The name of a <service> and the other ResourceTypes (platform, server) must be unique for a plug-in. You can't have two services named "CheckHttp", but you could write separate Tomcat5 and Tomcat6 plug-ins, each with a service named "connector".

One of the subelements is especially interesting for our example—the metric, which is explained in detail below. For all other tags, refer to the XML schema.

The metric element

The metric element is simple, with many attributes and no child tags. You've already seen an example above.

Its attributes are:

- **property:** Name of the metric. Can be obtained in the code via `getName()`.
- **description:** A human-readable description of the metric
- **displayName:** The name that gets displayed.
- **dataType:** Type of metric (numeric / trait /...).
- **units:** The measurement units for numerical data type.
- **displayType:** If `displayType` is set to "summary," the metric will show at the indicator charts and be collected by default.
- **defaultOn:** Whether this metric should be collected by default.
- **measurementType:** Characteristics the numerical values have (trends up, trends down, dynamic). For trends metrics, the system will automatically create additional per minute metrics.

For our example plug-in, we'll use a metric with numerical `dataType` for the response time and a `dataType` of "trait" for the status code. *Traits* are meant to be data values that rarely change, like OS version, IP address of an ethernet interface, or the hostname. RHQ is intelligent enough to store only changed traits to conserve space.

DISCOVERY COMPONENT

The discovery component is called by the InventoryManager in the agent to discover resources. This can be done by a process table scan (e.g. for the Postgres plug-in) or by any other means.

The most important thing to remember is that the discovery component must **return the same unique key each time for the same resource**.

The DiscoveryComponent needs to implement `org.rhq.core.pluginapi.inventory.ResourceDiscoveryComponent`, and you need to implement `discoverResources()`.

The code block that you will usually see in `discoverResources()` looks like this:

```
Set<DiscoveredResourceDetails> result = new
    HashSet<DiscoveredResourceDetails>();
for ( ... ) {
    ...
    DiscoveredResourceDetails detail = new DiscoveredResourceDetails(
        context.getResourceType(),
        uniqueResourceKey,
        resourceName,
        resourceVersion,
        description,
        configuration, // can be null if no configuration
        processInfo); // can be null for no process scan
    result.add(detail);
}
return result;
```

The context passed in gives you a lot of information that you can use to discover the resource and create a `DiscoveredResourceDetails` object per discovered resource. The list of result objects is then returned to the caller.

PLUG-IN COMPONENT

The plug-in component takes over after the discovery has finished. For each of the basic functions in the plugin descriptor, the plug-in component needs to implement an appropriate *facet*:

| DESCRIPTOR ELEMENT | FACET |
|--------------------------|--------------------|
| <metric> | MeasurementFacet |
| <operation> | OperationFacet |
| <resource-configuration> | ConfigurationFacet |
| ... | ... |

Each facet has its own methods to implement. In the case of the `MeasurementFacet`, the method is `getValues(MeasurementReport report, Set metrics)`. Add your results to the report passed in. Data should be gathered for the list of metrics. This can be all of your defined <metric>s at once or only a few of them, depending on the schedules the user configured in the GUI.

THE RHQ PROJECT STRUCTURE

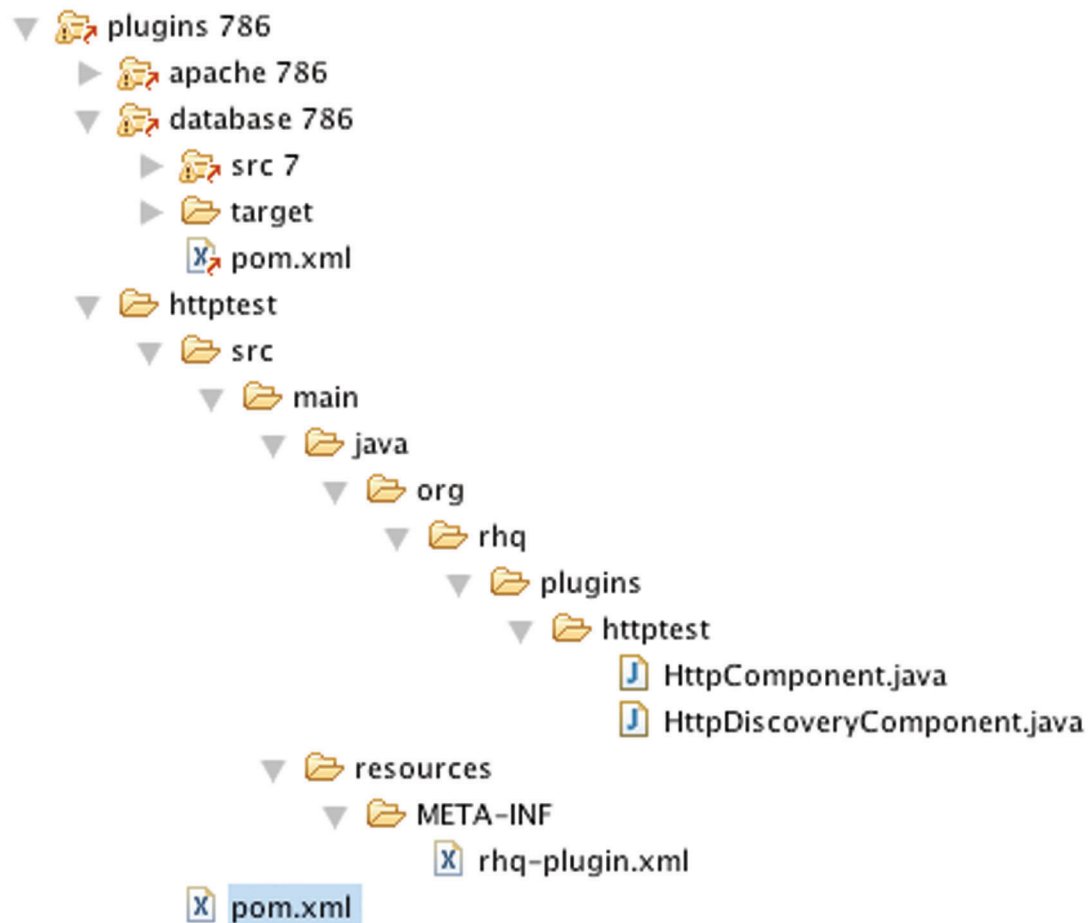
To make things easier, we'll host this plug-in only within the RHQ tree. Check out RHQ from <http://svn.rhq-project.org/repos/rhq>. Build the project as described on the build page of the wiki.² Then you'll be able to add our plug-in to modules/plugins/.

If you don't want to completely check out RHQ, you can use the skeleton-plug-in as described in the wiki.³

DIRECTORY LAYOUT

Create the following directory structure:

DIRECTORY LAYOUT



² <http://support.rhq-project.org/display/RHQ/Building+RHQ>

³ <http://support.rhq-project.org/display/RHQ/Plugins+-+Skeleton+Plugin> This page contains information about how to use the skeleton plugin, as well as a link to a download page.

Add `modules/plugins/httptest/src/main/java` to the build path in your IDE.

The classes within `org.rhq.plugins.httptest` form the plug-in discovery component and plug-in component and will be described below.

MAVEN POM

RHQ is a Mavenized project, thus we need to supply a POM file. The easiest way is to grab another POM, copy it over to the root of the plug-in subtree, and change the artifactId:⁴

```
<groupId>org.rhq</groupId>
<artifactId>rhq-httptest-plugin</artifactId>
<packaging>jar</packaging>
<name>RHQ HttpTest Plugin</name>
<description>A plugin to monitor http servers</description>
```

Please note that this defines the POM only for this subtree-it will not add it to the global project. To do that, you need to add the httptest plug-in to the parent POM at the `modules/plugins/` level:

```
<modules>
  <module>platform</module>
  ...
  <module>postgres</module>
  <module>httptest</module>
</modules>
```

PLUG-IN ARTIFACTS

Now we'll look at the three individual artifacts that make up a plug-in. The directory tree above shows where they are located.

Plug-in discovery component

Start by discovering the server. This is relatively simple and directly follows the description in the previous part.

⁴ Depending on the RHQ-version you are using, you also need to adjust the `version` element within the `<parent>` tag.

```
public class HttpDiscoveryComponent implements ResourceDiscoveryComponent
{
    public Set discoverResources(ResourceDiscoveryContext context) throws
        InvalidPluginConfigurationException, Exception
    {
        Set<DiscoveredResourceDetails> result = new
            HashSet<DiscoveredResourceDetails>();

        String key = „http://localhost:7080/“; // Jon server
        String name = key;
        String description = „Http server at „ + key;
        Configuration configuration = null;
        ResourceType resourceType = context.getResourceType();
        DiscoveredResourceDetails detail = new
            DiscoveredResourceDetails(resourceType, key, name,
                null, description, configuration, null );
        result.add(detail);
        return result;
    }
}
```

Again, it's extremely important that the key stays the same for each discovery performed.

Plug-in component

Next make the plug-in component that does the work:

```
public class HttpComponent implements ResourceComponent, MeasurementFacet {
    URL url;          // remote server url
    long time;        // response time from last collection
    String status;    // Status code from last collection
}
```

For monitoring, we need to implement the MeasurementFacet with the getValues() method. But first we have to implement two of the methods from ResourceComponent. The first returns the availability of the remote server. We return DOWN if the status is null or 500, otherwise, UP.

```
public AvailabilityType getAvailability() {
    if (status == null || status.startsWith("5"))
        return AvailabilityType.DOWN;
    return AvailabilityType.UP;
}
```

Be careful—the discovery won't happen if this method is returning DOWN. So you should provide a valid start value in the start() method from the ResourceComponent:

```
public void start(ResourceContext context) throws
    InvalidPluginConfigurationException, Exception
{
    url = new URL("http://localhost:7080/");
    // Provide an initial status,
    // so getAvailability() returns UP
    status = „200“;
}
```

Note: Analogous to start(), there is also a stop() method that can be used to clean up resources.

This leads us to getValues() from the MeasurementFacet:

```
public void getValues(MeasurementReport report,
    Set<MeasurementScheduleRequest> metrics) throws Exception
{
    getData();
    // Loop over the incoming requests and
    // fill in the requested data
    for (MeasurementScheduleRequest request : metrics) {
        if (request.getName().equals("responseTime")) {
            report.addData(new MeasurementDataNumeric( request,
                new Double(time)));
        }
        else if (request.getName().equals("status")) {
            report.addData(new MeasurementDataTrait (request, status));
        }
    }
}
```

We get data from the remote server, loop over the incoming request to see which metric is wanted, and fill it in. Depending on the type, we need to wrap it into the correct MeasurementData* class.

Finally, there's the implementation of getData(). Open a URL connection, take the time it takes to connect, and get the status code. The following is a simple solution. You may wish to optimize future versions.

```
private void getData()
{
    HttpURLConnection con = null; int code = 0;
    try {
        con = (HttpURLConnection) url.openConnection();
        con.setConnectTimeout(1000);
        long now = System.currentTimeMillis();
        con.connect();
        code = con.getResponseCode();
        long t2 = System.currentTimeMillis();
        time = t2 - now;
    } catch (Exception e) {
        e.printStackTrace();
    }
    if (con != null)
        con.disconnect();
    status = String.valueOf(code);
}
```

Plug-in descriptor

The plug-in descriptor is where everything is glued together. First we start off with some boilerplate code:

```
<?xml version="1.0" encoding="UTF-8" ?>
<plugin name="HttpTest"
    displayName="HttpTest plugin"
    package="org.rhq.plugins.httptest"
    version="2.0"
    description="Monitoring of http servers"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="urn:xmlns:rhq-plugin"
    xmlns:c="urn:xmlns:rhq-configuration">
```

The package attribute predefines the Java package for Java class names that appear later in the descriptor.

```
<server name="HttpServer"
    discovery="HttpDiscoveryComponent"
    class="HttpComponent"
    description="Http Server">
```

The plug-in is defined as a server. Intuition might suggest it was a service, but services can't live on their own. The attribute class denotes the plug-in component, and discovery gives the discovery component. If you have specified the package above, you can use the class name without a prefix.

```

<metric property="responseTime"
        displayName="Response Time"
        measurementType="dynamic"
        units="milliseconds"
        displayType="summary"/>

<metric property="status"
        displayName="Status Code"
        dataType="trait"
        displayType="summary"/>
</server>
</plugin>

```

Next are the two metrics. Again, responseTime is modeled as numerical data, while the status is modeled as trait.

READY, STEADY, GO.

To compile the plug-in, enter `mvn -Pdev install` at the root of the plug-in tree.

The dev mode allows Maven to automatically deploy the plug-in to a server instance as described in the Advanced Build Notes page on the RHQ-Wiki⁵ (<http://support.rhq-project.org/display/RHQ/Advanced+Build+Notes>).

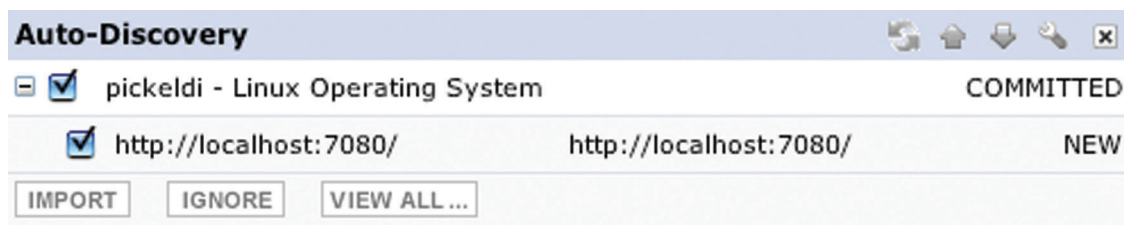
When the server is running or starting up, you will see a line like this in the server log:

```
21:49:31,874 INFO [ProductPluginDeployer] Deploying ON plugin HttpTest (HttpTest plugin)
```

The next step is to make the plug-in available to the agent. Remember that the agent is usually pulling plug-ins from the server when it's starting up. So if you haven't started the agent, there's nothing to do for you. If the agent has started, you can run `plug-ins update` at the command prompt to update them to the latest versions on the server.

Log into the GUI, go to the Autodiscovery portlet, and import the new server into Inventory.

AUTO-DISCOVERY



| Auto-Discovery | | |
|-------------------------------------|-----------------------------------|----------------------------|
| <input checked="" type="checkbox"/> | pickeldi - Linux Operating System | COMMITTED |
| <input checked="" type="checkbox"/> | http://localhost:7080/ | http://localhost:7080/ NEW |

IMPORT IGNORE VIEW ALL ...

⁵ If you did not set the 'dev-container' property in settings.xml, you can also just copy the generated plugin jar from target/ to \$RHQ-SERVER/jbossas/server/default/rhq.ear/rhq-downloads/rhq-plugins.



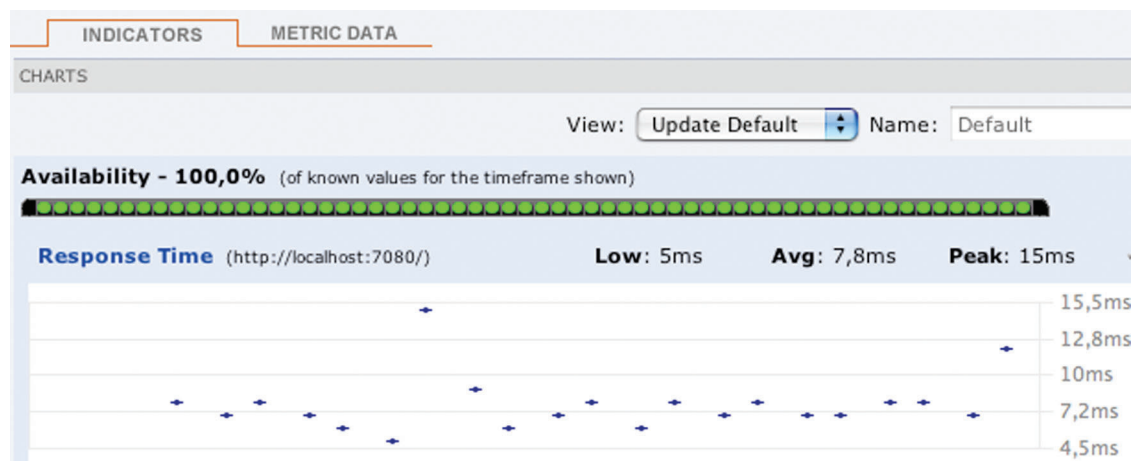
Next choose Servers in the resource browser, and you can see the server has been discovered by our plug-in:

RESOURCE BROWSER

| Platforms (1) Servers (6) Services (219) Compatible Groups (0) Mixed Groups | | | | |
|---|------------|---|---|------------------------|
| View: All Server Types | | | | |
| | Server ▲ 2 | | Server Type ▲ 1 | Description |
| | M | A | snert Embedded JBossWeb Server 2.0.0.GA (0.0.0.0) | Embedded Tomcat Server |
| | M | A | http://localhost:7080/ | HttpServer |

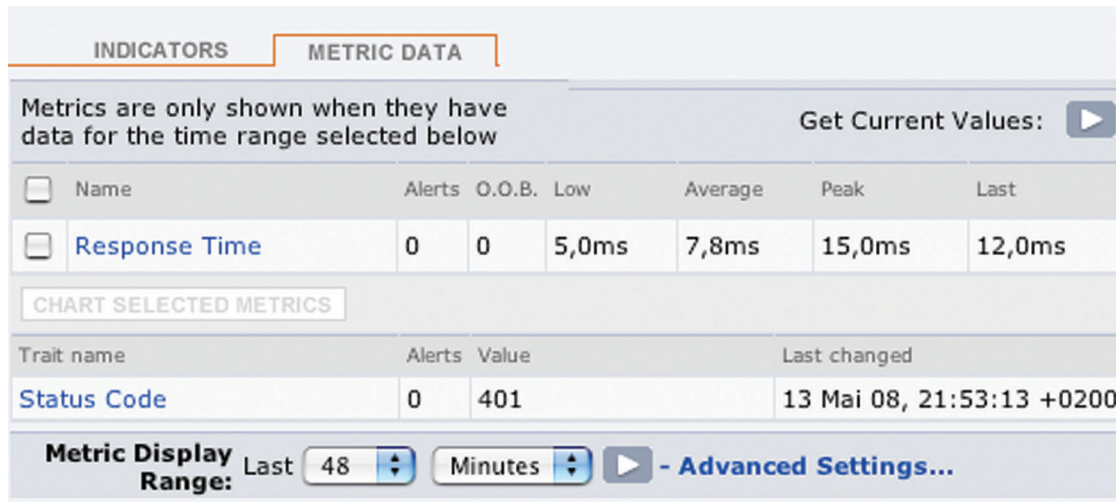
Clicking on the server name or the “M” icon (for “monitor”) leads you to the indicator charts, where you’ll be able to see response time values:

INDICATOR CHARTS



When you click on the Metric Data subtab, you can see the raw data for the server:

METRIC DATA



The screenshot shows the 'METRIC DATA' subtab selected. It displays a table of metrics for 'Response Time' and a section for 'CHART SELECTED METRICS' showing 'Status Code'.

| Name | Alerts | O.O.B. | Low | Average | Peak | Last |
|---------------|--------|--------|-------|---------|--------|--------|
| Response Time | 0 | 0 | 5,0ms | 7,8ms | 15,0ms | 12,0ms |

| Trait name | Alerts | Value | Last changed |
|-------------|--------|-------|---------------------------|
| Status Code | 0 | 401 | 13 Mai 08, 21:53:13 +0200 |

At the bottom, there is a 'Metric Display Range' section with a 'Last' dropdown set to '48', a 'Minutes' dropdown, and a button for '- Advanced Settings...'.

At the top you see the numerical response time data, and in the lower section the server status is trait, as expected.

WHAT DO WE HAVE NOW?

Congratulations, you just wrote your first RHQ plug-in, which can also be used in JBoss ON 2. Writing the plug-in consisted of three parts: discovery, plug-in component and plug-in descriptor. The agent with its plug-in container provides you with all the infrastructure to talk to the server, scheduling metric gathering, scheduling discovery, etc. This means that you can fully concentrate on the business code of your plug-in. RHQ does the rest.

The source code from this article is available as zip archive⁶ that you can unpack in the `modules/plugins/` directory.

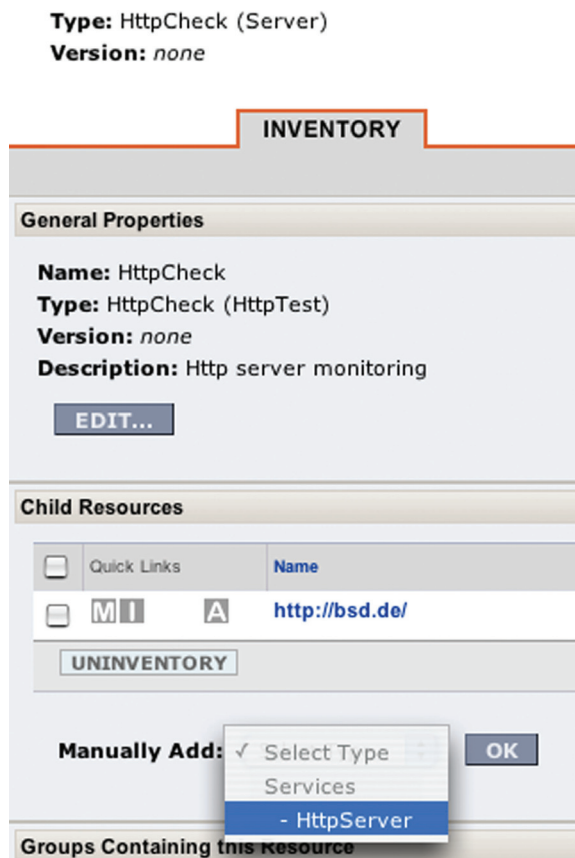
⁶ http://www.pilhuhn.de/hwr/misc/RHQ_httptest.zip

ENHANCING THE PLUG-IN

Our first RHQ plug-in works great, but hardcoding the target URL isn't very elegant. You'll now learn how to make the target URLs configurable from the GUI.

First we need to reshuffle things a little. We need a generic server `HttpCheck` that acts as parent for the individual HTTP servers that we want to monitor. Those will live as services under that server. In the server inventory, we'll add the option to manually add new HTTP servers on the go.

SERVER INVENTORY



As you may have already guessed, most of this is done in the plug-in descriptor. We'll also need some small code changes, but those are mostly to separate the concerns of the various files. Let's start with the changed plug-in descriptor.

CHANGED PLUG-IN DESCRIPTOR

The boilerplate code is the same as before. I changed the name of the server to `HttpCheck`, as having a descriptive name for the parent is nicer in the GUI.

```
<server name="HttpCheck"
  description="Httpserver pinging"
  discovery="HttpDiscoveryComponent"
  class="HttpComponent">
```

Now the interesting part starts:

```
<service name="HttpServer"
  discovery="HttpServiceDiscoveryComponent"
  class="HttpServiceComponent"
  description="One remote Http Server"
  supportsManualAdd="true">
```

Here we introduce a service as child of the above server. It has its own plug-in component and discovery classes (the names of the classes reflect that they belong to this service). They could have gone into the existing classes, but this way makes it more obvious who does what. The attribute `supportsManualAdd` tells RHQ that those `HttpServer` services can be added by the operator in the GUI—just what we want.

```
<plugin-configuration>
  <c:simple-property name="url" type="string" required="true" />
</plugin-configuration>
```

The `plugin-configuration` tells RHQ that this service can be configured with one simple property, the URL of the remote, which is required.

Finally we move the two metrics into the service tag (with the same details as before):

```
  <metric property="responseTime" ...
  <metric property="status" ...
</service>
</server>
```

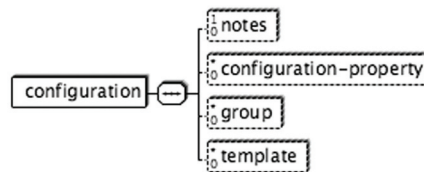
A WORD ABOUT CONFIGURATION AND PROPERTIES

The configuration type presented here can be used in two forms within a plug-in descriptor: `plugin-configuration` and `resource-configuration`.⁷ Check the structure diagram in the plug-in descriptor section to see where they belong.

⁷ The `plugin-configuration` serves to configure the plugin and how to connect to the managed resource. `Resource-configuration` on the other hand is used to configure the managed resource itself or to even create a new instance from scratch.

A configuration can consist of a number of sub-elements, most notably properties that are children of the abstract configurationType. This is described below.

CONFIGURATION



Created with <http://x2svg.sf.net/>

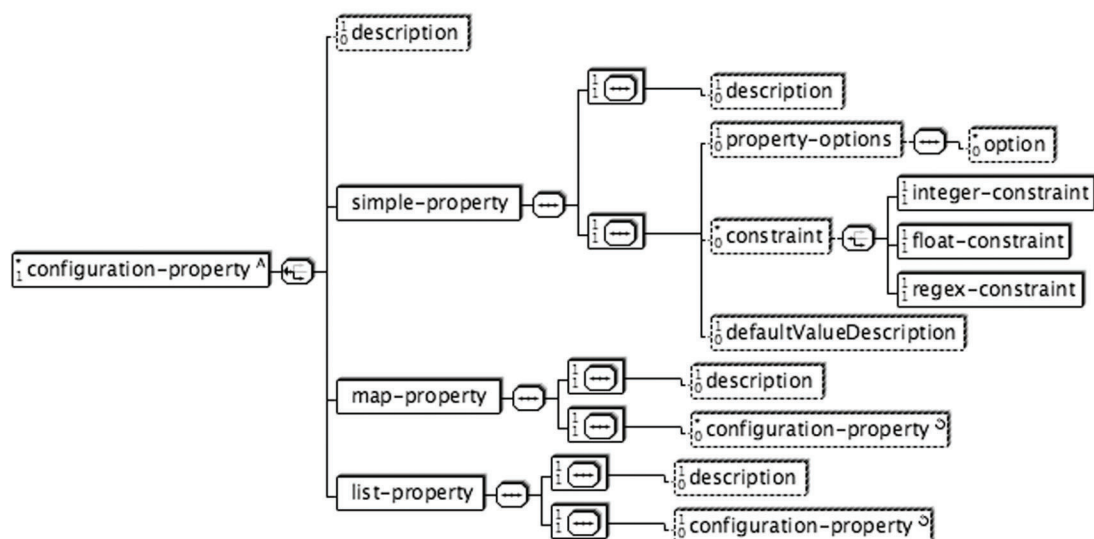
In addition, it's possible to group properties together in the group element. The GUI will show those in their own collapsible section. The group element allows one description element and instances of the abstract configuration-property as children. Templates let you preset some configuration properties so that the user needs to fill in only the information that is needed or that they want to change. The template itself is configuration type.

Properties

Properties allow you to specify individual aspects of a configuration. There are three types of properties:

- **simple-property:** For one key value pair, as shown above.
- **map-property:** For multiple key value pairs, following the `java.util.Map` concept.
- **list-property:** For a list of properties.

STRUCTURAL DIAGRAM



As you can see from the structural diagram, it is possible to nest configuration properties within list-property and map-property elements to compose more complex configurations.

If we would want to allow our Services to add multiple remote servers with properties of ,host', ,port', ,protocol' it *could* look like this:

```
<plugin-configuration>
  <c:list-property name="Servers">
    <c:map-property name="OneServer">
      <c:simple-property name="host"/>
      <c:simple-property name="port">
        <c:integer-constraint
          minimum="0"
          maximum="65535"/>
      </c:simple-property>
      <c:simple-property name="protocol">
        <c:property-options>
          <c:option value="http" default="true"/>
          <c:option value="https"/>
        </c:property-options>
      </c:simple-property>
    </c:map-property>
  </c:list-property>
</plugin-configuration>
```

This example also shows a few more possibilities we have. The port has a constraint so that the GUI can validate the input being between 0 and 216-1. For the protocol, we offer the user a dropdown list/radio buttons to choose the protocol from. It defaults to "http," as indicated on the option element.

CHANGE IN DISCOVERY COMPONENTS

These changes are, as already indicated, more or less just for clarity reasons and to clearly separate out the concerns of each component.

Server level: HttpDiscoveryComponent

The HttpDiscoveryComponent from above only got some minor adjustments to cater for the change in naming, so I am not showing it here – have a look at the provided sources archive for details.

Service level: `HttpServiceDiscoveryComponent`

The `HttpServiceDiscoveryComponent` is more interesting, as we no longer have the hard coded keys, but we get the URL passed in from the GUI when the user is adding a new one.

```
public class HttpServiceDiscoveryComponent implements
    ResourceDiscoveryComponent<HttpServiceComponent>
{
    public Set<DiscoveredResourceDetails> discoverResources
        (ResourceDiscoveryContext<HttpServiceComponent> context) throws
            InvalidPluginConfigurationException, Exception
    {
        Set<DiscoveredResourceDetails> result = new
            HashSet<DiscoveredResourceDetails>();
        ResourceType resourceType = context.getResourceType();
```

This basically the same code that we already know. The interesting part starts now:

```
List<Configuration> childConfigs = context.getPluginConfigurations();
for (Configuration childConfig : childConfigs) {
    String key = childConfig.getSimpleValue("url", null);
    if (key == null)
        throw new InvalidPluginConfigurationException( „No URL provided“);
```

We get a list of plug-in configurations passed from the context through which we loop to determine the passed parameters. As we have only one – the url – this is simple.

If there is no url provided we complain (actually that should never happen, as we marked the property as required in the plug-in descriptor above).

```
        String name = key;
        String description = „Http server at „ + key;
        DiscoveredResourceDetails detail = new
            DiscoveredResourceDetails( resourceType, key, name,
                null, description, childConfig, null );
        result.add(detail);
    }
    return result;
}
```

The remainder of the is the same as we know it already from above.

CHANGE IN PLUG-IN COMPONENTS

The change in plug-in components is simple – the old `HttpComponent` gets renamed to `HttpServiceComponent`, and we have a new pseudo `HttpComponent` on server level.

Server level: `HttpComponent`

`HttpComponent` is a dummy implementation, as it provides only placeholder methods from the `ResourceComponent` interface.

```
public AvailabilityType getAvailability() {  
    return AvailabilityType.UP;  
}
```

Set the availability to always be UP so that the component can successfully start. Leave the other two methods as empty implementations.

Service level: `HttpServiceComponent`

There's only one change from the old `HttpComponent`:

```
public void start(ResourceContext context) throws  
    InvalidPluginConfigurationException, Exception  
{  
    url = new URL(context.getResourceKey()); // Provide an initial status, so  
                                           // getAvailability() returns up  
    status = „200“;  
}
```

We're now setting the URL when the passed `ResourceContext` is starting to read the component.

BUILDING THE ENHANCED PLUG-IN

The updated plug-in can be built as shown in the previous part by calling `mvn -Pdev install` in the root of plug-in source tree.

SUMMARY

You've just seen how easy it is to pass plug-in configuration parameters from the GUI to the plug-in by expressing the parameters in the plug-in descriptor. Our plug-in can now have an arbitrary number of child services that each monitor a different remote HTTP server. The remaining changes are just a few more lines of XML and a little bit more Java code.

The sources are available as zip archive.⁸ If you installed the previous source, overwrite it to install these.

⁸ http://www.pilhuhn.de/hwr/misc/RHQ_httptest2.zip

THINGS TO CONSIDER WHEN WRITING A PLUG-IN

The method `getValues()` from the `MeasurementFacet` is called from the plug-in container in intervals given by the user. This is usually measured in minutes, but it could be shorter. As the container tries to call `getValues()` for all metrics of a resource (that are due for metric collection) at once, it means that taking a single metric can take no more than $(\text{interval}/\text{number of resources})$ time. So make your metrics gathering fast. If directly taking a metric takes a long time (e.g. because a connection to a resource needs to be established first), consider starting its own measurement thread to put the data in local storage. Then have `getValues()` read out from the local storage.

Also consider resource type grouping. When you plan to have multiple items in one category (e.g. multiple HTTP servers to check), then it's good to have a single parent for all of those, like the `HttpComponent` above. This is also a good practice if you plan to add new child resources, as the create code needs to be in the parent (`HttpComponent` for `HttpServices`).

DECOMPOSING PLUG-INS

When you try to manage larger systems like the JBoss Application Server with all its subsystems (Cache, Transactions, JbossWeb, etc.), your plug-in might get relatively large to support everything. You can decompose a large plug-in into smaller ones that together allow you to manage the larger system.

This decomposition not only allows you to more easily distribute the development load, but also enables re-use of the parts that have been broken out of the big chunk. The price you pay is relatively small and consists mostly of some additional directories and a Maven `pom.xml` file.

Use `<depends>` and `<runs-inside>` tags in your plug-in descriptor for the new plug-in:

```
<plugin name="JBossCache" ... >
  <depends plugin="JMX" />
  <depends plugin="JBossAS" useClasses="true"/>
</plugin>
```

We need the JMX plug-in and the JBossAS plug-in (part of Jopr/JBoss ON) to be deployed before the plug-in can start. The attribute `useClasses` means that the classloader of our plug-in gets access to the classes of the other plug-in (JBossAS here). That means we can use those classes too.

```
<service name="JBoss Cache" ...>
```

A service can't just "hang in the air." It needs another server or service as a container. That's where `runs-inside` comes into play:

```
  <runs-inside>
    <parent-resource-type name="JBossAS Server" plugin="JBossAS"/>
  </runs-inside>
```

So our plug-in service "JBoss Cache" will be contained in resources of type "JBossAS Server" that come from the JBossAS plug-in, which we declared in the `depends` element earlier.

Apart from this little magic in the plug-in descriptor, there's no more additional work to do.

USING PROCESS SCANS FOR DISCOVERY

When you want to discover resources, they often aren't virtual like the remote HTTP servers in our examples, but processes on the local machine. The RHQ agent offers through its SIGAR library the option to query the process table in order to detect those resources. That involves the plug-in descriptor, so let's look at it first before going to the discovery component.

PROCESS-SCANS IN THE PLUG-IN DESCRIPTOR

As you have seen in the structural diagram of the plug-in descriptor, each platform, server, and service can have `<process-scan>` elements. The element itself is empty, but it has two required attributes: *name* and *query*. *Name* names the specific scan method. *Query* is a string written in PIQL (Process Info Query Language), which is documented in the JavaDoc to its class.⁹

Example query 1: Find a JBossAS

We want to query for a process whose name starts with "java" and has an argument of `org.jboss.Main` – a JBoss server.

```
Process|basename|match=^java.* ,arg|org.jboss.Main|match=.*
```

The matching entry from `ps` is:

```
hrupp      2035    0.0 -1.5   724712  30616  p7  S+    9:49PM   0:01.61 java
-Dprogram.name=run.sh -Xms128m -Xmx512m -Dsun.rmi.dgc.client.gcInterval=3600000
-Dsun.rmi.dgc.server.gcInterval=3600000 -Djboss.platform.mbeanserver -Djava.
endorsed.dirs=/devel/jboss-4.0.5.GA/lib/endorsed -classpath /devel/jboss-
4.0.5.GA/bin/run.jar:/lib/tools.jar org.jboss.Main -c minimal
```

Example query 2: Find a process by its pid

Here the program ID is stored in a file in a well-known place:

```
process|pidfile|match=/etc/product/lock.pid
```

PIQL will take the pid from `/etc/product/lock.pid` and search for a process with that ID.

⁹ <http://viewvc.rhq-project.org/cgi-bin/viewvc.cgi/rhq/trunk/modules/core/native-system/src/main/java/org/rhq/core/system/pquery/ProcessInfoQuery.java?revision=7&view=markup>

DISCOVERY COMPONENT REVISITED

Now that you've seen what you can do with `<process-scan>` in the plug-in descriptor, let's see how we can process that info.

```
List<ProcessScanResult> autoDiscoveryResults =
    context.getAutoDiscoveredProcesses();
for (ProcessScanResult result : autoDiscoveryResults) {
    ProcessInfo procInfo = result.getProcessInfo();
    ....
    // as before
    DiscoveredResourceDetails detail =
        new DiscoveredResourceDetails( resourceType, key, name, null,
            description, childConfig, procInfo );
    result.add(detail);
}
```

Obtain the list of resources discovered by process scan (auto-discovered, as opposed to a manual add), and create the `DiscoveredResourceDetails` as before. You can use `ProcessInfo` to get more information about the process or to decide not to include it in the list of auto-discovered resources. The PIQL query would have looked for processes where the name starts with "post." That includes *postgres* and *postmaster*. Here you could still filter to get the ones you really want.

A FEW MORE FACETS

You've learned about `MeasurementFacet`. Below are a few of the other kinds of facets so that you can get an idea of what plug-ins are capable of doing.

CONFIGURATIONFACET

This facet indicates that the plug-in is able to read and write the configuration of a managed resource. It goes hand in hand with `<resource-configuration>` in the plug-in descriptor. The code to create a new managed resource from scratch needs to be on the parent resource, so it's a good idea to write plug-ins that use the `ConfigurationFacet` in a way that they have a parent resource for the subsystem and children for individual resources. You can find an example for this in the JBossAS plug-in when looking at the JBossMessaging subsystem and the individual JMS destinations.

OPERATIONFACET

An operation allows you to invoke functionality on the managed resource. This could be a restart operation or whatever you want to invoke on a target. Operations are described in `<operation>` elements in the plug-in descriptor. They can have argument and return values.

CONTENTFACET

This facet lets you upload content like files or archives into the managed resource. That way it's possible to centrally manage software distribution into managed resources. The `<content>` element is its counterpart.

EVENTS

Events are a way to inject asynchronous data into the RHQ server. They're a little like traits. The difference is that one event definition can match multiple event sources, and the number of events that are delivered to the RHQ server can be different each time the polling for events is called.

Events are processed by *EventPoller* – a method that gets called at a regular interval by the plug-in container and that delivers one or more events back into the system.

Two examples of EventPollers are the Logfile pollers, which check for new matching lines in logfiles, and the snmptrapd plug-in.

The plug-in descriptor for snmptrapd should look familiar, but with one new element:

```
<event name="SnmpTrap" description="One single incoming trap"/>
```

The important part is the name attribute, as later it will be the key into the EventDefinition object.

PLUG-IN COMPONENT

In the plug-in component, we use start() and stop() to start and stop polling for events:

```
public void start(ResourceContext context) throws
    InvalidPluginConfigurationException, Exception {

    eventContext = context.getEventContext();
    snmpTrapEventPoller = new SnmpTrapEventPoller();
    eventContext.registerEventPoller(snmpTrapEventPoller, 60);
```

In the above, you first get an EventContext from the passed ResourceContext, instantiate an EventPoller, and register that poller with the EventContext (with 60 seconds between polls).

The plug-in container will start its timer when this registration is done.

In stop(), you unregister the poller again:

```
eventContext.unregisterEventPoller(TRAP_TYPE);
```

TRAP_TYPE is the resource type name as string.

EVENT POLLER

This class is the new piece in the game:

```
public class SnmpTrapEventPoller implements EventPoller {
```

Implementing EventPoller means implementing two methods:

```
public String getEventType() {
    return SnmpTrapdComponent.TRAP_TYPE;
}
```



There we return the content of the name attribute from the <event> tag of the plug-in descriptor. The plug-in won't start if they don't match.

The other method to implement is poll():

```
public Set<Event> poll() {  
    Set<Event> eventSet = new HashSet<Event>();  
    ...  
    return eventSet;  
}
```

To create one Event object, you just instantiate it. You can get the needed type from a call to `getEventType()`.

PLUG-IN COMMUNITY

The RHQ wiki¹⁰ hosts a plug-in community page that shows available plug-ins at <http://support.rhq-project.org/display/RHQ/RHQ+Plugin+Community>. Jopr and its plug-ins are available at <http://www.jboss.org/jopr/>.

Those sites will also provide any updates about plug-in related information.

RHQ developers can be reached in #rhq on irc.freenode.net. Development forums are hosted on <http://forums.rhq-project.org/>.

ABOUT THE AUTHOR

Heiko W. Rupp is an RHQ, Jopr, and JBoss ON developer at Red Hat. He contributed to JBoss AS and other open source projects in the past. He wrote the first German JBoss book and one of the first German books on EJB3. Rupp lives with his family in Stuttgart, Germany.

¹⁰ <http://support.rhq-project.org/display/RHQ/Home>

JBoss SALES AND INQUIRIES NORTH AMERICA

1-888-REDHAT1
www.jboss.com