JBoss Operations Network 3.0 Writing Custom Plug-ins

guidelines for writing custom server and agent resource plug-ins
Edition 3.0.1

Ella Deon Lackey

JBoss Operations Network 3.0 Writing Custom Plug-ins

guidelines for writing custom server and agent resource plug-ins
Edition 3.0.1

Ella Deon Lackey
dlackey@redhat.com

Legal Notice

Abstract

JBoss Operations Network can be extended and customized by developing new plug-ins. The functionality of the server can be extended through server-side plug-ins. Additional resource types can be managed by JBoss ON by creating custom agent resource plug-ins. This guide outlines the requirements and available plug-in configurations for server-side and agent plug-ins to help plug-in writers deploycustom plug0ins in JBoss ON. This guide is not a tutorial in how to write plug-ins, generally.

Table of Contents

1. An Overview of JBoss ON Plug-ins

A plug-in makes an application more useful in a specific kind of way. It is a way of providing new functionality or more options for existing functionality. In JBoss ON, there are two categories of plug-ins, depending on what functionality needs to be created: server-side plug-ins and agent plug-ins. JBoss ON has a very simple and tightly integrated framework for deploying new plug-ins, which makes it relatively easy to extend JBoss ON to do a specific, custom task. Almost any subsystem or functionality in JBoss ON can be expanded and customized by writing additional plug-ins. This guide is an introduction to how to write and implement plug-ins in JBoss ON.

1.1. Extending JBoss ON: Plug-ins Defined

JBoss ON follows a hub and spoke approach with a server at its heart. The agents are deployed locally on resources and interact with the resources, as well as the JBoss ON server. The server (or cluster of servers) processes data coming in from agents. The data are stored in a database connected to the server. Users can look at the data and trigger operations through a web-based GUI on the server.

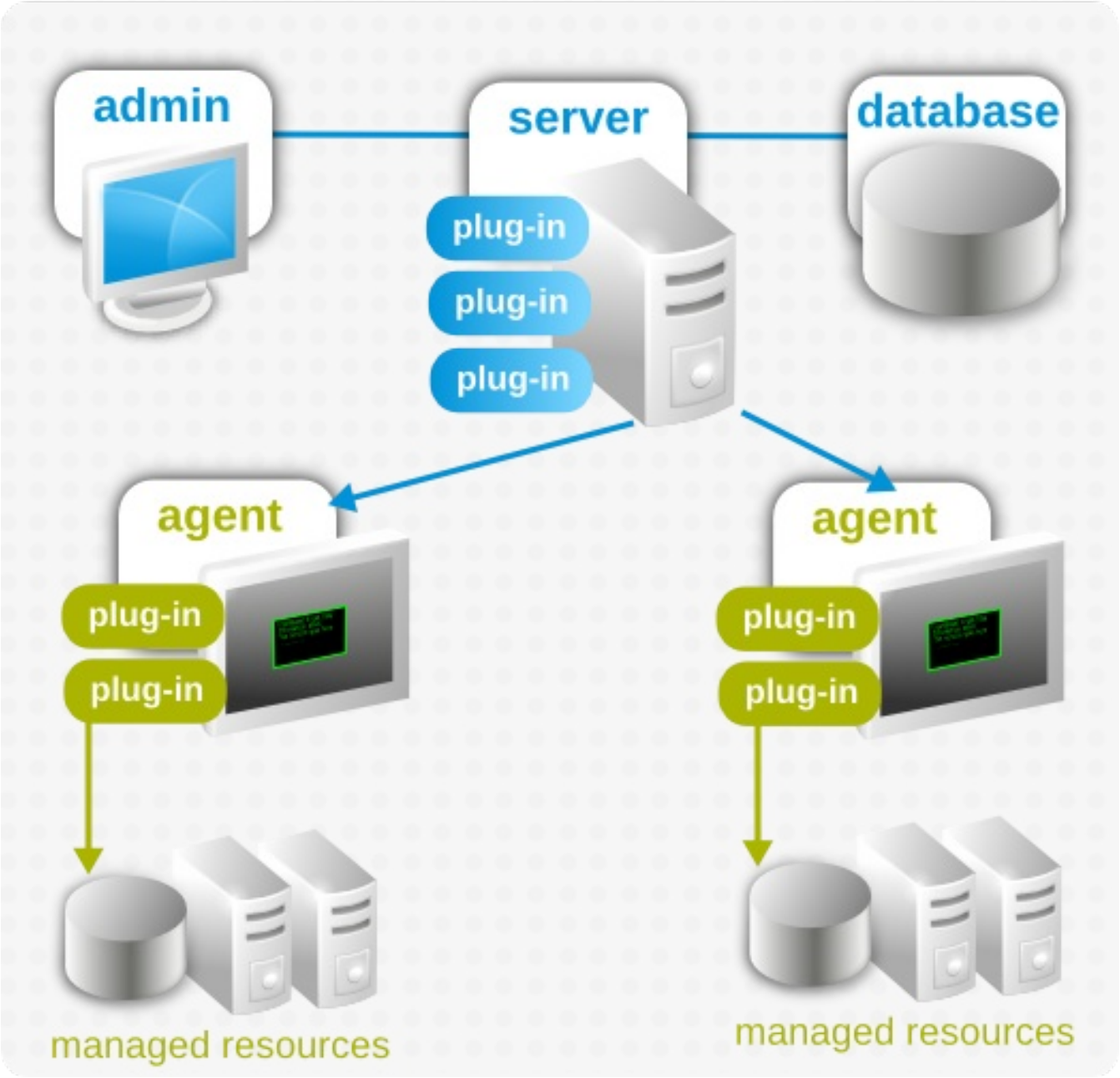


Figure 1. JBoss ON Architecture

A plug-in defines what features it is using and then it contains the code (or API) necessary carry out whatever features or operations it has defined. In JBoss ON, a plug-in is targeted to work on either a server or on an agent.

Server-side plug-ins relate to any operation or task that is performed by the server. This includes alerting and notifications, managing content and packages, setting GUI appearance and functionality, and integrating JBoss ON information with other applications. A server-side plug-in is first identified by the server subsystem which it is associated with, and then with its functionality.

Agent plug-ins are used for any task that relates to resources, primarily managing inventory (by defining resource types) and configuring monitoring. An gent plug-in, then, is associated purely by its resource type.

There are some similarities between server-side and agent plug-ins structurally.

- Every plug-in is packaged as a JAR file.
- Every plug-in has a required XML file, the *plug-in descriptor*, which defines all of the plug-in capabilities.
- Every plug-in contains compiled Java files which contain the code necessary to perform all of the actions defined in the descriptor.
- Plug-ins run inside a *plug-in container*, which is the entity that directly interacts with the plug-ins and starts and stops all plug-ins.
- All custom plug-ins are deployed to the JBoss ON server. Server-side plug-ins are propagated across the high availability cloud to all of the other servers, while agent plug-ins are made available through the server for the agents to download.

1.2. Basic Components of Plug-ins in JBoss ON

There are some common elements that comprise plug-ins in JBoss ON. Each of these elements is described in more detail in the server-side and agent plug-in sections, but this sections provides some more general context on these elements and compares some differences in the way that server-side and agent plug-ins use these elements.

1.2.1. Plug-in Containers

All JBoss ON plug-ins run inside a plug-in container. This container is responsible for loading, starting, and stopping all plug-ins. Neither the agent nor the server interacts directly with plug-ins; rather, the agent and server both host plug-in containers. The agent or server talks to the plug-in container, and the plug-in container talks to the plug-ins.

For an agent plug-in, there is nothing relevant about the plug-in container; all agent plug-ins use the same one. The container is essentially invisible to plug-in writers.

Server-side plug-ins, however ,have a very different relationship with the plug-in containers. The server runs multiple plug-in containers, each one designated for a specific subsystem or purpose. The plug-in container, itself, provides some configuration for server-side plug-ins by providing additional schema definitions and certain kinds of functionality. The plug-in container is the first identifying category for a server-side plug-in by distinguishing the type of server-side plug-in.

The plug-in container, along with controlling the relationship between the agent or server and the plug-in, also moderates the relationship between plug-ins and their classes. The plug-in container manages plug-in dependencies (for agent plug-ins), shared classes, and external libraries required by the plug-in.

1.2.2. Plug-in Descriptor

The plug-in descriptor is the file which defines what a specific plug-in does. This file loads the required API classes that allow the plug-in to interact with its plug-in container and, by extension, the server or agent. It defines the specific configuration for the plug-in instance, sets schedules or operations, and explicitly defines the intended functionality for the plug-in.

The plug-in descriptor is always an XML file. Both agent and server-side plug-ins require that the plug-in descriptor be placed in a **`META-INF/`** directory in the JAR file of the plug-in. For server-side plug-ins, this file must be named **`rhq-serverplugin.xml`**, and for agent plug-ins, **`rhq-plugin.xml`**.

1.2.3. Plug-in Schema Definitions

Since the plug-in descriptor is an XML file, there must be a schema definition to use to configure elements and attributes within the file. All of the plug-ins in JBoss ON use a core schema defined with the agent plug-ins, **`rhq-configuration.xsd`**. Server-side plug-ins extend that schema with an additional schema definition file, **`rhq-serverplugin.xsd`**, and then custom schema definitions for each server-side plug-in type.

1.2.4. Java Files

The actual code for the plug-in is contained in Java files within the plug-in JAR package.

Agent plug-ins usually have at least two and sometimes several Java files for each plug-in. There are several reasons for this:

- Agent plug-ins can define both parent and children elements (platforms, servers, and services) in the same plug-in, and each resource type uses its own plug-in code.
- Agent plug-ins have two and sometimes three discrete functions. Almost every agent plug-in must have a discovery component (discovery Java file) that dictates how to identify and inventory whatever resource type is defined by the plug-in. Additionally, agent plug-ins may enable event collection for resources, which requires a separate component (event poller Java file) to track the resource logs. Last, there has to be a component (Java file) which actually implements the plug-in functionality.
- Agent plug-ins allow dependencies. Parent plug-ins can share classes with their children. An agent plug-in can set a dependency on any other agent plug-in that allows it to load that plug-ins classes. To make plug-ins perform better and to make it easier to access the relevant plug-in code, agent plug-ins are frequently broken into smaller Java files to allow the plug-in code to be reused.

Server-side plug-ins usually have only a single Java file to define the plug-in behavior. Since server-side plug-ins do not have dependencies with each other and do not interact with other subsystems (like discovery and event monitoring) everything related to the plug-in can be defined in a single file.

1.2.5. External Libraries

Any library or class that a plug-in requires that is not contained within its own Java files is an *external* library.

Agent plug-ins can interact through dependencies and shared classes. External libraries or classes for an agent plug-in refer to libraries or classes defined in *another agent plug-in*. This is one reason that it is so common for agent plug-ins to require JMX plug-in dependency, because it makes all of the JMX and EMS libraries in that plug-in available to the other agent plug-in. Agent plug-ins can also share their classes with a child plug-in, which both simplifies library management (by making the same library available to multiple plug-ins at once) and simplifies plug-in writing.

Server plug-ins do not interact with one another, so it is not possible to establish dependencies or share classes between server-side plug-ins, even in the same plug-in container. However, server-side plug-ins do allow external libraries to be packaged in the plug-in JAR file and can access any library in the **`lib/`** directory within the JAR file.

1.3. Downloading the Plug-in Files

Sample plug-ins are available through the RHQ source code. To check out the code:

```
git clone http://git.fedorahosted.org/git/rhq/rhq.git
```

Example agent and server-side plug-ins are in the *sourceRoot*`/etc/samples/` directory. These include both fully-developed examples and plug-in templates that can be used for writing new plug-ins. Rather than checking out the entire source code, you can manually download the sample files at this URL:

```
http://git.fedorahosted.org/git/?
p=rhq/rhq.git;a=tree;f=etc/samples;hb=master
```

2. Writing Server-Side Plug-ins: Background

All JBoss ON plug-ins have a similar configuration and deployment style, with minor differences in what is required for the plug-ins to access the systems. *Server-side plug-ins* refer to any plug-in which accesses the core JBoss ON server to perform its actions; essentially, these are global plug-ins used for the central server behavior.

2.1. A Summary of Server-Side Plug-ins in JBoss ON

Server-side plug-ins extend the functionality of the JBoss ON server. JBoss ON comes with three major categories of server-side plug-ins already:

- Content plug-ins for managing resource configurations
- Alert sender plug-ins for methods to send alert notifications for resources
- GUI plug-ins for customizing work flows within the server interface or for getting additional functionality in the GUI

Server-side plug-ins are not limited to those three categories; the server-side plug-in framework allows substantial access to the server itself. Server-side plug-ins can be used to run remote scripts in response to monitoring events or to provision systems in a custom work flow — anything that is within the purview of the JBoss ON server.

This is a much more casual approach to implementing plug-ins than the more structured, formal agent plug-in system. This allows much more latitude in what plug-in developers are able to accomplish.

Important

All server-side plug-ins have full access to the server's stateless session beans (SLSBs). This allows a lot of latitude and versatility in the functionality of server-side plug-ins and allows access to any server subsystem. However, this also makes server-side plug-ins extremely powerful. Be cautious in writing and deploying server-side plug-ins.

Server-side plug-ins have a different framework for writing and deploying plug-ins than the framework for agent plug-ins. Here are some general bits of information that are useful as you begin writing server-side plug-ins:

- Once the server-side plug-in is built and deployed, the plug-in is a JAR file with a **`META-INF/`** directory which contains the **`rhq-serverplugin.xml`** plug-in descriptor.
- Each plug-in is independent of every other plug-in. Unlike agent plug-ins, server-side plug-ins do not interact with each other. There are no plug-in dependencies for server-side plug-ins.

Server-side plug-ins are organized according to their *type*, and the type corresponds to the subsystem or functional area which the plug-in extends. Each type of plug-in is contained within a defined *plug-in container*.

Server-side plug-ins are managed in the JBoss ON server within a plug-in container that relates to the function of the plug-in, and the plug-in container handles general tasks like starting and stopping plug-ins and providing general configuration settings for that type of plug-in. (All plug-in containers are, themselves, members of a single master plug-in container.)

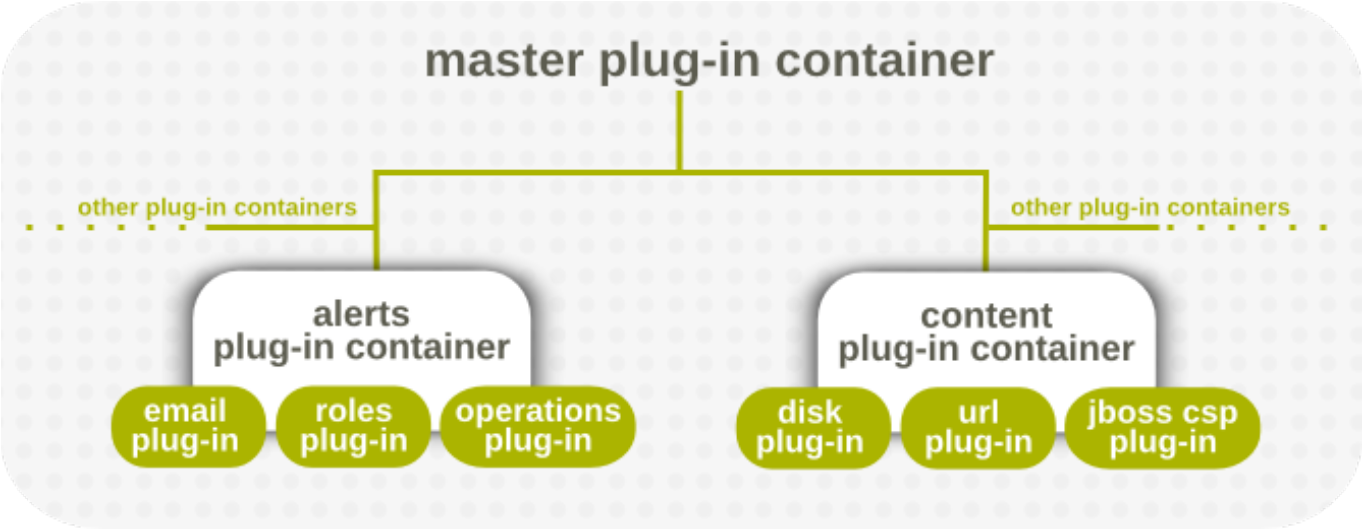


Figure 2. Server-Side Plug-in Containers

There is only one plug-in container for each type of plug-in, but there can be an unlimited number of server-side plug-ins within each plug-in container.

Note

A plug-in container defines what type a plug-in is. A plug-in, then, can only be in *one* plug-in container because it can only be of one type.

Table 1, “Available Plug-in Containers” summarizes the available plug-in containers with JBoss ON.

Table 1. Available Plug-in Containers

Plug-in Type	Description	Container Name
Generic	Catch-all type for any custom plug-ins. This type of plug-in only interacts with the plug-in container for the container to start and stop the plug-in and to initialize and shutdown the plug-in libraries.	Generic Plugin
Alert methods	Defines an alert notification method, or the way that an alert is sent.	AlertHandler

Plug-in Type	Description	Container Name
Content	Contains metadata for a a repository or a group of repositories.	PackageSource
Repository (also Package)	Defines a content repository. Plug-ins can define a single repository,which is then used for provisioning, entitlements, and updates for JBoss ON-managed resources.	ChannelSource
GUI	Defines UI elements to create new UI functionality, such as adding a work flow or creating a specific view.	PerspectivePlugin

Note

New plug-in containers cannot be created without rebuilding JBoss ON, because the plug-in containers are part of the core JBoss ON code. Rather than defining a new plug-in type, use the generic plug-in container, since this provides full access to the server functionality, anyway.

2.2. The Breakdown of Server-Side Plug-in Configuration

JBoss ON plug-ins are packaged **`.jar`** files.The directory structure, libraries, and classes used by those **`.jar`** files is completely up to the discretion and requirements of the plug-in writer, with only one requirement: All plug-in **`.jar`** files must have a plug-in descriptor file, **`META-INF/rhq-serverplugin.xml`**.

Note

The one major guideline when writing plugins is that it should implement the **`org.rhq.enterprise.server.plugin.pc.ServerPluginComponent`** class. This controls the lifecycle of the plug-in within the container.

The server-side plug-in is defined within the JBoss ON server through three types of files:

- An XML file which functions as the plug-in's descriptor
- Java files which pull in the descriptor information and implement the classes for the plug-in.
- Optional library dependencies. Any third-party libraries must be stored in the plug-in JAR file's **`lib/`** directory.

2.2.1. Descriptor and Configuration

The plug-in descriptor is the mechanism that tells the plug-in container how the plug-in should be deployed, along with any additional information about the plug-in configuration and behavior. The plug-in descriptor is contained in an XML file which can use any default or user-defined tags and attributes to define that configuration.

Note

The XML file for the server-side plug-in is defined in the **`rhq-serverplugin.xml`** file in the **`META-INF/`** directory in the plug-in's JAR file. (Default server-side plug-ins follow this same configuration.) This file is required.

The most important configuration in the plug-in descriptor is the basic definition for the plug-in which includes the type of plug-in, its name, and its version. Every plug-in has this basic definition. If the version number is not passed manually in the plug-in descriptor, then it is picked up automatically from the **`MANIFEST.MF`** file.

The key to server-side plug-ins is their flexibility. They have near absolute access to server functionality and can extend any of the existing functions of the server — monitoring, alerting, remote actions, provisioning, resource configuration, whatever. Maintaining this flexibility demands that server-side plug-ins at least have the option of advanced configuration in three general areas:

- Scheduling actions periodically or using cron schedules
- Setting global parameters for all instances of a specific plug-in type
- Allowing local or instance-specific configuration for a plug-in type

2.2.1.1. Definitions and Classes

Each server-side plug-in has a root element that contains attributes for the name, display name, package, version, and other plug-in information. This also imports and defines the XML schema definitions used for the plug-in configuration (which is described in more detail in Section 2.2.2, “Schema Files”).

Example 1. Plug-in Descriptor: Definition

```
<alert-plugin
         name="alert-email"
         displayName="Alert:Email"
         xmlns="urn:xmlns:rhq-serverplugin.alert"
         xmlns:c="urn:xmlns:rhq-configuration"
         xmlns:serverplugin="urn:xmlns:rhq-serverplugin"
         package="org.rhq.enterprise.server.plugins.alertEmail"
         description="Alert sender plugin that sends alert notifications
via email"
         version="1.0"
         >
```

The second part of the plug-in configuration sets the components or classes to use with the plug-in. Every server-side plug-in will implement the **`org.rhq.enterprise.server.plugin.pc.ServerPluginComponent`** class, which provides simple lifecycle management for the plug-in. This component provides the hook for the container to initialize, start, stop, and shut down the plug-in. When a plug-in is initialized, it is given a server plug-in context that provides information about the runtime environment of the plug-in.

This component is a stateful object, remaining alive for as long as the plug-in is initialized. While this object is alive, the plug-in can perform any tasks or call any methods to do the work they need to do.

Developers have the option of invoking a component for the plug-in in one of two ways:

- Using the **<plugin-component>** tag to specify the class (this is available to every type of plug-in)
- Using a user-defined tag to identify the class (this is available to some types of server-side plug-ins, depending on the available schema for the plug-in container)

It's not required to use any given invocation method for a plug-in, so using something like <plugin-component> is optional. Whatever the method of invoking the component, only one plug-in component can be specified in the descriptor.

Example 2. Plug-in Descriptor: Class Info

```
<serverplugin:plugin-component class="MyLifecycleListener" />
```

Alternatively, a container-defined tag (like **<plugin-class>** for the email alert server-side plug-in) can be created for the plug-in. Creating a class introduces the option to provide configuration options or other information with the component.

```
<plugin-class>RolesSender</plugin-class>
```

Note

The example in Example 2, “Plug-in Descriptor: Class Info” is specific to certain type of server-side plug-in. Not all server-side plug-ins support that structure.

Some descriptor tags are made available through the schema defined for that plug-in type. That is the schema that is defined in the plug-in container schema files. In this example, the alert sender plug-in container supplies the **<plugin-class>** element for any alert sender plug-in to hook into the alert mechanism in JBoss ON.

Container-defined schema isn't ad hoc. It can't be dropped into just any plug-in file, and developers cannot define their own schema elements.

2.2.1.2. Control Operations

Sometimes a user need to interact directly with a server plug-in's stateful component. This interaction can take any number of forms, such retrieving a list of agents or resources in contact with the plug-in to testing the plug-in itself.

To allow user-defined controls, the **ServerPluginComponent** class can optionally implement the ControlFacet interface. These control operations can then be invoked directly in the JBoss ON web interface, in the plug-in configuration area.

Control operations are configured in the plug-in descriptor using the **<control>** element, which is a child to the **<plugin-component>** element. Controls are optional, so you don't have to specify any, or you can specify multiple controls. Each control can also have optional parameters for the user to pass to the control operation, as well as (optional) result properties.

Example 3. Control Operation Configuration

```
<serverplugin:plugin-component class="MyLifecycleListener">
```

```
   <serverplugin:control name="testControl" description="A test control
operation">
      <serverplugin:parameters>
         <c:simple-property name="paramProp" required="true"
description="Set to 'fail' to simulate an error"/>
      </serverplugin:parameters>
      <serverplugin:results>
         <c:simple-property name="resultProp" required="false"/>
      </serverplugin:results>
   </serverplugin:control>
</serverplugin:plugin-component>
```

Control operations can be used with any server-side plug-in type.

2.2.1.3. Scheduling Jobs

One of the main advantages of the server-side plug-in framework is the capability to define scheduled jobs for the plug-in. The plug-in container handles actually scheduling and invoking those jobs. The plug-in descriptor has the scheduling information which simply tells the plugin container what classes and methods should be invoked when the job is triggered, how often those jobs should be triggered, and what configuration settings to pass to the job method when it is invoked.

The job can perform any work it needs to get done when it is invoked, including accessing the plug-in's stateful component, as well as any information about the job itself through the **`ScheduledJobInvocationContext`** component.

Job configuration is entirely flexible:

- A job class can be stateless (meaning each job class is instantiated for each job invocation) or it can be stateful by invoking the plug-in component instance.

  Note

  Any server-side plug-in can define a plug-in component to act as the lifecycle listener for the plug-in. Using a plug-in component is extremely useful; in fact, it is the only mechanism for a Generic server-side plug-in to connect with the core server.

- A job can be concurrent, meaning more than one invocation can be performed at any one time on any number of servers (including on the same server). If a job is not concurrent, that means one and only one job invocation can be performed at any time. (If a job is not concurrent *and* is not clustered, then only one job invocation can be performed anywhere in the JBoss ON server cloud).
- A job can be clustered, meaning the job can be run from any server in the JBoss ON server cloud. If a job is not clustered, the job always runs on the machine where the job was scheduled. This works in conjunction with the concurrent setting.
- The schedule can be either periodic (such as running every hour) or recurring on a pattern (such as every Monday at 5pm).
- There can be multiple jobs scheduled for the same plug-in, each in its own **`<map-property>`** under the plug-in's **`<scheduled-jobs>`** entry.

Each scheduled job is a mapping entry that sets the name, schedule, frequency, methods, or classes invoked by the job, and any callback data.

Example 4. Plug-in Descriptor: Scheduled Jobs

```
<serverplugin:scheduled-jobs>
        <!-- notice that we use the map name as the methodName -->
        <c:map-property name="myScheduledJobMethod1">
            <c:simple-property name="enabled" type="boolean"
required="true" default="true" summary="true" description="Whether or not
the job should be scheduled"/>
            <c:simple-property name="scheduleType" type="string"
required="true" default="cron" summary="true" description="Indicates when
the schedule triggers">
                <c:property-options>
                    <c:option value="periodic"/>
                    <c:option value="cron" default="true"/>
                </c:property-options>
            </c:simple-property>
            <c:simple-property name="scheduleTrigger" type="string"
required="true" default="0 0/5 * * * ?" summary="true" description="Based
on the schedule type, this is either the period, in milliseconds, or the
cron expression"/>
            <c:simple-property name="concurrent" type="boolean"
required="false" default="false" summary="true" description="Whether or
not the job can be run multiple times concurrently"/>
            <c:simple-property name="clustered" type="boolean"
required="false" default="true" summary="true" description="Whether or not
the job can be run anywhere in the JBoss ON server cluster, or if it must
be run on the server where the job was schedule."/>
        </c:map-property>
</serverplugin:scheduled-jobs>
```

There is only one **<scheduled-jobs>** container entry. Each individual job is within this container, in mapping (**<map-property>**) entries.

2.2.1.3.1. States for Jobs

Server-side plug-ins can run one of two jobs: stateless or stateful. The only difference between a stateful job and a stateless job is whether the job specifies a class. If a plug-in does *not* specify a class, the plug-in job is stateful because it uses the plug-in component. If the job specifies a class, then the class is instantiated every time a new job starts, so the job is stateless.

At its simplest, a stateless job requires only a class and a method to call when the plug-in is started. For example:

Example 5. Stateless Job Configuration

```
<c:map-property name="statelessJob1" description="invokes a stateless job
class but given a job context">
      <c:simple-property name="class" type="string" required="true"
readOnly="true" default="MyScheduledJob" summary="true" />
      <c:simple-property name="methodName" type="string" required="true"
readOnly="true" default="executeWithContext" summary="true" />
</c:map-property>
```

Aside from the class specified for stateless jobs, stateless and stateful jobs have similar configuration options. Both stateful and stateless jobs can take other optional parameters that help schedule the job. Scheduled jobs use the same configuration properties as other components in the plug-in, but scheduled jobs have a specialized semantics that require special properties to be defined to set create the job. Essentially, this is a property map for each job. These properties include:

1. A method name for the job to invoke. For stateful jobs, the target method is in the plug-in component; for stateless jobs, it is in the class specified with the class property. Either way, the method name tells the server what to call. A default method is already defined in the plug-in component, and stateful jobs can call on that without having a specific method name property.

   ```
   <c:simple-property name="methodName" type="string" required="true"
   readOnly="true" default="executeWithContext" summary="true" />
   ```

   Any method must either have no arguments or have a single argument of the type **`ScheduledJobInvocationContext`**.

2. A setting showing whether the job is enabled.

   ```
   <simple-property name="enabled" type="boolean" ... />
   ```

3. A schedule type showing whether it's a periodic or cron job. The type of job is identified in the option which is set to true. For example:

   ```
   <simple-property name="scheduleType" ... default="periodic" ... >
         <c:property-options>
               <c:option value="periodic" default="true"/>
               <c:option value="cron" />
         </c:property-options>
   </c:simple-property>
   ```

4. The actual schedule for when to run the job (the "trigger"), which can be a time period or a cron schedule. For a periodic job, this gives a time interval, in milliseconds:

   ```
   <simple-property name="scheduleTrigger" type="string" required="true"
   default="60000" ... />
   ```

   For a cron job, the default argument contains the full cron expression:

   ```
   <simple-property name="scheduleTrigger" type="string" required="true"
   default="0 0/5 * * * ?" ... />
   ```

   (A full description of the cron schedule format is at http://quartz.sourceforge.net/javadoc/org/quartz/CronTrigger.html.)

5. A setting on whether the job is concurrent (meaning, whether this job can be running multiple times on more than one server or at the same time). If this is false, so that only one instance of the job can be running at a time, then even if multiple servers are scheduled to run the job, it will only run on one of them.

   ```
   <simple-property name="concurrent" type="boolean" ... />
   ```

6. A job can allow a setting on whether it runs anywhere in the JBoss ON server cloud or if it must be run on the same machine where the job was scheduled. Setting the cluster value to true allows the

job to be called from any server in the JBoss ON cloud, so the job is clustered. This value should be false if the job must be run on all machines on schedule. Since all plug-ins are registered on all servers automatically, a non-clustered job will run on each server, independently.

```
<simple-property name="clustered" type="boolean" default="true" ... />
```

7. A job can optionally contain custom strings which accept callback data.

```
<simple-property name="custom1" type="boolean" required="true"
default="true" summary="true" description="A custom boolean for
callback data"/>
```

Callback data can be of any type — boolean, string, long, or whatever else is appropriate for the job being performed.

8. Stateless jobs have a property that passes the method name of the class. The method name can identify the class that is called in the plug-in component or, alternatively, it can call a class to instantiate when the job is invoked. Both the method and the class keys are shown in Example 5, “Stateless Job Configuration”. Whatever class is used as the target, it must have the method defined in the method name simple property.

   Typically, the class isn't specified because the job will target the stateful plug-in component. The class property allows the option of writing a stateless job, however.

2.2.1.3.2. Concurrent and Clustered Jobs

When a scheduled job is run is determined by its schedule (**`scheduleTrigger`** setting). *Where* and *how* a job is run is determined by two settings: concurrent and clustered.

Just because the scheduled time for a job arrives doesn't necessarily mean that a specific JBoss ON server should run it. The server has to determine which server should run the task, and this is given in the clustered setting. If the clustered setting is set to true, then any server in the JBoss ON server cloud can invoke the task; if it is set to false, then the task can only run on the server where it is scheduled.

Note

One thing about clustering is that while the job *can* run on any server in the JBoss ON server cloud, there is no way to predict or require which servers will run the job. Some machines might never run the job.

On the other hand, the JBoss ON server-side plug-ins are automatically propagated to all servers in the cloud when they are deployed. If clustering is turned off (meaning each job only runs from the local server), all JBoss ON servers will eventually run this job independent of when the other servers run the job. The end result is that you are guaranteed that all JBoss ON servers will run this job on a consistent schedule and may even run more than one of the jobs at the same time.

Once the JBoss ON server identifies where to run the task, then it must find out if the task is already running. If the concurrent setting is true, then the job is invoked every the schedule is triggered, even if the task is already running on another server (or even the same server). If concurrent is set to false and the job is already running somewhere in the JBoss ON server cloud, then the server must wait until that job invocation is complete before it can run the job.

The clustered and concurrent settings can play off each other in several ways.

If a job is clustered but not concurrent, then before the JBoss ON server can invoke a job, it has to check whether it is running anywhere else in the JBoss ON server cloud. If it is, then the server has to wait until that job completes before invoking the new job.

If the job is not clustered and not concurrent, then the JBoss ON server only checks the local machine to see if the job is running. If the job is not running locally, then the JBoss ON server can invoke the job, even if it is running on another server in the cloud because the job is not clustered.

Essentially, the clustered setting determines how strict the concurrency check should be. If clustered is false, the concurrency check is performed only on the machine where the job was scheduled; if clustered is true, the concurrency check is performed on all machines in the cluster.

Table 2. Comparison of Concurrent and Clustered Behavior

Concurrent	Clustered	When the schedule is triggered...
true	true	... the job will always be invoked. It may be invoked on any server in the JBoss ON server cloud.
true	false	... the job will always be invoked and will run on the server where the job is scheduled.
false	true	... the JBoss ON server checks to see if this job is running anywhere else in the JBoss ON server cloud. If it is, the new job must wait until that old job has finished before being invoked. Only one instance of this job can ever be running anywhere in the JBoss ON server cloud.
false	false	... the scheduler checks to see if the job is already running locally before invoking the job. Only one job invocation may be running on the server at any time, but multiple servers in the cloud may be running the job at the same time.

Note

To guarantee that a job will run on *all* servers on a consistent schedule, set *clustered* to false. *concurrent* will determine if you are allowed to start a new job on a machine while an old job is still running on that machine.

To run a job somewhere, but not on any one specified JBoss ON server, set *clustered* to true. *concurrent* will determine if you are allowed to have more than one job running anywhere at any one time.

2.2.1.4. Plug-in Configuration (Both Global and Local)

Global configuration settings can be set for default values or global settings for every instance of that server-side plug-in. All of the global configuration parameters are contained within a **<plugin-configuration>** entry (defined in the standard JBoss ON schema) and then each parameter is identified with a **<simple-property>** item. Global settings are useful for any plug-in which accesses a single identity, such as alerts which use the same email or SNMP account.

Example 6. Plug-in Descriptor: Global Configuration

```
<serverplugin:plugin-configuration>
        <c:simple-property name="user" type="string" required="false"/>
        <c:simple-property name="password" type="password"
required="false"/>
</serverplugin:plugin-configuration>
```

The same server-side plug-in can be created with multiple instances. These different instances are obviously going to require slightly different settings in order to fulfill different functions. For example, different instances of email alert senders should send notifications to different groups of sys admins.

These instance-specific configuration settings are identified in the plug-in descriptor through a configuration entry using schema specific to the server-side plug-in (such as **<alert-configuration>** and **<simple-property>** item).

Example 7. Plug-in Descriptor: Instance-Specific Configuration (Alerts)

```
<alert-configuration>
        <c:simple-property name="emailAddress" displayName="Receiver
Email Address(es)" type="longString"
                description="Email addresses (separated by comma) used
for notifications."/>

h5. </alert-configuration>
```

Each plug-in container type defines its own set of schema, relevant to that type of plug-in. For example, GUI or perspectives have separate explicit schema elements for different types of UI elements.

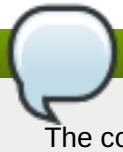
Example 8. Plug-in Descriptor: Instance-Specific Configuration (Perspectives)

```
<perspectivePlugin
   description="The Core Perspective defining Core UI Elements"
   displayName="Core Perspective"
   name="CorePerspective"
   package="org.rhq.perspective.core"
   xmlns="urn:xmlns:rhq-serverplugin.perspective"
   xmlns:serverplugin="urn:xmlns:rhq-serverplugin"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">


<!-- Menu -->
```

```
<menuItem name="logo" displayName="" url="/"
iconUrl="/images/JBossLogo_small.png">
   <position placement="firstChild" />
</menuItem>
```

Check the plug-in container schema in the *sourceRoot*`/modules/enterprise/server/xml-schemas/src/main/resources` directory to see what elements are available for the specific type of plug-in. Not all plug-in types accept local configuration settings; generic plug-ins, for example, only accept global plug-in configuration.



Note

The container schema is included with the RHQ source code, not the JBoss ON packages. To check out the code:

```
git clone http://git.fedorahosted.org/git/rhq/rhq.git
```

The example plug-ins are the *sourceRoot*`/etc/samples/custom-serverplugin/` directory which can be used as a template for writing new plug-ins. Rather than checking out the entire source code, you can manually download the custom-serverplugin files at this URL:

```
http://git.fedorahosted.org/git/?
p=rhq/rhq.git;a=tree;f=etc/samples/custom-serverplugin;hb=master
```

2.2.2. Schema Files

The server-side plug-in is defined through its metadata and configuration in its XML plug-in descriptor file. The configuration elements that are *available* to the descriptor are defined in the XML schema definition (XSD) files for the plug-in container type.

The descriptor file must conform to the elements within the plug-in types scheme. If the descriptor is improperly configured — such as missing required elements or attempting to use elements not defined in the plug-in container's schema — then the plug-in will fail to load.

Every plug-in — both agent plug-ins and server-side plug-ins — use the **`rhq-configuration.xsd`** file. This file defines the basic configuration options available to any plug-in.

The **`rhq-configuration.xsd`** is extended by **`rhq-serverplugin.xsd`**. This file provides additional XML elements that are specific to the functions of server-side plug-ins. This file is referenced by every server-side plug-in.

The last XSD file used by a server-side plug-in is one that is specific to its plug-in container. The plug-in container schema files may define required elements for plug-ins of that type (as with alert sender plug-ins) or may not have any specific schema elements (as with generic plug-ins).

This section takes a really high-level look at the configuration elements and attributes that are associated with each XSD to give you enough familiarity with XSD in general and specifically the files with JBoss ON in order to help write server-side plug-ins and extend the schema as necessary.

More information about each XSD file is available through the comments (in **`<xs:annotation>`** items) in the XSD files themselves. For more information on XSD files and XML schema, check out a reference guide for XML and XSD, like http://www.w3.org/TR/xmlschema-0/.

Note

The JBoss ON XSD files are annotated with descriptions of each configuration area within the file.

2.2.2.1. Parsing the Plug-in Container Schema Files

All of the schema elements available and required for a specific type of server-side plug-in are defined in the schema for that type of plug-in container. There are two relevant elements configured in the XSD files:

- Elements
- Attributes

Note

For more detailed information on XML schema, check out a reference guide for XML and XSD, like http://www.w3.org/TR/xmlschema-0/.

Elements translate into available tags for the plug-ins XML file. For example:

```
<xs:element name="alert-plugin">
```

In the plug-in's XML file, that element defines the tag:

```
<alert-plugin>
Stuff
</alert-plugin>
```

Attributes are flags that available to the tags in the XML file. For example:

```
<xs:attribute name="name">
```

The attribute looks like this in the XML file:

```
<alert-plugin name="myAlertPlugin">
Stuff
</alert-plugin>
```

Elements and attributes are arranged hierarchically in the XSD file. The container element for the plug-in file is defined at the top of the XSD. Child elements reference the parent element's type and are included as sub-elements within the parent's definition. Likewise, any attributes that are available to an element are included within the element's definition.

One of the easiest ways to find the tags and attributes defined for a type of server-side plug-in is to check the plug-in container schema in the *sourceRoot*`/modules/enterprise/server/xml-schemas/src/main/resources` directory and search for `<xs:element name="">` and `<xs:attribute name="">` entries.

Note

The container schema is included with the RHQ source code, not the JBoss ON packages. To check out the code:

```
git clone http://git.fedorahosted.org/git/rhq/rhq.git
```

2.2.2.2. The rhq-configuration.xsd File

The **`rhq-configuration.xsd`** file provides schema which is available for all JBoss ON plug-ins. This is used by both agent and server-side plug-ins.

Note

The container schema is included with the RHQ source code, not the JBoss ON packages. To check out the code:

```
git clone http://git.fedorahosted.org/git/rhq/rhq.git
```

The example plug-ins are the *sourceRoot*`/etc/samples/custom-serverplugin/` directory which can be used as a template for writing new plug-ins. Rather than checking out the entire source code, you can manually download the custom-serverplugin files at this URL:

```
http://git.fedorahosted.org/git/?
p=rhq/rhq.git;a=tree;f=etc/samples/custom-serverplugin;hb=master
```

The most commonly used elements defined in the **`rhq-configuration`** schema relate to setting configuration values for a plug-in, like **`<simple-property>`** and **`<map-property>`**.

Table 3. rhq-configuration.xsd Schema Elements

Element	Description
configuration-property	For adding a configuration attribute to a plug-in for user-defined settings.
simple-property	For setting a default configuration value.
option	For setting whether a property's values come from an enumerated list (false) or can be anything defined by the user (true).

The **`rhq-configuration.xsd file`** also defines the most common flags that can be used for the plug-in descriptor, including the required **`name`** and optional **`displayName`** attributes.

Table 4. rhq-configuration.xsd Schema Element Attributes

Attribute	Description
name	*Required.* Gives a unique name for the plug-in.

Attribute	Description
displayName	Gives the name to use for the plug-in in the GUI. If this isn't given, then the name value is used.
description	Gives a short description of the plug-in.

There are many other elements and attributes set in the **rhq-configuration.xsd** file. Each one is described by the text in the **<xs:annotation>** tags for the item.

2.2.2.3. The rhq-serverplugin.xsd File

The **rhq-serverplugin.xsd** is the central server-side plug-in schema file.

The **rhq-serverplugin.xsd** file provides schema elements that are important for every server-side plug-in. Possibly the two most important elements are **<server-plugin>** (for the plug-in's root element) and **<scheduled-jobs>** (for running jobs on a resource or server).

The most common elements in the **rhq-serverplugin.xsd** file are listed in Table 5, “rhq-serverplugin.xsd Schema Elements”.

Table 5. rhq-serverplugin.xsd Schema Elements

Element	Description
server-plugin	Contains the root element for the plug-in descriptor.
help	Contains additional usage information or other tips that can help users integrate the plug-in with other applications.
plugin-component	Identifies a class that will be notified when the plug-in stops or starts. This is a stateful object and is the target of any scheduled stateful jobs.
scheduled-jobs	Defines a schedule for the plug-in to execute any specified task

Most of the attributes defined within the **rhq-serverplugin.xsd** contain flags that are used within the root element of the plug-in descriptor. These add additional management attributes for controlling the release and updates of server-side plug-ins.

Table 6. rhq-serverplugin.xsd Schema Element Attributes

Attribute	Description
package	For setting the plug-in package name.
version	For setting the version of the plug-in. If the version isn't set in the descriptor, the plug-ins JAR file, **META-INF/MANIFEST.MF**, must define the version number in the **Implementation-Version** setting.
apiVersion	For setting the version of the API used to write the plug-in.

There are many other elements and attributes set in the **rhq-serverplugin.xsd** file. Each one is described by the text in the **<xs:annotation>** tags for the item.

2.2.3. Java Class Files

Any Java class files used by the plug-in to implement elements like **`ServerPluginComponent`** or **`ControlFacet`** must be available in the JAR file for the plug-ins.

2.3. Anatomy of Alert Sender Server-Side Plug-ins

An alert notification sender is simply the method used to send an alert. Each sender is implemented through an alert sender plug-in. Multiple instances of the same type of plug-in can be configured with different settings; all the plug-in provides is the functionality of sending an alert in that way.

Alert senders are implemented as server-side plug-ins (with the same general configuration concepts as those covered in Section 2.2, “The Breakdown of Server-Side Plug-in Configuration”. The server-side plug-in framework allows the alert sender configuration to be easily extended through custom plug-ins or even by editing the configuration of the default server-side plug-ins.

This section deconstructs the elements of one of the default server-side plug-ins — the email alert sender — to make the process of creating an alert sender clear and simple.

2.3.1. Default Alert Senders

JBoss ON provides multiple alert sender plug-ins with the default installation, which cover some of the most common ways of sending an alert.

Table 7. Default Alert Senders

Alert Method	Description	Plug-in Name
Email	Sends emails with the alert information to a user or list of users.	alert-email
Roles	Sends an internal message to a JBoss ON user role.	alert-roles
SNMP	Sends a notification to an SNMP trap.	alert-snmp
Operations	Initiated a JBoss ON-supported task on a target resource.	alert-operations
Subject	Sends a notification to a user in JBoss ON.	alert-subject

Plug-in developers and administrators can create and deploy custom alert sender plug-ins to cover other scenarios or formats that are specific to an organization, such as additional instant messaging systems.

2.3.2. Breakdown of a Real Alert Sender Plug-in

As described in Section 2.2, “The Breakdown of Server-Side Plug-in Configuration”, any server-side plug-in uses three types of files for its configuration:

- An XML plug-in descriptor that conforms to a given XML schema file (XSD)
- Java files

The XML plug-in descriptor and the Java files are unique to every plug-in. All of the default alert senders, however, use the same three schema files to provide attributes for the descriptor.

Section 3.3, “Deploying Server-Side Plug-ins” covers the process for building and deploying plug-ins. This section annotates the elements of each of the configuration files used to define a default alert sender (alert-email) as an example of how to write an alert plug-in.

2.3.2.1. Descriptor

Every plug-in descriptor is a file called **`rhq-serverplugin.xml`** in the **`src/main/resources/META-INF/`** file for that plug-in.

Note

The default alert schema has to be used with the plug-in descriptor for the alert plug-in validator to work and for the alert to tie into the monitoring system successfully.

The header in the plug-in descriptor pulls in the schema files to use with the plug-in and defines the package information (class, description, version number) for the plug-in. The **`displayName`** flag contains the name to give *for the plug-in* in the list of installed server-side plug-ins.

```
<alert-plugin
     name="alert-email"
     displayName="Alert:Email"
     xmlns="urn:xmlns:rhq-serverplugin.alert"
     xmlns:c="urn:xmlns:rhq-configuration"
     xmlns:serverplugin="urn:xmlns:rhq-serverplugin"
     package="org.rhq.enterprise.server.plugins.alertEmail"
     description="Alert sender plugin that sends alert notifications via
email"
>
```

The next section supplies help text for the alert.

```
<serverplugin:help>
     Used to send notifications to direct email addresses.
</serverplugin:help>
```

The help text is displayed in a help description section in the UI.

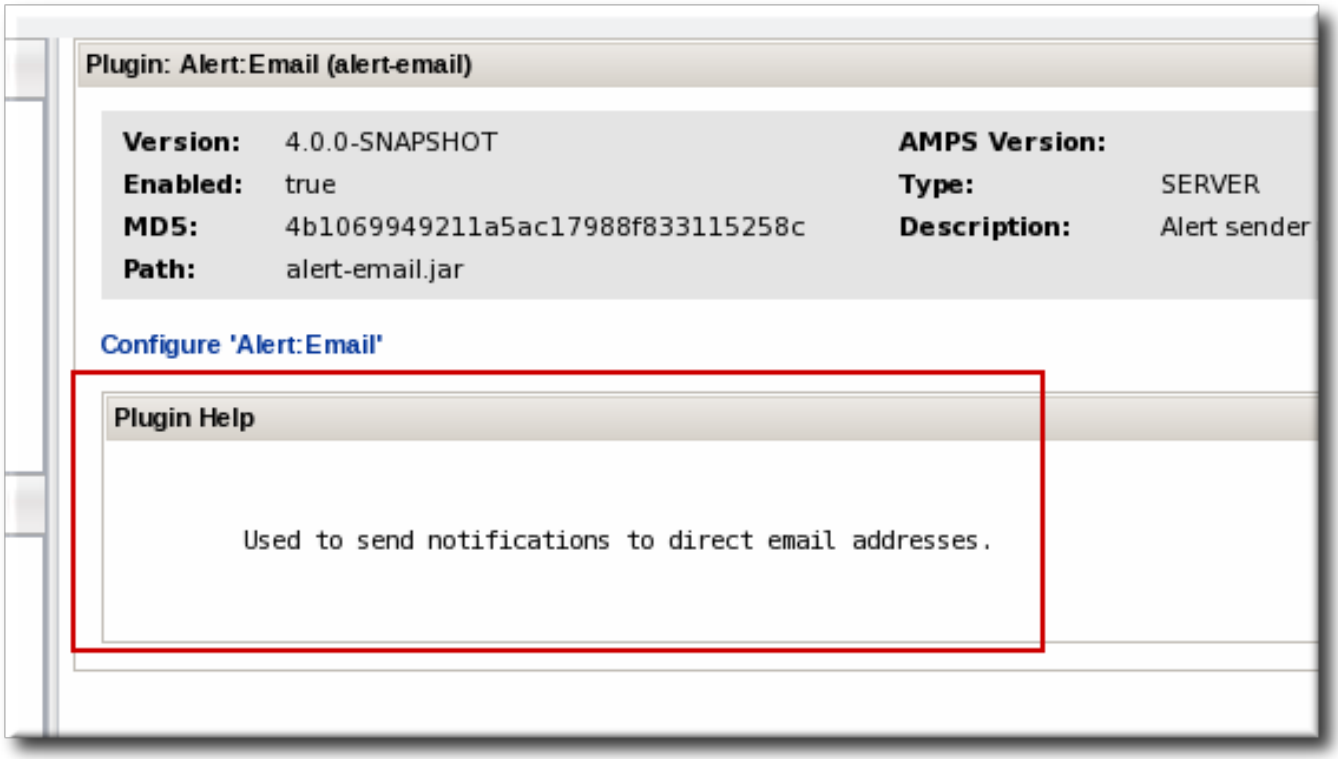


Figure 3. Alert Help Text

The next section in the descriptor is filler for the alert-email plug-in.

```
<!-- startup & tear down listener, + scheduled jobs
<serverplugin:plugin-component />
-->
```

For other types of server-side plug-ins, this area could contain scheduling information in a **<scheduled-jobs>** element or implement a Java class in a **<plugin-component>** element. There's no reason to schedule any jobs with an alert sender since the plug-ins don't perform tasks; they provide methods of sending message from the server when an event is detected.

The global preferences define parameters that apply to every single instance of the alert, meaning it applies to every notification which is configured to use that alert sender. These global configuration parameters can be configured in the XML file, but they can also be edited through the JBoss ON GUI, as described in Section 3.8, “Setting Plug-in Configuration Properties”.

For the alert-email plug-in, these parameters include the sender mail address to use for notifications, the mail server, and any login credentials.

```
<!-- Global preferences for all email alerts -->

<serverplugin:plugin-configuration>
     <c:simple-property name="mailserver" displayName="Mail server address"
type="longString"
          description="Address of the mail server to use (if not the default
JBoss ON one )"
          required="false"/>
     <c:simple-property name="senderEmail" displayName="Email of sender"
type="string"
          description="Email of the account from which alert emails should
come from"
```

```
          required="false"/>
     <c:simple-property name="needsLogin" displayName="Needs credentials?"
          description="Mark this field if the server needs credentials to
send email and give them below" type="boolean"
          default="false"/>
     <c:simple-property name="user" type="string" required="false"/>
     <c:simple-property name="password" type="password" required="false"/>
</serverplugin:plugin-configuration>
```

If defaults are set, then there is a default value given in the configuration itself. The alert-email plug-in, however, doesn't have defaults set for any of its parameters, so the values for the plug-in configuration have to be added through the plug-in configuration page.

The **<short-name>** element is required for every alert sender plug-in. This gives the name that is used for the alert sender type in the notification area of the alert definition.

```
<!-- How does this sender show up in drop downs etc -->

<short-name>Email</short-name>
```

Since the **<short-name>** value is used in drop-down menus and other user-oriented areas, this value is much more human-friendly than the displayName value.

The next section gives the plug-in class used to send the alert notification. The component for server-side plug-ins is typically **org.rhq.enterprise.server.plugins.***pluginName*, taken from the **package** element in the **<plugin>** element of the descriptor. For the alert-email plug-in, the full package name is **org.rhq.enterprise.server.plugins.alertEmail**, pointing to the **EmailSender.java** class.

```
 <!-- Class that does the actual sending -->

<plugin-class>EmailSender</plugin-class>
```

The last section in the alert-email descriptor provides the other half to the communication configuration. The global parameters set things that apply to every notification, like the mail server that the JBoss ON server used to send the email notification. The **<alert-configuration>** entry provides information that is configured individually, for every notification instance which uses that alert sender type. For **alert-email**, this is a field which allows a list of email addresses that will receive the emailed notifications.

```
 <!-- What can a user configure when defining an alert -->

<alert-configuration>
     <c:simple-property name="emailAddress" displayName="Receiver Email
Address(es)" type="longString"
          description="Email addresses (separated by comma) used for
notifications."/>
     </alert-configuration>
```

2.3.2.2. Java Resource

The first part of the Java file identifies the package name and imports whatever properties are required for that type of sender. For the email sender Java file, this includes configuration to pull in the alert send plug-in container, the notification templates, and other classes to define alerts.

```
package org.rhq.enterprise.server.plugins.alertEmail;
```

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

import org.rhq.core.domain.alert.Alert;
import org.rhq.core.domain.alert.notification.SenderResult;
import org.rhq.enterprise.server.plugin.pc.alert.AlertSender;
import org.rhq.enterprise.server.util.LookupUtil;
```

The rest of the EmailSender.java file pulls data from the notification configuration and the plug-in's global configuration.

The opening sets up the sender.

```
public class EmailSender extends AlertSender {

    @Override
    public SenderResult send(Alert alert) {
        String emailAddressString =
alertParameters.getSimpleValue("emailAddress", null);
        if (emailAddressString == null) {
            return SenderResult.getSimpleFailure("No email address given");
        }
```

The next lines pull in the email address to receive the notification from the notification configuration and the mail server to send the notification and the sender's email account from the global configuration.

```
        List<String> emails = AlertSender.unfence(emailAddressString,
String.class, ",");
        try {
            Set<String> uniqueEmails = new HashSet<String>(emails);
            Collection<String> badEmails = LookupUtil.getAlertManager()
                .sendAlertNotificationEmails(alert, uniqueEmails);

            List<String> goodEmails = new ArrayList<String>(uniqueEmails);
            goodEmails.removeAll(badEmails);

            SenderResult result = new SenderResult();
            result.setSummary("Target addresses were: " + uniqueEmails);
            if (goodEmails.size() > 0) {
                result.addSuccessMessage("Successfully sent to: " +
goodEmails);
            }
            if (badEmails.size() > 0) {
                result.addFailureMessage("Failed to send to: " + badEmails);
            }
            return result;
        } catch (Throwable t) {
            return SenderResult.getSimpleFailure("Error sending email
notifications to " + emails + ", cause: "
                + t.getMessage());
        }
```

```
    }

    @Override
    public String previewConfiguration() {
        String emailAddressString =
alertParameters.getSimpleValue("emailAddress", null);
        if (emailAddressString == null || emailAddressString.trim().length()
== 0) {
            return "<empty>";
        }
        return emailAddressString;
    }
}
```

The last part configures the responses for the email alert plug-in, simple failure or success.

```
      catch (Exception e) {
         log.warn("Sending of email failed: " + e);
         return SenderResult.getSimpleFailure("Sending failed :" +
e.getMessage());

      }
      return SenderResult.getSimpleSuccess("Send notification to " + txt +
", msg-id: " + status.getId());
   }
}
```

2.3.2.3. Schema Elements

The alert-email plug-in (as all of the default alert sender plug-ins) uses three schema files:

- **`rhq-configuration.xsd`**, which is used by all JBoss ON plug-ins
- **`rhq-serverplugin.xsd`**, which is used by all server-side plug-ins
- **`rhq-serverplugin-alert.xsd`**, which is used by alert plug-ins

The schema in these files build on and expand each other.

The **`rhq-serverplugin-alert.xsd`** file is required for any alert sender plug-in. While additional schema files can be added to contain other elements, the alert schema already contains several very useful schema elements for the alert sender plug-ins.

Table 8. Useful Alert Schema Elements

Schema Element	Description	Parent Tag
alert-plugin	The root element for a single alert plug-in definition.	None.
short-name	The display name for the plug-in, which is used in the UI.	alert-plugin
plugin-class	The class which implements the plug-in's functionality.	alert-plugin

Schema Element	Description	Parent Tag
alert-configuration	A (default) configuration element to display in the UI when the alert instance is configured. This includes general data like a username, password, URL, server name, or port.	alert-plugin

3. Writing Server-Side Plug-ins: Procedures

3.1. Writing Server-Side Plug-ins

Server-side plug-ins are highly flexible. While there are some categories that structure the behavior of server-side plug-ins (alerts, content, GUI), generic plug-ins can cover almost any server function.

Because of the flexibility of server-side plug-ins, there is no one way to write plug-ins. The outline of writing these plug-ins is simple:

1. The example plug-ins are the *sourceRoot*`/etc/samples/custom-serverplugin/` directory which can be used as a template for writing new plug-ins. Manually download the custom-serverplugin files to use as a template:

   ```
   http://git.fedorahosted.org/git/?
   p=rhq/rhq.git;a=tree;f=etc/samples/custom-serverplugin;hb=master
   ```

2. Identify the type of plug-in. Each server-side plug-in is managed by a higher level plug-in container, which correlates to the type or function of the plug-in.

3. *Optional*. Write custom schema for the plug-in configuration.

4. Create the directory for the custom plugin in the *sourceRoot*`/modules/enterprise/server/plugins` directory. For example:

   ```
   mkdir myPlugin
   cd myPlugin/
   mkdir -p src/main/java/org/rhq/enterprise/server/plugins/myPlugin
   mkdir -p src/main/resources/META-INF
   ```

5. Copy the **`pom.xml`** file from a similar existing plug-in to use for the Maven builds to package your new plug-in. For example:

   ```
   cp ../alert-email/pom.xml .
   ```

6. Edit the **`pom.xml`** file so that its properties reflect the new plug-in.

Note

Be sure to include the location of the parent repositories used by server-side plug-ins, which are in **`https://repository.jboss.org/nexus/content/groups/public/org/rhq/rhq-enterprise-server-plugins-parent/`**. For example:

```
  <repositories>
   <repository>
    <snapshots>
     <enabled>true</enabled>
    </snapshots>
    <id>jboss</id>
    <name>JBoss Repository</name>

<url>https://repository.jboss.org/nexus/content/groups/public/org
/rhq/rhq-enterprise-server-plugins-parent/</url>
   </repository>
...
  </repositories>
```

7. Write the plug-in descriptor that defines that specific plug-in instance. The plug-in descriptor is the heart of the plug-in since it defines everything from the plug-in classes to scheduled jobs. Plug-in descriptor elements are covered in Section 2.2.1.4, “Plug-in Configuration (Both Global and Local)”.
8. Implement the Java classes for the plug-in.
9. Build the plug-in. During the Maven build process, the plug-in files can be validated.

```
mvn install
```

10. Deploy the plug-in, as in Section 3.3, “Deploying Server-Side Plug-ins”. When a server-side plug-in is deployed on one server, it is automatically propagated to all of the other JBoss ON servers in the cloud.

3.2. Validating Server-Side Plug-ins

The JBoss ON server includes special classes that validate server-side plug-ins when the plug-in is built. This validation is done as part of the Maven build process.

Validation means that the build process checks that the server-side plug-in descriptor is acceptable and complete. Every server-side plug-in is checked for a handful of things:

- The XML is well-formed and validates with the configured server plug-in XML schema
- If a plug-in component is specified, its class is found in the plug-in JAR and can be instantiated
- All scheduled jobs are configured properly
- The plug-in has a valid version
- The plug-in configuration is declared correctly

Note

Plug-in validation is done during the Maven build process when creating the JAR file. If the plug-in JAR is built using another system or on a different machine, then validation isn't run.

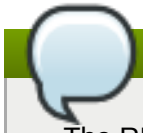
Plug-ins are not validated automatically just because they are built using Maven; any plug-ins to be validated must be added to the validator's **`pom.xml`** configuration file.

1. Open the **`pom.xml`** file in the *sourceRoot*`/modules/enterprise/server/plugins/validate-all-serverplugins/` directory.
2. Add a **`<pathelement>`** line to the file which points to the custom server-side plug-in JAR file. For example:

```
<pathelement location="../myPlugin/target/myPlugin.jar" />
```

3. Build the plug-in.

```
mvn install
```



Note

The RHQ source code includes a validator utility which can be used with custom plug-in infrastructure. This is contained in the **`org.rhq.enterprise.server.plugin.pc.ServerPluginValidatorUtil`** class.

3.3. Deploying Server-Side Plug-ins

Server-side are deployed in one of two ways:

- Copying the plug-in JAR file into the *sourceRoot*`/plugins/` folder in the server root directory (locally).
- Uploading the plug-in JAR file through the web interface (remotely).

Server-side plug-ins are hot-deployed, so they are active as soon as they are deployed without having to restart any of the JBoss ON servers. Every server-side plug-in is deployed globally and is automatically propagated among the server cloud. The configuration for each server is polled regularly (at an interval defined in the server properties file).

It doesn't matter the method used to deploy the server-side plug-in; the plug-in is active and propagated the same either way.

By default, every server-side plug-in is automatically enabled (and therefore active) unless the configuration in the plug-in descriptor explicitly disables the server. When a plug-in is deployed and enabled, it is automatically propagated to other JBoss ON servers in the infrastructure.

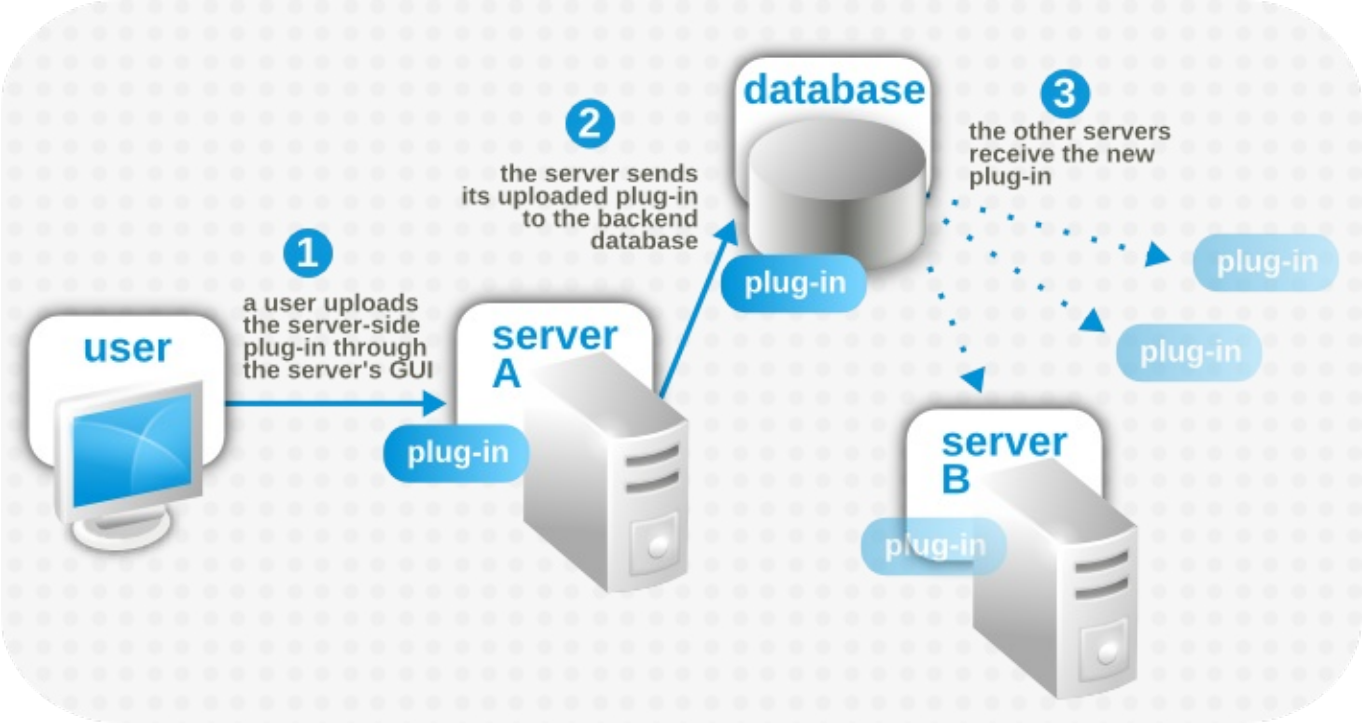


Figure 4. Server-Side Plug-in Propagation

There are three possible states for a server-side plug-in:

- Deployed and enabled
- Deployed and disabled
- Undeployed

When a plug-in is deployed, it is available to the server. Undeploying deletes the plug-in from the server plug-ins directory and lists it in the server's databases as deleted.

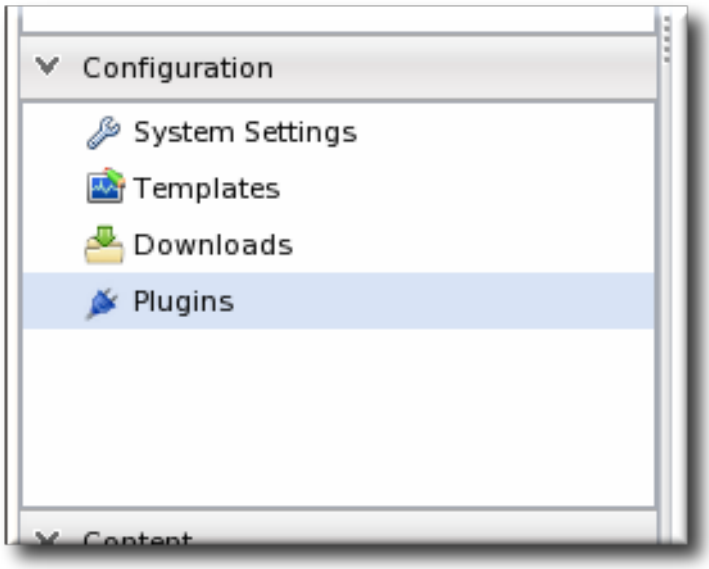
Undeploying the plug-in is a permanent deletion from the *server configuration*. Even if the JAR file still exists somewhere in the JBoss ON cloud, all of the JBoss ON servers will ignore it because it is marked as undeployed in the shared database. This prevents accidentally reloading and re-deploying the plug-in or applying upgrades to the plug-in.

If for some reason the plug-in should be re-deployed later on, then there is a purge option for undeployed plug-ins. This removes the database entry for the plug-in marking it as deleted (undeployed). Once that entry is purged, then the plug-in can be deployed again, fresh.

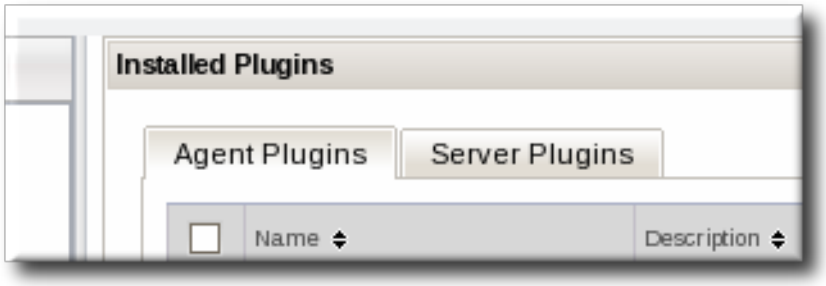
3.3.1. Remotely Deploying Plug-ins

1. In the top menu, click the **`Administration`** tab.

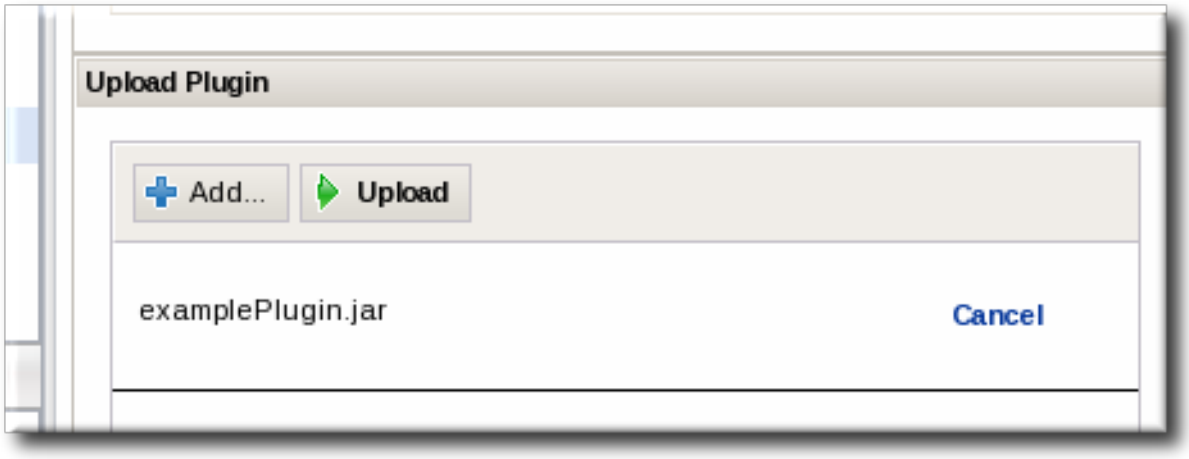
2. In the **`Configuration`** box on the left navigation bar, click the **`Plugins`** link.



3. Click the **`Server Plugins`** tab.



4. Scroll to the **`Upload Plugin`** section at the bottom of the page.
5. Click the **Add** button, and browse to the plug-in JAR file's location. If there are multiple plug-ins to deploy, just hit **Add** again and add in each one.
6. When all of the plug-ins to be deployed are listed in the box, click the **`Upload`** button.



Any plug-ins uploaded to one server will be automatically deployed and registered on all other JBoss ON servers in the cloud within a few minutes.

3.3.2. Locally Deploying Plug-ins

Each server installation has a top-level plugins/ directory. The server periodically polls this directory. Any new or updated JAR files are copied to the appropriate directory in the server configuration, and then the original JAR file is deleted from the plugins/ directory.

If the JAR file is on the same host machine as an JBoss ON server, the JAR file can just be copied into that *sourceRoot*`/plugins/` directory and the server will deploy it.

3.4. Updating Server-Side Plug-ins

Server-side plug-ins can be updated simply by deploying updated plug-in JAR files. The plug-in descriptor can contain a version number for the plug-in package. The server uses this version number (or, alternatively, the **`Implementation-Version`** setting found in the **`META-INF/MANIFEST.MF`** file in the JAR file) to identify later version of the plug-in and to update the plug-ins on the JBoss ON servers in the cloud.

3.5. Disabling Server-Side Plug-ins

Plug-ins are active when they are enabled. A flag in the plug-in descriptor indicated whether the plug-in is enabled. The assumption is that all plug-ins are enabled when they are deployed, so if that parameter is not explicitly set, then the server assumes that the plug-in is enabled.

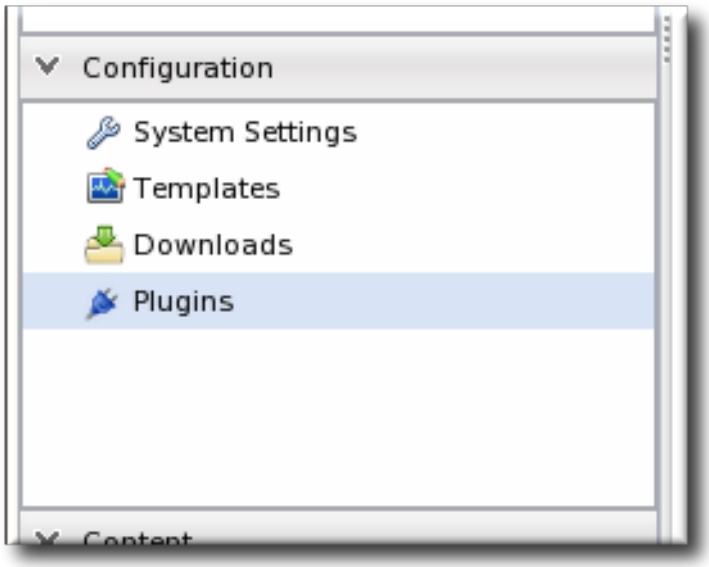
It's convenient in some situations to suspend running a plug-in, such as when the plug-in's scheduled jobs need to be suspended. In that situation, the plug-in can be disabled.

Disabling a plug-in is very different than undeploying it. Even when it's disabled, the plug-in is still listed in the configuration for all of the JBoss ON servers in the cloud; disabling the plug-in simply tells the server to skip loading and starting the plug-in. Undeploying a plug-in, on the other hand, removes the plug-in from the server configuration and prevents any server in the cloud from loading it.

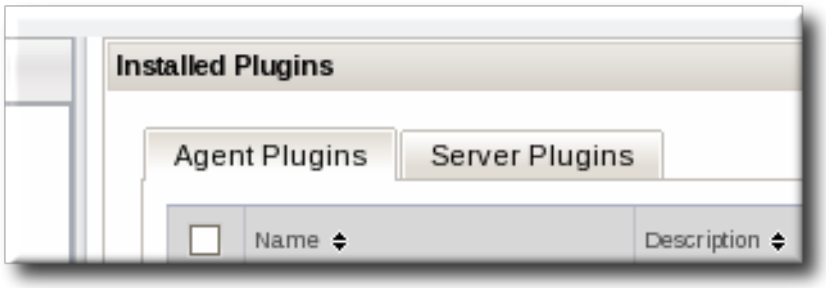
To disable a plug-in:

1. In the top menu, click the **`Administration`** tab.

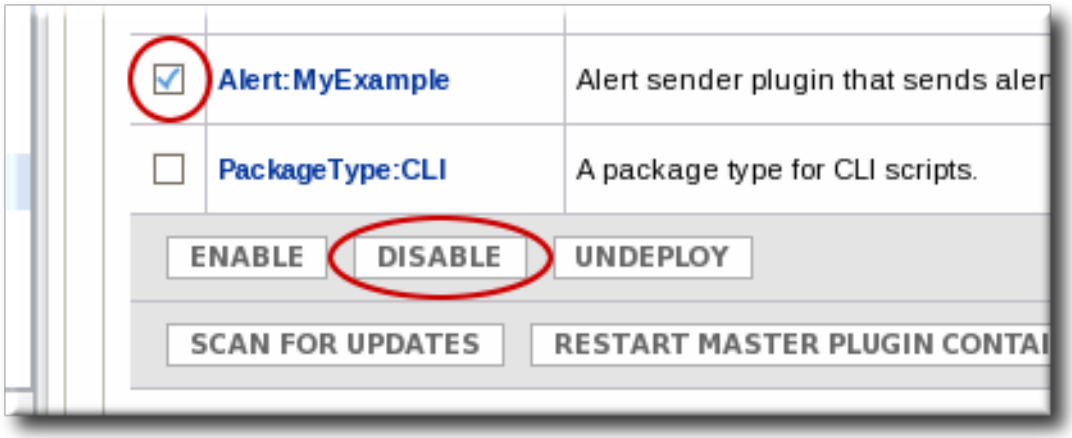
2. In the **`Configuration`** box on the left navigation bar, click the **`Plugins`** link.



3. Click the **`Server Plugins`** tab.



4. Check the box by any server-side plug-in which should be disabled.
5. Click the **DISABLE** button.



Any disabled plug-in can be re-enabled later by selecting that plug-in and clicking the **ENABLE** button.

3.6. Removing and Re-deploying Server-Side Plug-ins

As mentioned in Section 3.3, “Deploying Server-Side Plug-ins”, there are basically three states that a plug-in can be in:

- Deployed and enabled
- Disabled
- Undeployed

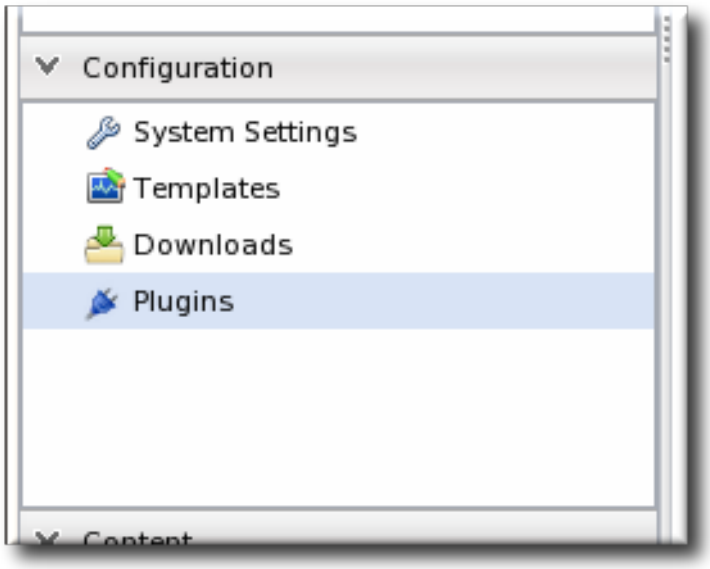
Deployed plug-ins — whether they're enabled or disabled — are listed in the JBoss ON servers as available and the JAR files are in the server configuration. However, undeployed plug-ins are deleted from the server configuration and have a special entry indicating that the plug-in should never be loaded, even if the JAR file is added again.

In other words, undeploying a server-side plug-in permanently removes it from the configuration. To re-deploy a plug-in, the undeploy entry for the plug-in has to be removed, and then the plug-in can be re-deployed.

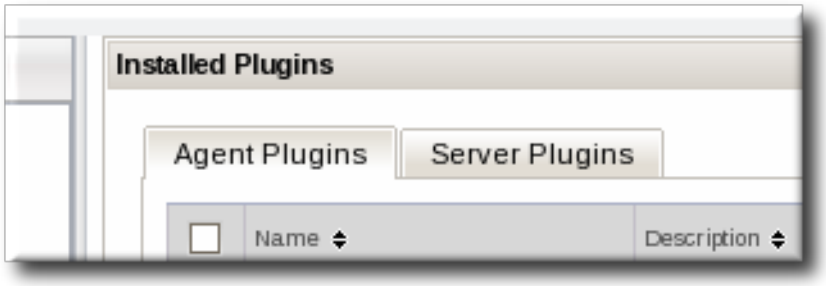
3.6.1. Undeploying a Plug-in

1. In the top menu, click the **Administration** tab.

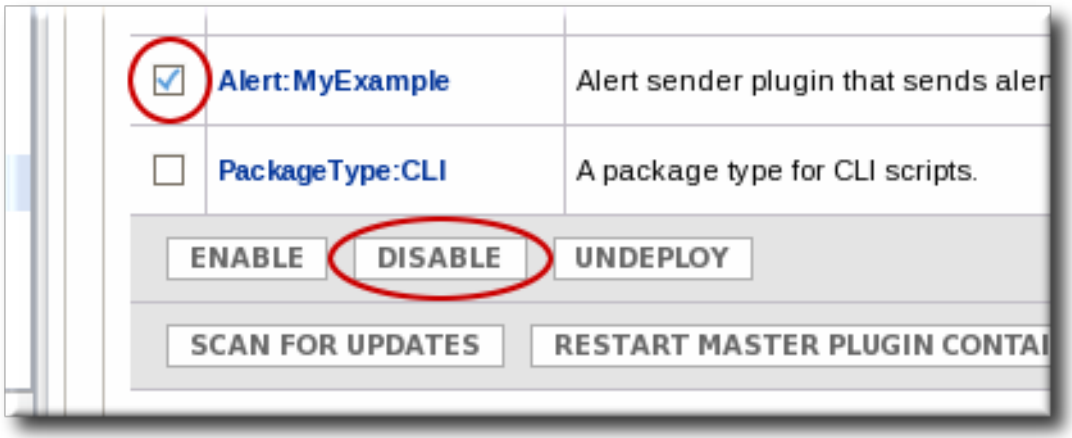
2. In the **Configuration** box on the left navigation bar, click the **Plugins** link.



3. Click the **`Server Plugins`** tab.



4. Select the checkbox by the plug-ins to undeploy.
5. Click the **`Undeploy`** button.



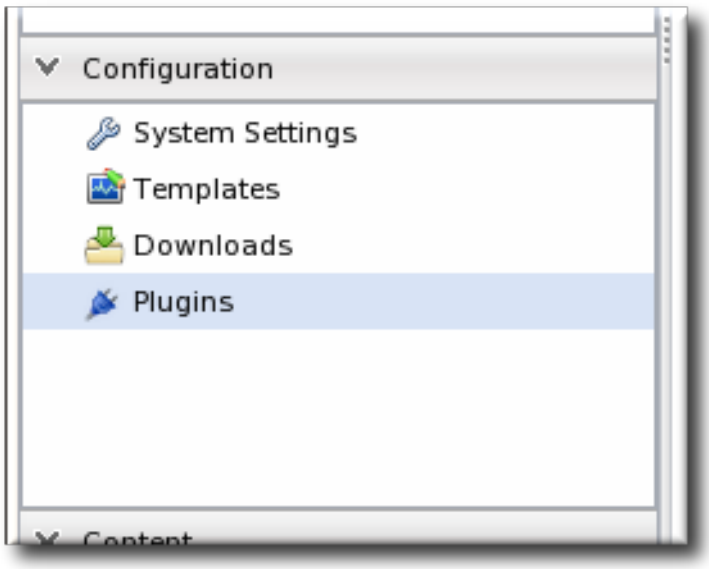
To view all of the undeployed plug-ins, click the **`SHOW UNDEPLOYED`** button.

3.6.2. Re-deploying a Plug-in

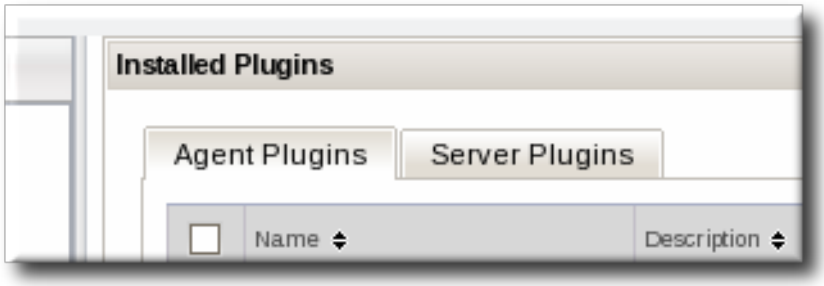
1. In the top menu, click the **`Administration`** tab.

2. In the **`Configuration`** box on the left navigation bar, click the **`Plugins`** link.

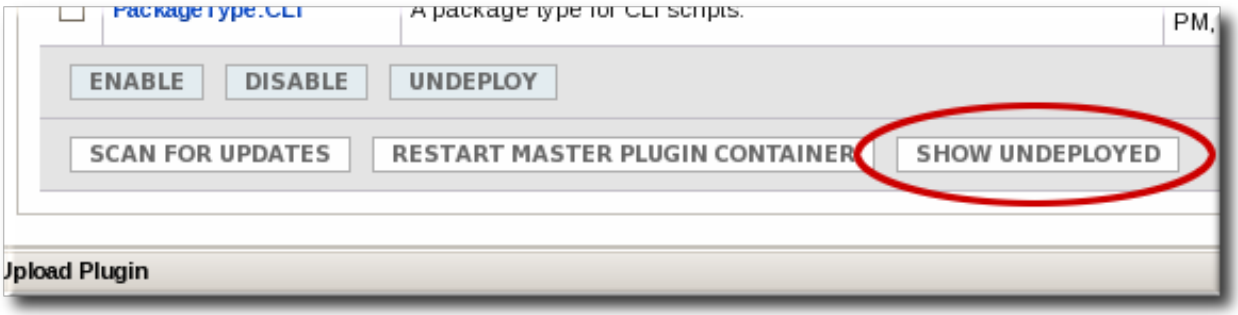
2. In the **Configuration** box on the left navigation bar, click the **Plugins** link.



3. Click the **Server Plugins** tab.

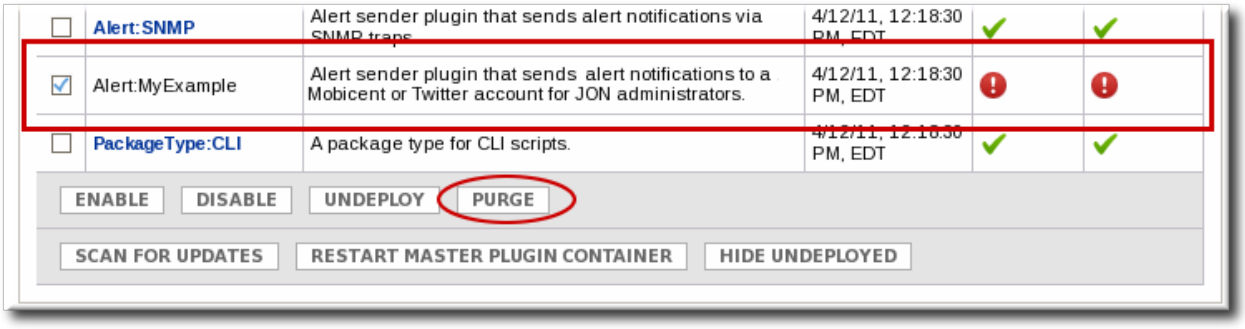


4. Click the **SHOW UNDEPLOYED** button at the bottom of the plug-ins list.



5. Select the checkbox by the plug-in to re-deploy, and then click the **PURGE** button. This removes the entry in the JBoss ON database that tells the servers to ignore that plug-in and any updates to it.

   Undeployed plug-ins have only black text, with no active link to the plug-in configuration. They also have two red marks, indicating they are both disabled and undeployed.

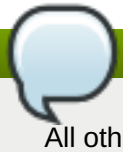


6. Add and upload the plug-in like it is being deployed as new. This is described in Section 3.3, "Deploying Server-Side Plug-ins".

3.7. Restarting Plug-in Containers

Each type of server-side plug-in is controlled by a corresponding plug-in container. Each plug-in container is controlled, in turn, by a master plug-in container. The plug-in containers load, start, and stop plug-ins.

As developers hot-deploy new server-side plug-ins, it can be useful to restart the plug-in container to check the plug-in performance. This is done by restarting the master plug-in container.

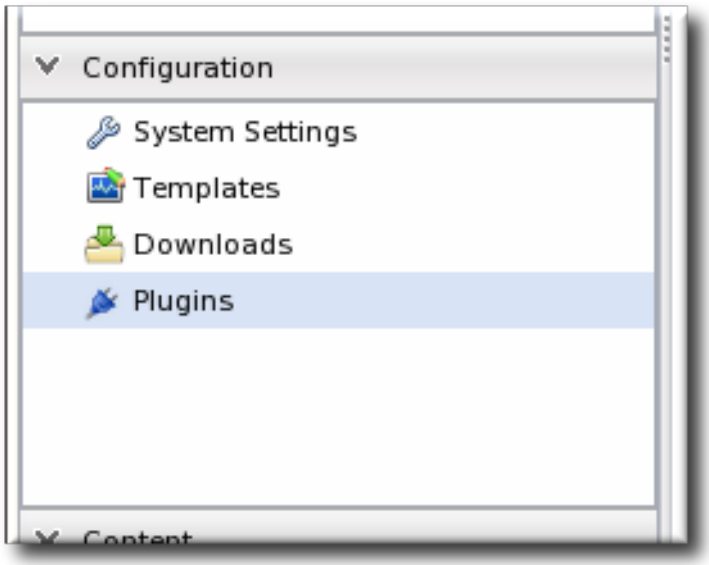


Note

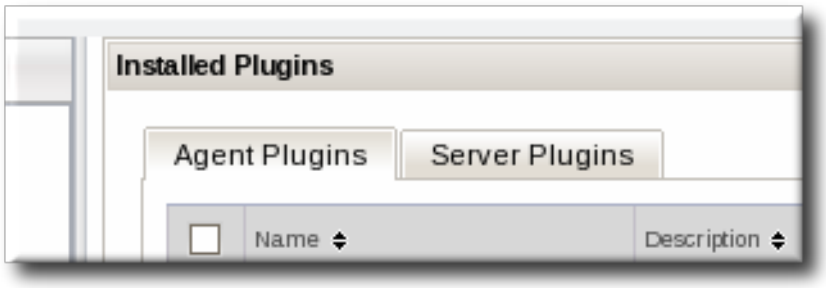
All other actions with server-side plug-ins occur *in the cloud*. When a new plug-in is added, it is added to the entire cloud; if it is undeployed, it is undeployed for the whole cloud. Plug-in containers, however, perform their tasks locally. Restarting the plug-in containers, then, restarts the master plug-in container for whichever JBoss ON server is hosting the web interface (in essence, the server which is local to the command) — and it only restarts that master plug-in container.

1. In the top menu, click the **`Administration`** tab.

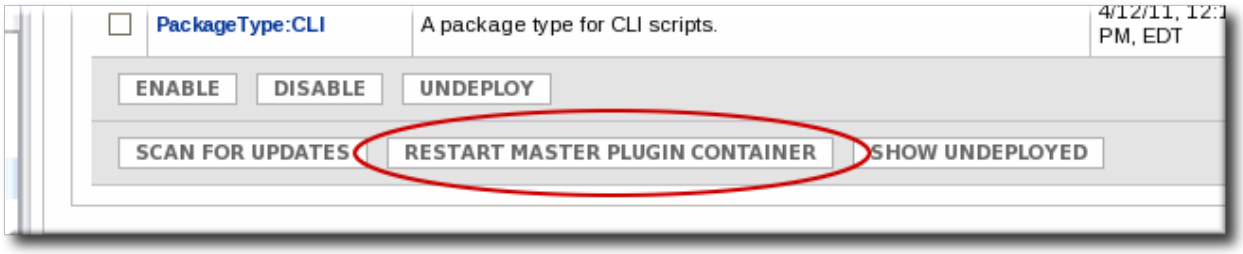
2. In the **`Configuration`** box on the left navigation bar, click the **`Plugins`** link.



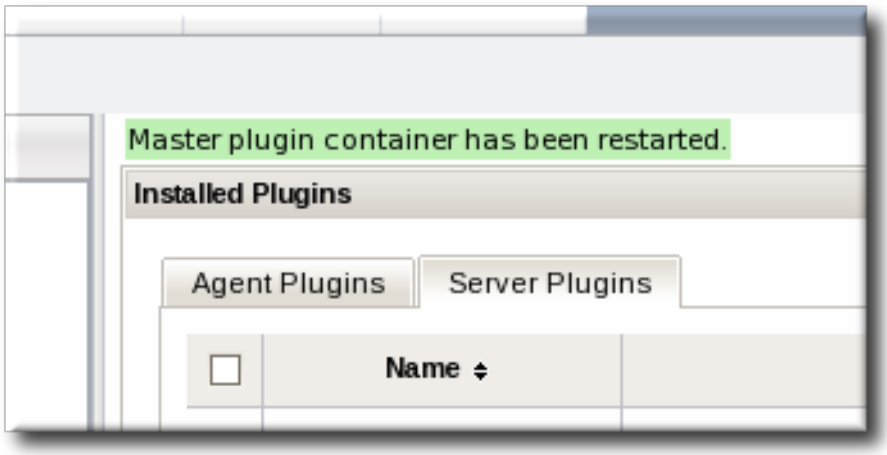
3. Click the **`Server Plugins`** tab.



4. Scroll to the bottom of the table, and click the **`RESTART MASTER PLUGIN CONTAINER`** button.



5. When the restart process is done (and assuming no problems were encountered), then there will be a success message in the upper left corner.



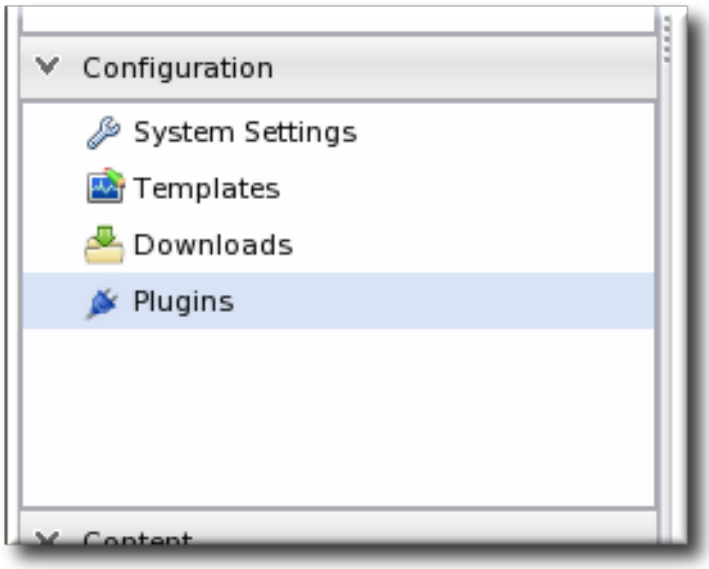
3.8. Setting Plug-in Configuration Properties

A few default server-side plug-ins and many custom plug-ins allow administrators to define specific configuration properties for the plug-in instance. The available properties are defined in the plug-in's **`rhq-plugin.xml`** file, and the values are then supplied in the JBoss ON UI.

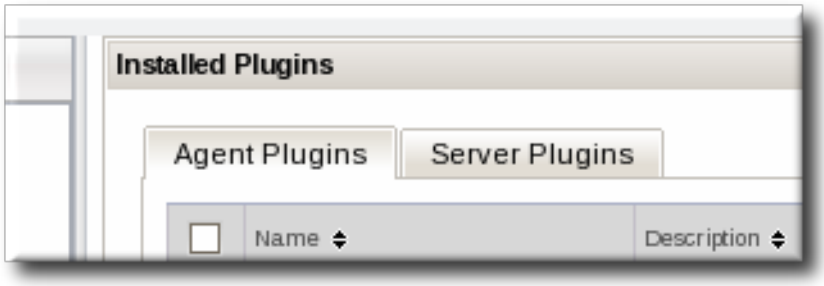
The descriptor file can define certain configuration parameters that apply to every instance of that plug-in (the descriptor parameters are described in Section 2.2.1, “Descriptor and Configuration” and Section 4.2.1, “Descriptor and Configuration”). The descriptor can set default values to use or can leave these fields blank. Either way, the global plug-in configuration parameters can be set or changed in the JBoss ON web UI.

1. In the top menu, click the **`Administration`** tab.

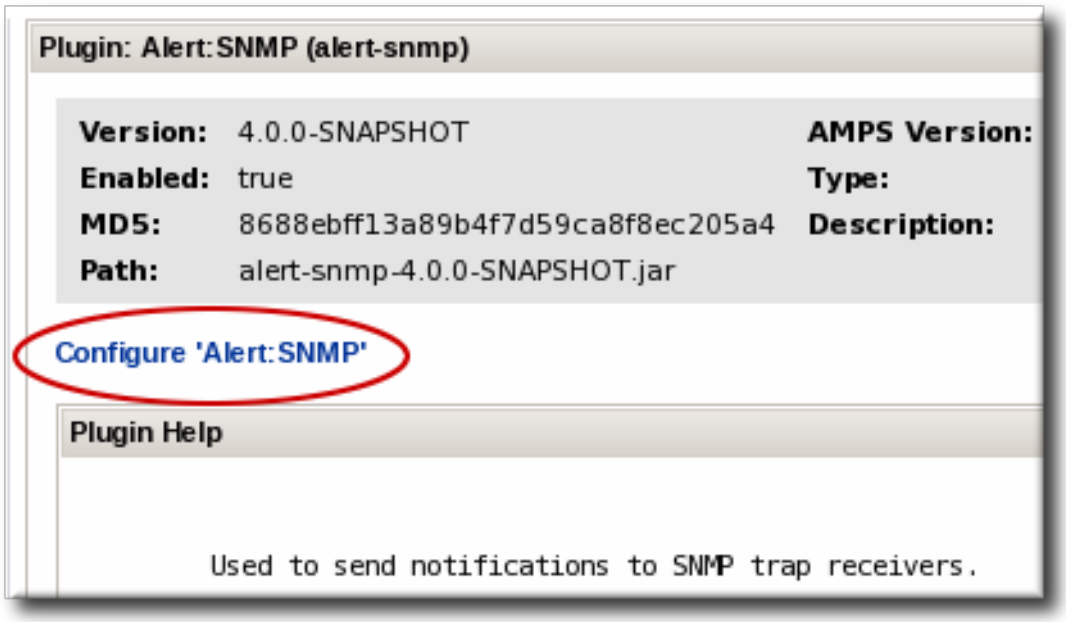
2. In the **`Configuration`** box on the left navigation bar, click the **`Plugins`** link.



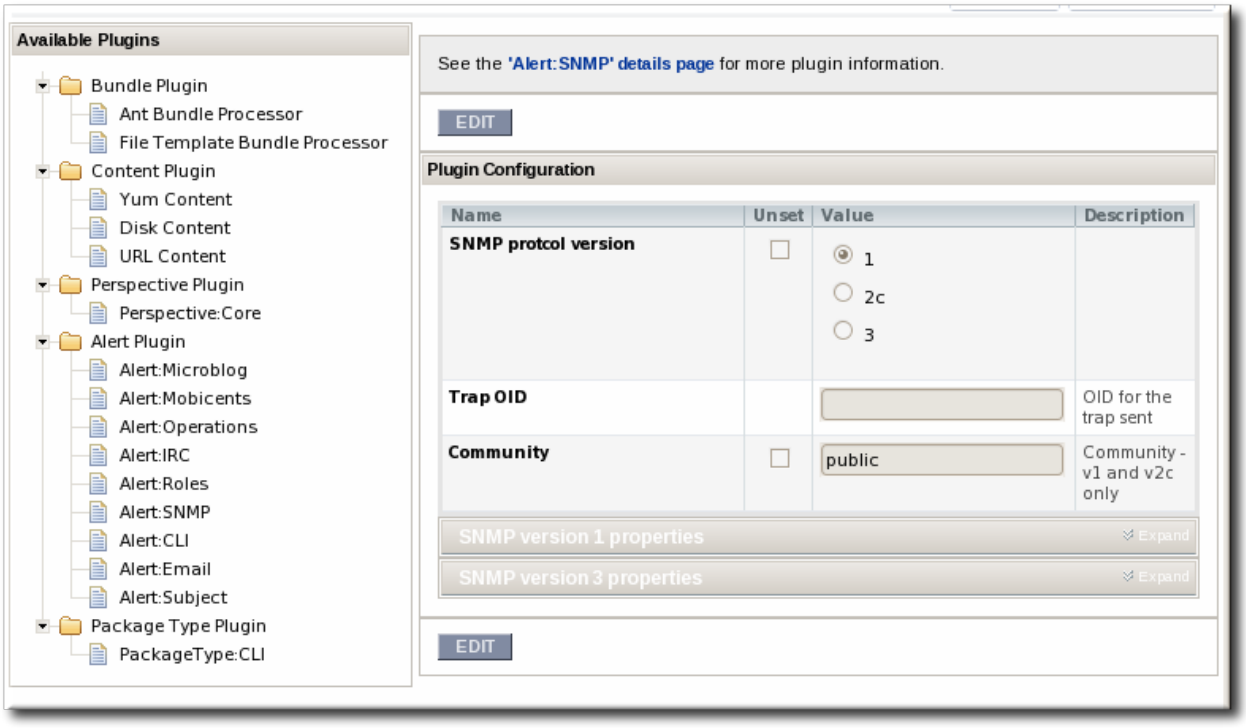
3. Click the **`Server Plugins`** tab.



4. Click the name of the server-side plug-in in the table.
5. In the middle of the plug-in details page, click the **`Configure 'Name'`** link.



6. All of the editable global configuration settings for the plug-in are listed in the **`Plugin Configuration`** table. To make any changes, click the **`EDIT`** button.



7. Unselect the **`Unset`** checkbox to activate the field to be edited.

8. Make any changes, and click **SAVE**.

4. Writing Agent Plug-ins: Background

Agent plug-ins extend the control capabilities that agents have as they interact with resources. This can include adding monitoring, operations, or configuration over resources.

4.1. About the Advanced Management Plug-in System (AMPS) for Agent Plug-ins

Agent resource plug-ins in JBoss ON have a certain pattern or system to how they are written. This is called the *Advanced Management Plug-in System* (AMPS). AMPS is defined in the core API for JBoss ON.

Basically, an agent plug-in is comprised of five parts (which, collectively, are the AMPS system):

- *The agent's plug-in container.* The plug-in contain runs inside the JBoss ON agent and it provides a manager for all of the deployed resource plug-ins.

  The plug-in container is what actually manages the lifecycle of the resource plug-ins. The agent starts the plug-in container, and the plug-in container starts the resource plug-ins. The plug-in container also handles all the classloading, threading, and running for resource plug-ins.

  Plug-in developers never need to interact with the plug-in container. As long as a plug-in is written with the appriproate components and with a valid plug-in descriptor, the agent will be able to manage the resource.

- *Domain objects*. This defines the individual objects for plug-ins, specifically resources, resource types, and configuration. All of the other elements in AMPS use the domain objects to define resource elements.

  One of the largest API sets within the domain object is *configuration*. The configuration API is used anywhere that a set of configuration properties is required, from plug-in configuration settings to connect to a resource to operation arguments.

- *The plug-in components*. These components define the actual component interfaces that are used by agent plug-ins, well as the *facets* that plug-ins can support.

  The plug-in components are the public API.

  This element within AMPS is the part that plug-in writers use. This contains the interfaces that plug-in writers implement in the resource plug-in.

- *Native System.* A lot of information require to monitor or manage a resource is available from the operating system information. The native system provides JNI or native access to that operating system information and can pull information from the process table, run external programs, or gather system metrics.

- *Resource plug-ins.*. JBoss ON has a set of resource plug-ins already defined. Each individual resource plug-in manages a particular product (applications and servers, services, or platforms). These plug-ins are loaded into the agent's plug-in container and implement the plug-in components defined in the API.

Note

Agent plug-ins can leverage the JBoss ON domain and native system APIs to define objects and communication layers, respectively.

4.2. The Breakdown of Agent Plug-in Configuration

Agents do not have a fixed functionality; their functions, from monitoring to the resources they can inventory, are defined by their plug-ins.

At its core, an agent plug-in consists of a single JAR file and an XML plug-in descriptor file (**`rhq-plugin.xml`** inside the **`META-INF/`** directory).

Along with the plug-in descriptor, an agent plug-in has up to three different types of Java files for each plug-in defined in the JAR file package:

- A plug-in component file that contains all of the code for the plug-in functionality
- A **`*Discovery.java`** file that configures the discovery process for the resources defined in the plug-in
- A **`*EventPoller.java`** that defines the events that can be collected by the resource

The definitions in the Java files closely track the configuration for the plug-in in the **`rhq-plugin.xml`** plug-in descriptor.

Note

Generating an agent plug-in template with the plug-in generator creates files with TODO markers to help guide writing the plug-in.

4.2.1. Descriptor and Configuration

A plug-in descriptor contains the metadata that described everything about the plug-ina dn the resource it configured. The descriptor describes the type and interactions of the resource type the agent plug-in defines.

The plug-in descriptor contains the information about name of the resource, the supported resource versions, the total resource hierarchy (meaning parents or children of the resource), configuration properties that the agent uses to connect to the resource, and all of the monitoring metrics, operations, and events related to the resource that can be managed by the agent.

The plug-in descriptor also contains information about the plug-in itself.

A resource definition in a plug-in descriptor can be a platform, server, or service. Multiple resources can be defined in a single plug-in descriptor; one resource is the root (parent) element, and the rest of the resources defines are its children.

Because the plug-in descriptor is an XML file, it follows a clear and structured schema definition. The schema definitions are what allow the plug-in descriptor to expose the resource's management interfaces to JBoss ON.

The plug-in descriptor, at a minimum, defines the resource type. Past that, it defines the different aspects of the resource that JBoss ON can manage:

- The names of the resource types (servers and services) supported by the plug-in
- Any configuration settings that the agent's plug-in components use to connect to the resource
- Any metrics (measurement definitions) to use to monitor the resource; this depends on the type of data issued by the resource itself.
- A set of operations that can be invoked on the resource. This is commonly start and stop operations, but it can include appliation-specific operations or other actions, like running a script.
- Resource configuration values that can be edited in the actual configuration of the resource.

  The plug-in configuration tells the components how to connect to the resource. The resource configuration, on the other hand, are settings in the resource itself that can be edited externally.
- Any child resources that are part of the resource hierarchy. For example, a JBoss server has data source services running within them, so the data source services are defined in the JBoss server resource plug-in, as a child resource of the JBoss server.

4.2.1.1. Resource Type, Metadata, and Plug-in Configuration

The top element in an agent plug-in is the **<plugin>** element.

```
<plugin name="JMX"
        displayName="Generic JMX"
        package="org.rhq.plugins.jmx"
        description="Supports management of JMX MBean Servers via various
remoting systems."
        ampsVersion="2.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns="urn:xmlns:rhq-plugin"
        xmlns:c="urn:xmlns:rhq-configuration">
```

Several attributes on this element are important:

- **name** and **displayName** give the internal and GUI name of the plug-in.
- **ampsVersion** gives the version number of the plug-in itself.
- **package** gives the name of the classes used by the components in the plug-in.

The next element in the plug-in descriptor defines the root resource that is defined by the plug-in. This can be **<platform>**, **<server>**, or **<service>**.

One or multiple resources can be defined in a plug-in descriptor. The plug-in descriptor not only defines those resource types, but it organizes those types in a parent-child hierarchy. For example, a JBoss EJB service only runs inside of a JBoss aerver, so the EJB service resource type should logically be a child type of the JBoss server resource type.

The hierarchy is defined by nesting **<service>** resource definitions inside **<server>** (or other **<service>**) resource definitions.

```
   <server name="JMX Server" discovery="JMXDiscoveryComponent"
class="JMXServerComponent"
           description="Generic JMX Server"
    supportsManualAdd="true" createDeletePolicy="neither">
```

Important

Do not rename resource types when you edit the resource plug-in. This breaks backward compatibility with any resource that was inventoried using the older version of the plug-in.

Two of the attributes are related to the Java resource files associated with the plug-in:

- **discovery** identifies the discovery component used to identify the resource type.
- **class** identifies the plug-in component which contains the actual code of the plug-in.

Two of the attributes define how resources of that type are added to the inventory:

- **supportsManualAdd** allows resources to be added to the inventory by administrators.
- **createDeletePolicy** sets whether children can be added or removed manually from inventory.

The last part that relates to configuring the plug-in is setting (optional) plug-in configuration properties. These are flexible and can define anything related to the information required to identify or set up a resource which matches the resource type in the plug-in, even setting constraints on allowed values or templates to define default settings.

```
<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>
```

The port has a constraint so, the GUI can validate the input being between 0 and 65535. The protocol can be selected from a list of drop-down menu items, with a default value of HTTP.

There are three types of properties:

- **<simple-property>**, which defines a one key-value pair
- **<map-property>**, which defines multiple key-value pairs related to a single entity, following the **java.util.Map** concept
- **<list-property>**, which contains a list of properties

Both **<map-property>** and **<list-property>** define groups of **<simple-property>** element. Additionally, these properties can be formally grouped together under **<group>** element. Using a **<group>** element creates a collapsible configuration ares in the UI.

Templates can preset some configuration properties.

```
         <c:template name="JDK 5" description="Connect to JDK 5">
            <c:simple-property name="type"
default="org.mc4j.ems.connection.support.metadata.J2SE5ConnectionTypeDescrip
tor"/>
            <c:simple-property name="connectorAddress"
default="service:jmx:rmi:///jndi/rmi://localhost:8999/jmxrmi"/>
         </c:template>
```

4.2.1.2. Discovery and Process Scans

The central functionality of JBoss ON is the inventory. Each resource must be present in that inventory, so the plug-in descriptor has to define how resources are detected and how they are added to inventory. This is done through the discovery component.

The **<plugin>** element has a **discovery** attribute which identifies the discovery Java file for the resource plug-in. (If there are multiple resources defined in the plug-in, then there will be multiple discovery components.)

When the agent plug-in is generated using the plug-in generator, it creates the appropriate template to add the discovery requirements. The discovery component has to have the information to find a resource instance and to assign a unique identifier to that resource.

```
/**
 * Discovery class
 */
public class testDiscovery implements ResourceDiscoveryComponent
,ManualAddFacet
{

    private final Log log = LogFactory.getLog(this.getClass());

      /**
       * Do the manual add of this one resource
       */
```

```
      public DiscoveredResourceDetails discoverResource(Configuration
pluginConfiguration, ResourceDiscoveryContext context) throws
InvalidPluginConfigurationException {

            // TODO implement this
            DiscoveredResourceDetails detail = null; // new
DiscoveredResourceDetails(
//                context.getResourceType(), // ResourceType
//            );

            return detail;
      }
}
```

Identifying resource instances is a critical part of maintaining the inventory. Any resource must be unique and must be consistently identified between discovery scans . Agents identify resources by constructing a unique *resource key*. A resource key varies by resource type. Whatever the key is, it has to intrinsic to the resources and detectable for the agent. Keys do not have to be globally unique, but they must be unique beneath a parent resource. For a JMX server, the resource key is its connector address, which is unique to the instance.

```
    public DiscoveredResourceDetails discoverResource(Configuration
pluginConfig,

ResourceDiscoveryContext discoveryContext)
            throws InvalidPluginConfigurationException {
        // TODO: Connect to the remote JVM to verify the user-specified conn
props are valid, and if connecting
        // fails, throw an exception.
        String resourceKey =
pluginConfig.getSimpleValue(CONNECTOR_ADDRESS_CONFIG_PROPERTY, null);
        String connectionType = pluginConfig.getSimpleValue(CONNECTION_TYPE,
null);

        // TODO (ips, 09/04/09): We should connect to the remote JVM in
order to obtain its version.
        String version = null;

        DiscoveredResourceDetails resourceDetails = new
DiscoveredResourceDetails(discoveryContext.getResourceType(),
                resourceKey, "Java VM", version, connectionType + " [" +
resourceKey + "]", pluginConfig, null);
        return resourceDetails;
    }
```

Optional dependencies can affect discovery. Embedded plug-ins copy the child type from the source plug-in, and then use the embedded plug-in's discovery component to run the discovery. Injected plug-ins take a parent resource and cycle through all potential children resource types and run a discovery for each type, injecting the parent component into each child type's discovery method.

Often, resources are processes running on the local machine. The JBoss ON agent can query the process table to detect those local processes. This is first defined in the plug-in descriptor using a **<process-scan>** element and then implemented in the discovery component.

Each resource type defined in the plug-in descriptor can have a **<process-scan>** child element. The **<process-scan>** element itself is empty, but has two required attributes: **name** and **query**. **name** identifies the specific scan method. **query** is the attribute that does something. The **query** is a string written in Process Info Query Language (PIQL). This value is used to search for the process.

The sourceforge PIQL API docs provide much more information on the syntax of PIQL queries.

The basic format of a PIQL process scan query is a three-part term which looks for the process, some kind of identifying marker, and then the value to match.

```
process|attribute|match=value,arg|attribute|match=value
```

For a process scan for discovery, the scan will usually look for a process name or a PID file.

Looking for a process by name can require additional attributes to help narrow the search down to a specific type of resource or even a specific instance. For example, a JBoss AS instance has a process name that starts with *java* and an argument with the value *org.jboss.Main*. The **ps** information contains both of those attributes.

```
jsmith     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
```

The PIQL query, then attempts to match the process name, using the **basename** query attribute, with a matching argument, defined in the **arg** query attribute.

```
process|attribute|match=value,arg|attribute|match=value
   |       |       |____      |_    |____     |______
   |       |            |       |        |           |
   |       |            |       |        |           |
   |       |            |       |        |           |
   |       |            |       |        |           |
process|basename|match=^java.*,arg|org.jboss.Main|match=.*
```

Alternatively, the process scan query can match a pidfile, which is a simple way to identify processes. A PID-based PIQL query has a slightly simpler format:

```
process|attribute|match=value
```

For example:

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

After defining the scan query in the **rhq-plugin.xml** descriptor file, then the discovery component must be written to implement the scan and process results.

> **Example 9. Process Scan Method in the Discovery Component**

```
    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);
    }
```

4.2.1.3. Metrics (Monitoring) and Operations

The agent managed resources by gathering metrics (for monitoring) and by launching scheduled operations, or tasks, on the resource. These two areas are configured similarly in the plug-in.

Monitoring areas are configured through **<metric>** elements in the plug-in descriptor.

```
        <metric displayName="Bytes Sent"
                description="Shows the rate that data bytes are sent by the
Web service."
                property="Bytes Sent/sec"
                defaultOn="true"
                displayType="summary"
                measurementType="trendsup"
                units="bytes"/>
```

The most relevant attributes on that attribute relate to the monitoring property (which is partially defined by the resource itself).

- **property** identifies the resource monitoring property.
- **measurementType** sets the data type being collected.
- **units** sets the units of the thing being monitored.

In the plug-in Java component, the metric is first set up by pulling in the **MeasurementFacet**.

```
public class testComponent implements ResourceComponent
, MeasurementFacet
, OperationFacet
```

Then, there is an entry for monitoring in a **MeasurementReport**, with a **MeasurementScheduleRequest** entity for each type of monitoring data.

```
    public void getValues(MeasurementReport report,
Set<MeasurementScheduleRequest> metrics) throws Exception {

        String propertyBase = "\\Web Service(_Total)\\";
        Pdh pdh = new Pdh();
```

```
        for (MeasurementScheduleRequest request : metrics) {
            double value = pdh.getRawValue(propertyBase +
request.getName());
            report.addData(new MeasurementDataNumeric(request, value));
        }
    }
```

Similarly, operations are configured in the plug-in descriptor in an **<operation>** element, implemented in the plug-in Java component through an **OperationFacet**, and then invoked in a **OperationResult** method.

4.2.1.4. Events

Events for resources are essentially types of log messages that are recognized by the agent.

In the plug-in descriptor, an event is configured by a simple **<event>** element, no children, that identifies the logging area by name.

```
<event name="errorLogEntry" description="an entry in the error log file"/>
```

The event handling is handled through an **EventPoller** component. This can be in the larger plug-in Java component, but it is usually broken into a separate ***EventPoller.java** component. The way to implement event polling depends on the resource and the nature of its logging. One of the simplest ways is to call the **EventPoller()**, then define the event type and set how the event is polled.

```
    public PerfTestEventPoller(ResourceContext resourceContext) {
        this.resourceContext = resourceContext;
    }

    public String getEventType() {
        return PERFTEST_EVENT_TYPE;
    }

    public Set<Event> poll() {
        int count =
Integer.parseInt(System.getProperty(SYSPROP_EVENTS_COUNT, "1"));
        String severityString = System.getProperty(SYSPROP_EVENTS_SEVERITY,
EventSeverity.INFO.name());
        EventSeverity severity = EventSeverity.valueOf(severityString);
        Set<Event> events = new HashSet<Event>(count);
        for (int i = 0; i < count; i++) {
            Event event = new Event(PERFTEST_EVENT_TYPE, "source.loc",
System.currentTimeMillis(), severity, "event #"
                + i);
            events.add(event);
        }
        return events;
    }
```

4.2.1.5. Resource Configuration

Resources can have parameters or settings changed through the JBoss ON GUI, managed by the agent. Those properties that can be edited are defined in the plug-in descriptor in **<resource-configuration>** elements. These configuration elements follow the same conventions as the **<plugin-configuration>** elements. The properties are defined as **<simple-property>** elements and can be listed (for options),

mapped, or organized into groups that are collapsible sections in the UI.

```
         <resource-configuration>
            <c:group name="Attributes">
               <c:simple-property
                  name="appBase"
                  required="true"
                  readOnly="true"
                  description="The Application Base directory for this
virtual host." />
               <c:simple-property
                  name="autoDeploy"
                  type="boolean"
                  description="Does this host deploy new applications
dropped in appBase at runtime?" />
               <c:simple-property
                  name="deployOnStartup"
                  type="boolean"
                  description="Does this host deploy applications in
appBase at startup?" />
               <c:simple-property
                  name="deployXML"
                  displayName="Deploy XML"
                  type="boolean"
                  description="deploy Context XML config files?" />
               <c:simple-property
                  name="unpackWARs"
                  displayName="Unpack WARs"
                  type="boolean"
                  description="Does this Host automatically unpack deployed
WAR files?" />
               <c:simple-property
                  name="aliases"
                  required="false"
                  type="longString"
                  description="Aliases assigned to the Host. When editing,
each alias must be on a new line. Aliases are automatically lowercased." />
            </c:group>
         </resource-configuration>
```

The first thing that is defined in the plug-in Java component is the ability to load the current configuration.

```
    public Configuration loadResourceConfiguration() {
        Configuration configuration = super.loadResourceConfiguration();
        try {
            resetConfig(CONFIG_ALIASES, configuration);
        } catch (Exception e) {
            log.error("Failed to reset role property value", e);
        }

        return configuration;
    }
```

The second part in the plug-in component allows the agent to change the configuration properties.

```
    public void updateResourceConfiguration(ConfigurationUpdateReport
report) {
        Configuration reportConfiguration = report.getConfiguration();
        // reserve the new alias settings
        PropertySimple newAliases =
reportConfiguration.getSimple(CONFIG_ALIASES);
        // get the current alias settings
        resetConfig(CONFIG_ALIASES, reportConfiguration);
        PropertySimple currentAliases =
reportConfiguration.getSimple(CONFIG_ALIASES);
        // remove the aliases config from the report so they are ignored by
the mbean config processing
        reportConfiguration.remove(CONFIG_ALIASES);

        // perform standard processing on remaining config
        super.updateResourceConfiguration(report);

        // add back the aliases config so the report is complete
        reportConfiguration.put(newAliases);

        // if the mbean update failed, return now
        if (ConfigurationUpdateStatus.SUCCESS != report.getStatus()) {
            return;
        }

        // try updating the alias settings
        try {
            consolidateSettings(newAliases, currentAliases, "addAlias",
"removeAlias", "alias");
        } catch (Exception e) {
            newAliases.setErrorMessage(ThrowableUtil.getStackAsString(e));
            report.setErrorMessage("Failed setting resource configuration -
see property error messages for details");
            log.info("Failure setting Tomcat VHost aliases configuration
value", e);
        }

        // If all went well, persist the changes to the Tomcat server.xml
        try {
            storeConfig();
        } catch (Exception e) {
            report
                .setErrorMessage("Failed to persist configuration change.
Changes will not survive Tomcat restart unless a successful Store
Configuration operation is performed.");
        }
    }
```

4.2.2. Lifecycle Listeners

Some plug-ins need to perform some initialization immediately when they load and some cleanup when they unload. Global initialization and shutdown of a plug-in is performed by its *lifecycle listener*.

The **`org.rhq.core.pluginapi.plugin.PluginLifecycleListener`** class allocates global resources needed by plug-in components and cleans up those resources.

Each plug-in can optionally define one lifecycle listener by specifying a ***pluginLifecycleListener*** attribute in the top-level **<plugin>** element.

```
<plugin name="Apache"
  displayName="Apache HTTP Server"
  description="Management of Apache web servers"
  package="org.rhq.plugins.apache"
  pluginLifecycleListener="ApachePluginLifecycleListener"
  ...
```

4.2.3. Plug-in Dependencies: Defining Relationships Between Plug-ins

Agent plug-ins can have relationships with other agent plug-ins. These relationships created dependencies between plug-ins, where plug-ins are only operable when other plug-ins are loaded, that allow plug-ins to share classes, or that extend the resource hierarchy by adding additional parent or child resources to an existing resource definition.

A *parent plug-in* is one that has another plug-in depending on it. A *child plug-in* is a plug-in that depends on another plug-in.

Agent plug-ins have three ways of defining dependencies:

1. Required dependencies are set using the **<depends>** element. Just using **<depends>** means that the required plug-in must be loaded or the other plug-in will fail to load. Adding the **useClasses** attribute makes the classes and JAR files for the parent plug-in available to the child plug-in.
2. An injection plug-in dependency means that a root-level resource runs inside another resource type, and that parent resource is defined as a parent plug-in. This essentially adds a new child to an existing resource type.
3. An embedded plug-in dependency means that a new parent resource type is added for an existing child. This can allow the child to be extended to share the new parent's classloader (depending on both plug-ins' configuration) or simply expand discovery.

Important

Embedded and injection plug-in dependencies are mutually exclusive. They cannot be used in the same plug-in definition.

There is one interesting similarity in all of these plug-in dependency models: Metadata and class definitions flow in only one direction, from a parent plug-in to its dependent plug-in. Information cannot flow in the other direction.

4.2.3.1. Required Plug-in Dependencies

The **<depends>** element directly under the <plug-in> element defines a parent plug-in that the plug-in depends on and required to be loaded. The **<depends>** element is what specifies a required dependency. The plug-in will not deploy successfully, unless all **<depends>** plug-ins are also successfully deployed.

The **<depends>** element can pull in JARs from the parent plug-in by specifying the **useClasses** attribute. The **useClasses** option can be set for only one required dependency in a single plug-in descriptor. If no **<depends>** element has a **useClasses** attribute, the last **<depends>** element specified in the plug-in descriptor, by default, has its useClasses attribute to true.

The **<depends>** element is used if the plug-in needs access to another plug-in's classes or if the plug-in should only be deployed when another plug-in is also deployed.

Note

The embedded and injection plug-in dependencies are *optional* dependencies. If the specified plug-in isn't loaded, then the plug-in simply runs without loading those dependencies. To make an embedded or injection plug-in a *required* dependency, then set the embedded or injection plug-in as a required plug-in using the **<depends>** element, as well as the other configuration.

4.2.3.2. Embedded Plug-in Dependencies

An embedded plug-in dependency adds a new parent resource for an existing child resource. The dependent plug-in (for the new parent resource) depends on the child.

With an embedded plug-in dependency, the server or service definition can be a copy of a source resource type found in another plug-in. That copy is defined in the plug-in descriptor by setting the **sourcePlugin** and **sourceType** attributes on the resource elements. When a plug-in source is specified, the server or service is copied from the source resource type, which means it has the same metadata as the source, with the exception that the embedded server or service can override the discovery and resource classes and, potentially, have a different name.

An embedded plug-in is an optional dependency.

4.2.3.3. Injection Plug-in Dependencies

A root-level resource type in an agent plug-in can define parent resource types that it can run inside of. This essentially injects the resource type as a child type to another, existing resource. This is an *injection* plug-in dependency.

The injection plug-in dependency illustrates that a child resource type knows about its parent resource types, but the parents do not know about the child. Knowledge of plug-ins flow down, not up. A parent plug-in's type information is known to its child plug-in, but a parent plug-in does not know anything about the child plug-ins that depend on it.

An injection dependency is a list of allowed parents, within a **<runs-inside>** element. Each parent is an optional dependency.

```
<runs-inside>
   <parent-resource-type name="JMX Server" plugin="JMX" />
   <parent-resource-type name="JBoss Server" plugin="JBoss AS" />
</runs-inside>
```

4.2.4. Class Sharing Between Plug-ins

All agent plug-ins have their own classloader while running in the plug-in container. Each resource in inventory will be assigned a classloader, which can be the same as its plug-in classloader.

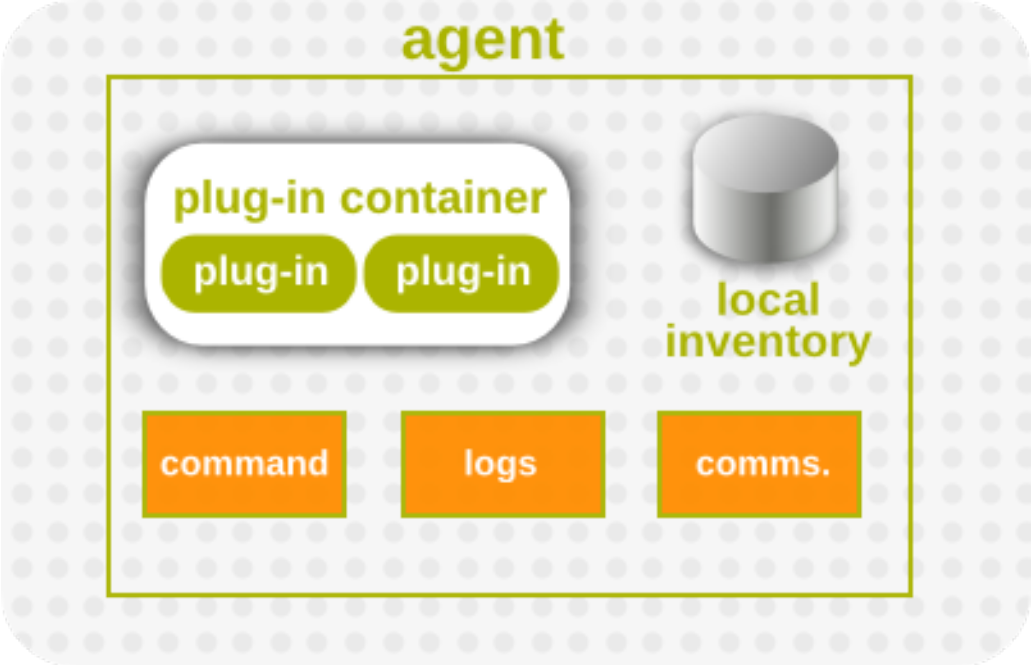


Figure 5. Agent Components, Together

All plug-in classloaders are isolated from one another, unless the **`<depends useClasses="">`** attribute is set to true. If a plug-in is a direct dependent of another plug-in, and that dependency is defined with <depends useClasses="true">, then that parent plug-in's JAR classes (and all of its parent JARs) are available to the dependent plug-in's classloader.

The most common reason to create that kind of dependency, and share JARs and classes between plug-ins, is because one resource defined in one plug-in is deployed and running in another resource defined in another plug-in. The child plug-in needs a way to connect to its parent resource to perform things like discovery and management of the child resource. By default, all managed resources are assigned a resource classloader that is shared. This classloader can belong to the plug-in or to the parent resource.

When a resource component needs to create a connection to its actual managed resource, that resource's classloader must be different than the normal, shared plug-in classloader. The resource usually needs to have the client JARs necessary to connect to the managed resource within its own classloader, and those client JARs are usually very specific to the version of the resource being managed. The connecting resource should be in charge of creating the connection, since it knows how to do it. When a resource requires its own connection classloader, specify the attribute **`classLoader="instance"`** on the resource type and make sure the resource type's discovery component implements the **`ClassLoaderFacet`** so it tells the plug-in container where any additional connection classes can be found for the specific version of the specific resource being managed.

In Example 10, “classLoader for Plug-in Z”, the Z1.server has a **`classLoader`** option set to shared. This means that Z1.server resources share their classloaders with their parent resources, and that classloader may be a resource classloader or a plug-in classloader. Every Z1.server resource uses the same classloader.

Example 10. classLoader for Plug-in Z

```
<plug-in name="Z">

<depends plugin="A" />

<server name="Z1.server" classLoader="shared">
```

```
   <runs-inside>
      <parent-resource-type name="B1.server" plugin="B"/>
      <parent-resource-type name="C1.server" plugin="C"/>
   </runs-inside>
</server>

<server name="Z2.server" sourcePlugin="D" sourceType="D1"
classLoader="instance">
</server>

<server name="Z3.server" classLoader="instance">
</server>

</plugin>
```

Normally, setting a **`classLoader`** option to instance means that each resource uses its own resource plug-in. However, for Z2.server, the Z2.server plug-in is extended by embedding the values for Plug-in D, so Z2.server resources share their classloaders with their parent plug-in.

Z3.server simply uses its own resource classloader because its **`classLoader`** option is set to instance and it has no injected or embedded dependencies. When the **`classLoader`** option is set to instance, the **`ResourceDiscoveryComponent`** implementation can optionally define a **`ClassLoaderFacet`** with a method (**`getAdditionalClasspathUrls`**) that returns a **`List<URL>`** pointing to additional JARs that should be placed in the resource's classloader. When the plug-in container needs to create a classloader for a resource, it checks if the resource's discovery component implements this facet, and, if so, it gets the additional classpath URLs and adds them to the resource classloader when it creates it.

If a resource type is defined with either an injection or embedded dependency, then its classloader depends on both its **`classLoader`** attribute value and its parent's **`classLoader`** attribute value.

Resource ClassLoader	Parent ClassLoader	ClassLoader Description
shared	shared	The **`useClasses`** value must be set to true so that the resource can access both its classes and the parent classes.
instance	shared	The resource primarily needs its own classes, but it may be beneficial for **`useclasses`** to be set to true to so that the child can use parent classes.
shared	instance	The resource uses only its own classloader.
instance	instance	The resource uses only its own classloader.

4.3. Extended Example: Content Types for Resources

All of the behavior for how a resource is managed in JBoss ON depends on how it is described in its resource plug-in for the agent. This includes all of the monitoring metrics that can be collected, any configuration that can be set, and any operations that can be performed. This also includes any content that can be deployed for that resource and what kind of content is expected.

A *package* refers to any piece of content. A package is usually a file of some kind, like a JBoss AS JAR file or an RPM for a Linux platform. However, since the package type for a resource is defined in the resource plug-in, a package can be anything, so long as the resource plug-in is configured to use it.

As with other elements related to the resource, the package type is defined in the plug-in descriptor for the resource type. Each package type can define certain attributes (listed in Table 9, “Package Attributes”), but every package must define its name and its type (of four given categories).

In the plug-in descriptor, package types are identified by **<content>** elements. The required properties are set as flags on the main **<content>** element; any configurable properties, which are set by the user when new packages are uploaded to the resource, are given in **<c:simple-property>** child elements. For example, this content element in the Platform Resource Plug-in identifies deployable (category) package types for Windows platforms:

```
      <content name="InstalledSoftware" displayName="Installed Software"
category="deployable" description="Installed Windows Software">
           <configuration>
               <c:simple-property name="Publisher"/>
               <c:simple-property name="Comments"/>
               <c:simple-property name="Contact"/>
               <c:simple-property name="HelpLink"/>
               <c:simple-property name="HelpTelephone"/>
               <c:simple-property name="InstallLocation"/>
               <c:simple-property name="InstallSource"/>
               <c:simple-property name="EstimatedSize" units="kilobytes"/>
           </configuration>
       </content>
```

While packages can be manually added to a resource, an agent can also actively *check* for new content and add any discovered content to its inventory. A package is inventoried in JBoss ON through a recurring *package discovery scan*. The interval at which this discovery occurs can be explicitly set in the package's definition in the plug-in's descriptor or it can use the default value given in the plug-in schema file.

Table 9. Package Attributes

Attribute	Description	Optional or Required
Name	The programmatic name of the package type.	Required
Display Name	A user interface friendly name of the package type.	Optional
Description	Describes the type of content found in packages of this type.	Optional
Category	One of four enumerated options: - Executable Script (which is potentially editable) - Executable Binary - Configuration (a configuration file for the resource) - Deployable	Required

Attribute	Description	Optional or Required
Discovery Interval	Defines the time between package discovery scans for this type; different package types can be configured with intervals to represent the likelihood of the package inventory changing.	Optional
Creation Type Flag	If set to true, a package of this type is used when creating resources of the enclosing resource type. An example of this situation is a Java EAR file. There is an EAR resource type that represents the enterprise application in JBoss ON. Under that resource type, there is a package type defined to represent the EAR file itself. This package type is flagged as a creation type; when creating a new EAR resource, the EAR file must be created at the same time. The default for this attribute is false, as packages will typically not represent the creation of a new resource.	Optional
Configuration	The configuration element allows the plug-in to define an open-ended set of attributes about the package type. These values will be populated during package discovery, and if not marked as read only, can be specified by the user at artifact creation time. An example of a property in this configuration element is a Boolean that describes if an EAR file is deployed as exploded or zipped. When EAR files are discovered, this flag will be populated and carry package type specified information. Additionally, when deploying a new EAR file through JBoss ON, this flag can be set to indicate how the package should be deployed on the AS instance.	Optional

4.4. Extended Example: HTTP Metrics

This example plug-in is written for the agent to connect to an HTTP server to monitor the web server itself for performance or availability. This plug-in performs two tasks:

- Issue a GET or HEAD request to the base URL for the given server.

- Collect both the HTTP return code and the response time as a resource trait.

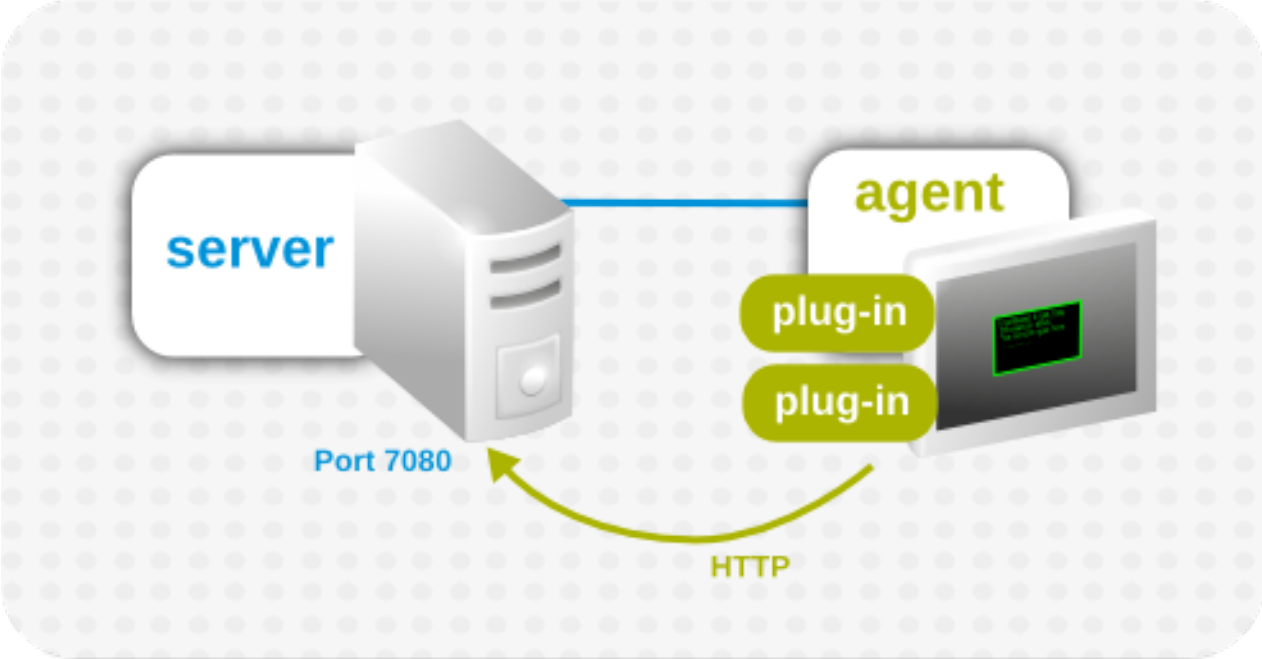


Figure 6. Basic Agent Plug-in Scenario

Note

For simplicity, this plug-in is written for an agent that is running on the same machine as a JBoss ON server.

The HTTP metrics plug-in is designed to run a server and a child service to collect two monitoring metrics. This requires two Java files which define the plug-in behavior and the behavior of a discovery component that the plug-in requires to locate URLs.

As with all agent plug-ins, this example also has a **`rhq-plugin.xml`** file as the plug-in descriptor, which is required for JBoss ON to recognize the plug-in configuration. The plug-in is built as a maven project, so it has a **`pom.xml`** file, although that is not a requirement, since any properly configured JAR file can be deployed as an agent plug-in.

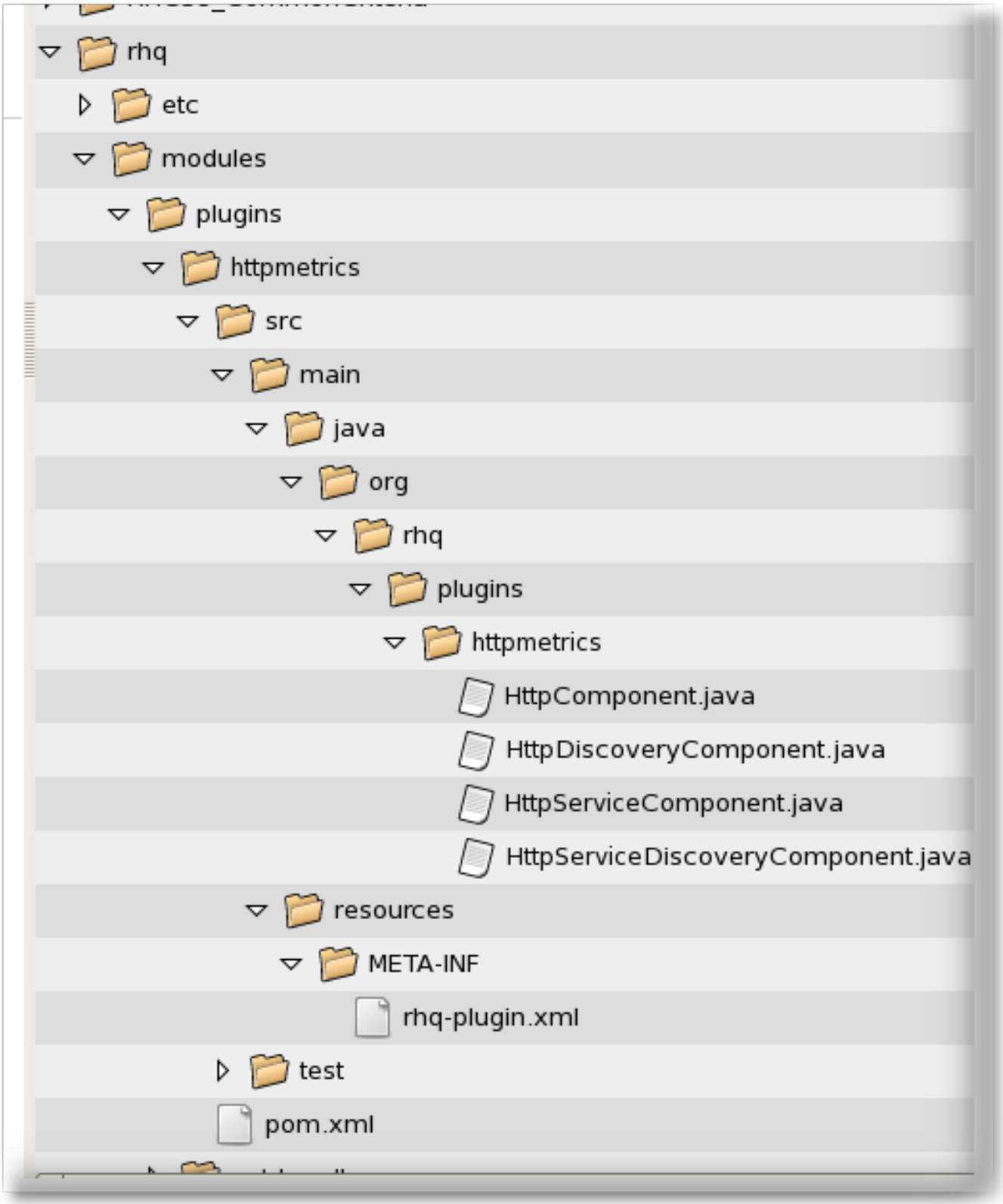


Figure 7. Directory Layout of an Agent Plug-in Project

4.4.1. Looking at the Plug-in Descriptor (rhq-plugin.xml)

The plug-in descriptor is where everything is glued together. The first part of the plug-in descriptor defines the basic information about the plug-in, like its name, version number, and schema.

> **Example 11. Basic Plug-in Information**

```
<?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 identifies the Java package for Java class names that are referenced in the plug-in configuration in the descriptor.

The HTTP Metrics plug-in is defines as a server which runs a child resource. Services cannot run on their own, so defining the HTTP Metrics resource as a server first allows multiple services to be launched off of it.

Example 12. Server Definition

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

The next step is to add a service element as a child of the server. To differentiate between the server and the service operations, the service has its own discovery and plug-in components, meaning its own **.java** files and classes. The **supportsManualAdd** option sells JBoss ON that the HTTP services can be added manually through the UI, which is important for administration.

Example 13. Service Definition

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

The middle of the **<service>** element defines the plug-in properties that are configured through the UI. This can be simple (setting a simple string for the URL).

Example 14. Simple Configuration Properties

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

The properties can be more advanced and allow specific values for the protocol, port, and hostname or IP address, depending on how much information needs to be configured.

Example 15. Complex Configuration Properties

```
<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>
```

The last part of the **<service>** element contains the metrics that are configured for the HTTP Metrics plug-in. The first metric, for the response time, collects a numeric dataType. The status metric collects a trait dataType. (JBoss ON is intelligent enough to only store changed traits to conserve space.)

Example 16. Defined Metrics

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

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

The attributes available to define the metric are defined in the agent plug-in XML schema. The attributes used in this example are listed in Table 10, “metric Attributes”.

Table 10. metric Attributes

Attribute	Description

Attribute	Description
property	Gives the unique name of this metric. The name can also be obtained in the code using the **`getName()`** call.
description	Gives a human readable description of the metric.
displayName	Gives the name that gets displayed in the JBoss ON UI.
dataType	Sets the type of metric, such as numeric or trait.
units	The measurement units to use for numerical dataType.
displayType	If the value is set to **`summary`**, the metric is displayed in the indicator charts and collected by default.
defaultOn	Sets whether the metric collected by default.
measurementType	Sets what characteristics the numerical values have. The options are trends up, trends down, or dynamic. For both trends metrics, the system automatically creates additional per minute metrics.

4.4.2. Looking at the Discovery Components (HttpDiscoveryComponent.java and HttpServiceDiscoveryComponent.java)

Two Java files define how to discover the HTTP metrics server and any URL defined to be monitored.

The first Java file, **`HttpDiscoveryComponent.java`**, discovers the HTTP metrics server. The discovery component is called by the InventoryManager in the agent to discover resources. This can be done by a process table scan, querying the MBeanServer, or other means. Whatever the method, the most important thing is that the discovery component returns 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()`**.

Basically, this retrieves a list of resources discovered by the process scan and creates the details about the discovered resource. Using ProcessInfo gets more information about the process and can be used to exclude certain types of discovered resources from the final list.

Example 17. HttpDiscoveryComponent.java

```
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;
    }
```

The service discovery component (defined in **`HttpServiceDiscoveryComponent.java`**) relies on information passed through the GUI to configure its resources, rather than a discovery scan. The initial definition in the Java file is similar to the one for the server discovery, but this definition has an additional **`List<Configuration> childConfigs`** which processes the information that is passed through the UI. This pulls the information for the required **`url`** information supplied by the user.

Example 18. Service Discovery

```
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();

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

The list of URLs that are configured are processed and added as a resource with the resource name, description, type, and (critically) the unique resource key that is used by the discovery process.

Example 19. Listing HTTP URL Resources

```
            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;
```

```
    }
```

4.4.3. Looking at the Plug-in Components (HttpComponent.java and HttpServiceComponent.java)

The plug-in component is the part of the plug-in that does the work after the discovery has finished.

For the server component (**`HttpComponent.java`**), the plug-in is pretty simple. The component only implements placeholder methods from the ResourceComponent interface to set the server availability. Setting the availability to UP automatically allows the resource component to start.

Example 20. Server Availability After Discovery

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

The service component (**`HttpServiceComponent.java`**) is more complex because it must carry out the operations defined in the plug-in descriptor.

Each of the basic functions in the plug-in descriptor is implemented through an appropriate agent facet. All of the gent facets are listed in Section 6.2, “Plug-in Facets”. The HTTP metrics component specifically maps the **`<metric>`** element in the descriptor to the **`MeasurementFacet`**.

Each facet has its own methods to implement. For monitoring and other operations that require processing metrics, the **`MeasurementFacet`** implements the following method:

```
getValues(MeasurementReport report, Set metrics)
```

The **`MeasurementReport`** passed in is where the monitoring results are added. The **`metrics`** value is a list of metrics for which data should be gathered. All of this information can be defined in the **`<metrics>`** element or in the UI configuration.

So the next part is the plug-in component to do 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
```

To monitor things, the **`getValues()`** method from the **`MeasurementFacet`** must be implemented, but that's not the first step to take. A resource cannot be discovered if the resource is down, so the first step is to set a start value to start the service from **`ResourceContext`** and give it an availability of UP.

Example 21. Service Resource Availability

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

```
        url = new URL(context.getResourceKey());
        // Provide an initial status, so
        // getAvailability() returns up
        status = "200";
    }
```

Once the service is started, then the **`getValues()`** can be implemented. This actually collects the monitoring data from the given URLs.

Example 22. Implementing getValues()

```
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));
            }
        }
    }
```

The final step is to process the information. Implementing the **`getData()`** method in the **`MeasurementFacet`** loops the incoming request to see which metric is wanted and then to supply the collected value. Depending on the type of data, the data may be to be wrapped into the correct **`MeasurementData*`** class.

Example 23. Implementing getData()

```
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);
    }
```

This implementation for the HTTP Metrics plug-in is very simple. It opens a URL connection, takes the time it takes to connect, and gets the status code. That's all.

4.5. Examples: Embedded and Injected Plug-in Dependencies

JBoss ON agent plug-ins have several different ways of defining dependencies between plug-ins: simple depends, embedded, and injected. The way that a dependency is defined has some affect on how the plug-in behaves. This is all explored more in Section 4.2.3, “Plug-in Dependencies: Defining Relationships Between Plug-ins”.

These examples show how each of the dependency types are defined in the plug-in descriptors for agent plug-ins.

4.5.1. Simple Dependency: JBoss AS and JMX Plug-ins

A required dependency is defined using the **<depends>** tag. This means that the required plug-in has to be deployed successfully before the plug-in which requires it can be deployed. A simple example of this is JBoss AS, which has a JMX server running inside it. The JBoss AS plug-in for JBoss ON, then, sets a dependency on the JMX plug-in.

The JMX plug-in descriptor simply defines the configuration for the JMX server.

Example 24. JMX Plug-in Descriptor

```
<plugin name="JMX">
   <server name="JMX Server" discovery="JMXDiscoveryComponent"
class="JMXServerComponent">
   ...
   </server>
</plugin>
```

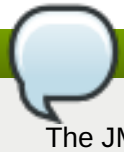
The JBoss AS plug-in descriptor lists the JMX plug-in as a dependency. This makes all of the JMX plug-in classes available to the JBoss AS plug-in classloader (since the **useClasses** argument is set to **true**), but the JBoss AS plug-in descriptor does not actually define or use any source types related to or referencing the JMX plug-in.

Example 25. JBoss AS Plug-in Descriptor

```
<plugin name="JBossAS">
   <depends plugin="JMX" useClasses="true"/>
   <server name="JBossAS Server" discovery="JBossASDiscoveryComponent"
class="JBossASServerComponent">
   ...
   </server>
</plugin>
```

Important

Plug-ins can require or depend on multiple plug-ins. However, only one of those dependencies can have the value of **`useClasses=true`**.



Note

The JMX plug-in is a generic plug-in and can be used by many other plug-ins. Any application that can be managed by JMX can use the JMX plug-in to pick up all of its dependencies, as well as EMS libraries which extend the JMX plug-in.

4.5.2. Embedded Dependency: JVM MBeanServer and JBoss AS

Java Virtual Machines have a built-in JMX MBeanServer called the *platform MBeanServer*. Any JVM with this platform MBeanServer can be monitored for memory, garbage collectors, threading, and other subsystems through MBeans in the MBeanServer.

The JMX plug-in in JBoss ON can define resource types for each platform MBean, allowing the JBoss ON agent to monitor those MBeans as JBoss ON service resources.

The platform MBeanServer can be found in a standalone JVM process or embedded inside a JBoss AS VM process. If the JVM being monitored is embedded in a JBoss server, then the JBoss ON plug-ins are configured with an embedded plug-in dependency. An *embedded dependency* means that one plug-in is aware that another plug-in is running inside it.

The JMX plug-in descriptor simply defines the JMX server.

Example 26. JMX Plug-in Descriptor

```
<plugin name="JMX">
   <server name="JMX Server" discovery="JMXDiscoveryComponent"
class="JMXServerComponent">
      <service name="VM Memory System"
               discovery="MBeanResourceDiscoveryComponent"
               class="MBeanResourceComponent"
               description="The memory system of the Java virtual
machine">
      ...
      </service>
   ...
   </server>
</plugin>
```

The embedding is done in the JBoss AS plug-in descriptor. Along with setting a required dependency for the JMX plug-in, the JBoss AS plug-in's **`<server>`** definition pulls in the **`sourcePlugin`** and **`sourceType`** attributes. The reason for this is to run a second JMX discovery scan, this one using the **`org.rhq.plugins.jmx.EmbeddedJMXServerDiscoveryComponent`** class to run a special discovery

scan looking for a JVM embedded in a JBoss AS instance. The **`sourcePlugin`** and **`sourceType`** attributes, then, copy the resource type and give it a unique name so that any embedded JVMs are treated as different resource types than standalone JVMs.

Example 27. JBoss AS Plug-in Descriptor

```
<plugin name="JBossAS">
   <depends plugin="JMX" useClasses="true"/>
   <server name="JBossAS Server" discovery="JBossASDiscoveryComponent"
class="JBossASServerComponent">
      <server name="JBoss AS JVM"
              description="JVM of the JBossAS"
              sourcePlugin="JMX"
              sourceType="JMX Server"

discovery="org.rhq.plugins.jmx.EmbeddedJMXServerDiscoveryComponent"
              class="org.rhq.plugins.jmx.JMXServerComponent">
      ...
      </server>
   ...
   </server>
</plugin>
```

This type of embedded plug-in also illustrates that an embedded resource type can be discovered using a different discovery component from that of the source plug-in type.

4.5.3. Injected Dependency: Hibernate with JVM and JBoss AS

An *injection dependency* is the logical opposite of an embedded dependency; it is an awareness in the plug-in that the resource configured by the plug-in is running inside another resource. The parent resources are listed as dependencies.

One common example of this is Hibernate. Hibernate can run in either a standalone J2SE JVM instance or a JBoss AS server; for this example, it runs inside a JVM in a JBoss AS instance. The plug-ins have chained dependencies, where either the JMX and JBoss AS plug-ins can be parents to the Hibernate plug-in, and both the Hibernate and JBoss AS plug-ins list the JMX plug-in as a required dependency.

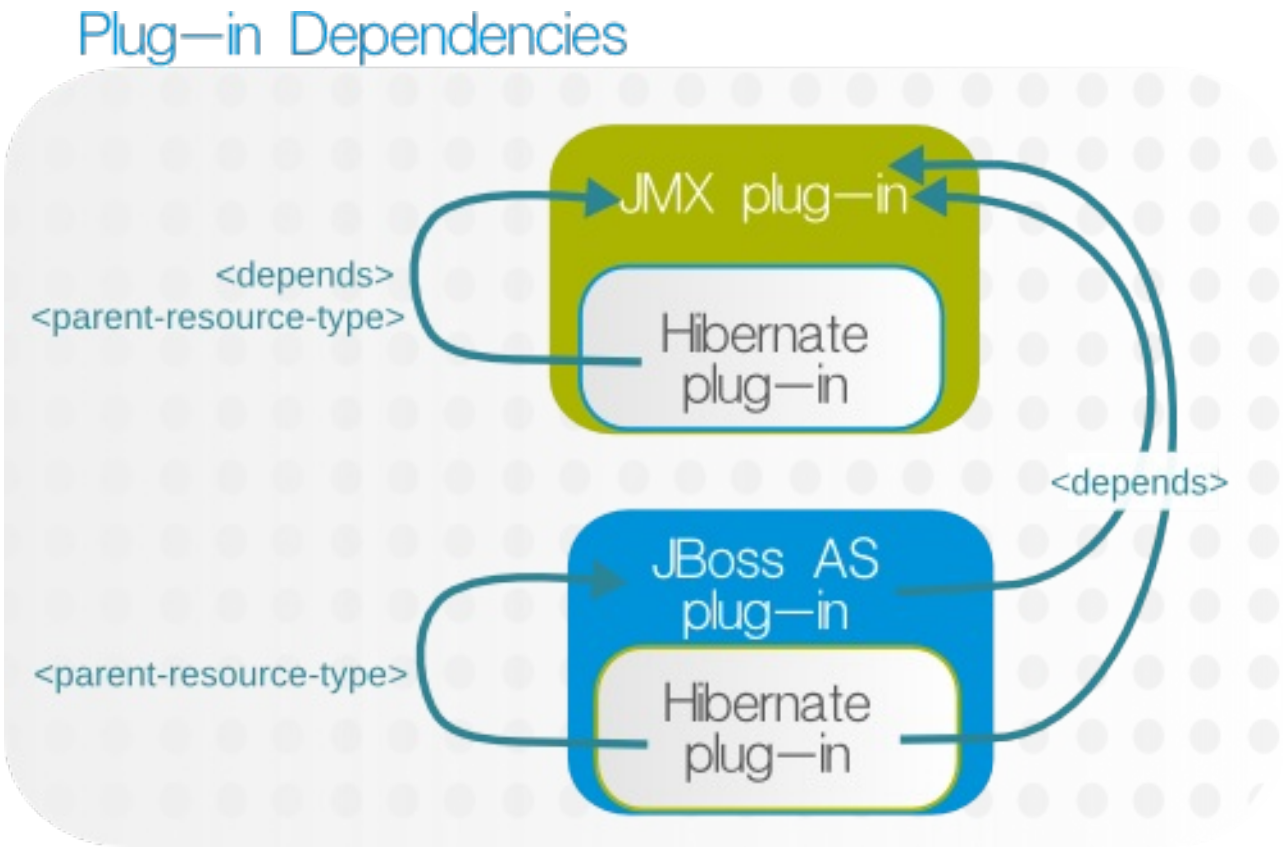


Figure 8. Hibernate, JMX, and JBoss AS Dependencies

As before, the JMX plug-in descriptor only defines the JMX plug-in, without any dependencies.

Example 28. JMX Plug-in Descriptor

```
<plugin name="JMX">
   <server name="JMX Server" discovery="JMXDiscoveryComponent"
class="JMXServerComponent">
   ...
   </server>
</plugin>
```

The JBoss AS plug-in sets a required dependency on the JMX plug-in, but no other dependencies are required to be defined (although they could be).

Example 29. JBoss AS Plug-in Descriptor

```
<plugin name="JBoss AS">
   <depends plugin="JMX" useClasses="true"/>
   <server name="JBossAS Server" discovery="JBossASDiscoveryComponent"
class="JBossASServerComponent">
   ...
   </server>
</plugin>
```

The most complex definition is for the Hibernate plug-in. This sets an explicit dependency on the JMX plug-in

using the **<depends>** element. The Hibernate plug-in then defines what resource types *could* operate as its parents by running a discovery scan (specifically for the Hibernate Statistics resource) against potential parent types. The list of parent resource types is contained in the **<runs-inside>** element, and each potential parent is identified by name and plug-in type in **<parent-resource-type>** elements.

Example 30. Hibernate Plug-in Descriptor

```
   <depends plugin="JMX" useClasses="true"/>
   <service name="Hibernate Statistics"

discovery="org.rhq.plugins.jmx.MBeanResourceDiscoveryComponent"
class="StatisticsComponent">
     <runs-inside>
         <parent-resource-type name="JMX Server" plugin="JMX"/>
         <parent-resource-type name="JBossAS Server" plugin="JBossAS"/>
      </runs-inside>
   ...
   </service>
</plugin>
```

When a plug-in is dependent on another plug-in, it is implicitly dependent on whatever other plug-ins are required. For example, Hibernate depends on the JBoss AS plug-in. Even if the Hibernate plug-in didn't explicitly state a dependency on the JMX plug-in, it would still be dependent on the JMX Plug-in because the JBoss AS plug-in requires it.

4.6. Extended Example: Drift Monitoring

Drift monitoring is allowed for a resource by defining a default drift definition in the plug-in. In the plug-in configuration, the drift definition creates a default template that indicates that a resource supports drift monitoring. (Additional templates can be created by users or the default template can be revised when it is applied to a resource.)

At its most basic, the drift definition sets a target location for the drift system to monitor. This location can be identified from several different configuration areas for the resource:

- fileSystem, which is any directory on the machine local to the resource
- pluginConfiguration, which is defined property in the resource plug-in, like a home directory
- resourceConfiguration, a resource configuration property
- measurementTrait, a trait that is gathered about the resource

This target location is the *base directory*. The element which identifies where to find the value for the base directory the *value name*, while the actual value is the *context*. For example, for a base directory of **/etc/**, the elements in the drift definition are:

```
Value name: fileSystem
Value context: /etc
```

The most basic drift definition only needs to define the value name and context.

Example 31. Base Directory Only

```
     <drift-definition name="Template-File System"
                       description="Monitor the file system for drift.
Definitions should set a more specific base directory as the file system
root is not recommended.">
         <basedir>
             <value-context>fileSystem</value-context>
             <value-name>/</value-name>
         </basedir>
     </drift-definition>
```

It is possible to set more information, like subdirectories or file types which should be explicitly included or excluded. These are additional *paths*, beneath the base directory, and file types can be identified by *patterns*.

If a directory or file type is explicitly included, then all other files and directories are implicitly excluded (and vice versa, if something is explicitly excluded, everything else is implicitly included). There can be multiple paths and patterns defined.

Example 32. Included Paths and Patterns

```
<drift-definition name="Template-Base Files"
                  description="Monitor base application server files for
drift. It defines monitoring for some standard sub-directories of the HOME
directory.  Note, it is not recommeded to monitor all files for an
application server. There are many files, and many temp files.">
    <basedir>
         <value-context>pluginConfiguration</value-context>
         <value-name>homeDir</value-name>
    </basedir>
    <includes>
 <include path="bin" pattern="*/*.sh" />
        <include path="lib" />
        <include path="client" />
    </includes>
</drift-definition>
```

Note

Multiple drift definitions can be defined for a resource type, but each drift definition defined only a single base directory to monitor for drift.

4.7. Extended Example: Provisioning and Content Deployments (Bundles)

Allowing content to be deployed to a resource using the bundle system is enabled by defining allowed target locations for content to be provisioned to.

As with drift configuration, bundle configuration identifies the target location based on information in one of four areas:

- fileSystem, which is any directory on the machine local to the resource

- pluginConfiguration, which is defined property in the resource plug-in, like a home directory
- resourceConfiguration, a resource configuration property
- measurementTrait, a trait that is gathered about the resource

That area is the *value name* in the bundle definition. The actual value is the *value context*.

Example 33. A Single Bundle Base Directory

```
     <bundle-target>
         <destination-base-dir name="Root File System" description="The
top root directory on the platform (/)" >
           <value-context>fileSystem</value-context>
           <value-name>/</value-name>
         </destination-base-dir>
     </bundle-target>
```

The potential target locations, the **<destination-base-dir>**, are presented to users as options when they are provisioning a bundle. Users can deploy a bundle to any, user-defined directory *beneath* that base directory, but they cannot deploy to a location *outside* that directory. If users will reasonably want to provision content to multiple directories, then each directory needs to be added to the **<bundle-target>** definition.

Example 34. Multiple Bundle Base Directories

```
<bundle-target>
    <destination-base-dir name="Install Directory" description="The top
directory where the JBossAS Server is installed. ">
      <value-context>pluginConfiguration</value-context>
      <value-name>homeDir</value-name>
   </destination-base-dir>
   <destination-base-dir name="Profile Directory" description="The profile
configuration directory.">
      <value-context>pluginConfiguration</value-context>
      <value-name>serverHomeDir</value-name>
   </destination-base-dir>
</bundle-target>
```

Note

There can be only one bundle definition for a resource type, but it can define multiple locations where content is allowed to be deployed.

5. Writing Agent Plug-ins: Procedures

5.1. Writing Agent Plug-ins Using a Template

The RHQ source files includes a plug-in generator. The generator goes through a series of questions that

cover most of the common configuration options for agent plug-ins, such as whether monitoring is enabled and whether there is a required JMX dependency. The generator creates the basic files including the **`pom.xml`** file, **`rhq-plugin.xml`** descriptor, and all of the associated Java files for the project. The agent plug-in can then be written and built on that template framework.

1. It is possible to build the plug-in generator from source; the files are in the *rhqInstallDir*`/modules/helpers/pluginGen/` directory. Alternatively, the latest version can be downloaded from http://rhq-project.org/display/RHQ/Plugins+-+Plugin+Generator.

2. Launch the generator.

```
java -jar rhq-pluginGen-version_#-jar-with-dependencies.jar
```

3. Answer the questions prompted in the generator. The first pass defines the root element for the plug-in descriptor; subsequent series configure children within the plug-in. The prompts are described in Table 11, “Plug-in Generator Questions”.

Table 11. Plug-in Generator Questions

Prompt Parameter	Description
The plugin root category PLATFORM(P), SERVER(S), SERVICE(I)	Sets the resource type of the root element in the plug-in. The value here corresponds to the **`<platform>`**, **`<server>`**, and **`<service>`**.
Name	Gives the name of the plug-in. This corresponds to the **`<plugin name=`***"name"* value.
PackagePrefix	Sets the name of the classes used by the plug-in. This corresponds to the **`package`** attribute in the **`<plugin`** element of the descriptor. This option is only given for the first plug-in element.
FileSystemRoot	Gives an absolute path to the location where the plug-in project directory should be created. This option is only given for the first plug-in element.
ComponentClass	Gives the name of the plug-in component Java file to create. This file is created with all of the required imported classes defined, based on the other functionality enabled in the generator.
DiscoveryClass	Gives the name of the discovery component Java file to create. This file is created with all of the resource discovery classes imported.
If it should support Events	Sets whether the plug-in will allow event (log) collection. If so, this options creates an **`<events>`** element in the descriptor and a corresponding **`*EventPoller.java`** file.
Description	

Prompt Parameter	Description
ParentType	Creates a parent plug-in type (**`<runs-inside>`**) element in the descriptor. This option is only given for the first plug-in element.
If it should support Monitoring	Sets whether the plug-in will allow monitoring. If so, this options creates a **`<metric>`** element in the descriptor.
If it should support Operations	Sets whether the plug-in will allow monitoring. If so, this options creates an **`<operation>`** element in the descriptor.
If it should support Singleton	Sets whether the plug-in is a singleton. If yes, it adds the **`singleton=true`** attribute to the resource element.
If it should support ResourceConfiguration	Sets whether this resource type has configuration properties that can be managed through the JBoss ON GUI. This corresponds to the **`<resource-configuration>`** properties in the descriptor.
If it should support SupportFacet	
If it should support CreateChildren	Sets whether children can be discovered and added to the resource inventory.
If it should support UsesExternalJarsInPlugin	Asks whether external (non-JBoss ON) JAR files are required by the plug-in. These files and classes must be imported by manually editing the plug-in files.
If it should support DeleteChildren	Sets whether children can be deleted from the resource inventory.
If it should support ManualAddOfResourceType	Sets whether children resources can be manually added to the resource's inventory.
If it should support UsePluginLifecycleListenerApi	Sets whether this plug-in will use the listener API. This option is only given for the first plug-in element.
If it should support DependsOnJmxPlugin	Sets whether this plug-in requires the JMX plug-in. This option is only given for the first plug-in element.
RhqVersion	Gives the version of the JBoss ON server and agent that this is used with. This option is only given for the first plug-in element.
Do you want to add a child to testServer?	Prompts whether to add a child element to the plug-in.

4. The second part of the generator prompts for any children elements to be added within the plug-in. These will commonly be services contained within the plug-in. These have the same series of questions as in step 3.

   The generator continues prompting for children until you enter N to complete the setup.

5. The generator creates the Maven product for the agent plug-in in the specified directory with the standard directory layout. It also creates templates for all of the required plug-in files:

- The project **`pom.xml`** file
- The **`rhq-plugin.xml`** descriptor file
- *name*Component.java files for the plug-in and any children
- *name*DiscoveryComponent.java files for the discovery component for the plug-in and any children
- *name*EventPoller.java files for any element which supports events

6. Edit the generated files to finish out the plug-in.
7. Build an agent plug-in.
    a. Go to the top directory for the agent plug-in project.
    b. Run the Maven build command:

```
mvn install
```

    This command builds the plug-in into a JAR file which can be uploaded to the JBoss ON server to deploy to the agents. The JAR file is named *pluginName-version*`.jar` and is located in a **`target/`** directory.

5.2. Validating Agent Plug-ins

If the agent plug-in was generated using the JBoss ON plug-in generator and then built with Maven, the plug-in itself can be validated using Maven. The JBoss ON/RHQ plug-in source files have a special validation class which can help ensure that the agent plug-in is valid before deploying it (and potentially harming an agent).

```
mvn org.rhq:rhq-plugin-validator:rhq-plugin-validate
```

If the agent plug-in was not created using the JBoss ON/RHQ plug-in generator, then add a **`<build>`** element to point to the validator and a pointer to the **`<pluginRepositories>`** element to point to the Maven repository.

```
<build>
   <plugins>
      <plugin>
         <groupId>org.rhq</groupId>
         <artifactId>rhq-core-plugin-validator</artifactId>
         <version>1.0.1-SNAPSHOT</version>
      </plugin>
   </plugins>
</build>
...

...
<pluginRepositories>
   <pluginRepository>
      <id>jboss</id>
      <name>JBoss Plugin Repository</name>
      <url>http://repository.jboss.org/maven2/</url>
      <snapshots>
         <enabled>false</enabled>
```

```
      </snapshots>
   </pluginRepository>
</pluginRepositories>
...
```

5.3. Notes on Editing Agent Plug-ins

The settings for a resource plug-in can be changed by editing the **`rhq-plugin.xml`** file and rebuilding the plug-in.

Important

Do not rename resource types when you edit the resource plug-in. This breaks backward compatibility with any resource that was inventoried using the older version of the plug-in.

5.4. Deploying Agent Plug-ins

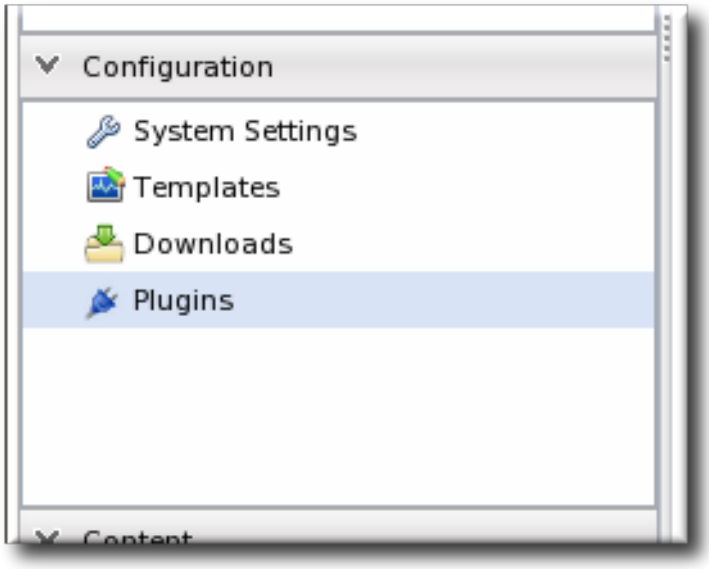
Agent plug-in files wind up in the *agentInstallDir*`/rhq-agent/plugins/` directory. Agent plug-ins are deployed, however, by uploading them to the JBoss ON *server*, and then the JBoss ON servers distribute them to the agents. As with server-side plug-ins, agent plug-ins can be deployed to a local JBoss ON server or through the JBoss ON UI.

Agent plug-ins are loaded when the agent starts. When a new agent plug-in is added, the agent needs to be restarted or a manual plug-in load operation has to be launched.

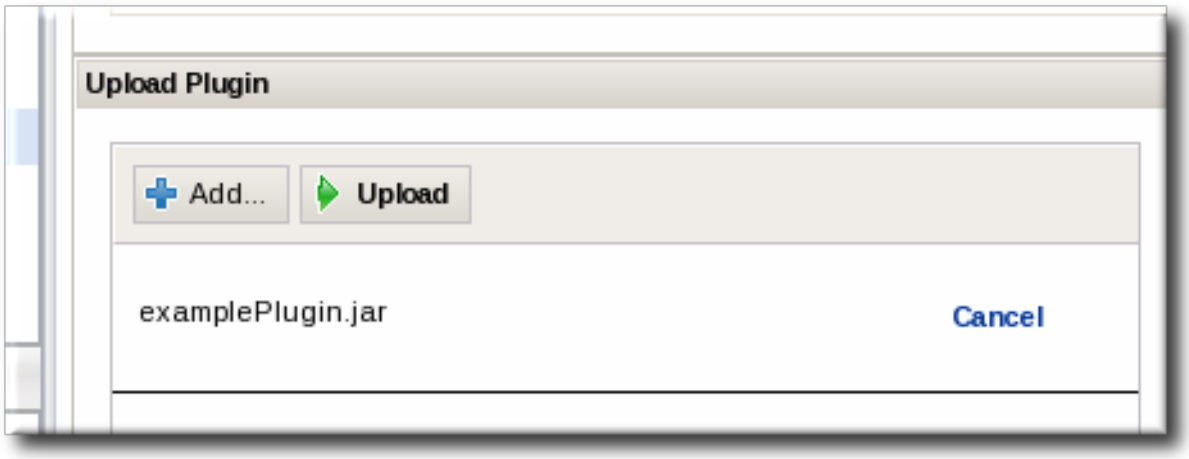
5.4.1. Remotely Deploying Agent Plug-ins

1. In the top menu, click the **`Administration`** tab.

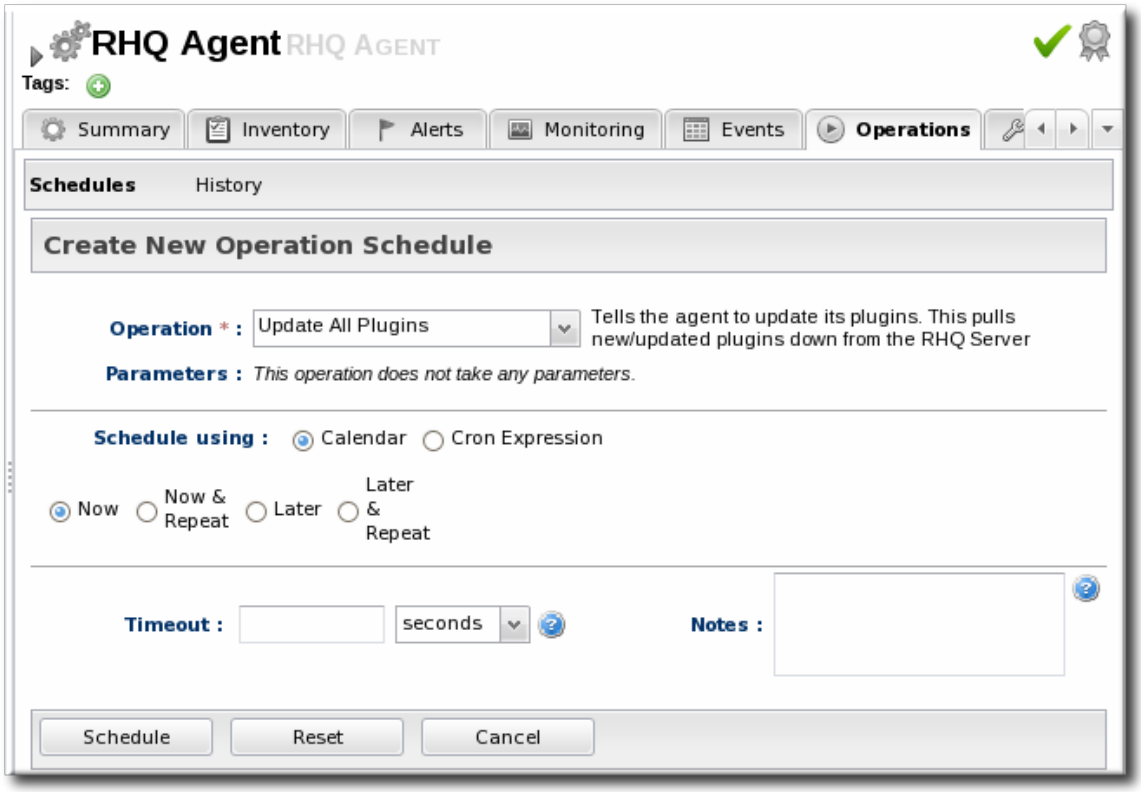
2. In the **`Configuration`** box on the left navigation bar, click the **`Plugins`** link.



3. In the **Agent Plugins** tab, scroll to the **Upload Plugin** section at the bottom of the page.
4. Click the **Add** button, and browse to the plug-in JAR file's location. If there are multiple plug-ins to deploy, just hit **Add** again and add in each one.
5. When all of the plug-ins to be deployed are listed in the box, click the **Upload** button.



6. Tell the agent to upload the new plug-in.
    a. Search for the agent, and then select it from the search results.
    b. Open the agent resource page.
    c. In the **Operations** tab, select the **Update Plugins** operation and schedule it to run immediately.



Agent plug-ins can also be updated by restarting the agent or by running the **plugins update** command in the agent command line.

5.4.2. Locally Deploying Agent Plug-ins

1. If the agent plug-in is built or accessible on the same machine as a JBoss ON server, then the agent plug-in can be dropped into a deployment folder:

   ```
   serverRoot/jon-server-
   3.0.0.GA1/jbossas/server/default/deploy/rhq.ear/rhq-downloads/rhq-
   plugins
   ```

   The JBoss ON server detects the new or updated plug-in and makes it available to the agents with other agent updates.

2. Have the agent load the new plug-ins. Either restart the agent or pass a command through the agent command line to upload the new plug-in.

   ```
   > plugins update
   ```

5.5. Removing and Re-deploying Agent Plug-ins

An agent plug-in, like a server-side plug-in, can be in one of three states:

- Deployed and active
- Deleted (disabled)
- Purged

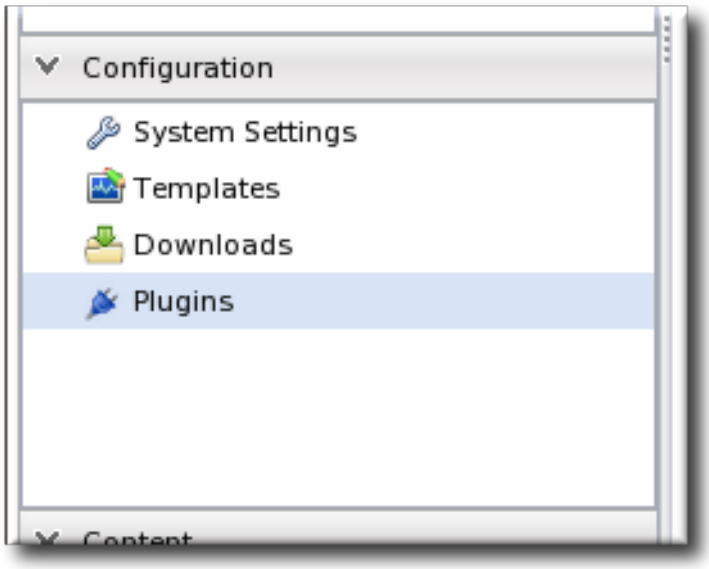
Both active and disabled plug-ins present, as JAR files, in the server and agent configuration. When a plug-in is present but disabled, it prevents any future updates of the plug-in from being deployed. Purging an agent plug-in permanently removes it from the configuration, which allows a new version of that agent plug-in to be deployed.

5.5.1. Deleting a Plug-in

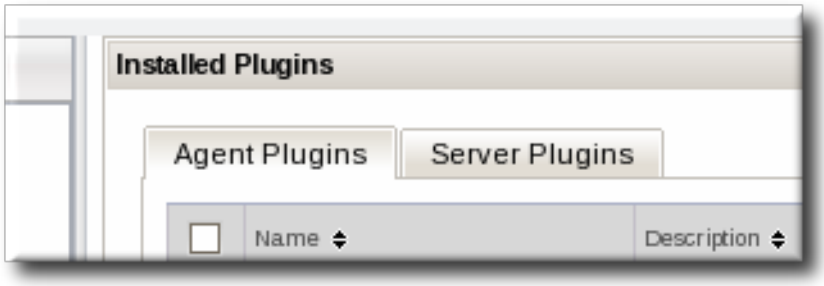
1. In the top menu, click the **`Administration`** tab.

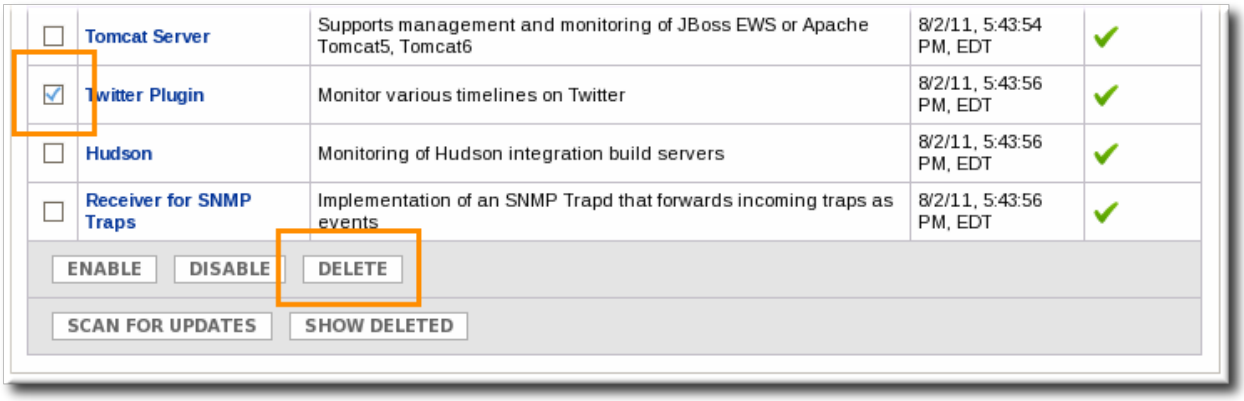
2. In the **`Configuration`** box on the left navigation bar, click the **`Plugins`** link.



3. Click the **`Agent Plugins`** tab.



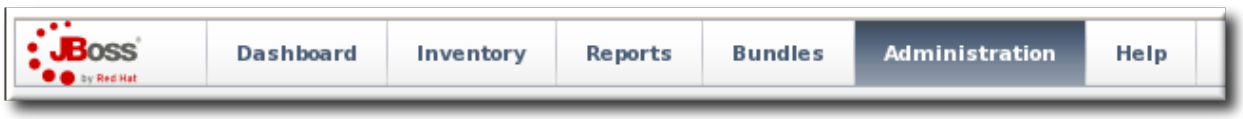
4. Select the checkbox by the plug-ins to delete.
5. Click the **`Delete`** button.



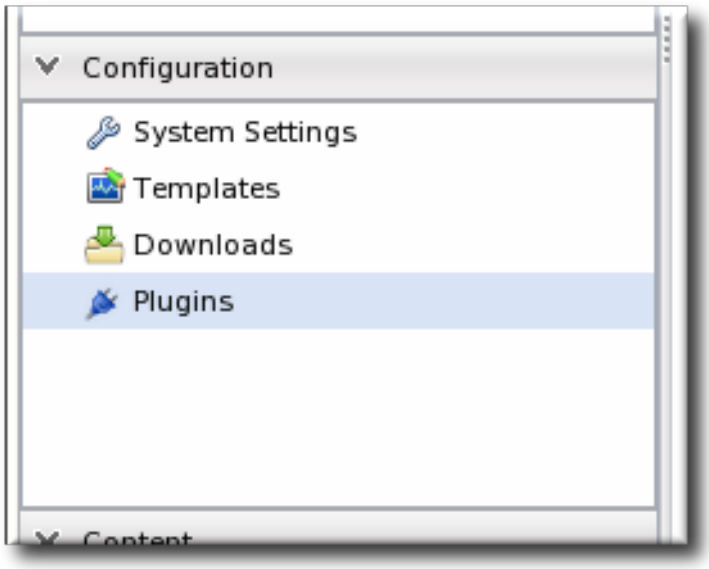
To view all of the undeployed plug-ins, click the **`SHOW DELETED`** button.

5.5.2. Purging and Re-deploying an Agent Plug-in

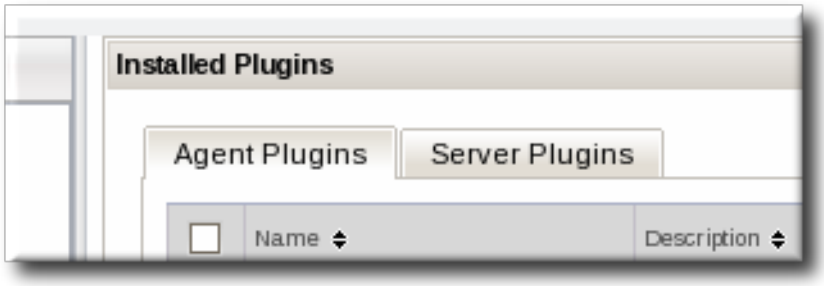
1. In the top menu, click the **`Administration`** tab.



2. In the **`Configuration`** box on the left navigation bar, click the **`Plugins`** link.

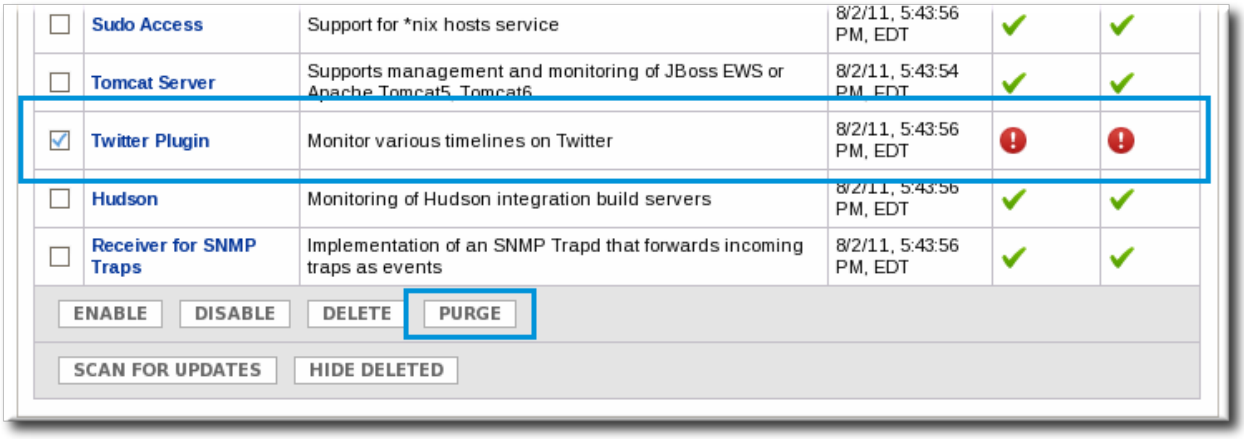


3. Click the **`Agent Plugins`** tab.



4. Click the **`SHOW DELETED`** button at the bottom of the plug-ins list.

5. Select the checkbox by the plug-in to re-deploy, and then click the **`PURGE`** button. This removes the entry in the JBoss ON database that tells the servers to ignore that plug-in and any updates to it.



6. Add and upload the plug-in like it is being deployed as new. This is described in Section 5.4, “Deploying Agent Plug-ins”.

6. Agent Advanced Management Plug-in System (AMPS) Reference

This is a reference of common components and elements used to write agent plug-ins.

6.1. Domain Objects

Domain objects include the basic pieces of the management inventory that define resources and types.

6.1.1. Resource and ResourceType

A *Resource* represents a single entity in inventory, be it a platform, server or service. The precise semantics of platforms, servers, adn services is vague, so a Resource object encapsulates any resource no matter what its category.

ResourceCategory is an enumeration that is associated with each Resource and indicates if the Resource is considered a platform, server, or service.

ResourceType represents types of resource instances that can be added to inventory. ResourceTypes are defined by plug-in descriptors. Another way of looking at it is that a ResourceType defines an application or service that can be managed by JBoss ON. ResourceTypes are added to JBoss ON as agent plug-ins are deployed, so a JBoss AS plug-in allows you to manage JBoss servers, a Tomcat plug-in manages Tomcat servers, and a custom plug-in can manage a custom application.

6.2. Plug-in Facets

A *facet* is simply an optional piece of functionality that a plug-in writer chooses to expose to the plug-in container and ultimately to the JBoss ON system as a whole. A plug-in writer is free to have his resource components implement some, all or none of these facets (obviously, the more facets that are implemented and exposed, the more powerful and useful the plug-in becomes).

6.2.1. AvailabilityFacet

This facet provides basic availability checking - is a managed resource up or down? When the plug-in container needs to know if a resource is running or not, it will ask the resource component's availability facet. Unlike the other facets, the AvailabilityFacet is required to be implemented by all resource components. You are forced to implement it because the ResourceComponent interface extends the AvailabilityFacet. You can optionally use the asynchronous availability collector to perform the avail checking.

6.2.2. ConfigurationFacet

This facet provides the ability for a resource component to modify the configuration of the actual managed resource. When a resource component implements this facet, it is saying it has the capability to get the current configuration of the managed resource as well as be able to change it. As an example, the JBoss AS Data Source Service resource component implements the ConfigurationFacet because it can report back to the user what the current settings are of that data source (e.g. its JDBC driver, its JNDI name, its connection pool size, etc) and it can allow the user to change those settings.

6.2.3. ContentFacet

Resources may have content associated with them including deployed software or software parts and other content. This system can be used to inventory these software parts and to install and remove them. Examples of deployed content are RPMs on Linux, EARs and WAR applications or libraries and deployment files on JBoss Application Server. Plug-ins can support arbitrary types of content with this system

Resources may have additional files (aka "content") associated with it - configuration files, deployment files, etc. Those resources that have associated content can implement the ContentFacet to help create, delete and manage that content.

6.2.4. ManualAddFacet

This facet should be implemented by the ResourceDiscoveryComponent class for types of resources that can be manually added to inventory via the JBoss ON GUI. In addition, the corresponding server or service elements in the plug-in descriptor must include the supportsManualAdd="true" attribute. Manual add can be a useful capability when a particular Resource cannot be auto-discovered for some reason.

6.2.5. MeasurementFacet

This facet exposes the capability for the component to collect measurement data from the managed resource and to report that data back to the server. For a measurement facet to work, the plug-in must define one or more metric definitions for the resource component's resource type in the plug-in descriptor. The resource component does not have to concern itself with how to schedule measurement collections and when it should collect the data. The only thing the MeasurementFacet requires the resource component to do is go out to the actual managed resource and collect the requested data. The plug-in container will manage all measurement collection schedules and will only call into the resource component's MeasurementFacet when the time is appropriate and it will only ask for the metrics that need to be collected at that time.

The measurement facet is what provides the graphs of measurement data that you see in the JBoss ON GUI Console.

6.2.6. OperationFacet

This facet allows the resource component to perform operations (aka control actions) on the managed resource itself. This allows, for example, the JBoss AS Server resource component to provide the capability to start and stop the JBoss AS server. Other examples of operations are being able to clear a connection pool for a data source, or ask a resource to empty a data cache. Whatever the managed resource can be told to do, a resource component can expose that as an operation to the JBoss ON user.

6.2.7. ResourceFactoryFacet

Some resource components can support the creation and deletion of child resources (for example, a resource component representing the JBoss AS server can create and delete JBoss AS data source services by creating and deleting *-ds.xml files). This facet exposes this functionality.

6.2.8. SupportFacet

To support managed resources, there is some data that support organizations might wish to know about regarding a managed resource, such as the contents of its log files, data files, and configuration files. The SupportFacet will provide a hook into this "support and maintenance" view of the managed resource.

6.3. Plug-in Components

6.3.1. ResourceDiscoveryComponent

The discovery component is an implementation written by the plug-in writer that performs the discovery of the actual managed resources. The discovery component's job is to scan the platform (that is, the machine the agent/plug-in container/plug-in is running on) and to report back what it finds. A discovery component is only responsible for finding resources that it directly is in charge of managing. That is to say, a JBoss AS plug-in discovery component is not responsible for discovering all Apache Web Servers - it only needs to find JBoss AS resources (leave the discovery of Apache Web Servers to the Apache plug-in).

A discovery component will be told to go hunt for resources at an appropriate time by the plug-in container. When the plug-in container asks the discovery component to go discover more resources, it will send in a **`ResourceDiscoveryContext`** object to the discovery component. This context contains all the information the component needs to perform its duties of finding and creating new resources. The discovery context is

also used to inject resources into the discovery component, in the case where the plug-in container was able to discover new resources on behalf of the discovery component. A plug-in container can only auto-discover resources if the appropriate metadata is supplied to it via the plug-in's descriptor.

6.3.2. ResourceComponent

A resource component is a plug-in abstraction that represents an actual managed resource. A resource component is stateful whose lifecycle is managed by the plug-in container.

The plug-in container will start and stop a resource component at the appropriate times. When a resource component is started, it typically connects to its underlying resource (the managed resource it represents) and maintains that connection until it is stopped by the plug-in container (this is an implementation detail that a plug-in writer is free to change). A resource component implementation provides a way for the plug-in container to ask for the availability of the managed resource ("is this resource up? or has it gone down?") and to access optional functionality facets that a plug-in writer chooses to expose. More on facets below.

6.4. Native System Information Access

All plug-ins have access to a set of native libraries that allow plug-in components to ask the underlying operating system for details about the machine on which the plug-in is running. Some of these features are not available on all hardware and OS platforms - only those platforms that have the native libraries available will be able to support all features described below. However, for those platforms that do not have the native libraries available, there will still be a limited feature set available.

6.4.1. SystemInfoFactory and SystemInfo

The plug-ins will have access to a SystemInfo object that is specific to the hardware/OS platform on which the plug-in is running. Once a plug-in obtains a SystemInfo object from its context (either **`ResourceDiscoveryContext`** or **`ResourceContext`**), it can make calls to that object which will call down into the native libraries to obtain the requested data from the operating system. If there are no native libraries available, the SystemInfo will be backed with a pure Java implementation of some, but not all, of the methods defined in the SystemInfo interface (see the JavaSystemInfo implementation of that interface). The methods that are not supported by the pure Java implementation will throw an **`UnsupportedOperationException`**.

6.4.2. ProcessInfoQuery

One interesting feature the SystemInfo interface provides is the ability to probe the operating system's process table. This is useful for your ResourceDiscoveryComponent implementations because they can scan the list of running processes and try to determine if it can auto-detect a managed resource that it is tasked to discover.

Through the use of the ProcessInfoQuery object, you can find processes that match a given set of criteria, defined by the Process Info Query Language (PIQL). You can even set pre-defined PIQL queries in your plug-in's descriptor via the process-scan tag to have the plug-in container scan the process table on behalf of your plug-in.

Note

Rather than repeat already documented information, refer to ProcessInfoQuery http://git.fedorahosted.org/git/rhq/rhq.git?p=rhq/rhq.git;a=blob;hb=master;f=modules/core/native-system/src/main/java/org/rhq/core/system/pquery/ProcessInfoQuery.java to learn more about the syntax and usage of PIQL.

7. Document Information

This guide is part of the overall set of guides for users and administrators of JBoss ON. Our goal is clarity, completeness, and ease of use.

7.1. Document History

Revision 3.0.1-1	**2013-10-31**	**Rüdiger Landmann**
Rebuild with publican 4.0.0		
Revision 3.0.1-0	**March 18, 2012**	**Ella Deon Lackey**
Updates for JBoss Operations Network 3.0.1.		
Revision 3.0-0	**December 5, 2011**	**Ella Deon Lackey**
Initial release of JBoss ON 3.0.		