Red Hat JBoss Operations Network 3.2
Development - Writing JON Command-Line Scripts

---

using the JBoss ON CLI and remote API
Edition 3.2

Ella Deon Ballard

Red Hat JBoss Operations Network 3.2 Development - Writing JON Command-Line Scripts

using the JBoss ON CLI and remote API
Edition 3.2

Ella Deon Ballard
dlackey@redhat.com

Legal Notice

Abstract

JBoss Operations Network provides its own command shell that can interact directly with the JBoss ON server. This CLI uses the JBoss ON remote API to perform most of the tasks available in the JBoss ON GUI, as well as additional operations like importing and exporting server configuration and exporting historic metric data. The CLI allows administrators to script and automate their JBoss ON deployment, which makes it easier to manage their infrastructure. This guide covers the basics of installing and using the JBoss ON CLI and provides examples of scripts for common tasks. It is intended primarily for administrators who will be using the default JBoss ON CLI to manage JBoss ON. This manual has a secondary audience for plug-in writers and developers who intend to write custom applications which leverage the remote API.

Table of Contents

About This Guide

The remote API for JBoss Operations Network, and affiliated APIs like the domain API for searches and the plug-in API for agent plug-ins, provides a framework for automating management tasks.

Automation can take different forms, from scheduling routine management tasks as cron jobs to initiating server scripts in response to fired alerts. The remote API lets administrators be more active and responsive in how they maintain resources.

Because of the different ways that scripts can be used with JBoss ON, there are different methods of both writing and calling scripts. Because of the variety of different implementing scripts, wriing scripts, and even tasks that scripts can perform, there is no way to give a comprehensive library of scripts so that "if you want to do A, run script B." The goal of this guide it to provide clarity on what the remote API provides to script writers and basic usage information and examples for the included interactive JBoss ON CLI utility.

With a good basis of understanding some of what JBoss ON server scripts can do, administrators can create whatever specific script they need for their environments.

1. Audience and Intent

JBoss ON itself is written in Java, and the installed CLI utility communicates to the server in Java. However, the CLI uses a JavaScript-style interpreter and can execute JavaScript files that are passed to it.

This guide is geared toward systems administrators and programmers with a basic understanding of JavaScript. It is beneficial (though not required) to have a basic grasp of Java, as well.

The JBoss ON remote API can be used to write services in other Java-compatible languages, such as Groovy and Scala. That is outside the scope of this guide.

2. Document History

Revision 3.2-10	**July 31, 2014**	**John Ha**
Formating and Updating content.		
Revision 3.2-6	**May 24, 2014**	**Ella Deon Ballard**
Fixing typos.		
Revision 3.2-5	**December 4, 2013**	**Ella Deon Ballard**
Bug fixes for JBoss Operations Network 3.2.		

Part I. Getting Started

JBoss ON can use scripts in different ways and through different methods. There are different APIs available depending on what, exactly, you want to accomplish. Understanding how the JBoss ON server interacts with clients through its different interfaces makes it easier to understand how to plan and write scripts for different management scenarios. Better scripting (and a better understanding of what JBoss ON is doing with scripts) improves your ability to automate how you manage resources in your IT environment.

Chapter 1. Understanding How Scripts Work with the JBoss ON Server and CLI

The JBoss Operations Network CLI and server-side scripts expose much of the core functionality of the JBoss ON server itself and give a lot of control to IT administrators on managing the resources in JBoss ON automatically and programmatically.

JBoss ON's remote API and the scripts which leverage it provide a way to access the JBoss ON server directly. Ultimately, this allows administrators to interact with JBoss ON differently than a strictly UI-based deployment.

Understanding the different JBoss ON APIs and how they interact with the JBoss ON server can help administrators plan what scripts they need for using the CLI programmatically, for planning alerts, and for designing clients.

1.1. A Summary of JBoss ON Public APIs

There are three relevant, public APIs for JBoss ON, which each interact differently with the JBoss ON server: the remote API, the domain API, and the plug-in API.

Table 1.1. JBoss ON APIs

API	Description
Remote	Resource management functions for tasks such as managing the inventory, changing configuration, uploading and managing content, initiating operations, viewing metrics and alerts, managing configuration drift, creating groups, and creating and managing users and roles. The remote API is accessible using standard Java enterprise client mechanisms or through the servlet-based JBoss remoting endpoint.
Domain	Functions which parallel the server's local manager beans, particularly in how a feature area is configured. Clients use the remote API to script actions. The remote API relies on the domain API to supply most of that functionality, such as the criteria to use for object searches or the different properties for viewing resource configuration.
Plug-in	Functions related to both agent (resource) plug-ins. This is not used at all for the JBoss ON CLI or server-side scripts. The APIs are used by the agent plug-ins to receive and convey information between the agent and the JBoss ON server.

1.2. The JBoss ON Server and Its Interfaces

The JBoss ON server is a Java application. All of the subsystems — like resources and groups, monitoring, alerting, drift, and provisioning — are contained in Enterprise JavaBeans in the server. The different JBoss ON APIs interact with those EJBs.

Specifically, most of the server EJBs are stateless session beans (SLSBs). These core server SLSBs follow a certain naming logic:

- The implementing class is **ManagerBean*. This implements both the local API and the remote API.
- SLSBs that are used internally by the server are **ManagerLocal*.
- SLSBs that define the remote API are **ManagerRemote*.

Each SLSB exposes its remote interface in the CLI with the naming convention **Manager*. For example, all of the methods associated with managing a resource belong to *ResourceManager*.

The server itself uses EJBs to create its data structure. The CLI interacts with the server EJBs through the remote API, the **`*Managers`**. The remote API is the compilation of the **`*Remote`** interfaces of the server's EJBs.

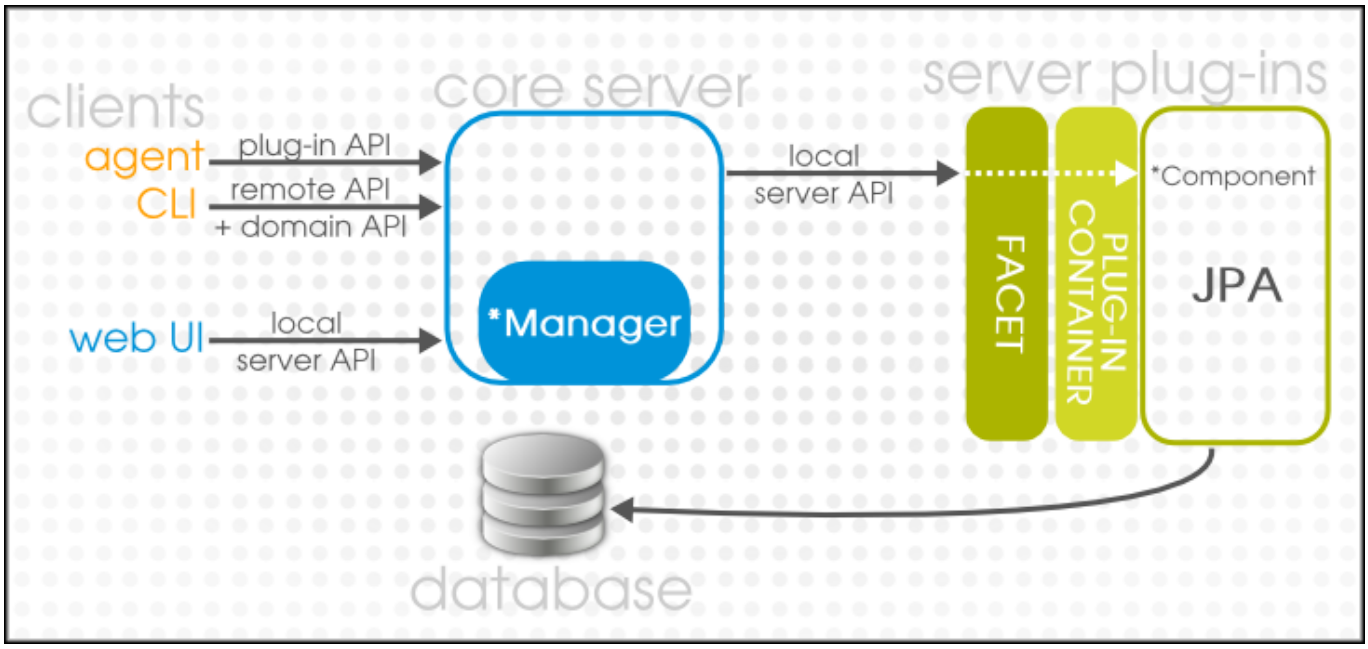


Figure 1.1. The Server Interfaces and Client Interactions

So, how does all of that fit in with the JBoss ON server and using server-side scripts?

All of the different APIs interact with clients and with the server in slightly different ways.

Agents, and their underlying resources and resource plug-ins, use the plug-in API to communicate to the core server (over the remoting framework). Agent and server plug-ins frequently work in pairs to extend JBoss ON functionality. For example, implementing an additional recipe style for bundles would require an agent plug-in to work with resources and a corresponding server plug-in to work with the server-side data structures.

The JBoss ON CLI uses the remote and domain APIs to communicate to the core server, also over the remoting framework. Unlike the agent, the CLI does not implement functionality (or at least not a lot); rather, it calls on the existing functionality to perform management tasks.

The GUI, which is hosted by the JBoss ON server, uses its local interface directly. Because the GUI uses a different API than the CLI, there are slight differences in what can be performed through the GUI as opposed to the CLI (Section 1.4, “Differences Between the JBoss ON CLI and JBoss ON GUI Operations”).

1.3. JBoss ON CLI Scripts and JBoss ON Server Scripts

There are two major ways to access the remote API: through the JBoss ON CLI and through a separate JBoss ON server script.

The scripts themselves are more or less identical, with the exception of some methods that are available when the JBoss ON CLI is run interactively. Both CLI scripts and server-side scripts are written in JavaScript and use the remote API primarily, with support from the domain API for tasks like searches.

The main difference is in how the scripts are invoked:

- The JBoss ON CLI can be run manually, execute a script file, and even be invoked automatically by a system tool like cron. The CLI can connect to to any JBoss ON server and can be run remotely.

  Essentially, the CLI is a script execution engine. It exposes the remote API in a scripting language, which makes it more convenient to interact dynamically the JBoss ON server.

- Server-side scripts are uploaded into a content repository, managed in the JBoss ON server database, and are then invoked in response to a fired alert. Even though the server-side scripts are located on the server, they still use the same APIs to communicate with the server as other CLI scripts.

1.3.1. Using CommonJS with JBoss ON CLI and Server Scripts

JBoss ON also supports use of CommonJS module loading mechanism for expressing dependencies among scripts. CommonJS is supported by both the CLI and for server scripts using the **`require`** function to locate and load scripts using variables.

Important

Use of the legacy **`exec`** function in the CLI interactive session, while still supported, is not the preferred method. It is recommended to port scripts to the loading mechanisms supported by CommonJS as they are more secure, portable and extensible.

For more information about using CommonJS, a list of script sources and their compatibility with the CLI and server scripts, as well as how to locate and load scripts, refer to Section 3, “Using Script Modules for Dependencies and Loading Custom Functions”.

1.4. Differences Between the JBoss ON CLI and JBoss ON GUI Operations

While there is a lot of overlap between what can be done in the JBoss ON UI and the CLI scripts, there is not complete parity. There are a couple of critical differences:

- Alert definition configuration. This is probably the most critical difference. While fired alerts can be retrieved through the CLI and scripts, there is no way to create or edit an alert definition, to set alert notifications, or to set alert conditions.
- Manually creating child resources
- Defining failover lists for servers
- Defining affinity groups for agents

Some other local managers are used internally by the server, so there is no reason for them to be exposed or accessed through the remote API.:

- Raw measurement data compression
- Processing, caching, and logging alert conditions
- Processing partition events

- Determining authorization to resources based on role membership
- Schedule loaders for metric collection, drift detection, and other scheduled events

1.5. Using Other Clients

Because JBoss ON exposes a large amount of functionality through its APIs, a number of different clients can be written to perform management tasks or to offer custom displays or views for resource data.

There are a number of different types of clients that can be written for JBoss ON:

- Java clients, such as a desktop application to view alerts or monitoring charts. The JBoss ON CLI itself is a Java shell that works as a script execution engine.
- Clients in JVM-compatible languages, such as Scala or Groovy
- REST clients

Writing and using custom clients is outside the scope of this guide. Still, the breadth and flexibility of JBoss ON's APIs open up opportunities for organizations to integrate JBoss ON with other applications or to create their own, environment-specific clients to make infrastructure management easier.

1.6. Additional Resources

Writing JBoss ON server scripts and using theJBoss ON CLI assumes a certain amount of familiarity with Java principles and JavaScript writing. These are some good tutorials and references to get a better understanding of Java and JavaScript.

- Java Scripting Programmer's Guide
- CommonJS resources
- Rhino: JavaScript for Java
- The Java Persistence Query Language (JEE 5 Tutorial)
- JBoss Operations Network Development: Writing Custom Plug-ins
- The JBoss ON API

Chapter 2. Installing the JBoss ON CLI

There are two default available ways to run a JBoss ON server script:

- In response to an alert
- Through the CLI utility

Running a script as part of an alert is configured with the alert definition and does not require any other user intervention.

Running a script manually requires that the CLI utility be installed.

2.1. Installing the CLI

The CLI utility can be installed on any system, not just a system with a JBoss ON server or agent installed. That CLI utility package is simply unzipped in the desired location.

Note

The Java 6 JDK must be installed on the system.

Important

The remote API **cannot** be run from a client inside an application server. For example, the remote API cannot be run from a client inside an EAP instance; it fails with errors like the following:

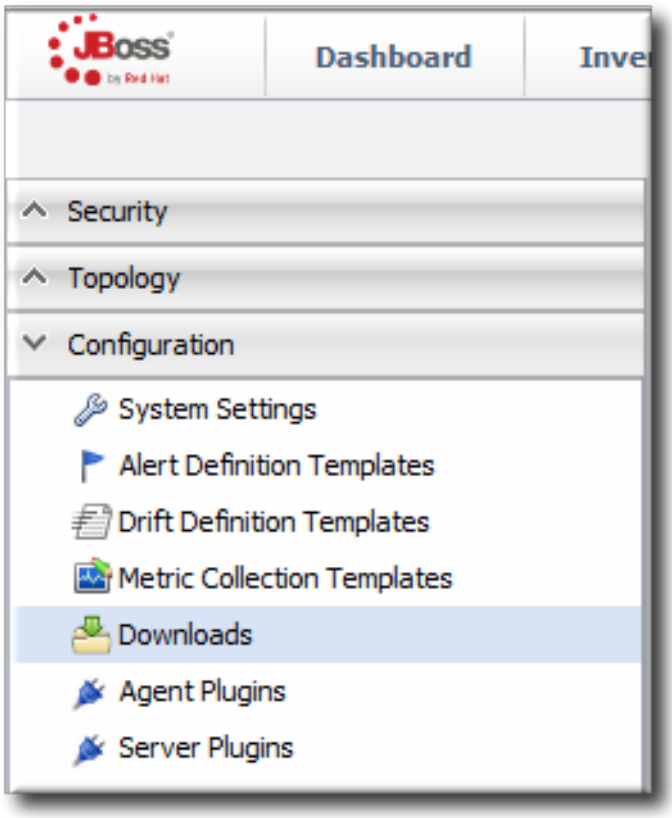
```
Caused by: java.lang.IllegalArgumentException: interface
org.rhq.enterprise.server.auth.SubjectManagerRemote is not visible from
class
loader
at java.lang.reflect.Proxy.getProxyClass(Proxy.java:353)
at java.lang.reflect.Proxy.newProxyInstance(Proxy.java:581)
at
org.rhq.enterprise.client.RemoteClientProxy.getProcessor(RemoteClientPr
oxy.java:69)
```

1. Open the JBoss ON GUI.

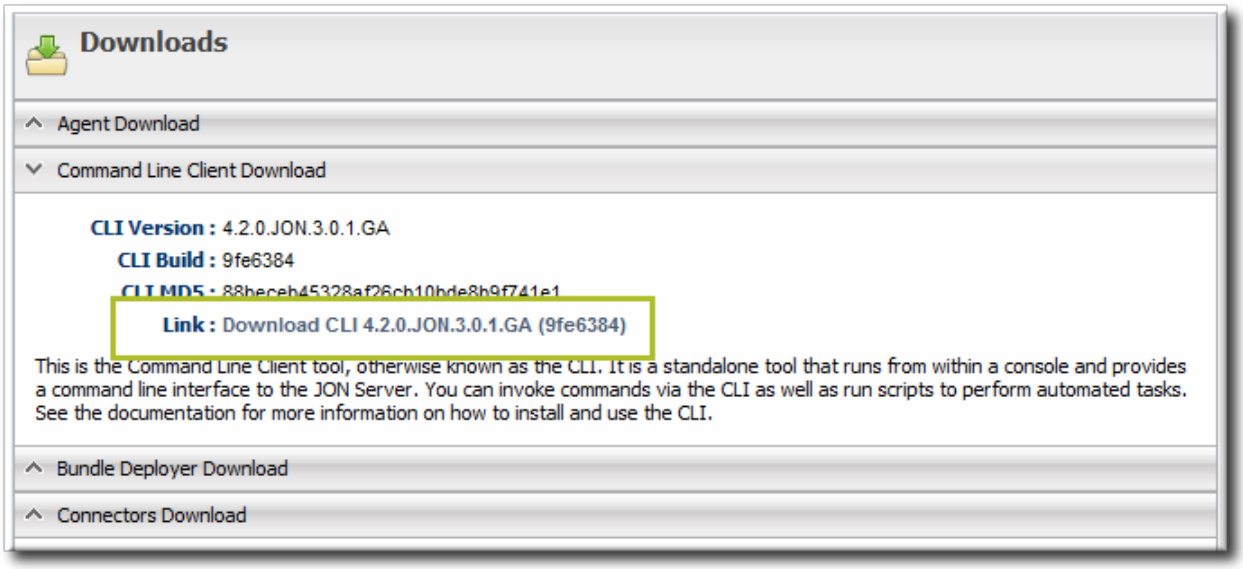
   ```
   http://server.example.com:7080
   ```

2. Click the **`Administration`** tab in the top menu.

3. Select the **`Downloads`** menu item.



4. Scroll to the **`Command Line Client Download`** section, and click **`Download CLI 4.9.JON.3.2`**.



5. Save the **`.zip`** file into the directory where the CLI should be installed.

6. Unzip the packages. For example:

```
[jsmith@server opt]$ unzip rhq-remoting-cli-4.9.JON.3.2.zip
```

2.2. Setting CLI Environment Variables

The JBoss ON CLI utility is a Java application, so it supports a large variety of environment variables, JVM settings, and other Java options.

The Java settings and environment variables are defined in the **rhq-cli-env.sh|bat** file.

Note

The **rhq-cli-env.sh|bat** file is fully annotated, so all available parameters are listed with full descriptions. If you want to change some of the JVM settings or parameters, read through the **rhq-cli-env.sh|bat** file to get an idea of what parameters are available.

Default values are supplied for most of these arguments, and those defaults are sufficient for most operating environments.

There is one environment variable which must be set for every installation: the Java home directory. This can be set by editing either the **RHQ_CLI_JAVA_HOME** or the **RHQ_CLI_JAVA_EXE_FILE_PATH** variable in the **rhq-cli-env.sh|bat** file. For example:

```
RHQ_CLI_JAVA_HOME="/usr/lib/jvm/java-1.6.0-openjdk-1.6.0.0.x86_64/jdk"
```

Important

Do not edit the **rhq-cli.sh|bat** file or attempt to set environment variables in that file. Environment variables and Java options should only be set in the **rhq-cli-env.sh|bat** file.

Editing the **rhq-cli.sh|bat** file may cause unexpected behavior or prevent the utility from running.

2.3. CLI Files and Directories

These are the directories and relevant files for the CLI utility.

Table 2.1. Important CLI Utility Directories and Files

File or Directory	Description
***cliRoot*/rhq-remoting-cli-4.9.0.JON320GA**	The installation directory. The CLI utility is simply unzipped, so the installation directory can be anywhere on a system.
CLI Scripts	
***cliRoot*/rhq-remoting-cli-4.9.0.JON320GA/bin**	Contains both Linux (shell) and Windows (batch) scripts for the CLI utility.
rhq-cli.sh | rhq-cli.bat	The CLI utility.
rhq-cli-env.sh | rhq-cli-env.bat	Sets environment variables for the CLI utility, such as the Java home directory and Java options.
***cliRoot*/rhq-remoting-cli-4.9.0.JON320GA/samples/modules/**	Contains sample CommonJS modules. This is also the default directory for loading CommonJS modules using the **modules:/** URI.

File or Directory	Description
util.js	Defines search functions for iterating through an array of object (**`foreach`**), to return the first matching object (**`find`**) or to return all matching objects (**`findAll`**). It also has functions for converting information from JavaScript hashes to JBoss ON configuration objects and back. The same as the **`samples/util.js`** script, but written as a CommonJS module.
bundles.js	Defines functions to create and deploy a bundle, create a bundle destination, or get information on supported base directories for a resource. The same as the **`samples/bundles.js`** script, but written as a CommonJS module.
drift.js	Defines functions to create and diff snapshots, get a definition, and show the history for a resource or specific file. The same as the **`samples/drift.js`** script, but written as a CommonJS module.
jbossas.js	Defines several functions to manage EAP 6 / AS 7 resources, including deploying web applications and managing EAP clusters. The same as the **`samples/deploy-to-and-restart-JBAS.js`** and **`samples/add-as7-standalone-server-to-cluster.js`** scripts, but combined and written as a CommonJS module.
Samples	
***`cliRoot`*`/rhq-remoting-cli-4.9.0.JON320GA/samples`**	Contains both sample JavaScript files and supplemental JavaScript files which can be adapted to work with other scripts.
`util.js`	A utility script which provides additional functions for search, mapping, and listing objects.
`measurement_utils.js`	A utility script which simplifies updating metrics schedules. Instead of having to understand the underlying measurement APIs, it provides a way to define the metrics to change based on their friendly (UI) name and then both enables/disables and sets collection intervals in simpler terms.
`drift.js`	A sample script which contains functions to create a snapshot, search for a specific drift definition, compare snapshots, and view drift histories.
`bundles.js`	A sample script which sets up a bundles definition. This includes generating a bundle archive, creating a destination group, and then deploying the bundle.
`deploy-to-and-restart-JBAS.js`	A sample script which deploys a new bundle or updated bundle to a group of JBoss servers, and then restarts the servers in the destination group.
Module Sample Scripts	
Logging	
***`cliRoot`*`/rhq-remoting-cli-4.9.0.JON320GA/conf`**	Contains the XML files to configure the log4j logger for the CLI utility.

File or Directory	Description
log4j.xml **log4j-debug.xml**	Sets the configuration for regular and debug logging for the CLI utility.
***cliRoot*/rhq-remoting-cli-4.9.0.JON320GA/logs**	Contains the error logs for the CLI utility. This directory is created when the CLI is first run.
rhq-server-cli.log	The error log for the CLI utility.
Libraries	
***cliRoot*/rhq-remoting-cli-4.9.0.JON320GA/lib**	Contains all of the libraries used specifically by the CLI utility, including some libraries for proxy resources and CLI-specific commands.

Part II. Basic Examples for Running JBoss ON Scripts Through the CLI

One of the ways to run scripts on the JBoss ON server is through the JBoss ON CLI. [1] The JBoss ON CLI is essentially a script interpreter which processes JavaScript — either interactively or from a file — and executes it inside the Java virtual machine. The CLI uses the server's remote API, an underlying domain API, and some of its own CLI-specific methods to make common tasks easier to execute.

The remote API, domain API, and CLI-specific methods are exposed to CLI and to scripts so that they can be easily used.

---

[1] Other methods include alert scripts and custom clients.

The JBoss Operations Network CLI is only one way to run scripts on the server, but it is a flexible and powerful way to create and execute scripts.

The JBoss ON CLI is essentially a script interpreter which processes JavaScript — either interactively or from a file — and executes it within the Java virtual machine. (The background concepts for the CLI are covered in more detail in Chapter 1, *Understanding How Scripts Work with the JBoss ON Server and CLI*.)

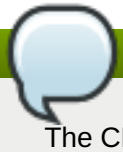
The JBoss ON CLI builds on the defined remote and domain API with its own internal commands, options, and methods which make writing scripts and managing resources easier. One of the biggest assets is the ability to create *resource proxies*, which are simplified (and therefore easier to use) resource objects.

This section focuses on the CLI options and basic usage tips. For scripting basics, see Short Examples.

1. Checking Compatible CLI and Server Versions

Like the agent and other JBoss ON components, the CLI utility has a version. The CLI utility must be the same version as the JBoss ON server to which it is trying to connect.

If the local JBoss ON CLI is not the same version as the JBoss ON server to which it is trying to connect, then the connection is refused.



Note

The CLI utility is not compatible with other versions of the JBoss ON server, neither newer nor older versions.

This behavior is configurable. There may be some instances where there is an external client or wrapper used with the JBoss ON CLI which does not (appear to) match the server version. To prevent the client connection from being rejected, add a Java option to disable the version checking property in the CLI configuration file.

```
[root@server ~]# vim cliRoot/rhq-remoting-cli-4.9.0.JON320GA/bin/rhq-cli-
env.sh

RHQ_CLI_ADDITIONAL_JAVA_OPTS=-Drhq.client.version-check=false
```

2. The JBoss ON CLI Command Syntax

The JBoss ON CLI is a shell and interpreter so that commands and statements can be executed interactively against the JBoss ON server. Scripts stored in files can also be executed, so it is possible to automate operations for the JBoss ON server.

2.1. The CLI Script

The CLI script is run directly from its *cli-install-dir*`/bin` directory. There are two files associated with launching the JBoss ON CLI:

- A script (**`rhq-cli.sh|bat`**)
- A file of environment variables (**`rhq-cli-env.sh|bat`**)

  The environment variables in the **`rhq-cli-env.sh|bat`** file use defaults that are reasonable for most deployments, so this file usually does not need to be edited. It is possible to reset variables to point a server that doesn't follow the default installation, such as a virtual machine or a non-default JVM. The comments at the top of the **`rhq-cli-env.|bat`** file contain a detailed list of available environment variables.

Important

Do not edit the **`rhq-cli.sh|bat`** file. Only set environment variables through the terminal or in the **`rhq-cli-env.sh|bat`** file, not the script itself.

Note

Be sure to set the correct path to the Java 6 installation in the **`RHQ_CLI_JAVA_HOME`** or the **`RHQ_CLI_JAVA_EXE_FILE_PATH`** variable.

The **`rhq-cli.sh|bat`** script has the following general syntax:

```
rhq-cli.sh|bat options
```

It is possible to launch the CLI script without any arguments, including specifying a username. This opens the CLI client *without* connecting to the server.

```
[jsmith@server bin]$ cliRoot/rhq-remoting-cli-4.9.0.JON320GA/bin/rhq-cli.sh
RHQ - RHQ Enterprise Remote CLI
unconnected$
```

While scripts can be executed without logging in, most of the functionality of the CLI is unavailable. To truly use the JBoss ON CLI, log into the server as a JBoss ON user, either by passing a username and password or by using the **login** command after starting the CLI.

```
[jsmith@server bin]$ rhq-cli -u rhqadmin -p rhqadmin
```

The CLI provides two modes of operation: interactive and non-interactive.

Interactive mode executes an individual statement. Interactive mode provides a simple environment for prototyping, testing, learning, and discovering features of the CLI.

Non-interactive mode loads a specified script file and executes multiple commands in sequence. Non-interactive mode provides the capability to automate tasks such as collecting metrics on managed resources or executing a scheduled operation.

Important

These native commands, like **quit**, are available only in interactive mode. They **cannot** be used in a script when the CLI is used in non-interactive mode, such as when running a script from file. In these instances, use the Java method.

After logging in, any commands (covered in Section 2.3, “Interactive CLI Commands”) can be passed to the server.

Important

Whatever user you run the CLI as — meaning, whatever system user runs the **rhq-cli.sh** script — must have write access to the **logs/** directory for the CLI.

If the CLI is installed as root, for example, then a regular user cannot run the CLI; it fails with write errors.

2.2. CLI Script Options

Both **rhq-cli.bat** and **rhq-cli.sh** scripts accept the options listed in Table 1, “Command-Line Options”.

Table 1. Command-Line Options

Short Option	Long Option	Description
-h	--help	Displays the help text of the command line options of the CLI.
-u	--user	The username used to log into the JBoss ON server.
-p	--password	The password used to log into the JBoss ON server.
-P		Displays a password prompt where input is not echoed backed to the screen.

Short Option	Long Option	Description
-s	--host	The JBoss ON server against which the CLI executes commands. Defaults to localhost.
-t	--port	The port on which the JBoss ON server is accepting HTTP requests. The default is 7080.
-c	--command	A command to be executed. The command must be encased in double quotes. The CLI will exit after the command has finished executing.
-f	--file	The full path and filename of a script to execute.
	--args-style	Indicates the style or format of arguments passed to the script.
-v	--version	Displays CLI and JBoss ON server version information once connected to the CLI.
	--transport	Determines whether or not SSL will be used for the communication layer protocol between the CLI and the JBoss ON server. If not specified the value is determined from the {port} option. If you use a port that ends in 443, SSL will be used. You only need to explicitly specify the transport when your JBoss ON server is listening over SSL on a port that does not end with 443.

2.3. Interactive CLI Commands

Some native commands are included in the *org.rhq.enterprise.client.commands* inside the CLI JAR itself. These commands are part of the CLI itself. Other input in the JBoss ON CLI is passed through the JavaScript interpreter to the server; these commands are passed to the CLI module.

- Section 2.3.1, “login”
- Section 2.3.2, “logout”
- Section 2.3.3, “quit”
- Section 2.3.4, “record”
- Section 2.3.5, “exec (deprecated)”

Important

These native commands are available only in interactive mode. They **cannot** be used in a script when the CLI is used in non-interactive mode, such as when running a script from file. In these instances, you must use the Java method.

2.3.1. login

Logs into a JBoss ON server with the specified username and password.

Optionally, the hostname (or IP address) and port can be specified. The hostname defaults to localhost, and the port defaults to 7080.

It is also possible to specify a transport protocol, which sets whether to use SSL to communicate with the server. If the transport is not given, then the CLI evaluates the transport based on the port. A port ending in 443 automatically uses SSL, while all other ports use standard connections. The only reason to explicitly set a transport method is if the server is listening over SSL over a port which does not end in 443.

```
login username password [host] [port]
```

The **login** command can be used in a module script or with the **rhq-cli.sh --f** option.

2.3.2. logout

Logs off of the JBoss ON server without existing from the CLI.

```
logout
```

The **logout** command can be used in a module script or with the **rhq-cli.sh --f** option.

2.3.3. quit

Exits the CLI.

```
quit
```

This only works when the CLI is running interactively. In a script, use **java.lang.System.exit**.

2.3.4. record

Records user input commands to a file. This is very useful if you are running the CLI interactively to test a new script that will later be run non-interactively in the CLI or as an alert server-side script.

```
record [-b | -e] [-a] -f filename
```

Option	Description
-b, --start	Specify this option to start recording.
-e, --end	Specify this option to stop recording.
-a, --append	Appends output to the end of a file. If not specified, output will be written starting at the beginning of the file.
-f, --file	The file where output will be written.

2.3.5. exec (deprecated)

Note

While an external script can be loaded into the CLI session using the **exec** command, this method of loading custom scripts and functions is deprecated. It is recommended that developers use a CommonJS module script. The default location for module scripts is *cliRoot*/rhq-remoting-cli-4.9.0.JON320GA/samples/modules.

Executes a statement or a script with the specified file name. A statement wraps onto multiple lines using backslashes.

Option	Description
-f, --file	The full path filename of the script to execute. The full path must be given, or the CLI cannot locate the script.
-s, --style=named|indexed	Indicates the style or format of arguments passed to the script. It must have a value of either **indexed** or **named**.

3. Using Script Modules for Dependencies and Loading Custom Functions

When automating management tasks or creating clients to interact withJBoss ON, it is frequently necessary to create custom classes and functions. The JBoss ON CLI implements CommonJS to support *script modules*.

Note

Script modules can be either JavaScript or Python files.

3.1. About Script Dependencies and Exporting Functions

When executing a script from a file by using the **rhq-cli.sh -f** option, there is no way to define an explicit dependency within an external file. Any functions required by that script must be contained in the script *or accessible to the script*.

The JBoss ON CLI uses CommonJS to support script *modules*. CommonJS is designed as a loading mechanism to express dependencies between scripts. If a script is in the default modules directory, then the script can be loaded into the CLI session or referenced in a script, and any exported function is available to the CLI. Modules can be used to load other modules into the CLI session, as well.

The only part of a script that is exposed is a function prepended with **exports** (following the CommonJS convention).

For example, the sample **util.js** file exports a **foreach** function which can search for any object in JBoss ON, as generic objects, arrays, criteria (search) objects, or maps.

```
exports.foreach = function (obj, fn) {
  var criteriaExecutors = {
  Alert: function(criteria) { return
AlertManager.findAlertsByCriteria(criteria); },
```

```
... 8< ...
}
```

All of the functions and objects that are *not* exported are private and cannot be used outside the module.

A module can be loaded into another mdule (creating a dependency) by listing the other module as a requirement.

```
var printResults = require("modules:/samples/modules/util.js");
```

Note

Only an exported function is available, and then it is only available through a variable defined in the script. For example, for the **`printResults`** variable, the **`foreach`** function is available, referenced as **`printResults.foreach`**.

This means that difference script modules can have functions defined in them with the same name, but because there are different variable namespaces, there is no conflict.

3.2. Default Locations for Script Module Sources

The **`require`** statement loads a modules from an identified location to be used in a script. The JBoss ON CLI, CLI script, or server-side script can load a module from any of several locations, for both the local and remote sources.

The **`require`** line has the format *sourceType:/path/moduleFile*.

The default module locations are defined in the following list.

Modules Directory

Description — Loads the specified module file from a configured location on the local system. The CLI has a system property (***`rhq.scripting.modules.root-dir`***) which defines a default root directory for modules. The modules location value then sets a subdirectory, within that root, where the module is located.

The location can be set or changed by updating the Java properties in the **`rhq-cli-env.sh`** file:

```
RHQ_CLI_ADDITIONAL_JAVA_OPTS='-Drhq.scripting.modules.root-
dir=/opt/rhq/modules' bin/rhq-cli.sh
```

Format — modules*:/relative-path*

Example:

```
var myModule =
require("modules:/myCompany/production/serverLocations"); //This will
load the module from a file called "serverLocations.js" from a file
that is located in the "myCompany/production" directory under a
configured "root-dir" location.
```

Available For — CLI

Local Filesystem

Loads the specified module file from an absolute path on the local system.

Format — file:*/path*

Example:

```
var myModule = require("file:/opt/jon/js-modules/myModule"); //This
will load the module from a file called "myModule.js" in the
/opt/jon/js-modules directory on the local filesystem.
```

Available For — CLI

JBoss ON Repository

Downloads the specified module file from the given JBoss ON repository. Because this connects to the server, a user must be logged into the JBoss ON server first.

Format — rhq://repositories/*repo_name/module_name*

Example:

```
var myModule = require("rhq://repositories/myRepo/myModule"); //This
will load the module stored in the RHQ server repository "myRepo"
called "myModule.js"
```

Available for — CLI

JBoss ON Downloads

Downloads the specified module file from the downloads location on the JBoss ON server. JBoss ON could make some scripts available that can be used for alert server-side scripts or for the CLI. Because this connects to the server, a user must be logged into the JBoss ON server first.

Format — rhq://downloads/*path/module_name*

Example:

```
var myModule = require("rhq://downloads/script-modules/util"); //This
will load the module stored in the RHQ server downloads section
called "js/util.js"
```

Available For — CLI, Server

3.3. Creating a New Module Source Type

JBoss ON can load a CommonJS script module from different locations, including both local directories and remote repositories.

It is possible to create support for loading modules from a different source, such as a URL. Creating a new module source location requires three things:

- Creating a new **`ScriptSourceProvider`** interface.
- Adding the Java service entry in the **`META-INF/services`** directory of a JAR for the class.
- Putting the new JAR into the **`lib/`** directory of the CLI installation directory to be available to clients and ir

***serverRoot*/jon-server-3.2/modules/org/rhq/server-startup/main/deployments/rhq.ear/lib** on the JBoss ON server so that it is available to server-side scripts (alert scripts).

Example 1. HTTP Module Source Location

This script source provider downloads a script module from a given URI as long as te Java version supports HTTP or HTTPS protocols.

```
  public class URLScriptSourceProvider implements ScriptSourceProvider {
  public Reader getScriptSource(URI scriptUri) {
  if (scriptUri == null) {
  return null;
  }
  try {
  return new InputStreamReader(scriptUri.toURL().openStream())
  }
  catch (MalformedURLException e) {
  return null;
  }
  catch (IOException e) {
  return null;
  }
  }
}
```

In the **META-INF** directory of the script provider JAR, create a **services** directory.

Then, within the **META-INF/services/** directory, create a file called **org.rhq.scripting.ScriptSourceProvider**. Put the full class names of any source providers implemented in the JAR, one per line. For example, if the **URLScriptSourceProvider** is the only source provider:

```
com.example.URLScriptSourceProvider
```

This is the layout of the JAR:

```
* com
* example
* URLScriptSourceProvider.class
* META-INF
* services
* org.rhq.scripting.ScriptSourceProvider
```

This JAR must then ba added to the **lib** of the CLI installation directory. After that, scripts run through the JBoss ON CLI on that system can use the HTTP provider to download the file from the network:

```
var myModule = require("http://server.example.com/rhq-scripts/stuff.js");
```

For the same source provider to be available to alert scripts, the JAR needs to be added to the **modules/org/rhq/server-startup/main/deployments/rhq.ear/lib/** directory of the JBoss ON server.

4. Available Implicit Variables in the JBoss ON API

In the Java programming language, classes in the *java.lang* package do not have to be imported; they are automatically made available. Classes in other packages, however, have to be explicitly imported.

In the JBoss ON CLI, there are a number of classes, particularly from the domain API, that are used routinely. To simplify using the JBoss ON CLI, everything under the **`org.rhq.core.domain`** class is automatically imported, which makes it easier to use the CLI for managing resources, alerts, and other configuration areas. For example, the class **`org.rhq.core.domain.criteria.ResourceCriteria`** is commonly used to query resources. The entire class path can be given when calling that class:

```
var criteria = new org.rhq.core.domain.criteria.ResourceCriteria();
var resource = new org.rhq.core.domain.resource.Resource();
```

Because the domain class is already imported, this can be more succinctly written as:

```
var criteria = new ResourceCriteria();
var resource = new Resource();
```

Common variables used with the CLI scripts are listed in Table 2, “Variables Available by Default to the JBoss ON CLI”. Methods and other information about these variables are in Section 5, “Methods Specific to the JBoss ON CLI”.

Table 2. Variables Available by Default to the JBoss ON CLI

Variable	Type	Description	Access Requires Login
rhq	org.rhq.enterprise.client.Controller	Provides built-in commands to the interactive CLI: login, logout, quit, exec, and version. Two of these methods, login and logout, can be called in server script files, such as **`rhq.login('rhqadmin', 'rhqadmin')`**.	YES
subject	org.rhq.core.domain.auth.Subject	Represents the current, logged in user. For security purposes, all remote service invocations require the subject to be passed; however, the CLI will implicitly pass the subject for you.	YES
Assert	org.rhq.bindings.util.ScriptAssert	Provides assertion utilities for CLI scripts.	NO

Variable	Type	Description	Access Requires Login
pretty	org.rhq.enterprise.client.TabularWriter	Provides for tabular-formatted printed and handles converting objects, particularly domain objects in the packages under *org.rhq.core.domain*, into a format suitable for display in the console.	NO
unlimitedPC	org.rhq.core.domain.util.PageControl		NO
pageControl	org.rhq.core.domain.util.PageControl	Used to specify paging and sorting on data retrieval operations	NO
exporter	org.rhq.enterprise.client.Exporter	Used to export output to a file. Supported formats are plain text in tabular format and CSV.	NO
ProxyFactory	org.rhq.bindings.client		NO
scriptUtil	org.rhq.enterprise.client.utility.ScriptUtil	Provides methods that can be useful when writing scripts.	NO
AlertManager	org.rhq.enterprise.server.alert.AlertManagerRemote	Provides an interface into the alerts subsystem.	YES
AlertDefinitionManager	org.rhq.enterprise.server.alert.AlertDefinitionManagerRemote	Provides an interface into the alerts definition subsystem.	YES
AvailabilityManager	org.rhq.enterprise.server.measurement.AvailabilityManagerRemote	Provides an interface into the measurement subsystem that can be used to determine resources' availability.	YES
CallTimeDataManager	org.rhq.enterprise.server.measurement.CallTimeDataManagerRemote	Provides an interface into the measurement subsystem for retrieving call-time metric data.	YES
RepoManager	org.rhq.enterprise.server.content.RepoManagerRemote	Provides an interface into the content subsystem for working with repositories.	YES
ConfigurationManager	org.rhq.enterprise.server.configuration.ConfigurationManagerRemote	Provides an interface into the configuration subsystem.	YES
DataAccessManager	org.rhq.enterprise.server.report.DataAccessRemote	Provides an interface for executing user-defined queries.	YES
EventManager	org.rhq.enterprise.server.event.EventManagerRemote	Provides an interface into the events subsystem.	YES

Variable	Type	Description	Access Requires Login
MeasurementBaselineManager	org.rhq.enterprise.server.measurement.MeasurementBaselineManagerRemote	Provides an interface into the measurement subsystem for working with measurement baselines.	YES
MeasurementDataManager	org.rhq.enterprise.server.measurement.MeasurementDataManagerRemote	Provides an interface into the measurement subsystem for working with measurement data.	YES
MeasurementDefinitionManager	org.rhq.enterprise.server.measurement.MeasurementDefinitionManagerRemote	Provides an interface into the measurement subsystem for working with measurement definitions.	YES
MeasurementScheduleManager	org.rhq.enterprise.server.measurement.MeasurementScheduleManagerRemote	Provides an interface into the measurement subsystem for working with measurement schedules.	YES
OperationManager	org.rhq.enterprise.server.operation.OperationManagerRemote	Provides an interface into the operation subsystem.	YES
ResourceManager	org.rhq.enterprise.server.resource.ResourceManagerRemote	Provides an interface into the resource subsystem.	YES
ResourceGroupManager	org.rhq.enterprise.server.resource.group.ResourceGroupManagerRemote	Provides an interface into the resource group subsystem.	YES
GroupDefinitionManager	org.rhq.enterprise.server.resource.group.definition.GroupDefinitionManagerRemote	Provides an interface for creating and managing dynagroups.	YES
ResourceTypeManager	org.rhq.enterprise.server.resource.ResourceTypeManagerRemote	Provides an interface into the resource subsystem for working with resource types.	YES
RoleManager	org.rhq.enterprise.server.authz.RoleManagerRemote	Provides an interface into the security subsystem for working with security rules and roles.	YES
SubjectManager	org.rhq.enterprise.server.auth.SubjectManagerRemote	Provides an interface into the security subsystem for working with users.	YES
SupportManager	org.rhq.enterprise.server.support.SupportManagerRemote	Provides an interface into the reporting subsystem for getting reports of managed resources.	YES

5. Methods Specific to the JBoss ON CLI

Some classes and methods are available to the JBoss ON CLI and JBoss ON server scripts which are not part of the regular API.

5.1. Methods Available to the CLI and Server Scripts

5.1.1. Assert

Provides assertion utilities for CLI scripts. More information about using Java assertions is in the Java language documentation.

Method	Signature

Assert.assertEquals	`assertEquals(float, float, float, String)`
	`assertEquals(short, short, String)`
	`assertEquals(double, double, double)`
	`assertEquals(long, long, String)`
	`assertEquals(byte, byte, String)`
	`assertEquals(Object, Object)`
	`assertEquals(char, char, String)`
	`assertEquals(Object, Object, String)`
	`assertEquals(double, double, double, String)`
	`assertEquals(byte[], byte[])`
	`assertEquals(boolean, boolean)`
	`assertEquals(Object[], Object[], String)`
	`assertEquals(Collection, Collection)`
	`assertEquals(Object[], Object[])`
	`assertEquals(byte, byte)`
	`assertEquals(float, float, float)`
	`assertEquals(char, char)`
	`assertEquals(int, int)`
	`assertEquals(long, long)`
	`assertEquals(Collection, Collection, String)`
	`assertEquals(short, short)`
	`assertEquals(String, String, String)`
	`assertEquals(byte[], byte[], String)`
	`assertEquals(boolean, boolean, String)`
	`assertEquals(String, String)`
	`assertEquals(int, int, String)`

Method	Signature
Assert.assertEqualsNoOrder	`assertEqualsNoOrder(Object[], Object[], String)` `assertEqualsNoOrder(Object[], Object[])`
Assert.assertExists	`assertExists(String)`
Assert.assertFalse	`assertFalse(boolean)` `assertFalse(boolean, String)`
Assert.assertNotNull	`assertNotNull(Object)` `assertNotNull(Object, String)`
Assert.assertNotSame	`assertNotSame(Object, Object, String)` `assertNotSame(Object, Object)`
Assert.assertNull	`assertNull(Object)` `assertNull(Object, String)`
Assert.assertNumberEqualsJS	`assertNumberEqualsJS(double, double, String)`
Assert.assertSame	`assertSame(Object, Object, String)` `assertSame(Object, Object)`
Assert.assertTrue	`assertTrue(boolean, String)` `assertTrue(boolean)`

Assert.fail	`fail()`
	`fail(String, Throwable)`
	`fail(String)`

5.1.2. Subject

Represents the current logged-in JBoss ON user.

Method	Signature
subject.addLdapRole	`addLdapRole(Role)`
subject.addRole	`addRole(Role)` `addRole(Role, boolean)`
subject.department	Prints the department value (if any) for the current user.
subject.emailAddress	Prints the email address for the current user.
subject.factive	Prints whether the user account is active.
subject.firstName	Prints whether the first name of the user.
subject.fsystem	
subject.id	Prints the ID number for the user account within JBoss ON.
subject.lastName	Prints the surname for the user.
subject.ldapRoles	Lists any roles associated with LDAP groups to which the current user is a member.
subject.name	Prints the JBoss ON user ID of the current user.
subject.ownedGroups	
subject.phoneNumber	Prints the phone number, if any exists, for the current user.
subject.removeLdapRole	`removeLdapRole(Role)`
subject.removeRole	`removeRole(Role)`
subject.roles	Prints the role name, permissions, associated LDAP users and groups, associated resource groups, and other information about every role to which the current user belongs.
subject.sessionId	Prints the current session ID number.
subject.smsAddress	Returns the pager number, if it exists, for the user.
subject.toString	`String toString()`
subject.userConfiguration	Returns all of the dashboard information, based on the configured portlets, dashboards, and settings that are specific to the logged-in user.

5.1.3. pretty

Converts CLI objects (particularly search results and other domain objects) into a pretty-print format in the output.

Method	Signature
pretty.exportMode	Prints the current export setting for the server.

Method	Signature

pretty.print	`print(String[][])`
	`print(PropertySimple, int)`
	`print(Configuration)`
	`print(PropertyMap, int)`
	`print(PropertyList, int)`
	`print(Collection)`
	`print(Map)`
	`print(Object[])`
	`print(Object)`

pretty.width	Prints the current width settings for the console display.

5.1.4. unlimitedPC and pageControl

Sets paging and sorting settings for returned data.

Method	Signature
unlimitedPC.addDefaultOrderingField	`addDefaultOrderingField(String, PageOrdering)` `addDefaultOrderingField(String)`
unlimitedPC.clone	`clone()`
unlimitedPC.firstRecord	Returns the first record in the results page.
unlimitedPC.getExplicitPageControl	`PageControl getExplicitPageControl(int, int)`
unlimitedPC.getSingleRowInstance	`PageControl getSingleRowInstance()`
unlimitedPC.getUnlimitedInstance	`PageControl getUnlimitedInstance()`
unlimitedPC.initDefaultOrderingField	`initDefaultOrderingField(String)` `initDefaultOrderingField(String, PageOrdering)`
unlimitedPC.orderingFields	
unlimitedPC.orderingFieldsAsArray	
unlimitedPC.pageNumber	Returns the current page number for paged results.
unlimitedPC.pageSize	Returns the current configured page size (number of returned entries per page).
unlimitedPC.primarySortColumn	

Method	Signature
unlimitedPC.primarySortOrder	
unlimitedPC.removeOrderingField	`removeOrderingField(String)`
unlimitedPC.reset	`reset()`
unlimitedPC.setPrimarySort	`setPrimarySort(String, PageOrdering)`
unlimitedPC.setPrimarySortOrder	`setPrimarySortOrder(PageOrdering)`
unlimitedPC.sortBy	`sortBy(String)`
unlimitedPC.startRow	Returns the current starting row number.
unlimitedPC.toString	`String toString()`
unlimitedPC.truncateOrderingFields	`truncateOrderingFields(int)`

5.1.5. exporter

Writes the CLI output to a specified file.

Method	Signature
exporter.close	`close()`
exporter.file	
exporter.format	Shows the current configured output format.
exporter.pageWidth	Shows the configured line length for content in the output file.
exporter.setFormat	`setFormat(String)`
exporter.setFile	`setFile(String)`
exporter.setPageWidth	`setPageWidth(int)`

Method	Signature
exporter.setTarget	`setTarget(String, String)`
exporter.write	`write(Object)`

5.1.6. ProxyFactory

Provides specialized methods to make it easier and simpler to manage resource objects.

Method	Signature
ProxyFactory.getResource	`ResourceClientProxy getResource(int)`
ProxyFactory.outputWriter	
ProxyFactory.remoteClient	Returns information about the managers and configuration used by the remote client. In the interactive CLI, this prints information about the manager beans used by the interactive CLI.
ProxyFactory.resource	

5.1.7. scriptUtil

Provides utilities to use for writing CLI scripts.

Method	Signature
scriptUtil.findResources	`PageList<Resource> findResources(String)`
scriptUtil.getFileBytes	`byte[] getFileBytes(String)`
scriptUtil.isDefined	`boolean isDefined(String)`
scriptUtil.saveBytesToFile	`saveBytesToFile(byte[], String)`
scriptUtil.sleep	`sleep(long)`

Method	Signature
scriptUtil.waitForScheduledOperationToComplete	`ResourceOperationHistory waitForScheduledOperationToComplete (ResourceOperationSchedule, long, int)` `ResourceOperationHistory waitForScheduledOperationToComplete (ResourceOperationSchedule)`

5.2. Methods Available to Proxy Resources

The ProxyFactory classes provide shortcuts for a lot of common resource management tasks, such as viewing monitoring data, running operations, or changing the resource or plug-in configuration. These methods are not in the regular API, but they can be used both by the JBoss ON CLI and by JBoss ON server-side scripts.

The shortcuts and methods available through ProxyFactory are different, depending on the resource type. Methods are only available if the resource type supports that functional area.

This section lists the three most common resource types:

- Table 3, “Proxy Methods for Platforms”
- Table 4, “Proxy Methods for JBoss AS/EAP Servers”
- Table 5, “Proxy Methods for Content Sources (EARs, WARs, JARs)”

Note

Use tab-complete in the interactive CLI to find the specific methods available for a resource type or to get the method signatures for individual methods.

Using proxy resources is covered in Section 6.2, “Using Resource Proxies”.

Table 3. Proxy Methods for Platforms

Information Methods			
measurements		Displays a pretty-print list of the available metrics, current values, and description of all measurements for the platform resource.	
operations		Lists the available operations for the resource type.	
Shortcut Metric Methods			
OSName	OSVersion	architecture	createdDate
description	distributionName	distributionVersion	freeMemory
freeSwapSpace	hostname	idle	totalMemory
systemLoad	totalSwapSpace	usedSwapSpace	usedMemory
userLoad	modifiedDate	waitLoad	version

Shortcut Resource Entry Methods		
id (inventory ID number)	resourceType	name (inventory name)
Shortcut Operation Methods		
manualAutodiscovery	cleanYumMetadataCache	viewProcessList
Shortcut Configuration Methods		
editPluginConfiguration()	pluginConfiguration	
pluginConfigurationDefinition		
Shortcut Content Methods		
contentTypes		
Shortcut Inventory Methods		
children		

Method	Signature
platform.getChild	`ResourceClientProxy getChild(String)`
platform.getMeasurement	`Measurement getMeasurement(String)`
platform.updatePluginConfiguration	`PluginConfigurationUpdate updatePluginConfiguration(Configuration)`
platform.toString	`String toString()`

Table 4. Proxy Methods for JBoss AS/EAP Servers

Information Methods			
measurements		Displays a pretty-print list of the available metrics, current values, and description of all measurements for the JBoss resource.	
operations		Lists the available operations for the resource type.	
Shortcut Metric Methods			
JVMFreeMemory	JVMMaxMemory	JVMTotalMemory	activeThreadCount
activeThreadGroupCount	buildDate	createdDate	description
modifiedDate	startDate	totalTransactions	totalTransactionsperMinute
transactionsCommitted	transactionsCommittedperMinute	transactionsRolledback	transactionsRolledbackperMinute
partitionName	versionName	version	
Shortcut Resource Entry Methods			
id (inventory ID number)	resourceType	name (inventory name)	

Shortcut Operation Methods		
restart	shutdown	start
Shortcut Configuration Methods		
editPluginConfiguration()	pluginConfiguration	
pluginConfigurationDefinition		
Shortcut Content Methods		
contentTypes		
Shortcut Inventory Methods		
children		

Method	Signature
jbossas.getChild	`ResourceClientProxy getChild(String)`
jbossas.getMeasurement	`Measurement getMeasurement(String)`
jbossas.updatePluginConfiguration	`PluginConfigurationUpdate updatePluginConfiguration(Configuration)`
jbossas.toString	`String toString()`

Table 5. Proxy Methods for Content Sources (EARs, WARs, JARs)

Information Methods		
measurements	Displays a pretty-print list of the available metrics, current values, and description of all measurements for the content resource.	
operations	Lists the available operations for the resource type.	
Shortcut Metric Methods		
createdDate	modifiedDate	description
path	version	exploded
Shortcut Resource Entry Methods		
id (inventory ID number)	resourceType	name (inventory name)
Shortcut Operation Methods		
revert		
Shortcut Configuration Methods		
editPluginConfiguration()	pluginConfiguration	
pluginConfigurationDefinition		
Shortcut Content Methods		
contentTypes	backingContent	

Shortcut Inventory Methods	
children	
Method	**Signature**
content.getChild	`ResourceClientProxy getChild(String)`
content.getMeasurement	`Measurement getMeasurement(String)`
content.updatePluginConfiguration	`PluginConfigurationUpdate updatePluginConfiguration(Configuration)`
content.toString	`String toString()`
content.retrieveBackingContent	`retrieveBackingContent(String fileName)`
content.updateBackingContent	`updateBackingContent(String filename, String displayVersion)`

6. Common Actions with JBoss ON CLI Scripts

Important

The remote API **cannot** be run from a client inside an application server. For example, the remote API cannot be run from a client inside an EAP instance; it fails with errors like the following:

```
Caused by: java.lang.IllegalArgumentException: interface
org.rhq.enterprise.server.auth.SubjectManagerRemote is not visible from
class
loader
at java.lang.reflect.Proxy.getProxyClass(Proxy.java:353)
at java.lang.reflect.Proxy.newProxyInstance(Proxy.java:581)
at
org.rhq.enterprise.client.RemoteClientProxy.getProcessor(RemoteClientPr
oxy.java:69)
```

6.1. Logging In

The CLI actually connects to the JBoss ON server, much like connecting through the GUI. This means that

you have to log into the server before you can perform most tasks.

There are two ways to log into the server through the CLI:

- By passing user credentials when the **`rhq-cli.sh`** script is run.
- By using the **`command`** in a script or after starting the CLI without connecting to the server.

When logging into the server, other connection information, such as a server name and port number, can be passed with the login command.

Example 2. Logging in to a Specified Server

This connects to the CLI and logs into the JBoss ON server on 192.168.1.100 over port 70443. Because the port number ends with 443, the CLI automatically attempts to connect over SSL.

```
rhq-cli.sh -u rhqadmin -p rhqadmin -s 192.168.1.100 -t 70443
```

Example 3. Prompting for a Password

Instead of sending the password in clear text, it is possible to have the server prompt for a password. This is probably unnecessary when connecting to a server on the local host, but it is useful if the target JBoss ON server is on a different system.

```
rhq-cli.sh -u rhqadmin -P
```

6.2. Using Resource Proxies

The JBoss ON CLI interacts directly with the JBoss ON server through remote APIs for handling resource objects and through the domain APIs for tasks like searches.

The JBoss ON CLI itself provides another API layer that can make it easier to perform common operations. The CLI can create a *resource proxy* object in the CLI, and then that object uses the classes available in the **`ProxyFactory`** to interact with the remote and domain API.

One thing to remember is that proxy resources still use the remote and domain API. The proxy API just provides a simpler and clearer API *on top of* the remote and domain APIs that can make it easier to script many operations.

Note

The **`ProxyFactory`** is available to the JBoss ON CLI in interactive mode or when using a script file. It is also available to server scripts, such as scripts used for alerting.

The **`ProxyFactory`** gets information about a resource, which is identified in the **`getResource()`** method with the resource's ID number.

At its simplest, **`ProxyFactory`** can return a complete summary of information about the specified resource, such as its current monitoring data and traits, resource name, available metrics, available operations, content information, and child inventory, all dependent on the resource type. For example:

```
rhqadmin@localhost:7080$ ProxyFactory.getResource(10001)
ResourceClientProxy_$$_javassist_0:
                              OSName: Linux
                           OSVersion: 2.6.32-220.4.1.el6.x86_64
                        architecture: x86_64
                            children:
                        contentTypes: {rpm=RPM File}
                         createdDate: Mon Feb 06 11:24:50 EST 2012
                         description: Linux Operating System
                    distributionName: Red Hat Enterprise Linux Server
                 distributionVersion: release 6.2 (Santiago)
                          freeMemory: 16.7GB
                       freeSwapSpace: 25.6GB
                             handler:
                            hostname: server.example.com
                                  id: 10001
                               idle: 70.8%
                        measurements: [Wait Load, Used Memory, System
Load, Distribution Version, Total Memory, OS Name, Free Memory, Hostname,
Architecture, Distribution Name, Idle, Total Swap Space, Used Swap Space,
User Load, OS Version, Free Swap Space]
                        modifiedDate: Mon Feb 06 11:24:50 EST 2012
                                name: server.example.com
                          operations: [viewProcessList,
cleanYumMetadataCache, manualAutodiscovery]
                 pluginConfiguration:
         pluginConfigurationDefinition: ConfigurationDefinition[id=10009,
name=Linux]
                        resourceType: Linux
                          systemLoad: 0.0%
                         totalMemory: 23.5GB
                      totalSwapSpace: 25.6GB
                          usedMemory: 6.8GB
                       usedSwapSpace: 0.0B
                            userLoad: 15.8%
                             version: Linux 2.6.32-220.4.1.el6.x86_64
                            waitLoad: 0.0%
```

To truly manage resources, the **`ProxyFactory`** creates a resource proxy object.

Example 4. Defining a Platform Proxy Resource

```
var rhelServerOne = ProxyFactory.getResource(10001)
```

The methods that are available to a resource proxy depend on the resource type and the resource's own configuration. There are five major types of operations that can be performed on resource proxies:

- Viewing basic information about the resource, such as its children
- Getting measurement information
- Running operations
- Changing resource and plug-in configuration

- Updating and retrieving content

For each resource type, methods are exposed which allow you to find and use specific information about the resource. Additionally, the proxy API includes "shortcuts" which provide one-word methods, without requiring any parameters, to perform common remote API tasks, like getting monitoring information.

The proxy API for common resource types is listed in Section 5.2, “Methods Available to Proxy Resources”.

Note

Use tab-complete in the interactive CLI to find the specific methods available for a resource type or to get the method signatures for individual methods.

Example 5. Viewing a Resource's Children

`ProxyFactory` has a method for all proxy objects, **`children`**, which lists all of the children for the proxy resource.

```
var rhelServerOne = ProxyFactory.getResource(10001)

rhqadmin@localhost:7080$ platform.children
Array of org.rhq.bindings.client.ResourceClientProxy
[10027] Bundle Handler - Ant (Ant Bundle Handler::AntBundlePlugin)
[10026] CPU 6 (CPU::Platforms)
[10025] CPU 0 (CPU::Platforms)
[10024] CPU 5 (CPU::Platforms)
[10023] CPU 1 (CPU::Platforms)
[10022] CPU 4 (CPU::Platforms)
[10021] CPU 2 (CPU::Platforms)
[10020] CPU 3 (CPU::Platforms)
[10019] CPU 7 (CPU::Platforms)
[10018] /boot (File System::Platforms)
[10017] / (File System::Platforms)
[10016] /dev/shm (File System::Platforms)
[10015] /home (File System::Platforms)
[10014] eth1 (Network Adapter::Platforms)
[10013] eth2 (Network Adapter::Platforms)
[10012] eth0 (Network Adapter::Platforms)
[10011] lo (Network Adapter::Platforms)
[10004] postgres (Postgres Server::Postgres)
[10003] AS server.example.com RHQ Server (JBossAS Server::JBossAS)
[10002] RHQ Agent (RHQ Agent::RHQAgent)
```

Example 6. Viewing Resource Metrics

`ProxyFactory` provides a set of shortcut metrics for each individual measurement for a resource type. This corresponds to the **`findLiveData()`** method in the remote API, but it is much easier to get monitoring information quickly and it is simpler to identify what metrics are available.

To get a single measurement value, use the method for that resource type. (Get a list of all methods for a proxy object using tab-complete.)

```
var jbossas = ProxyFactory.getResource(14832)

rhqadmin@localhost:7080$ jbossas.JVMTotalMemory
Measurement:
                name: JVM Total Memory
        displayValue: 995.3MB
 description: The total amount of memory currently available in the app
server JVM for current and fut...
```

Alternatively, simply get a list of metrics with their current values using the **`measurements`** method:

```
var rhelServerOne = ProxyFactory.getResource(10001)

rhqadmin@localhost:7080$ rhelServerOne.measurements
Array of org.rhq.bindings.client.ResourceClientProxy$Measurement
name                 displayValue                          description
-------------------------------------------------------------------------
-------------------------------------------------
Wait Load            0.0%                                  Percentage of
all CPUs waiting on I/O
Used Memory          6.3GB                                 The total used
system memory
System Load          0.0%                                  Percentage of
all CPUs running in system mode
Distribution Version release 6.2 (Santiago)                version of the
Linux distribution
Total Memory         31.4GB                                The total
system memory
OS Name              Linux                                 Name that the
operating system is known as
Free Memory          25.2GB                                The total free
system memory
Hostname             server.example.com                    Name that this
platform is known as
Architecture         x86_64                                Hardware
architecture of the platform
Distribution Name    Red Hat Enterprise Linux Server       name of the
Linux distribution
Idle                 92.6%                                 Idle
percentage of all CPUs
Total Swap Space     33.6GB                                The total
system swap
Used Swap Space      0.0B                                  The total used
system swap
User Load            16.7%                                 Percentage of
all CPUs running in user mode
OS Version           2.6.32-220.4.2.el6.x86_64             Version of the
operating system
Free Swap Space      33.6GB                                The total free
system swap
16 rows
```

Example 7. Running Operations on a Proxy

`ProxyFactory` has a shortcut method for every operation available for a resource.

First, get the list of operations available for the resource type using the **`operations`** method:

```
var rhelServerOne = ProxyFactory.getResource(10001)

rhqadmin@localhost:7080$ rhelServerOne.operations
Array of org.rhq.bindings.client.ResourceClientProxy$Operation
name                  description
-----------------------------------------------------------------------
-------------------------------------------------
viewProcessList       View running processes on this system
cleanYumMetadataCache Deletes all cached package metadata
manualAutodiscovery   Run an immediate discovery to search for resources
3 rows
```

Then, run the given operation method.

```
rhqadmin@localhost:7080$ rhelServerOne.viewProcessList();
Invoking operation viewProcessList
Configuration [11951] - null
  processList [305] {
pid   name                                          size       userTime
kernelTime
-----------------------------------------------------------------------
-----------------------------------------
1     init                                          19865600   150
10050
....
26285 httpd                                         214618112  90
80
26286 httpd                                         214618112  90
80
26288 httpd                                         214618112  110      70
26289 httpd                                         214618112  90
80
27357 java                                          4734758912 1289650
373890
30458 postgres                                      218861568  1820
27440
30460 postgres                                      180985856  1210
5330
30462 postgres                                      218984448  13080
42200
30463 postgres                                      218861568  3970
26940
30464 postgres                                      219328512  10600
15320
30465 postgres                                      181407744  18680
78760
30482 httpd                                         185905152  1660
7520
32410 bash                                          108699648  0
10
```

```
32420 java                                          6024855552 3890240
669810
305 rows
}
```

Example 8. Changing Configuration Properties

If the resource type supports resource configuration editing or if the resource type has plug-in connection properties, then the resource proxy has methods — **`editResourceConfiguration()`** and **`editPluginConfiguration()`**, respectively — to edit those properties.

The current configuration can be printed using the **`get*Configuration)`**. For example, for the plug-in configuration:

```
var rhelServerOne = ProxyFactory.getResource(10001)

rhqadmin@localhost:7080$ rhelServerOne.getPluginConfiguration()
Configuration [10793] - null
  metadataCacheTimeout = 1800
  enableContentDiscovery = false
  yumPort = 9080
  enableInternalYumServer = false
  logs [0] {
  }
```

The **`edit*Configuration()`** method brings up a configuration wizard that goes through all of the properties individually and prompts to keep or change each value. The properties are even grouped according to the same organization that the JBoss ON web UI uses. For example:

```
rhqadmin@localhost:7080$ rhelServerOne.editPluginConfiguration();
Non-Grouped Properties:
Group: Content
enableContentDiscovery[false]:
enableInternalYumServer[false]:
yumPort[9080]:
metadataCacheTimeout[1800]:
Group: Event Logs
[R]eview, [E]dit, Re[V]ert [S]ave or [C]ancel:
...
```

After each group, you have the option to revert or save the changes. Once the changes are saved, they are immediately updated on the JBoss ON server.

Keys	Action
return	Selects the default or existing value for a property.
ctrl-d	The same as selecting the unset checkbox in the configuration UI.
ctrl-k	Exits the configuration wizard.
ctrl-e	Displays the help description for the current property.

Example 9. Managing Content on Resources

Some types of resources have content associated with them. These are typically EAR or WAR resources within an application server. The content file actually associated with that EAR/WAR resource is called *backing content*. These are usually JARs.

This content can be updated or downloaded from the resource.

To retrieve backing content (meaning, to download the JAR/EAR/WAR file), specify the filename and file path *on the application server*. For example:

```
var contentResource = ProxyFactory.getResource(14932)
contentResource.retrieveBackingContent("/resources/backup/original.war")
```

To update the content for the resource, use the **`updateBackingContent`** method and specify the filename with the path on the application server to put the content and the version number of the content. For example:

```
contentResource.updateBackingContent("/resources/current/new.war", "2.0")
```

6.3. Passing Command and Script Arguments

When connecting to the CLI using **`rhq-cli.sh`**, commands or full scripts can be passed simultaneously. This is a non-interactive way to connect to the CLI, since the CLI runs the specified command or script and then exits, rather than staying connected in interactive mode.

Example 10. Passing Variables to the Server

A single command can be passed to the CLI by using the **`-c`**. In this example, the server searches for and prints all supported resource types for the server and prints the results to **`resource_types.txt`**

```
rhq-cli.sh -u rhqadmin -p rhqadmin -c
"pretty.print(ResourceTypeManager.findResourceTypesByCriteria(new
ResourceTypeCriteria()))" > resource_types.txt
```

The **`ResourceTypeManager.findResourceTypesByCriteria(new ResourceTypeCriteria())`** class invokes the **`findResourceTypesByCriteria`** operation on **`ResourceTypeManager`**. A new **`ResourceTypeCriteria`** object is passed as the argument.

Nothing has been specified on the criteria object so all resource types will be returned.

`pretty` is an implicit object made available to commands and scripts by the CLI. This is useful for outputting objects in a readable, tabular format which is designed for domain objects.

This single command provides a nicely formatted, text-based report of the resource types in the inventory.

Example 11. Running a Script

This executes the script file, **`my_script.js`**. The CLI terminates immediately after the script has finished executing.

```
cliRoot/rhq-remoting-cli-4.9.0.JON320GA/bin/rhq-cli.sh -f
/export/myScripts/my_script.js
```

Example 12. Handling Script Arguments

A feature common to most programming languages is the ability to pass arguments to the program to be executed. In Java, the entry point into a program is a class's *main method*, and it takes a String array as an argument. That array holds any arguments passed to the program. Similarly, arguments can be passed to CLI scripts. Arguments passed to a script can be accessed in the implicit *args array*:

```
if (args.length < 2) {
    throw "Not enough arguments!";
}

for (i in args) {
     println('args[' + i + '] = ' + args[i]);
}
```

The custom **args** variable is only available if it is explicitly defined in a CLI script or if it is added as a module in the ***cliRoot*/rhq-remoting-cli-4.9.0.JON320GA/samples/modules** directory.

In addition to the traditional style of indexed-based arguments, named arguments can also be passed to a script:

```
[jsmith@server ~]$ cliRoot/rhq-remoting-cli-4.9.0.JON320GA/bin/rhq-cli.sh
-s jon-server.example.com -u rhqadmin -p rhqadmin -t 7080 echo_args.js --
args-style=named x=1 y=2
```

The **echo_args.js**, for example, is written to accept the two named option with the script invocation, x and y.

```
for (i in args) {
     println('args[' + i + '] = ' + args[i]);
}
println('named args...');
println('x = ' + x);
println('y = ' + y);
```

This simple script echoes the given x and y values.

```
args[0] = 1
args[1] = 2
named args...

x = 1
y = 2
```

Be aware of the following:

- Explicitly specify that you are using named arguments with the **--args-style** option.
- The values of the named arguments are still accessible through the implicit args array.
- The named arguments, such as **x** and **y**, are bound into the script context as variables.

Example 13. Executing a Single Statement

When running the CLI interactively, commands can still be passed and executed, same as using the **-c** option with **rhq-cli.sh**.

```
localhost:7080> var x = 1
```

Example 14. Executing a Multi-Line Statement

The CLI is a Java shell interpreter, and it can handle multi-line statements properly. To indicate a new line in the same statement, use the backslash (\) character.

```
localhost:7080(rhqadmin)> for (i = 1; i < 3; ++i) { \
localhost:7080(rhqadmin)>    println(i); \
localhost:7080(rhqadmin)> }
1
2

localhost:7080(rhqadmin)>
```

Example 15. Executing a Script

To run a script from a file, use the **-f** option with the **rhq-cli.sh|bat** script.

The **-f** option must give the absolute location of the script, even if it is in the same directory as the **rhq-cli.sh** script. The CLI will not find a script with only a relative path.

```
[jsmith@server ~]$ cliRoot/rhq-remoting-cli-4.9.0.JON320GA/bin/rhq-cli.sh
-u rhqadmin -p rhqadmin -s jon-server.example.com -t 7080 -f
/absolute/path/to/myscript.js
```

Example 16. Executing a Script with Arguments

A script can be written to accept or require arguments; this is described more in Section 6.3, “Defining Arguments and Other Parameters for the CLI Scripts”. Indexed arguments can be passed simply by supplying them in the proper order, as specified in the JavaScript file.

```
[jsmith@server ~]$ cliRoot/rhq-remoting-cli-4.9.0.JON320GA/bin/rhq-cli.sh
-u rhqadmin -p rhqadmin -s jon-server.example.com -t 7080 -f
/absolute/path/to/myscript.js 1 2 3
```

Example 17. Executing a Script with Named Arguments

Script arguments can be indexed or named.

```
[jsmith@server ~]$ cliRoot/rhq-remoting-cli-4.9.0.JON320GA/bin/rhq-cli.sh
-u rhqadmin -p rhqadmin -s jon-server.example.com -t 7080 --args-
style=named -f /absolute/path/to/myscript.js x=1 y=2 y=3
```

6.4. Displaying Pretty-Print Output

The JBoss ON CLI has a special class that formats JBoss ON information into table-style output. This class (**`TabularWriter`**) is implicit for all CLI commands, so almost all output is properly formatted automatically. This class is also available as an implicit variable called *pretty*, which is useful when writing scripts.

For example:

```
rhqadmin@localhost:7080$ criteria = ResourceCriteria()
rhqadmin@localhost:7080$ criteria.addFilterResourceTypeName('service-alpha')
rhqadmin@localhost:7080$ criteria.addFilterParentResourceName('server-omega-
0')
rhqadmin@localhost:7080$ resources =
ResourceManager.findResourcesByCriteria(criteria)
id    name            version resourceType
-----------------------------------------------
11373 service-alpha-8 1.0     service-alpha
11374 service-alpha-1 1.0     service-alpha
11375 service-alpha-0 1.0     service-alpha
11376 service-alpha-4 1.0     service-alpha
11377 service-alpha-2 1.0     service-alpha
11378 service-alpha-3 1.0     service-alpha
11379 service-alpha-5 1.0     service-alpha
11380 service-alpha-9 1.0     service-alpha
11381 service-alpha-6 1.0     service-alpha
11382 service-alpha-7 1.0     service-alpha
10 rows
```

`pretty` formats any object defined in the domain (*org.rhq.core.domain*) package.

Simply printing the output is much less readable:

```
rhqadmin@localhost:7080$ println(resources)
PageList[Resource[id=11373, type=service-alpha, key=service-alpha-8,
name=service-alpha-8, version=1.0],
Resource[id=11374, type=service-alpha, key=service-alpha-1, name=service-
alpha-1, version=1.0],
.... 8< ....
```

For a single object, **`pretty`** checks for the summary information (**`@Summary`**), so that it only displays a subset of information. It then prints the summary information for the single object as a formatted list. For example:

```
rhqadmin@localhost:7080$ pretty.print(resources.get(0))
Resource:
           id: 11373
         name: service-alpha-8
      version: 1.0
 resourceType: service-alpha
```

6.5. Exporting Output

`export` is another implicit script variable that writes output to a specified file. **`exporter`** uses **`pretty.print`** to output all of the information to a plaintext file that matches the table-style formatting used in the interactive display.

```
rhqadmin@localhost:7080$ exporter.setTarget('raw', 'output.txt')
rhqadmin@localhost:7080$ exporter.write(resources)
```

File IO operations like opening or closing the file are not a problem because **exporter** handles the IO operations.

Alternatively, **exporter** can write the raw resource or other information to a CSV file:

```
rhqadmin@localhost:7080$ exporter.setTarget('csv', 'output.csv')
rhqadmin@localhost:7080$ exporter.write(resources)
```

7. Tips and Tricks for Using the CLI

JBoss ON CLI scripts make it possible to automate tasks, from simply importing discovered resources to running complex management operations to remedy configuration drift or schedule web app upgrades.

The JBoss ON CLI has some usability features to make it easier to use interactively for developing scripts, to integrate with system tools to help task automation, and to create custom functions for more versatile, real-life applications.

7.1. Using Tab Complete

In interactive mode, the JBoss ON CLI is aware of the implicit variables (Section 4, “Available Implicit Variables in the JBoss ON API”) in the domain API and the JBoss ON remote API, as well as the specific CLI commands and methods (Section 5, “Methods Specific to the JBoss ON CLI”). When running the CLI interactively, any of these methods can be filled in using **Tab** complete.

Filling in part of a class lists potential matching classes (or, if it matches, commands). For example:

```
[jsmith@server bin]# ./rhq-cli.sh -u rhqadmin -p rhqadmin
RHQ Enterprise Remote CLI 3.2
Remote server version is: 3.2 (2484565)
Login successful
rhqadmin@localhost:7080$ Resource

ResourceFactoryManager   ResourceGroupManager     ResourceManager
ResourceTypeManager

rhqadmin@localhost:7080$ ex

exporter   exec
```

After selecting a class, hitting **Tab** once lists all of the methods for that class.

```
rhqadmin@localhost:7080$ ResourceManager.

availabilitySummary           disableResources
enableResources               findChildResources
findResourceLineage           findResourcesByCriteria
getAvailabilitySummary        getLiveResourceAvailability
getParentResource             getResource
```

```
getResourcesAncestry          liveResourceAvailability
parentResource                resource
toString                      uninventoryResources
updateResource
```

Hitting **Tab** twice prints the full method signatures:

```
rhqadmin@localhost:7080$

              List<Integer> enableResources(GenericArrayTypeImpl[int])
                   Resource updateResource(Resource resource)
              List<Integer> uninventoryResources(GenericArrayTypeImpl[int]
resourceIds)
ResourceAvailabilitySummary getAvailabilitySummary(int resourceId)
         PageList<Resource> findChildResources(int resourceId, PageControl
pageControl)
        Map<Integer,String>
getResourcesAncestry(GenericArrayTypeImpl[Integer] resourceIds,
ResourceAncestryFormat format)
              List<Integer> disableResources(GenericArrayTypeImpl[int])
             List<Resource> findResourceLineage(int resourceId)
       ResourceAvailability getLiveResourceAvailability(int resourceId)
         PageList<Resource> findResourcesByCriteria(ResourceCriteria
criteria)
                   Resource getResource(int resourceId)
                     String toString()
   Resource getParentResource(int resourceId)
```

Autocomplete is very useful for finding what methods or objects are implicitly available, for building criteria-based searches (because it is easier to select criteria), and for developing server scripts.

7.2. Differences Between Running the CLI Interactively and with Files

Scripts can be run in the CLI either by entering the lines directly (in interactive mode), by passing a script file to the CLI using the **-f** option, or by defining the function in a CommonJS module script.

Scripts that are entered interactively and scripts in a file are substantively the same, with one exception. In interactive mode, the CLI has a set of commands (Section 2.3, “Interactive CLI Commands”). These commands are part of the CLI, and most of those commands are *only* available in interactive mode. These commands cannot be referenced inside a referenced script file:

- quit (which exits the CLI)
- record

The login and logout commands are not directly available, either, but there are login and logout functions available which can be used within a script.

```
rhq.login('rhqadmin', 'rhqadmin');

rhq.logout();
```

As with the differences between the GUI, CLI, and alert scripts, this is a matter of context. Most of the CLI-defined commands only make sense within an interactive environment, such as recording inputted commands. Outside that interactive context, other Java methods should be used.

7.3. API Differences Between Resources Types and Versions

One important thing to remember about the structure of objects in JBoss ON is that each resource type is defined individually, in its own plug-in descriptor. (There can be multiple resource types in a single plug-in descriptor, and these are all related as parent and children resource types.)

Each resource type has different management APIs, which reflect the software application. Obviously, a Tomcat server is different than a Postgres database, so they have different metrics, operations, and configuration properties. Even seemingly common traits like stopping and starting a resource are different depending on the resource type descriptor. Tomcat has an operation to stop gracefully (something not needed for a database or most services), while JBoss AS 5 servers are stopped and then started as separate tasks instead of restarting in one step.

An agent plug-in defines a resource type not only by the application or service, but also by the version of that application or service. An EJB resource is difference than an EJB3 resource, with a different definition.

This distinction — that different versions of the same application are treated as different resource types — is particularly critical for server resources.

Different resource types have configuration properties and operations available. This can impact CLI scripts written to manage those resources. For example, a script written for a JBoss AS 5 resource may not work for a JBoss AS 4 or JBoss AS 7 resource because the different resources have different APIs.

If you are writing a script which may be used for different versions of a server, create a CLI script which first identifies the resource type and then calls the appropriate method.

7.4. Available Utility and Sample Scripts

Several scripts are provided that supply additional functionality, like simplified functions for parsing search results, deploying bundles, or managing drift. These *utility scripts* add functions without performing any concrete task. They are available as a reference for and extension to custom scripts. These example functions can be copied into a script file or can be loaded before a script is run, as in Section 3.1, “About Script Dependencies and Exporting Functions”.

A couple of scripts are available that both provide additional functions (for reference or ease of use) and also perform management tasks when run. This scripts can be used directly to manage resources, without having to write a custom script.

Important

Utility and sample scripts have not been extensively tested and may not be updated between releases. These are included for convenience and for real-life, complete examples of writing server scripts.

Test these scripts for quality and consistency within your environment.

Table 6. Utility and Sample Scripts

Script	Description	Location
CommonJS Module Utility Scripts		

Script	Description	Location
util.js	Defines search functions for iterating through an array of object (**`foreach`**), to return the first matching object (**`find`**) or to return all matching objects (**`findAll`**). It also has functions for converting information from JavaScript hashes to JBoss ON configuration objects and back. The same as the **`samples/util.js`** script, but written as a CommonJS module.	*cliRoot*/rhq-remoting-cli-4.9.0.JON320GA/samples/modules/
bundles.js	Defines functions to create and deploy a bundle, create a bundle destination, or get information on supported base directories for a resource. The same as the **`samples/bundles.js`** script, but written as a CommonJS module.	*cliRoot*/rhq-remoting-cli-4.9.0.JON320GA/samples/modules/
drift.js	Defines functions to create and diff snapshots, get a definition, and show the history for a resource or specific file. The same as the **`samples/drift.js`** script, but written as a CommonJS module.	*cliRoot*/rhq-remoting-cli-4.9.0.JON320GA/samples/modules/
jbossas.js	Defines several functions to manage EAP 6 / AS 7 resources, including deploying web applications and managing EAP clusters. The same as the **`samples/deploy-to-and-restart-JBAS.js`** and **`samples/add-as7-standalone-server-to-cluster.js`** scripts, but combined and written as a CommonJS module.	*cliRoot*/rhq-remoting-cli-4.9.0.JON320GA/samples/modules/
Utility Scripts		
util.js	Defines search functions for iterating through an array of object (**`foreach`**), to return the first matching object (**`find`**) or to return all matching objects (**`findAll`**). It also has functions for converting information from JavaScript hashes to JBoss ON configuration objects and back.	*cliRoot*/rhq-remoting-cli-4.9.0.JON320GA/samples/
measurement_utils.js	Defines functions to enable, disable, or update metric schedules.	*cliRoot*/rhq-remoting-cli-4.9.0.JON320GA/samples/

Script	Description	Location
bundles.js	Defines functions to create and deploy a bundle, create a bundle destination, or get information on supported base directories for a resource.	*cliRoot*/rhq-remoting-cli-4.9.0.JON320GA/samples/
drift.js	Defines functions to create and diff snapshots, get a definition, and show the history for a resource or specific file.	*cliRoot*/rhq-remoting-cli-4.9.0.JON320GA/samples/
Sample Scripts		
add-as7-standalone-server-to-cluster.js	Defines a series of functions that add a specified AS 7 to a cluster.	*cliRoot*/rhq-remoting-cli-4.9.0.JON320GA/samples/
deploy-to-and-restart-JBAS.js	Defines two functions to deploy new content to EAP servers and to update existing content. With the proper variables set, this script can be run directly to deploy content and restart the given app server.	*cliRoot*/rhq-remoting-cli-4.9.0.JON320GA/samples/
fix-bundle-deployment.js	An example server-side alert script that reverts a bundle deployment to a specified version.	*serverRoot*/jon-server-3.2/jbossas/server/default/deploy/rhq.ear/rhq-downloads/cli-alert-scripts/

7.5. Defining Custom Functions

Custom functions are permissible and supported in CLI scripts as long as they are properly structured and conform to the JBoss ON remote API.

7.6. Scheduling Script Runs with Cron

Administrators may create some scripts to perform routine maintenance on their JBoss ON systems, such as deleting old bundle versions to improve database performance or to add newly-discovered resources to the inventory automatically.

Scripts can be run through the JBoss ON CLI using a simple cron job. For example, an administrator writes **`import.js`** to add discovered resources to the inventory automatically and runs it daily:

```
vim /etc/cron.daily/rhq-cli

#!/bin/sh
# run JON CLI every morning to import new resources
cliRoot/rhq-remoting-cli-4.9.0.JON320GA/bin/rhq-cli.sh -u rhqadmin -p
rhqadmin -f /export/scripts/import.js
```

More information about scheduling jobs with cron is available in the crontab manpage.

7.7. Using Wrapper Scripts

Ultimately, JBoss ON CLI scripts are regular JavaScript files. For simplicity or ease of programming, you may write a number of different scripts, each performing a different task or for a different resource type.

This collection of scripts can be controlled through wrapper scripts. Using simple shell scripts can make

running the JBoss ON CLI scripts more user-friendly (by providing a nicer interface) or it can make it easier to pass variables to manage specific resources.

There is a simplistic example of using a wrapper script in Section 6, “Example: Managing Grouped Servers (JBoss EAP 5)”.

7.8. Permissions and Setup for JBoss ON Users

Every CLI script, either for a logged in user or through the **`rhq.login`** method, is run on the JBoss ON server by a JBoss ON user.

The JBoss ON CLI does enforce access controls to resources touched by JBoss ON CLI scripts. This means that the script's user must meet the same authorization requirements as a GUI user to perform the given operation.

Additionally, any user who runs a CLI script must *already exist* in JBoss ON as a user. When the JBoss ON server is configured to create LDAP users, these users are created automatically the first time the LDAP user logs into the JBoss ON web UI. The same account creation does not occur when logging in with the CLI. Therefore, LDAP users must log into the web UI before they can run the JBoss ON CLI.

CLI scripts follow a certain flow of events. Each functional element within that workflow, such as searching for a resource or object and then running an operation, has its own classes in the remote API.

Most actions in JBoss ON CLI scripts are repeatable — the part for running a search for a platform or for collecting live data for a metric is pretty similar across CLI scripts.

This section provides some basic examples for running a single, specific functional task. These individual script examples can be used consistently in larger script workflows and as part of task automation.

1. Searches

All object managers define operations for retrieving data. Most of the managers define *criteria-based* operations for data retrieval — meaning that the search can be based on attributes within the JBoss ON objects.

Criteria-based searches have methods in the form **`find`***Object*`ByCriteria`, so a resource find method is **`findResourcesByCriteria`** and a group find method is **`findResourceGroupsByCriteria`**.

Searches are translated into a corresponding JPA-QL query.

The criteria classes reside in the *org.rhq.core.domain.criteria* package.

1.1. Setting Basic Search Criteria

The simplest criteria is to define results based on what they are, such as resource type, without any additional search parameters.

For example, this fetches all committed resources in the inventory because it has no filters to limit by resource type or ID.

```
rhqadmin@localhost:7080$ var criteria = new ResourceCriteria() // this sets
the criteria to use for the search
rhqadmin@localhost:7080$ criteria.clearPaging() // this clears the 200 item
page size to return all entries
rhqadmin@localhost:7080$ var resources =
ResourceManager.findResourcesByCriteria(criteria) // this actually runs the
```

```
search
rhqadmin@localhost:7080$ pretty.print(resources) // this prints the search
results
id    name   versio curren resour
------------------------------------------------------------------------
-------
10001 server Linux  UP     Linux
10002 server AS 4.2 UP     JBossA
10392 full-h        UP     Profil
10014 AlertH        UP     EJB3 S
10015 Adviso        UP     EJB3 S
10016 DataAc        UP     EJB3 S
10017 Affini        UP     EJB3 S
10011 Access        UP     Access
10391 ha            UP     Profil
...8<...
```

While the **`findResourcesByCriteria()`** is what runs the search, the **`pretty.print`** method is required to display the results.

This basic criteria search is translated into the following JPA-QL query:

```
SELECT r
FROM Resource r
WHERE ( r.inventoryStatus = InventoryStatus.COMMITTED
```

1.2. Using Sorting

The basic search criteria can be refined so that the resource results are sorted by plug-in.

To add sorting, call **`criteria.addSortPluginName()`**. Sorting criteria have methods in the form **`addSortXXX(PageOrdering order)`**.

For example:

```
rhqadmin@localhost:7080$ var criteria = new ResourceCriteria()
rhqadmin@localhost:7080$ criteria.addSortPluginName(PageOrdering.ASC) //
adds a sort order to the results
rhqadmin@localhost:7080$ resources =
ResourceManager.findResourcesByCriteria(criteria)
```

This criteria is translated into the following JPA-QL query:

```
SELECT r
FROM Resource r
WHERE ( r.inventoryStatus = InventoryStatus.COMMITTED )
ORDER BY r.resourceType.plugin ASC
```

1.3. Using Filtering

Adding additional matching criteria, like resource name in this example, further narrows the search results. To add filtering to any criteria, use methods of the form **`addFilterXXX()`**.

```
rhqadmin@localhost:7080$ var criteria = new ResourceCriteria()
```

```
rhqadmin@localhost:7080$ criteria.addFilterResourceTypeName('JBossAS
Server') // a search filter
rhqadmin@localhost:7080$ resources =
ResourceManager.findResourcesByCriteria(criteria)
```

The resulting JPA-QL query is as follows:

```
SELECT r
FROM Resource r
WHERE ( r.inventoryStatus = InventoryStatus.COMMITTED
AND LOWER( r.resourceType.name )  like 'JBossAS Server' ESCAPE '\\' )
```

This code retrieves all JBoss servers in the inventory. There can be multiple filters used with a single search. For example, this searches for JBoss servers that have been registered by a particular agent:

```
rhqadmin@localhost:7080$ var criteria = new ResourceCriteria()
rhqadmin@localhost:7080$ criteria.addFilterResourceTypeName('JBossAS
Server')
rhqadmin@localhost:7080$
criteria.addFilterAgentName('localhost.localdomain')
rhqadmin@localhost:7080$ resources =
ResourceManager.findResourcesByCriteria(criteria)
```

This generates the following JPA-QL query:

```
SELECT r
FROM Resource r
WHERE ( r.inventoryStatus = InventoryStatus.COMMITTED
AND LOWER( r.agent.name )  like 'localhost.localdomain' ESCAPE '\\' )
```

1.4. Fetching Associations

An *association* shows the hierarchy of parent and child resources. After retrieving the resources, it is possible to view their associated parent or child resources using a special *fetch* method.

Simply printing a list of children after a search will fail, even if the given server has child resources.

```
...8<...
rhqadmin@localhost:7080$ resource = resources.get(0)
rhqadmin@localhost:7080$ if (resource.childResources == null) print('no
child resources')
```

The reason for this is that lazy loading is used throughout the domain layer for one-to-many and many-to-many associations. Since child resources are lazily loaded, the list of children has to explicitly set as a fetch in the search criteria.

For example:

```
rhqadmin@localhost:7080$ criteria.addFilterResourceTypeName('JBossAS
Server')
rhqadmin@localhost:7080$ criteria.fetchChildResources(true)
rhqadmin@localhost:7080$ resources =
ResourceManager.findResourcesByCriteria(criteria)
rhqadmin@localhost:7080$ resource = resources.get(0)
```

```
rhqadmin@localhost:7080$ if (resource.childResources == null) print('no
child resources'); else pretty.print(resource.childResources)
id  name                                 versio resourceType
------------------------------------------------------------
222 AlertManagerBean                     EJB3 Session Bean
222 SchedulerBean                        EJB3 Session Bean
222 AlertDefinitionManagerBean           EJB3 Session Bean
222 AlertConditionConsumerBean           EJB3 Session Bean
222 PartitionEventManagerBean            EJB3 Session Bean
...8<...
```

The output varies depending on what is inventoried. These are the child resources of the JBoss ON server. The JPA-QL query that is generated appears as follows:

```
SELECT r
FROM Resource r
LEFT JOIN FETCH r.childResources
WHERE ( r.inventoryStatus = InventoryStatus.COMMITTED
AND LOWER( r.resourceType.name )  like 'JBossAS Server' ESCAPE '\\' )
```

1.5. Setting Page Sizes

Almost all searches return a paged list of results. By default, paged results are capped at 200 entries. So, for example, attempting to return all resources in the inventory only returns the first 200 resources, while querying the database directly may return several hundred resources.

The **`Criteria`** class defines some methods which can be used to control page sizes for search results.

Example 1. Clearing the Page Size

If there are more than 200 results, and all matching resources need to be contained in a single results set, the page limit can be cleared with the **`clearPaging`**

```
var criteria = new ResourceCriteria()
criteria.clearPaging()
var resources = ResourceManager.findResourcesByCriteria(criteria)
```

Example 2. Setting a Page Size

There can be times when a different page limit needs to be used, but for clarity or control, some paging still needs to be in effect.

The **`setPaging`** method sets the number of pages and the page size for the given search. Generally, since there is only a single page, the page number is set to 0, and then the page size can be reset higher or lower, as desired.

```
rhqadmin@localhost:7080$ var criteria = new ResourceCriteria()

rhqadmin@localhost:7080$ criteria.getPageSize()
200

rhqadmin@localhost:7080$ criteria.getPageNumber()
0
```

```
rhqadmin@localhost:7080$ criteria.setPaging(0,300)

rhqadmin@localhost:7080$ var resources =
ResourceManager.findResourcesByCriteria(criteria)
id    name                                          currentAvailability
resourceType
-------------------------------------------------------------------------
-----------------------------------------
10032 RHQDS                                         UP
Datasource
10033 ResourceFactoryManagerBean                    UP
EJB3 Session Bean
10034 CoreTestBean                                  UP
EJB3 Session Bean
10035 rhq-postinstaller.war (//localhost/installer) UP                Web
Application (WAR)
10036 ResourceMetadataManagerBean                   UP
EJB3 Session Bean
10037 SystemInfoManagerBean                         UP
EJB3 Session Bean
10105 wstools.sh                                    UP
Script
10038 PartitionEventManagerBean                     UP
EJB3 Session Bean
10039 CallTimeDataManagerBean                       UP
EJB3 Session Bean
10040 AlertDefinitionManagerBean                    UP
EJB3 Session Bean
10041 DiscoveryTestBean                             UP
EJB3 Session Bean
10123 wsconsume.sh                                  UP
Script
10042 ROOT.war (//localhost/)                       UP
Web Application (WAR)
10044 AlertManagerBean                              UP
EJB3 Session Bean
10045 AgentManagerBean                              UP
EJB3 Session Bean
... 8< ...
300 rows
```

2. Getting the JBoss ON ID for an Object

Everything in JBoss ON — resources, configuration properties, bundle archives, templates, alerts, everything — is identified with a unique ID number. Most of these ID numbers are visible in the web UI as part of the URL which goes to that object's details page.

Not every ID is easily found, though. For example, searching for a metric template requires the resource type ID, but the resource type ID is hard to find.

In other instances, you may want to run a script against multiple resources, which means there are multiple resource IDs which may not be explicitly known at the time you write the script.

Searching for a resource (Section 1, “Searches”) retrieves the resource's details, including its ID.

```
rhqadmin@localhost:7080$ criteria = new ResourceCriteria();
ResourceCriteria:
        inventoryManagerRequired: false
                 persistentClass: class
org.rhq.core.domain.resource.Resource

rhqadmin@localhost:7080$ criteria.addFilterResourceTypeName('Linux')

rhqadmin@localhost:7080$ ResourceManager.findResourcesByCriteria(criteria);
one row
Resource:
                          id: 10001
                        name: gs-dl585g2-01.rhts.eng.bos.redhat.com
                     version: Linux 2.6.32-220.el6.x86_64
        currentAvailability: UP
               resourceType: Linux
```

This is true for any **`*Criteria`** search. It retrieves the ID for the object is searches for, even if that ID is not explicitly displayed.

This ID can then be used in other operations.

```
rhqadmin@localhost:7080$ var metrics =
MeasurementDataManager.findLiveData(resources.get(0).id, [mdefs.get(0).id]);
```

3. Getting Data for Single and Multiple Resources

When you run any sort of operation or function, you frequently have to pull the results from that operation, the data, into another function.

For example, this exports and writes whatever the results from the *data* variable was:

```
exporter.write(data.get(0))
```

Using **`get(0)`** takes the first object that was returned and uses it.

There can be instances when there are a lot of objects returned that could potentially be used in the next step of the script, and there are a couple of different ways to handle it.

One option, instead of **`get(0)`**, is to work incrementally through each object in a list. For example, this gets the data for each metric definition (i) for a resource.

Example 3. get(i)

```
if( mdefs != null ) {
  if( mdefs.size() > 1 ) {
     for( i =0; i < mdefs.size(); ++i) {
          mdef = mdefs.get(i);
          var data =
MeasurementDataManager.findDataForResource(resources.get(0).id,
[mdef.id],start,end,1)
```

```
   exporter.write(data.get(0));
     }
  }
  ...8<...
```

Other objects can be sent to an array, and then the operation or task can be performed for each object in that array. This example searches for all JBoss AS 5 servers, sends the results to an array, and then performs a restart operation on each server.

Example 4. An Array

```
//find the resources
//use a plugin filter to make sure they are all of the same type
criteria = new ResourceCriteria();
criteria.addFilterPluginName('JBossAS5')

var resources =
ResourceManager.findResourcesByCriteria(criteria).toArray();
var resType = ResourceTypeManager.getResourceTypeByNameAndPlugin('JBossAS
Server', 'JBossAS5');

// go through the array
var idx=0;
var jbossServers = new Array();

for( i in resources ) {
     if( resources[i].resourceType.id == resType.id ) {
          jbossServers[idx] = resources[i];
          idx = idx + 1;
     }
}

// restart the resources
for( a in resources ) {
     var jboss = ProxyFactory.getResource(jbossServers[a].id);
     jboss.restart()
}
```

The **`util.js`** support script has a **`foreach`** function that also iterates cleanly through arrays, collections, and maps, as well as generic objects.

Which method you use to handle multiple objects — or whether you even want to handle multiple objects — depends on the type of information and the purpose of the script. These are just some options to keep in mind.

4. Resources and Groups

4.1. Creating and Updating Content-Backed Resources (Web Apps)

Web applications (EARs and WARs) are *content-backed resources*, a cross between a managed resource and a content package.

When running a CLI script, there are some methods available specifically to retrieve or create the content for the resource.

To create a content-backed resource, upload the file with a specified version number.

This script's steps are:

1. Search for the resource to upload the content to. This example looks for a JBoss AS 5 server.
2. Make sure the server is running. The JBoss server has to be running for content to be deployed successfully.

Example 5. Creating a Content-Backed Resource

```
// fill this information in before running the script
var pathName = '/home/jon/myExampleApp.ear'

var resTypeName = 'JBossAS Server'
var pluginName = "JBossAS5"
var appTypeName = "Enterprise Application (EAR)"

// define a custom function to parse the filename and path info
function PackageParser(pathName) {
    var file = new java.io.File(pathName);
    var fileName = file.getName();
    var packageType = fileName.substring(fileName.lastIndexOf('.')+1);
    var tmp = fileName.substring(0, fileName.lastIndexOf('.'));
    var version = 1;
    var realName = tmp;
    var packageName = fileName;

    // parse the package version, only if version is included
    if(tmp.indexOf('-') != -1){
 realName = tmp.substring(0, tmp.lastIndexOf('-'));
 version = tmp.substring(tmp.lastIndexOf('-') + 1);
        packageName = realName + "." + packageType;
    }

    this.packageType = packageType.toLowerCase();
    this.packageName = packageName;
    this.version     = version;
    this.realName    = realName;
    this.fileName    = fileName;
}

criteria = new ResourceCriteria();
criteria.addFilterResourceTypeName(resTypeName);
criteria.addFilterPluginName(pluginName);
var resources = ResourceManager.findResourcesByCriteria(criteria);

// create the config options for the new EAR
var deployConfig = new Configuration();
deployConfig.put( new PropertySimple("deployExploded", "false"));
deployConfig.put( new PropertySimple("deployFarmed", "false"));
```

```
// stream in the file bytes
var file = new java.io.File(pathName);
var inputStream = new java.io.FileInputStream(file);
var fileLength = file.length();
var fileBytes = java.lang.reflect.Array.newInstance(java.lang.Byte.TYPE,
fileLength);
for (numRead=0, offset=0; ((numRead >= 0) && (offset < fileBytes.length));
offset += numRead ) {
    numRead = inputStream.read(fileBytes, offset, fileBytes.length -
offset);
}

// parse the filename and path info
PackageParser(pathName);

// identifies the type of resource being created
var appType =
ResourceTypeManager.getResourceTypeByNameAndPlugin(appTypeName,
pluginName);

// create the new EAR resource on each discovered app server
if( resources != null ) {
  for( i =0; i < resources.size(); ++i) {
       var res = resources.get(i);println("res: " + res);
       ResourceFactoryManager.createPackageBackedResource(
           res.id,
           appType.id,
           packageName,
           null,  // pluginConfiguration
           packageName,
           version,
           null, // architectureId
           deployConfig,
           fileBytes,
           null // timeout
       );
  }
}
```

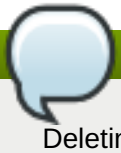
Updating a package is slightly simpler. It requires sending a new file and version number to the existing EAR resource.

Example 6. Updating a Content-Backed Resource

```
// update this
var fullPathName = '/export/myfiles/updatedApp.ear'

// define a custom function to parse the filename and path info
function PackageParser(pathName) {
    var file = new java.io.File(pathName);

    var fileName = file.getName();
    var packageType = fileName.substring(fileName.lastIndexOf('.')+1);
    var tmp = fileName.substring(0, fileName.lastIndexOf('.'));
```

```
    var version = 1;
    var realName = tmp;
    var packageName = fileName;

    // parse the package version, only if version is included
    if(tmp.indexOf('-') != -1){
        realName = tmp.substring(0, tmp.lastIndexOf('-'));
        version = tmp.substring(tmp.lastIndexOf('-') + 1);
        packageName = realName + "." + packageType;
    }

    this.packageType = packageType.toLowerCase();
    this.packageName = packageName;
    this.version     = version;
    this.realName    = realName;
    this.fileName    = fileName;
}

// parse the filename and path info
PackageParser(fullPathName);

// search for the JBoss AS 5 server by name
criteria = new ResourceCriteria();
criteria.addFilterName('My JBoss AS 5 Server');
var res = ResourceManager.findResourcesByCriteria(criteria);

var jboss = ProxyFactory.getResource(res.get(0).id);

var children = jboss.children;
for( c in children ) {
    var child = children[c];
    if( child.name == packageName ) {
        child.updateBackingContent(fullPathName,version);
    }
}
```

While a content-backed resource is added or updated as if it were a software package (which it is), it is deleted as if it were a resource.



Note

Deleting is not the same as uninventorying a resource. Uninventorying removes the resource from the inventory but leaves it intact on the platform. Deleting a resource deletes it from the platform itself.

Example 7. Deleting a Content-Backed Resource

```
// search for the content resource by name
criteria = new ResourceCriteria();
criteria.addFilterName('updatedApp.ear');
var res = ResourceManager.findResourcesByCriteria(criteria);

ResourceFactoryManager.deleteResource(res.get(0).id)
```

4.2. Creating a Resource Group and Adding Members

When a group is created by a script, it is initially created as a mixed group. All empty groups are treated as mixed groups. Once members are added, if all the members are the same resource type, then the group is automatically tagged as a compatible group.

Compatible groups are great for scripted management tasks, because the same operation or configuration change can be performed on each member iteratively.

The script steps are:

1. Search for the resource type.
2. Create the group, based on the resource type.
3. Find resources of that resource type.
4. Iterate through the returned resources and add them to the group.

Example 8. Annotated Example

```
// search for the resource type to use for the compat group
var resType =
ResourceTypeManager.getResourceTypeByNameAndPlugin("Linux","Platforms");

//create the new resource group
var rg = new ResourceGroup(resType);
rg.setRecursive(false);
rg.setName('Linux Group - ' + java.util.Date());

rg = ResourceGroupManager.createResourceGroup(rg);

//find resources to add to the group based on their resource type
criteria = new ResourceCriteria();
criteria.addFilterResourceTypeId(resType.id);

var resources = ResourceManager.findResourcesByCriteria(criteria);

// add the found resources to the group
if( resources != null ) {
  for( i =0; i < resources.size(); ++i) {
       var resource = resources.get(i);
       ResourceGroupManager.addResourcesToGroup(rg.id, [resource.id]);
  }
}
```

4.3. Creating and Editing Dynagroups

A dynamic groups specifies a search term to use to search the inventory and identify matching resources to belong to the group. Since the search results change automatically as results are added and removed from the inventory, the group membership is always changing and always current.

Dynamic groups (or *dynagroups*) have their own classes to create and update groups.

A dynagroup is created by defining a set of expressions which are search filters to use to search the inventory for matching members. The group membership is calculated (using those expressions) by an administrative task.

Note

Dynagroups expressions are covered extensively in the *Users Guide*.

Both for creating an editing a dynagroup, then, it is necessary to recalculate the group membership to capture changes to matching members. It can also be reasonable to recaculate group membership whenever there are changes to the inventory, to update the dynagroups with new members.

Example 9. Creating a Dynagroup

This uses simple expressions to identify group members.

```
// create the new dynagroup
var dynaGroupDef = new GroupDefinition("Linux Group");
var expr = "resource.type.name=Linux" + "\n" +
"resource.type.category=Platform"
dynaGroupDef.setExpression(expr);
var def = GroupDefinitionManager.createGroupDefinition(dynaGroupDef);

// calculate the group members
GroupDefinitionManager.calculateGroupMembership(def.getId());
```

It is possible to use more complex expressions such as *groupby* within the **`setExpression`** value.

Example 10. Editing a Dynagroup

This adds another expression to the dynagroup, updates the definition, and then recalculates the group membership.

```
// search for the dynagroup
criteria = new ResourceGroupDefinitionCriteria()
criteria.addFilterName("All agents");
var orig = GroupDefinitionManager.findGroupDefinitionsByCriteria(criteria)

// get the dynagroup entry
var originalGroupDef = orig.get

// add the new expression
originalGroupDef.setExpression("resource.name=*.example.com");
var def = GroupDefinitionManager.updateGroupDefinition(originalGroupDef);

// calculate the group members
GroupDefinitionManager.calculateGroupMembership(def.getId());
```

Dynagroup memberships are only updated after a calculation operation. So, if there are changes to the inventory, even if there are not changes to the dynagroup expression itself, the dynagroup must be updated to update the members.

Example 11. Recalculating a Group Definition

This updates a single dynagroup, based on the group name.

```
// search for the dynagroup
criteria = new ResourceGroupDefinitionCriteria()
criteria.addFilterName("All agents");
var orig = GroupDefinitionManager.findGroupDefinitionsByCriteria(criteria)

// calculate the group members
GroupDefinitionManager.calculateGroupMembership(orig.getId());
```

It is possible to recalculate all dynagroups or a set of dynagroups, by incrementing through the search results and running the ***`calculatedGroupMembership()`*** method. This is covered in Section 3, “Getting Data for Single and Multiple Resources”.

5. Resource Configuration

There are two areas of resource configuration:

- The plug-in configuration. In the web UI, this is called the *connection setting*; it is the information which the agent uses to discover and connect to the resource, such as a PID file path or a port number.

  Every resource has some kind of plug-in configuration.

- Resource configuration. Configuration properties are drawn from the configuration files for a resource, such as **`.conf`** or **`.xml`**. The configuration files and properties that are exposed within JBoss ON are defined in the plug-in descriptor.

  Resource configuration is optional, and many resources do not support configuration editing or only expose a subset of possible properties.

Note

When changing the configuration for a resource, it is simplest to use the configuration editor with a proxy resource, as in Example 8, “Changing Configuration Properties”.

The configuration editor is only available with the interactive CLI; for running an alert script or script automatic changes to resource configuration, the remote API can be used directly.

5.1. Viewing Current Configuration

Plug-in (connection settings) configuration and resource configuration are both defined per resource type, in the resource type's plug-in descriptor.

The **`get*ConfigurationDefinitionForResourceType`** methods can display the descriptor-defined templates used for the resource *type's* configuration. These are the configuration properties available to every resource of that type.

Example 12. Viewing Plug-in Configuration for a Resource Type

```
rhqadmin@localhost:7080$ var res =
ResourceTypeManager.getResourceTypeByNameAndPlugin('Linux','Platforms')
//get the resource type ID

rhqadmin@localhost:7080$
ConfigurationManager.getPluginConfigurationDefinitionForResourceType(res.i
d) //use the type ID to search for the resource type template

ConfigurationDefinition:
         configurationFormat: Structured
             defaultTemplate: ConfigurationTemplate[id=10443,
name=default, config=Linux]
                 description:
            groupDefinitions: [PropertyGroupDefinition[id=10058,
name=Content], PropertyGroupDefinition[id=10059, name=Event Logs]]
                          id: 10437
                        name: Linux
        nonGroupedProperties: []
         propertyDefinitions: {logs=PropertyDefinitionList[id=11792,
name=logs, config=Linux, group=Event Logs],
metadataCacheTimeout=SimpleProperty[metadataCacheTimeout] (Type: INTEGER)
(Group: Content),
enableContentDiscovery=SimpleProperty[enableContentDiscovery] (Type:
BOOLEAN)(Group: Content),
enableInternalYumServer=SimpleProperty[enableInternalYumServer] (Type:
BOOLEAN)(Group: Content), yumPort=SimpleProperty[yumPort] (Type: INTEGER)
(Group: Content)}
              templates: {default=ConfigurationTemplate[id=10443,
name=default, config=Linux]}
```

The **`propertyDefinitions`** parameter contains the information about the configuration properties that can be set, including details about the configuration property setup. For example, for the event logs configuration:

```
logs=[id=11792, name=logs, config=Linux, group=Event Logs]
```

The property name is *logs*, and the type of property is a list, *PropertyDefinitionList*. The property list ID is *11792*, though not every type of property has a template ID. The event log properties are organized together in the *Event Logs* group.

The cache timeout property has a slightly different configuration. This is a simple property type (*SimpleProperty*), and the expected value is given in the *Type:* element. The group for the property is *Content*.

```
metadataCacheTimeout=SimpleProperty[metadataCacheTimeout] (Type: INTEGER)
(Group: Content)
```

Group settings for a property are mainly used to create a logical organization in the web UI. For the event logs configuration, all of the members in the properties list belong to the group, and only those values. For the cache timeout, it belongs to the content group but other, discrete properties also belong to the group, so the way the group is defined is different.

The resource configuration template can be retrieved similarly to retrieving the plug-in configuration template.

Example 13. Viewing the Configuration Properties for the Resource Type

```
rhqadmin@localhost:7080$ var res =
ResourceTypeManager.getResourceTypeByNameAndPlugin('Samba Server',
'Samba') //get the resource type ID

rhqadmin@localhost:7080$
ConfigurationManager.getResourceConfigurationDefinitionForResourceType(res
.id)
ConfigurationDefinition:
         configurationFormat: Structured
             defaultTemplate: ConfigurationTemplate[id=11123,
name=default, config=Samba Server]
                 description:
            groupDefinitions: [PropertyGroupDefinition[id=10905,
name=Basic Configurations], PropertyGroupDefinition[id=10906,
name=Security], PropertyGroupDefinition[id=10907, name=Printing],
PropertyGroupDefinition[id=10908, name=Active Server Directory]]
                           id: 11087
                         name: Samba Server
        nonGroupedProperties: []
         propertyDefinitions: {winbind enum groups=SimpleProperty[winbind
enum groups] (Type: BOOLEAN)(Group: Active Server Directory), winbind
separator=SimpleProperty[winbind separator] (Type: STRING)(Group: Active
Server Directory), cups options=SimpleProperty[cups options] (Type:
STRING)(Group: Printing), workgroup=SimpleProperty[workgroup] (Type:
STRING)(Group: Basic Configurations), encrypt
passwords=SimpleProperty[encrypt passwords] (Type: BOOLEAN)(Group:
Security), winbind enum users=SimpleProperty[winbind enum users] (Type:
BOOLEAN)(Group: Active Server Directory),
security=SimpleProperty[security] (Type: STRING)(Group: Security),
template shell=SimpleProperty[template shell] (Type: STRING)(Group: Active
Server Directory), password=SimpleProperty[password] (Type: PASSWORD)
(Group: Active Server Directory), load printers=SimpleProperty[load
printers] (Type: BOOLEAN)(Group: Printing),
username=SimpleProperty[username] (Type: STRING)(Group: Active Server
Directory), realm=SimpleProperty[realm] (Type: STRING)(Group: Active
Server Directory), idmap gid=SimpleProperty[idmap gid] (Type: STRING)
(Group: Active Server Directory), server string=SimpleProperty[server
string] (Type: STRING)(Group: Basic Configurations),
controller=SimpleProperty[controller] (Type: STRING)(Group: Active Server
Directory), enableRecycleBin=SimpleProperty[enableRecycleBin] (Type:
BOOLEAN)(Group: Basic Configurations), idmap uid=SimpleProperty[idmap uid]
(Type: STRING)(Group: Active Server Directory)}
                   templates: {default=ConfigurationTemplate[id=11123,
name=default, config=Samba Server]}
```

The property values for a specific resource can be viewed using the **`get*Configuration`** methods.

Note

If you are only viewing the resource configuration and are running the interactive CLI, this is done more easily using a proxy resource, as in Example 8, “Changing Configuration Properties”.

Example 14. Viewing a Resource's Configuration Settings

```
rhqadmin@localhost:7080$ criteria = new ResourceCriteria(); // find the
resource
rhqadmin@localhost:7080$ criteria.addFilterResourceTypeName('Samba')
rhqadmin@localhost:7080$ criteria.addFilterAgentName('agent1.example.com')
rhqadmin@localhost:7080$ var resource =
ResourceManager.findResourcesByCriteria(criteria);

rhqadmin@localhost:7080$
ConfigurationManager.getResourceConfiguration(resource.get(0).id)
Configuration [12082] - Loaded from Augeas at Wed May 02 12:04:24 EDT 2012
  winbind separator = null
  winbind enum groups = null
  cups options = null
  workgroup = null
  winbind enum users = null
  encrypt passwords = null
  security = null
  template shell = null
  password = null
  load printers = null
  username = null
  realm = null
  idmap gid = null
  server string = null
  controller = null
  enableRecycleBin = false
  idmap uid = null
```

5.2. Changing Simple Configuration Properties

The way that a configuration property is edited depends on the type of property, whether it is simple, list, or map. The process for changing configuration is the same for both plug-in configuration properties and resource configuration properties.

The steps in this script:

1. Search for the resource.
2. Create a configuration object.
3. Set the new property value. The method to set the property value depends on the format of the property. In this case, it uses **`setSimpleValue`** since this is a simple property. For a simple property, the value is set by passing (*'property','value'*).

4. Run an update operation for the resource. For a resource configuration update, the method is **`updateResourceConfiguration`**. For a plug-in configuration update, it is **`updatePluginConfiguration`**.

Example 15. Changing a Simple Property

```
// find the resource
criteria = new ResourceCriteria();
criteria.addFilterResourceTypeName('Samba')
// this only updates the resource for this specific agent
criteria.addFilterAgentName('agent1.example.com')
var resources = ResourceManager.findResourcesByCriteria(criteria);

//get current configuration
var config =
ConfigurationManager.getResourceConfiguration(resources.get(0).id);

//set the new value in the form 'property', 'value'
config.setSimpleValue("workgroup","example")

// run the update operation
ConfigurationManager.updateResourceConfiguration(resources.get(0).id,confi
g)
```

6. Operations

6.1. Starting and Stopping a Resource

A resource can simple be started by using an operation.

This example looks for a specific resource by name, and then runs the **`start()`** function.

Example 16. Simple Start

```
//find the resource
criteria = new ResourceCriteria();
criteria.addFilterName('My JBossAS')

var servers = ResourceManager.findResourcesByCriteria(criteria);
var myJBossAS = ProxyFactory.getResource(servers.get(0).id)
myJBossAS.start()
```

Each resource type has its own defined operations, and even simple tasks like start and stop may have different methods depending on the resource. Try using a proxy resource and then the **`operations`** method to list the available operations.

```
rhqadmin@localhost:7080$ server.operations
Array of org.rhq.bindings.client.ResourceClientProxy$Operation
name     description
------------------------------------------------------------------------
```

```
restart  Shutdown and then start this application server.
start    Start this application server.
shutdown Shutdown this application server via script or JMX.
3 rows
```

A more complex start or stop script can be used to iterate over an array of resources of the same type.

Example 17. Starting an Array 1

```
//find the resources
//use a plugin filter to make sure they are all of the same type
criteria = new ResourceCriteria();
criteria.addFilterPluginName('JBossAS5')

var resources =
ResourceManager.findResourcesByCriteria(criteria).toArray();
var resType = ResourceTypeManager.getResourceTypeByNameAndPlugin('JBossAS
Server', 'JBossAS5');

// go through the array
var idx=0;
var jbossServers = new Array();

for( i in resources ) {
     if( resources[i].resourceType.id == resType.id ) {
          jbossServers[idx] = resources[i];
          idx = idx + 1;
     }
}

// restart the resources
for( a in resources ) {
     var jboss = ProxyFactory.getResource(jbossServers[a].id);
     jboss.restart()
}
```

Example 18. Starting an Array 2

```
//find the resources
//use a plugin filter to make sure they are all of the same type
criteria = new ResourceCriteria();
criteria.addFilterPluginName('JBossAS5')
criteria.addFilterResourceTypeName('JBossAS Server');

var jbossServers =
ResourceManager.findResourcesByCriteria(criteria).toArray();

// restart the resources
for( a in jbossServers ) {
     var jboss = ProxyFactory.getResource(jbossServers[a].id);
     jboss.restart()
}
```

6.2. Scheduling Operations

The simplest way to run an operation is to create a proxy for the resource, and then run the operation on that proxy, in the form *proxyName.operationName()*.

Example 19. Immediate Operation

```
rhqadmin@localhost:7080$ var agent = ProxyFactory.getResource(10008)
rhqadmin@localhost:7080$ agent.executeAvailabilityScan(true)
Invoking operation executeAvailabilityScan
Configuration [13903] - null
  isChangesOnly = true
  agentName = server.example.com
  resourceAvailabilities [0] {
  }
```

Operations can be run on a schedule. A schedule requires several configuration pieces:

- The resource ID
- The operation name
- A delay period, meaning when in the future to start the operation (optional)
- A repeat interval and count (optional)
- A timeout period (optional)
- Configuration parameters, if required by the operation
- A description of the scheduled operation (optional)

This example runs an availability scan on a specific agent.

Example 20. Scheduled Operation Example

```
// find the agent
var rc = ResourceCriteria();
rc.addFilterResourceTypeName("RHQ Agent");
rc.addFilterVersion("3.2");

var agent = ResourceManager.findResourcesByCriteria(rc);

//set the config properties for the operation
var config = new Configuration();
config.put(new PropertySimple("changesOnly", "true") );

//schedule the operation
OperationManager.scheduleResourceOperation(
          agent.get(0).id,
          "executeAvailabilityScan",
   0, // 0 means that the delay was skipped
          1,
   0, // this skips the repeat count
```

```
            10000000,
    config,
            "test from cli"
    );
```

This immediately prints the information for the scheduled operation.

```
ResourceOperationSchedule:
        resource: Resource[id=10008, uuid=e11390ec-34c4-49df-a4b6-
c37c516f545c, type={RHQAgent}RHQ Agent, key=server.example.com RHQ Agent,
name=RHQ Agent, parent=server.example.com, version=3.2]
```

6.3. Retrieving the Results of an Operation

Some operations really only report a success or failure, and that is all the information that is required. Other operations, however, may retrieve information from a resource or make a more complex change on the resource. In that case, that information needs to be returned so it can be used in further tasks.

The **`fetchResults(true)`** method can be used to return the results of the operation as part of the search for the operation history.

```
// find the resource
criteria = new ResourceCriteria();
criteria.addFilterResourceTypeName('Linux');
criteria.addFilterName('server.example.com');
ResourceManager.findResourcesByCriteria(criteria);

var resource = ResourceManager.findResourcesByCriteria(criteria);

// search for the operation history
var opcrit = ResourceOperationHistoryCriteria();
// get the operation for the resource ID
opcrit.addFilterResourceIds(resource.get(0).id);
// filter by the operation name
opcrit.addFilterOperationName("viewProcessList")
opcrit.fetchResults(true);

// get the data and print the results
var r = OperationManager.findResourceOperationHistoriesByCriteria(opcrit)
var h = r.get(0);
var c = h.getResults();
pretty.print(c)
```

Example 21. Printing the Results of a Process Scan

In this longer example, the script first runs an operation (a process scan on a platform) and then prints the results.

The first part of the script sets up the requirement for the resource ID and then searchs for that resource.

```
if (args.length != 1) {
    throw "we need a resource id as an argument";
}
```

```
var platform = ResourceManager.getResource(args[0]);
```

The next part schedules the process scan operation. As covered in Section 6.2, “Scheduling Operations”, this sets the resource ID, operation name, and operation settings.

```
var ros = OperationManager.scheduleResourceOperation(
   platform.id,
   "viewProcessList",
   0,
   1,
   0,
   15,
   null,
   "test operation"
);
```

The last part retrieves the operation history with two additional search settings:

- **`fetchResults(true)`**, which is required to include the operation result data and not just the status
- a sort method, in this case **`addSortStartTime`**

There are also a couple of sleeps in the script to ensure that the operation has time to complete before the script attempts to retrieve the results.

```
var opcrit = ResourceOperationHistoryCriteria();
opcrit.addFilterResourceIds(platform.id);
opcrit.fetchResults(true); // request the additional optional data in the
result
opcrit.addSortStartTime(PageOrdering.DESC); // sort by start time
java.lang.Thread.sleep(1000); // wait a second to make sure operation is
in the history

// wait for up to 15 seconds for last operation to complete, then print
result
now=new Date().getTime();
while (new Date().getTime() - now < 15000 ) {
   operations =
OperationManager.findResourceOperationHistoriesByCriteria(opcrit);
   if (operations.get(0).getResults() == null) {
      println("operation still pending result");
      java.lang.Thread.sleep(1000);
   } else {
      pretty.print(operations.get(0).getResults());
      break;
   }
}
if (operations.get(0).getErrorMessage() != null) {
      println("Error getting process list: ");
      pretty.print(operations.get(0).getErrorMessage());
}
```

This script prints the results of the operation as long as the operation has completed successfully:

```
   } else {
```

```
        pretty.print(operations.get(0).getResults());
        break;
    }
```

6.4. Checking a Resource's Operations History

The operation history for a resource exists as an object, so it can be searched for, by criteria, same as other objects.

Example 22. Viewing the Operation History

```
// find the resource
var rc = ResourceCriteria();
rc.addFilterPluginName("RHQAgent");
rc.addFilterName("RHQ Agent");
rc.addFilterResourceTypeName("RHQ Agent");
rc.addFilterDescription("Agent");

var agent = ResourceManager.findResourcesByCriteria(rc);

// print the operation history for the resource
var opcrit = ResourceOperationHistoryCriteria()
opcrit.addFilterResourceIds(agent.get(0).id)
OperationManager.findResourceOperationHistoriesByCriteria(opcrit);
```

The (successful) operation results are in **`Configuration`** objects in the results table.

```
resource                                                    results
------------------------------------------------------------------------------
---------------------------------------
Resource[id=10008, uuid=e11390ec-34c4-49df-a4b6-c37c516f545c, type=
{RHQAgent}RHQ Agent, key=server.example.com RHQ Age Configuration[id=13903]
Resource[id=10008, uuid=e11390ec-34c4-49df-a4b6-c37c516f545c, type=
{RHQAgent}RHQ Agent, key=server.example.com RHQ Age Configuration[id=13913]
2 rows
```

7. Monitoring

7.1. Getting Resource Availability

Many operations should only occur if the target resource is running, such as deploying a new web application. Including an availability check in a larger script is helpful for controlling when state-dependent operations are run.

The script steps are:

1. Search for the resource. In this case, the script just looks for any resource which matches the resource type (Linux platform), and uses the first match for the availability scan.
2. Get the current availability status.

Example 23. Current Availability

```
// get the resource ID
criteria = new ResourceCriteria();
criteria.addFilterResourceTypeName('Linux')

var res = ResourceManager.findResourcesByCriteria(criteria);

// check the current availability
AvailabilityManager.getCurrentAvailabilityForResource(res.get(0).id)
```

The JBoss ON server returns the resource information, its current status, and the time what the current status began (meaning, if the server is up, the time the server started).

```
Availability:
        availabilityType: UP
                 endTime:
                      id: 10192
                resource: Resource[id=10001, uuid=null, type=<null>,
key=null, name=null, parent=<null>]
        startTime: 1335974397214
```

7.2. Getting Specific Metrics

The monitoring information in JBoss ON is not a live reading. There are two reasons for that: scans are periodic, not streaming, and the information for baselines and averages are processed (aggregated).

The **`findLiveData`** method is a way to pull in the current, un-average, live reading of a given metric.

The script steps are:

1. Search for the available metric definitions, based on the resource type and then filtered to a single metric. This example grabs the free memory metric for the Linux platform.
2. Search for the resource.
3. Get the current reading for the metric.
4. Print the data to the terminal.

Example 24. Annotated Example

```
// search for the resource
criteria = new ResourceCriteria();
criteria.addFilterResourceTypeName('Linux');
var resources = ResourceManager.findResourcesByCriteria(criteria);

// search for the resource type to use in the metrics definition
var rt = ResourceTypeManager.getResourceTypeByNameAndPlugin("Linux",
"Platforms");

// search for the metric definition
var mdc = MeasurementDefinitionCriteria();
mdc.addFilterDisplayName("Free Memory");
```

```
mdc.addFilterResourceTypeId(rt.id);
var mdefs =
MeasurementDefinitionManager.findMeasurementDefinitionsByCriteria(mdc);

//get the data
var metrics = MeasurementDataManager.findLiveData(resources.get(0).id,
[mdefs.get(0).id]);

// as a nice little display, print the retrieved metrics value
if( metrics !=null ) {
        println(" Metric value for " + resources.get(0).id + " is " +
metrics );
}
```

With this example, the current, live reading for the metric is printed to the screen.

```
 Metric value for 10001 is [MeasurementDataNumeric[value=[6.3932239872E10],
MeasurementData [MeasurementDataPK: timestamp=[Tue May 08 20:10:15 EDT
2012], scheduleId=[10002]]]]
```

7.3. Exporting Metric Data for a Resource

Raw metrics data are only saved in the database for a week by default. After that, only the processed (aggregated) data are saved. It can be useful to export raw measurements to a CSV or text file so that long-term historical data can be preserved.

The script steps are:

1. Search for the available metric definitions, based on the resource type. In this example, it is for the Linux platform.
2. Search for the resource.
3. Set a date range for the metric information. This is configured in seconds, relative to the time the script is run.
4. Set up the file information to which to write the data.
5. Iterate through all the metric definitions for the resource, and print the data to the given CSV file.

Example 25. Exporting All Metrics Definitions for a Linux Server

```
// search for the available metrics definitions
var rt =
ResourceTypeManager.getResourceTypeByNameAndPlugin("Linux","Platforms")
var mdc = MeasurementDefinitionCriteria();
mdc.addFilterResourceTypeId(rt.id);
var mdefs =
MeasurementDefinitionManager.findMeasurementDefinitionsByCriteria(mdc);

// search for the resource
criteria = new ResourceCriteria();
criteria.addFilterResourceTypeName('Linux')
var resources = ResourceManager.findResourcesByCriteria(criteria);
```

```
// give the date range for the metrics collection
// this is in seconds
var start = new Date() - 8* 3600 * 1000;
var end = new Date()

// setup up the CSV to dump the data to
exporter.file = '/opt/myfile.csv'
exporter.format = 'csv'


// iterate through the metrics definitions for the resource
// and export all the collected metrics for all definitions
// within the given date range
if( mdefs != null ) {
  if( mdefs.size() > 1 ) {
     for( i =0; i < mdefs.size(); ++i) {
          mdef = mdefs.get(i);
          var data =
MeasurementDataManager.findDataForResource(resources.get(0).id,
[mdef.id],start,end,1)

   exporter.write(data.get(0)); // write the data to the CSV file
     }
  }
  else if( mdefs.size() == 1 ) {
     mdef = mdefs.get(0);
     var data =
MeasurementDataManager.findDataForResource(resources.get(0).id,
[mdef.get(0).id],start,end,60)
     exporter.write(data.get(0))
  }
}
```

7.4. Getting Baseline Calculations

Baselines are the normal operating ranges for a specific resource, based on its own performance.

Getting a baseline is really easy; all it requires is the resource ID.

```
rhqadmin@localhost:7080$
MeasurementBaselineManager.findBaselinesForResource(10001)
one row
MeasurementBaseline:
        computeTime: Tue May 08 21:28:05 EDT 2012
                 id: 10001
                max: 6.4005419008E10
               mean: 6.3933904981333336E10
                min: 6.380064768E10
           schedule: [MeasurementSchedule, id=10002]
        userEntered: true
```

8. Alerts

Alert definitions cannot be created or edited using the CLI, but there are some management tasks that can still be performed. For fired alerts themselves, server-side scripts can be used as alert responses (and the alert referenced as an implicit variable) and the fired alert can be viewed and acknowledged by an administrator. Definitions can be enabled or disabled.

8.1. Using Alerts with Scripts

One possible response to an alert is for the server to automatically run a stored server script (essentially a stored CLI script file).

That server-side alert script can reference the alert object for the triggered alert. The server defines an implicit *alert* variable, which pulls in the alert information, ID, definition, and other information for the fired alert.

Because the **`alert`** method identifies the alert definition, it identifies the resource which triggered the alert. This allows you to create a reusable proxy resource definition in the script that could be applied to any resource which uses that alert script.

For example:

```
var myResource = ProxyFactory.getResource(alert.alertDefinition.resource.id)
```

Note

This method is only available to server-side script used with alerts, not to the interactive CLI or external CLI script files.

8.2. Acknowledging Alerts

One common administrative action when an alert is fired is for an administrator to review and then acknowledge the alert, which effectively closes it. This script example acknowledges all of the current alerts for all Linux platform resources.

The criteria for the search can be set to be more restrictive so that only certain alerts or only certain resources are included in the action.

The steps in this script:

1. Search for fired alerts; in this case, the search is based on the resource type (Linux).
2. Retrieve the data for the search results.
3. Acknowledge all returned alerts.

Example 26. Acknowledging Alerts for Platform Resources

```
// set the criteria and search for the alerts
var criteria = new AlertCriteria()
criteria.addFilterResourceTypeName('Linux')
var alerts = AlertManager.findAlertsByCriteria(criteria)

// go through the results and then acknowledge the alerts
if( alerts != null ) {
  if( alerts.size() > 1 ) {
```

```
      for( i =0; i < alerts.size(); ++i) {
           alert = alerts.get(i);
           AlertManager.acknowledgeAlerts([alert.id])
      }
   }
   else if( alerts.size() == 1 ) {
      alert = alerts.get(0);
      AlertManager.acknowledgeAlerts([alert.id])
   }
}
```

8.3. Enabling or Disabling Alert Definitions

Alert definitions cannot be created or edited through the CLI, but they can be enabled or disabled.

This example script disables all of the definitions returned in the search. In real life, the criteria for the search are crucial to make sure that only the right definitions are disabled, rather than disabling large blocs of definitions.

The steps in this script:

1. Search for matching alert definitions based on priority (low, in this case).
2. Retrieve the data for the search results.
3. Disable all returned alert definitions, based on the IDs in the retrieved search list.

Example 27. Disabling Alerts Based on Priority

```
// set the search criteria for the alert definitions with a reasonable
filter
var criteria = new AlertDefinitionCriteria()
criteria.addFilterPriority(AlertPriority.LOW)

//search for the alert definitions
alertdefs =
AlertDefinitionManager.findAlertDefinitionsByCriteria(criteria)

//get the data from the results
alertdef = alertdefs.get(0);

println("  alert: " + alertdef.id );

//disable the matching alerts, based on ID
AlertDefinitionManager.disableAlertDefinitions([alertdef.id]);
```

9. Users and Roles

Roles are the primary access control method in JBoss ON. A role creates a relationship between users and resources (through resource groups). The permissions set for the role define what permissions the users in the role have for the resource in the role.

9.1. Creating Roles

A user can only see and manage what resources are in the roles to which a user belongs. If there are no resources, then the user can do very little in JBoss ON, regardless of whatever their permissions are.

This script creates a role and adds a mixed group to it. It could be made more complex, like using different criteria to get different types of groups or adding multiple groups to the role at once.

The script steps are:

1. Create a role and assigning the appropriate permissions. In this case, the role has manage inventory and view user permissions.
2. Search for the group to add as a member.
3. Search for the new role entry.
4. Add the group to the role.

Example 28. A New Role

```
// create the role
var role = Role('Role Name - ' + java.util.Date());
role.description = 'This role is an example';
role.addPermission(Permission.MANAGE_INVENTORY);
role.addPermission(Permission.VIEW_USERS);
RoleManager.createRole(role);

//search for the group to add to the role
groupcriteria = new ResourceGroupCriteria();
groupcriteria.addFilterGroupCategory.toString('MIXED');

var groups =
ResourceGroupManager.findResourceGroupsByCriteria(groupcriteria);

//search for the new role
var c = new RoleCriteria();
c.addFilterName('Role Name');
var roles = RoleManager.findRolesByCriteria( c );
RoleManager.addResourceGroupsToRole(roles.get(0).id,[groups.get(0).id]);
```

9.2. Creating Users

There are two parts to a user entry: the descriptive entry in JBoss ON and the principal, which is the login username/password pair.

The script steps are:

1. Create a new user (subject) entry.
2. Create a principal for the new user.
3. Search for roles to add the user to and create an array.
4. Add the user to the roles.

Example 29. Creatting a User and Adding Roles

```
//create the new user entry
var newSubject = new Subject();
newSubject.setEmailAddress( 'admin@example.com' );
newSubject.setFirstName('John');
newSubject.setLastName('Smith' );
newSubject.setFactive(true);
newSubject.setFsystem(false);
newSubject.setName('jsmith');
var s = SubjectManager.createSubject(newSubject);

//create the login principal for the user
SubjectManager.createPrincipal( s.name, 'password' );

//search for the role and create an array
var c = new RoleCriteria();
c.addFilterName('Role Name');
var roles = RoleManager.findRolesByCriteria( c );
var role = roles.get(0);
var rolesArray = new Array(1);
rolesArray[0] = role.getId();

//add the new user to the roles in the array
RoleManager.addRolesToSubject(s.getId(), rolesArray );
```

Part III. Extended Examples and Use Scenarios

Short Examples runs through discrete code examples that perform limited and specific tasks. In real life, those code examples (and others) are going to be strung together to perform more coherent and useful management tasks. These examples illustrate some potential script workflows.

1. Example: Scripts to Manage Inventory (All Resource Types)

Servers and services are routinely added or removed from a local machine. While discovery scans are scheduled regularly, actually adding or removing that resource within the JBoss ON inventory is all manual — and administrator must actually choose to import the resource.

An administrator can manage the JBoss ON inventory by automatically adding new resources and automatically deleting other ones based on whatever criteria are defined.

1.1. Automatically Import New Resources: autoimport.js

As soon as a resource is discovered it is, technically, already in the JBoss ON inventory. It is included with a status of **`NEW`**. That's an in-between state, because JBoss ON is aware that the resource exists, but the resource has not been committed so JBoss ON can't manage it.

A script can be created and run regularly so that any newly-discovered resources can be automatically added to the inventory. This script bases its identification on new resources on the inventory state, so ignored or already imported resources aren't included.

The CLI script runs through three steps:

- It identifies new resources using the **`findUncommittedResources()`** method.
- It gets those new resources' IDs.
- It then imports those resources by invoking the discovery system's import operation.

```
//Usage: autoImport.js
//Description: Imports all auto-discovered inventory into JON
// autoImport.js
rhq.login('rhqadmin', 'rhqadmin');
println("Running autoImport.js");

var resources = findUncommittedResources();
var resourceIds = getIds(resources);
DiscoveryBoss.importResources(resourceIds);

rhq.logout();
```

Only one of the operations is already defined in the remote API — **`importResources`**. The other two functions — **`findUncommittedResources`** and **`getIds`** — have to be defined in the script.

Uncommitted (new) resources can be identified through a **`ResourceCriteria`** search by adding a search parameter based on the inventory status.

```
// returns a java.util.List of Resource objects
// that have not yet been committed into inventory
function findUncommittedResources() {
    var criteria = ResourceCriteria();
```

```
    criteria.addFilterInventoryStatus(InventoryStatus.NEW);

    return ResourceManager.findResourcesByCriteria(criteria);
}
```

The second function checks that the inventory search actually returned resources and, if so, gets the ID for each resource in the array.

```
// returns an array of ids for a given list
// of Resource objects. Note the resources argument
// can actually be any Collection that contains
// elements having an id property.
function getIds(resources) {
 var ids = [];

 if (resources.size() > 0) {
  println("Found resources to import: ");
  for (i = 0; i < resources.size(); i++) {
   resource = resources.get(i);
   ids[i] =  resource.id;
   println("  " + resource.name);
  }
 } else {
  println("No resources found awaiting import...");
 }

    return ids;
}
```

1.2. Simple Inventory Count: inventoryCount.js

Searches are performed using **`*Criteria`** classes; for resources, this is **`ResourceCriteria`**. A search can be very specific, passing criteria so that it returns only one resource or a small subset of resource. It is also possible to return everything in inventory.

This script runs a search with no specific criteria (**`ResourceCriteria()`**), so that every resource matches the search. It then takes the size of the results to produce a simple inventory count.

```
// inventory.js
rhq.login('rhqadmin', 'rhqadmin');
var resources = ResourceManager.findResourcesByCriteria(ResourceCriteria());
println('There are ' + resources.size() + ' resources in inventory');

// end script
```

1.3. Uninventory a Resource After an Alert: uninventory.js

Removing a resource from the inventory simply removes it from JBoss ON; the server or application itself remains intact on the local system. (This allows the resource to be re-discovered and re-imported later.)

The alert system can launch CLI scripts in response to a fired alert (covered in "Setting up Monitoring, Alerts, and Operations"). One possible response is to uninventory a resource which is not performing well.

This can be a pretty simple little script. To uninventory the resource, simply use the resource ID which was included in the alert and the **uninventoryResource** method:

```
List<Integer> uninventoryResources(Subject subject, int[] resourceIds);
```

It is possible to combine the uninventory operation with another task. For example, uninventory one resource and automatically create and import another resource to take its place.

Note

This script should be uploaded to the server and then referenced as an alert notification.

2. Example: Scripts to Manage Resources of a Specific Type

Section 1, “Example: Scripts to Manage Inventory (All Resource Types)” shows a general resource discovery and import script. In some environments, it may be more useful to discover and import only specific types of resources. For example, and administrator may be constantly deploying new applications to an app server, or maybe spinning up and destroying application servers dynamically to respond to load. In that case, the administrator only wants to find the specific resources which he knows are frequently created.

For this example, the workflow has some pretty basic steps:

1. Search for new, uncommitted resources of a specific type.
2. Get the resource IDs.
3. Import those new resources.
4. Do something with the newly-imported resources.

Steps 1 through 3 are basically the same as the **autoimport.js** example in Section 1.1, “Automatically Import New Resources: autoimport.js”, with one significant change. The search has an extra filter for the resource type.

Uncommitted (new) resources can be identified through a **ResourceCriteria** search by adding a search parameter based on the inventory status and the resource type (in this example, JBoss EAP 6 domain deployments, for new web applications). The new **findUncommittedJbasApps()** function runs that search.

```
function findUncommittedJbasApps() {
    var criteria = ResourceCriteria();
    criteria.addFilterInventoryStatus(InventoryStatus.NEW);
    criteria.addFilterResourceTypeName('DomainDeployment');

    return ResourceManager.findResourcesByCriteria(criteria);
}
```

As with the **autoimport.js** example, a **getIds** function retrieves an array of the resource IDs.

```
function getIds(resources) {
 var ids = [];

 if (resources.size() > 0) {
```

```
  println("Found resources to import: ");
  for (i = 0; i < resources.size(); i++) {
   resource = resources.get(i);
   ids[i] =  resource.id;
   println("  " + resource.name);
  }
 } else {
  println("No resources found awaiting import...");
 }

    return ids;
}
```

Those two functions accomplish the first two steps: search for uncommitted resources of a specific type and then retrieve those IDs.

Step 3 then runs the **`DiscoveryBoss`** method to import the discovered resources.

The first half of the script, then mimics the original **`autoimport.js`**, with the slight adjustment to the new function to search for JBoss EAP 6 domain deployments.

```
rhq.login('rhqadmin', 'rhqadmin');
println("Running autoImport.js");

var resources = findUncommittedJbasApps();
var resourceIds = getIds(resources);
DiscoveryBoss.importResources(resourceIds);
```

The last step simply does something with the new resource.

For a new domain deployment, then it is probably most useful to assign the new domain deployment to server group, so the new application is deployed. For example, this assigns all new applications automatically to a server group used for the staging environment, using the promote operation:

```
var config = new Configuration();
config.put(new PropertySimple("server-group", "Staging-Server-Group") );

OperationManager.scheduleResourceOperation(
          resources.get(0).id,
          "promote",
   0, // 0 means that the delay was skipped
          1,
   0, // this skips the repeat count
          10000000,
   config,
          "promote new app to server group"
   );
```

Alternatively, if this script is used to add a new service or a new server, then it may be more appropriate to add the new resource to a group. This example uses an explicit group ID (15001) for an existing compatible group. The assumption is that the group has already been created, before this script is run, and there is no need to create a new group for the resources.

```
ResourceGroupManager.addResourcesToGroup(15001, [resourceIds]);
```

Then, have the script log out from the CLI and close.

```
rhq.logout();
```

3. Example: Scripting Resource Deployments (JBoss EAP 5)

A common use case for management tools is to automate deployments of new or existing applications. This example creates an easy script for basic management tasks:

1. Find all JBoss EAP instances for a specified JBoss ON group.
2. Shut down each EAP instance.
3. Update binaries for existing deployed applications or create new deployments.
4. Restart the EAP instance.
5. End the loop.

3.1. Declaring Custom Functions

This script will use two custom functions to deploy the packages to create new resources.

```
function usage() {
 println("Usage: deployToGroup <fileName> <groupName>");
 throw "Illegal arguments";
}

function PackageParser(fullPathName) {
 var file = new java.io.File(fullPathName);

 var fileName = file.getName();
 var packageType = fileName.substring(fileName.lastIndexOf('.')+1);
 var tmp = fileName.substring(0, fileName.lastIndexOf('.'));
 var realName = tmp.substring(0, tmp.lastIndexOf('-'));
 var version = tmp.substring(tmp.lastIndexOf('-') + 1);
 var packageName = realName + "." + packageType;

 this.packageType = packageType.toLowerCase();
 this.packageName = packageName;
 this.version     = version;
 this.realName    = realName;
}
```

3.2. Checking the JBoss ON Groups and Inventory

The script should have two command-line parameters. The first should be the path of the new application that is installed in the group. The second is the name of the group itself. These parameters are parsed in the script (as described in more detail in Section 6.3, “Passing Command and Script Arguments”).

For example:

```
if( args.length < 2 ) usage();

var fileName = args[0];
var groupName = args[1];
```

Next, check if the path is valid and if the current user can read it. This is done by using Java classes as shown here:

```
// check that the file exists and that we can read it
var file = new java.io.File(fileName);

if( !file.exists() ) {
    println(fileName + " does not exist!");
    usage();
}

if( !file.canRead() ) {
    println(fileName + " can't be read!");
    usage();
}
```

Verify that the group really exists on the JBoss ON server:

```
// find resource group
var rgc = new ResourceGroupCriteria();
rgc.addFilterName(groupName);
rgc.fetchExplicitResources(true);
var groupList = ResourceGroupManager.findResourceGroupsByCriteria(rgc);
```

The important part here is to fetch the resources.

```
rgc.fetchExplicitResources(true);
```

Check if there is a group found:

```
if( groupList == null || groupList.size() != 1 ) {
    println("Can't find a resource group named " + groupName);
    usage();
}

var group = groupList.get(0);

println("  Found group: " + group.name );
println("  Group ID   : " + group.id );
println("  Description: " + group.description);
```

After validating that there is a group with the specified name, check if the group contains explicit resources:

```
if( group.explicitResources == null || group.explicitResources.size() == 0 )
{
    println("  Group does not contain explicit resources --> exiting!" );
    usage();
}
var resourcesArray = group.explicitResources.toArray();
```

resourceArray now contains all resources which are part of the group. Next, check if there are JBoss AS 5 Server instances which need to be restarted before the application is deployed.

```
for( i in resourcesArray ) {
    var res = resourcesArray[i];
    var resType = res.resourceType.name;
    println("  Found resource " + res.name + " of type " + resType + " and
ID " + res.id);

    if( resType != "JBossAS5 Server") {
        println("    ---> Resource not of required type. Exiting!");
        usage();
    }

    // get server resource to start/stop it and to redeploy application
    var server = ProxyFactory.getResource(res.id);
}
```

This requires a group with only JBoss AS 5 Server resource types as top level resources. Now **server** contains the JBoss AS 5 instance. This requires re-reading the server because it needs to be fully populated. Internally, the CLI is using simple JPA persistence, and it is necessary to not always fetch all dependent objects.

Next, traverse all the children of the server instance and find the resource name of the application:

```
var children = server.children;
for( c in children ) {
    var child = children[c];

    if( child.name == packageName ) {
    }
}
```

packageName is the name of the application without version information and path as shown in the JBoss ON GUI as deployed applications.

Create a backup of the original version of the application:

```
println("    download old app to /tmp");
child.retrieveBackingContent("/tmp/" + packageName + "_" + server.name +
"_old");
```

A copy of the old application with the server name decoded in path is available in the **/tmp/** directory.

Shut down the server and upload the new application content to the server.

```
println("    stopping " + server.name + "....");
try {
    server.shutDown();
}
catch( ex ) {
    println("   --> Caught " + ex );
}
```

```
println("    uploading new application code");
child.updateBackingContent(fileName);

println("    restarting " + server.name + "....." );

try {
    server.start();
}
catch( ex ) {
    println("   --> Caught " + ex );
}
```

3.3. Deploying the New Resource

At this point, existing application can be updated. The next step is to create the resource through the CLI and then deploy it to the JBoss server.

First, get the resource type for the application. This depends on several parameters:

1. The type of the application (e.g., WAR or EAR)
2. The type of the container the app needs to be deployed on (such as Tomcat or JBoss AS 5)

Note

All of the information about the resource type, such as the appType and appTypeName, is defined in the resource agent plug-in, in the **`rhq-plugin.xml`** descriptor. The attributes, configuration parameters, operations, and metrics for each default resource type are listed in the *Resource Monitoring and Operations Reference*.

For example:

```
var appType = ResourceTypeManager.getResourceTypeByNameAndPlugin(
appTypeName, "JBossAS5" );
if( appType == null ) {
    println("  Could not find application type. Exit.");
    usage();
}
```

Then get the package type of the application.

```
var realPackageType = ContentManager.findPackageTypes( appTypeName,
"JBossAS5" );

if( realPackageType == null ) {
    println("  Could not find JBoss ON's packageType. Exit.");
    usage();
}
```

Each resource in JBoss ON has some configuration parameters, including the WARs or EARs deployed on a JBoss AS 5 resource. As with the descriptive information, this is defined in the resource type's agent plug-in, in the **`rhq-plugin.xml`** descriptor. To be able to create a new resource, these parameters need to be filled in.

```
// create deployConfig
var deployConfig = new Configuration();
deployConfig.put( new PropertySimple("deployExploded", "false"));
deployConfig.put( new PropertySimple("deployFarmed", "false"));
```

The property names can be retrieved by calling a list of supported properties by the package type by calling this method:

```
var deployConfigDef =
ConfigurationManager.getPackageTypeConfigurationDefinition(realPackageType.g
etId());
```

Provide the package bits as a byte array:

```
var inputStream = new java.io.FileInputStream(file);
var fileLength = file.length();
var fileBytes = java.lang.reflect.Array.newInstance(java.lang.Byte.TYPE,
fileLength);
for (numRead=0, offset=0; ((numRead >= 0) && (offset < fileBytes.length));
offset += numRead ) {
    numRead = inputStream.read(fileBytes, offset, fileBytes.length -
offset);
}
```

Then, create the resource. The information is defined in the resource type's agent plug-in, in the **`rhq-plugin.xml`** descriptor. For example:

```
ResourceFactoryManager.createPackageBackedResource(
    server.id,
    appType.id,
    packageName,
    null,  // pluginConfiguration
    packageName,
    packageVersion,
    null, // architectureId
    deployConfig,
    fileBytes,
    null // timeout
);
```

Make sure that the given JBoss AS 5 server instance is still running and that JBoss ON knows that it is running, or it will throw an exception saying that the JBoss ON agent is not able to upload the binary content to the server.

4. Example: Deploying an Application with Bundles (JBoss EAP 4, 5, and 6)

Bundles are a very clean and easy way to deploy updated applications to JBoss EAP 6 servers. The bundles system maintains multiple versions of a given package and can deploy any of those versions to a specified compatible group. This is a great workflow for application lifecycles, since a stable version can be deployed to production servers while a development version can be deployed to test machines.

4.1. Creating a New Application

The **jbossas.js** script in the ***cliRoot*/rhq-remoting-cli-4.9.0.JON320GA/samples/modules** directory defines a set of custom functions that quickly create a bundle version and definition and then deploy it to the given JBoss servers.

The **jbossas.js** script requires the **util.js** and **bundles.js** scripts for supporting functions. These are all loaded automatically from the ***cliRoot*/rhq-remoting-cli-4.9.0.JON320GA/samples/modules** directory.

The **deploy-to-and-restart-JBAS.js** script defines a function called **createAppAndRestartJBAS** that creates and then deploys a bundle. The function takes seven parameters:

- The path to the bundle archive file.
- A configuration object that contains any tokens or variables required for the bundle configuration. In this example, no properties are passed, so the value is null. Details about the configuration are in the comments in the **bundles.js** file and general configuration information is in Section 5.2, “Changing Simple Configuration Properties”.
- The name of the compatible group to which to deploy the bundle.
- A name of the bundle destination.
- A description for the bundle destination.
- The base directory to which to deploy the bundle and, optionally, a subdirectory to deploy the bundle to.

For example:

```
rhqadmin@localhost:7080$ var bundleZipFile = /export/bundles/myBundle.zip
rhqadmin@localhost:7080$ var deploymentConfiguration = null
rhqadmin@localhost:7080$ var groupName = "JBoss EAP 6 Group"
rhqadmin@localhost:7080$ var destinationName = "My App - JBoss EAP 6
Destination"
rhqadmin@localhost:7080$ var destinationDescription = "For my application"
rhqadmin@localhost:7080$ var baseDirName = "Deploy Directory"
rhqadmin@localhost:7080$ var deployDir = "helloWorldApp"
rhqadmin@localhost:7080$ createAppAndRestartJBAS(bundleZipFile,
deploymentConfiguration, groupName, destinationName, destinationDescription,
baseDirName, deployDir)
```

There are a number of private functions defined in the **deploy-to-and-restart-JBAS.js** script which provide support to the **createAppAndRestartJBAS** function. Those are explained in the comments in the **deploy-to-and-restart-JBAS.js** file.

Note

The *baseDirName* variable must have a value of **`Deploy Directory`** or **`Profile Directory`**. Those two locations are then identified based on the connection information for the specified JBoss EAP resource which is accessed by the script.

The **`createAppAndRestartJBAS`** function has three parts: finding the resource group, creating the bundle, and restarting the JBoss servers in the group.

The first part searches for the specific group by the given group name, and it must match only a single resource group. Otherwise, it returns an error.

```
function createAppAndRestartJBAS(bundleZipFile, deploymentConfiguration,
groupName, destinationName, destinationDescription, baseDirName, deployDir)
{
    var gcrit = new ResourceGroupCriteria;
    gcrit.addFilterName(groupName);
    gcrit.fetchResourceType(true);

    var groups = ResourceGroupManager.findResourceGroupsByCriteria(gcrit);
    if (groups.empty) {
        throw "Could not find a resource group called " + groupName;
    } else if (groups.size() > 1) {
        throw "There are more than 1 groups called " + groupName;
    }

    var group = groups.get(0);
    var targetResourceType = group.resourceType;
```

Part of the search instruction is to fetch the resource type for the given compatible group. That resource type is then used to create the bundle destination, which is associated with a group.

The next part creates the bundle and restarts the server resources.

```
    var deployFn = function(restartFn) {
```

The first part of create a bundle deployment is uploading the bundle archive to create a bundle version in the JBoss ON configuration.

```
        var bundleVersion = createBundleVersion(bundleZipFile);
```

Next, it creates the destination, which is the definition of where the bundle is to be deployed (the base and deployment directories), associated with a compatible group and with the bundle version.

```
        var destination =
BundleManager.createBundleDestination(bundleVersion.bundle.id,
destinationName, destinationDescription, baseDirName, deployDir, group.id);
```

Once the destination and the version are defined, then they can be saved and deployed to the target resource group.

```
        var deployment = deployBundle(destination,bundleVersion,
deploymentConfiguration, "Web application", false);
```

```
        if (deployment.status != BundleDeploymentStatus.SUCCESS) {
            throw "Deployment wasn't successful: " + deployment;
        }
```

When the deployment completes, then the script iterates through the group members (defined in one of the help functions in the **deploy-to-and-restart-JBAS.js** script) and restarts each resource. It then prints the deployment information.

```
        restartFn(group);

        return deployment;
    };
```

The **deploy-to-and-restart-JBAS.js** script can deploy content to any supported version of JBoss EAP: 4, 5, or 6. The bundle system uses the configuration defined in the resource plug-in, based on the resource type of the group. For the restart operation, different restart function are defined for each version of JBoss EAP.

```
    if (targetResourceType.plugin == "JBossAS" && targetResourceType.name ==
"JBossAS Server") {
        return deployFn(_restartAS4);
    } else if (targetResourceType.plugin == "JBossAS5" &&
targetResourceType.name == "JBossAS Server") {
        return deployFn(_restartAS5);
    } else if (targetResourceType.plugin == "JBossAS7" &&
            (targetResourceType.name == "JBossAS7 Standalone Server" ||
                    targetResourceType.name == "JBossAS-Managed")) {
        return deployFn(_restartAS7);
    }

    throw "The resource group the destination targets doesn't seem to be a
JBoss AS server group.";
}
```

4.2. Updating Applications

Updating an application is simpler than creating one because the bundle definition already exists.

To update an application, give the path to the updated bundle archive, any tokens or properties to set, and the existing bundle destination.

```
rhqadmin@localhost:7080$ var bundleZipFile = /export/bundles/myBundle.zip
rhqadmin@localhost:7080$ var deploymentConfiguration = null
rhqadmin@localhost:7080$ var jbasDestination = "My App - JBoss EAP 6
Destination"
rhqadmin@localhost:7080$ updateAppAndRestartJBAS(bundleZipFile,
jbasDestination, deploymentConfiguration)
```

If the bundle requires any tokens to be realized, like a port number to be entered, then you must create a configuration object and pass the values to that. In this example, no properties are passed, so the value is null. Details about the configuration are in the comments in the **bundles.js** file and general configuration information is in Section 5.2, “Changing Simple Configuration Properties”.

The destination identifier — which could be the destination name or the ID — is used in a criteria search to fetch the resource type for the compatible group. This, as with the create function, identifies which version of JBoss EAP is being used.

```
function updateAppAndRestartJBAS(bundleZipFile, jbasDestination,
deploymentConfiguration) {
    // first figure out the jbas version we are deploying to
    var destinationId = jbasDestination;
    if (typeof(jbasDestination) == 'object') {
        destinationId = jbasDestination.id;
    }

    var destCrit = new BundleDestinationCriteria
    destCrit.fetchGroup(true)
```

It uses the name to search for the destination ID. When it retrieves the destination entry (in the **`get(0)`** call), the destination configuration contains the resource type.

```
    var destinations =
BundleManager.findBundleDestinationsByCriteria(destCrit);

    if (destinations.empty) {
        throw "No destinations corresponding to " + jbasDestination + "
found on the server.";
    }

    var destination = destinations.get(0);

    var targetResourceType = destination.group.resourceType;

    if (targetResourceType == null) {
        throw "This function expects a compatible group of JBoss AS (4,5,6
or 7) resources but the provided destination is connected with " +
destination.group;
    }
```

Then, it uploads the new bundle archive as a new bundle version, in the **`createBundleVersion`** method.

```
    var deployFn = function(restartFn) {
 var bundleVersion = createBundleVersion(bundleZipFile);
```

Then, it deploys the new bundle version to the existing destination, along with any defined or required tokens in a configuration object (*deploymentConfiguration*). When the deployment completes, it restarts the JBoss resources in the group and prints the deployment information.

```
        var deployment = deployBundle(destination,bundleVersion,
deploymentConfiguration, "Web application", false);

        if (deployment.status != BundleDeploymentStatus.SUCCESS) {
            throw "Deployment wasn't successful: " + deployment;
        }
```

```
        restartFn(destination.group);

        return deployment;
    };
```

As with the creation function, there are version-specific restart methods for any supported version of JBoss EAP: 4, 5, or 6. The bundle system uses the configuration defined in the resource plug-in, based on the resource type of the group.

```
    if (targetResourceType.plugin == "JBossAS" && targetResourceType.name ==
"JBossAS Server") {
        return deployFn(_restartAS4);
    } else if (targetResourceType.plugin == "JBossAS5" &&
targetResourceType.name == "JBossAS Server") {
        return deployFn(_restartAS5);
    } else if (targetResourceType.plugin == "JBossAS7" &&
            (targetResourceType.name == "JBossAS7 Standalone Server" ||
                    targetResourceType.name == "JBossAS-Managed")) {
        return deployFn(_restartAS7);
    }

    throw "The resource group the destination targets doesn't seem to be a
JBoss AS server group.";
}
```

5. Example: Remote JNDI Lookups After an Alert (JBoss EAP 5)

Important

This script is intended to be run directly on the server, such as using the **`-f`** parameter or through a server-side alert script. This cannot be run using the interactive CLI.

For information on running server-side scripts in response to alerts, see "Using JBoss Operations Network for Monitoring, Deploying, and Managing Resources."

Important

For security reasons, it is *not* possible to run a local JNDI lookup from an alert CLI script. It is possible to perform a remote JNDI lookup.

The alert system can run a script in response to a fired alert. One possible response for a JBoss AS 5 server is to check the JNDI directory and look up the JMX information.

This script first connects to the remote JNDI directory over JNP, then uses the **`assertNotNull`** method to get the JMX object. The script then prints the JMX information.

```
//This test requires a remote JBoss AS 5 server running with JNDI directory
remotely accessible using JNP (without authz)
//This script assumes that there is a bound object called "jmx" in the
directory (which it should be)
```

```
var jbossHost = 'localhost';
var jbossJnpPort = 1299;

var env = new java.util.Hashtable();
env.put('java.naming.factory.initial',
'org.jboss.naming.NamingContextFactory');
env.put('java.naming.provider.url', "jnp://" + jbossHost + ":" +
jbossJnpPort);
var ctx = new javax.naming.InitialContext(env);
var jmx = ctx.lookup('jmx');
assertNotNull(jmx);
pretty.print(jmx);
```

6. Example: Managing Grouped Servers (JBoss EAP 5)

A lot of enterprise servers have a concept of *managed servers*. A managed server means that there is a central instance that deploys content or sends configuration to all registered application servers. Using managed servers helps administrators ensure that all active application servers have the same version of the deployed packages and configuration.

JBoss ON can imitate the behavior of managed or clustered servers for applications like Tomcat or JBoss EAP 5 by creating a management script that can be invoked to perform actions simultaneously on all members of a JBoss ON group. All of the EAP 5 instances are functionally managed servers, while JBoss ON itself acts as the domain controller.

Note

JBoss EAP 6 has a very different server topology than JBoss EAP 5, so domain controllers, managed servers, and domain configuration are defined and manageable by default.

6.1. The Plan for the Scripts

The JBoss ON CLI can run defined JavaScripts using the `-f` parameter. The idea here is to create a series of small management scripts that perform specific tasks on a group of JBoss EAP servers. This example has seven scripts for:

- Creating a group
- Adding EAP instances to the group
- Checking EAP status
- Starting the EAP instance
- Scheduling an operation
- Deploying new content to the group
- Checking metrics

A wrapper script and configuration file will be set up so that only one command needs to be run; the wrapper invokes the appropriate JBoss ON CLI script depending on the command passed to the wrapper.

6.2. Creating the Wrapper Script and .conf File

The wrapper script takes command-line arguments and calls the JBoss ON CLI with one of the scripts as argument. The command-line arguments themselves are defined in the JBoss ON JavaScript files.

This wrapper script makes a few assumptions:

- The wrapper script is run as a regular user, which means that any JavaScript files must be accessible to a regular user.
- The scripts are located in a **`scripts/`** directory that is in the same directory as the wrapper script.
- A separate configuration file defines connection information for the JBoss ON server.
- Each JavaScript file is invoked by a separate CLI command invocation, defined in the wrapper.
- Any options or information required by the JBoss ON CLI command is defined in the JavaScript file and can, potentially, be passed with the wrapper script as an option.

```
#!/bin/bash
#
# groupcontrol
# ------------
# This is a simple wrapper script for all the java script scripts in this
folder.
# Start this script with some parameters to automate group handling from
within the
# command line.
#
# With groupcontrol you can do the following:
#   create   : Create a new group
#   addMember: Add a new EAP instance to the specified group
#   status   : Print the status of all resources of a group
#   start    : start all EAP instances specified by group name
#   deploy   : Deploys an application to all AS instances specified by group
name
#   avail    : Runs an availability operation on all discovered agent
instances
#   metrics  : Gets the specified metric value for all AS instances
specified by group name
#
#

## Should not be run as root.
if [ "$EUID" = "0" ]; then
   echo " Please use a normal user account and not the root account"
   exit 1
fi

## Figure out script home
MY_HOME=$(cd `dirname $0` && pwd)
SCRIPT_HOME=$MY_HOME/scripts

## Source some defaults
. $MY_HOME/groupcontrol.conf

## Check to see if we have a valid CLI home
```

```
if [ ! -d ${JON_CLI_HOME} ]; then
     echo "JON_CLI_HOME not correctly set. Please do so in the file"
     echo $MY_HOME/groupcontrol.conf
     exit 1
fi

RHQ_OPTS="-s $JON_HOST -u $JON_USER -t $JON_PORT"
# If JBoss ON_PWD is given then use it as argument. Else let the user enter
the password
if [ "x$JON_PWD" == "x" ]; then
     RHQ_OPTS="$RHQ_OPTS -P"
else
     RHQ_OPTS="$RHQ_OPTS -p $JON_PWD"
fi

#echo "Calling groupcontrol with $RHQ_OPTS"

usage() {
     echo "  Usage $0:"
     echo "  Use this tool to control most group related tasks with a simple
script."
     echo "  ------------------------------------------------------------
-------------- "
}
```

Each command that the wrapper should define has a **do***Command* **( )** section which defines the JBoss ON CLI command to run and the JavaScript file to use.

```
doDeploy() {
     $JON_CLI_HOME/bin/rhq-cli.sh $RHQ_OPTS -f $SCRIPT_HOME/deploy.js $2 $3
}

doCreate() {
     $JON_CLI_HOME/bin/rhq-cli.sh $RHQ_OPTS -f $SCRIPT_HOME/group.js $2
}

doAddMember() {
     $JON_CLI_HOME/bin/rhq-cli.sh $RHQ_OPTS -f $SCRIPT_HOME/addMember.js $2
$3 $4
}

doStatus() {
     $JON_CLI_HOME/bin/rhq-cli.sh $RHQ_OPTS -f $SCRIPT_HOME/status.js $2
}

doRestart() {
     $JON_CLI_HOME/bin/rhq-cli.sh $RHQ_OPTS -f $SCRIPT_HOME/restart.js $2
}

doAvail() {
     $JON_CLI_HOME/bin/rhq-cli.sh $RHQ_OPTS -f $SCRIPT_HOME/avail.js
}

doMetrics() {
     $JON_CLI_HOME/bin/rhq-cli.sh $RHQ_OPTS -f $SCRIPT_HOME/metrics.js $2 $3
}
```

```
case "$1" in
'deploy')
 doDeploy $*
 ;;
'create')
 doCreate $*
 ;;
'addMember')
 doAddMember $*
 ;;
'status')
 doStatus $*
 ;;
'restart')
 doRestart $*
 ;;
'avail')
 doAvail $*
 ;;
'metrics')
 doMetrics $*
 ;;
*)
        usage $*
        ;;
esac
```

This script uses a configuration file, **`groupcontrol.conf`**, which defines the connection information to connect to the JBoss ON server (which is required by the JBoss ON CLI).

```
##
## This file contains some defaults for the groupcontrol script
##
JON_CLI_HOME=cliRoot/rhq-remoting-cli-4.9.0.JON320GA
JON_HOST=localhost
JON_PORT=7080

# The user you want to connect with
JON_USER=rhqadmin

# if you omit the password here, you'll be prompted for it.
JON_PWD=rhqadmin
```

6.3. Defining Arguments and Other Parameters for the CLI Scripts

There may be multiple groups or some tasks (like searching for resources or running an operation) may have multiple options.

Each JavaScript file can define its own script options in **`args`** methods. At a minimum, each script should accept the name of the group on which to perform the task.

It is also a really good idea to define a **`usage`** function, so that each command can print what options are expected. For example:

```
function usage() {
        println("Usage: deploy groupName");
        throw "Illegal arguments";
}

if( args.length < 1 ) usage();
var groupName = args[0];
```

Note

When adding arguments for a script, be sure to set the proper number of tokens in the wrapper script for the CLI invocation. For example, for *groupName* and *fileName*, add **$2 $3**.

```
doDeploy() {
      $JON_CLI_HOME/bin/rhq-cli.sh $RHQ_OPTS -f $SCRIPT_HOME/deploy.js
$2 $3
}
```

Aside from the script for creating a group, every script must also include a search for the group to perform the operations on. For example:

```
groupcriteria = new ResourceGroupCriteria();
groupcriteria.addFilterName(groupName);

var groups =
ResourceGroupManager.findResourceGroupsByCriteria(groupcriteria);
if( groups != null ) {
  if( groups.size() > 1 ) {
        println("Found more than one group.");
  }
  else if( groups.size() == 1 ) {
     group = groups.get(0);
  }
}
```

6.4. Creating a Group: group.js

Set up the script. This script only uses a single argument, for the name of the new group (*groupName*). The resource type in the example is hard-coded to JBossAS5, which is a JBoss AS 5 server; optionally, it is possible to also add arguments to set the plug-in name and type so that other JBoss versions could be specified.

```
function usage() {
        println("Usage: deploy groupName");
        throw "Illegal arguments";
}

if( args.length < 1 ) usage();
var groupName = args[0];
```

Create the group:

```
var rg = new ResourceGroup(resType);
rg.setRecursive(false);
rg.setDescription("Created via groupcontrol scripts on " + new
java.util.Date().toString());
rg.setName(groupName);

rg = ResourceGroupManager.createResourceGroup(rg);

var resType = ResourceTypeManager.getResourceTypeByNameAndPlugin("JBossAS 5
Server","JBossAS5");
```

6.5. Adding Resources to a Group: addMember.js

Set up the script. This identifies three required arguments for the script:

- *groupName* for the group to add the resources to
- *resourceName* for the name of the resource to add; this is one of the search criteria
- *resourceTypeName* for the type of resource to add; this is one of the search criteria

This also includes a search to find the group specified in the argument.

```
function usage() {
        println("Usage: addMember groupName resourceName resourceTypeName");
        throw "Illegal arguments";
}

if( args.length < 3 ) usage();
var groupName = args[0];
var resourceName = args[1];
var resourceTypeName = args[2];

groupcriteria = new ResourceGroupCriteria();
groupcriteria.addFilterName(groupName);

var groups =
ResourceGroupManager.findResourceGroupsByCriteria(groupcriteria);
if( groups != null ) {
  if( groups.size() > 1 ) {
        println("Found more than one group.");
  }
  else if( groups.size() == 1 ) {
     group = groups.get(0);
  }
}
```

Search for the resources to add to the group. The script is designed to add only a single resource to the group, so the given search criteria, *resourceName* and *resourceTypeName*, must be specific enough to match only a single resource.

```
criteria = new ResourceCriteria();
criteria.addFilterName(resourceName);
```

```
criteria.addFilterResourceTypeName(resourceTypeName);

var resources = ResourceManager.findResourcesByCriteria(criteria);
if( resources != null ) {
  if( resources.size() > 1 ) {
        println("Found more than one JBossAS Server instance. Try to
specialize.");
     for( i =0; i < resources.size(); ++i) {
          var resource = resources.get(i);
          println("  found " + resource.name );
     }
  }
  else if( resources.size() == 1 ) {
     resource = resources.get(0);
     println("Found one JBossAS Server instance. Trying to add it.");
     println("  " + resource.name );
        ResourceGroupManager.addResourcesToGroup(group.id, [resource.id]);
     println("  Added to " + group.name + "!");
  }
  else {
        println("Did not find any JBossAS Server instance matching your
pattern. Try again.");
  }
}
```

When this script is run, it prints the name of the found JBoss instance and that it was added to the group.

```
[jsmith@server cli]$ ./wrapper.sh addMember myGroup "JBossAS App 1" "JBossAS
Server"
Remote server version is: 3.0.1.GA (b2cb23b:859b914)
Login successful
Found one JBossAS Server instance. Trying to add it.
  AS server.example.com JBossAS App 1
  Added to myGroup!
```

6.6. Getting Inventory and Status Information: status.js

This is a simple little script, just to print the current status of all the JBoss instances in the group.

As with the other scripts, set up the group information.

```
function usage() {
        println("Usage: status groupName");
        throw "Illegal arguments";
}

if( args.length < 1 ) usage();
var groupName = args[0];

groupcriteria = new ResourceGroupCriteria();
groupcriteria.addFilterName(groupName);

var groups =
ResourceGroupManager.findResourceGroupsByCriteria(groupcriteria);
if( groups != null ) {
```

```
  if( groups.size() > 1 ) {
        println("Found more than one group.");
  }
  else if( groups.size() == 1 ) {
     group = groups.get(0);
  }
}
```

Also include information to search for the resources, based on the group:

```
criteria = new ResourceCriteria();

var groupArray= new Array();
groupArray[0]=group.id;
criteria.addFilterExplicitGroupIds(groupArray);

var resources = ResourceManager.findResourcesByCriteria(criteria);
for( i =0; i < resources.size(); ++i) {
     var resource = resources.get(i);
     println("  found " + resource.name );
}
```

Then, run through the resources and print their availability.

```
var server = ProxyFactory.getResource(resource.id);
var avail  =
AvailabilityManager.getCurrentAvailabilityForResource(server.id);

println("  " + server.name );
println("    - Availability: " + avail.availabilityType.getName());
println("    - Started     : " + avail.startTime.toGMTString());
println("");

var avail =
AvailabilityManager.getCurrentAvailabilityForResource(server.id);

if( avail.availabilityType.toString() == "DOWN" ) {
           println("  Server is DOWN. Please first start the server and run
this script again!");
           println("");
}
```

When the script is run, it prints the availability and last start time for the servers.

```
[jsmith@server cli]$ ./wrapper.sh status myGroup
Remote server version is: 3.0.1.GA (b2cb23b:859b914)
Login successful
  found AS server.example.com JBossAS App 1
  AS server.example.com JBossAS App 1
    - Availability: UP
    - Started     : 11 Feb 2012 04:07:37 GMT
```

6.7. Starting, Stopping, and Restarting the Server: restart.js

Set up the script with the usage information and the group search, as in Section 6.6, “Getting Inventory and Status Information: status.js”.

This example only performs one operation, restarting a JBoss server. It iterates through all the resources in the group.

It is possible to write similar scripts for starting and stopping the server.

- **`shutdown()`** for AS4 servers and **`shutDown()`** for AS5 servers
- **`start()`**

```
criteria = new ResourceCriteria();

var groupArray= new Array();
groupArray[0]=group.id;
criteria.addFilterExplicitGroupIds(groupArray);

var resources = ResourceManager.findResourcesByCriteria(criteria);
for( i =0; i < resources.size(); ++i) {
     var resource = resources.get(i);
     var resType = resource.resourceType.name;
     println("  found " + resource.name );

     if( resType != "JBossAS Server") {
          println("    ---> Resource not of required type. Exiting!");
          usage();
     }

     var server = ProxyFactory.getResource(resource.id);
     println("    stopping " + server.name + "....");
     try {
         server.shutdown()
     }
     catch( ex ) {
         println("   --> Caught " + ex );
     }

     println("    restarting " + server.name + "....." );
     try {
         server.start();
     }
     catch( ex ) {
         println("   --> Caught " + ex );
     }
}
```

6.8. Deploying Applications to the Group Members: deploy.js

Set up the usage information and the group search as in the other scripts, then use the deployment script described in Section 3, “Example: Scripting Resource Deployments (JBoss EAP 5)”.

The script uses two parameters, one for the group name and one for the file to upload.

As one easy improvement, the last part of Section 3.2, "Checking the JBoss ON Groups and Inventory" stops the JBoss server, uploads the content, and restarts it. Instead, simply check that the server is running first, and then upload the content:

```
// we need check to see if the given server is up and running
var avail =
AvailabilityManager.getCurrentAvailabilityForResource(server.id);

// unfortunately, we can only proceed with deployment if the server is
running. Why?
if( avail.availabilityType.toString() == "DOWN" ) {
    println("  Server is DOWN. Please first start the server and run this
script again!");
    println("");
    continue;
}
```

6.9. Scheduling an Availability Operation: avail.js

Unlike the other tasks in this script set, the operation task is run on the agent, so it is not necessary to search for the group or JBoss resource. This runs an availability scan on the agent; it is also possible to run a specific command on the agent using the **`Execute prompt command`** operation.

First, get a list of all agent resources:

```
println("Scanning all RHQ Agent instances");
var rc = ResourceCriteria();
var resType = ResourceTypeManager.getResourceTypeByNameAndPlugin("RHQ
Agent", "RHQAgent");
rc.addFilterPluginName("RHQAgent");
rc.addFilterResourceTypeName("RHQ Agent");
rc.addFilterParentResourceTypeId("10001");

var resources = ResourceManager.findResourcesByCriteria(rc).toArray();

var idx=0;
for( i in resources ) {
     if( resources[i].resourceType.id == resType.id ) {
          resources[idx] = resources[i];
          idx = idx + 1;
     }
}
```

Then, traverse the agents array and schedule the operation:

```
for( a in resources ) {
     var agent = resources[a]

     var resType = agent.resourceType.name;
     println("  Found resource " + agent.name + " of type " + resType + "
and ID " + agent.id);

     println("  executing availability scan on agent" );
     println("    -> " + agent.name + " / " + agent.id);
     var config = new Configuration();
```

```
     config.put(new PropertySimple("changesOnly", "true") );
     var ros = OperationManager.scheduleResourceOperation(
          agent.id,
          "executeAvailabilityScan",
          0,
          1,
          0,
          10000000,
          config,
          "test from cli"
     );

     println(ros);
     println("");
}
```

6.10. Gathering Metric Data of Managed Servers: metrics.js

JBoss ON collects a number of metrics for each resource type. This information can be retrieved by using the **`findLiveData`** method, which returns the current active value for the resource.

This script takes two arguments, the *groupName* and the *metricName*. As with the other scripts, this searches for the group and then the resource by the group ID.

```
function usage() {
        println("Usage: metrics groupName metricName");
        throw "Illegal arguments";
}

if( args.length < 2 ) usage();
var groupName = args[0];
var metricName = args[1];

groupcriteria = new ResourceGroupCriteria();
groupcriteria.addFilterName(groupName);

var groups =
ResourceGroupManager.findResourceGroupsByCriteria(groupcriteria);
if( groups != null ) {
  if( groups.size() > 1 ) {
        println("Found more than one group.");
  }
  else if( groups.size() == 1 ) {
     group = groups.get(0);
  }
}

criteria = new ResourceCriteria();
var groupArray= new Array();
groupArray[0]=group.id;
criteria.addFilterExplicitGroupIds(groupArray);
```

The actual metric search looks for the metrics available to the resource type (hard-coded to JBoss AS 5 in this example). The metric itself is identified solely by the *metricName* argument.

```
var rt = ResourceTypeManager.getResourceTypeByNameAndPlugin("JBossAS 5
Server","JBossAS5");
var mdc = MeasurementDefinitionCriteria();
mdc.addFilterDisplayName(metricName);
mdc.addFilterResourceTypeId(rt.id);
var mdefs =
MeasurementDefinitionManager.findMeasurementDefinitionsByCriteria(mdc);
var resources = ResourceManager.findResourcesByCriteria(criteria);
var metrics = MeasurementDataManager.findLiveData(resources.get(0).id,
[mdefs.get(0).id]);

if( metrics !=null ) {
        println(" Metric value for " + resources.get(0).id + " is " +
metrics );
}
```

When the script is run, it prints the resource ID and the current value for the metric.

```
[jsmith@server cli]$ ./wrapper.sh metrics myGroup "Active Thread Count"
Remote server version is: 3.0.1.GA (b2cb23b:859b914)
Login successful
 Metric value for 10003 is [MeasurementDataNumeric[value=[64.0],
MeasurementData [MeasurementDataPK: timestamp=[Wed Feb 15 22:14:38 EST 2012],
scheduleId=[1]]]]
```

7. Example: Deploying a Standalone Server to a Cluster (JBoss EAP 6)

There is a sample script in the *cliRoot*/rhq-remoting-cli-4.9.0.JON320GA**/samples** directory the defines a series of functions that allow a JBoss EAP 6 standalone server to be added to an existing cluster.

JBoss EAP 6 has the idea of *domains*, which can be subdivided into *server groups* which share configuration. Server groups allows multiple server instances to have consistent, uniform configuration settings, to share profiles, and to deploy the same applications through a central command point.

However, JBoss EAP 6 still has the idea of a classic standalone server, a single JBoss instance that is unaffiliated with a domain or server group. A standalone server can be joined in a *cluster*, a loose association of standalone servers that work together to distribute the work load, either for load balancing or high availability. Unlike a domain, a cluster does not manage configuration or content.

JBoss ON provides the centralized management over configuration and content for standalone servers, introducing some of the ease of maintenance that EAP 6 domains offer. The ***cliRoot*/rhq-remoting-cli-4.9.0.JON320GA/samples/modules/jbossas.js** file defines some useful functions that simplify identifying cluster servers, deployed content, and relevant configuration settings.

The bulk of the functions defined are private and are well-documented within the ***cliRoot*/rhq-remoting-cli-4.9.0.JON320GA/samples/modules/jbossas.js** file, so they are out of the scope of this example. The main purpose of this is example is to review the two public functions which perform two distinct cluster management tasks:

- **addToCluster**
- **copyDeployments**

7.1. Adding a Standalone EAP 6 Server to a Cluster

A cluster is defined by servers which use the same multicast properties on the same network. If the servers are configured with the same settings, they all automatically associate with each other in a cluster.

There are three configuration properties for a standalone server within a cluster:

- An identifying name for the JBoss instance to use within the cluster (the *node-name*)
- Multicast settings, including the multicast port, a UDP port, and multicast address
- Socket-binding group information used by mod_cluster

A cluster does not directly manage either configuration properties or deployed content. However, if two EAP 6 standalone servers are in the JBoss ON inventory, then JBoss ON can work somewhat as a bridge, comparing the configuration and content deployments between servers and copying between them. That comparison is what the **`addToCluster`** function uses to add a standalone server to a cluster. It uses the configuration properties in an existing cluster member and copies them over to the standalone server.

Actually running the script requires the name of the standalone server, the name of an existing cluster member, a node name for the standalone server, and a boolean that sets whether to copy over the deployments from the existing cluster to the new member.

Assuming that you already know resource IDs of the standalone server and an existing cluster member:

```
[root@server bin]# ./rhq-cli.sh -u rhqadmin -p rhqadmin
rhqadmin@localhost:7080$ var newAs7Resource =
ProxyFactory.getResource(10381)
rhqadmin@localhost:7080$ var existingClusterMemberResource =
ProxyFactory.getResource(10577)
rhqadmin@localhost:7080$ var newNodeName = jbas7-standalone1
rhqadmin@localhost:7080$ addToCluster(newAs7Resource, newNodeName,
existingClusterMemberResource, true)
```

The **`addToCluster`** function makes some assumptions that the cache configuration between the two servers is compatible, both for concrete caches and the cache containers for individual subsystems.

The script runs through a few steps to copy the configuration from the cluster to the standalone server:

1. It checks the plug-in connection properties in the cluster server and compares them to the plug-in connection properties in the standalone server. If necessary, it copies over the plug-in configuration from the cluster server and restarts the standalone server, loading the new configuration.
2. It checks the given node name for the standalone server. If necessary, it changes the default node name to the one passed with the function.
3. It then compares the socket-binding settings for the cluster and standalone servers. If necessary, it copies over the socket-binding configuration for the jgroups, messaging, and mod_cluster bindings from the cluster server and restarts the standalone server.
4. If set, then the script copies the deployments from the cluster server to the standalone server and restarts the standalone server.

The first part of the function pulls the plug-in configuration (defined by the private function **`_getClusterSignificantConfig`**) for the cluster and then for the standalone server.

```
function addToCluster(newAs7Resource, newNodeName,
existingClusterMemberResource, copyDeployments) {
    println("Reading config of the existing cluster member");
    var clusterConfig =
```

```
_getClusterSignificantConfig(existingClusterMemberResource);

    println("Reading config of the new member");
    var memberConfig = _getClusterSignificantConfig(newAs7Resource);

    var memberResourceConfiguration = newAs7Resource.resourceConfiguration;
```

If the configuration properties are different, then the script copies over the new plug-in configuration and restarts the standalone server to load the new connection settings.

```
    if (memberConfig['config'] != clusterConfig['config']) {
        println("The configurations of the servers differ.\n" +
            "The new cluster member's configuration will be changed to match
the configuration of the existing member.");

        //switch to the same configuration
        var pluginConfig = newAs7Resource.pluginConfiguration;
        pluginConfig.getSimple('config').setValue(clusterConfig['config']);
        newAs7Resource.updatePluginConfiguration(pluginConfig);

        //we need to restart straight away so that we see the changes to the
        //rest of the configuration caused by the change of current config.
        println("Restarting the new cluster member to switch it to the new
configuration.");
        newAs7Resource.restart();

        //refresh the resource
        newAs7Resource = ProxyFactory.getResource(newAs7Resource.id);

        //refresh the cluster specific config after the restart with the new
        //config
        memberConfig = _getClusterSignificantConfig(newAs7Resource);
        memberResourceConfiguration = newAs7Resource.resourceConfiguration;
    }
```

It then applies the node name that was given with the script, if it is different than the one set by default.

```
    //now check what's the node name we see
    if (memberConfig['node-name'] != newNodeName) {
        println("Updating the node name of the new cluster member from '" +
memberConfig['node-name'] + "' to '" + newNodeName + "'");
        _updateNodeName(memberResourceConfiguration, newNodeName);

newAs7Resource.updateResourceConfiguration(memberResourceConfiguration);
     }
```

The next configuration area for the cluster is the socket-binding settings for important subsystems, jgroups, messaging, and mod_cluster.

```
    //now apply the socket binding changes for jgroups and other cluster
    //significant subsystems
    //first find the socket binding group config in the new member
    for(i in newAs7Resource.children) {
        var child = newAs7Resource.children[i];
        if (child.resourceType.name == 'SocketBindingGroup' &&
```

```
                child.resourceType.plugin == 'jboss-as-7') {


            println("Updating socket bindings of jgroups, messaging and
modcluster subsystems");

            var portOffset =
javascriptString(child.resourceConfiguration.getSimpleValue('port-offset',
'0'));
            var clusterMemberPortOffset = clusterConfig['port-offset'];

            var newConfig = child.resourceConfiguration.deepCopy(false);

            _updateSocketBindings(newConfig, portOffset,
clusterMemberPortOffset, clusterConfig['jgroups']);
            _updateSocketBindings(newConfig, portOffset,
clusterMemberPortOffset, clusterConfig['messaging']);
            _updateSocketBindings(newConfig, portOffset,
clusterMemberPortOffset, clusterConfig['modcluster']);

            child.updateResourceConfiguration(newConfig);
        }
    }

    println("Restarting the new member for the new socket bindings to take
effect.");
    newAs7Resource.restart();
```

Although not strictly part of the cluster configuration, part of what JBoss ON can do is compare other parts of the resource setup, like deployed applications. Synchronizing the deployed applications between one server and another, even standalone instances, helps maintain consistency, and this can be done conveniently at the time that a server is added to a cluster simply by syncing the given cluster server's deployments.

```
    if (copyDeployments) {
        println("Copying the deployments to the new cluster member...");
        copyDeployments(existingClusterMemberResource, newAs7Resource);

        println("Restarting the new cluster member.");
        newAs7Resource.restart();
    }
}
```

7.2. Copying Deployed Applications Between Standalone Servers

While EAP 6 server groups manage content centrally for all group members, standalone servers are on their own. JBoss ON can help as an intermediary to sync application content between separate server instances. The **`addToCluster`** function has this as an option when joining a standalone server to a cluster. The **`copyDeployments`** function can copy deployments between any two standalone instances.

Invoking the function requires only the name of the source EAP 6 server (the one to copy the deployments *from*) and then the name of the target EAP 6 server (the one to copy the deployments*to*).

Assuming that you already know resource IDs of the two EAP 6 server resources, set the source and target resources. For example, in interactive mode:

```
[root@server bin]# ./rhq-cli.sh -u rhqadmin -p rhqadmin
rhqadmin@localhost:7080$ var source = ProxyFactory.getResource(10381)
rhqadmin@localhost:7080$ var target = ProxyFactory.getResource(10577)
rhqadmin@localhost:7080$ copyDeployments(source, target)
```

The first part of the function gets the server resource IDs.

```
function copyDeployments(sourceAS7, targetAS7) {
    if (typeof sourceAS7 == 'object') {
        sourceAS7 = sourceAS7.id;
    }

    if (typeof targetAS7 == 'object') {
        targetAS7 = targetAS7.id;
    }
```

All of the deployed applications are listed as children of the source JBoss EAP 6 server. The **`copyDeployments`** function retrieves each deployment by searching for all of the children of the server that are of a deployment resource type.

```
    var deploymentResourceType =
ResourceTypeManager.getResourceTypeByNameAndPlugin('Deployment', 'jboss-as-
7');

    var deploymentsCrit = new ResourceCriteria;
    deploymentsCrit.addFilterParentResourceId(sourceAS7);
    deploymentsCrit.addFilterResourceTypeId(deploymentResourceType.id);

    var unlimitedPageControl = PageControl.unlimitedInstance;

    var sourceDeployments =
ResourceManager.findResourcesByCriteria(deploymentsCrit);
    var iterator = sourceDeployments.iterator();
    while (iterator.hasNext()) {
        var deploymentResource = iterator.next();
        //get a resource proxy for easy access to configurations, etc.
        deploymentResource =
ProxyFactory.getResource(deploymentResource.id);

 println("Copying deployment " + deploymentResource.name);
```

Each discovered deployment is then copied over as a new child resource to the target server. These are content-backed resources, so they are exported and uploaded as content. The function also searches for and pulls in the content metadata and the content history, so that any important historical information about the deployment is also copied over.

```
        var installedPackage =
ContentManager.getBackingPackageForResource(deploymentResource.id);
        var content = ContentManager.getPackageBytes(deploymentResource.id,
installedPackage.id);

        var runtimeName =
deploymentResource.resourceConfiguration.getSimpleValue('runtime-name',
deploymentResource.name);
```

```
        var deploymentConfiguration = new Configuration;
        deploymentConfiguration.put(new PropertySimple('runtimeName',
runtimeName));

        //so now we have both metadata and the data of the deployment, let's
        //push a copy of it to the target server
        var history =
ResourceFactoryManager.createPackageBackedResource(targetAS7,
            deploymentResourceType.id, deploymentResource.name,
            deploymentResource.pluginConfiguration,
            installedPackage.packageVersion.generalPackage.name,
            installedPackage.packageVersion.version,
            installedPackage.packageVersion.architecture.id,
            deploymentConfiguration, content, null);

        while (history.status.name() == 'IN_PROGRESS') {
            java.lang.Thread.sleep(1000);
            //the API for checking the create histories is kinda weird..
            var histories =
ResourceFactoryManager.findCreateChildResourceHistory(targetAS7, null, null,
unlimitedPageControl);
            var hit = histories.iterator();
            var found = false;
            while(hit.hasNext()) {
                var h = hit.next();

                if (h.id == history.id) {
                    history = h;
                    found = true;
                    break;

                }
            }

            if (!found) {
                throw "The history object for the deployment seems to have
disappeared, this is very strange.";
            }
        }

        println("Deployment finished with status: " +
history.status.toString() +
            (history.status.name() == 'SUCCESS' ? "." : (", error message: "
+ history.errorMessage + ".")));
    }
}
```

8. Example: Deploying Applications Through Bundles (General)

Bundles are a very clean and easy way to deploy full applications, configuration files, or other content to resources. Whether a given resource type supports bundles is defined in its plug-in descriptor. By default, platform resources and JBoss AS 4, 5, and 6 resources all support bundles.

Bundles are convenient from an administrative perspective because all of the content is maintained in a single place that is resource-agnostic. The main bundle entry or bundle definition contains a set of versions of

the actual bundle files and a set of destinations for where that content can be deployed. A destination is a combination of a compatible group, resource type, and directory path. When a version is actually deployed to a destination, it is saved as a specific deployment for that destination.

The bundles system maintains multiple versions of a given package and can deploy any of those versions to any destination. This is a great workflow for application lifecycles, since a stable version can be deployed to production servers while a development version can be deployed to test machines. Having each deployment represented as a different child of the destination makes it easy to revert changes; you can move from the live version to a previous version and know exactly what that previous deployment looked like.

8.1. Setting up Bundle Versions and Destinations

There are two parts to the bundle definition: the bundle version and the destination. Both of these parts are set up independently, and then saved into the final definition.

The **bundles.js** script in the ***cliRoot*/rhq-remoting-cli-4.9.0.JON320GA/samples/modules/** directory defines a set of custom functions that quickly create a bundle version and definition. (The **bundles.js** script requires the **util.js** script.)

To create the destination, give the absolute path on the local system to the bundle archive to upload.

```
rhqadmin@localhost:7080$ var path = '/export/files/myApp.zip'
rhqadmin@localhost:7080$ createBundleVersion(path)
```

The **createBundleVersion** function in the **bundles.js** file uploads the files as a byte array.

```
function createBundleVersion(pathToBundleZipFile) {
        var bytes = getFileBytes(pathToBundleZipFile)
        return BundleManager.createBundleVersionViaByteArray(bytes)
}
```

The second part of the bundle definition is creating at least one destination where the bundle version could be deployed. Creating a destination requires two things to exist already:

- A bundle version (which was just created with **createBundleVersion**)
- A compatible group

A destination is a combination of a compatible group and the directory to deploy to. Each resource type defines its own available base directory, then a subdirectory beneath that root can be specified as the deployment directory.

The other configuration properties are details for the entry, such as the destination name and description.

```
rhqadmin@localhost:7080$ var destinationName = 'New Destination'
rhqadmin@localhost:7080$ var description = 'My new example destination'
rhqadmin@localhost:7080$ var bundleName = 'myApp'
rhqadmin@localhost:7080$ var groupName = 'Linux Group'
rhqadmin@localhost:7080$ var baseDirName = '/'
rhqadmin@localhost:7080$ var deployDir = 'var/www/html/'
rhqadmin@localhost:7080$ createBundleDestination(destinationName,
description, bundleName, groupName, baseDirName, deployDir)
```

The **createBundleDestination** function runs the searches for the group and the bundle based on the specified names, which makes it possible to set up the destination without having to run additional searches.

```
function createBundleDestination(destinationName, description, bundleName,
groupName, baseDirName, deployDir) {
        var groupCrit = new ResourceGroupCriteria;
        groupCrit.addFilterName(groupName);
        var groups =
ResourceGroupManager.findResourceGroupsByCriteria(groupCrit);

... 8< ...

        var group = groups.get(0);

        var bundleCrit = new BundleCriteria;
        bundleCrit.addFilterName(bundleName);
        var bundles = BundleManager.findBundlesByCriteria(bundleCrit);

... 8< ...
}
```

8.2. Deploying Bundles

Deploying a bundle sends a bundle version to a specific destination. The ***cliRoot*/rhq-remoting-cli-4.9.0.JON320GA/samples/modules/bundles.js** file has a function, **deployBundle**, which makes this pretty easy, but you need to obtain some information first.

Get the ID for the destination. This searches for the destination by name.

```
rhqadmin@localhost:7080$ var destinationName = "New Destination"
rhqadmin@localhost:7080$ var destcrit = new BundleDestinationCriteria()
rhqadmin@localhost:7080$ destcrit.addFilterName(destinationName)

var dest = BundleManager.findBundleDestinationsByCriteria(destcrit)
```

Then, get the ID number for the bundle version to deploy. *Any* version can be deployed, not just the most recent. This little script prints all of the versions for the bundle, with their ID numbers.

```
rhqadmin@localhost:7080$ var crit = new BundleVersionCriteria()

rhqadmin@localhost:7080$ crit.addFilterBundleName(name)

rhqadmin@localhost:7080$ var vers =
BundleManager.findBundleVersionsByCriteria(crit)

rhqadmin@localhost:7080$ if( vers != null ) { \
rhqadmin@localhost:7080$   if( vers.size() > 1 ) { \
rhqadmin@localhost:7080$      for( i =0; i < vers.size(); ++i) { \
rhqadmin@localhost:7080$           ver = vers.get(i); \
rhqadmin@localhost:7080$           println("Version: " + ver.version + "   "
+ "ID: " + ver.id) \
rhqadmin@localhost:7080$      } \
rhqadmin@localhost:7080$   } \
rhqadmin@localhost:7080$   else if( vers.size() == 1 ) { \
rhqadmin@localhost:7080$      ver = vers.get(0); \
rhqadmin@localhost:7080$      println("Version: " + ver.version + + "   " +
"ID: " + ver.id) \
```

```
rhqadmin@localhost:7080$   } \
rhqadmin@localhost:7080$ }
Version: 2.0 ID: 10021
Version: 1.0 ID: 10012
```

With those two ID numbers, you can deploy the bundle. The first parameter is the destination ID, then the bundle version ID, then a configuration object if the bundle configuration has any tokens to realize. In this example, no properties are passed, so the value is null. Details about the configuration are in the comments in the **bundles.js** file and general configuration information is in Section 5.2, “Changing Simple Configuration Properties”.

```
rhqadmin@localhost:7080$ deployBundle(dest.get(0).id,10021,null,'my
description',true)
BundleDeployment:
                     bundleVersion:
BundleVersion[id=10021,name=null,version=null]
                     configuration: Configuration[id=15021]
                             ctime: 1337286719259
                       description: my description
                       destination: BundleDestination[id=10021,
bundle=driftBundle, group=Linux Group - Thu May 10 15:10:28 EDT 2012,
name=NewDestination]
                          duration: 0
                      errorMessage:
                                id: 10051
                              live: true
                             mtime: 1337286719259
                              name: Deployment [1] of Version [2.0] to
[NewDestination]
        replacedBundleDeploymentId:
               resourceDeployments: [BundleResourceDeployment: bdd=
[BundleDeployment[id=10051, name=Deployment [1] of Version [2.0] to [new-
test]]], resource=[Resource[id=10001, uuid=535b3f54-0bd8-4653-bdd3-
323ea69b98fd, type={Platforms}Linux, key=gs-dl585g2-
01.rhts.eng.bos.redhat.com, name=server.example.com, parent=<null>,
version=Linux 2.6.32-220.el6.x86_64]]]
                           status: Failure
                      subjectName: rhqadmin
                tags:
```

The **deployBundle** function runs through a couple of steps to manage the deployment. This uses one of the functions from the **util.js** file to convert the deployment configuration (if any is sent) into the proper into a hash.

```
... 8< ...

        var deploymentConfig = deploymentConfiguration;
        if (!(deploymentConfiguration instanceof Configuration)) {
                deploymentConfig = asConfiguration(deploymentConfiguration);
 }
```

The next creates the deployment (through the remote API) and then schedules the deployment.

```
... 8< ...
        var deployment =
```

```
BundleManager.createBundleDeployment(bundleVersionId, destinationId,
description, deploymentConfig);

        deployment = BundleManager.scheduleBundleDeployment(deployment.id,
isCleanDeployment);
... 8< ...
```

8.3. Reverting a Bundle

Reverting a bundle automatically moves a destination one step backward, from whatever version is currently deployed to whatever version was last deployed.

This is done only with the remote API, not using any functions from the **`bundles.js`** file.

The method to run is **`scheduleRevertBundleDeployment`**. This requires two interesting pieces of information. The first is the destination ID, which can be retrieved with a simple criteria search.

```
rhqadmin@localhost:7080$ var destinationName = "NewDestination"
rhqadmin@localhost:7080$ var destCrit = new BundleDestinationCriteria()
rhqadmin@localhost:7080$ destCrit.addFilterName(destinationName)

rhqadmin@localhost:7080$ var dest =
BundleManager.findBundleDestinationsByCriteria(destCrit)
```

The next, and more interesting, piece of information is the deployment description. The description is what is passed to the revert method to help identify the deployment to revert.

```
rhqadmin@localhost:7080$ var depCrit = new BundleDeploymentCriteria()
rhqadmin@localhost:7080$ depCrit.addFilterDestinationName(destinationName)
rhqadmin@localhost:7080$ var deploy =
BundleManager.findBundleDeploymentsByCriteria(depCrit)
rhqadmin@localhost:7080$ var dep = deploy.get(0);
rhqadmin@localhost:7080$ var description = dep.description;
```

The last part actually invokes the method.

```
rhqadmin@localhost:7080$
BundleManager.scheduleRevertBundleDeployment(dest.get(0).id, description,
true)
```

9. Example: Remedying Resource Drift

Maintain servers, particularly production or business critical servers and applications, requires keeping rein on the configuration files and packages on those systems. When an unexpected change occurs, the system moves away from the administrator-defined state. That is *configuration drift*.

JBoss ON can monitor configuration files and target directories and track any changes to those area. This uses a *drift definition* which sets where the JBoss ON agent monitors configuration and at what frequency. If drift is detected, then the JBoss ON server can fire an alert and run an alert CLI script that reverts, or remedies, the changed configuration files.

9.1. The Plan for the Scripts

There are two different scripts in play because there are two different sets of situations for managing drift:

- First, there is a script to set up drift for a resource. This shell script runs through a series of setup steps at once:
    - It creates the drift definition for a resource (through the **`driftDef.js`** CLI script).
    - It creates a generic **`deploy.xml`** recipe, zips the drift directory, and creates a new bundle and bundle deployment (through the **`createBundle.js`** CLI script).
    - After waiting for the initial snapshot, it then pins the snapshot to the definition (through the **`snapshot.js`** CLI script).

  All of those files are generated by the shell script.
- An alert definition has to be created through the UI (not the CLI), but it can be configured to use any drift detection as a condition and then to run a server-side script in response. This second script simply deploys the bundle that was made of the pristine base directory and overwrites the drift.

9.2. Setting up the Drift Definition and Preparing the Bundle

The setup script actually runs through three CLI scripts and some system commands. Having all of the steps in a single script makes it possible to set up a drift definition and a backup bundle by running a single command:

```
[root@server ~]# ./driftBundle.sh
```

Note

Both drift definitions and bundle deployments take a lot of resource- and infrastructure-specific settings. The **`driftBundle.sh`** script in this example defines a lot of variables in the script to account for each piece of information.

The variables could be defined using a **`.conf`** or even a set of **`.conf`** files (cf. Section 6.2, “Creating the Wrapper Script and .conf File”), but for simplicity in this example, all of the variables are defined in the **`driftBundle.sh`** script itself.

The first part of the script simply defines the connection settings to use when running the JBoss ON CLI. This example only defines a username and password, so it assumes that the script is run on a system which also has a JBoss ON server running locally. The options could be edited to supply a remote JBoss ON server hostname and port.

There are three general variables defined:

- The location of the **`rhq-cli.sh`** script
- Any options, such as the username and password, to pass with the CLI command
- The directory to use both to save the generated JavaScript files and to use for the path to JavaScript files

```
#!/bin/bash
# options for the CLI
CLI='cliRoot/rhq-remoting-cli-4.9.0.JON320GA/bin/rhq-cli.sh'
OPTS=' -u rhqadmin -p rhqadmin'
```

```
SCRIPTS='/opt'
```

The first part of the script sets up the drift definition. By default, drift is only enabled for a handful of resource types — JBoss servers, Tomcat servers, and platforms — so it is easiest to identify the resource based on a combination of its resource type and name.

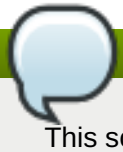
Once the resource is identified, then the definition can be created. The full list of possible definition settings is covered in the drift documentation, but a general definition will identify the base directory to monitor, set some rules about what files or subdirectories to ignore (like log files), and set an interval or frequency for drift detection scans.

All of these definition parameters are defined as individual variables in the shell script. In this example, drift is configured for a platform.

```
# set parameters for the drift definition
RESTYPE='Linux'
RESPLUGIN='Platforms'
RESNAME="server.example.com"
NAME='example drift'
DESC='drift from script'
BASEDIR='/opt/drift'
BASEDIRTYPE='fileSystem'
EXCLUDE='./logs/'
PATTERN=
MODE='normal'
INTERVAL='3600'
```

The shell script will eventually create a CLI script that is run automatically in the CLI. The first part of the CLI script defines a resource type criteria search for the platform, and then the resource platform itself.

```
driftDef() {
cat <<-EOF

//set the resource type
var resType =
ResourceTypeManager.getResourceTypeByNameAndPlugin("$RESTYPE","$RESPLUGIN");

//get the resource to associate with the drift definition
rcrit = ResourceCriteria()
rcrit.addFilterResourceTypeName("$RESTYPE")
rcrit.addFilterName("$RESNAME")
var resources = ResourceManager.findResourcesByCriteria(rcrit)
var res = resources.get(0)
```



Note

This script searches for a single resource to configure for drift. You could also create the script to search for multiple resource and add them to a compatible group, and the iterate through the compatible group to add the drift definition to each resource.

The next part configures the drift definition itself. The **`DriftDefinitionManager`** is a wrapper for a **`Configuration()`** object. The CLI script first calls for the default drift template for the given resource type and then creates a definition object based on that template.

```
//get the default template for the resource type
criteria = DriftDefinitionTemplateCriteria()
criteria.addFilterResourceTypeId(resType.id)
templates = DriftTemplateManager.findTemplatesByCriteria(criteria)
template = templates.get(0)
//create a new drift definition instance, based on the template
definition = template.createDefinition()
```

Once the configuration object is created, then the definition options are assigned values.

This script creates a real drift definition with one exception: it sets a very low scan interval, 30 seconds. In fact, that is the shortest configurable interval. This allows the agent to collect the initial snapshot fairly quickly, which helps the overall setup go faster. This interval will be reset to a more reasonable value (the one defined in the variables) at the end of the script execution.

```
//set the drift definition configuration options
definition.resource = res
definition.name = '$NAME'
definition.description = '$DESC'
definition.setAttached(false) // this is false so that template changes
don't affect the definition
// this is set low to trigger an early initial detection run
definition.setInterval(30)
var basedir = new
DriftDefinition.BaseDirectory(DriftConfigurationDefinition.BaseDirValueConte
xt.valueOf('$BASEDIRTYPE'),'$BASEDIR')
definition.basedir = basedir

// there can be multiple exclude statements made, as desired
var f = new Filter("$EXCLUDE", "$PATTERN") // location, pattern
definition.addExclude(f)

//this defaults to normal, which means that any changes will
// trigger an alert. plannedChanges is the other option, which
// disables alerting for drift changes.
definition.setDriftHandlingMode(DriftConfigurationDefinition.DriftHandlingMo
de.valueOf('$MODE'))
```

Once the configuration is complete, it needs to be written to the definition.

```
//apply the new definition to the resource
DriftManager.updateDriftDefinition(EntityContext.forResource(res.id),definit
ion)

EOF
}
```

Note

The drift definition uses an *entity context* rather than the resource ID alone to identify the resource. An entity context first identifies the type of object (the entity) and then its associated inventory ID.

There are actually several different steps for creating a "bundle" because there is no one part to a "bundle."

The script makes a ZIP archive of the given drift base directory, and that makes the bundle archive. Then, for defining the bundle, there are two steps. There is defining the *bundle destination*, which is a compatible group to which bundles (any bundles) can be deployed plus the location on the resources for deploying the bundles. Then the package itself is uploaded as a *bundle version*.

The variables define both the information for the bundle version and bundle archive and for the bundle destination.

There is one other variable included: the path to the CLI's samples directory. Helper functions to create bundle versions, to create bundle destinations, and to deploy specified bundles are already defined in the **`bundles.js`** sample script. Using those functions makes deploying bundles very easy.

```
# options for the bundle
SAMPLES='cliRoot/rhq-remoting-cli-4.9.0.JON320GA/samples'
DESTNAME='drift destination'
BUNDLEDESC='bundle to remediate drift'
BUNDLENAME='driftBundle'
GROUPNAME='Linux Group'
ZIP='driftBundle.zip'
BVER='1.0'
BUNDLE='/opt/bundles/'$ZIP
ARCHIVE='/opt/bundles/'$ZIP
```

This particular bundle deployment is pretty simple. The target bundle destination is the same as the drift base directory.

Since there are no tokens to realize or external content to pull in, just the backup archive itself, the recipe can be pretty simple. This script creates the recipe (**`deploy.xml`**) which is used in the bundle archive.

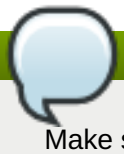
```
deploy() {
cat << _EOF_
<?xml version="1.0"?>
<project name="$BUNDLENAME" default="main"
        xmlns:rhq="antlib:org.rhq.bundle">
    <rhq:bundle name="$BUNDLENAME" version="$BVER"
description="$BUNDLEDESC">
        <rhq:deployment-unit name="drift" compliance="full">
            <rhq:archive name="$ZIP" exploded="true">
            </rhq:archive>
        </rhq:deployment-unit>
    </rhq:bundle>
<target name="main" />

</project>
_EOF_
}
```

The **`bundles.js`** sample script already defines all of the functions required to deploy bundles, but it relies on the **`util.js`** sample script. When the CLI is run non-interactively, there is no way to import an external script that another script requires.

So, this shell script first concatenates the **`bundles.js`** and **`util.js`** scripts together, and then appends the calls to create the bundle version and the bundle destination.

```
createBundle() {
cat $SAMPLES/util.js $SAMPLES/bundles.js
```

```
cat  << _EOF_

// set the location of the bundle archive
var path = '$BUNDLE'

// create the bundle version in JON
createBundleVersion(path)

// set all of the variables for the bundle destination
var destinationName = '$DESTNAME'
var description = '$BUNDLEDESC'
var bundleName = '$BUNDLENAME'
var groupName = '$GROUPNAME'
var baseDirName = '$BASEDIR'
var deployDir = "."

// create the new destinition in JON
createBundleDestination(destinationName, description, bundleName, groupName,
baseDirName, deployDir)

_EOF_
}
```



Note

Make sure that the resource already belongs to a compatible group and that the compatible group has a unique enough name so that it is the only one returned in the search.

The last CLI script created by the shell script pins the initial snapshot to the new drift definition. A *snapshot*, as the name implies, is a picture of the current settings of the base directory. Pinning a snapshot to a definition sets a baseline, or comparison, for the agent to use to evaluate drift. A pinned snapshot is a specific and identified configuration that must be maintained (as opposed to rolling changes).

Once the snapshot is pinned, this script then resets the drift definition configuration so that it uses a longer (more realistic) interval between scans.

```
snapshot() {
cat <<- _EOF_
//find the resource
rcrit = ResourceCriteria()
rcrit.addFilterResourceTypeName("$RESTYPE")
rcrit.addFilterName("$RESNAME")
var resources = ResourceManager.findResourcesByCriteria(rcrit)
var res = resources.get(0)

//find the new drift definition
criteria = DriftDefinitionCriteria()
criteria.addFilterName('$NAME')
criteria.addFilterResourceIds(res.id)
def = DriftManager.findDriftDefinitionsByCriteria(criteria)
definition = def.get(0)
definition.setInterval($INTERVAL)

// it is necessary to redefine the complete configuration when you're
```

```
// resetting the interval or the other values will be overwritten with
default
// or set to null
var basedir = new
DriftDefinition.BaseDirectory(DriftConfigurationDefinition.BaseDirValueConte
xt.valueOf('$BASEDIRTYPE'),'$BASEDIR')
definition.basedir = basedir
definition.name = '$NAME'
// there can be multiple exclude statements made, as desired
var f = new Filter("$EXCLUDE", "$PATTERN") // location, pattern
definition.addExclude(f)
DriftManager.updateDriftDefinition(EntityContext.forResource(res.id),definit
ion)

// pin to the initial snapshot, which is version 0
// this gets the most recent snapshot if that is the better version to use
// snap = DriftManager.getSnapshot(DriftSnapshotRequest(definition.id))
DriftManager.pinSnapshot(definition.id,0)
_EOF_
}
```

The last part of the script actually runs all of the defined JBoss ON CLI scripts and sets up both the drift definition and the bundle definition (as a backup in case any drift is detected).

There are two system commands sandwiched between the JBoss ON CLI scripts. The first is the **`zip`** commands to create the bundle archive. The second is a **`sleep`** command which pauses the script to give the JBoss ON agent time to collect the initial snapshot for drift before attempting to pin the snapshot.

```
# create the drift definition

driftDef > $SCRIPTS/driftDef.js
$CLI $OPTS -f $SCRIPTS/driftDef.js


# create the recipe file and then zip up the
# drift base directory to make the bundle archive

deploy > /deploy.xml
zip $ARCHIVE $BASEDIR
zip $BUNDLE $ARCHIVE /deploy.xml

# create the bundle from the recipe and archive
# and then create the bundle definition

createBundle > $SCRIPTS/createBundle.js
$CLI $OPTS -f $SCRIPTS/createBundle.js


# sleep to allow the server to get the first snapshot
# this only sleeps for a minute, but it really depends on your environment
# whether that is long enough

sleep 1m

# this pins the new snapshot to the new drift definition
# and then changes the drift interval to the longer, variable-specified
```

```
# value

snapshot > $SCRIPTS/snapshot.js
$CLI $OPTS -f $SCRIPTS/snapshot.js
```

There is no error handling in this shell script. If any step fails, like the initial snapshot taking longer than the sleep period, there is no indication of what went wrong aside from malformed drift or bundle configuration.

9.3. Remedying Drift

To remediate drift, define an alert in the UI and upload a CLI script which can be run, automatically, whenever drift is detected. All the script has to do is deploy the backup bundle to the resource, and there are several different ways to do that.

This example goes through all the basic steps: it pulls the resource information from the alert, searches for the bundle version, and then deploys it to the resource. One nifty thing about this script is that it writes a log file, capturing the alert information that triggered the remediation.

This script can be uploaded directly when the alert definition is created. Before uploading the script, be sure tc set the variables to the bundle destination and bundle version that you created when the drift definition was set up.

```
// - The 'alert' variable is seeded by the alert sender

// SET THESE VARIABLES
var bundleDestinationName = 'drift destination'
var bundleVersion = 1.0
var logFile = '/tmp/alert-cli-demo/logs/alert-' + alert.id + '.log'

// Log what we're doing to a file tied to the fired alert id
//
var e = exporter
e.setTarget( 'raw', logFile )

// Dump the alert
//
e.write( alert )

// get a proxy for the alerted-on Resource
//
var alertResource =
ProxyFactory.getResource(alert.alertDefinition.resource.id)

// Dump the resource
//
e.write( " " )
e.write( alertResource )


// Remediate file

// Find the Bundle Destination
//
var destCrit = new BundleDestinationCriteria()
destCrit.addFilterName( bundleDestinationName )
var result = BundleManager.findBundleDestinationsByCriteria( destCrit )
```

```
var dest = result.get( 0 )

// Find the Bundle Version
//
var versionCrit = new BundleVersionCriteria()
versionCrit.addFilterVersion( bundleVersion )
result = BundleManager.findBundleVersionsByCriteria( versionCrit )
var ver = result.get( 0 )

// Create a new Deployment for the bundle version and the destination
//
var deployment = BundleManager.createBundleDeployment(ver.getId(),
dest.getId(), 'remediate drift', new Configuration())

// Schedule a clean deploy of the deployment. This will wipe out the edited
file and lay down a clean copy
//
BundleManager.scheduleBundleDeployment(deployment.getId(), true)

e.write( " " )
e.write( "REMEDIATION COMPLETE!" )
```

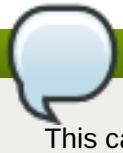
10. Example: Managing JBoss ON Server Configuration

Even in different environments, JBoss ON servers can share a lot of the same configuration. For example, different JBoss ON servers may manage a development environment, staging environment, and production environment, yet on all three, the servers use similar metric templates and configuration settings. To simplify managing separate but similar environments, JBoss ON can export the configuration for a server and then import that configuration into another server.

Any user with permissions to manage settings can export the server configuration.

10.1. Simple Export/Import Synchronization

At its simplest, synchronizing server settings exports all metrics and server configuration settings and then imports that information directly, without any adjustment to the data before its imported or filters on what data to import.



Note

This can be automated by including login information for both JBoss ON servers, and then running the script. This could also be broken into two scripts, one run against server1 and the other against server2. Using a wrapper script would allow you to run the export script, then to run a utility like SCP to copy over the archive, and then to run the import script.

The first part of the script exports the data from server1 and creates a zipped XML archive.

```
//log into the first server
rhq.login('rhqadmin','rhqadmin','server1.example.com','7080');

//export the settings
var ex = SynchronizationManager.exportAllSubsystems();
```

```
rhqadmin@localhost:7080$ saveBytesToFile(ex.exportFile, 'export.xml.gz');

// log out of the first server
rhq.logout()
```

The archive file then needs to be copied over to server2 in some way.

When the archive is copied over, it can then be imported into server2. The *null* parameter means that the import process uses the default settings in the XML file or, if the defaults are missing from the XML, that it uses the settings defined on the target server.

```
// log into the second server
rhq.login('rhqadmin','rhqadmin','server2.example.com','7080');

// import the settings
var data = getFileBytes('export.xml.gz');
SynchronizationManager.importAllSubsystems(data, null);

// log out of the second server
rhq.logout()
```

10.2. Changing Server Configuration Before Importing

Metrics schedules and server configuration are applied through *synchronizers*. Synchronizers control what elements are imported into the JBoss ON server and how to apply them to the server. The synchronizer has a default template which applies configuration changes to every import operation.

The synchronizer configuration can be changed to change what settings are imported into the target server.

The first part of the script would export the XML archive, as before.

```
//log into the first server
rhq.login('rhqadmin','rhqadmin','server1.example.com','7080');

//export the settings
var ex = SynchronizationManager.exportAllSubsystems();
rhqadmin@localhost:7080$ saveBytesToFile(ex.exportFile, 'export.xml.gz');

// log out of the first server
rhq.logout()
```

The XML file contains the full configuration information, so just checking that file can give you an idea of what settings to change.

On the second server, change the synchronizer settings.

1. Get the default definition.

   ```
   rhqadmin@localhost:7080$ var
   systemSettingsImportConfigurationDefinition =
   SynchronizationManager.getImportConfigurationDefinition('org.rhq.enter
   prise.server.sync.SystemSettingsSynchronizer')
   ```

2. Create a new configuration instance.

```
rhqadmin@localhost:7080$ var configurationObject =
systemSettingsImportConfigurationDefinition.configurationDefinition.de
faultTemplate.createConfiguration()

rhqadmin@localhost:7080$ var systemSettingsImportConfiguration = new
ImportConfiguration(systemSettingsImportConfigurationDefinition.synchr
onizerClassName, configurationObject)
```

3. Change the settings.

   For example, this edits the server synchronizer so that it imports only the database settings for storing monitoring data.

   ```
   rhqadmin@localhost:7080$
   configurationObject.getSimple('propertiesToImport').setValue('CAM_DATA
   _PURGE_1H, CAM_DATA_PURGE_6H, CAM_DATA_PURGE_1D,
   CAM_DATA_MAINTENANCE')
   ```

   For the metrics template synchronizer, define which metric schedules to import per resource type, based on a properties list or a properties map. For example:

   ```
   rhqadmin@localhost:7080$
   configurationObject.getSimple('updateAllSchedules').setBooleanValue(tr
   ue)
   rhqadmin@localhost:7080$ var updateList = new
   PropertyList('metricUpdateOverrides')
   rhqadmin@localhost:7080$ var update = new
   PropertyMap('metricUpdateOverride')
   rhqadmin@localhost:7080$ update.put(new PropertySimple('metricName',
   'MCBean|ServerInfo|*|freeMemory'))
   rhqadmin@localhost:7080$ update.put(new
   PropertySimple('resourceTypeName', 'JBossAS Server'))
   rhqadmin@localhost:7080$ update.put(new
   PropertySimple('resourceTypePlugin', 'JBossAS5'))
   rhqadmin@localhost:7080$ update.put(new
   PropertySimple('updateSchedules', 'true'))

   rhqadmin@localhost:7080$ updateList.add(update)

   rhqadmin@localhost:7080$ configurationObject.put(updateList)
   ```

After changing the synchronizer settings, then import the configuration.

```
rhqadmin@localhost:7080$ var configsToImport = new java.util.ArrayList()
rhqadmin@localhost:7080$
configsToImport.add(systemSettingsImportConfiguration);
rhqadmin@localhost:7080$
configsToImport.add(metricTemplatesImportConfiguration);
rhqadmin@localhost:7080$ SynchronizationManager.importAllSubsystems(ex,
configToImport);
```