JBoss Operations Network 3.0 Deploying Applications and Content

for provisioning applications and managing content streams
Edition 3.0.1

Ella Deon Lackey

JBoss Operations Network 3.0 Deploying Applications and Content

for provisioning applications and managing content streams
Edition 3.0.1

Ella Deon Lackey
dlackey@redhat.com

Legal Notice

Abstract

JBoss Operations Network can control content for its resources in a number of different ways: deploying and upgrading applications through provisioning; creating content repositories; and defining content streams for resources, such as the JBoss Customer Service Portal. This guide provides GUI-based procedures to manage content that can be used by resources.

Table of Contents

1. Summary: Using JBoss ON to Deploy Applications and Update Content

One of the core management tools for JBoss Operations Network is to create, update, or remove content from its managed resources. *Content* can be anything associated with a resource or configuration, such as text files, binary files like JARs, EARs, and WARs, patches, and XML files. That content can be deployed on a managed resource to update that resource's configuration, to create a child resource, or to deploy an entirely new application.

There are two ways to manage content for resources:

- Resource-level content through the **`Content`** tabs
- Provisioning applications through *bundles*

Resource-level content allows a specific managed resource, usually a JBoss application server or a web server, to be associated with stored and versioned packages in named repositories. These packages can be uploaded into JBoss ON (so JBoss ON is essentially the content repository), they can be pulled from an external repository, or they can be discovered through agent plug-ins. In other words, there are three actions that resource-level content management can perform:

- It can deliver packages, updates, and patches to a resource.
- It can deploy content to a resource and even create a new child resource. This is particularly useful with web and application servers which can have different contexts as children.
- It can discover the current packages installed on a resource, creating a package digest that administrators can use to manage that asset.

Resource-level content management is limited how far it can be used to create resources. That is why JBoss ON has another system of deploying content, one that allows it to deploy full application servers or to consistently apply content across multiple resources: *provisioning through bundles*.

Bundles are added to the JBoss ON server, so they are not restricted to a single resource. They are deployed to compatible groups of resources, either platforms or JBoss servers (or other resource types which define a bundle target in their plug-in descriptor). This allows multiple resources to be updated at once, using the same content.

Bundle provisioning also allows more flexible and complex deployment options:

- Use Ant calls to perform operations before or after deploying the bundle
- Allow user-defined values or edits to configuration at the time a bundle is provisioned
- Have multiple versions of the same content bundle deployed and deployable to resources at the same time
- Revert to an earlier bundle version

2. Provisioning Applications and Content

Provisioning is a way that administrators can define and control applications, from development to production. The ultimate effect of the provisioning system is simplifying how applications are deployed. Administrators can control which versions of the same application are deployed to different resources, from different content sources, within the same application definition (the *bundle definition*). Resources can be reverted to different versions or jump ahead in deployment.

2.1. An Introduction to Provisioning Content Bundles

Provisioning takes one set of files (a *bundle*) and then pushes it to a platform or an application server (*the destination*). There are more complex ways of defining the content, the destinations, and the rules for that deployment, but the core of the way that provisioning handles content is to take versioned bundles and send it to the designated resource.

Provisioning works with compatible groups, not individual resources. Administrators can define groups based on disparate environments and consistently apply application changes (upgrades, new deployments, or reversions) across all group members, simultaneously.

And the type of content which can be deployed, itself, is flexible. A bundle can contain raw configuration files, scripts, ZIP archives, JAR files, or full application servers — the definition of *content* is fairly loose.

This is in contrast to the resource-level content management in JBoss ON. The type of content is relatively limited. Patches or configuration is applied per-resource. New applications can only be deployed as children of existing resources and it has to be another resource type.

Provisioning focuses on *application management*, not purely resource management.

2.1.1. Bundles: Content and Recipes

A *bundle* is a set of content, packaged in an archive. In real life, a bundle is usually an application, but it can also contain a set of configuration files, scripts, libraries, or any other content required to set up an application or a resource.

The purpose of a bundle is to take that defined set of content and allow JBoss ON to copy it onto a remote resource. The provisioning process basically builds the application on the targeted resource, so in that sense, the bundle is an application distribution. Each bundle version has its own *recipe* which tells JBoss ON what files exist in the bundle, any *tokens* which need to have real values supplied at deployment, and how to handle the bundle and existing files on the remote machine.

The recipe, configuration files, and content are all packaged together into the bundle. This is usually a ZIP file, which the agent unpacks during provisioning.

As with other content managed in JBoss ON, the bundle is versioned. Different versions can be deployed to different resources, which is good for handling different application streams in different environments (say, QA and production). Versioning bundles also allows JBoss ON to revert or upgrade bundles easily.

The bundle can contain almost any kind of content, but it has to follow a certain structure for it to be properly deployed by JBoss ON. The recipe is an Ant build file called **`deploy.xml`**; this must always be located in the top level of the bundle archive.

Past the placement of the recipe, the files and directories within the bundle can be located anywhere in the archive. In fact, the files do not necessarily need to be included in the bundle file at all; when the bundle is created, any or all files for the bundle can be pulled off a URL, which allows the content to be taken from an SVN or GIT repository, FTP server, or website.

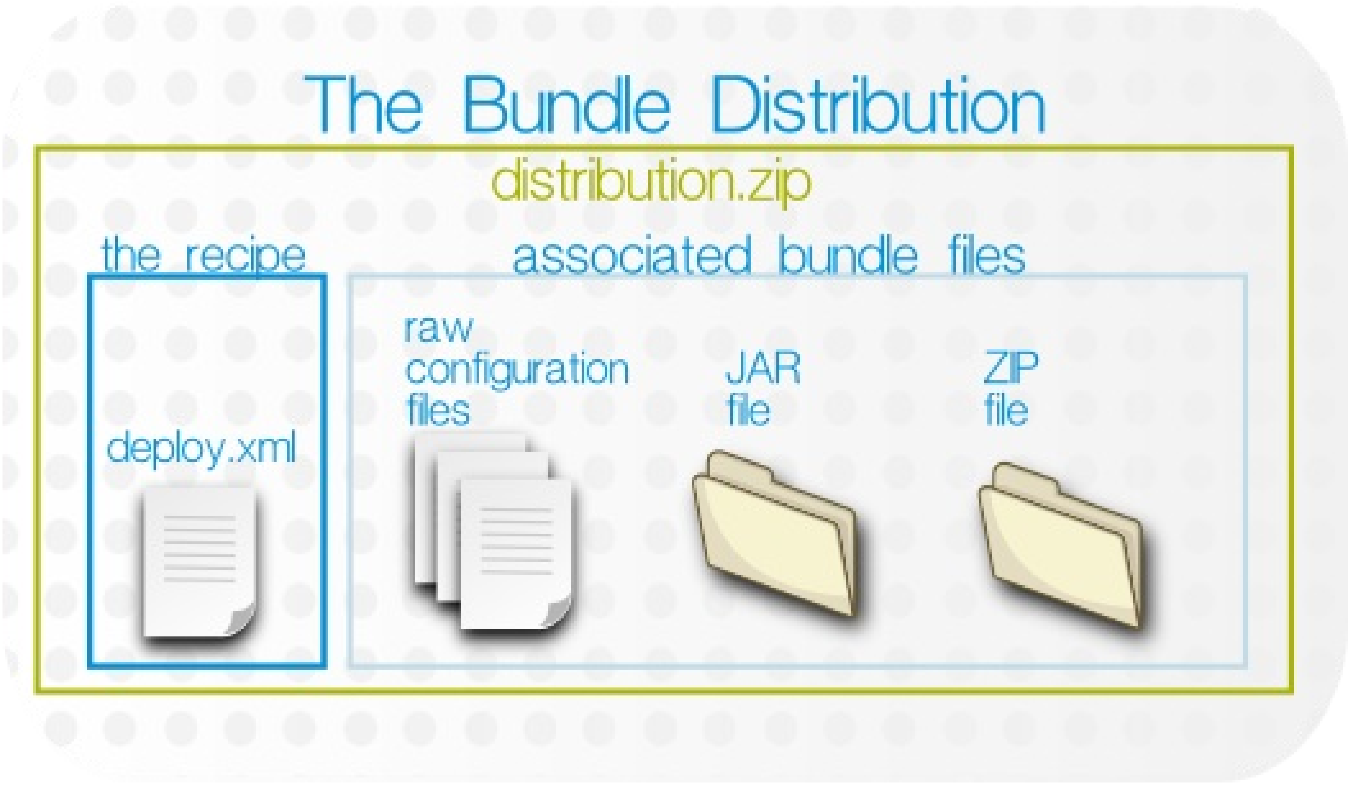


Figure 1. Bundle Layout

The bundle archive can contain other archives, such as JAR, WAR, and ZIP files. Provisioning uses Ant to build out bundles on the target machine, so any files which Ant can process can be processed as part of the bundle. The Ant provisioning system can process WAR, JAR, and ZIP archive files.

2.1.2. Destinations (and Bundle Deployments)

Uploading a bundle to JBoss ON does not push the bundle anywhere, so it is not automatically associated with a resource or group. (Bundles, unlike content, is resource-independent. It exists as its own definition in JBoss ON, apart from the inventory.) When the bundle is actually provisioned, then the provisioning wizard prompts for the administrator to define the *definition*.

A destination is the place where bundles get deployed. The destination is the combination of three elements:

- A compatible resource group (of either platforms or JBoss servers)
- A *base location*, which is the root directory to use to deploy the bundle. Resource plug-ins define a base location for that specific resource type in the **`<bundle-target>`** element. This can be the root directory or, for JBoss servers, common directories like the profile directory. There may be multiple available base locations.
- The *deployment directory*, which is a subdirectory beneath the base directory where the bundle content is actually sent.

For example, an administrator wants to deploy a web application to a JBoss EAP 5 server, in the **`deploy/myApp/`** directory. The JBoss AS5 plug-in defines two possible base locations, one for the installation directory and one for the profile directory. The administrator chooses the profile directory, since the application is an exploded JAR file. The agent then derives the real, absolute path of the application from those three elements:

```
JBoss AS group + {$PROFILE_DIR} + deploy/myApp/
```

If the *PROFILE_DIR* is **`/opt/jbossas/default/server/`**, then the destination is:

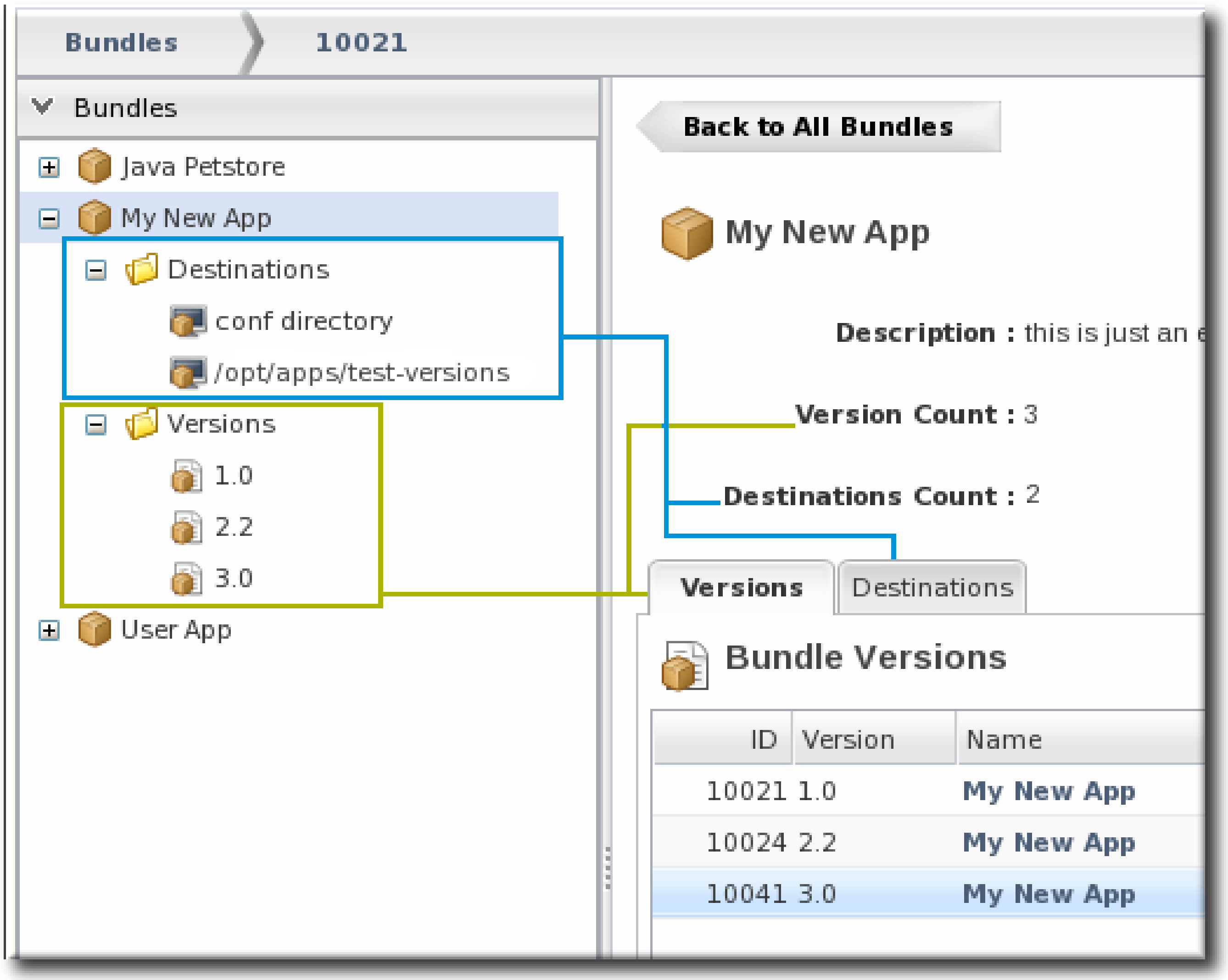
```
/opt/jbossas/default/server/deploy/myApp/
```

If the same resource group contains a JBoss EAP instance running on a Windows server, with a *PROFILE_DIR* of **`C:\jbossas\server\`**, then the path is derived slightly differently, appropriate for the platform:

```
C:\jbossas\default\server\deploy\myApp
```

It is up to the agent, based on the platform and resource information for its inventory, to determine the absolute path for the destination to which the bundle should be deployed.

Once a bundle is actually deployed to a destination, then that association — bundle version and destination — is the *bundle deployment*.



Description : this is just an e

Version Count : 3

Destinations Count : 2

ID	Version	Name
10021	1.0	My New App
10024	2.2	My New App
10041	3.0	My New App

Figure 2. Bundles, Versions, and Destinations

2.1.3. File Handling During Provisioning

A bundle file just contains a set of files and directories that should be pushed to a resource. However, the provisioning process does not merely copy the files over to the deployment directory; provisioning treats a bundle as, essentially, a template that defines the entire content structure for the deployment directory.

For example, a bundle contains these files:

```
app.conf
lib/myapp.jar
```

If the deployment directory is **`deploy/`**, then the final directory configuration is going to be:

```
deploy/app.conf
deploy/lib/myapp.jar
```

By default, if there are any files in **`deploy/`**, then they will be removed before the bundle is copied over, so that the deployment directory looks exactly the way the bundle is configured.

For an application-specific destination, like **`deploy/myApp/`**, then that behavior is totally acceptable because the defined application content should be the only content in that directory. However, bundles can contain a variety of different files and directories and can be deployed almost anywhere on a platform or within a JBoss server. In a lot of deployment scenarios, the deployment directory will have existing data that should be preserved.

The deployment directory is the *root directory* for the bundle. The bundle can define a parameter that tells the provisioning process how to handle data in that root directory. The *manageRootDir* option tells provisioning to delete everything and force the directory to match the bundle content. In other words, the bundle defines the content and structure of the root directory. Alternatively, if the data in that directory must be saved, the *manageRootDir* option can be set to false, which means that provisioning will copy over the bundle and create the appropriate files and subdirectories, but it will not manage (remove) the existing content in the directory.

Note

Any existing content in the root directory is backed up before it is deleted, so it can be restored later.

After the initial deployment, there can be instances where files are added to the deployment directory, such as log files or additional data.

Within the deployment directory, the provisioning process overwrites any bundle-associated files with the latest version and removes any files that are not part of the bundle. Log files, other bundles files, and other data — as with the root directory — need to be preserved between upgrades. Those known files and directories can be called out in the recipe using the **`<rhq:ignore>`** element, which tells the provisioning process to ignore those files*within* the deployment directory.

Setting these options in the recipe is described in Section 2.3.2.2, “Saving Files During Provisioning”.

Important

Purging a bundle deployment removes all of the bundle files from the target resources.

The exact files that are purged mirrors how the bundle manages the deployment directory. By default, purging includes deleting the deployment directory (**`manageRootDir=true`**). If the deployment directory is used by other applications – like an app server **`deploy/`** directory — then those other applications or files will also be deleted. After purging, there is no live deployment and nothing to revert.

2.1.4. Requirements and Resource Types

By default, three resource types support bundles:

- Platforms, all types
- JBoss AS 4
- JBoss AS 5 and any server which uses the JBoss AS 5 plug-in

Bundle support is defined in the plug-in descriptor, so custom plug-ins can be created that add bundle support for those resource types. For examples of writing agent plug-ins with bundle support, see "Writing Custom JBoss ON Plug-ins."

2.1.5. Additional Ant References

Provisioning relies on Ant configuration and tasks, so a good understanding of the Ant build process is beneficial. There are several resources for additional Ant information:

- Apache Ant documentation main page
- Apache Ant documentation for the build file
- Liquibase Database Schema Tasks
- Ant Contrib Tasks

2.2. *Extended Example*: Provisioning Applications to a JBoss EAP Server (Planning)

The Setup

Tim the IT Guy at Example Co. has to manage the full application lifecycle for Example Music's online band management application, MusicApp. There are two environments: one for QA and one for the live site. Both environments contain a mix of Windows and Linux servers.

Tim wants to deploy the latest development version weekly to the QA environment, based on the most current build in their development GIT repo. He wants the most stable version of the application to be deployed to the production environment, based on a static package.

What to Do

The best plan for Tim is to work backwards, starting with the way he wants his ideal QA and production environments to be configured.

Tim's first step is to identify his destinations, based on his environments. Because he has two separate environments, he wants to create two separate groups, one for QA and one for production. MusicApp runs on a JBoss server, so his compatible groups will be for the JBoss AS/EAP resources rather than platforms.

Additionally, the needs for each of his environments is different:

- The QA environment needs ...
  - New builds directly from the GIT repository, every week.
  - A completely clean directory to begin from with every deployment.
  - There is a separate QA environment for each of Example Co.'s web applications, so MusicApp is the only application running on those specific servers.
- The production environment needs ...
  - A stable build that can be safely stored in JBoss ON.
  - To save historic data. The production environment has both log directories and user-supplied data directories that need to be preserved between application upgrades.
  - A couple of different web applications run on the same production servers.

The application itself is the same for both environments. Instance-specific configuration — port numbers, the application name, the machine IP address — are based on tokens that are realized when the application is deployed. The JAR files contained in the bundle should be extracted at the time the application is deployed, with the exception of one client which site members can either install or launch locally.

Tim decides to use different versions of the same bundle, labeling the QA versions as *devel* and the production versions as *stable*.

There are some similarities between the devel and stable bundle recipes:

- MusicApp should be deployed to the **`deploy/`** directory, but because it is not the only application that they run, it will have its own webapp context subdirectory. While this is not strictly necessary in the devel environment (where MusicApp is the only application), this maintains consistency with the final deployment destination.
- Both recipes will configure the application JAR file, **`MusicApp.jar`**, to be exploded when it is deployed.
- The client archive file, **`MyMusic.jar`**, will not be exploded (**`<rhq:file ... exploded="false">`**).
- Tokens are defined in the raw configuration files and the recipe for the port numbers, IP addresses, and application names.

And then there are differences in the recipes, related to how the devel and stable versions should handle existing files.

- The QA environment always requires a pristine deployment. This requires three settings:
  - The **`manageRootDir`** value is always true, so no existing files are preserved during the initial deployment.
  - No **`<rhq:ignore>`** elements are set, so no generated files are preserved during an upgrade.

    - The **cleanDeployment** option is always set in the JBoss ON CLI script that automates deployments. This removes all bundle-associated files in the directory before deploying the new bundle.

- The production environment needs to preserve its existing data between upgrades, which requires two settings:

    - The **manageRootDir** value is always false, which preserves existing files during the initial deployment.

    - Two **<rhq:ignore>** elements are set, one for the log directory and one for the data directory containing the site member uploads.

The last significant action comes when the bundles are actually uploaded to JBoss ON.

Version 1 of the application is already stable and complete, so Tim creates the first bundle as a stable version. He packages the **deploy.xml** with the other application files in a ZIP file and uploads the entire bundle directly to JBoss ON, so it is stored in the JBoss ON database.

Version 2 is a devel version. The QA environment requires frequent updates based on the latest build in GIT. Tim uploads the **deploy.xml** separately, but he points the provisioning wizard to the GIT URL for all of the associated packages. When the bundle is deployed, JBoss ON takes the packages from the repository.

The Results

Tim deployed version 1 of the bundle to the production environment, and he deployed version 2 to the QA environment.

This means that Tim has deployed different versions of the same application, pulled from different sources, to different resources. If he ever has a problem with the production server, he can simply revert it to the last stable version.

Additionally, he can script bundle deployments to the QA environment, so his tests can be fully automated.

2.3. Creating Ant Bundles

Bundles are archive files that is stored on the server and then downloaded by an agent to deploy to a platform or resource. A bundle distribution is comprised of two elements:

- An Ant recipe file named **deploy.xml**

- Any associated application files. These application files can be anything; commonly, they fall into two categories:

    - Archive files (JAR or ZIP files)

    - Raw text configuration files, which can include tokens where users define the values wher the bundle is deployed

2.3.1. Using Templatized Configuration Files

A bundle can contain configuration files for an application. These configuration files can use hard-coded values or they can use *tokens* that are filled in (automatically or with user-supplied values) when the bundle is actually deployed.

Note

For a user-defined token to be realized, it must be referenced in the recipe so that the bundle deployment wizard will prompt for the value, using the **`<rhq:input-property>`** key in the Ant recipe. For examples, see Section 2.3.2.4.2, "rhq:input-property" and Example 1, "Simple Ant Recipe".

User-defined tokens can be any property; the values are supplied through the provisioning UI and inserted into the templatized configuration file.

The token key is a simple attribute-value assertion, with the *input_field* as the element in the UI and the *property* being the value in the configuration file. The property of user-defined tokens must contain only alphanumeric characters, an underscore (_), or a period (.); no other characters are allowed.

```
input_field=@@property@@
```

For example, to set a port number token in a configuration file, define the property:

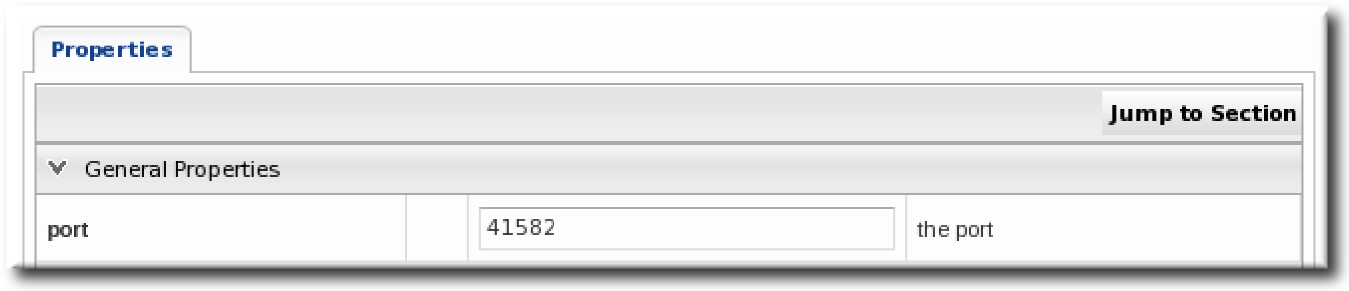
```
port=@@listener.port@@
```

The user-defined token then must be noted in the recipe, so that the provisioning process knows to realize the phrase. To configure a property in an Ant recipe, add a **`<rhq:input-property>`** key in the Ant XML file.

For example:

```
<rhq:input-property
    name="listener.port"
    ... />
```

The provisioning wizard prompts for a value for all of the user-defined tokens referenced in the recipe.



port	41582	the port

Figure 3. Port Token During Provisioning

Along with user-defined variables that can be specified in the recipe file, there are variables that are made implicitly available to recipes. These tokens can be used in a templatized file as a user-defined variable without having to define the token template in the recipe itself.

Table 1. Variables Defined by JBoss ON

Token	Description
rhq.deploy.dir	The directory location where the bundle will be installed.
rhq.deploy.id	A unique ID assigned to the specific bundle deployment.
rhq.deploy.name	The name of the bundle deployment.

Additionally, some tokens can be realized by the provisioning process pulling information from the local system. These values, listed in Table 2, “System-Defined Tokens”, are taken either from the Java API or from Java system properties. They can be inserted directly in the templatized configuration file without having to put a corresponding entry in the recipe. For example:

```
@@rhq.system.hostname@@
```

Table 2. System-Defined Tokens

Token Name	Taken From...	Java API
rhq.system.hostname	Java API	SystemInfo.getHostname()
rhq.system.os.name	Java API	SystemInfo.getOperatingSystemName()
rhq.system.os.version	Java API	SystemInfo.getOperatingSystemVersion()
rhq.system.os.type	Java API	SystemInfo.getOperatingSystemType().toString()
rhq.system.architecture	Java API	SystemInfo.getSystemArchitecture()
rhq.system.cpu.count	Java API	SystemInfo.getNumberOfCpus()
rhq.system.interfaces.java.address	Java API	InetAddress.getByName(SystemInfo.getHostname()).getHostAddress()
rhq.system.interfaces.*network_adapter_name*.mac	Java API	NetworkAdapterInfo.getMacAddress()
rhq.system.interfaces.*network_adapter_name*.type	Java API	NetworkAdapterInfo.getType()
rhq.system.interfaces.*network_adapter_name*.flags	Java API	NetworkAdapterInfo.getAllFlags()
rhq.system.interfaces.*network_adapter_name*.address	Java API	NetworkAdapterInfo.getUnicastAddresses().get(0).getHostAddress()
rhq.system.interfaces.*network_adapter_name*.multicast.address	Java API	NetworkAdapterInfo.getMulticastAddresses().get(0).getHostAddress()
rhq.system.sysprop.java.io.tmpdir	Java system property	
rhq.system.sysprop.file.separator	Java system property	
rhq.system.sysprop.line.separator	Java system property	
rhq.system.sysprop.path.separator	Java system property	

Token Name	Taken From...	Java API
rhq.system.sysprop.java.home	Java system property	
rhq.system.sysprop.java.version	Java system property	
rhq.system.sysprop.user.timezone	Java system property	
rhq.system.sysprop.user.region	Java system property	
rhq.system.sysprop.user.country	Java system property	
rhq.system.sysprop.user.language	Java system property	

2.3.2. Creating Ant Recipes

Note

The process and guidelines for actually creating an Ant recipe are outside the scope of this documentation. This document outlines the options and requirements for using Ant recipes *specifically* to work with the JBoss ON provisioning system.

For basic instructions, options, and tutorials for writing Ant tasks, see the Apache Ant documentation at http://ant.apache.org/manual/index.html.

- Section 2.3.2.1, “Breakdown of an Ant Recipe”
- Section 2.3.2.2, “Saving Files During Provisioning”
- Section 2.3.2.3, “Using Ant Tasks”
- Section 2.3.2.4, “A Reference of JBoss ON Ant Recipe Elements”

2.3.2.1. Breakdown of an Ant Recipe

The Ant recipe for JBoss ON bundles is the same basic file as a standard Apache Ant file and is processed by an integrated Ant build system in JBoss ON. This Ant recipe file must be bundled in the top directory of the distribution ZIP file and be named **`deploy.xml`**.

The JBoss ON Ant recipes allows all of the standard tasks that are available for Ant builds, which provides flexibility in scripting a deployment for a complex application. The JBoss ON Ant recipe *must* also provide additional information about the deployment that will be used by the provisioning process; this includes information about the destination and, essentially, metadata about the application itself.

Example 1. Simple Ant Recipe

For provisioning, the Ant recipe is more of a definition file than a true script file, although it can call Ant targets and do pre- and post-provisioning operations. As with other Ant scripts, the JBoss ON Ant recipe uses a standard XML file with a **`<project>`** root element and defined targets and tasks. The elements defined in the **`<rhq:bundle>`** area pass metadata to the JBoss ON provisioning system when the project is built.

The first part of the **deploy.xml** file simply identifies the file as an an script and references the provisioning Ant elements.

```
<?xml version="1.0"?>
<project name="test-bundle" default="main"
 xmlns:rhq="antlib:org.rhq.bundle">
```

The next element identifies the specific bundle file itself. The provisioning system can manage and deploy multiple versions of the same application; the **<rhq:bundle>** element contains information about the specific version of the bundle (including, naturally enough, an optional version number).

```
    <rhq:bundle name="Example App" version="2.4" description="an example
bundle">
```

All that is *required* for a recipe is the**<rhq:bundle>** element that defines the name of the application. However, the bundle element contains all of the information about the application and, importantly, how the provisioning system should handle content contained in the application.

The first item to address is any *templatized property* that is used in a configuration file. This is covered in Section 2.3.1, “Using Templatized Configuration Files”. Any token used in a configuration file must be defined in the recipe for it to be realized (to have a value supplied) during provisioning. For the ***port*** token defined in Section 2.3.1, “Using Templatized Configuration Files”, the **<rhq:input-property>** element identifies it in the recipe. The***name*** argument is the *input_field* value in the token, the***description*** argument gives the field description used in the UI and the other arguments set whether the value is required, what its allowed syntax is, and any default values to supply. (This doesn't list the files which use tokens, only the tokens themselves.)

```
  <rhq:input-property
              name="listener.port"
              description="This is where the product will listen for
incoming messages"
              required="true"
              defaultValue="8080"
              type="integer"/>
```

There is a single element which identifies all of the content deployed by the bundle, the **<rhq:deployment-unit>** element. The entire application — its name, included ZIP or JAR files, configuration files, Ant targets — are all defined in the **<rhq:deployment-unit>** parent element.

The name and any Ant targets are defined as arguments on **<rhq:deployment-unit>** directly. In this, the name is **appserver**, and one preinstall target and one postinstall target are set.

```
        <rhq:deployment-unit name="appserver"
preinstallTarget="preinstall" postinstallTarget="postinstall"
manageRootDir="false">
```

There is one other critical element on the **<rhq:deployment-unit>** element: the **manageRootDir** argument. Provisioning doesn't simply copy over files; as described in Section 2.1.3, “File Handling During Provisioning”, it remakes the directory to match what is in the bundle. If there are any existing files in the deployment directory when the bundle is first deployed, they are deleted by default. Setting **`manageRootDir`** to false means that the provisioning process does *not* manage the deployment directory — meaning any existing files are left alone when the bundle is deployed.

Any configuration file is identified in an **`<rhq:file>`** element. The ***name*** is the name of the configuration file *within the bundle*, while the **`destinationFile`** is the relative (to the deployment directory) path and filename of the file after it is deployed.

```
  <rhq:file name="test-v2.properties"
 destinationFile="conf/test.properties" replace="true"/>
```

Bundles can contain archive files, either ZIP or JAR files. Every archive file is identified in an **`<rhq:archive>`** element within the deployment-unit. The**`<rhq:archive>`** element does three things:

- Identify the archive file by name.
- Define how to handle the archive. Simply put, it sets whether to copy the archive over to the destination and then leave it as-is, still as an archive, or whether to extract the archive once it is deployed. This is called *exploding* the archive. If an archive is exploded, then a postinstall task can be called to move or edit files, as necessary.
- Identify any files within the archive which contain tokens that need to be realized. This is a child element, **`<rhq:fileset>`**. This can use wildcards to include types of files or files within subdirectories or it can explicitly state which files to process.

```
             <rhq:archive name="MyApp.zip" exploded="true">
                 <rhq:replace>
                     <rhq:fileset>
                         <include name="**/*.properties"/>
                     </rhq:fileset>
                 </rhq:replace>
             </rhq:archive>
```

Another possible child element sets how to handle any files within the deployment directory that are not part of the bundle. For example, the application may generate log files or it may allow users to upload content. By default, the provisioning process cleans out a directory from non-bundle content every time a bundle is provisioned. However, logs, user-supplied data, and other types of files are data that should remain intact after provisioning. Any files or subdirectories which should be ignored by the provisioning process (and therefore preserved) are identified in the **`<rhq:ignore>`** element. In this case, any**`*.log`** files within the **`logs/`** directory are saved.

```
             <rhq:ignore>
                 <rhq:fileset>
                     <include name="logs/*.log"/>
                 </rhq:fileset>
             </rhq:ignore>
         </rhq:deployment-unit>
     </rhq:bundle>
```

This only applies to upgrading a bundle, meaning *after* the initial deployment.

The last elements set the Ant tasks to run before or after deploying the content, as identified initially in the **`<rhq:deployment-unit>`** arguments. Most common Ant tasks are supported (as described in Section 2.3.2.3, “Using Ant Tasks”). This uses a preinstall task to print which directory the bundle is being deployed to and whether the operation was successful. The postinstall task prints a message when the deployment is complete.

```
<target name="main" />

    <target name="preinstall">
        <echo>Deploying Test Bundle v2.4 to ${rhq.deploy.dir}...</echo>
 <property name="preinstallTargetExecuted" value="true"/>
 <rhq:audit status="SUCCESS" action="Preinstall Notice"
info="Preinstalling to ${rhq.deploy.dir}" message="Another optional
message">
  Some additional, optional details regarding
  the deployment of ${rhq.deploy.dir}
 </rhq:audit>
    </target>

    <target name="postinstall">
        <echo>Done deploying Test Bundle v2.4 to ${rhq.deploy.dir}.</echo>
        <property name="postinstallTargetExecuted" value="true"/>
    </target>
</project>
```

Section 2.3.2.4, "A Reference of JBoss ON Ant Recipe Elements" lists the different JBoss ON elements in the Ant recipe file. For information on standard Ant tasks, see the Apache Ant documentation.

2.3.2.2. Saving Files During Provisioning

One important thing to consider with an Ant recipe is how to handle files in the deployment directory. (This is touched on in Section 2.1.3, "File Handling During Provisioning".)

By default, deploying or updating a bundle replaces everything in the deployment directory, either by overwriting it or deleting it. The file handling rules are very similar to RPM package upgrade rules. This is very simplified, but the provisioning process responds in one of two ways to existing files the deployment directory:

1. The file in the current directory is also in the bundle. In this case, the bundle file always overwrites the current file. (There is one exception to this. If the file in the bundle has not been updated and is the same version as the local file, but the local file has modifications. In that case, the local file is preserved.)

2. The file in the current directory does not exist in the bundle. In that case, the bundle deletes the file in the current directory.

The behavior for #2, when a file is deleted, can be changed by settings in the Ant recipe.

There are three ways to manage if and how files are preserved during provisioning: **`manageRootDir`**, **`<rhq:ignore>`**, and **`cleanDeployment`**.

manageRootDir

All of the information about the application being deployed is defined in the **`<rhq:deployment-unit>`** element in a bundle recipe. The **`manageRootDir`** attribute on the **`<rhq:deployment-unit>`** element sets how the provisioning process should handle existing files in the deployment directory.

The default value is *manageRootDir=true* which means that the provisioning process deletes any other files in the root directory.

Alternately, the value can be set to false, which tells the provisioning process to ignore any existing files in the root directory, as long as there is not a corresponding file in the bundle.

The **`manageRootDir`** attribute applies to both the initial deployment and upgrade operations, so this can be used to preserve files that may exist in a directory before a bundle is ever deployed.

See Section 2.3.2.4.3, “rhq:deployment-unit”.

Note

When a bundle will no longer be used on a resource, it can be entirely removed from the filesystem. This is called *purging*. The way that the provisioning system handles files when purging a bundle mirrors that way that it handles files when provisioning a system. By default, purging a bundle deletes everything in the deployment directory. If the *manageRootDir* option is set in the bundle, then the provisioning process removes all of the files and directories associated with the bundle and leaves unrelated files and directories intact.

<rhq:ignore>

There can be files that are used or created by an application, apart from the bundle, which need to be preserved after a bundle deployment. This can include things like log files, instance-specific configuration files, or user-supplied content like images. These files can be ignored during the provisioning process, which preserves the files instead of removing them.

To save files, use the **`<rhq:ignore>`** element and list the directories or files to preserve.

```
<rhq:ignore>
    <rhq:fileset>
        <include name="logs/*.log"/>
    </rhq:fileset>
</rhq:ignore>
```

The **`<rhq:ignore>`** element only applies when bundles are updated; it does not apply when a bundle is initially provisioned.

Also, the **`<rhq:ignore>`** element only applies to file that exist outside the bundle. Any files that are in the bundle will overwrite any corresponding files in the deployment directory, even if they are specified in the **`<rhq:ignore>`** element.

See Section 2.3.2.4.10, “rhq:ignore”.

Clean Deployment

Both **`manageRootDir`** and **`<rhq:ignore>`** are set in the recipe. At the time that the bundle is actually provisioned, there is an option to run a *clean deployment*. The clean deployment option deletes everything in the deployment directory and provisions the bundle in a clean directory, regardless of the **`manageRootDir`** and **`<rhq:ignore>`** settings in the recipe.

See Section 2.4.5, “Deploying a Bundle to a Clean Destination”.

2.3.2.3. Using Ant Tasks

An Ant bundle distribution file is just an Ant recipe and its associated files. As Example 1, "Simple Ant Recipe" shows, the Ant recipe is the expected **deploy.xml** file with some JBoss ON-specific elements. An Ant bundle distribution file supports more complex Ant configuration, including Ant tasks and targets.

2.3.2.3.1. Supported Ant Tasks

Any standard Ant task can be run as part of the Ant bundle provisioning (with the exception of **<antcall>** and **<macrodef>**). This includes common commands like **echo**, **mkdir**, and **touch** — whatever is required to deploy the content fully.

Important

The **<antcall>** element *cannot* be used with the Ant recipe. **<antcall>** calls a target within the **deploy.xml** file, which loops back to the file, which calls the **<antcall>** task again, which calls the **deploy.xml** file again. This creates an infinite loop.

To perform the same operations that would be done with **<antcall>**, use the **<ant>** task to reference a separate XML file which contains the custom Ant targets. This is described in Section 2.3.2.3.3, "Calling Ant Targets".

Important

The **macrodef** call, and therefore macro definitions, are not supported with Ant bundles.

Along with the standard Ant tasks, Ant bundle recipes can use optional Ant tasks:

- Liquibase Database Schema Tasks
- Ant Contrib Tasks

2.3.2.3.2. Using Default, Pre-Install, and Post-Install Targets

As with other Ant tasks, the **<project>** allows a default target, which is required by the provisioning system. This is a no-op because the Ant recipe mainly defines the metadata for and identifies files used by the provisioning process. Other operations aren't necessary. This target is required by Ant, even though it is a no-op target. Use pre- and post-install targets to perform tasks with the bundle before and after it is unpacked.

For example:

```
<target name="main" />
```

Additionally, JBoss ON provisioning tasks can define both pre- and post-install targets. This allows custom tasks, like simple progress messages or setting properties.

2.3.2.3.3. Calling Ant Targets

As mentioned in Section 2.3.2.3.1, "Supported Ant Tasks", using **<antcall>** does not work in an Ant bundle recipe; it self-referentially calls the **<rhq:bundle>** task in an infinite loop. However, it is possible to process tasks that are *outside* the default target. This can be done using pre- and post install targets (Section 2.3.2.3.2, "Using Default, Pre-Install, and Post-Install Targets").

1. In **deploy.xml** for the Ant recipe, add a**<rhq:deployment-unit>** element which identifies the Ant target.

```
<rhq:deployment-unit name="jar" postinstallTarget="myExampleCall">
```

2. Then, define the target.

```
    <target name="myExampleCall">
       <ant antfile="another.xml" target="doSomething">
          <property name="param1" value="111"></property>
       </ant>
    </target>

```

3. Create a separate **another.xml** file in the same directory as the**deploy.xml** file. This file contains the Ant task.

```
<?xml version="1.0"?>
<project name="another" default="main">
    <target name="doSomething">
       <echo>inside doSomething. param1=${param1}</echo>
    </target>
</project>
```

2.3.2.4. A Reference of JBoss ON Ant Recipe Elements

- Section 2.3.2.4.1, “rhq:bundle”
- Section 2.3.2.4.2, “rhq:input-property”
- Section 2.3.2.4.3, “rhq:deployment-unit”
- Section 2.3.2.4.4, “rhq:archive”
- Section 2.3.2.4.5, “rhq:url-archive”
- Section 2.3.2.4.6, “rhq:file”
- Section 2.3.2.4.7, “rhq:url-file”
- Section 2.3.2.4.8, “rhq:audit”
- Section 2.3.2.4.9, “rhq:replace”
- Section 2.3.2.4.10, “rhq:ignore”
- Section 2.3.2.4.11, “rhq:fileset”
- Section 2.3.2.4.12, “rhq:system-service”
- Section 2.3.2.3.2, “Using Default, Pre-Install, and Post-Install Targets”

2.3.2.4.1. rhq:bundle

Contains the definition for the main JBoss ON-related Ant task that is required for any Ant bundle recipe. This element defines basic information about the bundle and is the parent element for all of the specific details about what is in the bundle and how it should be provisioned.

Element Attributes

Attribute	Description	Optional or Required
name	The name given to the bundle.	Required
version	The version string for this specific bundle. Bundles can have the same name, but each bundle of that name must have a unique version string. These version strings normally conform to an OSGi style of versioning, such as **`1.0`** or **`1.2.FINAL`**.	Required
description	A readable description of this specific bundle version.	Optional

Example

```
<rhq:bundle name="example" version="1.0" description="an example bundle">
```

2.3.2.4.2. rhq:input-property

Adds a property to the bundle task that defines a template token that must have its value supplied by a user at the time the bundle is deployed. This is similar to standard Ant properties.

Note

All of the system properties listed in Table 2, “System-Defined Tokens” and the Ant-specific tokens in Table 1, “Variables Defined by JBoss ON” are available to be used as templatized tokens in bundle configuration *without* having to set a **`<rhq:input-property>`** definition.

All input properties set some parameter that must have its value defined by a user when the bundle is provisioned on a resource, and the fields to enter those values are automatically generated in the JBoss ON UI bundle deployment wizard.

Element Attributes

Attribute	Description	Optional or Required
name	The name of the user-defined property. Within the recipe, this property can be referred to by this name, in the format **`${`***property_name***`}`**.	Required
description	A readable description of the property. This is the text string displayed in the JBoss ON bundle UI when the bundle is deployed.	Required

Attribute	Description	Optional or Required
type	Sets the syntax accepted for the user-defined value. There are several different options: • string • longString • long • password • file • directory • boolean • integer • float • double	Required
required	Sets whether the property is required or optional for configuration. The default value is **`false`**, which means the property is optional. If this argument isn't given, then it is assumed that the property is optional.	Optional
defaultValue	Gives a value for the property to use if the user does not define a value when the bundle is deployed.	Optional

Example

```
<rhq:input-property
    name="listener.port"
    description="This is where the product will listen for incoming
messages"
    required="true"
    defaultValue="8080"
    type="integer"/>
```

See Also

- Section 2.3.2.4.4, “rhq:archive”
- Section 2.3.2.4.6, “rhq:file”

2.3.2.4.3. rhq:deployment-unit

Defines the bundle content — such as applications or configuration files — being deployed by the bundle. A deployment unit can be simple text files, archives, or a full software product, including an application server, web server, or database. A deployment unit can have multiple archive and configuration files associated with it.

Only a single deployment unit is provisioned at a time by the provisioning process, so there can be only one **`<rhq:deployment-unit>`** element in a bundle recipe.

Element Attributes

Attribute	Description	Optional or Required
name	The name of the application.	Required
manageRootDir	Sets whether JBoss ON should manage all files in the top root directory (deployment directory) where the bundle is deployed. If false, any unrelated files found in the top deployment directory are ignored and will not be overwritten or removed when future bundle updates are deployed. The default is true.	Optional
preinstallTarget	An Ant target that is invoked before the deployment unit is installed.	Optional
postinstallTarget	An Ant target that is invoked after the deployment unit is installed.	Optional

Example

```
<rhq:deployment-unit name="appserver" preinstallTarget="preinstall"
postinstallTarget="postinstall">
```

See Also

- Section 2.3.2.3.2, “Using Default, Pre-Install, and Post-Install Targets”

2.3.2.4.4. rhq:archive

Defines any archive file that is associated with deploying the application. An archive can be a ZIP or JAR file. A bundle doesn't require an archive file, so this element is optional.

Element Attributes

Attribute	Description	Optional or Required
name	The filename of the archive file to include in the bundle. **Important** If the archive file is packaged with the Ant recipe file inside the bundle distribution ZIP file, then the **`name`** must contain the *relative path* to the location of the archive file in the ZIP file.	Required

Attribute	Description	Optional or Required
exploded	Sets whether the archive's contents will be extracted and stored into the bundle destination directory (true) or whether to store the files in the same relative directory as is given in the **name** attribute (false). If the files are exploded, they are extracted starting in the deployment directory. Post-install targets can be used to move files after they have been extracted.	Optional

Example

```
<rhq:archive name="file.zip">
    <rhq:replace>
         <rhq:fileset>
               <include name="**/*.properties"/>
         </rhq:fileset>
    </rhq:replace>
</rhq:archive>
```

See Also

- Section 2.3.2.4.2, “rhq:input-property”
- Section 2.3.2.4.11, “rhq:fileset”
- Section 2.3.2.4.9, “rhq:replace”

2.3.2.4.5. rhq:url-archive

Defines remote archive to use, which is accessed through the given URL. This is similar to **`rhq:archive`** except that the server accesses the archive over the network rather than including the archive directly in the bundle distribution file.

Element Attributes

Attribute	Description	Optional or Required

Attribute	Description	Optional or Required
url	Gives the URL to the location of the archive file. The archive is downloaded and installed in the deployment directory. **Note** For the bundle to be successfully deployed, the URL must be accessible to all agent machines where this bundle is to be deployed. If an agent cannot access the URL, it cannot pull down the archive and thus cannot deploy it on the machine.	Required
exploded	If true, the archive's contents will be extracted and stored into the bundle destination directory; if false, the zip file will be compressed and stored in the top level destination directory. **Note** If the files are exploded, they are extracted starting in the deployment directory. Post-install targets can be used to move files after they have been extracted.	Optional

Example

```
<rhq:url-archive url="http://server.example.com/apps/files/archive.zip">
    <rhq:replace>
         <rhq:fileset>
               <include name="**/*.properties"/>
         </rhq:fileset>
    </rhq:replace>
</rhq:url-archive>
```

See Also

- Section 2.3.2.4.4, “rhq:archive”
- Section 2.3.2.4.2, “rhq:input-property”
- Section 2.3.2.4.11, “rhq:fileset”
- Section 2.3.2.4.9, “rhq:replace”

2.3.2.4.6. rhq:file

Contains the information to identify and process configuration files for the application which have token values that must be realized. Normally, configuration files are copied directly from the bundle package into the deployment directory. The **`<rhq:file>`** element calls out files that require processing before they should be copied to the destination. The attributes on the **`<rhq:file>`** element set the name of the raw file in the bundle distribution ZIP file and the name of the target file that it should be copied to.

Raw files can be included with the archive files that contain properties or configuration for the application. These configuration files can be templatized with user-defined or system-defined tokens, like those listed in Section 2.3.1, “Using Templatized Configuration Files”. Any templatized files that are included in the bundle distribution file that are templatized must be listed in the Ant recipe so that they are processed and the tokens are realized.

Element Attributes

Attribute	Description	Optional or Required
name	The name of the raw configuration file. **Important** If the configuration file is packaged with the Ant recipe file inside the bundle distribution ZIP file, then the **`name`** must contain the *relative path* to the location of the file within the ZIP file.	Required

Attribute	Description	Optional or Required
destinationFile	The full path and filename for the file on the destination resource. Relative paths must be relative to the final deployment directory (defined in the **`rhq.deploy.dir`** parameter when the bundle is deployed). It is also possible to use absolute paths, as long as both the directory and the filename are specified. **Note** If the **`destinationDir`** attribute is used, the **`destinationFile`** attribute *cannot* be used.	Required, unless destinationDir is used
destinationDir	The directory where this file is to be copied. If this is a relative path, it is relative to the deployment directory given by the user when the bundle is deployed. If this is an absolute path, that is the location on the filesystem where the file will be copied. This attribute sets the *directory* for the file to be copied to. The actual file name is set in the **`name`** attribute. If the **`destinationFile`** attribute is used, the **`destinationDir`** attribute *cannot* be used.	Required, unless destinationFile is used
replace	Indicates whether the file is templatized and requires additional processing to realize the token values.	Required

Example

```
<rhq:file name="test-v2.properties" destinationFile="subdir/test.properties"
replace="true"/>
```

If neither the **`destinationDir`** nor the **`destinationFile`** attribute is used, then the raw file is placed in the same location under the deployment directory as its location in the bundle distribution.

2.3.2.4.7. rhq:url-file

As with **`rhq:file`**, contains the information to identify and process configuration files for the application which have token values that must be realized. This option specifies a remote file which is downloaded from the given URL, rather than being included in the bundle archive.

Element Attributes

Attribute	Description	Optional or Required
url	Gives the URL to the templatized file. The file is downloaded and installed in the deployment directory. **Note** For the bundle to be successfully deployed, the URL must be accessible to all agent machines where this bundle is to be deployed. If an agent cannot access the URL, it cannot pull down the archive and thus cannot deploy it on the machine.	Required

Attribute	Description	Optional or Required
destinationFile	The full path and filename for the file on the destination resource. Relative paths must be relative to the final deployment directory (defined in the **rhq.deploy.dir** parameter when the bundle is deployed). It is also possible to use absolute paths, as long as both the directory and the filename are specified. **Note** If the **destinationDir** attribute is used, the **destinationFile** attribute *cannot* be used. This attribute must give both the path name and the file name.	Required, unless destinationDir is used
destinationDir	The directory where this file is to be copied. If this is a relative path, it is relative to the deployment directory given by the user when the bundle is deployed. If this is an absolute path, that is the location on the filesystem where the file will be copied. This attribute sets the *directory* for the file to be copied to. The actual file name is set in the **name** attribute. If the **destinationFile** attribute is used, the **destinationDir** attribute *cannot* be used.	Required, unless destinationFile is used
replace	Indicates whether the file is templatized and requires additional processing to realize the token values.	Required

Example

```
<rhq:url-file url="http://server.example.com/apps/files/test.conf"
destinationFile="subdir/test.properties" replace="true"/>
```

If neither the **`destinationDir`** nor the **`destinationFile`** attribute is used, then the raw file is placed in the same location under the deployment directory as its location in the bundle distribution.

See Also

- Section 2.3.2.4.6, "rhq:file"

2.3.2.4.8. rhq:audit

Sets custom audit trail messages to use during the provisioning process. This is useful with complex recipes that perform some additional custom tasks. As the tasks are processed, the **`rhq:audit`** configuration sends information to the server about the additional processing steps and their results.

Element Attributes

Attribute	Description	Optional or Required
status	The status of the processing. The possible values are SUCCESS, WARN, and FAILURE. The default is SUCCESS.	Optional
action	The name of the processing step.	Required
info	A short summary of what the action is doing, such as the name of the target of the action or an affected filename.	Optional
message	A brief text string which provides additional information about the action.	Optional

Example

```
<rhq:audit status="SUCCESS" action="Preinstall Notice" info="Preinstalling
to ${rhq.deploy.dir}" message="Another optional message">
 Some additional, optional details regarding
 the deployment of ${rhq.deploy.dir}
</rhq:audit>
```

2.3.2.4.9. rhq:replace

Lists templatized files, in children **`<rhq:fileset>`** elements, contained in the archive which need to have token values realized when the archive is deployed.

Any file which uses a token that must be replaced with a real value is a templatized file. When the provisioning process runs, the token value is substituted with the defined value. This element lists all of the files which are templatized; the only files which are processed by the provisioning system for token substitution are the ones listed in the **`<rhq:replace>`** element.

Example

```
<rhq:archive name="file.zip">
    <rhq:replace>
        <rhq:fileset>
            <include name="**/*.properties"/>
        </rhq:fileset>
    </rhq:replace>
</rhq:archive>
```

See Also

- Section 2.3.2.4.11, “rhq:fileset”
- Section 2.3.2.4.4, “rhq:archive”

2.3.2.4.10. rhq:ignore

Lists files in the deployment directory which should not be deleted when a new bundle is deployed. **This only applies to upgrade operations, not to the initial deployment of a bundle.**

Once an application is deployed, instance-specific files — like data files or logs — can be created and should be retained if the application is ever upgraded. This element, much like **`<rhq:replace>`**, contains a list of files or directories in the instance to save.

Note

If a file is ignored in the recipe, then the file is not deleted when the bundle is deployed. However, if a file of the same name exists in the bundle, then the local file is overwritten.

Do not attempt to ignore files that are packaged in the bundle. Only files generated by the applications, such as log and data files, should be ignored by the provisioning process since they should be preserved for the upgraded instance.

Important

It is possible to deploy one bundle to a subdirectory of another bundle (such as Bundle A is deployed to **`/opt/myapp`** and Bundle B to **`/opt/myapp/webapp1`**).

In that case, set the recipe in Bundle A to ignore the directory to which Bundle B will be deployed. This prevents updates or reversions for Bundle A from overwriting the configuration from Bundle B.

Example

```
<rhq:ignore>
    <rhq:fileset>
        <include name="logs/*.log"/>
    </rhq:fileset>
</rhq:ignore>
```

See Also

- Section 2.3.2.4.11, "rhq:fileset"

2.3.2.4.11. rhq:fileset

Provides a list of files.

Two JBoss ON elements — **`<rhq:replace>`** and **`<rhq:ignore>`** — define file lists in either the archive file or the deployment directory. This element contains the list of files.

Child Element

Child Element	Description
<include name=*filename* />	The filename of the file. For **`<rhq:replace>`**, this is a file within the archive (JAR or ZIP) file which is templatized and must have its token values realized. For **`<rhq:ignore>`**, this is a file in the application's deployment directory which should be ignored and preserved when the bundle is upgraded.

Example

```
<rhq:replace>
     <rhq:fileset>
         <include name="**/*.properties"/>
     </rhq:fileset>
</rhq:replace>
```

See Also

- Section 2.3.2.4.10, "rhq:ignore"
- Section 2.3.2.4.9, "rhq:replace"

2.3.2.4.12. rhq:system-service

Points to a script file to launch as part of the provisioning process. This is usually an init file or similar file that can be used by the deployed application to set the application up as a system service.

Element Attributes

Attribute	Description	Optional or Required
name	The name of the script.	Required

Attribute	Description	Optional or Required
scriptFile	The filename of the script. If the script file is packaged with the Ant recipe file inside the bundle distribution ZIP file, then the **`scriptFile`** must contain the *relative path* to the location of the file in the ZIP file.	Required
configFile	The name of any configuration or properties file used by the script. If the configuration file is packaged with the Ant recipe file inside the bundle distribution ZIP file, then the **`configFile`** must contain the *relative path* to the location of the file in the ZIP file.	Optional
overwriteScript	Sets whether to overwrite any existing init file to configure the application as a system service.	Optional
startLevels	Sets the run level for the application service.	Optional
startPriority	Sets the start order or priority for the application service.	Optional
stopPriority	Sets the stop order or priority for the application service.	Optional

Example

```
<rhq:system-service name="example-bundle-init" scriptFile="example-init-
script"
      configFile="example-init-config" overwriteScript="true"
      startLevels="3,4,5" startPriority="80" stopPriority="20"/>
```

2.3.3. Creating an Associated Archive File

The application that is being deployed itself has to be built into an archive file of some kind. JBoss ON allows JAR and ZIP formats. The bundle archive file can also include raw files that are used to configuration the application, such as XML, **`.conf`**, and text files. These can be templatized to supply user- and system-specific information (as described in Section 2.3.1, “Using Templatized Configuration Files”).

Any required archive or file must be referenced in the recipe so that the server knows to copy it during deployment.

The bundle files can be uploaded and stored in the JBoss ON server or they can be zipped up, with the recipe files, into a single distribution file.

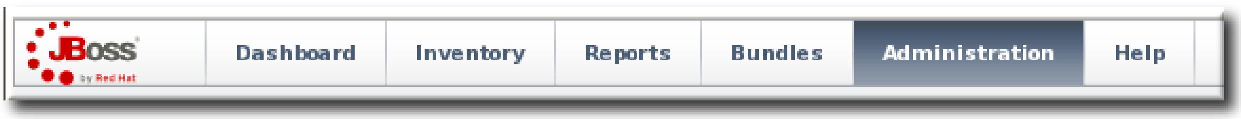
2.3.4. Testing Bundle Packages

Ant recipes can be complex, so it's important (and useful) to test a bundle before deploying it. JBoss ON includes a command-line tool that can be used to test Ant provisioning bundles quickly.

2.3.4.1. Installing the Bundle Deployer Tool

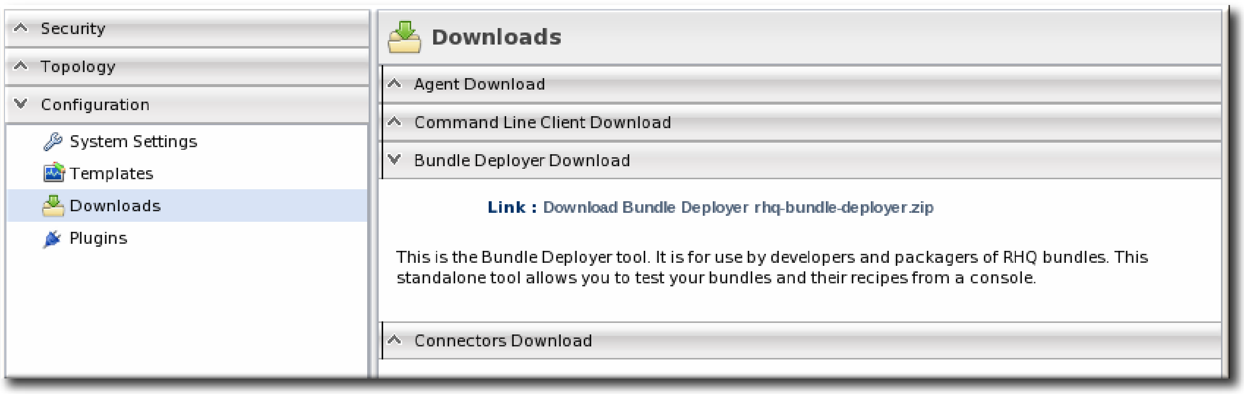
This tool can be downloaded and installed on any machine, independent of any JBoss ON server or agent.

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

2. Select the **`Downloads`** in the left menu table.

3. Scroll to the **`Bundle Deployer Download`** section, and click the package download link.

   This is the Bundle Deployer tool. It is for use by developers and packagers of RHQ bundles. This standalone tool allows you to test your bundles and their recipes from a console.

4. Save the **`.zip`** file into the directory where the bundle tool should be installed, such as **`/opt/`**.

5. Unzip the packages.

   ```
   cd /opt/

   unzip rhq-bundle-deployer-version.zip
   ```

2.3.4.2. Using the Bundle Deployer Tool

Important

This bundle deployment tool is *only* to test the provisioning process and deployed application. This tool does not interact with the JBoss ON server or agent, so JBoss ON is unaware of any applications deployed with this tool and cannot manage them.

1. Unzip the bundle distribution package to check (or copy an unzipped directory that contains the application files). For example:

   ```
   mkdir /tmp/test-bundle
   cd /tmp/test-bundle
   unzip MyBundle.zip
   ```

2. Open the top directory of the bundle distribution, where the **deploy.xml** Ant recipe file is.

3. Set the bundle deployer tool location in the PATH.

```
PATH="/opt/rhq-bundle-deployer-3.0.0/bin:$PATH"
```

4. Run the bundle deploy tool, and use the format **-D***input_properties* to pass the values to user-defined tokens in the templatized files. For example:

```
rhq-ant -Drhq.deploy.dir=/opt/exampleApp -Dlistener.port=7081
```

This installs the application in **/opt/exampleApp** and sets a port value of 7081.

Note

Optionally, use the **rhq.deploy.id** attribute to set an identifier for the deployment. The default is 0, which means a new deployment. When bundles are deployed in the UI, the server assigns a unique ID to the deployment. Using the **rhq.deploy.id** attribute on a new deployment simulates the server's ID assignment.

Using the **rhq.deploy.id** attribute if there is already a previous deployment allows you to test the upgrade performance of the bundle. Performing an upgrade requires a new, unique ID number.

2.4. Provisioning Bundles

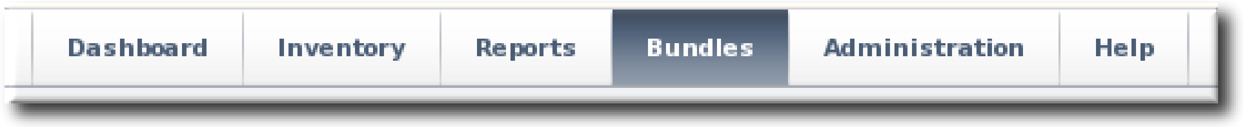
2.4.1. Uploading Bundles to JBoss ON

All of the files associated with a distribution — the recipe, any JARs or ZIPs, and any configuration files — have to be accessible to JBoss ON. Either the files need to be uploaded and stored in the JBoss ON database or a URL to the packages needs to be configured.

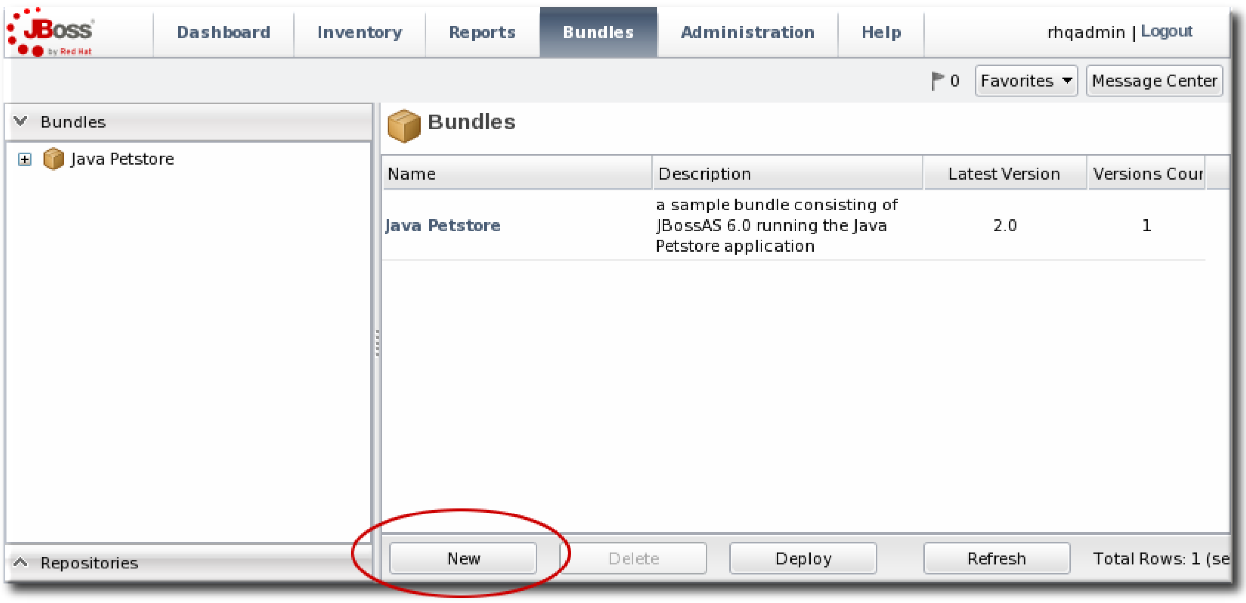
Note

If the files are all combined in a single ZIP file to upload, then the recipe file must be in the top level of the package.

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



2. Scroll to the bottom of the window and click the **New** button.

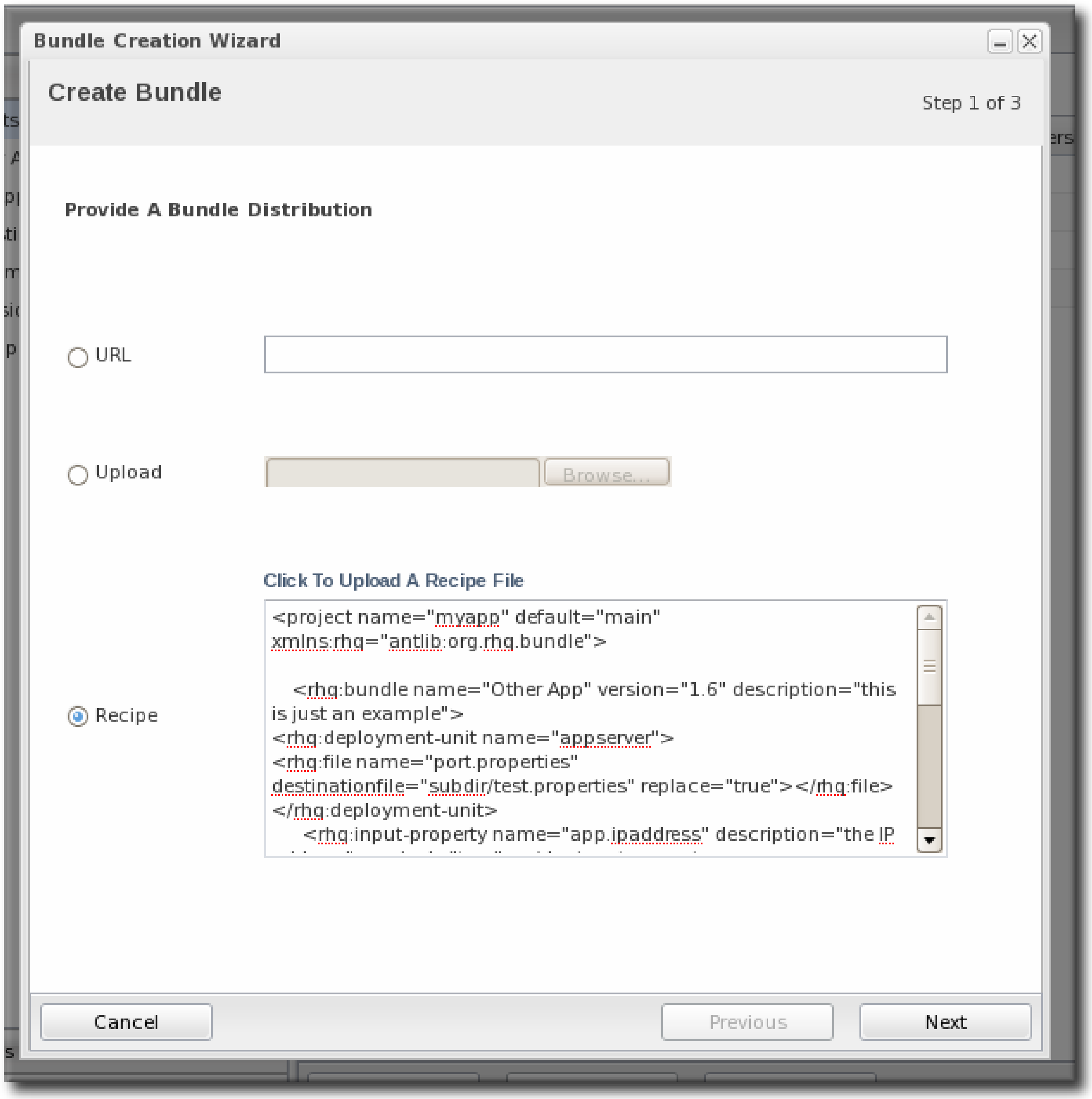


Bundles

Name	Description	Latest Version	Versions Cour
Java Petstore	a sample bundle consisting of JBossAS 6.0 running the Java Petstore application	2.0	1

Total Rows: 1 (se

3. Upload the distribution package or the recipe file.



( ) URL
( ) Upload
(●) Recipe

```
<project name="myapp" default="main"
xmlns:rhq="antlib:org.rhq.bundle">

   <rhq:bundle name="Other App" version="1.6" description="this
is just an example">
<rhq:deployment-unit name="appserver">
<rhq:file name="port.properties"
destinationfile="subdir/test.properties" replace="true"></rhq:file>
</rhq:deployment-unit>
     <rhq:input-property name="app.ipaddress" description="the IP
```

There are three options on how the bundle distribution is made available to the JBoss ON server:

- **URL** points to any URL, such as an FTP site or SVN or GIT repo, where there is a complete bundle distribution file available.

  Note

  Using an SVN or GIT repo allows you to pull the packages directly from a build system.

- **Upload** uploads a single bundle distribution file (which includes both the recipe an all associated files) from the local system to the JBoss ON server.

- **Recipe** uploads a recipe file only, and then any additional files required for the bundle are uploaded separately. This option includes an edit field where the uploaded recipe can be edited before it is sent to the server.

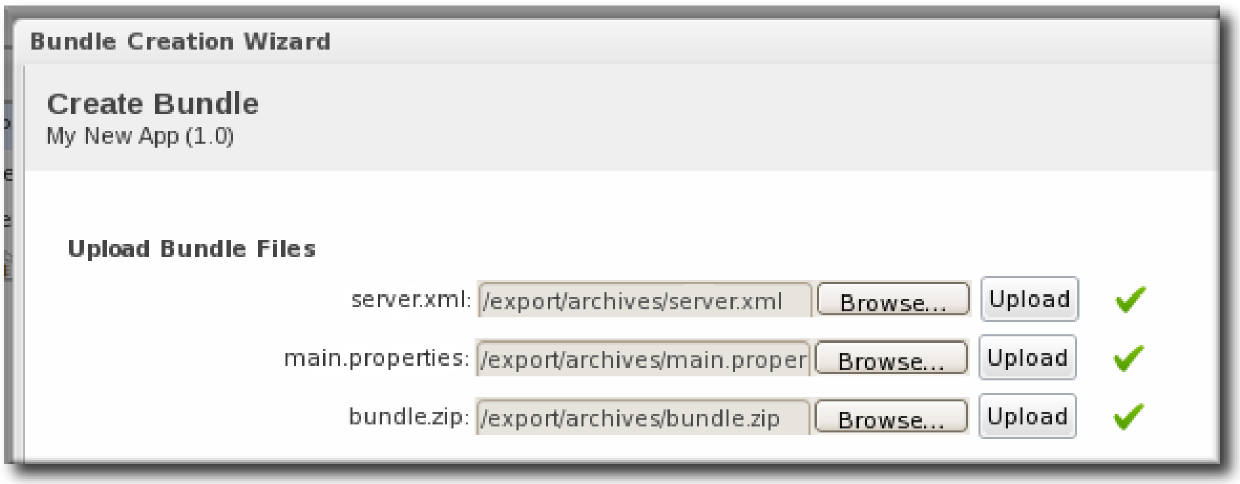
  Note

  When uploading a recipe file separately than the bundle archive files, every closing tag be explicitly stated (meaning every entry must have the format **`<tag></tag>`**, not the abbreviated format**`<tag />`**). Otherwise, the recipe may be incorrectly interpreted in the text box and fail to upload to the server.

  The XML must be well-formed, or the recipe fails validation and the upload fails.

  Additionally, the **Recipe** option's upload button does not work on Internet Explorer. To add a recipe file using this option with Internet Explorer, copy the entire recipe file and paste it directly into the text box.

4. In the next screen, upload any associated files that were not uploaded previously. For the **URL** and **Upload**, all of the files are usually uploaded in a single file, so there is nothing to do on this screen. For the **Recipe** option, all of the files listed in the recipe must be uploaded manually at this step.


Bundle Creation Wizard

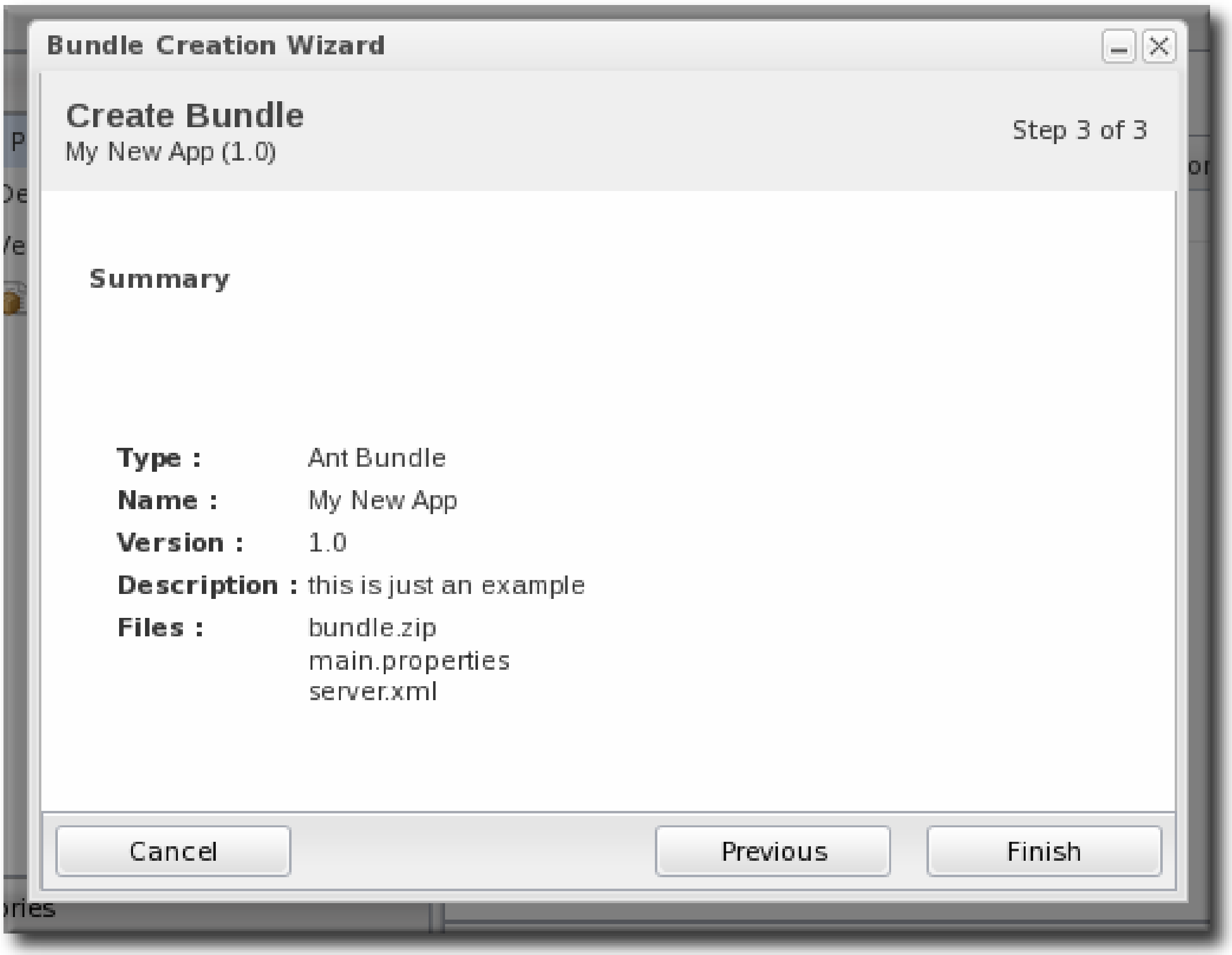
Create Bundle
My New App (1.0)

Upload Bundle Files

server.xml:	/export/archives/server.xml	Browse...	Upload	✓
main.properties:	/export/archives/main.proper	Browse...	Upload	✓
bundle.zip:	/export/archives/bundle.zip	Browse...	Upload	✓

5. The final screen shows all of the information for the new bundle. Click **Finish** to save

the new bundle.



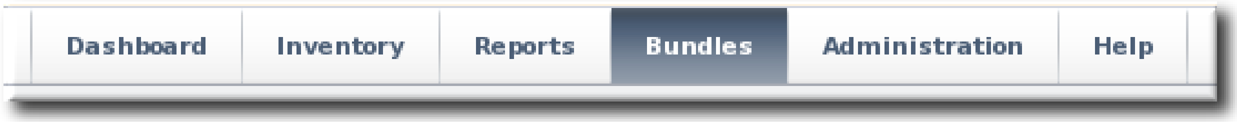
Type :	Ant Bundle
Name :	My New App
Version :	1.0
Description :	this is just an example
Files :	bundle.zip main.properties server.xml

2.4.2. Deploying Bundles to a Resource

Bundles are deployed to *resources* by deploying the bundle to a*JBoss ON group*. Any compatible group that contains resources which support bundles (platforms and JBoss AS resources by default) is automatically listed as an option for the destination.

For platforms, the groups cannot contain different operating systems and architectures. However, the same bundle distribution file and properties can be used for any platform because the provisioning process will automatically format the deployment directory and provisioned files to match the platform's architecture.

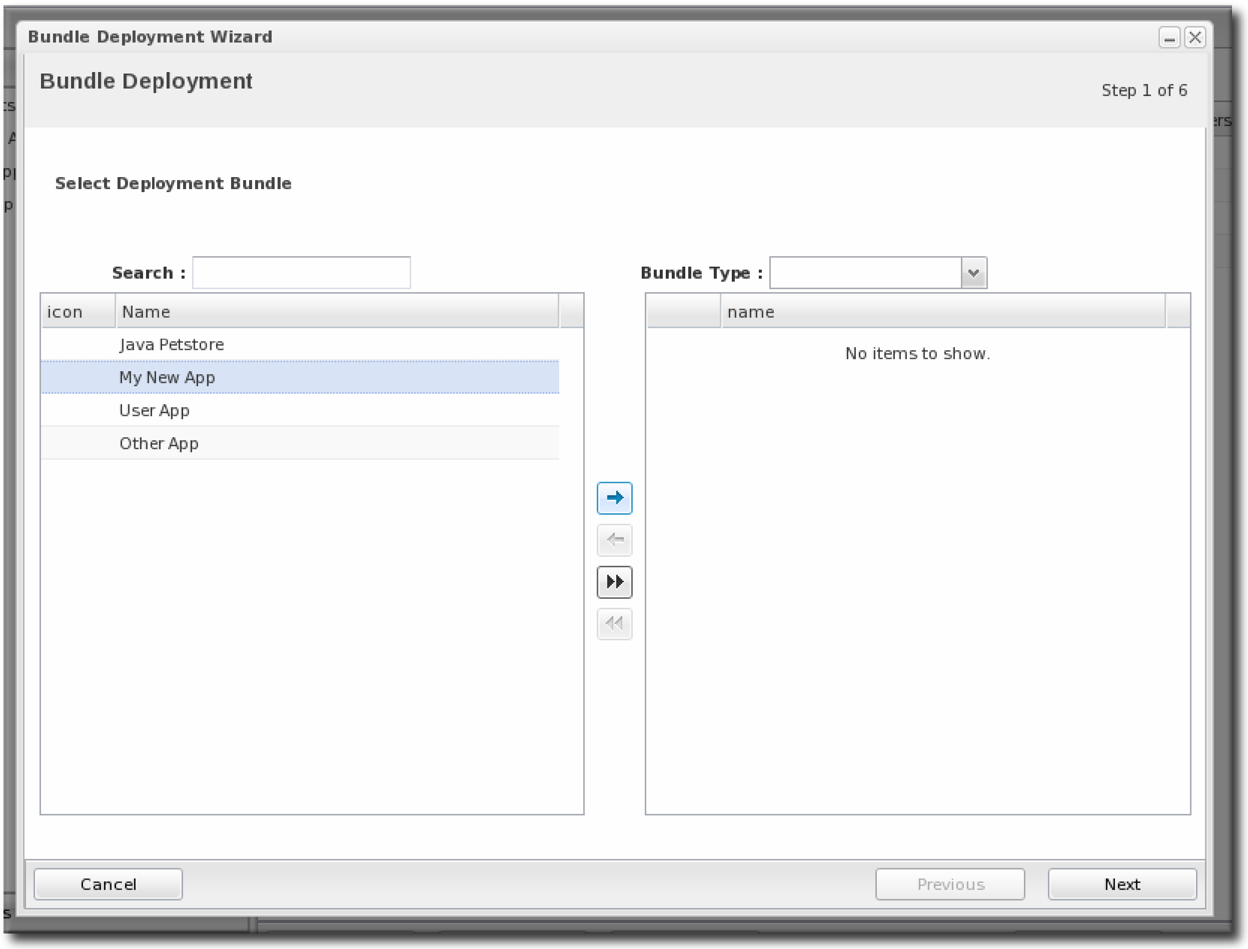
1. In the top menu, click the **`Bundles`** tab.

2. Scroll to the bottom of the window and click the **`Deploy`** button.

   Alternatively, click the name of the bundle in the list, and then click the deploy button at the top of the bundle page.

3. Select the bundles to deploy from the list on the left and use the arrows to move them to the box on the right.



4. Once the bundles are selected, define the destination information.

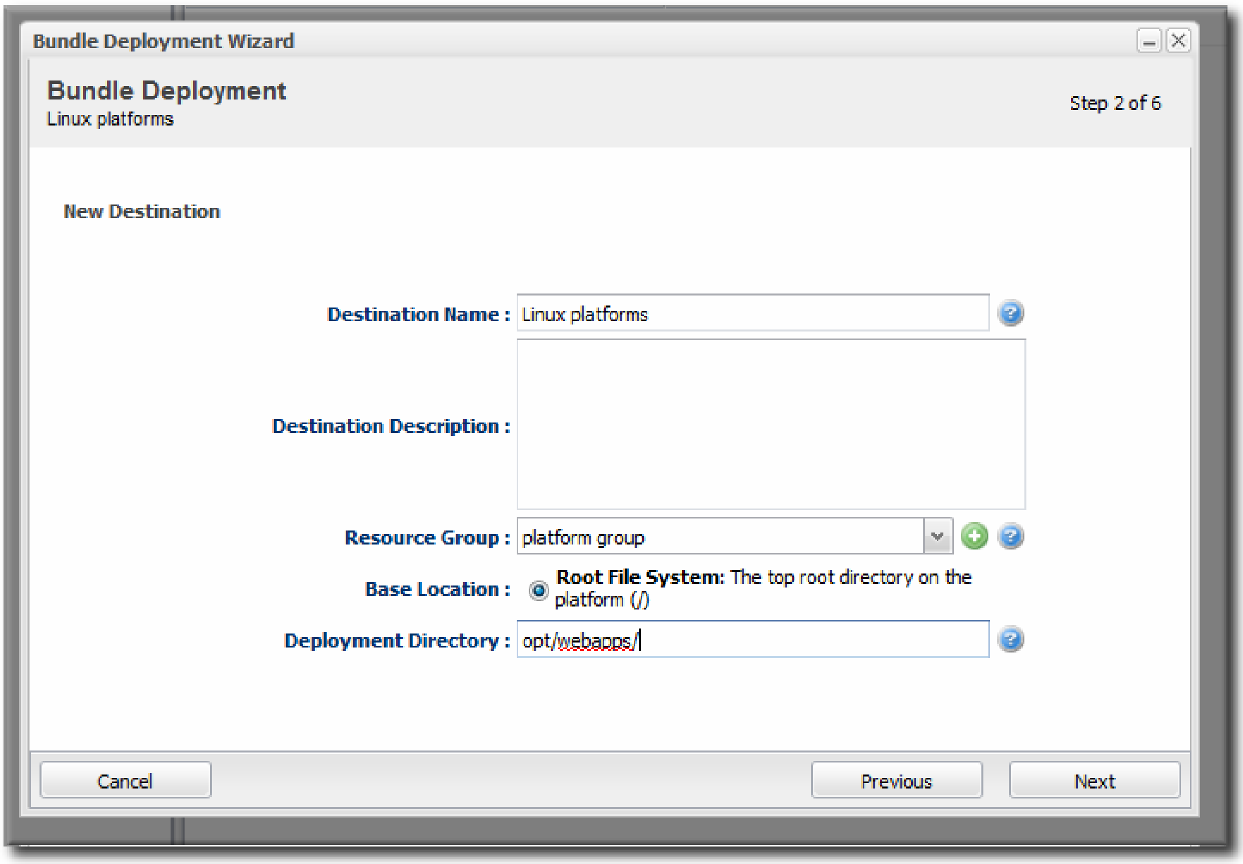
   The destination is a combination of the resources the bundle is deployed on and the directory to which is it deployed. Each destination is uniquely defined for each bundle.

   To define the destination, first select the resource group from the **Resource** drop-down menu. The resource group identifies the type of resource to which the bundle is being deployed, and the resource type defines other deployment parameters. When the group is selected, then the *base location* is defined. For a platform, this is the root directory. For a JBoss AS instance, it is the installation directory. For custom resources, the base location is defined in the plug-in descriptor.

   Note

   If you haven't created a compatible group or if you want to create a new group specifically for this bundle deployment, click the **+** icon to create the group. Then, continue with the provisioning process.

   Set the actual deployment directory to which to deploy the bundle. This directory is a relative path to the plug-in-defined base location.


Bundle Deployment Wizard

Bundle Deployment
Linux platforms

Step 2 of 6

New Destination

Destination Name : Linux platforms

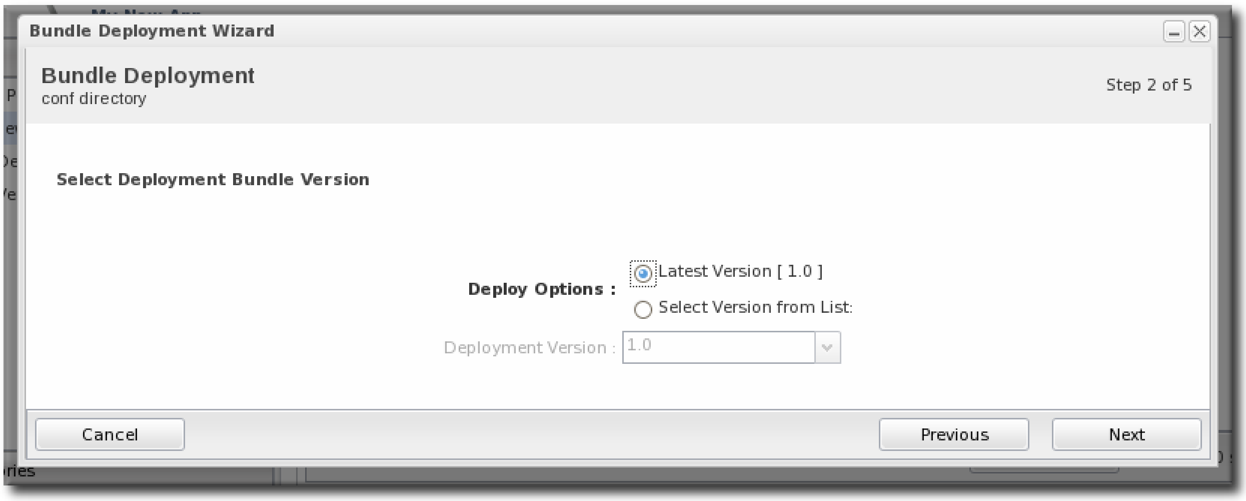
Destination Description :

Resource Group : platform group

Base Location : [✓ **Root File System:** The top root directory on the platform (/)]

Deployment Directory : opt/webapps/

5. Select the version of the bundle to deploy. If there are multiple versions of a bundle available, then any of those versions can be selected. There are also quick options to deploy the latest version or the currently deployed version.


Bundle Deployment Wizard

Bundle Deployment
conf directory

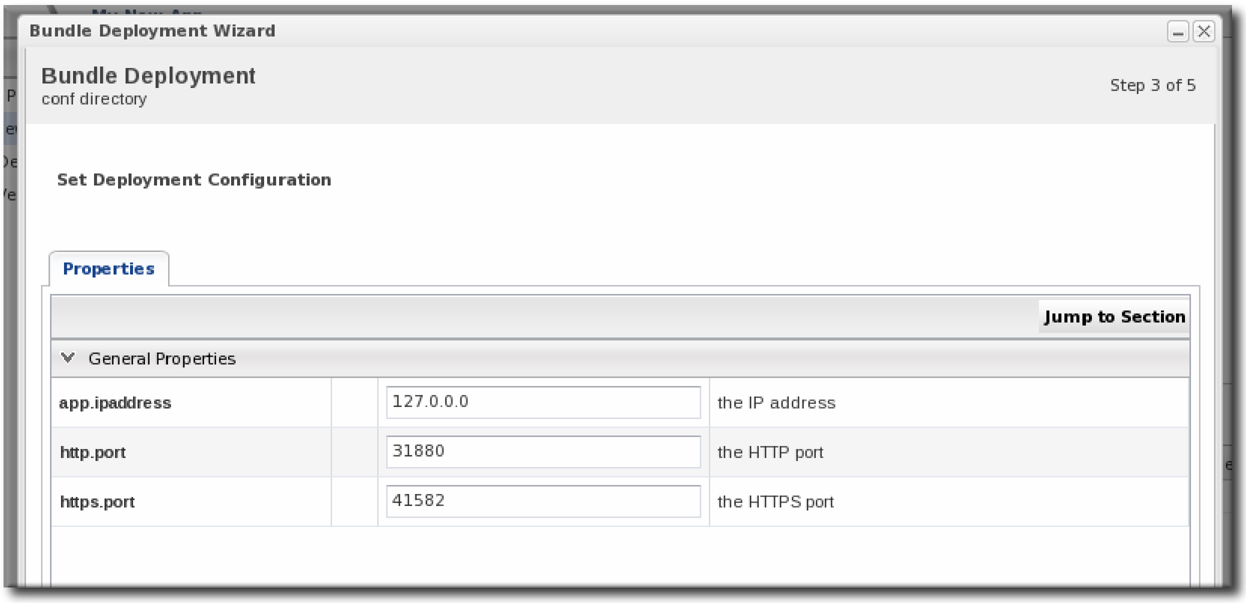
Step 2 of 5

Select Deployment Bundle Version

Deploy Options : [✓ Latest Version [1.0]] Select Version from List:

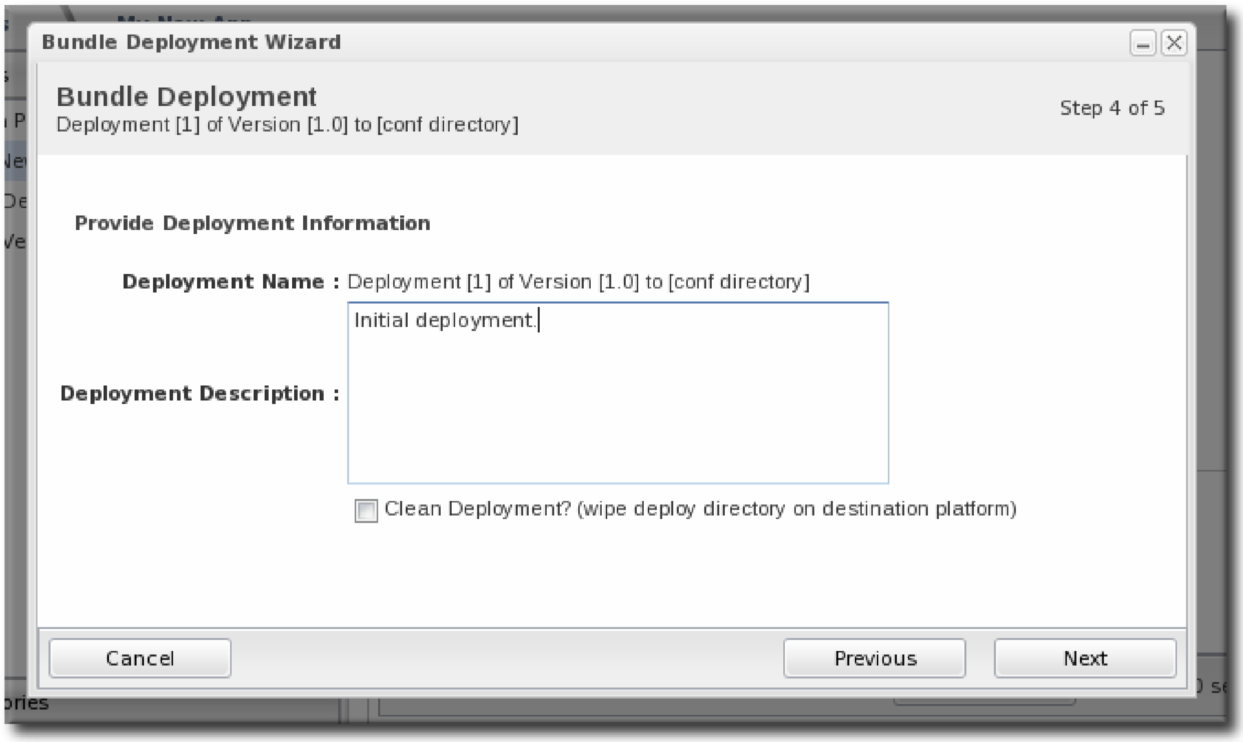
Deployment Version : 1.0

6. If there are any user-defined properties, then they are entered in the fields in the next page. User-defined properties are configured in the bundle recipe using tokens.



app.ipaddress	127.0.0.0	the IP address
http.port	31880	the HTTP port
https.port	41582	the HTTPS port

7. Fill in the information about the specific deployment instance. The checkbox sets the option on whether to overwrite anything in the existing deployment directory or whether to preserve any existing files.



Deployment [1] of Version [1.0] to [conf directory]

Deployment Name : Deployment [1] of Version [1.0] to [conf directory]

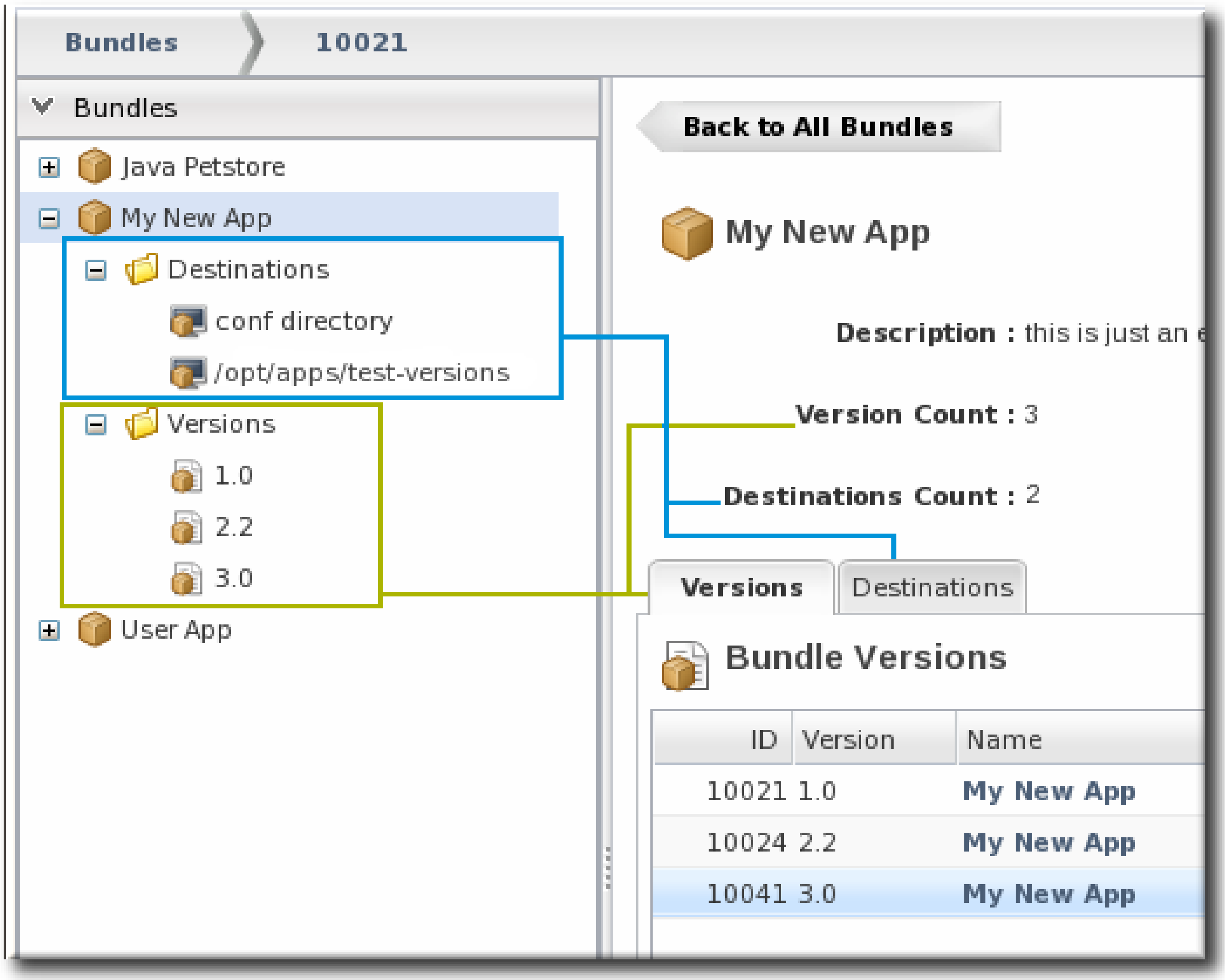
Deployment Description : Initial deployment.

☐ Clean Deployment? (wipe deploy directory on destination platform)

8. The final screen shows the progress for deploying the packages. Click **Finish** to complete the deployment.

2.4.3. Viewing the Bundle Deployment History

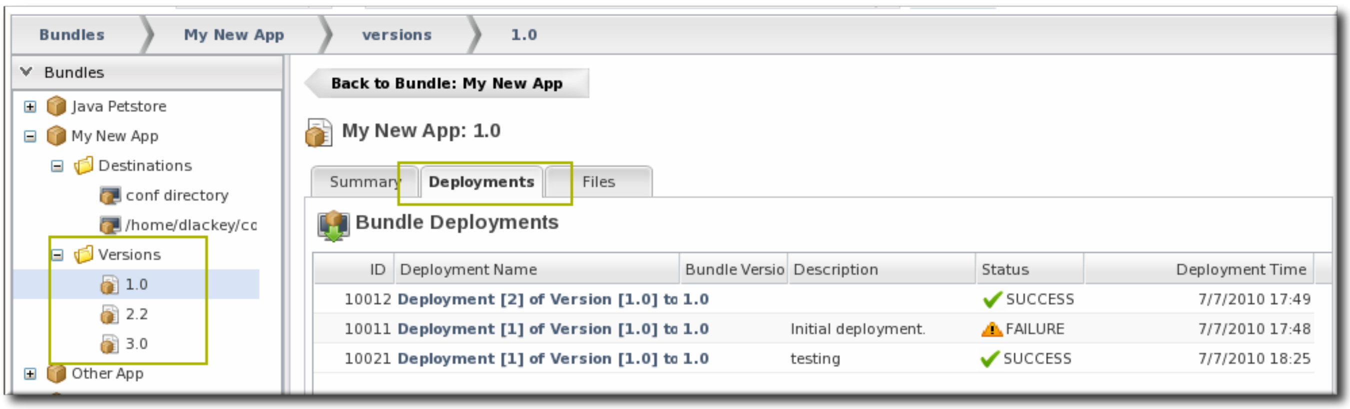
A bundle has two areas of information: one for its versions and one for its destinations (places where it is deployed). The main bundle entry shows only those two things, the versions and the destinations. The version area is a way to track and control the *content of the bundle*, while the destinations area is a way to track and control *the process of deploying bundles*.



ID	Version	Name
10021	1.0	My New App
10024	2.2	My New App
10041	3.0	My New App

Figure 4. Bundles, Versions, and Destinations

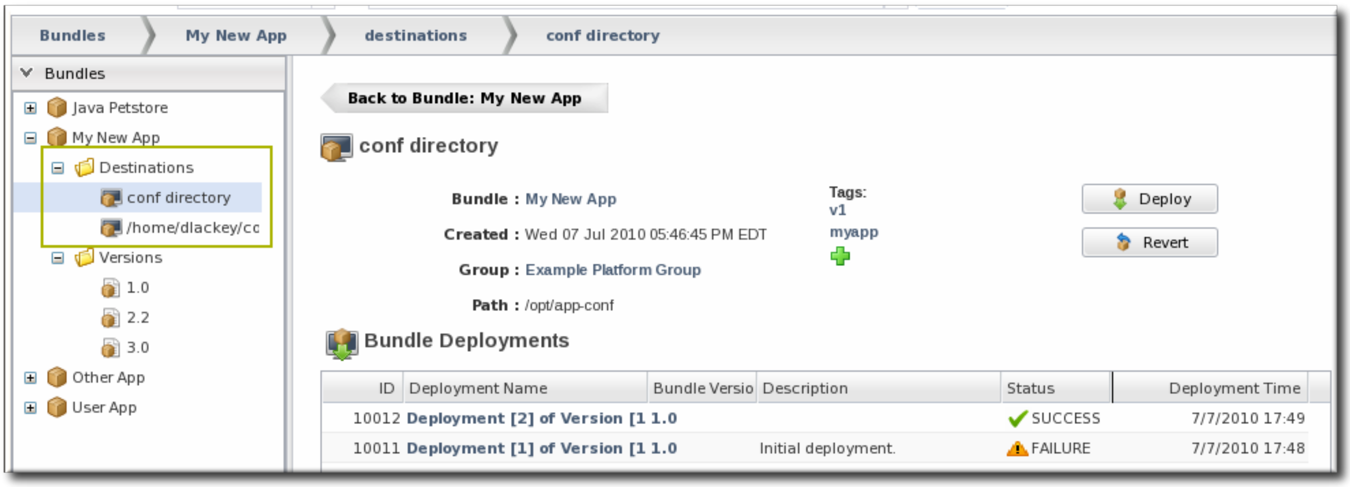
Selecting a version under the main bundle entry shows its recipe (on the **Summary** tab) and a list of all of the files associated with that particular version (on the **Files** tab). The **Deployments** tab shows every destination, with timestamps and comments, that that particular version of the bundle has been deployed to.



ID	Deployment Name	Bundle Versio	Description	Status	Deployment Time
10012	Deployment [2] of Version [1.0] to	1.0		✓ SUCCESS	7/7/2010 17:49
10011	Deployment [1] of Version [1.0] to	1.0	Initial deployment.	⚠ FAILURE	7/7/2010 17:48
10021	Deployment [1] of Version [1.0] to	1.0	testing	✓ SUCCESS	7/7/2010 18:25

Figure 5. Deployment Information for a Version

A destination entry shows only a list of versions that have been deployed to that destination. In a sense, the destination area is the best areas to track the audit history of an application. Along with shows the history of deployments and updates, the destinations area is the place where new versions can be deployed or reverted most directly.



conf directory

Bundle : My New App
Created : Wed 07 Jul 2010 05:46:45 PM EDT
Group : Example Platform Group
Path : /opt/app-conf
Tags: v1 myapp

Bundle Deployments

ID	Deployment Name	Bundle Versio	Description	Status	Deployment Time
10012	Deployment [2] of Version [1	1.0		✓ SUCCESS	7/7/2010 17:49
10011	Deployment [1] of Version [1	1.0	Initial deployment.	⚠ FAILURE	7/7/2010 17:48

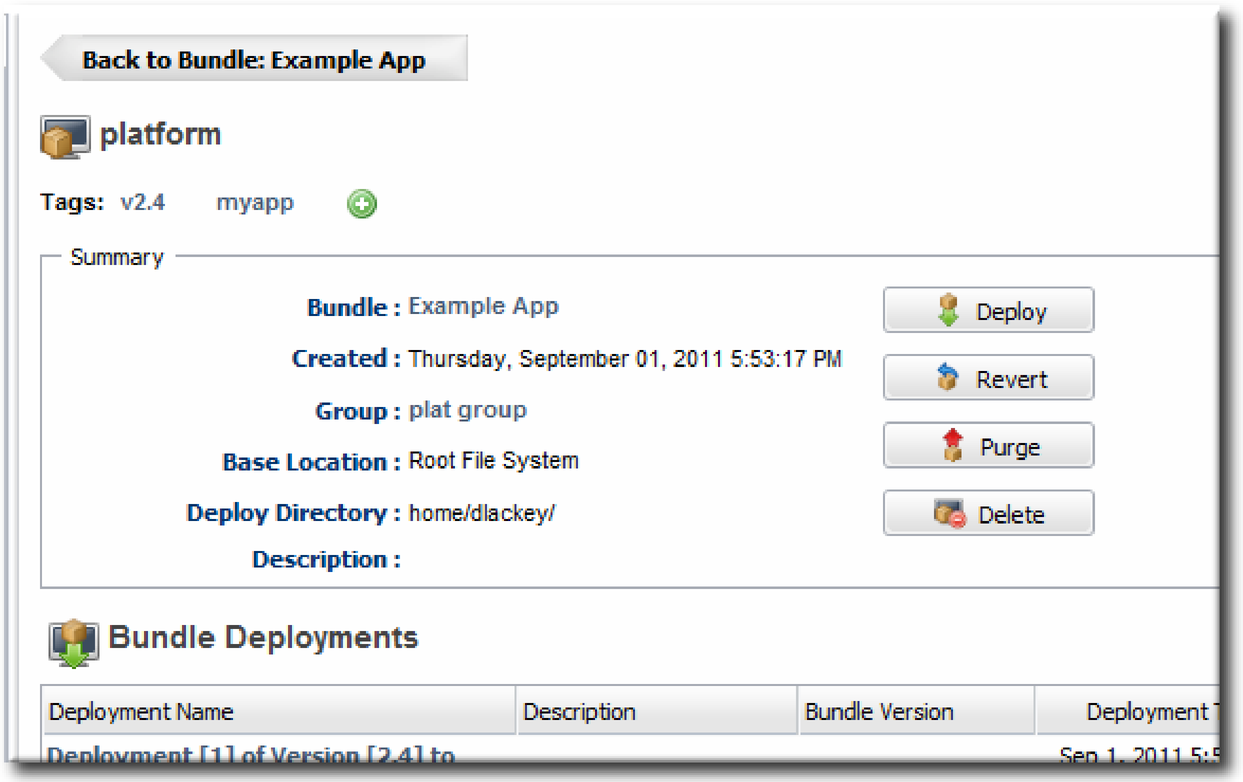
Figure 6. Deployment History for a Destinations

2.4.4. Reverting a Deployed Bundle

Ant bundles can be rolled back to a previous version number or a previous deployment of that bundle. This provides some extra protection and flexibility when deploying and managing applications, particularly for testing and production systems.

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

2. In the left navigation window, expand the bundle node, and then open the **Destinations** folder beneath it.
3. Select the destination from the left navigation.
4. In the main window for the destination, click the **Revert** button.



Summary

Bundle : Example App

Created : Thursday, September 01, 2011 5:53:17 PM

Group : plat group

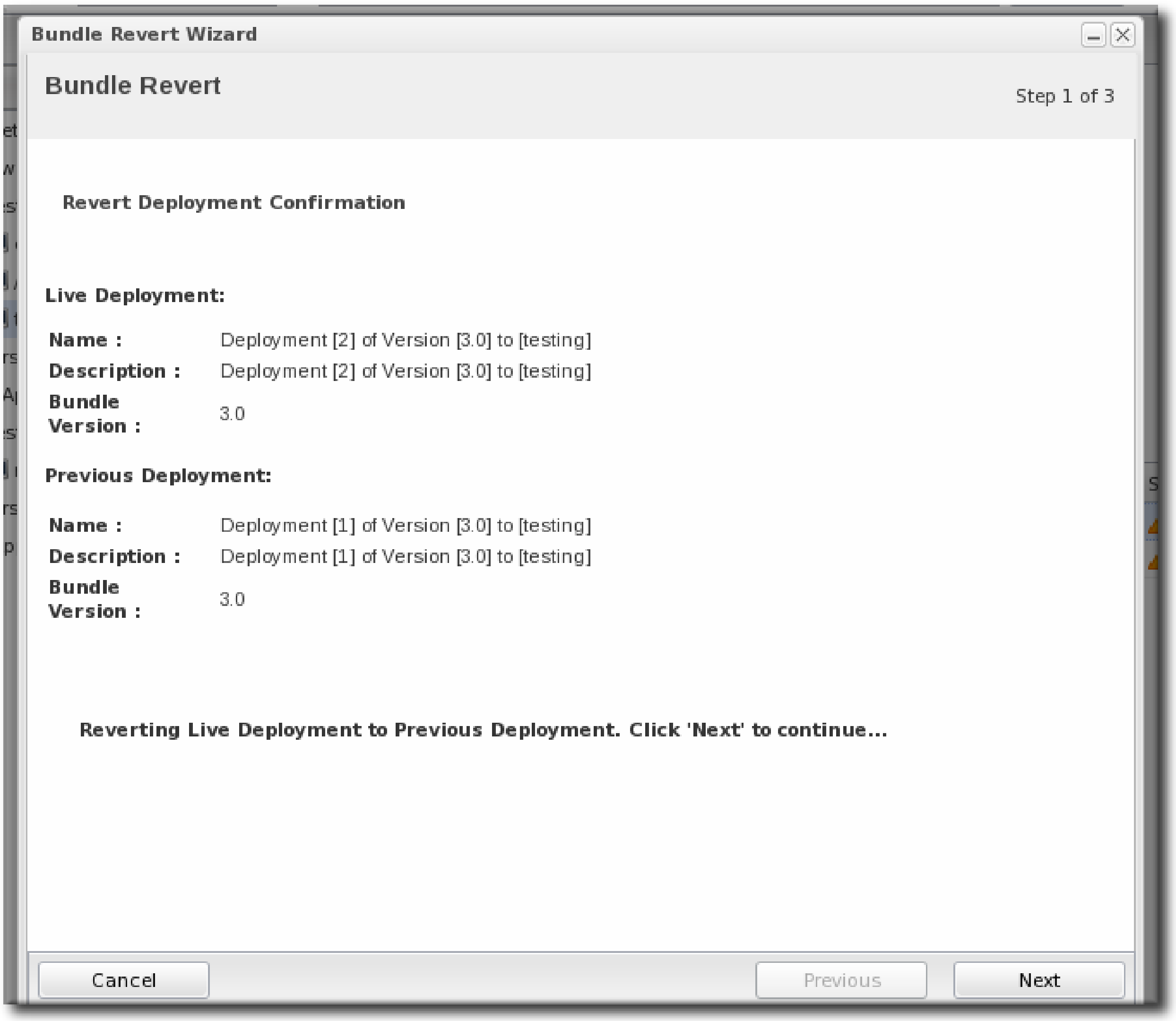
Base Location : Root File System

Deploy Directory : home/dlackey/

Description :

Deployment Name	Description	Bundle Version	Deployment T
Deployment [1] of Version [2.4] to			Sep 1, 2011 5:5

5. The next page shows the summary of the current deployment and the immediate previous deployment, which it will be reverted to.



Revert Deployment Confirmation

Live Deployment:

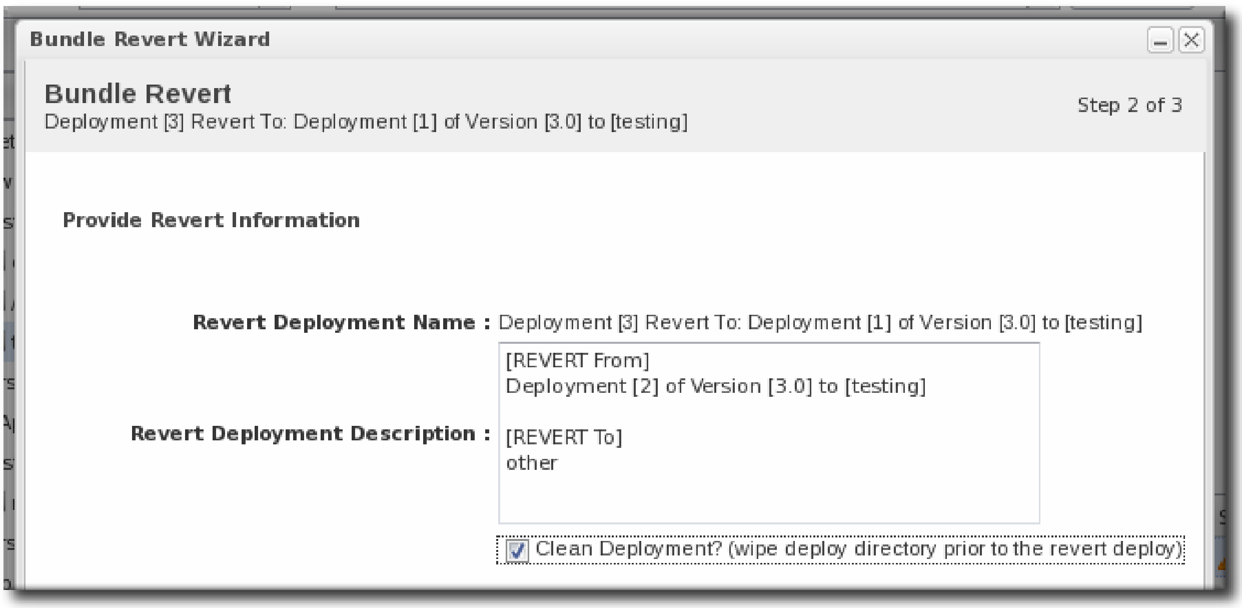
Name :	Deployment [2] of Version [3.0] to [testing]
Description :	Deployment [2] of Version [3.0] to [testing]
Bundle Version :	3.0

Previous Deployment:

Name :	Deployment [1] of Version [3.0] to [testing]
Description :	Deployment [1] of Version [3.0] to [testing]
Bundle Version :	3.0

Reverting Live Deployment to Previous Deployment. Click 'Next' to continue...

6. Add any notes to the revert action. Optionally, select the checkbox to clean the deployment directory and install the previous version fresh.



Bundle Revert
Deployment [3] Revert To: Deployment [1] of Version [3.0] to [testing]
Step 2 of 3

Provide Revert Information

Revert Deployment Name : Deployment [3] Revert To: Deployment [1] of Version [3.0] to [testing]

Revert Deployment Description : [REVERT From]
Deployment [2] of Version [3.0] to [testing]

[REVERT To]
other

[✓] Clean Deployment? (wipe deploy directory prior to the revert deploy)

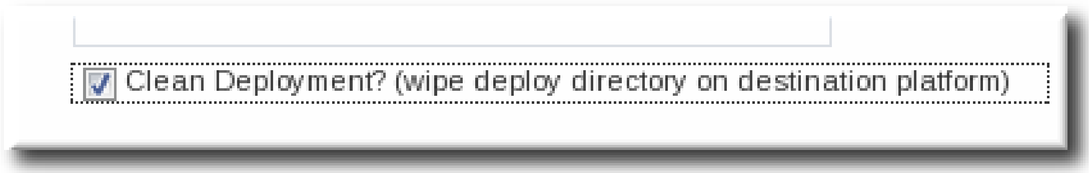
7. Click **Finish** on the final screen to complete the rollback.

2.4.5. Deploying a Bundle to a Clean Destination

A bundle can be deployed to a destination where there may already be an application, files, or even a previous bundle deployment. When deploying a new bundle, there are two options for how the provisioning process handles the update:

- Preserve the existing files and directories, with appropriate upgrades, according to the recipe configuration (Section 2.3.2.2, “Saving Files During Provisioning”)
- Completely overwrite the existing files and deploy the bundle in an empty directory

To deploy the bundle in a clean directory, then select the **Clean Deploy** checkbox when running through the deployment wizard in Section 2.4.2, “Deploying Bundles to a Resource”.


[✓] Clean Deployment? (wipe deploy directory on destination platform)

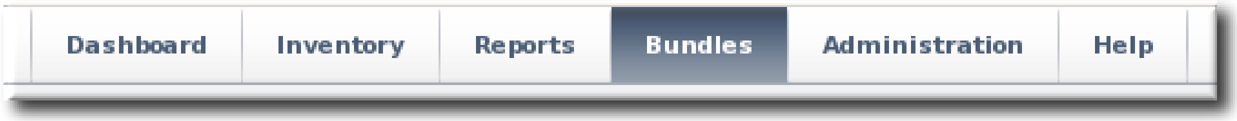
2.4.6. Purging a Bundle from a Resource

Purging a bundle removes all of the files associated with the bundle from all of the target resources. However, this does not remove the bundle from the JBoss ON database, so it can be easily re-deployed to the same resources later or to other resources.

Important

The exact files that are purged mirrors how the bundle manages the deployment directory. By default, purging includes deleting the deployment directory (**`manageRootDir=true`**). If the deployment directory is used by other applications – like an app server **`deploy/`** directory — then those other applications or files will also be deleted. After purging, there is no live deployment and nothing to revert.

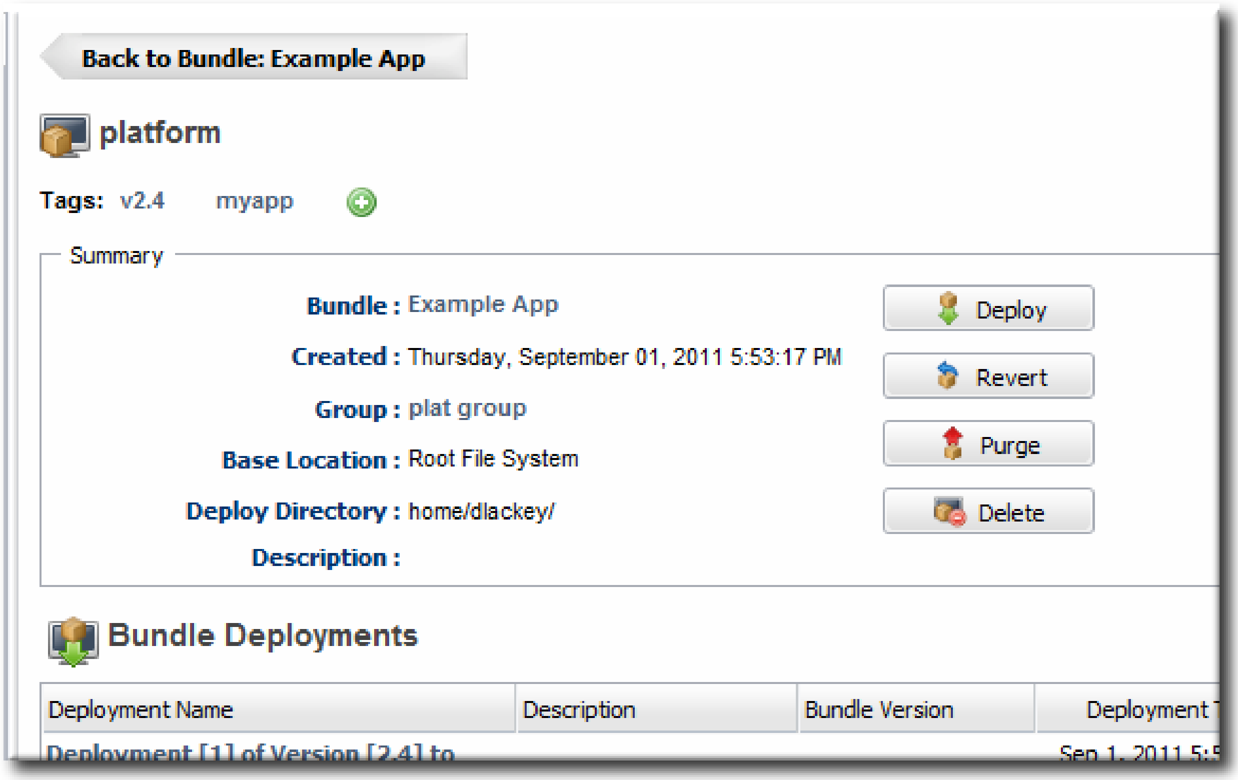
1. In the top menu, click the **`Bundles`** tab.



2. In the left navigation window, expand the bundle node, and then open the **`Destinations`** folder beneath it.

3. Select the destination from the left navigation.

4. In the main window for the destination, click the **`Purge`** button.



platform

Tags: v2.4 myapp

Summary

Bundle : Example App

Created : Thursday, September 01, 2011 5:53:17 PM

Group : plat group

Base Location : Root File System

Deploy Directory : home/dlackey/

Description :

Bundle Deployments

Deployment Name	Description	Bundle Version	Deployment T
Deployment [1] of Version [2.4] to			Sep 1, 2011 5:5

5. When prompted, confirm that you want to remove the bundled application and configuration from the target resources.

2.4.7. Upgrading Ant Bundles

The bundle upgrade process decides whether to upgrade (meaning, overwrite) files within the application's deployment directory by comparing the MD5 hash codes on the files. There are several different upgrade scenarios:

- If the hash code on the new file is different than the original file and there are no local modifications, then JBoss ON installs the new file over the existing file.
- If the hash code on the new file is different than the original file and there *are* local modifications, then JBoss ON backs up the original file and installs the new file.
- If the hash code on the new file and the original file is the same and there *are* local modifications on the original file, then the provisioning process preserves the original file, in place.
- If there was no file in the previous bundle but there is one in the new bundle, then the new file is used and any file that was added manually is backed up.

Backed up files are saved to a **`backup/`** directory within the deployment's destination directory. If the original file was located outside the application's directory (like, it was stored according to an absolute location rather than a relative location), then it is saved in an **`ext-backup/`** directory within the deployment's destination directory.

Note

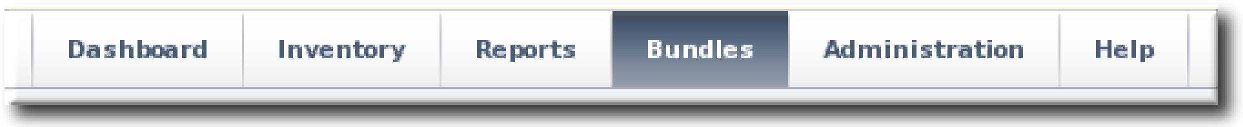
If a file is ignored in the recipe, then the file is left unchanged. *Never* ignore files packaged in the bundle. Only files generated by the applications, such as log and data files, should be ignored by the provisioning process since they should be preserved for the upgraded instance.

If a completely fresh installation is required, then it is possible to run a clean deployment. This is described in Section 2.4.5, “Deploying a Bundle to a Clean Destination”.

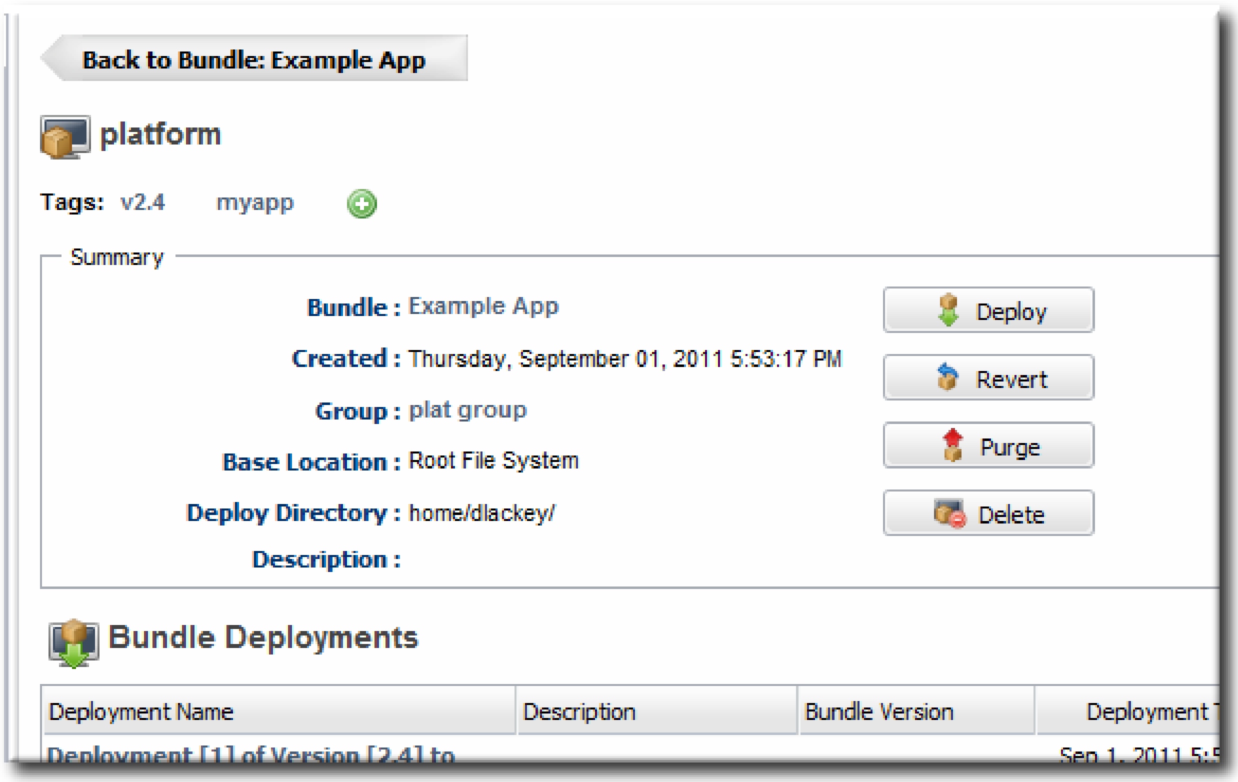
2.4.8. Deleting a Bundle from the JBoss ON Server

Deleting a bundle removes all of its recipes and associated files from the JBoss ON database. The deployed applications or configuration remain intact on the target resources.

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



2. In the left navigation window, expand the bundle node, and then open the **`Destinations`** folder beneath it.
3. Select the destination from the left navigation.
4. In the main window for the destination, click the **`Delete`** button.


Back to Bundle: Example App

platform

Tags: v2.4 myapp

Summary

Bundle : Example App
Created : Thursday, September 01, 2011 5:53:17 PM
Group : plat group
Base Location : Root File System
Deploy Directory : home/dlackey/
Description :

Bundle Deployments

Deployment Name	Description	Bundle Version	Deployment T
Deployment [1] of Version [2.4] to			Sep 1, 2011 5:5

5. When prompted, confirm that you want to delete the bundle.

2.5. Bundles and JBoss ON Servers and Agents

2.5.1. Resource Support and the Agent Resource Plug-in

Whether provisioning is supported is defined in the resource type. For a resource type to allow provisioning, the resource plug-in descriptor must defined a *bundle target*. That is the indication to the agent the provisioning is supported.

The **`<bundle-target>`** element simply defines allowed base directories for the resource which can be used as base directories in the bundle definition.

```
<server name="JBossAS:JBossAS Server" ...>
   <bundle-target>
      <destination-base-dir name="Library Directory" description="Where the
jar libraries are">
         <value-context>pluginConfiguration</value-context>
         <value-name>lib.dir</value-name>
      </destination-base-dir>
      <destination-base-dir name="Deploy Directory" description="Where the
deployments are">
         <value-context>pluginConfiguration</value-context>
         <value-name>deploy.dir</value-name>
      </destination-base-dir>
   </bundle-target>
</server>
```

Every resource plug-in descriptor defines a *base directory*, the root for all deployments, apart from provisioning configuration. For platforms, this is the root directory. For servers, it is usually the installation directory. The **`<bundle-target>`** can use the already-configured base directory or it can set different directories to use. In the example, two directories — the **`deploy/`** and

`lib/` directories — are given as supported base directories. When a bundle definition is created, the wizard offers the choice of which directory to use.

2.5.2. Server-Side and Agent Plug-ins for Recipe Types

By default, JBoss ON supports one type of recipe, an Ant build file. However, other types of recipes could be supported because the recipe type is defined in a pair of plug-ins, one for the server and one for the agent.

The server-side plug-in tells the JBoss ON server how to manage bundles and destinations for that type of recipe.

The agent plug-in creates a child resource for the platform which is used to perform provisioning operations on the platform or target resource. For example, Ant bundles are actually deployed by the special JBoss ON resource, *Ant Bundle Handler*. This resource is added automatically to platforms as a child resource to enable Ant-based provisioning.

Note

Since recipe type support is implemented on the agent side through a special resource, that resource must exist in the JBoss ON inventory for it to perform provisioning. For example, without the Ant bundle handler in the inventory for a platform, JBoss ON cannot perform provisioning on that platform.

Administrators do not have to interact directly with the Ant bundle handler resource, but that child resource must be present in the platform's inventory for Ant provisioning to work.

2.6. Managing and Deploying Bundles with the JBoss ON CLI

Both uploading bundles to JBoss ON and deploying bundles to resources can be performed using the JBoss ON CLI.

The ability to script bundle deployments is very powerful, because it allows content or configuration updates, even new application servers, to be deployed automatically based on activity in other resources across JBoss ON. This is particularly useful with using JBoss ON CLI scripts in response to an alert:

- A new JBoss application server can be deployed when an existing JBoss server experiences a heavy load or decreased performance.
- Configuration files for a selected snapshot image can be immediately deployed to a platform or JBoss server to remedy configuration drift, in response to a drift alert.
- A new web context can be deployed when another web is disabled within a **`mod_cluster`** domain.

Scripting also allows updates to be applied on schedule, such as having daily or weekly scheduled updates to a QE environment — which is also useful because the bundle content can be pulled from a GIT or SVN repository used by a build system first, and then deployed for testing.

The bundles API is in the Javadocs at https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Operations_Network/3.1/html/API/ch01.html.

3. Managing Resource-Level Content Updates

JBoss Operations Network can be used to store and deploy content to resources. This can be done to apply updates and patches (as with JBoss AS servers) or to set up repositories used for provisioning applications and deploying custom software.

3.1. About Content

Content for a resource can be almost anything, such as WAR and EAR files, configuration files, or scripts. JBoss ON provides a central framework to associate content, repositories, and resources in the inventory.

3.1.1. What Content Is: Packages

A *package* is anything that is installed on a platform or for a server or application. This can be an RPM, a JAR file, or even just a configuration file. A package simply provides some form of content for a resource. Packages can be sent to a resource through a JBoss ON-recognized repository or simply by uploading the package to the JBoss ON server and then sending it to the resource.

A resource can only be associated with or manage content if the resource plug-in identifies that content is available and the type of content that is supported. For example, application and web servers like JBoss AS/EAP and Tomcat support EAR, WAR, and JAR files as content; platforms support content like RPMs; but a database like PostgreSQL does not support any content types.

In a sense, content is both the software bits, scripts, or configuration files associated with a resource and also a resource itself. When content is added to a resource, it becomes a child resource in the JBoss ON hierarchy — but it can be managed, reverted, updated, or replaced by uploading new software bits. The parent resource (such as the application server) supports content; the child resource is a *content backed resource*.

Content can be added to a resource either by manually creating a child resource (and uploading the packages) or by adding the package to a content source and deploying it to the parent resource. The agent can also actively *check* for new content as part of its discovery scan and add any discovered content to its inventory. The agent's recurring package discovery scan has a default interval of 24 hours, as with the services scan.

3.1.2. Where Content Comes From: Providers and Repositories

Content sources are developers and distributors of content. Sources can be external third party software developers or internal development teams that create custom content. The type of content available from sources includes both software packages (such as RPMs or configuration scripts) and updates (version upgrades, patches, and errata).

A *repository* is a user-defined collection of software packages, which can come from one or multiple content sources. A repository may contain packages for an application or family of applications or for a specific purpose, like repositories for laptop configuration and repositories for installing web servers.

Repositories aren't siloed, separate containers for packages; they are essentially views that show a subset of available packages. All packages are stored in the JBoss ON database. A JBoss ON repository is a way of grouping those packages, both to make it easier to administer with resources and to provide a mechanism of access control for users (Section 3.1.4, “Authorization to Repositories and Packages”).

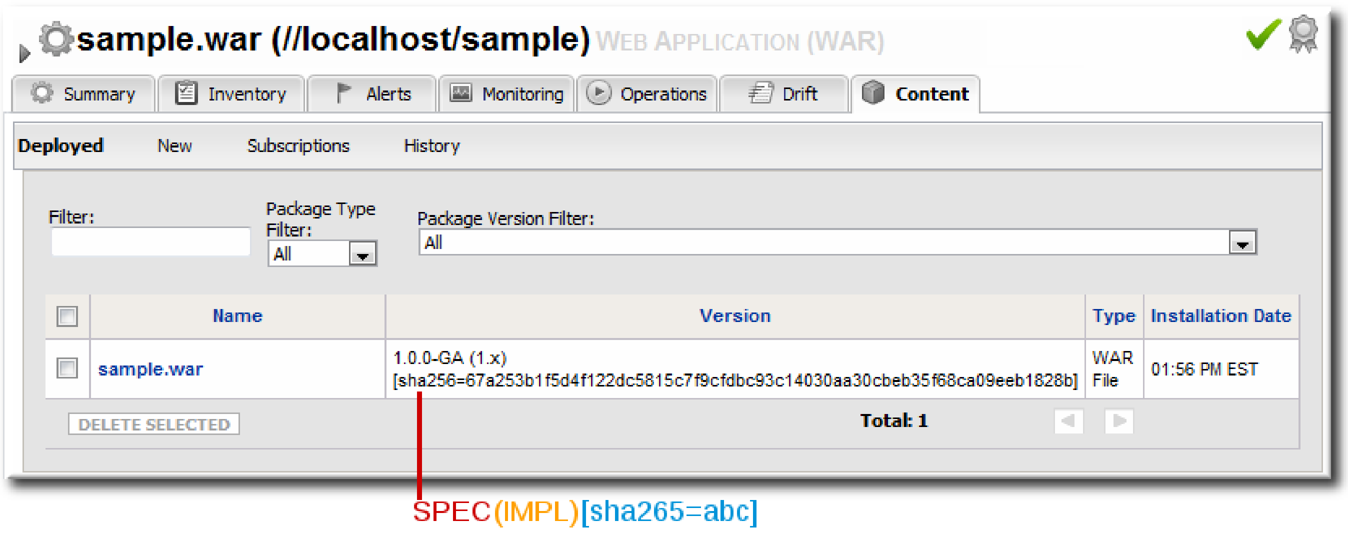
Resources can be subscribed to content repositories that are configured in JBoss ON, which provides a smooth and reliable mechanism for delivering consistent, administrator-configured content to resources.

3.1.3. Package Versions and History

Packages are *versioned* within JBoss ON itself. When a package is added to a resource or content source, the installer prompts for a version number; this is used as the UI display number.

This display version number is not required; if it is not given, then the JBoss ON server derives a number based on a calculated SHA-256 checksum for the package and the specification version and the implementation version in the **META-INF/MANIFEST.MF** file (for EARs and WARs).

```
SPEC(IMPLMENTATION)[sha256=abcd1234]
```



Filter:

Package Type Filter: All

Package Version Filter: All

	Name	Version	Type	Installation Date
	sample.war	1.0.0-GA (1.x) [sha256=67a253b1f5d4f122dc5815c7f9cfdbc93c14030aa30cbeb35f68ca09eeb1828b]	WAR File	01:56 PM EST

Total: 1

Figure 7. Package Version Numbers

For example, for a **META-INF/MANIFEST.MF** file with these version numbers:

```
Manifest-Version: 1.0
Created-By: Apache Maven
Specification-Title: My Example App
Specification-Version: 1.0.0-GA
Specification-Vendor: Example, Corp.
Implementation-Title: My Example App
Implementation-Version: 1.x
Implementation-Vendor-Id: org.example
Implementation-Vendor: Example, Corp.
...
```

This creates a version number for the package like this:

```
1.0.0-GA(1.x)[sha256=abcd1234]
```

If the **META-INF/MANIFEST.MF** file does not contain one of the specification version or the implementation version, then only one is used. For example, if only the implementation version is given:

```
(1.x)[sha256=abcd1234]
```

If no version number is given, then the SHA is used as the identifier. (The SHA is used as the identifier internally, anyway.)

```
[sha256=abcd1234]
```

For exploded WARs and EARs, the calculated SHA-256 checksum is added to the **MANIFEST.MF** file. This allows the agent to check the file during discovery scans to verify the version of the package quickly.

```
Manifest-Version: 1.0
Created-By: Apache Maven
RHQ-Sha256: 570f196c4a1025a717269d16d11d6f37
...
```

For unexploded (archived) content, the checksum is recalculated with every package discovery scan and compared to the checksum in inventory.

Note

Exploded WARs and EARs can be deployed on JBoss and Tomcat servers. Because the content deployment process edits the **META-INF/MANIFEST.MF** file, the deployed content is not exactly identical to the content packages that were uploaded.

A clear versioning system makes it possible to handle package lifecycles in a clear and effective way. Updated content can be tracked as it is deployed, updates can be applied consistently, and packages can be reverted to a previous version. The same repository can also contain different versions of the same package, making it possible to apply different versions to different resources.

Note

Package versions from different content sources can be associated with the same repository.

Whenever a package is installed on a resource, it is recorded in the content history for the resource and the package. Since there can be multiple files associated with a single package, then there can be multiple files recorded in the content history, all associated with that package version.

Note

Versioning only matters to content knit with a resource, like EARs and WARs. Other types of content stored in content sources (like CLI scripts used for alerting) do not track versions. Content deployed in bundles handles versioning through the bundle definition, not the content system.

3.1.4. Authorization to Repositories and Packages

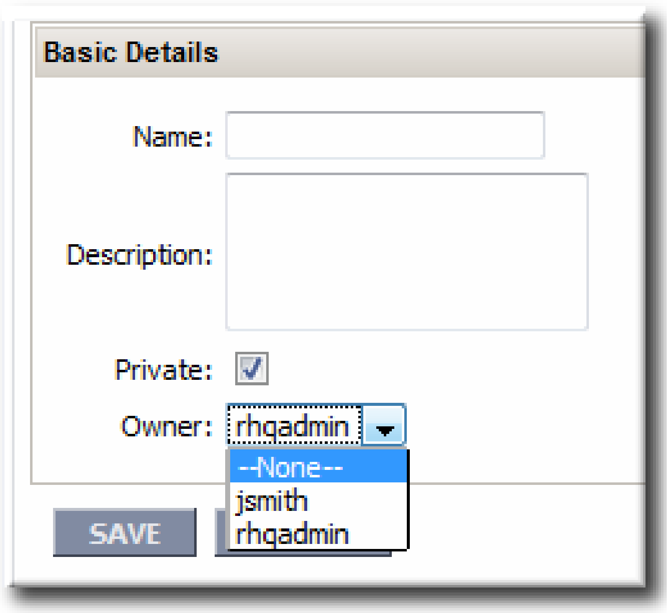
There are a lot of reasons that users need to be able to access content in repositories. The most common is to manage packages on resources, but there are other reasons, too, like using the server CLI scripts in a repository to respond to alerts.

JBoss ON provides a way to balance the need for clear and simple access to content with the need to protect private or sensitive information. JBoss ON defines clear *authorization* rules for content repositories.

Every user has the ability to create repositories and to upload packages to them — regardless of the permissions for that user.

When a repository is created, there are settings which control access to them:

- *Owner* sets write access to a repository. It assigns the repository to belong to a specific user. If no user is specified, then only users with the manage repositories permission have the right to access those repositories.
- *Private* sets read (download) access to the repository. It sets whether the repository can be viewed by anyone or only by the owner and users with the manage repositories permission. Public repositories are viewable by everyone, regardless of the owner.



Basic Details

Name:	
Description:	
Private:	✓
Owner:	rhqadmin

Figure 8. Repository Ownership and Access Settings

Repo managers (users with the manage repositories permission) can change the ownership and privacy settings of a repository. Users without the manage repositories permission can change the privacy settings but they cannot change the ownership; the repository is always owned by them or managed by the repo manager.

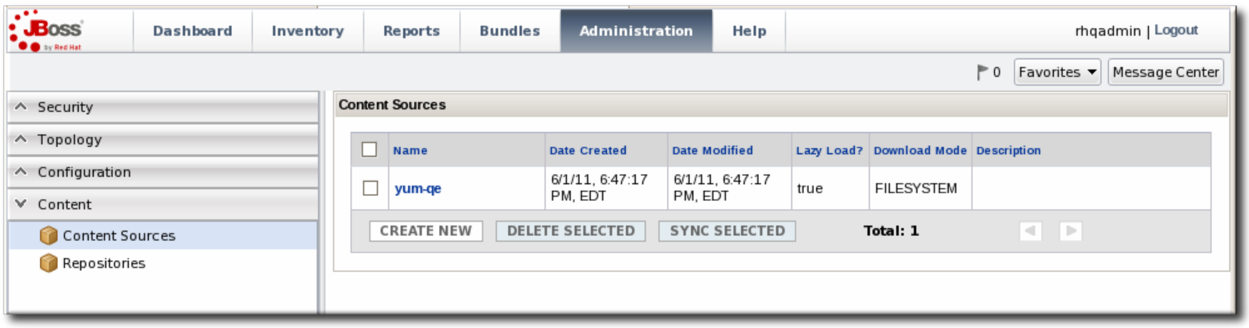
Note

Be very careful when switching public repositories to private. Any operations which relied on those repositories, such as running server CLI scripts in response to alerts, will no longer work if the privileges of the user are insufficient to access the repository.

JBoss ON uses the *repositories* access control permission to define users with administrative access to repositories. Any user with that permission can manage any configured repository, regardless of who the repository's owner is. Repositories without an owner can only be managed by users with the repositories permission. Lastly, only users with this permission can associate a content source with a repository; all other users must add packages to the repository manually.

3.2. Creating a Content Source

1. In the top menu, click the **`Administration`** tab.
2. In the **`Content`** menu table on the left, select the**Content Sources** item.
3. Below the list of current content sources, click the **`CREATE NEW`** button.

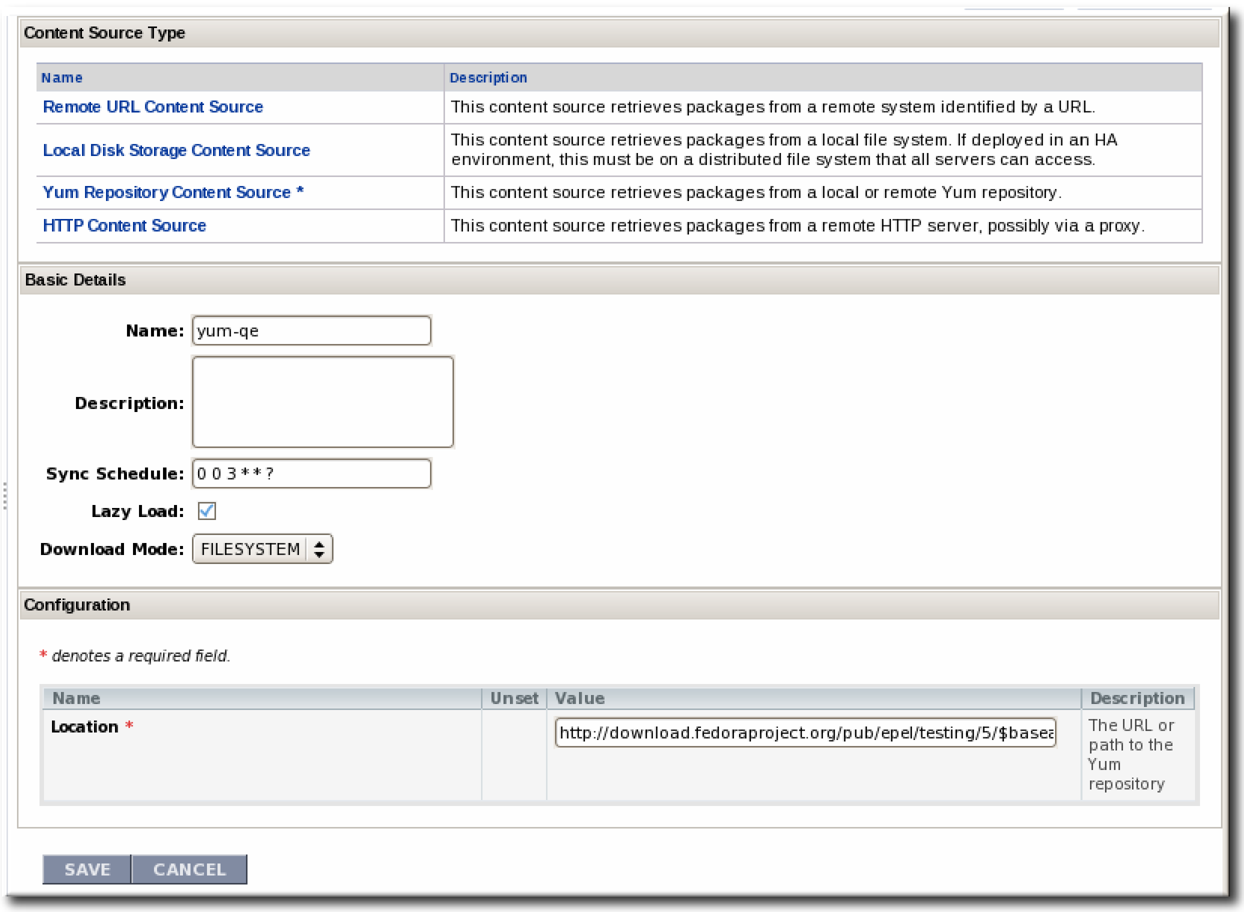


Content Sources

	Name	Date Created	Date Modified	Lazy Load?	Download Mode	Description
☐	yum-qe	6/1/11, 6:47:17 PM, EDT	6/1/11, 6:47:17 PM, EDT	true	FILESYSTEM	

Total: 1

4. Select the content source type, which defines how the content is delivered from the source. A content source type can be a remote URL, an HTTP server, a yum repository, or local disk.
5. When the content source type is selected, a form automatically opens to fill in the basic details and configuration for the resource. These basic details identify the content source in the JBoss ON server and are the same for each content source type, while the configuration is specific to the content source type.



Content Source Type

Name	Description
Remote URL Content Source	This content source retrieves packages from a remote system identified by a URL.
Local Disk Storage Content Source	This content source retrieves packages from a local file system. If deployed in an HA environment, this must be on a distributed file system that all servers can access.
Yum Repository Content Source *	This content source retrieves packages from a local or remote Yum repository.
HTTP Content Source	This content source retrieves packages from a remote HTTP server, possibly via a proxy.

Basic Details

Name:	yum-qe
Description:	
Sync Schedule:	0 0 3 * * ?
Lazy Load:	✓
Download Mode:	FILESYSTEM

Configuration

* denotes a required field.

Name	Unset	Value	Description
Location *		http://download.fedoraproject.org/pub/epel/testing/5/$basea	The URL or path to the Yum repository

- Give a unique name and optional description for the content source provider.
- The schedule sets how frequently the content in the JBoss ON database is updated by the content source; this uses a standard Quartz Cron Syntax.
- The lazy load setting sets whether to download packages only when they are installed (**Yes**) or if all packages should be download immediately.
- The download mode sets how the content is stored in JBoss ON. The default is **DATABASE**, which stores all packages in the JBoss ON database instance. The other options are to store the packages on a network filesystem or not to store them at all.

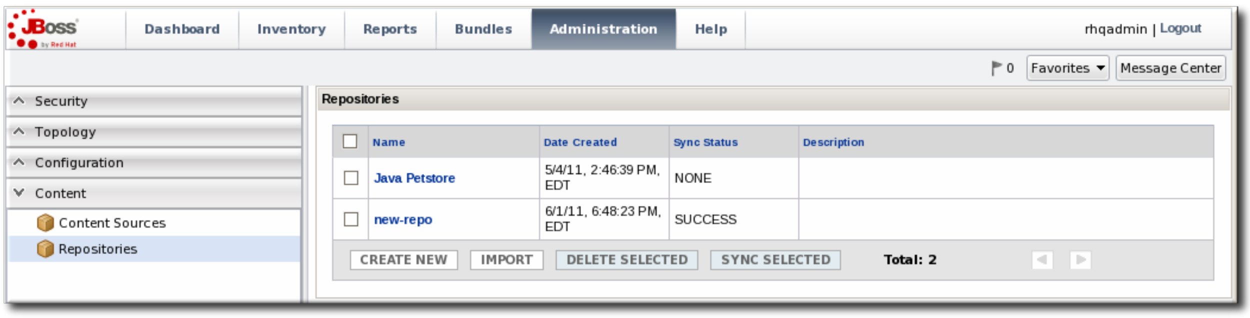
6. Fill in the other configuration information for the content source. The required information varies depending on the content source type. This is going to require some kind of connection information, such as a URL or directory path, and possibly authentication information, like a username and password.

3.3. Managing Repositories

A repository is essentially a mapping between the data in a content source and specific resources in the JBoss ON inventory.

3.3.1. Creating a Repository

1. In the top menu, click the **Administration** tab.
2. In the **Content** menu table on the left, select the**Repositories** item.
3. Below the list of current repositories, click the **CREATE NEW** button.



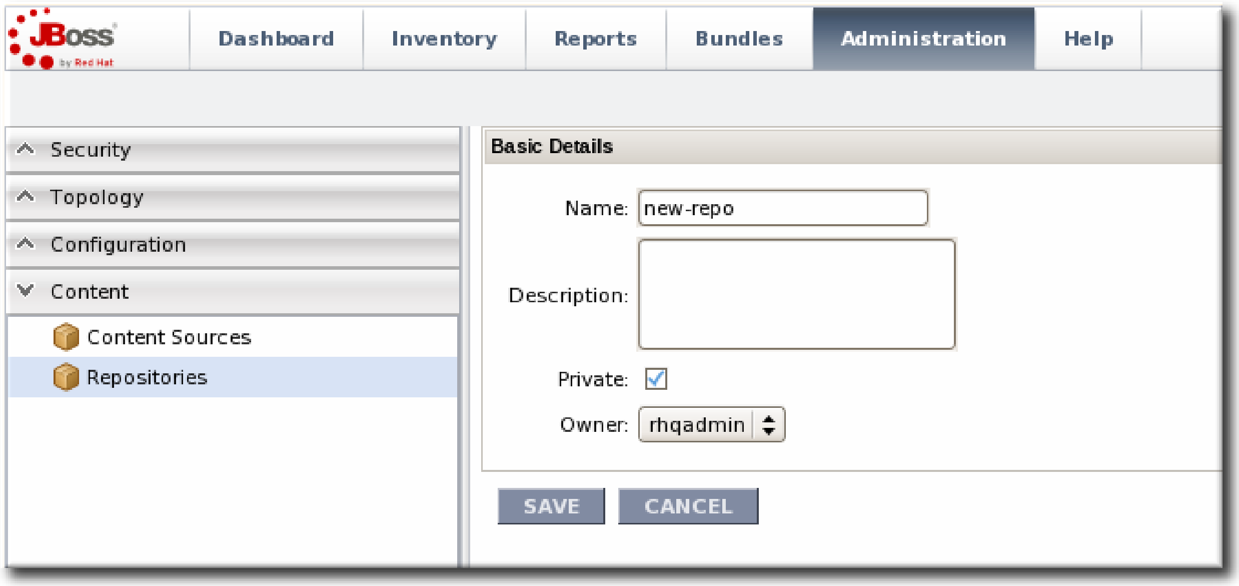
Repositories

	Name	Date Created	Sync Status	Description
☐	Java Petstore	5/4/11, 2:46:39 PM, EDT	NONE	
☐	new-repo	6/1/11, 6:48:23 PM, EDT	SUCCESS	

Total: 2

4. Fill in the name and a description. Additionally, set the authorization restrictions for the repository by setting an owner for the repo and whether it is public or private.

   Only users with the repositories permission can set an owner. All repositories created by users without the repositories permission automatically belong to that user.



Basic Details

Name: new-repo

Description:

Private: ✓

Owner: rhqadmin

5. Click **Save**.

6. On the **Repositories** page, click the name of the new repository in the list.

7. *Optional*. To change the default synchronization schedule, click the**Edit** button and enter a new schedule, in a cron format, in the **Sync Schedule** field.

8. Add content sources to supply content to the repository, as in Section 3.3.2.1, “Associating Content Sources with a Repository”.

   More than one content source can supply content to a repository.

9. Associate resources with the repository, as in Section 3.3.3, “Associating Resources with the Repository”. A resource can only receive packages from a repository if it is associated with the repository.

Note

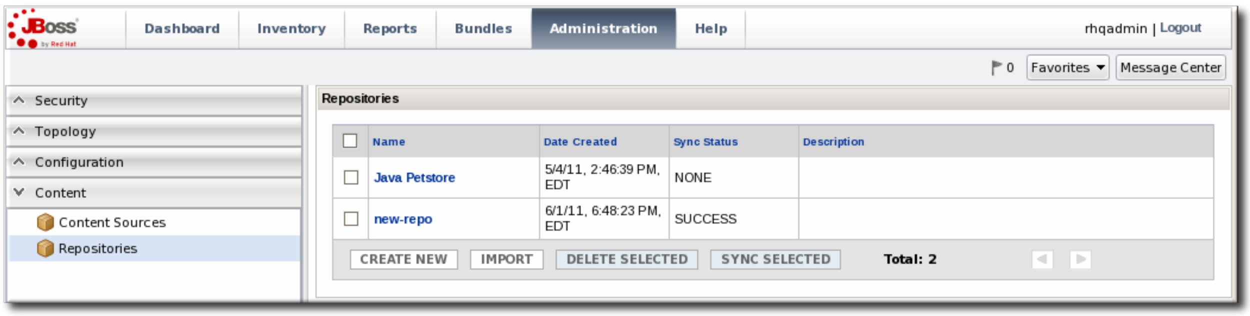
You can search for specific resources or types of resources and subscribe multiple resources at once.

3.3.2. Linking Content Sources to Repositories

There are a couple of ways to map the repositories to the right content sources. A repository can be subscribed to multiple content sources by editing the repository configuration. A content source can be added to multiple repositories simultaneously by importing the content source.

3.3.2.1. Associating Content Sources with a Repository

1. In the top menu, click the **Administration** tab.
2. In the **Content** menu table on the left, select the**Repositories** item.
3. On the **Repositories** page, click the name of the repository in the list.

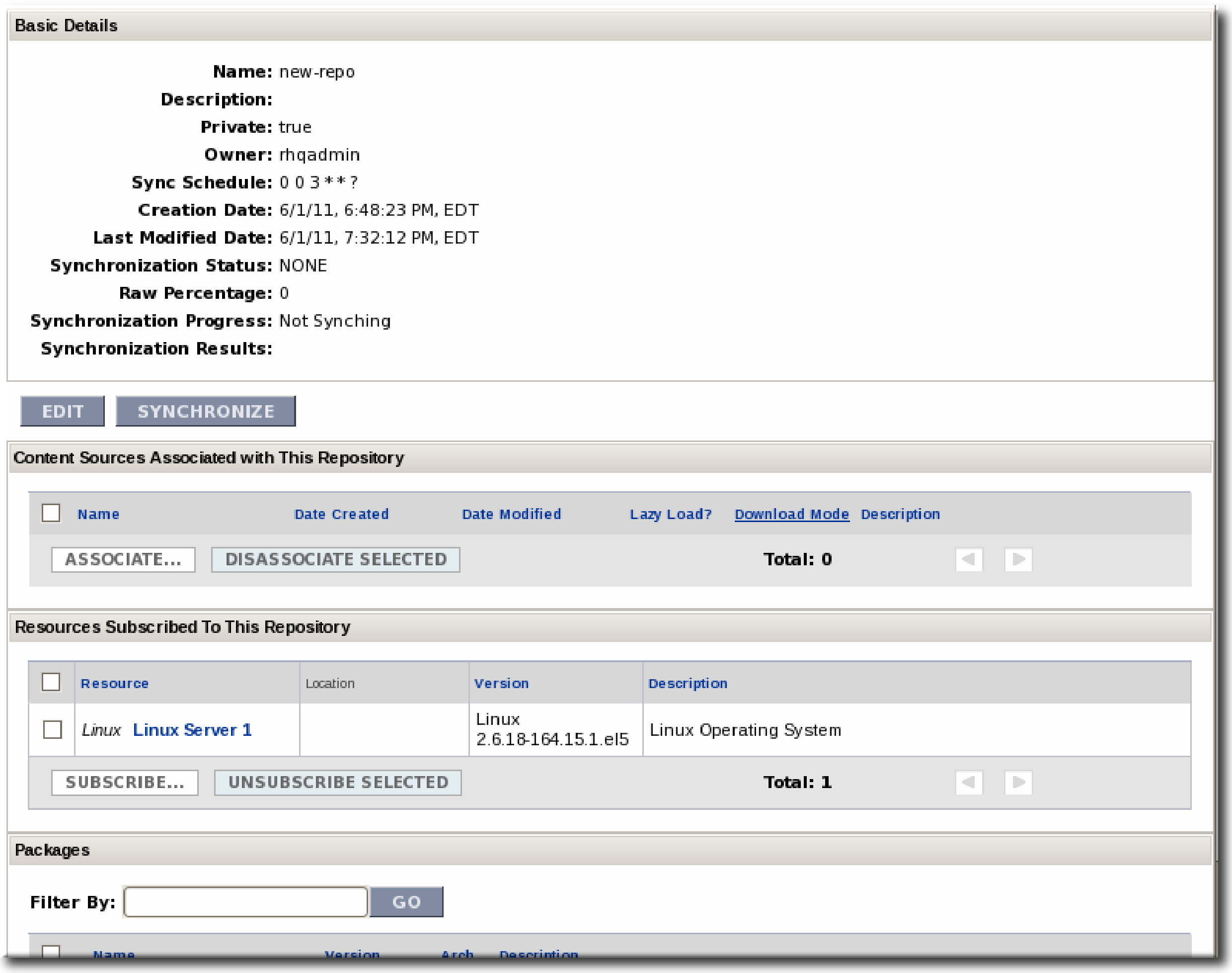


Repositories

	Name	Date Created	Sync Status	Description
☐	Java Petstore	5/4/11, 2:46:39 PM, EDT	NONE	
☐	new-repo	6/1/11, 6:48:23 PM, EDT	SUCCESS	

Total: 2

4. In the **Content Sources** section of the repository's details page, click the**Associate** button to add existing content sources to the repository.



Basic Details

Name:	new-repo
Description:	
Private:	true
Owner:	rhqadmin
Sync Schedule:	0 0 3 * * ?
Creation Date:	6/1/11, 6:48:23 PM, EDT
Last Modified Date:	6/1/11, 7:32:12 PM, EDT
Synchronization Status:	NONE
Raw Percentage:	0
Synchronization Progress:	Not Synching
Synchronization Results:	

Content Sources Associated with This Repository

	Name	Date Created	Date Modified	Lazy Load?	Download Mode	Description

Total: 0

Resources Subscribed To This Repository

	Resource	Location	Version	Description
☐	*Linux* Linux Server 1		Linux 2.6.18-164.15.1.el5	Linux Operating System

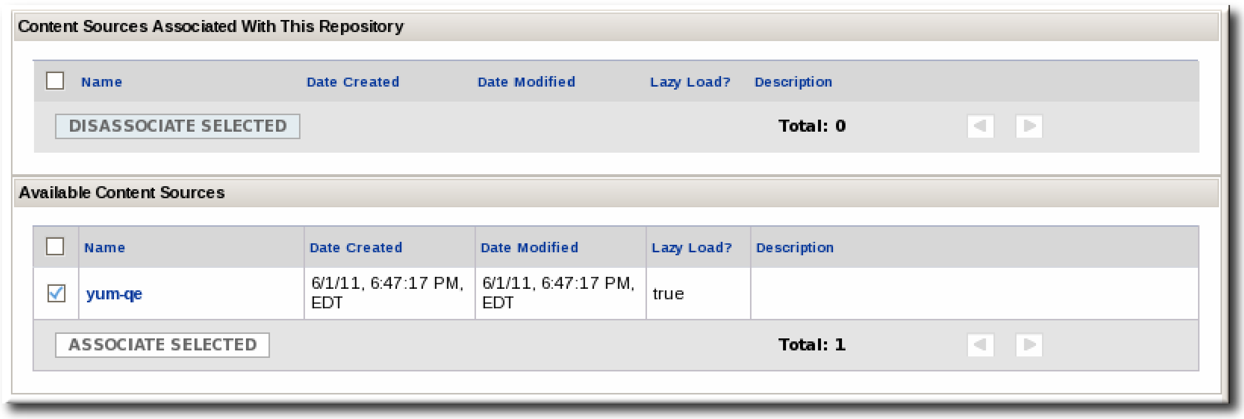
Total: 1

Packages

Filter By:

	Name	Version	Arch	Description

5. Select checkboxes next to the content sources to associate with the repository.


Content Sources Associated With This Repository

☐	Name	Date Created	Date Modified	Lazy Load?	Description

DISASSOCIATE SELECTED — Total: 0

Available Content Sources

☐	Name	Date Created	Date Modified	Lazy Load?	Description
✓	yum-qe	6/1/11, 6:47:17 PM, EDT	6/1/11, 6:47:17 PM, EDT	true	

ASSOCIATE SELECTED — Total: 1

6. Click the **`ASSOCIATE SELECTED`** button.

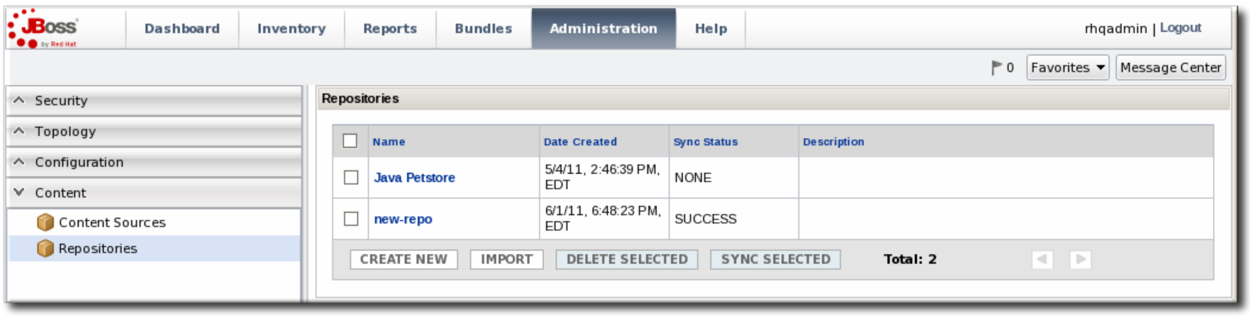
3.3.2.2. Importing a Content Source into Repositories

If the same content source will be associated with multiple repositories, the content source can be imported into all of them simultaneously.

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

2. In the **`Content`** menu table on the left, select the**`Repositories`** item.

3. On the **`Repositories`** page, click the **`IMPORT`** button.



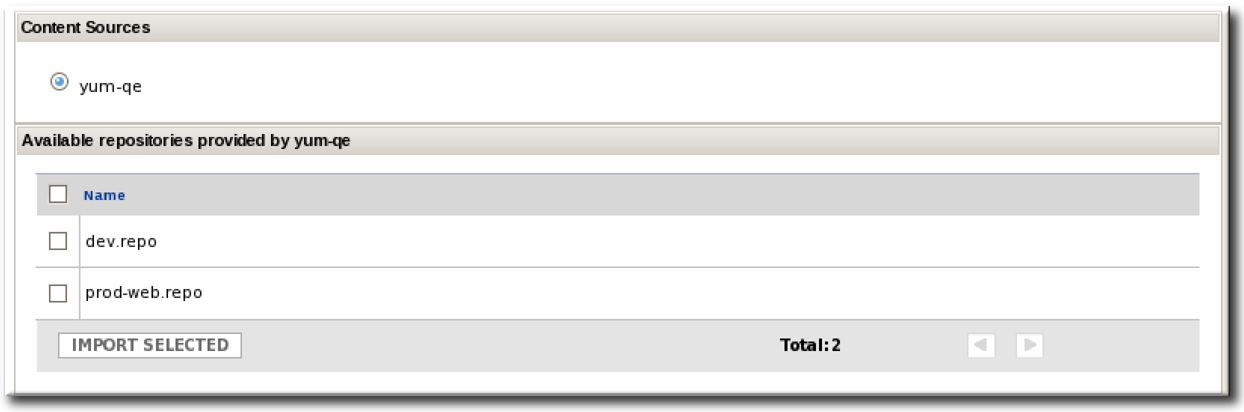
Repositories

☐	Name	Date Created	Sync Status	Description
☐	Java Petstore	5/4/11, 2:46:39 PM, EDT	NONE	
☐	new-repo	6/1/11, 6:48:23 PM, EDT	SUCCESS	

CREATE NEW — IMPORT — DELETE SELECTED — SYNC SELECTED — Total: 2

4. Select the radio button by the name of the content source to import.

5. When the content source is selected, then a list of available repositories for that content source automatically opens. In the **`Available repositories....`** area, select the checkbox by the name of each repository to associate with the content source.


Content Sources

● yum-qe

Available repositories provided by yum-qe

☐	Name
☐	dev.repo
☐	prod-web.repo

IMPORT SELECTED — Total: 2

6. Click the **`IMPORT SELECTED`** button.

Note

As described in Section 3.1.2, “Where Content Comes From: Providers and Repositories”, a repository is a user-defined view of a subset of packages stored in the JBoss ON database. A repository is not a separate container.

When adding a package to one repository through the UI, it may fail with an error claiming that the package already exists, even if the package isn't in the specified repository. This is because a package with the same name exists in *another* repository and it causes a collision in the database.

It is currently not possible to have the same package in two repositories or to move or share a package between repositories.

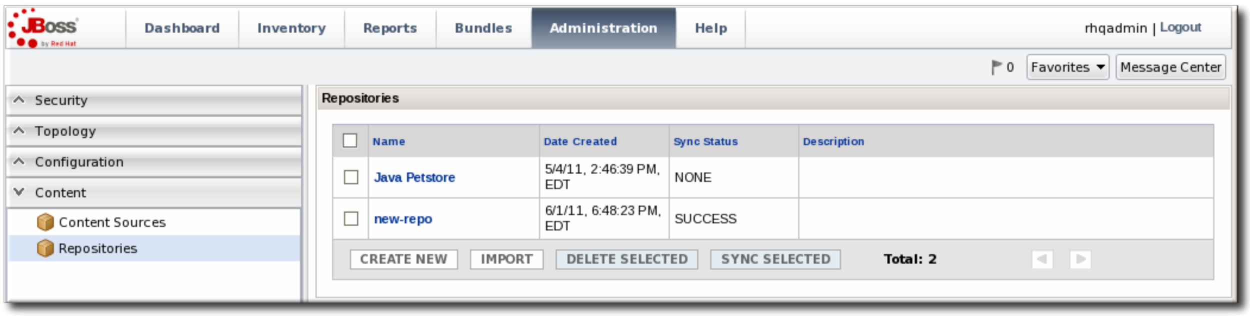
It is possible to work around this issue by using CLI scripts. The JBoss ON CLI scripts store the username of the person uploading the package in the package version data automatically. If a person has access to all of the packages one has uploaded, then it is possible to extrapolate which repository contains the package and then manage the package there.

3.3.3. Associating Resources with the Repository

Content can only be sent to a resource if that resource is first associated with a repository. A resource-repository association can be made by editing the resource entry or by editing the repository entry.

3.3.3.1. Adding Resources to a Repository

1. In the top menu, click the **`Administration`** tab.
2. In the **`Content`** menu table on the left, select the**Repositories** item.
3. On the **`Repositories`** page, click the name of the repository to edit.

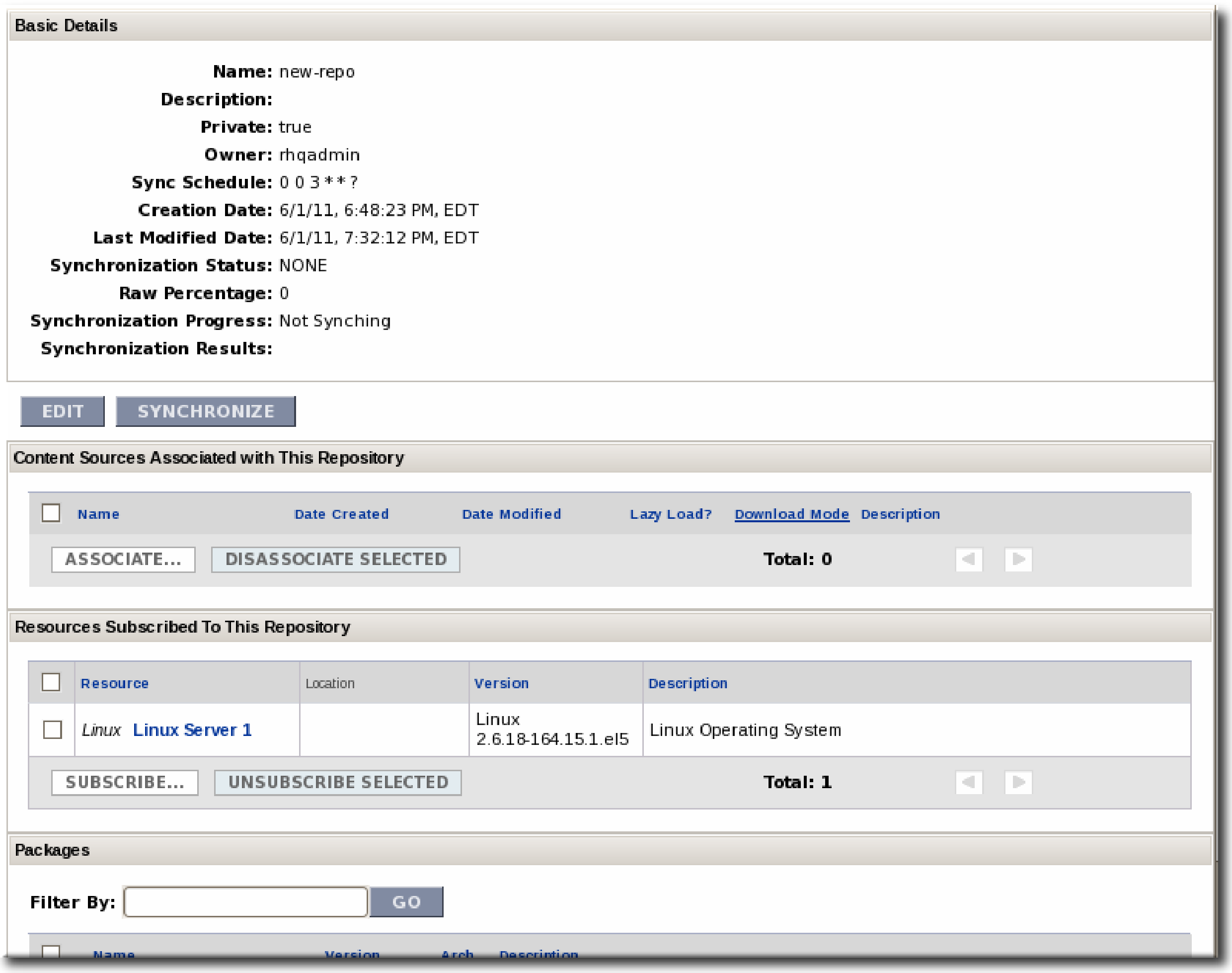


Repositories

	Name	Date Created	Sync Status	Description
☐	Java Petstore	5/4/11, 2:46:39 PM, EDT	NONE	
☐	new-repo	6/1/11, 6:48:23 PM, EDT	SUCCESS	

Total: 2

4. In the **`Resources`** section, click the **`SUBSCRIBE`** button to add resources to the repository.



Basic Details

Name:	new-repo
Description:	
Private:	true
Owner:	rhqadmin
Sync Schedule:	0 0 3 * * ?
Creation Date:	6/1/11, 6:48:23 PM, EDT
Last Modified Date:	6/1/11, 7:32:12 PM, EDT
Synchronization Status:	NONE
Raw Percentage:	0
Synchronization Progress:	Not Synching
Synchronization Results:	

EDIT SYNCHRONIZE

Content Sources Associated with This Repository

☐	Name	Date Created	Date Modified	Lazy Load?	Download Mode	Description

ASSOCIATE... DISASSOCIATE SELECTED **Total: 0**

Resources Subscribed To This Repository

☐	Resource	Location	Version	Description
☐	*Linux* Linux Server 1		Linux 2.6.18-164.15.1.el5	Linux Operating System

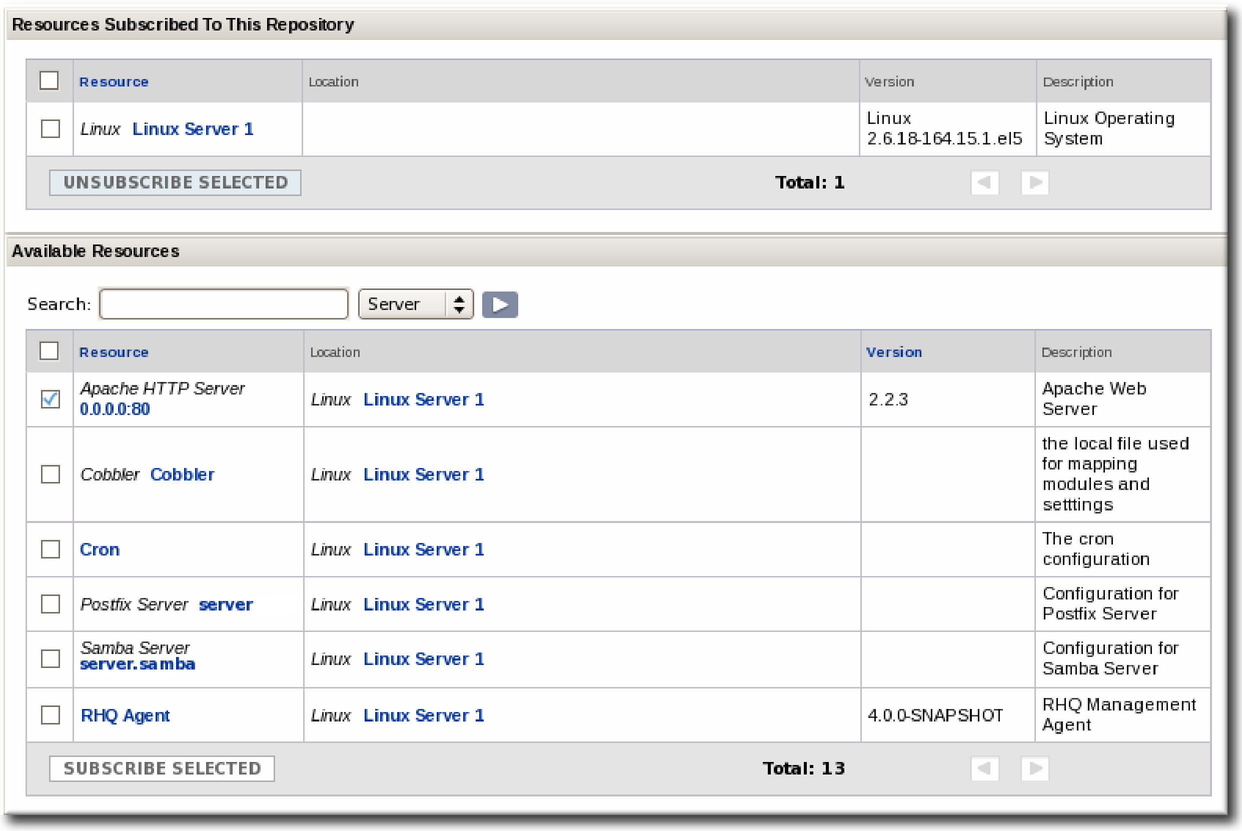
SUBSCRIBE... UNSUBSCRIBE SELECTED **Total: 1**

Packages

Filter By: GO

☐	Name	Version	Arch	Description

5. Select checkboxes next to the resources to associate with the repository. It is possible to filter the list of resources by name or by type.



Resources Subscribed To This Repository

☐	Resource	Location	Version	Description
☐	*Linux* Linux Server 1		Linux 2.6.18-164.15.1.el5	Linux Operating System

UNSUBSCRIBE SELECTED **Total: 1**

Available Resources

Search: Server

☐	Resource	Location	Version	Description
☑	*Apache HTTP Server* 0.0.0.0:80	*Linux* Linux Server 1	2.2.3	Apache Web Server
☐	*Cobbler* Cobbler	*Linux* Linux Server 1		the local file used for mapping modules and setttings
☐	Cron	*Linux* Linux Server 1		The cron configuration
☐	*Postfix Server* server	*Linux* Linux Server 1		Configuration for Postfix Server
☐	*Samba Server* server.samba	*Linux* Linux Server 1		Configuration for Samba Server
☐	RHQ Agent	*Linux* Linux Server 1	4.0.0-SNAPSHOT	RHQ Management Agent

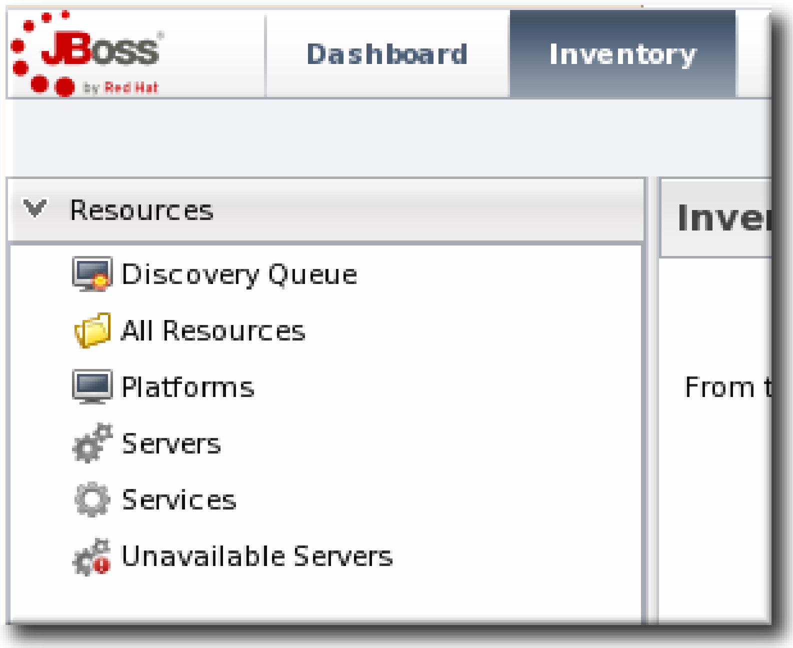
SUBSCRIBE SELECTED **Total: 13**

6. Click the **`SUBSCRIBE SELECTED`** button.

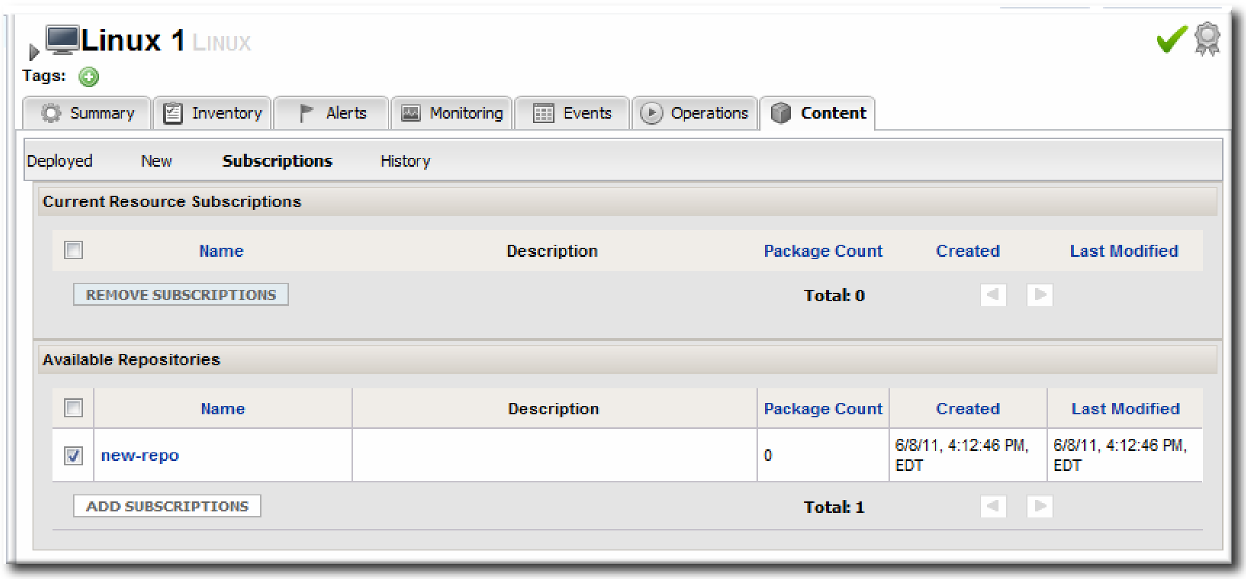
3.3.3.2. Managing the Repositories for a Resource

A few resource types, like platforms, have content tabs in their configuration which allows them to control their content subscriptions.

1. Select the resource type in the **Resources** menu table on the left, and then browse or search for the resource.



2. Click the **Content** tab of the resource.
3. Open the **Subscriptions** subtab.
4. The **Available Repositories** section has a list of repositories that the resource isn't subscribed to. Click the checkboxes by all of the repositories to subscribe the resource to.



Current Resource Subscriptions

☐	Name	Description	Package Count	Created	Last Modified

Total: 0

Available Repositories

☐	Name	Description	Package Count	Created	Last Modified
☑	new-repo		0	6/8/11, 4:12:46 PM, EDT	6/8/11, 4:12:46 PM, EDT

Total: 1

5. Click **ADD SUBSCRIPTIONS**.

The same process can be used to unsubscribe a resource from content repositories.

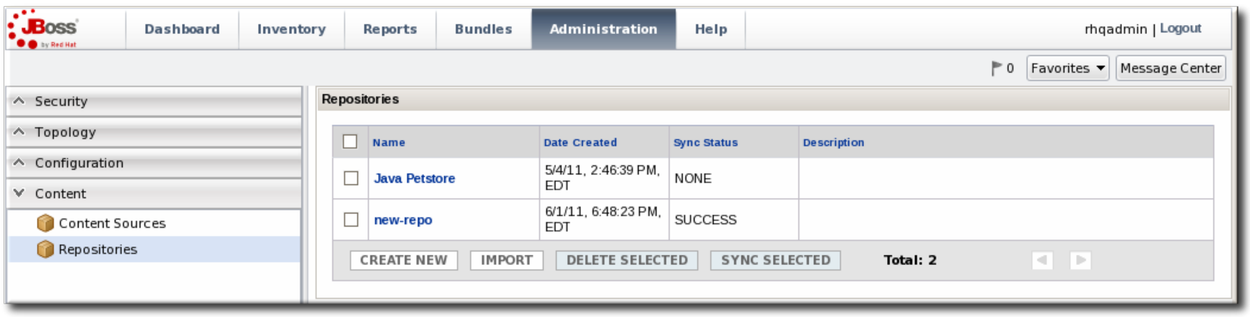
3.4. Uploading Packages

Packages can be pulled from a content source, but individual packages can also be uploaded directly to the JBoss ON server. A variety of package types are supported, including JAR files, RPMs, basic scripts, JBoss ON CLI scripts, and patches.

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

2. In the **`Content`** menu table on the left, select the**Repositories** item.

3. On the **`Repositories`** page, click the name of the repository in the list.



Repositories

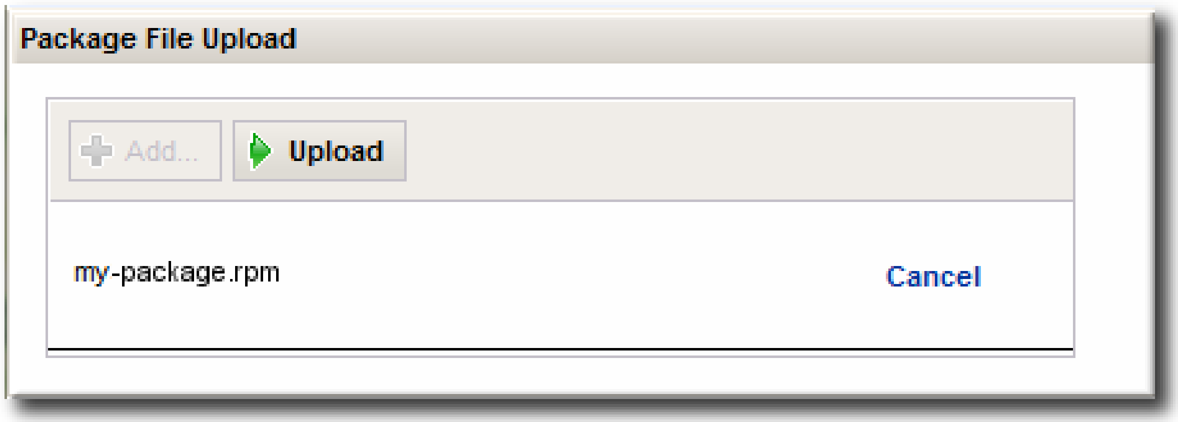
Name	Date Created	Sync Status	Description
Java Petstore	5/4/11, 2:46:39 PM, EDT	NONE	
new-repo	6/1/11, 6:48:23 PM, EDT	SUCCESS	

Total: 2

4. Scroll to the bottom of the page, to the **`Upload Packages`** section.

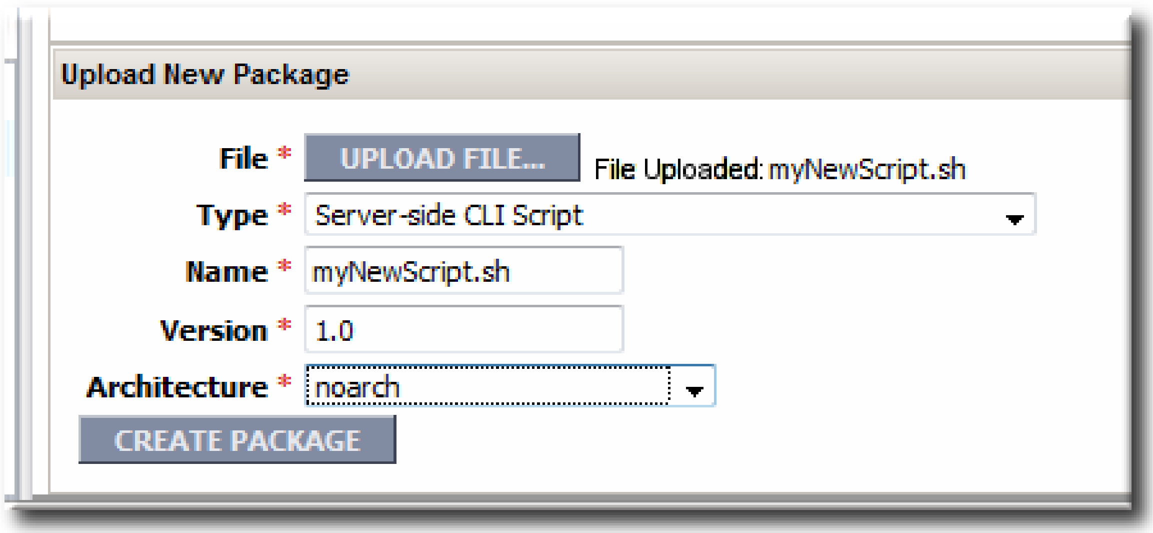
5. Click the **`Upload File`** button to upload the package.

6. In the pop-up window, click the **`Add`** button to browse to the package, then click the **`Upload`** button.



my-package.rpm

7. Some information about the package is automatically filled in, including the name and a default UI version number. Set the package type, architecture, and any other necessary information.



Upload New Package

File *	UPLOAD FILE... File Uploaded: myNewScript.sh
Type *	Server-side CLI Script
Name *	myNewScript.sh
Version *	1.0
Architecture *	noarch

If a version number is set, then this value is displayed in the UI. If not, then a version number is calculated, based on the spec version and implementation version in **MANIFEST.MF** (for EARs and WARs) or the calculated SHA-256 value for the package itself. Internally, the package is identified by the SHA value.

```
SPEC(IMPLMENTATION)[sha256=abcd1234]
```

Note

For exploded content for EARs and WARs, the calculated SHA-256 version number is written into the **MANIFEST.MF** file.

8. Click the **CREATE PACKAGE** button to finish adding the package to the repository.

3.5. Synchronizing Content Sources or Repositories

The original source of content is external to JBoss ON, and the content packages are pulled into JBoss ON and stored. Any changes that are made at the original content source need to be pulled into JBoss ON by *synchronizing* the two sources.

Likewise, any changes in the content source are carried over to the repository when the source and repository are synchronized.

3.5.1. Scheduling Synchronization

Synchronization is already scheduled in the content source entry in JBoss ON. This schedule has the standard cron format.

```
*      *     *     *      *  [sync-command]
-      -     -     -      -
|      |     |     |      |
|      |     |     |      +----- Day of Week (0=Sunday ... 6=Saturday)
|      |     |     +------- Month (1 - 12)
|      |     +--------- Date (1 - 31)
|      +----------- Hour (0 - 23)
+------------- Minute (0 - 59)
```

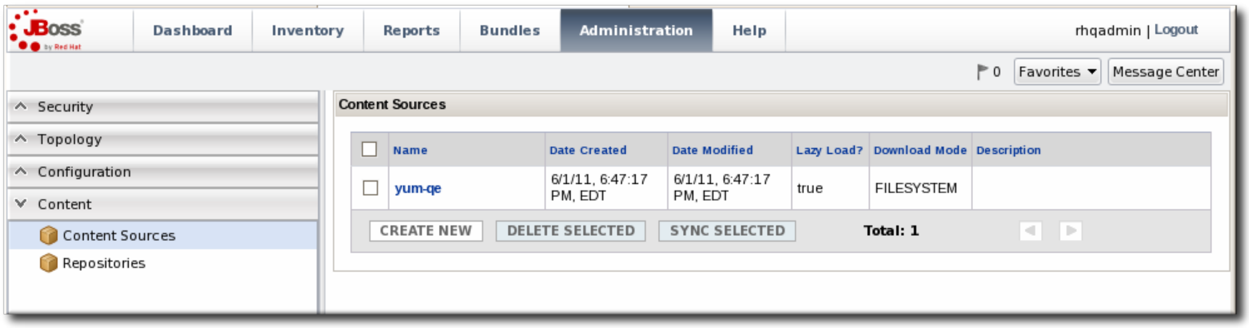
For example, to synchronize the source with JBoss ON on Tuesday and Friday at 3am:

```
0 3 * * 2,5
```

The Quartz documentation explains the cron syntax in more detail.

To edit the schedule synchronization times for a source:

1. In the top menu, click the **`Administration`** tab.
2. In the **`Content`** menu table on the left, select either the**Content Sources** or**Repositories** item.
3. Click the name of the item to edit.

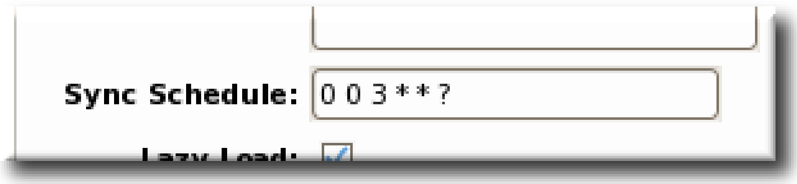


Content Sources

	Name	Date Created	Date Modified	Lazy Load?	Download Mode	Description
☐	yum-qe	6/1/11, 6:47:17 PM, EDT	6/1/11, 6:47:17 PM, EDT	true	FILESYSTEM	

Total: 1

4. Reset the cron schedule in the **`Sync Schedule`** field.



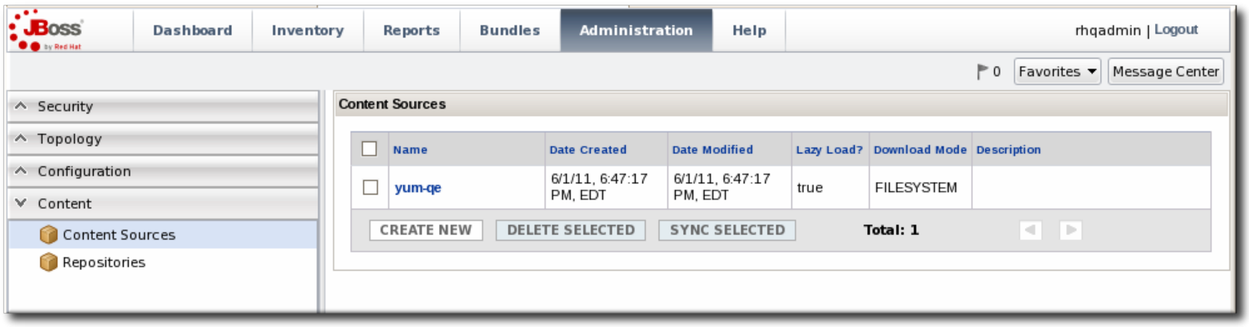
Sync Schedule: 0 0 3 * * ?

5. Click **Save**.

3.5.2. Manually Synchronizing Content Sources or Resources

If a major change happens to the content source, then the changes can be manually sent over to the JBoss ON server by initiating a synchronization manually.

1. In the top menu, click the **`Administration`** tab.
2. In the **`Content`** menu table on the left, select the**Content Sources** or **Repositories** item.
3. Click the name of the item to edit.

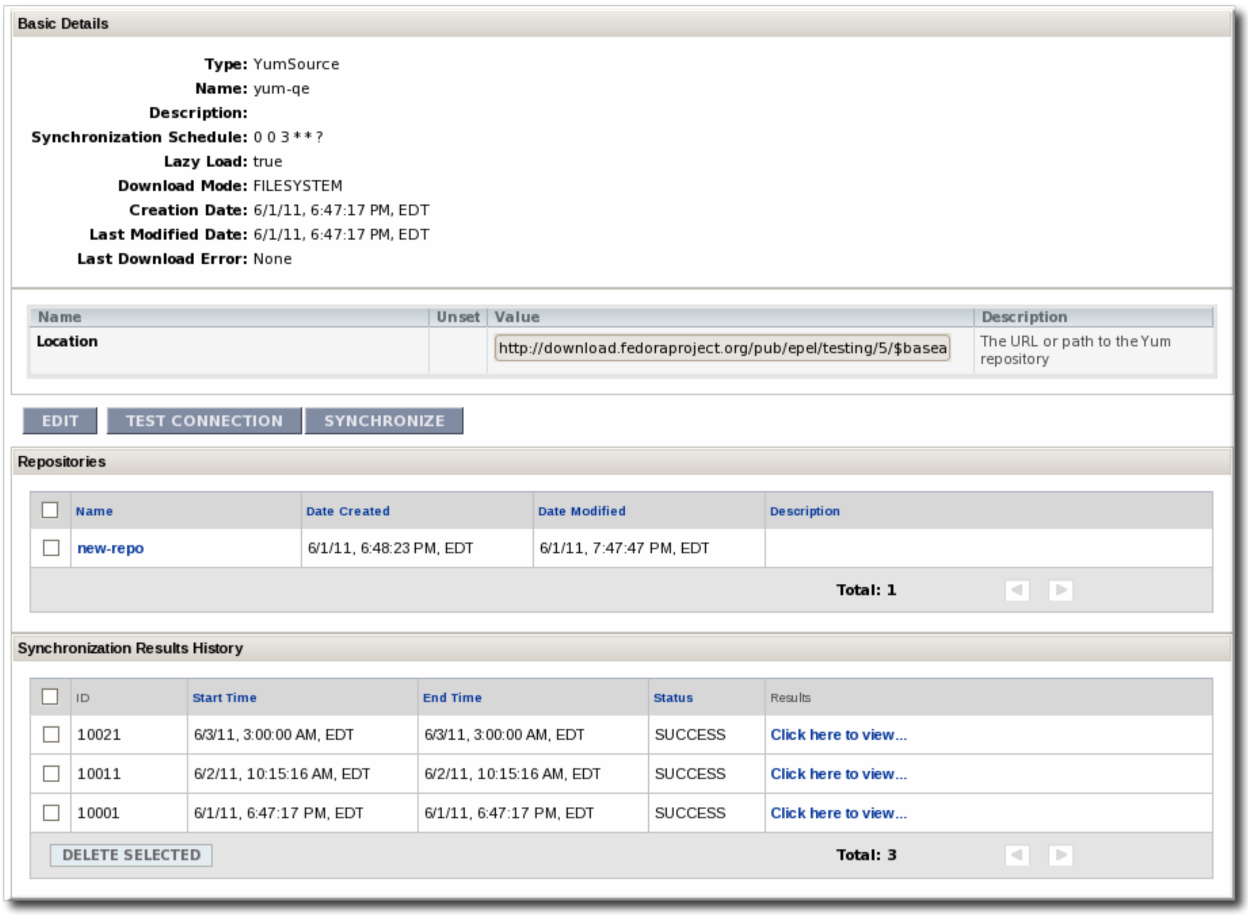


Content Sources

	Name	Date Created	Date Modified	Lazy Load?	Download Mode	Description
☐	yum-qe	6/1/11, 6:47:17 PM, EDT	6/1/11, 6:47:17 PM, EDT	true	FILESYSTEM	

Total: 1

4. Click the **Synchronize** button. All of the synchronization attempts, with the outcome of the operation, are listed at the bottom of the screen.



Basic Details

Type:	YumSource
Name:	yum-qe
Description:	
Synchronization Schedule:	0 0 3 * * ?
Lazy Load:	true
Download Mode:	FILESYSTEM
Creation Date:	6/1/11, 6:47:17 PM, EDT
Last Modified Date:	6/1/11, 6:47:17 PM, EDT
Last Download Error:	None

Name	Unset	Value	Description
Location		http://download.fedoraproject.org/pub/epel/testing/5/$basea	The URL or path to the Yum repository

Repositories

	Name	Date Created	Date Modified	Description
☐	new-repo	6/1/11, 6:48:23 PM, EDT	6/1/11, 7:47:47 PM, EDT	

Total: 1

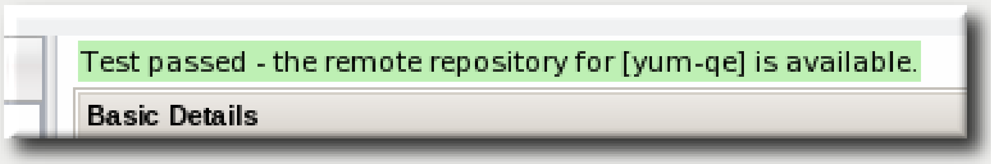
Synchronization Results History

	ID	Start Time	End Time	Status	Results
☐	10021	6/3/11, 3:00:00 AM, EDT	6/3/11, 3:00:00 AM, EDT	SUCCESS	Click here to view...
☐	10011	6/2/11, 10:15:16 AM, EDT	6/2/11, 10:15:16 AM, EDT	SUCCESS	Click here to view...
☐	10001	6/1/11, 6:47:17 PM, EDT	6/1/11, 6:47:17 PM, EDT	SUCCESS	Click here to view...

Total: 3

Note

You can test the connection to a source or repository by clicking the **Test Connection** button. This ensures that the JBoss ON server can connect to the content source before attempting to pull down the packages.



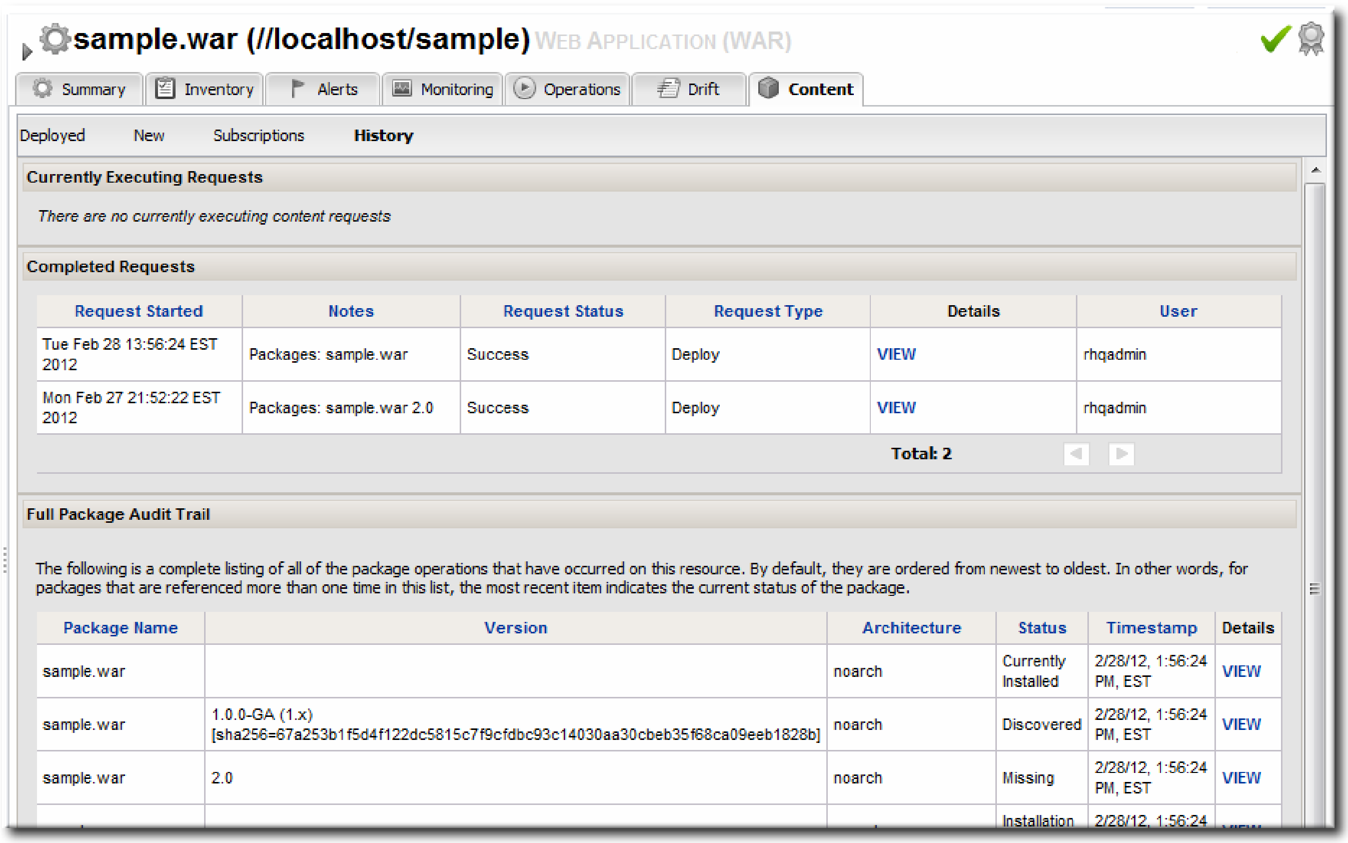
Test passed - the remote repository for [yum-qe] is available.

Basic Details

To synchronize multiple sources, stay on the main content sources or repositories page, select the checkbox by each of the content sources to synchronize, and click the **Sync Selected** button.

3.6. Tracking Content Versions for a Resource

Every time a package is installed on a resource through a repository, the resource shows the operation. This includes even installation failures. The content package history for a resource is viewable in the **Content** tab, under the **History** subtab.



Currently Executing Requests

There are no currently executing content requests

Completed Requests

Request Started	Notes	Request Status	Request Type	Details	User
Tue Feb 28 13:56:24 EST 2012	Packages: sample.war	Success	Deploy	VIEW	rhqadmin
Mon Feb 27 21:52:22 EST 2012	Packages: sample.war 2.0	Success	Deploy	VIEW	rhqadmin

Total: 2

Full Package Audit Trail

The following is a complete listing of all of the package operations that have occurred on this resource. By default, they are ordered from newest to oldest. In other words, for packages that are referenced more than one time in this list, the most recent item indicates the current status of the package.

Package Name	Version	Architecture	Status	Timestamp	Details
sample.war		noarch	Currently Installed	2/28/12, 1:56:24 PM, EST	VIEW
sample.war	1.0.0-GA (1.x) [sha256=67a253b1f5d4f122dc5815c7f9cfdbc93c14030aa30cbeb35f68ca09eeb1828b]	noarch	Discovered	2/28/12, 1:56:24 PM, EST	VIEW
sample.war	2.0	noarch	Missing	2/28/12, 1:56:24 PM, EST	VIEW
			Installation	2/28/12, 1:56:24	

Figure 9. Package History for a Resource

The package history shows both the time the operation was initiated and completed and the user who initiated it. This is valuable for auditing changes, correlating incidents and response, and tracking resource configuration.

4. Deploying Applications on Application Servers

Applications such as EAR and WAR files that are deployed on an application server are cross between a child resource (of the application server) and content that is managed in a repository.

For these content-backed resources, the child resource is created first, by uploading a package to the JBoss server. After that, they are managed like content, with updated packages added to a content repo and then applied to the application server.

4.1. Setting Permissions for Agent and Resource Users

The assumption is that the JBoss ON agent and resources like a JBoss server or Tomcat server run as the same system user. This allows the agent and the application server itself to manage resource content and configuration simultaneously.

However, if the agent user is different than the resource user, then there can be problems when one entity makes a configuration change and the other attempts a change later.

For example, when deploying an application, the deployment operation is initiated by the agent and the content is supplied through the agent, and then the application server completes the actual deployment. When deleting an application, the application server handles the undeployment by itself.

If a WAR file is deployed exploded without a **MANIFEST.MF** file, the agent creates one when it writes the SHA-256 value for the package. When the JBoss AS server tries to remove the WAR application later (and the JBoss AS user is different than the agent user), then the removal fails. The JBoss AS server cannot delete the **MANIFEST.MF** file. The agent then rediscovers the application directory and re-initiates the deployment operation for the removed WAR.

Note

This situation only occurs when the application is exploded *and* does not contain the **MANIFEST.MF** file — meaning, a situation where the agent creates a file within the deployment directory. Even if the agent and JBoss AS users are different, this situation does not occur if the application is not exploded or where the agent does not write any files.

This situation can be avoided. If the agent user and resource user are different, then change the system settings:

1. Add the agent user and the resource user to the same primary group.

2. Set the **umask** value for the agent user to give read and write permissions, such as **660**. For example:

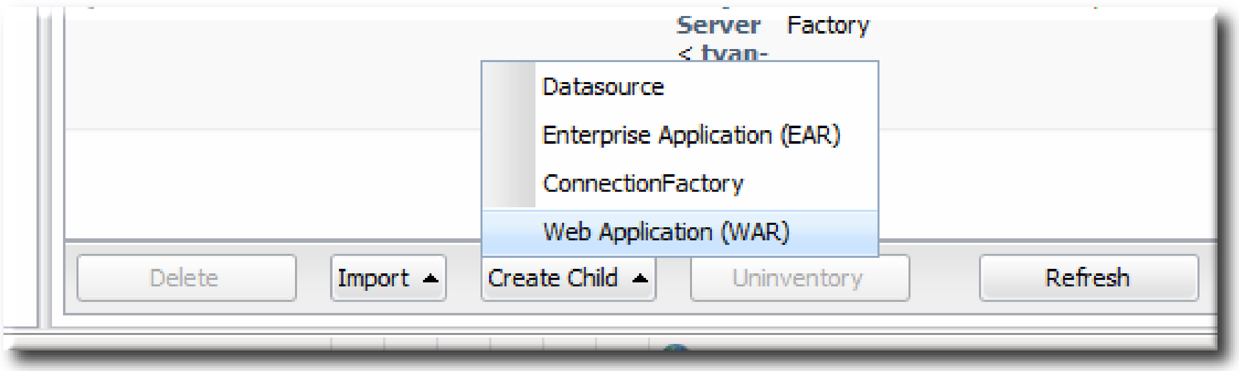
```
vim /home/rhqagent/.bashrc

umask 660
```

4.2. Deploying EAR and WAR Files

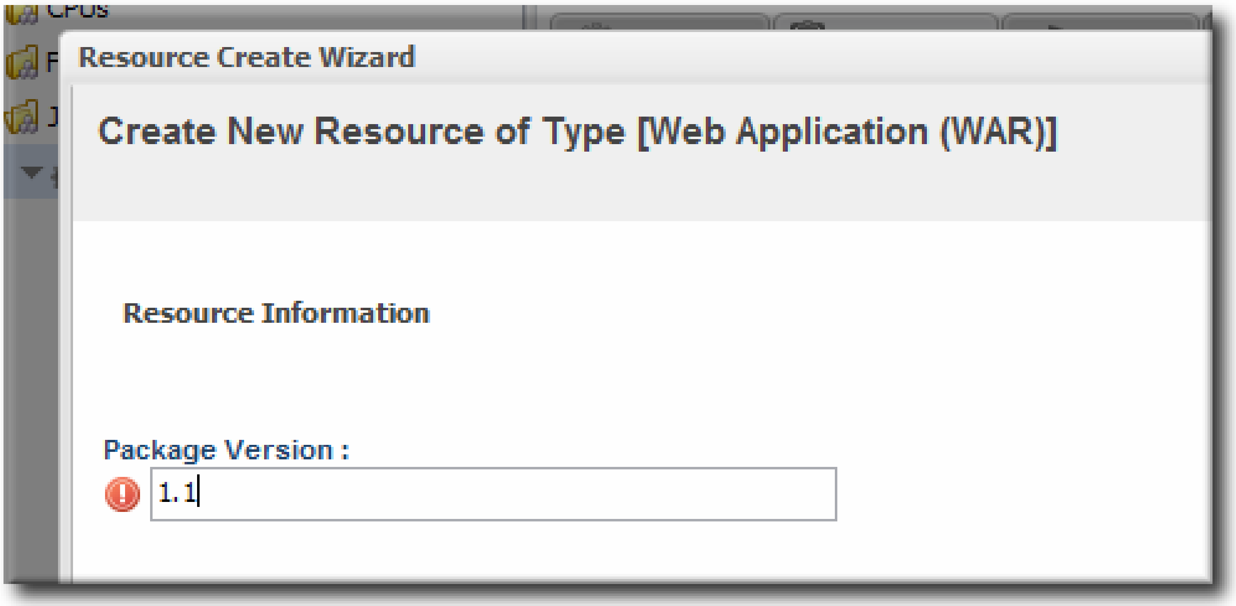
1. Search for the JBoss server instance to which to deploy the EAR or WAR.

2. On the details page for the selected JBoss server instance, open the **Inventory** tab.

3. In the **Create New** menu at the bottom, select the item for **- Web Application (WAR)** or **- Enterprise Application (EAR)**, as appropriate.



4. Enter the version number.



Create New Resource of Type [Web Application (WAR)]

Resource Information

Package Version : 1.1

This is not used for the resource. The actual version number is calculated based on the spec version and implementation version in **MANIFEST.MF**, if any are given, or the caluclated SHA-256 value for the package itself:

```
SPEC(IMPLMENTATION)[sha256=abcd1234]
```

If no version numbers are defined in **MANIFEST.MF**, then the SHA value is used. The SHA value is always used to identify the package version internally.

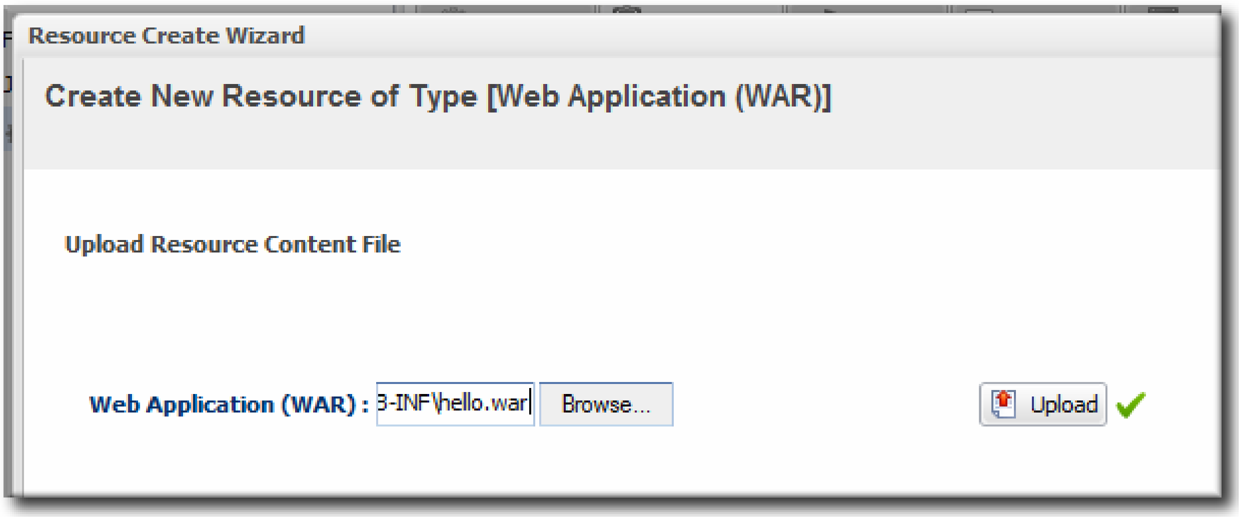
Note

When the EAR or WAR file is exploded after it is deployed, the **MANIFEST.MF** file is updated to include the calculated SHA version number. For example:

```
Manifest-Version: 1.0
Created-By: Apache Maven
RHQ-Sha256: 570f196c4a1025a717269d16d11d6f37
...
```

For more information on package versioning, see "Deploying Applications and Content".

5. Upload the EAR/WAR file.

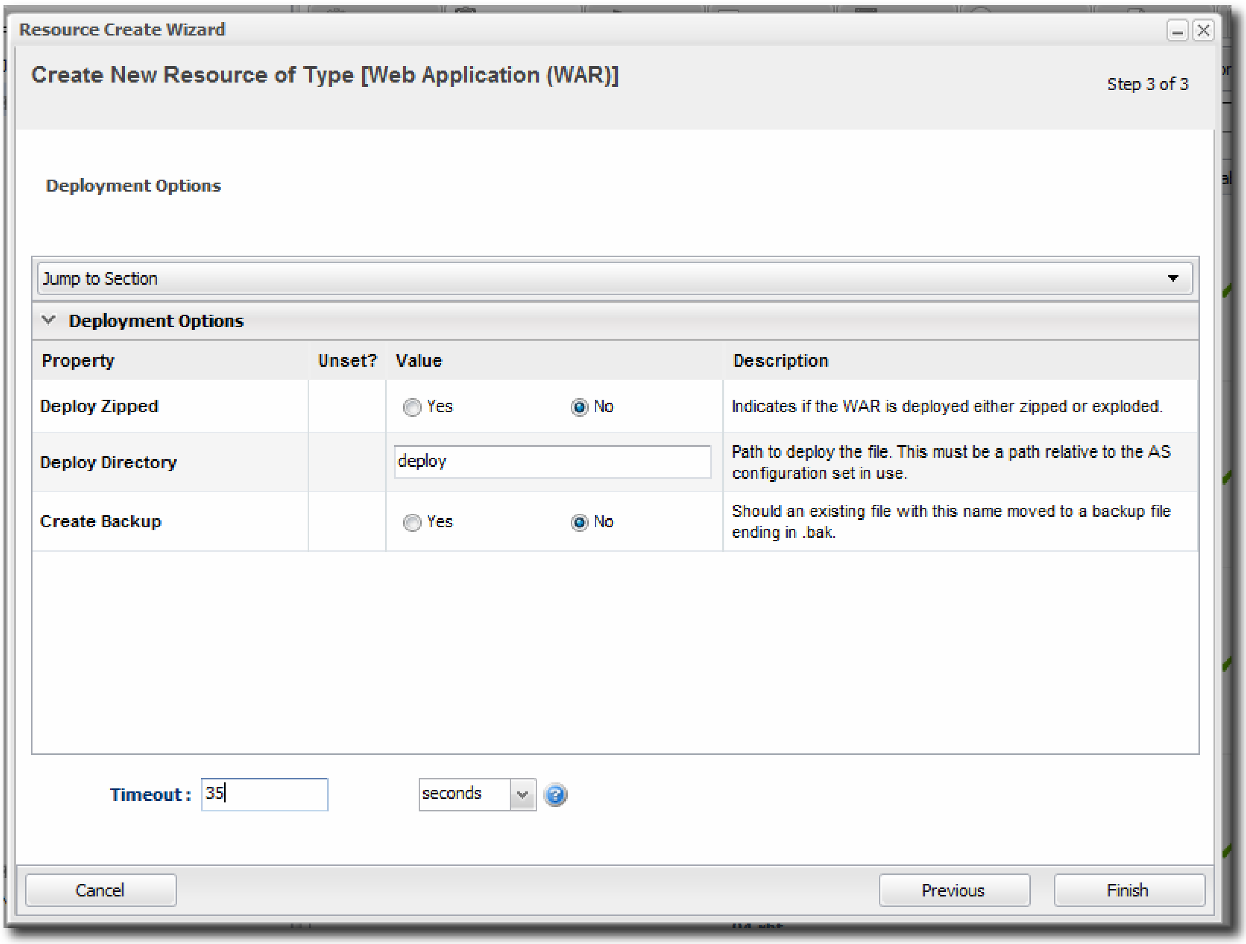

Resource Create Wizard

Create New Resource of Type [Web Application (WAR)]

Upload Resource Content File

Web Application (WAR) : 3-INF\hello.war

6. Enter the information for the application to be deployed.


Resource Create Wizard

Create New Resource of Type [Web Application (WAR)]

Step 3 of 3

Deployment Options

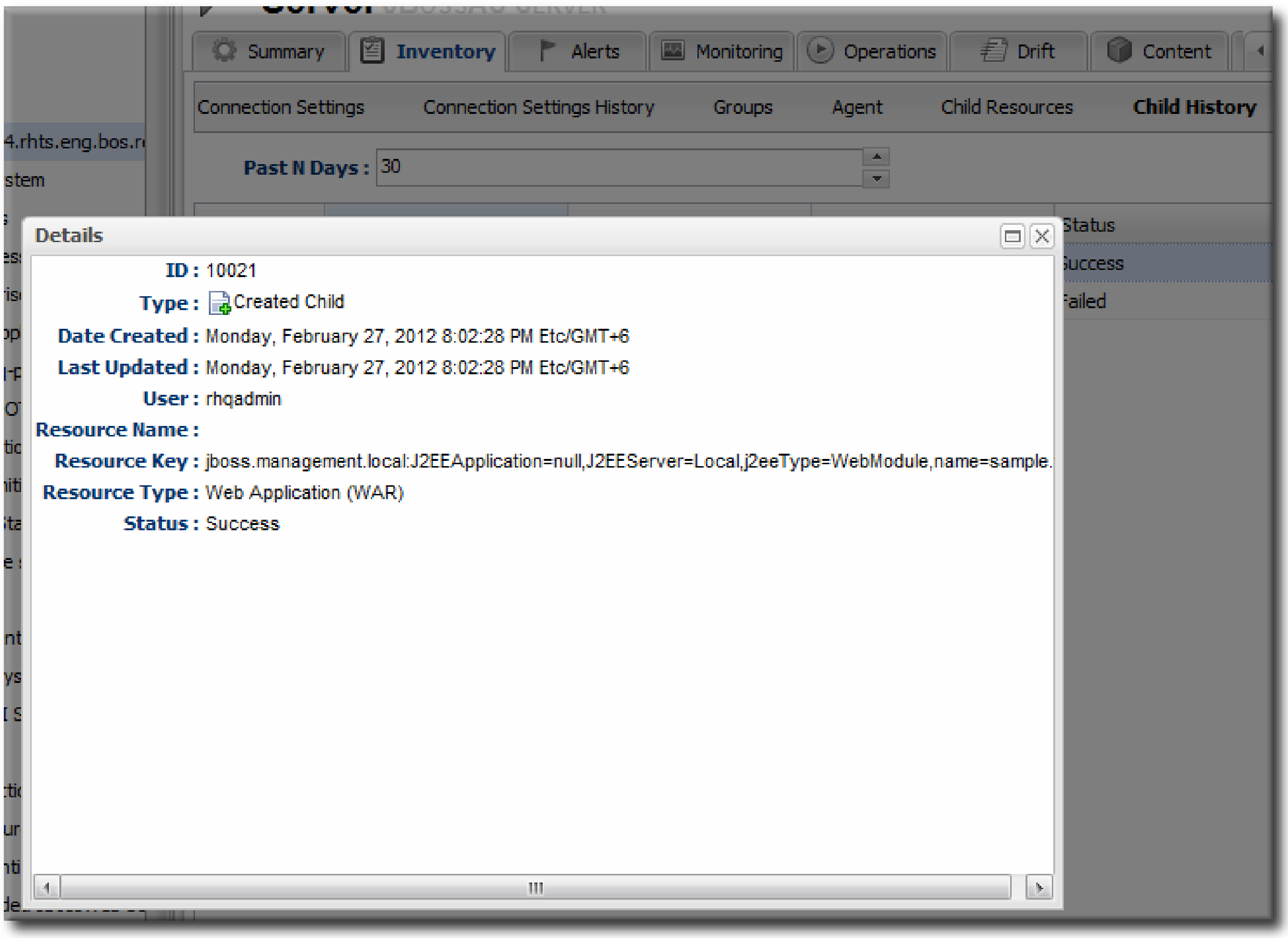
Deployment Options

Property	Unset?	Value	Description
Deploy Zipped		Yes [✓ No]	Indicates if the WAR is deployed either zipped or exploded.
Deploy Directory		deploy	Path to deploy the file. This must be a path relative to the AS configuration set in use.
Create Backup		Yes [✓ No]	Should an existing file with this name moved to a backup file ending in .bak.

Timeout : 35 seconds

- Whether the file should be exploded (unzipped) when it is deployed.
- The path to the directory to which to deploy the EAR or WAR package. The destination directory is relative to the JBoss server instance installation directory; this cannot contain an absolute path or go up a parent directory.
- Whether to back up any existing file with the same name in the target directory.

Once the EAR/WAR file is confirmed, the new child resource is listed in the **`Child History`** subtab of the **`Inventory`** tab.



ID : 10021
Type : Created Child
Date Created : Monday, February 27, 2012 8:02:28 PM Etc/GMT+6
Last Updated : Monday, February 27, 2012 8:02:28 PM Etc/GMT+6
User : rhqadmin
Resource Name :
Resource Key : jboss.management.local:J2EEApplication=null,J2EEServer=Local,j2eeType=WebModule,name=sample.
Resource Type : Web Application (WAR)
Status : Success

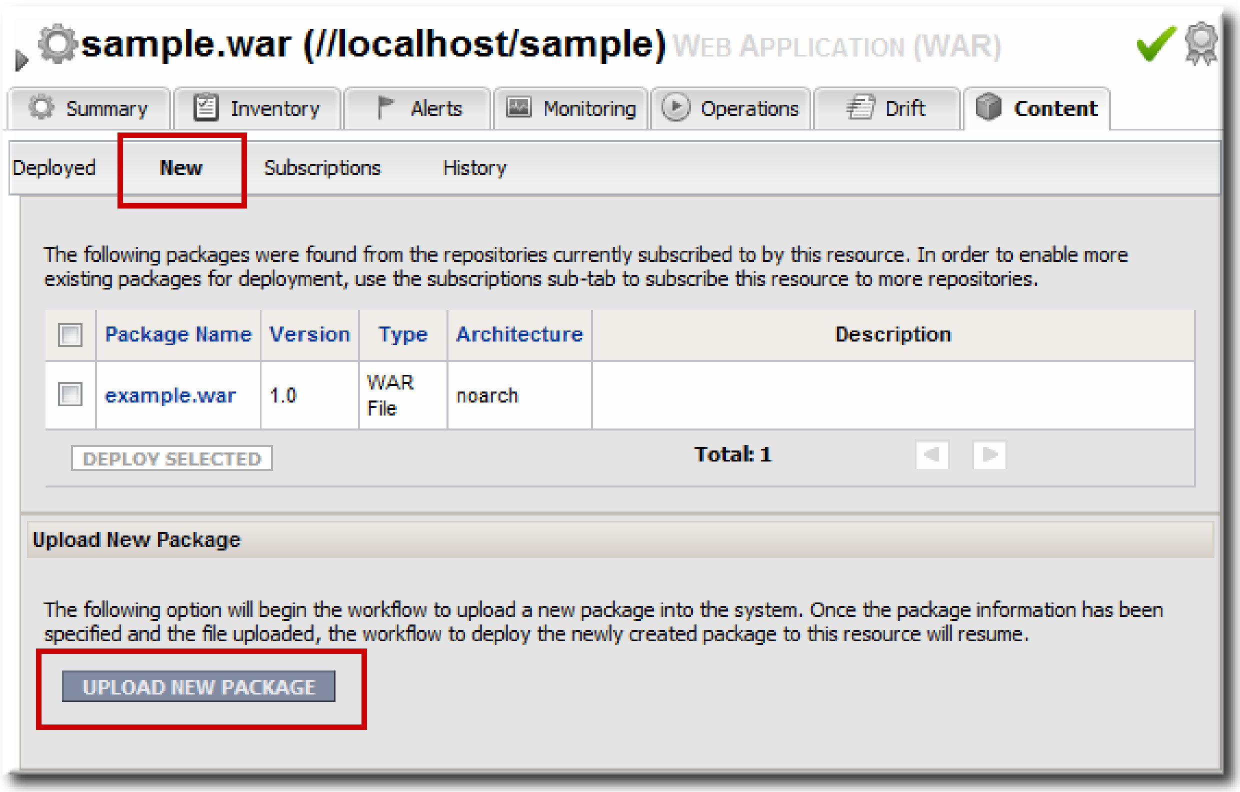
Figure 10. WAR Child Resource

4.3. Updating Applications

After the EAR or WAR resource is created, changes are treated like updated content packages. Updating the EAR/WAR resource is the same as uploading and applying new packages to that EAR/WAR resource entry.

1. Browse to the EAR or WAR resource in the JBoss ON UI.

2. In the EAR or WAR resource details page, open the **`Content`** tab, and click the **`New`** subtab.



The following packages were found from the repositories currently subscribed to by this resource. In order to enable more existing packages for deployment, use the subscriptions sub-tab to subscribe this resource to more repositories.

☐	Package Name	Version	Type	Architecture	Description
☐	example.war	1.0	WAR File	noarch	

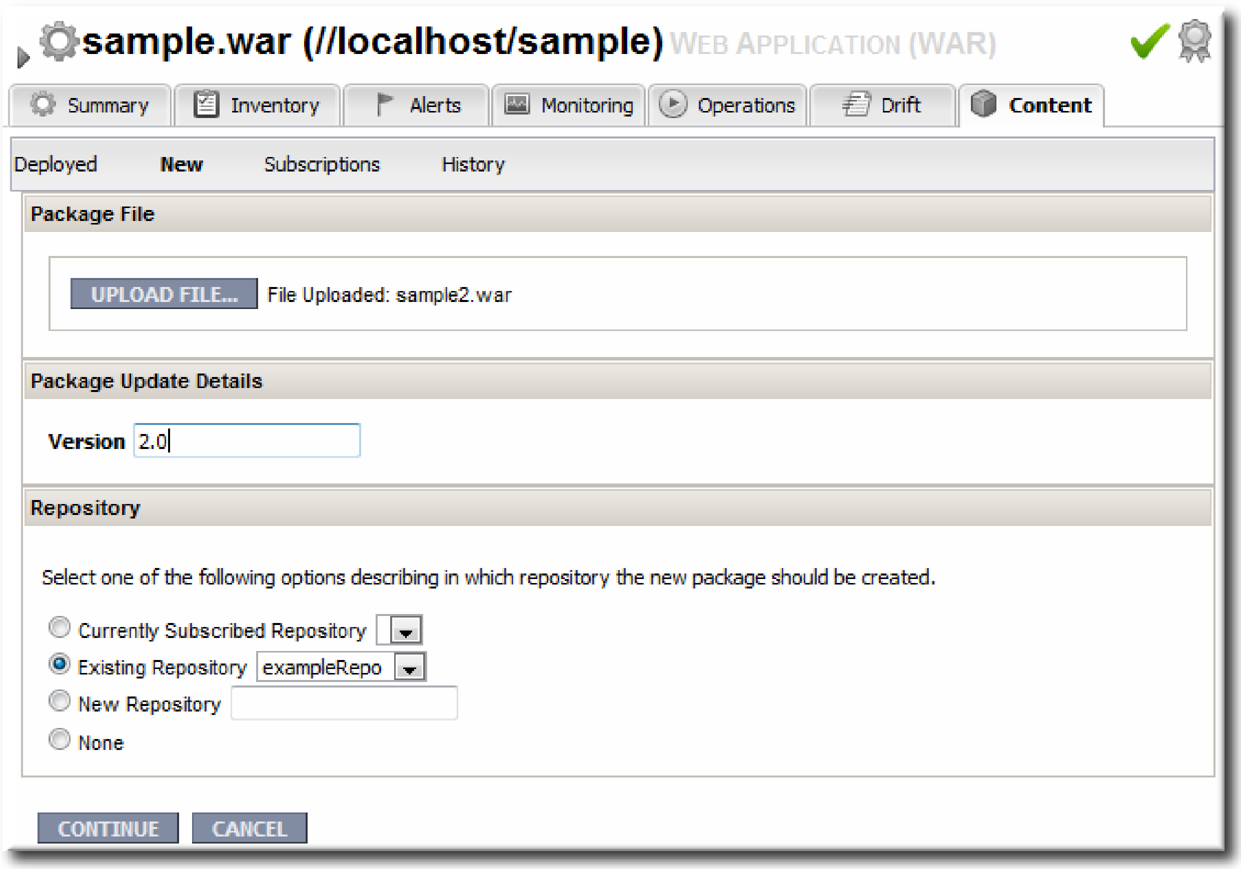
Total: 1

Upload New Package

The following option will begin the workflow to upload a new package into the system. Once the package information has been specified and the file uploaded, the workflow to deploy the newly created package to this resource will resume.

3. Click the **`UPLOAD NEW PACKAGE`** button.

4. Click the **`UPLOAD FILE`** button.



Package File

File Uploaded: sample2.war

Package Update Details

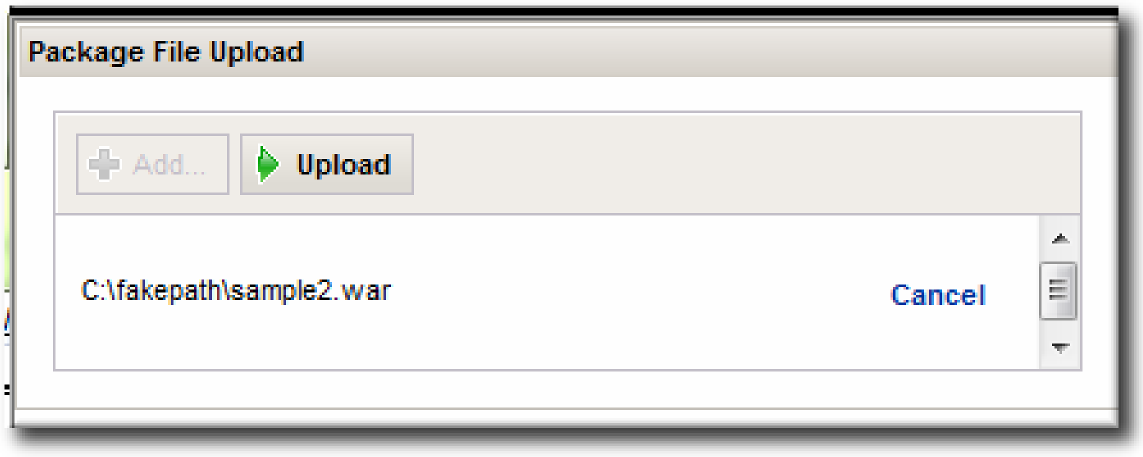
Version 2.0

Repository

Select one of the following options describing in which repository the new package should be created.

- ○ Currently Subscribed Repository
- ● Existing Repository exampleRepo
- ○ New Repository
- ○ None

5. In the pop-up window, click the **Add** button, and browse the local filesystem to the updated WAR or EAR file to be uploaded.



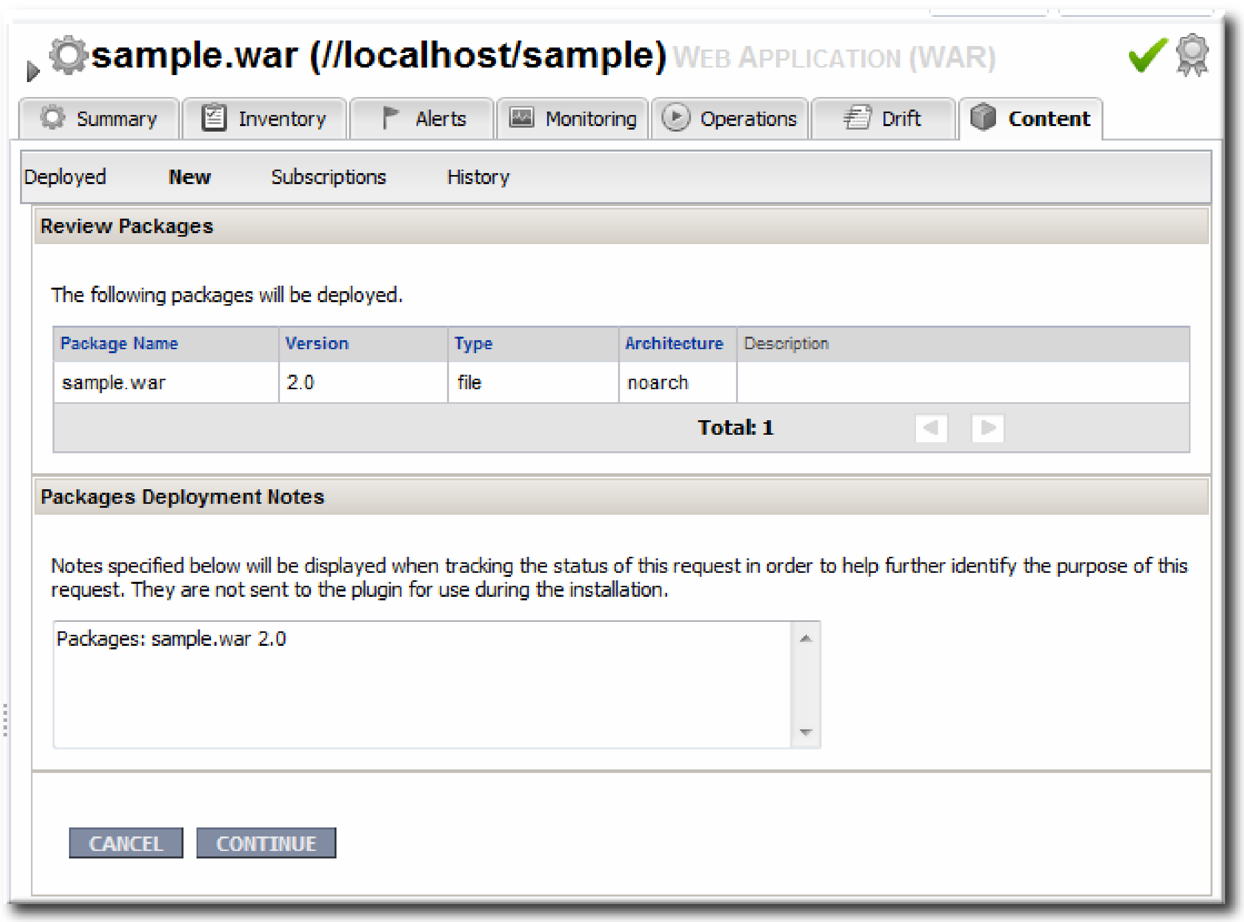
C:\fakepath\sample2.war

6. Click the **`UPLOAD`** button to load the file and dismiss the window.
7. In the main form, select the repository where the WAR or EAR file package should be stored. If one exists, select an existing repository or a subscribed repository for the resource. Otherwise, create a new repository.
8. Optionally, set the version number for the EAR/WAR package.

   If this is set, then this value is displayed in the UI. If not, then a version number is calculated, based on the spec version and implementation version in **`MANIFEST.MF`**, if any are given, or the calculated SHA-256 value for the package itself. Internally, the package is identified by the SHA value.

   ```
   SPEC(IMPLMENTATION)[sha256=abcd1234]
   ```

   For more information on package versioning, see "Deploying Applications and Content".
9. Confirm the details for the new package, then click **`CONTINUE`**.



Review Packages

The following packages will be deployed.

Package Name	Version	Type	Architecture	Description
sample.war	2.0	file	noarch	

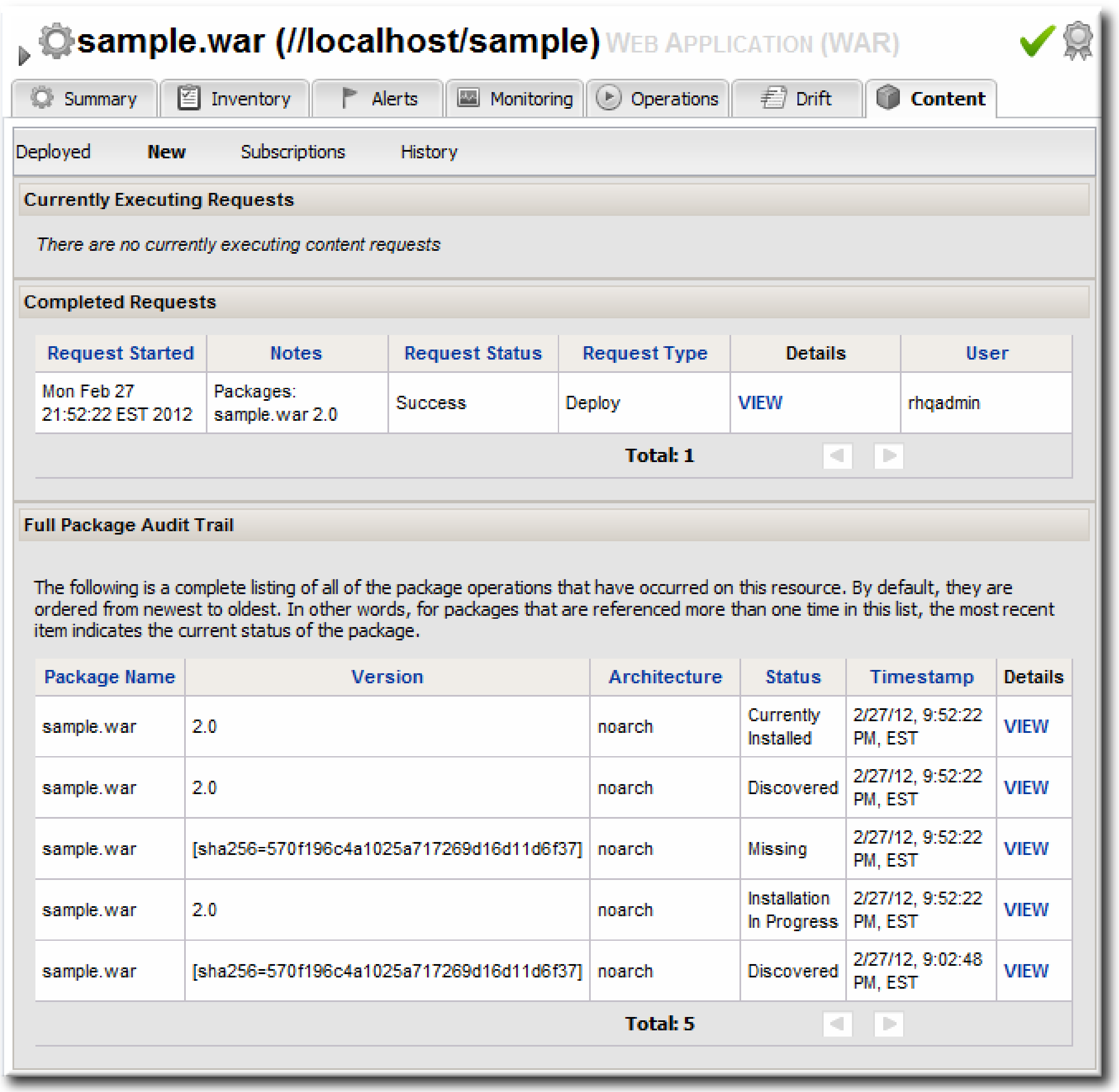
Total: 1

Packages Deployment Notes

Notes specified below will be displayed when tracking the status of this request in order to help further identify the purpose of this request. They are not sent to the plugin for use during the installation.

Packages: sample.war 2.0

When the package is successfully uploaded, the UI redirects to the history page on the **`Content`** tab.



Currently Executing Requests

There are no currently executing content requests

Completed Requests

Request Started	Notes	Request Status	Request Type	Details	User
Mon Feb 27 21:52:22 EST 2012	Packages: sample.war 2.0	Success	Deploy	VIEW	rhqadmin

Total: 1

Full Package Audit Trail

The following is a complete listing of all of the package operations that have occurred on this resource. By default, they are ordered from newest to oldest. In other words, for packages that are referenced more than one time in this list, the most recent item indicates the current status of the package.

Package Name	Version	Architecture	Status	Timestamp	Details
sample.war	2.0	noarch	Currently Installed	2/27/12, 9:52:22 PM, EST	VIEW
sample.war	2.0	noarch	Discovered	2/27/12, 9:52:22 PM, EST	VIEW
sample.war	[sha256=570f196c4a1025a717269d16d11d6f37]	noarch	Missing	2/27/12, 9:52:22 PM, EST	VIEW
sample.war	2.0	noarch	Installation In Progress	2/27/12, 9:52:22 PM, EST	VIEW
sample.war	[sha256=570f196c4a1025a717269d16d11d6f37]	noarch	Discovered	2/27/12, 9:02:48 PM, EST	VIEW

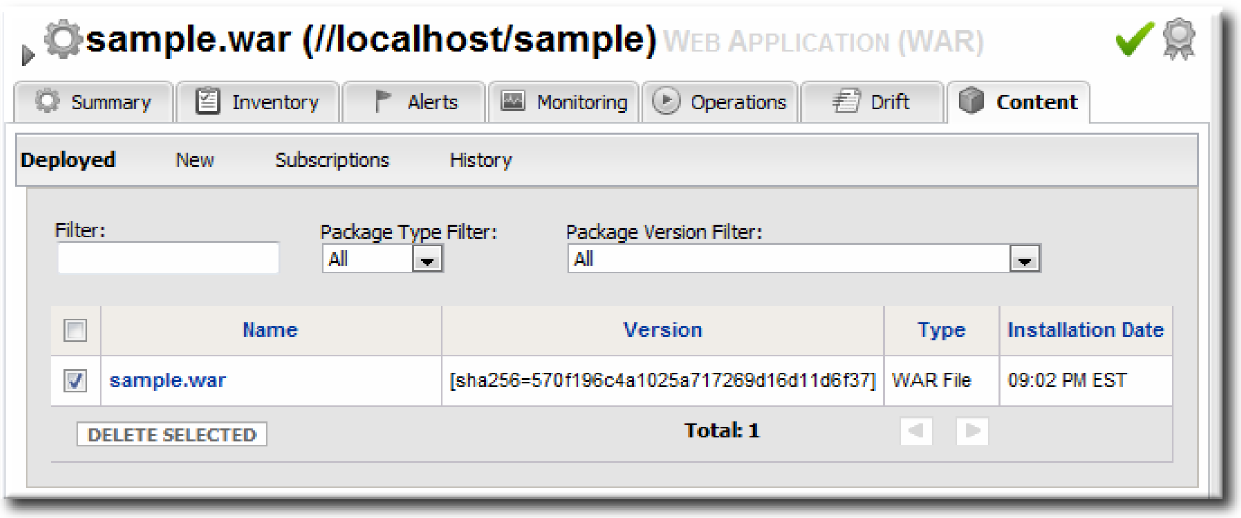
Total: 5

Figure 11. Deployment History for a Resource

4.4. Deleting an Application

Deleting an EAR/WAR application is the same as deleting the currently deployed package associated with that EAR/WAR resource entry.

1. Browse to the EAR or WAR resource in the JBoss ON UI.
2. In the EAR or WAR resource details page, open the **Content** tab, and click the **Deployed** subtab.
3. Select the checkbox by the EAR/WAR package, and click the **DELETE SELECTED** button.



Filter:

Package Type Filter: All

Package Version Filter: All

	Name	Version	Type	Installation Date
✓	sample.war	[sha256=570f196c4a1025a717269d16d11d6f37]	WAR File	09:02 PM EST

Total: 1

5. 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.

5.1. 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**
Updates for JBoss Operations Network 3.0.1.

Revision 3.0-1 **January 26, 2012** **Ella Deon Lackey**
Edits to the bundle section based on feedback from John Mazzitelli, Jay Shaughnessy, Charles Crouch, and other SME's.

Revision 3.0-0 **December 7, 2011** **Ella Deon Lackey**
Initial release of JBoss ON 3.0.

Index