JBoss Operations Network 3.0
Running JBoss ON Command-Line Scripts

---

using the JBoss ON CLI and remote API
Edition 3.0.1

Ella Deon Lackey

JBoss Operations Network 3.0 Running JBoss ON Command-Line Scripts

using the JBoss ON CLI and remote API
Edition 3.0.1

Ella Deon Lackey
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

1. Document Information **2**
1.1. Document History 2

2. Using the JBoss ON CLI to Script Tasks **2**
2.1. About the JBoss ON CLI 2
2.2. More Java Resources 3

3. Installing the JBoss ON Command-Line Tool **4**

4. Running the JBoss ON CLI **4**
4.1. JBoss ON CLI Options 6
4.2. JBoss ON CLI Commands 6
4.3. Available Implicit Variables in the JBoss ON API 8
4.4. Passing Script Arguments in the JBoss ON CLI 11
4.5. Configuring Criteria-Based Searching 13
4.6. Displaying Output 16
4.7. Simple CLI Examples 18
4.8. Using Resource Proxies 19

5. Simple Example: Scripts to Manage Inventory **25**
5.1. Automatically Import New Resources: autoimport.js 25
5.2. Simple Inventory Count: inventoryCount.js 26
5.3. Uninventory a Resource After an Alert: uninventory.js 27
5.4. JNDI Lookups for a JBoss AS 5 Server After an Alert: jndi.js 27

6. Example: Scripting Resource Deployments **27**
6.1. Scripting JBoss AS 4 Deployments 28
6.2. Scripting JBoss AS 5 Deployments 32

7. Example: Managing Grouped Servers **37**
7.1. The Plan for the Scripts 37
7.2. Creating the Wrapper Script and .conf File 38
7.3. Defining Arguments and Other Parameters for the CLI Scripts 40
7.4. Creating a Group: group.js 41
7.5. Adding Resources to a Group: addMember.js 41
7.6. Getting Inventory and Status Information: status.js 43
7.7. Starting, Stopping, and Restarting the Server: restart.js 44
7.8. Deploying Applications to the Group Members: deploy.js 45
7.9. Scheduling an Operation: ops.js 45
7.10. Gathering Metric Data of Managed Servers: metrics.js 46

8. Example: Writing a Custom Java Client **47**
8.1. Getting the API 48
8.2. Example Custom Java Client 48

9. Reference: Methods Specific to the JBoss ON CLI **56**
9.1. Methods Available to the CLI and Server Scripts 56
9.2. Methods Available to Proxy Resources 62

1. Document Information

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

1.1. Document History

Revision 3.0.1-5	**2013-10-31**	**Rüdiger Landmann**

Rebuild with publican 4.0.0

Revision 3.0.1-0	**March 18, 2012**	**Ella Deon Lackey**

Fixing example scripts and rewriting examples for JBoss Operations Network 3.0.1.

Revision 3.0-1	**January 26, 2012**	**Ella Deon Lackey**

Fixing a typo in the echo_args.js example script, per Bugzilla 784887.
Fixing a typo in the CLI login example, per Bugzilla 784703.
Updating API call for createPackageBackedResource method to include timeout parameter, per Bugzilla 770011.
Breaking out separate JBoss AS 4 and JBoss AS 5 examples for scripting an EAR deployment to a JBoss group, per Bugzilla 772366.

Revision 3.0-0	**December 5, 2011**	**Ella Deon Lackey**

Initial release of JBoss ON 3.0.

2. Using the JBoss ON CLI to Script Tasks

A large subset of JBoss Operations Network functionality is exposed through its remote APIs. These APIs allow clients to access the server functionality — alerting, monitoring, managing inventory and resources, even agents. JBoss ON has several different ways that clients can leverage the remote API. Two of them are supplied with JBoss ON automatically: the JBoss ON web UI and the JBoss ON Java CLI. Custom clients can be written using the remote API for customers to create their own management interfaces, like desktop clients, provisioning automation, or alert reporting integration.

Building custom user interfaces is outside the scope of this guide. For reference, there is an example JEE client in Section 8, “Example: Writing a Custom Java Client” and references to the JBoss ON public remote API, available at https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Operations_Network/3.1/html/API/ch01.html.

2.1. About the JBoss ON CLI

The backend logic that defines the JBoss Operations Network subsystems, functionality, and interactions is contained in the Enterprise JavaBeans for the server. These EJBs expose the relevant JBoss ON APIs to different interfaces. JBoss ON supports two kinds of interfaces:

- Web services, like the JBoss ON UI, using SOAP and leveraging the JBoss Remoting framework for communication
- Java clients using (obviously) Java and calling the remote JBoss ON APIs

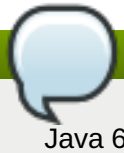
JBoss ON already includes a client of each type. The JBoss ON server GUI uses the web services and remoting framework to connect to the server. The additional JBoss ON CLI which can be downloaded and installed locally uses Java and the remote API to support scripting and command line services.

Custom services can be built for web UIs using SOAP or for Java Enterprise (JEE) clients written in any JVM-compatible language, including Java, Scala, and Groovy.

The default JBoss ON CLI is a Java-based scripting interface. This contrasts with the JBoss ON web UI, which provides a simple, visual way to manage JBoss ON.

The CLI is a standalone Java application that uses the Java Scripting API. The CLI better integrates JBoss ON into a network environment because administrators can interact with JBoss ON programmatically.

The JBoss ON CLI has a JavaScript-style scripting environment by using the Rhino engine in Java 6. By using the Remoting services in the JBoss ON communications system, the CLI can be used for remote scripting services, not just local operations.



Note

Java 6 ships with the Rhino JavaScript engine, so JavaScript is the supported scripting language in the CLI.

The JBoss Operations Network CLI itself is a Java shell the allows administrators to connect to the JBoss ON server over the command line. Essentially, the CLI is a script execution engine. It treats the JBoss ON API as if it was written in a scripting language, which makes it more convenient to manage the JBoss ON server.

The JBoss ON CLI is opened through a script, **`rhq-cli.sh|bat`**. This script emulates a shell, accepting basic commands and allowing administrators to navigate the server with shell tools like autocomplete.

The JBoss ON CLI uses the remote API. This can be used to create both Java and web-based clients by leveraging the JBoss Remoting framework. This is available at the remote API javadoc.

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)
```

2.2. More Java Resources

For additional information on using a Java client like the JBoss ON CLI, check out these resources:

- The JBoss ON API
- Java Scripting Programmer's Guide
- Rhino: JavaScript for Java

- The Java Persistence Query Language (JEE 5 Tutorial)

3. Installing the JBoss ON Command-Line Tool

The JBoss ON server contains packages called the Remote Client, which contain the JBoss ON CLI packages, **rhq-client-***3.0***.zip**.

Note

Only the corresponding version of the CLI can be used to manage the JBoss ON server. Other versions are not compatible.

To install the CLI:

1. Open the JBoss ON GUI.

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

2. Click the **Administration** link in the main menu.
3. Select the **Downloads** menu item.
4. Scroll to the **Command Line Client Download** section, and click **Download Client Installer**.
5. Save the **.zip** file into the directory where the CLI should be installed.
6. Unzip the packages.

   ```
   unzip rhq-client-version.zip
   ```

4. Running the JBoss ON CLI

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.

The CLI script, **rhq-cli.sh|bat**, is run directly from its *cli-install-dir***bin** directory and used to log into the server. There are two files associated with launching the JBoss ON CLI:

- A script
- A file of environment variables

These are listed in Table 1, “JBoss ON CLI Files”.

Table 1. JBoss ON CLI Files

Operating System	CLI Script	Environment Variables File
Red Hat Enterprise Linux	rhq-cli.sh	rhq-cli-env.sh
Microsoft Windows	rhq-cli.bat	rhq-cli-env.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 JVM that isn't the default. If any variables need to be edited, *always* set them in this file. 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 commands
```

It is possible to launch the CLI script without any arguments. This opens the CLI client *without* connecting to the server.

```
cliRoot/rhq-remoting-cli-3.0.0.GA1/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.

```
rhq-cli -u rhqadmin -p rhqadmin
```

The CLI provides two modes of operation: interactive and non-interactive. Interactive mode executes an individual statement. In non-interactive mode, multiple commands can be executed in sequence, in the form of a script. Non-interactive mode provides the capability to automate tasks such as collecting metrics on managed resources or executing a scheduled operation. (Interactive mode provides a simple environment for prototyping, testing, learning and discovering features of the CLI, and these examples are given in interactive mode, though they are also available in non-interactive mode.)

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, you must use the Java method.

For more information about integration with the underlying Java platform, look at the Rhino documentation at http://www.mozilla.org/rhino/doc.html.

After logging in, any commands (covered in Section 4.2, “JBoss ON CLI Commands”) can be passed to the server.

4.1. JBoss ON CLI Options

Both **rhq-cli.bat** and **rhq-cli.sh** scripts accept the options listed in Table 2, “Command-Line Options”.

Table 2. 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.
-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.
-v	--version	Displays CLI and JBoss ON server version information once connected to the 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.
	--args-style	Indicates the style or format of arguments passed to the script. See the Executing Scripts below for additional information.

4.2. JBoss ON 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.

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.

4.2.1. login

Logs into a JBoss ON server with the specified username and password. The host name (or IP address) and port can be specified. The host name defaults to localhost and the port defaults to 7080. The transport argument is optional. It 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 argument. 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 in 443.

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

4.2.2. logout

Logs off of the JBoss ON server but does not exit from the CLI.

```
logout
```

4.2.3. quit

Exits the CLI.

```
quit
```

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

4.2.4. exec

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

```
exec statement | [-s indexed|named] -f /path/to/file [args]
```

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	Indicates the style or format of arguments passed to the script. It must have a value of either **`indexed`** or **`named`**.

Example 1. Executing a Single Statement

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

Example 2. Executing a Multi-Line Statement

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

localhost:7080(rhqadmin)>
```

Example 3. Executing a Named Script without Arguments

```
localhost:7080(rhqadmin)> exec -f myscript.js
```

Example 4. Executing a Named Script with Arguments

```
localhost:7080(rhqadmin)> exec -f myscript.js 1 2 3
```

Example 5. Executing a Named Script with Named Arguments

```
localhost:7080(rhqadmin)> exec --style=named -f myscript.js x=1 y=2 y=3
```

4.2.5. record

Records user input commands to a file.

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

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.

4.3. Available Implicit Variables in the JBoss ON API

The JBoss ON API makes a number of variables available to interfaces. These variables are bound to the CLI script context.

The public JBoss ON API can be viewed at https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Operations_Network/3.1/html/API/index.html.

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 from various packages that likely to be used on a routine basis. To simplify using the JBoss ON CLI and remote clients, certain classes are also imported. 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 aspects of JBoss ON.

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 3, “Variables Available by Default to the JBoss ON CLI”. Methods and other information about these variables are in Section 9, “Reference: Methods Specific to the JBoss ON CLI”.

Table 3. 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.enterprise.client.utility.ResourceClientProxy.Factory		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
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

4.4. Passing Script Arguments in the JBoss ON CLI

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`**:

Example 6. Handling Script Arguments

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

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

Important

The **args** variable is only available when executing a script in non-interactive mode or with **exec -f**.

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

```
rhqadmin@localhost:7080$ exec -f echo_args.js --args-style=named x=1, y=2
```

Example 7. echo_args.js

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

This produces the following output:

Example 8. echo_args.js

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

x = 1
y = 2
```

Be aware of the following:

- You have to explicitly specify that you are using named arguments via the **--args-style** option

- The values of the named arguments are still accessible via the implicit **`args array`**
- The named arguments, **`x`** and **`y`**, are bound into the script context as variables

The CLI is built using the Java Scripting API that was introduced in Java 6. The majority of commands and scripts used with the JBoss ON CLI are executed by the underlying script engine. Built-in commands, however, are native Java code and are not executed by the underlying script engine. This is similar to other interpreters like Python where some modules are implemented in C and built into the interpreter. This distinction is important because built-in commands cannot be processed by the script engine. Objects, however, that provide hooks into the built-in commands, are exposed to the scripting environment.

4.5. Configuring Criteria-Based Searching

All of the managers define operations for retrieving data. Most of the managers define *criteria-based* operations for data retrieval. The criteria API provides a flexible framework for fine-tuned query operations.

The criteria classes reside in the *org.rhq.core.domain.criteria* package. Criteria-based searches can be implemented in several different ways.

4.5.1. Setting Basic Search Criteria

The simplest criteria is to define results based on what they are, such as resource type.

```
rhqadmin@localhost:7080$ var criteria = new ResourceCriteria()
rhqadmin@localhost:7080$ var resources =
ResourceManager.findResourcesByCriteria(criteria)
```

It isn't necessary to import the **`ResourceCriteria`** class because the **`org.rhq.core.domain.criteria`** package is automatically imported.

The method, **`findResourcesByCriteria`** follows the naming format of all criteria-based query operations, **`findXXXByCriteria`**. This basic criteria search is translated into the following JPA-QL query:

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

This fetches all committed resources in the inventory.

4.5.2. Using Sorting

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

```
rhqadmin@localhost:7080$ criteria.addSortPluginName(PageOrdering.ASC)
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
```

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

4.5.3. Using Filtering

Adding additional matching criteria, like resource name in this example, further narrows the search results.

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

To add filtering to any criteria, use methods of the form **`addFilterXXX()`**. The resulting JPA-QL query will appear as follows:

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

This code is all that is required to retrieve all JBoss servers in your inventory. You can further refine your criteria to find JBoss servers that have been registered by a particular agent:

```
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 '\\' )
```

4.5.4. Fetching Associations

After retrieving the resources, it is possible to view child resources. For example:

```
rhqadmin@localhost:7080$ criteria.addFilterResourceTypeName('JBossAS
Server')
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')
```

This code will print the string *no child resources*, even if the JBoss server has 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, specify the criteria for the fetch order.

```
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)
```

As with the call **`criteria.fetchChildResources(true)`**, all criteria methods that specify that a particular lazy association should be fetched are of the form, **`fetchXXX()`**.

```
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
222 AlertTemplateManagerBean                                        EJB3
Session Bean
223 RHQ Server Group Definition / DynaGroups Subsystem              RHQ Server
Group Definition / DynaGrou
222 DiscoveryTestBean                                               EJB3
Session Bean
222 PerspectiveManagerBean                                          EJB3
Session Bean
222 ResourceAvailabilityManagerBean                                 EJB3
Session Bean
222 AlertDampeningManagerBean                                       EJB3
Session Bean
218 localhost.localdomain  Embedded JBossWeb Server 2.0. 2.0.1. Embedded
Tomcat Server
222 ResourceGroupManagerBean                                        EJB3
Session Bean
222 FailoverListManagerBean                                         EJB3
Session Bean
222 ResourceFactoryManagerBean                                      EJB3
Session Bean
222 AccessBean                                                      EJB3
Session Bean
222 MeasurementTestBean                                             EJB3
Session Bean
223 wstools.sh                                                      Script
222 EventManagerBean                                                EJB3
Session Bean
```

```
222 ContentSourceManagerBean                                    EJB3
Session Bean
223 RHQ Server Alerts Engine Subsystem                          RHQ Server
Alerts Engine Subsystem
222 AlertConditionManagerBean                                   EJB3
Session Bean
222 ResourceMetadataManagerBean                                 EJB3
Session Bean
222 ResourceManagerBean                                         EJB3
Session Bean
222 GroupDefinitionExpressionBuilderManagerBean                 EJB3
Session Bean
222 MeasurementViewManagerBean                                  EJB3
Session Bean
218 JmsXA Connection Factory
ConnectionFactory
222 ResourceTypeManagerBean                                     EJB3
Session Bean
223 JBoss Cache subsystem                               1.0
JBossCacheSubsystem
218 NoTxRHQDS Datasource                                        Datasource
222 DataAccessBean                                              EJB3
Session Bean
222 AlertConditionCacheManagerBean                              EJB3
Session Bean
222 MeasurementProblemManagerBean                               EJB3
Session Bean
222 ServerManagerBean                                           EJB3
Session Bean
222 OperationHistorySubsystemManagerBean                        EJB3
Session Bean
222 ClusterManagerBean                                          EJB3
Session Bean
222 run.sh                                                      Script
...
```

The output will vary depending on what you have inventoried. These are the child resources of the JBoss ON server we have used in these examples. 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 '\\' )
```

4.6. Displaying Output

4.6.1. TabularWriter

`TabularWriter` provides for tabular-formatted output. In addition to formatting, it handles converting objects, particularly domain objects in the packages under *org.rhq.core.domain*, into a format suitable for display in the interactive console. Let's look at an example to illustrate the utility of **`TabularWriter`**:

```
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
```

The **`TabularWriter`** instance that is bound in the script context under the name pretty is implicitly invoked to display the results of **`ResourceManager.findResourcesByCriteria`**. The returned resources are displayed in very readable, tabular format. Now let's look at the output if we do not use **`TabularWriter`**.

```
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],
Resource[id=11375, type=service-alpha, key=service-alpha-0, name=service-
alpha-0, version=1.0],
Resource[id=11376, type=service-alpha, key=service-alpha-4, name=service-
alpha-4, version=1.0],
Resource[id=11377, type=service-alpha, key=service-alpha-2, name=service-
alpha-2, version=1.0],
Resource[id=11378, type=service-alpha, key=service-alpha-3, name=service-
alpha-3, version=1.0],
Resource[id=11379, type=service-alpha, key=service-alpha-5, name=service-
alpha-5, version=1.0],
Resource[id=11380, type=service-alpha, key=service-alpha-9, name=service-
alpha-9, version=1.0],
Resource[id=11381, type=service-alpha, key=service-alpha-6, name=service-
alpha-6, version=1.0],
Resource[id=11382, type=service-alpha, key=service-alpha-7, name=service-
alpha-7, version=1.0]]
```

For display purposes, this output is not very usable. Let's look at one more example in which we display a single entity.

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

The formatting is different when displaying a single object.

Only a subset of the properties in the **`Resource`** class are displayed. **`TabularWriter`** determines the properties to display through the **`Summary`** annotation. If a field of an entity has the **`@Summary`** annotation, **`TabularWriter`** will include it in the output.

4.6.2. Exporter

An implicit script variable that can assist with writing output to a file is **`exporter`**. It uses **`TabularWriter`**; however, in addition to writing plain text in a tabular format, it also supports CSV-formatting. First, we will look at an example of exporting to a file as plain text:

```
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.

Next, export the results to a CSV file:

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

4.7. Simple CLI Examples

Example 9. Logging in to a Specified Server

You will be connected to the CLI and logged in with the specified credentials on the JBoss ON server running on localhost.

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

You will be connected to the CLI and logged into the JBoss ON server on 192.168.1.100 that is listening on port 70443. Because the port number ends with 443, the CLI will attempt to communicate with the JBoss ON server over SSL using the sslservlet transport strategy.

Example 10. Prompting for a Password

```
rhq-cli -u rhqadmin -P
```

You will be connected to the CLI and prompted for a password.

Example 11. Passing Variables to the Server

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

This connects you to the CLI, logs you into the JBoss ON server running on localhost, executes the command in quotes, and redirects the output to the file **`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. The next portion is **pretty.print(...)**. An implicit object made available to commands and scripts by the CLI, **pretty** is useful for outputting objects in a readable, tabular format, designed with enhanced capabilities for domain objects. This single command provides a nicely formatted, text-based report of the resource types in the inventory.

Example 12. Running a Script in the JBoss ON CLI

```
cliRoot/rhq-remoting-cli-3.0.0.GA1/bin/rhq-cli.sh -f my_script.js
```

This connects you to the CLI and executes the script file, **my_script.js**. The CLI will terminate immediately after the script has finished executing.

4.8. 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: sun-x8420-01.rhts.eng.bos.redhat.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: sun-x8420-
01.rhts.eng.bos.redhat.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 13. Creating 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 remoteAPI tasks, like getting monitoring information.

The proxy API for common resource types is listed in Section 9.2, “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 14. 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 tyan-gt24-04.rhts.eng.bos.redhat.com RHQ Server (JBossAS
Server::JBos                                          sAS)
[10002] RHQ Agent (RHQ Agent::RHQAgent)
```

Example 15. 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             tyan-gt24-04.rhts.eng.bos.redhat.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 16. 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 17. 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 18. 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")
```

5. Simple Example: Scripts to Manage Inventory

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.

Similar behavior can be emulated in JBoss ON 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 instances are functionally managed servers, while JBoss ON itself acts as the domain controller.

5.1. 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.

Uncommited (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;
}
```

5.2. 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
```

5.3. 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.

5.4. JNDI Lookups for a JBoss AS 5 Server After an Alert: jndi.js

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 "Setting up Monitoring, Alerts, and Operations".

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

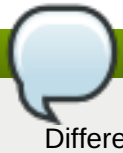
```
//This test requires a remote JBoss AS 5 server running with JNDI directory
remotely accessible using JNP (without authz)
//This script assumes that there is a bound object called "jmx" in the
directory (which it should be)
var jbossHost = 'localhost';
var jbossJnpPort = 1299;

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: Scripting Resource Deployments

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.



Note

Different resource types have configuration properties and operations available, which can impact CLI scripts written to manage those resources. This can be true even for different versions of the same application — which is the case with JBoss AS 4 and JBoss AS 5 resources.

JBoss AS 4 and JBoss AS 5 have different APIs for operations like shutting down instances. In this example, separate scripts are shown for the two resource types. It is also possible to create a CLI script which identifies the resource type and then calls the appropriate API.

6.1. Scripting JBoss AS 4 Deployments

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;
}
```

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 4.4, “Passing Script Arguments in the JBoss ON CLI”).

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 the call 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 4 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 != "JBossAS 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 JBossAS Server resource types as top level resources. Now **server** contains the JBossAS 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 );
}
```

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 4)

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, "JBossAS" );
if( appType == null ) {
    println("  Could not find application type. Exit.");
    usage();
}
```

Then get the package type of the application.

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

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 4 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("deployDirectory", "deploy"));
deployConfig.put( new PropertySimple("deployZipped", "true"));
deployConfig.put( new PropertySimple("createBackup", "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 4 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.

6.2. Scripting JBoss AS 5 Deployments

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;
}
```

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 4.4, “Passing Script Arguments in the JBoss ON CLI”).

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 the call 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 );
}
```

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.

7. Example: Managing Grouped Servers

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.

Similar behavior can be emulated in JBoss ON 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 instances are functionally managed servers, while JBoss ON itself acts as the domain controller.

7.1. 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.

7.2. 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 acessible 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
#   ops      : Runs an operation on all AS instances specified by group name
#   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. For example, for the **deploy** command to deploy content to the EAP instances:

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


case "$1" in
'deploy')
 doDeploy $*
 ;;
*)
        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
##
ON_CLI_HOME=cliRoot/rhq-remoting-cli-3.0.0.GA1
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
```

7.3. 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 a really good idea to also 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);
  }
}
```

7.4. 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");
```

7.5. 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!
```

7.6. 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();
criteria.addFilterExplicitGroupIds(group.id);

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
```

7.7. Starting, Stopping, and Restarting the Server: restart.js

Set up the script with the usage information and the group search, as in Section 7.6, “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();
criteria.addFilterExplicitGroupIds(group.id);

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 );
     }
}
```

7.8. 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 6, "Example: Scripting Resource Deployments".

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 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;
}
```

7.9. Scheduling an Operation: ops.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("");
}
```

7.10. 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();
criteria.addFilterExplicitGroupIds(group.id);
```

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]]]]
```

8. Example: Writing a Custom Java Client

As alluded to in Section 2, “Using the JBoss ON CLI to Script Tasks”, the clients in JBoss ON use either the JBoss Remoting framework or the JBoss ON remote APIs to access server functionality. The JBoss ON CLI is essentially a Java skin over the remote API. Any application written in Java or a JVM-compatible language can access the JBoss ON remote API.

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)
```

The power of the JBoss ON Java interface is the programmatic way that it handles remote clients. The Java interface creates a client object and works with it as if that client object were in the command line:

```
{
 Client joprClient = new Client(...)
 jopClient.getResourceManager().findResources(...)
 ...
}
```

This is the recommended approach for a programmatic remote client.

8.1. Getting the API

The JBoss ON remote API is downloaded and installed with the JBoss ON CLI package, as described in Section 3, “Installing the JBoss ON Command-Line Tool”. The CLI usage examples in this guide can be used as general guidelines for calling the API in the Java code.

The javadocs for the JBoss ON remote API can be viewed at https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Operations_Network/3.1/html/API/ch01.html.

8.2. Example Custom Java Client

This example creates an LDAP integration for LDAP group-based authorization for JBoss ON. The sample Java class pulls in the authorization and search classes from the JBoss ON API, and then the script starts a simple synchronization service that maps the LDAP groups and users to the JBoss ON roles and users.

Note

LDAP-based group authorization is already configured in JBoss ON. This client is simply used as an example to show how a remote Java client can interact with the JBoss ON server.

8.2.1. Sample Java Class Using the JBoss ON API

This Java class uses the JBoss ON API for users, permissions, roles, and searching and sorting resource entries. The class then sets up a mapping between the LDAP database and the JBoss ON database, so that the user and role information in each is synchronized.

```
package org.rhq.sample.client.java.ldap;

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

import org.rhq.core.domain.auth.Subject;
import org.rhq.core.domain.authz.Permission;
import org.rhq.core.domain.authz.Role;
import org.rhq.core.domain.criteria.ResourceCriteria;
import org.rhq.core.domain.criteria.ResourceGroupCriteria;
import org.rhq.core.domain.criteria.RoleCriteria;
import org.rhq.core.domain.resource.Resource;
import org.rhq.core.domain.resource.group.ResourceGroup;
import org.rhq.core.domain.util.PageList;
import org.rhq.enterprise.client.RemoteClient;
import org.rhq.enterprise.server.auth.SubjectManagerRemote;
import org.rhq.enterprise.server.authz.RoleManagerRemote;
import org.rhq.enterprise.server.resource.ResourceManagerRemote;
import org.rhq.enterprise.server.resource.group.ResourceGroupManagerRemote;

/**
 * This sample program utilizes the RHQ Remote API via a Java Client.
 *
 * The RHQ CLI is the preferred remote client approach for script-based
clients.  Programmatic Java clients
 * can utilize the Remote API via the same mechanism used by the CLI, making
use of ClientMain object, as
 * done in this sample.  This is the recommended mechanism although a remote
Java client could als use the
 * remote API exposed as WebServices.
 *
 * @author Jay Shaughnessy
 */
public class SampleLdapClientMain {
    // A remote session always starts with a login, define default
user/password/server/port
    private static String username = "rhqadmin";
    private static String password = "rhqadmin";
    private static String host = "localhost";
    private static int port = 7080;

    /**
     * This is a standalone remote client but calls to the remote API could
be embedded into another application.
     */
    public static void main(String[] args) {
        if (args.length > 0) {
            if ((args.length != 2) && (args.length != 4)) {
                System.out
                    .println("\nUsage: SampleLdapClientMain [ [ username
```

```
password ] | [username password host port] ]");
                System.out.println("\n\nDefault credentials:
rhqadmin/rhqadmin");
                System.out.println("\n\nDefault host: determined from
wsconsume of WSDL");
                return;
            } else {
                username = args[0];
                password = args[1];

                if (args.length == 4) {
                    host = args[2];
                    port = Integer.valueOf(args[3]);
                }
            }
        }

        LdapClient ldapClient = null;

        try {
            ldapClient = new LdapClient();
            ldapClient.synchLdapJbasManagers();

        } catch (Throwable t) {
            System.out.println("Error: " + t);
            t.printStackTrace();
        } finally {
            if (null != ldapClient) {
                // clean up the session by logging out from the RHQ server
                ldapClient.logout();
            }
        }
    }

    /**
     * The LdapClient interacts with the RHQ Server to help synchronize a
(fake) LDAP server with RHQ.
     */
    public static class LdapClient {
        // group containing all jbas resources
        private static final String JBAS_GROUP = "jbas-resource-group";

        // role for jbas managers
        private static final String JBAS_MANAGER_ROLE = "jbas-manager-role";

        // the users that should be assigned the JBAS_MANAGER_ROLE
        private static final List<String> JBAS_MANAGERS = new
ArrayList<String>();

        // the prmissions that should be assigned the JBAS_MANAGER_ROLE
        private static final Set<Permission> JBAS_MANAGER_PERMISSIONS = new
HashSet<Permission>();

        // jbas AS Server resource type (note, this picks up AS4 and AS5
resources as they share the same type name)
        private static final String JBAS_SERVER_NAME = "JBossAS Server";
```

```
        /* The Remote API offers different remote "managers" roughly broken
down by subsystem/function
         * Below are the managers needed by this client, there are several
others that offer
         * interfaces into areas such as operations, alerting, content, etc.
See the API.
         */
        private ResourceGroupManagerRemote resourceGroupManager;
        private ResourceManagerRemote resourceManager;
        private RoleManagerRemote roleManager;
        private SubjectManagerRemote subjectManager;

        /* This represents the RHQ user that is logged in and making the
remote calls. This user must
         * already exist.  For the work being done here the user must also
have SECURITY_MANAGER permissions.
         */
        private Subject subject;

        /* This is the object through which we access the remote API */
        private RemoteClient remoteClient;

        static {
            // add some fake users since we're not actually hooked into an
ldap server
            JBAS_MANAGERS.add("mgr-1");
            JBAS_MANAGERS.add("mgr-2");

            // add some permissions since we're not actually hooked into an
ldap server
            JBAS_MANAGER_PERMISSIONS.addAll(Permission.RESOURCE_ALL);
        }

        public LdapClient() throws Exception {
            this.remoteClient = new RemoteClient(null, host, port);
            this.subject = remoteClient.login(username, password);

            this.resourceGroupManager =
this.remoteClient.getResourceGroupManagerRemote();
            this.resourceManager =
this.remoteClient.getResourceManagerRemote();
            this.roleManager = this.remoteClient.getRoleManagerRemote();
            this.subjectManager =
this.remoteClient.getSubjectManagerRemote();
        }

        /*
         * This method simulates a sync between an Ldap server that has
defined a group of JBAS managers
         * and wants to associate them with a role allowing jbas management.
Meaning, a role that
         * has the proper permissions and is associated with the jbas
resources.
         */
        private void synchLdapJbasManagers() throws Exception {
```

```
            // create the jbas manager role if necessary
            // use a criteria search with a name filter to look for the role
            RoleCriteria roleCriteria = new RoleCriteria();
            roleCriteria.setFilterName(JBAS_MANAGER_ROLE);
            PageList<Role> jbasManagerRoles =
roleManager.findRolesByCriteria(subject, roleCriteria);
            Role jbasManagerRole;
            if (1 == jbasManagerRoles.size()) {
                jbasManagerRole = jbasManagerRoles.get(0);
            } else {
                // if it doesn't exist, create it
                jbasManagerRole = new Role(JBAS_MANAGER_ROLE);
                jbasManagerRole = roleManager.createRole(subject,
jbasManagerRole);
            }
            // ensure the proper permissions are granted to the role by
using an update
            jbasManagerRole.setPermissions(JBAS_MANAGER_PERMISSIONS);
            roleManager.updateRole(subject, jbasManagerRole);

            // create, populate and associate the jbas group if necessary
            ResourceGroupCriteria resourceGroupCriteria = new
ResourceGroupCriteria();
            resourceGroupCriteria.addFilterName(JBAS_GROUP);
            PageList<ResourceGroup> jbasGroups =
resourceGroupManager.findResourceGroupsByCriteria(subject,
                resourceGroupCriteria);
            ResourceGroup jbasGroup;
            if (1 == jbasGroups.size()) {
                jbasGroup = jbasGroups.get(0);
            } else {
                jbasGroup = new ResourceGroup(JBAS_GROUP);
                jbasGroup =
resourceGroupManager.createResourceGroup(subject, jbasGroup);
                // Ensure the group is recursive to make all the children
available.
                // In this case a specific method is available, so a
general update call is not needed.
                resourceGroupManager.setRecursive(subject,
jbasGroup.getId(), true);
            }

            // Now find all of the JBAS server resources by adding a
criteria filter on resource type name
            ResourceCriteria resourceCriteria = new ResourceCriteria();
            resourceCriteria.addFilterResourceTypeName(JBAS_SERVER_NAME);
            PageList<Resource> jbasServers =
resourceManager.findResourcesByCriteria(subject, resourceCriteria);
            if (!jbasServers.isEmpty()) {
                int[] jbasServerIds = new int[jbasServers.size()];
                int i = 0;
                for (Resource jbasServer : jbasServers) {
                    jbasServerIds[i++] = jbasServer.getId();
                }
```

```
                // ..and add them to the group which will be associated
with the manager role
                resourceGroupManager.addResourcesToGroup(subject,
jbasGroup.getId(), jbasServerIds);
            }

            // Now, associate the mixed group of Jbas servers to the manager
role
            roleManager.addResourceGroupsToRole(subject,
jbasManagerRole.getId(), new int[] { jbasGroup.getId() });

            // sync managers with the role
            // 1. remove obsolete managers
            roleCriteria = new RoleCriteria();
            roleCriteria.setFilterId(jbasManagerRole.getId());
            // add a fetch criteria to the criteria object to get the
optionally returned subjects for the role.
            roleCriteria.setFetchSubjects(true);
            jbasManagerRole = roleManager.findRolesByCriteria(subject,
roleCriteria).get(0);
            Set<Subject> subjects = jbasManagerRole.getSubjects();
            if ((null != subjects) && !subjects.isEmpty()) {
                for (Subject subject : subjects) {
                    if (!JBAS_MANAGERS.contains(subject.getName())) {
                        roleManager.removeSubjectsFromRole(subject,
jbasManagerRole.getId(), new int[] { subject
                                .getId() });
                    }
                }
            }

            // 2. add new managers, create subjects for the managers, if
necessary
            Subject jbasManagerSubject;
            for (String jbasManager : JBAS_MANAGERS) {
                jbasManagerSubject =
subjectManager.getSubjectByName(jbasManager);
                // add the required fields for a subject, note that we skip
credentials since this is
                // simulating ldap
                if (null == jbasManagerSubject) {
                    jbasManagerSubject = new Subject();
                    jbasManagerSubject.setName(jbasManager);

jbasManagerSubject.setEmailAddress("jbas.manager@sample.com");
                    jbasManagerSubject.setFactive(true);
                    jbasManagerSubject.setFsystem(false);
                    jbasManagerSubject =
subjectManager.createSubject(subject, jbasManagerSubject);
                }

                // Finally, make sure my current set of jbas managers is
associated with the manager role.
                roleManager.addSubjectsToRole(subject,
jbasManagerRole.getId(),
                    new int[] { jbasManagerSubject.getId() });
```

```
            }
        }

        public void logout() {
            if ((null != subjectManager) && (null != subject)) {
                try {
                    subjectManager.logout(subject);
                } catch (Exception e) {
                    // just suppress the exception, nothing else we can do
                }
            }
        }
    }
}
```

8.2.2. Sample LDAP Script

The sample **.bat** script invokes the custom Java class.

```
@echo off

rem
=============================================================================
rem RHQ Remote Client LDAP Example Startup Script
rem
rem The following variables must be set
rem
rem RHQ_CLIENT_HOME    The home directory of the RHQ Client Installation.
The
rem                    RHQ Client can be downloaded from the RHQ GUI under
rem                    the Administration->Downloads menu.
rem
=============================================================================

rem -----------------------------------------------------------------------
--
rem Set Environment Variables
rem -----------------------------------------------------------------------
--
set RHQ_CLIENT_HOME=*MUST BE SET*


rem -----------------------------------------------------------------------
--
rem Prepare the classpath
rem Add all jar files supplied by the RHQ remote client install
rem -----------------------------------------------------------------------
--

set CLASSPATH=.
call :append_classpath "%RHQ_CLIENT_HOME%\conf"
for /R "%RHQ_CLIENT_HOME%\lib" %%G in ("*.jar") do (
   call :append_classpath "%%G"
)
```

```
rem -------------------------------------------------------------------
--
rem Prepare the VM command line options to be passed in
rem -------------------------------------------------------------------
--

if not defined RHQ_CLIENT_JAVA_OPTS (
   set RHQ_CLIENT_JAVA_OPTS=-Xms64m -Xmx128m -Djava.net.preferIPv4Stack=true
)

rem -------------------------------------------------------------------
--
rem Uncomment For debugging on port 9999
rem -------------------------------------------------------------------
--

rem set RHQ_CLIENT_ADDITIONAL_JAVA_OPTS=-
agentlib:jdwp=transport=dt_socket,address=9999,server=y,suspend=y


rem -------------------------------------------------------------------
--
rem Execute the VM which starts the CLIENT
rem -------------------------------------------------------------------
--

set CMD="%JAVA_HOME%\bin\java.exe" %RHQ_CLIENT_JAVA_OPTS%
%RHQ_CLIENT_ADDITIONAL_JAVA_OPTS% -cp "%CLASSPATH%"
org.rhq.sample.client.java.ldap.SampleLdapClientMain
%RHQ_CLIENT_CMDLINE_OPTS% %*

cmd.exe /S /C "%CMD%"

goto :done

rem -------------------------------------------------------------------
--
rem CALL subroutine that appends the first argument to CLASSPATH
rem -------------------------------------------------------------------
--

:append_classpath
set _entry=%1
if not defined CLASSPATH (
   set CLASSPATH=%_entry:"=%
) else (
   set CLASSPATH=%CLASSPATH%;%_entry:"=%
)
goto :eof

rem -------------------------------------------------------------------
--
rem CALL subroutine that exits this script normally
rem -------------------------------------------------------------------
--
```

```
:done

endlocal

exit /B 0
```

9. Reference: 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.

9.1. Methods Available to the CLI and Server Scripts

9.1.1. Assert

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

Method	Signature

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)
Assert.assertEqualsNoOrder	assertEqualsNoOrder(Object[], Object[], String) assertEqualsNoOrder(Object[], Object[])
Assert.assertExists	assertExists(String)

Method	Signature
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)`

9.1.2. Subject

Represents the current logged-in JBoss ON user. smsAddress toString userConfiguration

Method	Signature
subject.addLdapRole	`addLdapRole(Role)`
subject.addRole	`addRole(Role)` `addRole(Role, boolean)`

Method	Signature
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 thephone 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.

9.1.3. 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.

9.1.4. 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)`

9.1.5. 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 ofor 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)`

9.1.6. 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	

9.1.7. 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)`

9.2. 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 4, “Proxy Methods for Platforms”
- Table 5, “Proxy Methods for JBoss AS/EAP Servers”
- Table 6, “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 4.8, “Using Resource Proxies”.

Table 4. 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 5. 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 6. 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)`