Red Hat JBoss Fuse 6.3 Configuring and Running JBoss Fuse

Managing the runtime container

JBoss A-MQ Docs
Team

Red Hat JBoss Fuse 6.3 Configuring and Running JBoss Fuse

Managing the runtime container

JBoss A-MQ Docs Team
Content Services
fuse-docs-support@redhat.com

Legal Notice

Copyright © 2016 Red Hat.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at
http://creativecommons.org/licenses/by-sa/3.0/
. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat Software Collections is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

This guide provides information and instructions for starting/stopping Red Hat JBoss Fuse, using remote and child instances of the runtime, configuring Red Hat JBoss Fuse, configuring logging for the entire runtime or per component application, configuring where persistent data (messages, log files, OSGi bundles, transaction logs) is stored, and configuring failover deployments.

Table of Contents

CHAPTER 1. CONFIGURING THE INITIAL FEATURES IN A STANDALONE CONTAINER 5
OVERVIEW 5
MODIFYING THE DEFAULT INSTALLED FEATURES 5
MODIFYING THE DEFAULT SET OF FEATURE URLS 5

CHAPTER 2. INSTALLING RED HAT JBOSS FUSE AS A SERVICE 6
2.1. OVERVIEW 6

CHAPTER 3. BASIC SECURITY 10
3.1. CONFIGURING BASIC SECURITY 10
3.2. DISABLING BROKER SECURITY 13

CHAPTER 4. STARTING AND STOPPING JBOSS FUSE 15
4.1. STARTING JBOSS FUSE 15
4.2. STOPPING JBOSS FUSE 17

CHAPTER 5. CREATING A NEW FABRIC 20
STATIC IP ADDRESS REQUIRED FOR FABRIC SERVER 20
MAKE QUICKSTART EXAMPLES AVAILABLE 20
PROCEDURE 21
FABRIC CREATION PROCESS 23
EXPANDING A FABRIC 23

CHAPTER 6. JOINING A FABRIC 25
OVERVIEW 25
JOINING A FABRIC AS A MANAGED CONTAINER 25
JOINING A FABRIC AS AN NON-MANAGED CONTAINER 25
HOW TO JOIN A FABRIC 25
HOW TO DISCOVER THE URL OF A FABRIC SERVER 27

CHAPTER 7. SHUTTING DOWN A FABRIC 28
OVERVIEW 28
SHUTTING DOWN A MANAGED CONTAINER 28
SHUTTING DOWN A FABRIC SERVER 28
SHUTTING DOWN AN ENTIRE FABRIC 28
NOTE ON SHUTTING DOWN THE ENSEMBLE 30

CHAPTER 8. USING REMOTE CONNECTIONS TO MANAGE A CONTAINER 31
8.1. CONFIGURING A CONTAINER FOR REMOTE ACCESS 31
8.2. CONNECTING AND DISCONNECTING REMOTELY 31
8.3. STOPPING A REMOTE CONTAINER 40

CHAPTER 9. MANAGING CHILD CONTAINERS 42
9.1. STANDALONE CHILD CONTAINERS 42
9.2. FABRIC CHILD CONTAINERS 45

CHAPTER 10. DEPLOYING A NEW BROKER INSTANCE 49
OVERVIEW 49
STANDALONE CONTAINERS 49
EXAMPLE 50

CHAPTER 11. CONFIGURING JBOSS FUSE 52
11.1. INTRODUCING JBOSS FUSE CONFIGURATION 52
11.2. SETTING OSGI FRAMEWORK AND INITIAL CONTAINER PROPERTIES 55
11.3. CONFIGURING STANDALONE CONTAINERS USING THE COMMAND CONSOLE 56

11.3. CONFIGURING STANDALONE CONTAINERS USING THE COMMAND CONSOLE
11.4. CONFIGURING FABRIC CONTAINERS 58

CHAPTER 12. CONFIGURING THE HOT DEPLOYMENT SYSTEM . . . 61
OVERVIEW 61
SPECIFYING THE HOT DEPLOYMENT FOLDER 61
SPECIFYING THE SCAN INTERVAL 61
EXAMPLE 61

CHAPTER 13. CONFIGURING JMX . . . 63
OVERVIEW 63
CHANGING THE RMI PORT AND JMX URL 63
SETTING THE JMX USERNAME AND PASSWORD 63
TROUBLESHOOTING ON LINUX PLATFORMS 63

CHAPTER 14. CONFIGURING JAAS SECURITY . . . 65
14.1. ALTERNATIVE JAAS REALMS 65
14.2. JAAS CONSOLE COMMANDS 66
14.3. STANDALONE REALM PROPERTIES FILE 69

CHAPTER 15. SECURING FABRIC CONTAINERS . . . 70
DEFAULT AUTHENTICATION SYSTEM 70
MANAGING USERS 70
OBFUSCATING STORED PASSWORDS 70
ENABLING LDAP AUTHENTICATION 71

CHAPTER 16. LOGGING . . . 72
16.1. LOGGING OVERVIEW 72
16.2. LOGGING CONFIGURATION 72
16.3. LOGGING PER APPLICATION 75
16.4. LOG COMMANDS 76

CHAPTER 17. PERSISTENCE . . . 78
OVERVIEW 78
THE DATA FOLDER 78
CHANGING THE BUNDLE CACHE LOCATION 79
FLUSHING THE BUNDLE CACHE 79
CHANGING THE GENERATED-BUNDLE CACHE LOCATION 79
ADJUSTING THE BUNDLE CACHE BUFFER 79

CHAPTER 18. FAILOVER DEPLOYMENTS . . . 80
18.1. USING A SIMPLE LOCK FILE SYSTEM 80
18.2. USING A JDBC LOCK SYSTEM 80
18.3. CONTAINER-LEVEL LOCKING 84

CHAPTER 19. APPLYING PATCHES . . . 86
19.1. PATCHING OVERVIEW 86
19.2. FINDING THE RIGHT PATCHES TO APPLY 86
19.3. INSTALLING A ROLLUP PATCH AS A NEW INSTALLATION 88
19.4. PATCHING A STANDALONE CONTAINER 89
19.5. PATCHING A CUSTOM ASSEMBLY 94
19.6. PATCHING A FABRIC CONTAINER WITH A ROLLUP PATCH 95
19.7. PATCHING A FABRIC CONTAINER WITH AN INCREMENTAL PATCH 101

CHAPTER 20. FABRIC MAVEN PROXIES . . . 105
20.1. CLUSTER OF FABRIC MAVEN PROXIES 105
20.2. HOW A MANAGED CONTAINER RESOLVES ARTIFACTS 108

20.2. HOW A MANAGED CONTAINER RESOLVES ARTIFACTS 108
20.3. HOW A MAVEN PROXY RESOLVES ARTIFACTS 112
20.4. CONFIGURING MAVEN PROXIES DIRECTLY 114
20.5. CONFIGURING MAVEN PROXIES THROUGH SETTINGS.XML 117
20.6. AUTOMATED DEPLOYMENT 120
20.7. FABRIC MAVEN CONFIGURATION REFERENCE 121

CHAPTER 21. MAVEN INDEXER PLUGIN 127

CHAPTER 22. WELCOME BANNER 128

CHAPTER 23. BRANDING JBOSS FUSE CONSOLE 130

INDEX 132

CHAPTER 1. CONFIGURING THE INITIAL FEATURES IN A STANDALONE CONTAINER

Abstract

If you are using a standalone container, you can change the features it automatically loads the first time it is started.

OVERVIEW

The *first* time you start a standalone container, the container looks in the **`etc/org.apache.karaf.features.cfg`** file to discover the feature URLs (feature repository locations) and to determine which features it will load. By default, Red Hat JBoss Fuse loads a large number of features and you may not need all of them. You may also decide you need features that are not included in the default configuration.

> **Warning**
>
> The features loaded by a Fabric Container are controlled by the container's profiles. Changing the values as described below will have no effect on a Fabric container.

The values in **`etc/org.apache.karaf.features.cfg`** are only used the *first* time the container is started. On subsequent start-ups, the container uses the contents of the ***`InstallDir`*`/data`** directory to determine what to load. If you need to adjust the features loaded into a container, you can delete the **`data`** directory, but this will also destroy any state or persistence information stored by the container.

For more on features and how they are used in Red Hat JBoss Fuse, see chapter "Deploying Features" in "Deploying into Apache Karaf".

MODIFYING THE DEFAULT INSTALLED FEATURES

By default, JBoss Fuse installs a large number of features, including some that you may not want to deploy.

You can change the initial set of installed features by editing the featuresBoot property.

MODIFYING THE DEFAULT SET OF FEATURE URLS

JBoss Fuse registers a number of URLs that point to feature repositories on start-up. You can change the initial set of feature URLs by editing the featureRepositories property.

CHAPTER 2. INSTALLING RED HAT JBOSS FUSE AS A SERVICE

Abstract

This chapter provides information on how you can start the Red Hat JBoss Fuse instance as a system service by using the templates.

2.1. OVERVIEW

By using the Service Script templates, you can run a fuse instance with the help of operating system specific init scripts. You can find these templates under the **`bin/contrib`** directory.

2.1.1. Running Fuse as a Service

The **`karaf-service.sh`** utility helps you to customize the templates. This utility will automatically identify the operating system and the default init system and generates ready to use init scripts. You can also customize the scripts to adapt them to its environment, by setting JAVA_HOME and few other environment variables.

The generated scripts are composed by two files:

1. the init script

2. the init configuration file

2.1.2. Customizing karaf-service.sh Utility

You can customize the **`karaf-service.sh`** utility, by defining an environment variable or by passing command line options:

Table 2.1.

Command Line Option	Environment Variable	Description
-k	KARAF_SERVICE_PATH	Karaf installation path
-d	KARAF_SERVICE_DATA	Karaf data path (default to **`\${KARAF_SERVICE_PATH}/data`**)

Command Line Option	Environment Variable	Description
-c	KARAF_SERVICE_CONF	Karaf configuration file (default to **\${KARAF_SERVICE_PATH/ etc/\${KARAF_SERVICE_NAME}.conf**)
-t	KARAF_SERVICE_ETC	Karaf etc path (default to **\${KARAF_SERVICE_PATH/ etc}**)
-p	KARAF_SERVICE_PIDFILE	Karaf pid path (default to **\${KARAF_SERVICE_DATA} /\${KARAF_SERVICE_NAME }.pid)**)
-n	KARAF_SERVICE_NAME	Karaf service name (default karaf)
-e	KARAF_ENV	Karaf environment variable
-u	KARAF_SERVICE_USER	Karaf user
-g	KARAF_SERVICE_GROUP	Karaf group (default **\${KARAF_SERVICE_USER**)
-l	KARAF_SERVICE_LOG	Karaf console log (default to **\${KARAF_SERVICE_DATA} /log/\${KARAF_SERVICE_NAME}-console.log**)
-f	KARAF_SERVICE_TEMPLATE	Template file to use
-x	KARAF_SERVICE_EXECUTABLE	Karaf executable name (defaul karaf support daemon and stop commands)

```
CONF_TEMPLATE="karaf-service-template.conf"
SYSTEMD_TEMPLATE="karaf-service-template.systemd"
SYSTEMD_TEMPLATE_INSTANCES="karaf-service-template.systemd-instances"
INIT_TEMPLATE="karaf-service-template.init"
INIT_REDHAT_TEMPLATE="karaf-service-template.init-redhat"
INIT_DEBIAN_TEMPLATE="karaf-service-template.init-debian"
SOLARIS_SMF_TEMPLATE="karaf-service-template.solaris-smf"
```

2.1.3. Systemd

When the ***karaf-service.sh*** utility identifies Systemd, it generates three files:

- a systemd unit file to manage the root Apache Karaf container
- a systemd environment file with variables used by the root Apache Karaf container
- a systemd template unit file to manage Apache Karaf child containers

Here is an example:

```
$ ./karaf-service.sh -k /opt/karaf-4 -n karaf-4
Writing service file "/opt/karaf-4/bin/contrib/karaf-4.service"
Writing service configuration file ""/opt/karaf-4/etc/karaf-4.conf"
Writing service file "/opt/karaf-4/bin/contrib/karaf-4@.service"
$ cp /opt/karaf-4/bin/contrib/karaf-4.service /etc/systemd/system
$ cp /opt/karaf-4/bin/contrib/karaf-4@.service /etc/systemd/system
$ systemctl enable karaf-4.service
```

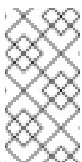
2.1.4. SysV

When the ***karaf-service.sh*** utility identifies a SysV system, it generates two files:

- an init script to manage the root Apache Karaf container
- an environment file with variables used by the root Apache Karaf container

Here is an example:

```
$ ./karaf-service.sh -k /opt/karaf-4 -n karaf-4
Writing service file "/opt/karaf-4/bin/contrib/karaf-4"
Writing service configuration file "/opt/karaf-4/etc/karaf-4.conf"
$ ln -s /opt/karaf-4/bin/contrib/karaf-4 /etc/init.d/
$ chkconfig karaf-4 on
```



Note

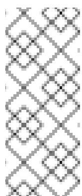
To enable the service startup upon boot, Refer your operating system init guide.

2.1.5. Solaris SMF

When the karaf-service.sh utility identifies a Solaris operating system, it generates a single file.

Here is an example:

```
$ ./karaf-service.sh -k /opt/karaf-4 -n karaf-4
Writing service file "/opt/karaf-4/bin/contrib/karaf-4.xml"
$ svccfg validate /opt/karaf-4/bin/contrib/karaf-4.xml
$ svccfg import /opt/karaf-4/bin/contrib/karaf-4.xml
```



Note

The generated SMF descriptor is defined as transient, so that you can execute the start method only once.

2.1.6. Windows

Installation of Apache Karaf as windows service is supported through winsw.

To install Apache Karaf as windows service, perform the following:

- Rename the **`karaf-service-win.exe`** file to match with **`karaf-4.exe`** service name.
- Rename the **`karaf-service-win.xml`** file to match with **`karaf-4.xml`** service name.
- Customize the service descriptor as per your requirements.
- Use the service executable to install, start and stop the service.

Here is an example:

```
C:\opt\apache-karaf-4\bin\contrib> karaf-4.exe install
C:\opt\apache-karaf-4\bin\contrib> karaf-4.exe start
```

CHAPTER 3. BASIC SECURITY

Abstract

This chapter describes the basic steps to configure security before you start Red Hat JBoss Fuse for the first time. By default, JBoss Fuse is secure, but none of its services are remotely accessible. This chapter explains how to enable secure access to the ports exposed by JBoss Fuse.

3.1. CONFIGURING BASIC SECURITY

Overview

The Red Hat JBoss Fuse runtime is secured against network attack by default, because all of its exposed ports require user authentication and no users are defined initially. In other words, the Red Hat JBoss Fuse runtime is remotely inaccessible by default.

If you want to access the runtime remotely, you must first customize the security configuration, as described here.

Before you start the container

If you want to enable remote access to the JBoss Fuse container, you must create a secure JAAS user before starting the container:

Create a secure JAAS user

By default, no JAAS users are defined for the container, which effectively disables remote access (it is impossible to log on).

To create a secure JAAS user, edit the ***InstallDir*/etc/users.properties** file and add a new user field, as follows:

```
Username=Password,Administrator
```

Where ***Username*** and ***Password*** are the new user credentials. The **Administrator** role gives this user the privileges to access all administration and management functions of the container. For more details about JAAS, see Chapter 14, *Configuring JAAS Security*.

Do not define a numeric username with a leading zero. Such usernames will always cause a login attempt to fail. This is because the Karaf shell, which the console uses, drops leading zeros when the input appears to be a number. For example:

```
FuseMQ:karaf@root> echo 0123
123
FuseMQ:karaf@root> echo 00.123
0.123
```

```
FuseMQ:karaf@root>
```

> **Warning**
>
> It is strongly recommended that you define custom user credentials with a strong password.

Role-based access control

The JBoss Fuse container supports role-based access control, which regulates access through the JMX protocol, the Karaf command console, and the Fuse Management console. When assigning roles to users, you can choose from the set of standard roles, which provide the levels of access described in Table 3.1, “Standard Roles for Access Control”.

Table 3.1. Standard Roles for Access Control

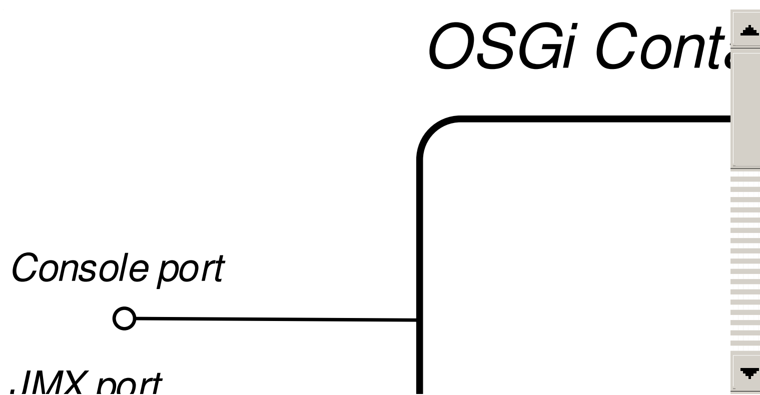
Roles	Description
`Monitor`, `Operator`, `Maintainer`	Grants read-only access to the container.
`Deployer`, `Auditor`	Grants read-write access at the appropriate level for ordinary users, who want to deploy and run applications. But blocks access to sensitive container configuration settings.
`Administrator`, `SuperUser`	Grants unrestricted access to the container.

For more details about role-based access control, see section "Role-Based Access Control" in "Security Guide".

Ports exposed by the JBoss Fuse container

Figure 3.1, “Ports Exposed by the JBoss Fuse Container” shows the ports exposed by the JBoss Fuse container by default.

Figure 3.1. Ports Exposed by the JBoss Fuse Container



The following ports are exposed by the container:

- *Console port*—enables remote control of a container instance, through Apache Karaf shell commands. This port is enabled by default and is secured both by JAAS authentication and by SSH.
- *JMX port*—enables management of the container through the JMX protocol. This port is enabled by default and is secured by JAAS authentication.
- *Web console port*—provides access to an embedded Jetty container that can host Web console servlets. By default, the Fuse Management Console is installed in the Jetty container.

Enabling the remote console port

You can access the remote console port whenever both of the following conditions are true:

- JAAS is configured with at least one set of login credentials.
- The JBoss Fuse runtime has *not* been started in client mode (client mode disables the remote console port completely).

For example, to log on to the remote console port from the same machine where the container is running, enter the following command:

```
./client -u Username -p Password
```

Where the ***Username*** and ***Password*** are the credentials of a JAAS user with **Administrator** privileges. For more details, see Chapter 8, *Using Remote Connections to Manage a Container*.

Strengthening security on the remote console port

You can employ the following measures to strengthen security on the remote console port:

- Make sure that the JAAS user credentials have strong passwords.

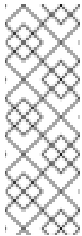
- Customize the X.509 certificate (replace the Java keystore file, ***InstallDir*/etc/host.key**, with a custom key pair).

Enabling the JMX port

The JMX port is enabled by default and secured by JAAS authentication. In order to access the JMX port, you must have configured JAAS with at least one set of login credentials. To connect to the JMX port, open a JMX client (for example, **jconsole**) and connect to the following JMX URI:

```
service:jmx:rmi:///jndi/rmi://localhost:1099/karaf-root
```

You must also provide valid JAAS credentials to the JMX client in order to connect.



Note

In general, the tail of the JMX URI has the format **/karaf-*ContainerName***. If you change the container name from **root** to some other name, you must modify the JMX URI accordingly.

Strengthening security on the Fuse Management Console port

The Fuse Management Console is already secured by JAAS authentication. To add SSL security, see chapter "Securing the Jetty HTTP Server" in "Security Guide".

3.2. DISABLING BROKER SECURITY

Overview

Prior to Fuse ESB Enterprise version 7.0.2, the Apache ActiveMQ broker was insecure (JAAS authentication not enabled). This section explains how to revert the Apache ActiveMQ broker to an insecure mode of operation, so that it is unnecessary to provide credentials when connecting to the broker.

Warning

After performing the steps outlined in this section, the broker has no protection against hostile clients. This type of configuration is suitable only for use on internal, trusted networks.

Standalone server

These instructions assume that you are running Red Hat JBoss Fuse in standalone mode (that is, running in an OSGi container, but not using Fuse Fabric). In your installation of JBoss Fuse, open the ***InstallDir*/etc/broker.xml** file using a text editor and look for the following lines:

```
...
<plugins>
 <jaasAuthenticationPlugin configuration="karaf" />
</plugins>
...
```

To disable JAAS authentication, delete (or comment out) the **`jaasAuthenticationPlugin`** element. The next time you start up the Red Hat JBoss Fuse container (using the ***`InstallDir`*****`/bin/fusemq`** script), the broker will run with unsecured ports.

CHAPTER 4. STARTING AND STOPPING JBOSS FUSE

Abstract

Red Hat JBoss Fuse provides simple command-line tools for starting and stopping the server.

4.1. STARTING JBOSS FUSE

Abstract

The default way for deploying the Red Hat JBoss Fuse runtime is to deploy it as a standalone server with an active console. You can also deploy the runtime as a background process without a console.

Overview

The default way for deploying the Red Hat JBoss Fuse runtime is to deploy it as a standalone server with an active console. You can also deploy the runtime to run as a background process without a console.

Setting up your environment

You can start the JBoss Fuse runtime directly from the **bin** subdirectory of your installation, without modifying your environment. However, if you want to start it in a different folder you will need to add the **bin** directory of your JBoss Fuse installation to the **PATH** environment variable, as follows:

Windows

set PATH=%PATH%;*InstallDir*\bin

Linux/UNIX

export PATH=$PATH,*InstallDir*/bin

Launching the runtime in console mode

If you are launching the JBoss Fuse runtime from the installation directory use the following command:

Windows

```
bin\fuse.bat
```

Linux/UNIX

```
./bin/fuse
```

If JBoss Fuse starts up correctly you should see the following on the console:

```
      _ ____                  ______
     | |  _ \                |  ____|
     | | |_) | ___  ___ ___  | |__ _   _ ___  ___
 _   | |  _   / _ \/ __/ __| |  __| | | / __|/ _ \
| |__| | |_) | (_) \__ \__ \ | |  | |_| \__ \  __/
 \____/|____/ \___/|___/___/ |_|   \__,_|___/\___|


  JBoss Fuse (6.3.0.redhat-187)
  http://www.redhat.com/products/jbossenterprisemiddleware/fuse/

Hit '<tab>' for a list of available commands
and '[cmd] --help' for help on a specific command.
Hit '<ctrl-d>' or 'osgi:shutdown' to shutdown JBoss Fuse.

JBossFuse:karaf@root>
```

Note

Since version JBoss Fuse 6.2.1, launching in console mode creates two processes: the parent process **`./bin/karaf`**, which is executing the Karaf console; and the child process, which is executing the Karaf server in a **`java`** JVM. The shutdown behaviour remains the same as before, however. That is, you can shut down the server from the console using either Ctrl-D or **`osgi:shutdown`**, which kills both processes.

Launching the runtime in server mode

Launching in server mode runs Red Hat JBoss Fuse in the background, without a local console. You would then connect to the running instance using a remote console. See Section 8.2, “ Connecting and Disconnecting Remotely” for details.

To launch JBoss Fuse in server mode, run the following

Windows

```
bin\start.bat
```

Linux/UNIX

```
./bin/start
```

Launching the runtime in client mode

In production environments you might want to have a runtime instance accessible using only a local

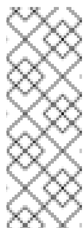
console. In other words, you cannot connect to the runtime remotely through the SSH console port. You can do this by launching the runtime in client mode, using the following command:

Windows

```
bin\fuse.bat client
```

Linux/UNIX

```
./bin/fuse client
```



Note

Launching in client mode suppresses only the SSH console port (usually port 8101). Other Karaf server ports (for example, the JMX management RMI ports) are opened as normal.

4.2. STOPPING JBOSS FUSE

Abstract

You can stop an instance of Red Hat JBoss Fuse either from within a console, or using a **stop** script.

Stopping an instance from a local console

If you launched the Karaf instance by running **fuse** or **fuse client**, you can stop it by doing one of the following at the **karaf>** prompt:

- Type **shutdown**
- Press **Ctrl**+**D**

Stopping an instance running in server mode

You can stop a locally running Karaf instance (root container), by invoking the **stop(.bat)** from the ***InstallDir*/bin** directory, as follows:

Windows

```
bin\stop.bat
```

Linux/UNIX

```
./bin/stop
```

-

The shutdown mechanism invoked by the Karaf **stop** script is similar to the shutdown mechanism implemented in Apache Tomcat. The Karaf server opens a dedicated shutdown port (*not* the same as the SSH port) to receive the shutdown notification. By default, the shutdown port is chosen randomly, but you can configure it to use a specific port if you prefer.

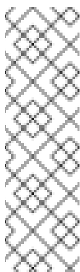
You can optionally customize the shutdown port by setting the following properties in the ***InstallDir*/etc/config.properties** file:

karaf.shutdown.port

Specifies the TCP port to use as the shutdown port. Setting this property to **-1** disables the port. Default is **0** (for a random port).

karaf.shutdown.host

Specifies the hostname to which the shutdown port is bound. This setting could be useful on a multi-homed host. Defaults to **localhost**.

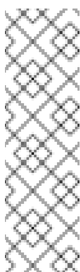


Note

If you wanted to use the **bin/stop** script to shut down the Karaf server running on a remote host, you would need to set this property to the hostname (or IP address) of the remote host. But beware that this setting also affects the Karaf server located on the same host as the **etc/config.properties** file.

karaf.shutdown.port.file

After the Karaf instance starts up, it writes the current shutdown port to the file specified by this property. The **stop** script reads the file specified by this property to discover the value of the current shutdown port. Defaults to **${karaf.data}/port**.



Note

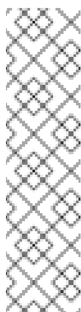
If you wanted to use the **bin/stop** script to shut down the Karaf server running on a remote host, you would need to set this property equal to the remote host's shutdown port. But beware that this setting also affects the Karaf server located on the same host as the **etc/config.properties** file.

karaf.shutdown.command

Specifies the UUID value that must be sent to the shutdown port in order to trigger shutdown. This provides an elementary level of security, as long as the UUID value is kept a secret. For example, the **etc/config.properties** file could be read-protected to prevent this value from being read by ordinary users.

When Apache Karaf is started for the very first time, a random UUID value is automatically

generated and this setting is written to the end of the **etc/config.properties** file. Alternatively, if **karaf.shutdown.command** is already set, the Karaf server uses the pre-existing UUID value (which enables you to customize the UUID setting, if required).



Note

If you wanted to use the **bin/stop** script to shut down the Karaf server running on a remote host, you would need to set this property to be equal to the value of the remote host's **karaf.shutdown.command**. But beware that this setting also affects the Karaf server located on the same host as the **etc/config.properties** file.

Stopping a child container instance

Apache Karaf enables you to create and manage child container instances using the **admin** family of console commands (for example, using **admin:create**, **admin:start**, **admin:stop**, and so on). To stop the ***ChildContainerName*** child container running on your local host, invoke the **admin(.bat)** script from the command line, as follows:

Windows

```
bin\admin.bat stop ChildContainerName
```

Linux/UNIX

```
./bin/admin stop ChildContainerName
```

Stopping a remote instance

You can stop a container instance running on a remote host as described in Section 8.3, “Stopping a Remote Container”.

CHAPTER 5. CREATING A NEW FABRIC

Abstract

When there are no existing fabrics to join, or you want to start a new fabric, you can create a new one from a standalone container.

STATIC IP ADDRESS REQUIRED FOR FABRIC SERVER

The IP address and hostname associated with the Fabric Servers in the Fabric ensemble are of critical importance to the fabric. Because these IP addresses and hostnames are used for configuration and service discovery (through the Zookeeper registry), they *must not change* during the lifetime of the fabric.

You can take either of the following approaches to specifying the IP address:

- For simple examples and tests (with a single Fabric Server) you can work around the static IP requirement by using the loopback address, **`127.0.0.1`**.
- For distributed tests (multiple Fabric Servers) and production deployments, you *must* assign a static IP address to each of the Fabric Server hosts.

> **Warning**
>
> Beware of volatile IP addresses resulting from VPN connections, WiFi connections, and even LAN connections. If a Fabric Server binds to one of these volatile IP addresses, it will cease to function after the IP address has gone away. It is recommended that you always use the **`--resolver manualip --manual-ip`** ***`StaticIPAddress`*** options to specify the static IP address explicitly, when creating a new Fabric Server.

MAKE QUICKSTART EXAMPLES AVAILABLE

The default behavior is that profiles for quickstart examples are not available in a new fabric. To create a fabric in which you can run the quickstart examples, edit the **`$FUSE_HOME/fabric/io.fabric8.import.profiles.properties`** file by uncommenting the line that starts with the following:

```
# importProfileURLs =
```

If you create a fabric without doing this and you want to run the quickstart examples, follow these steps to make them available:

1. Edit the **`$FUSE_HOME/quickstarts/pom.xml`** file to add a fabric I/O plugin, for example:

   –

```
<plugin>
     <groupId>io.fabric8</groupId>
     <artifactId>fabric8-maven-plugin</artifactId>
     <version>1.2.0.redhat-630187</version>
</plugin>
```

2. In the **$FUSE_HOME/quickstarts** directory, change to the directory for the quickstart example you want to run, for example:

   cd beginner

3. In that directory, execute the following command:

   mvn fabric8:deploy

   You would need to run this command in each directory that contains a quickstart example that you want to run.

PROCEDURE

To create a new fabric:

1. *(Optional)* Customise the name of the root container by editing the ***InstallDir*/etc/system.properties** file and specifying a different name for this property:

   ```
   karaf.name=root
   ```

   Note

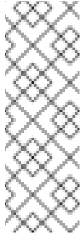
   For the first container in your fabric, this step is optional. But at some later stage, if you want to join a root container to the fabric, you might need to customise the container's name to prevent it from clashing with any existing root containers in the fabric.

2. Any existing users in the ***InstallDir*/etc/users.properties** file are automatically used to initialize the fabric's user data, when you create the fabric. You can populate the **users.properties** file, by adding one or more lines of the following form:

   ```
   Username=Password[,RoleA][,RoleB]...
   ```

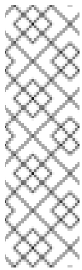
   But there must *not* be any users in this file that have administrator privileges (**Administrator**, **SuperUser**, or **admin** roles). If the ***InstallDir*/etc/users.properties** already contains users with administrator privileges, you should *delete those users* before creating the fabric.

> **Warning**
>
> If you leave some administrator credentials in the **`users.properties`** file, this represents a security risk because the file could potentially be accessed by other containers in the fabric.



> **Note**
>
> The initialization of user data from **`users.properties`** happens only once, at the time the fabric is created. After the fabric has been created, any changes you make to **`users.properties`** will have *no effect* on the fabric's user data.

3. If you use a VPN (virtual private network) on your local machine, it is advisable to log off VPN *before* you create the fabric and to *stay logged off* while you are using the local container.



> **Note**
>
> A local Fabric Server is permanently associated with a fixed IP address or hostname. If VPN is enabled when you create the fabric, the underlying Java runtime is liable to detect and use the VPN hostname instead of your permanent local hostname. This can also be an issue with multi-homed machines.

4. Start up your local container.

   In JBoss Fuse, start the local container as follows:

```
cd InstallDir/bin
./fuse
```

5. Create a new fabric by entering the following command:

```
JBossFuse:karaf@root> fabric:create --new-user AdminUser --new-user-password AdminPass --new-user-role Administrator --zookeeper-password ZooPass --resolver manualip --manual-ip StaticIPAddress --wait-for-provisioning
```

   The current container, named **`root`** by default, becomes a Fabric Server with a registry service installed. Initially, this is the only container in the fabric. The **`--new-user`**, **`--new-user-password`**, and **`--new-user-role`** options specify the credentials for a new **`Administrator`** user. The Zookeeper password is used to protect sensitive data in the Fabric registry service (all of the nodes under **`/fabric`**). The **`--manual-ip`** option specifies the Fabric Server's static IP address ***`StaticIPAddress`*** (see the section called “Static IP address required for Fabric Server”).

   For more details on **`fabric:create`** see section "fabric:create" in "Console Reference".

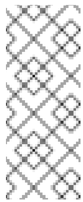
For more details about resolver policies, see section "fabric:container-resolver-list" in "Console Reference" and section "fabric:container-resolver-set" in "Console Reference".

FABRIC CREATION PROCESS

Several things happen when a fabric is created from a standalone container:

1. The container installs the requisite OSGi bundles to become a Fabric Server.

2. The Fabric Server starts a registry service, which listens on TCP port 2181 (which makes fabric configuration data available to all of the containers in the fabric).

   Note

   You can customize the value of the registry service port by specifying the **`--zookeeper-server-port`** option.

3. The Fabric Server installs a new JAAS realm (based on the ZooKeeper login module), which overrides the default JAAS realm and stores its user data in the ZooKeeper registry.

4. The new Fabric Ensemble consists of a *single* Fabric Server (the current container).

5. A default set of profiles is imported from ***`InstallDir`*`/fabric/import`** (can optionally be overridden).

6. After the standalone container is converted into a Fabric Server, the previously installed OSGi bundles and Karaf features are completely cleared away and replaced by the default Fabric Server configuration. For example, some of the shell command sets that were available in the standalone container are no longer available in the Fabric Server.

EXPANDING A FABRIC

You can expand a fabric by creating new managed containers. Fabric supports the *container provider* plug-in mechanism, which makes it possible to define how to create new containers in different contexts. Currently, Fabric makes container providers available for the following kinds of container:

- *Child container*, created on the local machine as a child process in its own JVM.

  Instructions on creating a child container are found in Child Containers.

- *SSH container*, created on any remote machine for which you have **`ssh`** access.

  Instructions on creating a SSH container are found in SSH Containers.

- *Cloud container*, created on compute instance in the cloud.

  Instructions on creating a cloud container are found in Cloud Containers.

Fabric provides container creation commands that make it easy to create new containers. Using these commands, Fabric can automatically install JBoss Fuse on a remote host (uploading whatever dependencies are needed), start up the remote container process, and join the container to the existing fabric, so that it becomes a fully-fledged managed container in the fabric.

CHAPTER 6. JOINING A FABRIC

OVERVIEW

Any standalone container can be joined to an existing fabric using the **fabric:join** console command. You need to supply the URL of one of the Fuse Servers in the fabric and the standalone container is then added to the fabric. The container can join the fabric as either a managed container or a non-managed container:

- A *managed container* is a full member of the fabric and is managed by a Fabric Agent. The agent configures the container based on information provided by the fabric's ensemble. The ensemble knows which profiles are associated with the container and the agent determines what to install based on the contents of the profiles.

- A *non-managed container* is not managed by a Fabric Agent. Its configuration remains intact after it joins the fabric and is controlled as if the container were a standalone container. Joining the fabric in this manner registers the container with the fabric's ensemble and allows clients to locate the services running in the container using the fabric's discovery mechanism.

JOINING A FABRIC AS A MANAGED CONTAINER

The default behavior of the **fabric:join** command is to wipe out the container's configuration and replace it with the **fabric** profile. If you want to preserve the previous configuration of the container, however, you must ensure that the fabric has an appropriately configured *profile*, which you can deploy into the container after it joins the fabric.

The **fabric:join** command's **-p** option enables you to specify a profile to install into the container once the agent is installed.

For details of how to create and edit a profile, see , section "fabric:profile-create" in "Console Reference", and section "fabric:profile-edit" in "Console Reference".

JOINING A FABRIC AS AN NON-MANAGED CONTAINER

When a container joins a fabric as a non-managed container, its deployment mechanisms continue to function like a standalone container (based on **osgi:install**, **features:install**, and hot deployment), because a Fabric Agent does *not* take control of its configuration. The agent only registers the container with the fabric's ensemble and keeps the registry entries for it up to date. This enables the newly joined container to discover services running in the container (through Fabric's discovery mechanisms) and to administer these services.

Joining a fabric as an non-managed container is a convenient approach to take when you want to use your local container as a console to administer a fabric. For example, this is an approach that is typically taken with the Fuse Management Console (FMC).

HOW TO JOIN A FABRIC

To join a container to a fabric, perform the following steps:

1. Get the registry service URL for one of the Fabric Servers in the existing fabric. The registry service URL has the following format:

   ```
   Hostname[:IPPort]
   ```

   Normally, it is sufficient to specify just the hostname, *Hostname*, because the registry service uses the fixed port number, 2182, by default. In exceptional cases, you can discover the registry service port by following the instructions in the section called “How to discover the URL of a Fabric Server”.

2. Get the ZooKeeper password for the fabric. An administrator can access the fabric's ZooKeeper password at any time, by entering the following console command (while logged into one of the Fabric Containers):

   ```
   JBossFuse:karaf@root> fabric:ensemble-password
   ```

3. Connect to the standalone container's command console.

4. Join a container in one of the following ways:

   - *Join as a managed container, with a default profile*—uses the **fabric** profile.

     ```
     JBossFuse:karaf@root> fabric:join --zookeeper-password ZooPass
     URL ContainerName
     ```

   - *Join as a managed container, specifying a custom profile*—uses a custom profile.

     ```
     JBossFuse:karaf@root> fabric:join --zookeeper-password ZooPass -p
     Profile URL ContainerName
     ```

   - *Join as a non-managed container*—preserves the existing container configuration.

     ```
     JBossFuse:karaf@root> fabric:join -n --zookeeper-password ZooPass
     URL ContainerName
     ```

   Where you can specify the following values:

   ZooPass

   The existing fabric's ZooKeeper password.

   URL

   The URL for one of the fabric's registry services (usually just the hostname where a Fabric Server is running).

`ContainerName`

The new name of the container when it registers itself with the fabric.

> **Warning**
>
> If the container your're adding to the fabric has the same name as a container already registered with the fabric, both containers will be reset and will always share the same configuration.

`Profile`

The name of the custom profile to install into the container after it joins the fabric (managed container only).

5. If you joined the container as a *managed container*, you can subsequently deploy a different profile into the container using the **`fabric:container-change-profile`** console command.

HOW TO DISCOVER THE URL OF A FABRIC SERVER

If you suspect that a Fabric Server is not using the default TCP port, 2181, for its registry service, you can discover the port as follows:

1. Connect to the command console of one of the containers in the fabric.

2. Enter the following sequence of console commands:

```
JBossA-MQ:karaf@root> config:edit io.fabric8.zookeeper
JBossA-MQ:karaf@root> config:proplist
   service.pid = io.fabric8.zookeeper
   zookeeper.url =
myhostA:2181,myhostB:2181,myhostC:2181,myhostC:2182,myhostC:2183
   fabric.zookeeper.pid = io.fabric8.zookeeper
JBossA-MQ:karaf@root> config:cancel
```

The **`zookeeper.url`** property holds a comma-separated list of Fabric Server URLs. You can use any one of these URLs to join the fabric.

CHAPTER 7. SHUTTING DOWN A FABRIC

OVERVIEW

This chapter describes how to shut down part or all of a fabric. In particular, when shutting down the ensemble servers (Fabric Servers), special care is needed.

SHUTTING DOWN A MANAGED CONTAINER

From the console, you can shut down a managed container at any time using the **fabric:container-stop** command, specifying the name of the managed container—for example:

```
fabric:container-stop ManagedContainerName
```

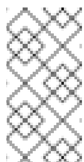
The **fabric:container-stop** command looks up the container name in the registry and retrieves the data it needs to shut down that container. This approach works no matter where the container is deployed: whether on a remote host or in a cloud.

SHUTTING DOWN A FABRIC SERVER

Occasionally, you might want to shut down a Fabric Server for maintenance reasons. It is possible to do this without disabling the fabric, as long as you ensure *more than half of the Fabric Servers in the ensemble remain up and running*.

For example, if you have an ensemble consisting of three servers, **registry1**, **registry2**, and **registry3**, you can shut down at most one of these Fabric Servers at a time using the **fabric:container-stop** command—for example:

```
fabric:container-stop -f registry3
```



Note

The **-f** flag is required when shutting down a container that belongs to the ensemble.

After performing the necessary maintenance, you can restart the Fabric Server as follows:

```
fabric:container-start registry3
```

SHUTTING DOWN AN ENTIRE FABRIC

In a production environment, it is rarely ever necessary to shut down an entire fabric. The idea of a fabric is that it provides redundancy, enabling you to shut down part of the fabric and restart it *without* having to shut down the whole fabric. You can even apply patches to a fabric without shutting down containers.

If you do need to shut down an entire fabric, however, you can do so as follows:

1. To take a concrete example, consider a fabric which consists of the following containers:

   - Three Fabric Servers (ensemble servers): **`registry1`**, **`registry2`**, **`registry3`**.
   - Four managed containers: **`managed1`**, **`managed2`**, **`managed3`**, **`managed4`**.

2. Use the **`client`** console utility to log on to one of the containers in the fabric. Because this will be the last container to shut down, it is convenient to choose one of the Fabric Servers. For example, to log on to the **`registry1`** server, enter the following command:

   ```
   ./client -u AdminUser -p AdminPass -h Registry1Host
   ```

   Where ***`Registry1Host`*** is the host where **`registry1`** is running and ***`AdminUser`*** and ***`AdminPass`*** are the credentials of a user with administration privileges. It is assumed that the **`registry1`** server is listening for console connections on the default TCP port (that is, **`8101`**)

3. Shut down all of the managed containers in the fabric, using the **`fabric:container-stop`** command—for example:

   ```
   fabric:container-stop managed1
   fabric:container-stop managed2
   fabric:container-stop managed3
   fabric:container-stop managed4
   ```

4. Remove all but one of the Fabric Servers from the ensemble, using the **`fabric:ensemble-remove`** command. For example, given the ensemble consisting of **`registry1`**, **`registry2`**, and **`registry3`** (where you are logged on to **`registry1`**), remove **`registry2`** and **`registry3`** from the ensemble as follows:

   ```
   fabric:ensemble-remove registry2 registry3
   ```

5. You can now shut down the **`registry2`** and **`registry3`** containers using the **`fabric:container-stop`** command, as follows:

   ```
   fabric:container-stop registry2
   fabric:container-stop registry3
   ```

6. Assuming you are logged on to **`registry1`** (the sole remaining Fabric Server), shut it down as follows:

   ```
   shutdown -f
   ```

7. Whenever you restart the fabric, you will have to remember to recreate the ensemble, so that it consists of three Fabric Servers again. For example, to recreate the ensemble

consisting of **registry1**, **registry2**, and **registry3**, you would restart the three servers, and then enter the following command:

```
fabric:ensemble-add registry2 registry3
```

NOTE ON SHUTTING DOWN THE ENSEMBLE

If you are logged on to a container that is connected to a fabric, the following obvious procedure for shutting down the fabric ensemble does *not* work:

```
fabric:container-stop registry1
fabric:container-stop registry2
fabric:container-stop registry3
```

The third invocation of **fabric:container-stop** will fail and throw an error. This is because of the quorum-based voting system used by the ensemble (which is designed to protect against network splits). After the first two Fabric servers (**registry1** and **registry2**) are shut down, fewer than half of the ensemble servers are available. At this point, the registry shuts down and refuses to service any more requests, because there is no longer a quorum of ensemble servers available (that is, fewer than 50% of the ensemble servers are available). This causes a problem for the **fabric:container-stop** command, which normally contacts the registry to retrieve details about the container it is trying to shut down.

The solution to this problem is to adopt the procedure described in the section called “Shutting down an entire fabric”, where you first reduce the size of the ensemble using **fabric:ensemble-remove**, before attempting to shut down the ensemble servers.

CHAPTER 8. USING REMOTE CONNECTIONS TO MANAGE A CONTAINER

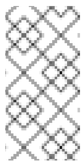
Abstract

It does not always make sense to use a local console to manage a container. Red Hat JBoss Fuse has a number of ways of remotely managing a container. You can use a remote container's command console or start a remote client.

8.1. CONFIGURING A CONTAINER FOR REMOTE ACCESS

Overview

When you start the Red Hat JBoss Fuse runtime in default mode or in server mode, it enables a remote console that can be accessed over SSH from any other JBoss Fuse console. The remote console provides all of the functionality of the local console and allows a remote user complete control over the container and the services running inside of it.



Note

When run in client mode the JBoss Fuse runtime disables the remote console.

Configuring a standalone container for remote access

The SSH hostname and port number are configured in the ***InstallDir*/etc/org.apache.karaf.shell.cfg** configuration file. Example 8.1, "Changing the Port for Remote Access" shows a sample configuration that changes the port used to 8102.

Example 8.1. Changing the Port for Remote Access

```
sshPort=8102
sshHost=0.0.0.0
```

Configuring a fabric container for remote access

The SSH hostname and port number are set at the time a container is created. It is *not* possible to change the SSH address of a Fabric container after the container has been created.

8.2. CONNECTING AND DISCONNECTING REMOTELY

Abstract

There are two alternative ways of connecting to a remote container. If you are already running an Red Hat JBoss Fuse command shell, you can invoke a console command to connect to the remote container. Alternatively, you can run a utility directly on the command-line to connect to the remote container.

8.2.1. Connecting to a Standalone Container from a Remote Container

Overview

Any container's command console can be used to access a remote container. Using SSH, the local container's console connects to the remote container and functions as a command console for the remote container.

Using the ssh:ssh console command

You connect to a remote container's console using the **`ssh:ssh`** console command.

Example 8.2. ssh:ssh Command Syntax

`ssh:ssh` { -l *username* } { -P *password* } { -p *port* } { *hostname* }

`-l` ***`username`***

The username used to connect to the remote container. Use valid JAAS login credentials that have **`admin`** privileges (see Chapter 14, *Configuring JAAS Security*).

`-P` ***`password`***

The password used to connect to the remote container.

`-p` ***`port`***

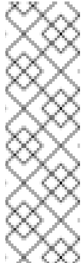
The SSH port used to access the desired container's remote console.

By default this value is **`8101`**. See the section called “Configuring a standalone container for remote access” for details on changing the port number.

`hostname`

The hostname of the machine that the remote container is running on. See the section called “Configuring a standalone container for remote access” for details on changing the hostname.

> **Warning**
>
> We recommend that you customize the username and password in the **`etc/users.properties`** file. See Chapter 14, *Configuring JAAS Security* for details.



Note

If your remote container is deployed on an *Oracle VM Server for SPARC* instance, it is likely that the default SSH port value, 8101, is already occupied by the Logical Domains Manager daemon. In this case, you will need to reconfigure the container's SSH port, as described in the section called “Configuring a standalone container for remote access”.

Example 8.3. Connecting to a Remote Console

```
JBossFuse:karaf@root>ssh:ssh -l smx -P smx -p 8108 hostname
```

To confirm that you have connected to the correct container, type **`shell:info`** at the prompt. Information about the currently connected instance is returned, as shown in Example 8.4, “Output of the shell:info Command”.

Example 8.4. Output of the shell:info Command

```
Karaf Karaf version 2.2.5.fuse-beta-7-052 Karaf home
/Volumes/ESB/jboss-fuse-full-6.0.0.redhat-0XX Karaf base
/Volumes/ESB/jboss-fuse-full-6.0.0.redhat-0XX/instances/child1 OSGi
Framework org.apache.felix.framework - 4.0.3.fuse-beta-7-052 JVM Java
Virtual Machine Java HotSpot(TM) 64-Bit Server VM version 20.6-b01-
415 Version 1.6.0_31 Vendor Apple Inc. Uptime 6 minutes Total
compile time 24.048 seconds Threads Live threads 62 Daemon threads
43 Peak 287 Total started 313 Memory Current heap size 78,981 kbytes
Maximum heap size 466,048 kbytes Committed heap size 241,920 kbytes
Pending objects 0 Garbage collector Name = 'PS Scavenge', Collections
= 11, Time = 0.271 seconds Garbage collector Name = 'PS MarkSweep',
Collections = 1, Time = 0.117 seconds Classes Current classes loaded
5,720 Total classes loaded 5,720 Total classes unloaded 0 Operating
system Name Mac OS X version 10.7.3 Architecture x86_64 Processors 2
```

Disconnecting from a remote console

To disconnect from a remote console, enter **`logout`** or press **`Ctrl`**+**`D`** at the prompt.

You will be disconnected from the remote container and the console will once again manage the local container.

8.2.2. Connecting to a Fabric Container From another Fabric Container

Overview

When containers are deployed into a fabric, they are all connected to each other. You can easily connect to any container's command console from any of its peers. When connecting using fabric, you do not need to know any of the location details for the container you want to connect to. The fabric's runtime registry stores all of the location details needed to establish the remote connection.

Using the fabric:container-connect command

In the context of a fabric, you should connect to a remote runtime's console using the **`fabric:container-connect`** command.

Example 8.5. fabric:container-connect Command Syntax

`fabric:container-connect` { -u *username* } { -p *password* } { *containerName* }

`-u` ***`username`***

The username used to connect to the remote console. The default value is **`admin`**.

`-p` ***`password`***

The password used to connect to the remote console. The default value is **`admin`**.

`containerName`

The name of the container.

> **Warning**
>
> We recommend that you change the default administrator username and password. See *Chapter 14, Configuring JAAS Security* for details.

Example 8.6. Connecting to a Remote Container

```
JBossFuse:karaf@root>fabric:container-connect -u admin -p admin
containerName
```

To confirm that you have connected to the correct container, type **shell:info** at the prompt. Information about the currently connected instance is returned, as shown in Example 8.7, “Output of the shell:info Command”.

Example 8.7. Output of the shell:info Command

```
Karaf Karaf version 2.3.0.fuse-71-044 Karaf home
/Volumes/SAMSUNG/Programs/ESB/jboss-fuse-full-6.0.0.redhat-0XX Karaf
base /Volumes/SAMSUNG/Programs/ESB/jboss-fuse-full-6.0.0.redhat-
0XX/instances/child1 OSGi Framework org.apache.felix.framework -
4.0.3.fuse-71-044 JVM Java Virtual Machine Java HotSpot(TM) 64-Bit
Server VM version 20.8-b03-424 Version 1.6.0_33 Vendor Apple Inc.
Uptime 7 minutes Total compile time 5.336 seconds Threads Live
threads 42 Daemon threads 31 Peak 96 Total started 123 Memory
Current heap size 32,832 kbytes Maximum heap size 466,048 kbytes
Committed heap size 104,960 kbytes Pending objects 0 Garbage
collector Name = 'PS Scavenge', Collections = 7, Time = 0.063 seconds
Garbage collector Name = 'PS MarkSweep', Collections = 1, Time =
0.060 seconds Classes Current classes loaded 4,019 Total classes
loaded 4,019 Total classes unloaded 0 Operating system Name Mac OS X
version 10.7.4 Architecture x86_64 Processors 2
```

Disconnecting from a remote console

To disconnect from a remote console, enter **logout** or press **Ctrl+D** at the prompt.

You will be disconnected from the remote container and the console will once again manage the local container.

8.2.3. Connecting to a Container Using the Client Command-Line Utility

Using the remote client

The remote client allows you to securely connect to a remote Red Hat JBoss Fuse container without having to launch a full JBoss Fuse container locally.

For example, to quickly connect to a JBoss Fuse instance running in server mode on the same machine, open a command prompt and run the **client[.bat]** script (which is located in the ***InstallDir*/bin** directory), as follows:

```
client
```

More usually, you would provide a hostname, port, username, and password to connect to a remote instance. If you were using the client within a larger script, for example in a test suite, you could append console commands as follows:

```
client -a 8101 -h hostname -u username -p password shell:info
```

Alternatively, if you omit the **-p** option, you will be prompted to enter a password.

For a standalone container, use any valid JAAS user credentials that have **admin** privileges.

For a container in a fabric, the default username and password is **admin** and **admin**.

To display the available options for the client, type:

```
client --help
```

Example 8.8. Karaf Client Help

```
Apache Felix Karaf client -a [port] specify the port to connect to -
h [host] specify the host to connect to -u [user] specify the user
name -p [password] specify the password --help shows this help
message -v raise verbosity -r [attempts] retry connection
establishment (up to attempts times) -d [delay] intra-retry delay
(defaults to 2 seconds) [commands] commands to run If no commands are
specified, the client will be put in an interactive mode
```

Remote client default credentials

You might be surprised to find that you can log into your Karaf container using **bin/client**, without supplying any credentials. This is because the remote client program is pre-configured to use default credentials. If no credentials are specified, the remote client automatically tries to use the following default credentials (in sequence):

- *Default SSH key*—tries to login using the default Apache Karaf SSH key. The corresponding configuration entry that would allow this login to succeed is commented out by default in the **etc/keys.properties** file.
- *Default username/password credentials*—tries to login using the **admin**/**admin** combination of username and password. The corresponding configuration entry that would allow this login to succeed is commented out by default in the **etc/users.properties** file.

Hence, if you create a new user in the Karaf container simply by uncommenting the default **admin**/**admin** credentials in **users.properties**, you will find that the **bin/client** utility can log in without supplying credentials.

Important

For your security, JBoss Fuse has disabled the default credentials (by commenting out) when the Karaf container is first installed. If you simply uncomment these default credentials, however, *without* changing the default password or SSH public key, you will open up a security hole in your Karaf container. You must *never* do this in a production environment. If you find that you can login to your container using **`bin/client`** without supplying credentials, *this shows that your container is insecure and you must take steps to fix this in a production environment*.

Disconnecting from a remote client console

If you used the remote client to open a remote console, as opposed to using it to pass a command, you will need to disconnect from it. To disconnect from the remote client's console, enter **`logout`** or press **`Ctrl`**+**`D`** at the prompt.

The client will disconnect and exit.

8.2.4. Connecting to a Container Using the SSH Command-Line Utility

Overview

You can also use the **`ssh`** command-line utility (a standard utility on UNIX-like operating systems) to log in to the Red Hat JBoss Fuse container, where the authentication mechanism is based on public key encryption (the public key must first be installed in the container). For example, given that the container is configured to listen on TCP port 8101, you could log in as follows:

```
ssh -p 8101 jdoe@localhost
```

Important

Key-based login is currently supported only on standalone containers, not on Fabric containers.

Prerequisites

To use key-based SSH login, the following prerequisites must be satisfied:

- The container must be standalone (Fabric is not supported) with the **`PublickeyLoginModule`** installed.
- You must have created an SSH key pair (see the section called “Creating a new SSH key pair”).
- You must install the public key from the SSH key pair into the container (see the section called “Installing the SSH public key in the container”).

Default key location

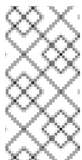
The **ssh** command automatically looks for the private key in the default key location. It is recommended that you install your key in the default location, because it saves you the trouble of specifying the location explicitly.

On a *NIX operating system, the default locations for an RSA key pair are:

```
~/.ssh/id_rsa
~/.ssh/id_rsa.pub
```

On a Windows operating system, the default locations for an RSA key pair are:

```
C:\Documents and Settings\Username\.ssh\id_rsa
C:\Documents and Settings\Username\.ssh\id_rsa.pub
```



Note

Red Hat JBoss Fuse supports only RSA keys. DSA keys do *not* work.

Creating a new SSH key pair

Generate an RSA key pair using the **ssh-keygen** utility. Open a new command prompt and enter the following command:

```
ssh-keygen -t rsa -b 2048
```

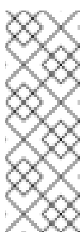
The preceding command generates an RSA key with a key length of 2048 bits. You will then be prompted to specify the file name for the key pair:

```
Generating public/private rsa key pair.
Enter file in which to save the key (/Users/Username/.ssh/id_rsa):
```

Type return to save the key pair in the default location. You will then be prompted for a pass phrase:

```
Enter passphrase (empty for no passphrase):
```

You can optionally enter a pass phrase here or type return twice to select no pass phrase.



Note

If you want to use the same key pair for running Fabric console commands, it is recommended that you select *no pass phrase*, because Fabric does not support using encrypted private keys.

Installing the SSH public key in the container

To use the SSH key pair for logging into the Red Hat JBoss Fuse container, you must install the SSH public key in the container by creating a new user entry in the ***InstallDir*/etc/keys.properties** file. Each user entry in this file appears on a single line, in the following format:

```
Username=PublicKey,Role1,Role2,...
```

For example, given that your public key file, **~/.ssh/id_rsa.pub**, has the following contents:

```
ssh-rsa
AAAAB3NzaC1kc3MAAACBAP1/U4EddRIpUt9KnC7s5Of2EbdSPO9EAMMeP4C2USZpRV1AIlH7W
T2NWPq/xfW6MPbLm1Vs14E7
gB00b/JmYLdrmVClpJ+f6AR7ECLCT7up1/63xhv4O1fnfqimFQ8E+4P208UewwI1VBNaFpEy9
nXzrith1yrv8iIDGZ3RSAHHAAAAFQCX
YFCPFSMLzLKSuYKi64QL8Fgc9QAAAnEA9+GghdabPd7LvKtcNrhXuXmUr7v6OuqC+VdMCz0Hg
mdRWVeOutRZT+ZxBxCBgLRJFnEj6Ewo
FhO3zwkyjMim4TwWeotifI0o4KOuHiuzpnWRbqN/C/ohNWLx+2J6ASQ7zKTxvqhRkImog9/hW
uWfBpKLZl6Ae1UlZAFMO/7PSSoAAACB
AKKSU2PFl/qOLxIwmBZPPIcJshVe7bVUpFvyl3BbJDow8rXfskl8wO63OzP/qLmcJM0+JbcRU
/53Jj7uyk31drV2qxhIOsLDC9dGCWj4
7Y7TyhPdXh/0dthTRBy6bqGtRPxGa7gJov1xm/UuYYXPIUR/3x9MAZvZ5xvE0kYXO+rx
jdoe@doemachine.local
```

You can create the **jdoe** user with the **admin** role by adding the following entry to the ***InstallDir*/etc/keys.properties** file (on a single line):

```
jdoe=AAAAB3NzaC1kc3MAAACBAP1/U4EddRIpUt9KnC7s5Of2EbdSPO9EAMMeP4C2USZpRV1A
IlH7WT2NWPq/xfW6MPbLm1Vs14E7
gB00b/JmYLdrmVClpJ+f6AR7ECLCT7up1/63xhv4O1fnfqimFQ8E+4P208UewwI1VBNaFpEy9
nXzrith1yrv8iIDGZ3RSAHHAAAAFQCX
YFCPFSMLzLKSuYKi64QL8Fgc9QAAAnEA9+GghdabPd7LvKtcNrhXuXmUr7v6OuqC+VdMCz0Hg
mdRWVeOutRZT+ZxBxCBgLRJFnEj6Ewo
FhO3zwkyjMim4TwWeotifI0o4KOuHiuzpnWRbqN/C/ohNWLx+2J6ASQ7zKTxvqhRkImog9/hW
uWfBpKLZl6Ae1UlZAFMO/7PSSoAAACB
AKKSU2PFl/qOLxIwmBZPPIcJshVe7bVUpFvyl3BbJDow8rXfskl8wO63OzP/qLmcJM0+JbcRU
/53Jj7uyk31drV2qxhIOsLDC9dGCWj4
7Y7TyhPdXh/0dthTRBy6bqGtRPxGa7gJov1xm/UuYYXPIUR/3x9MAZvZ5xvE0kYXO+rx,admi
n
```

Important

Do not insert the entire contents of the **id_rsa.pub** file here. Insert just the block of symbols which represents the public key itself.

Checking that public key authentication is supported

After starting the container, you can check whether public key authentication is supported by running the **jaas:realms** console command, as follows:

```
Index Realm                Module Class
    1 karaf
org.apache.karaf.jaas.modules.properties.PropertiesLoginModule
    2 karaf
org.apache.karaf.jaas.modules.publickey.PublickeyLoginModule
```

You should see that the **PublickeyLoginModule** is installed. With this configuration you can log in to the container using either username/password credentials or public key credentials.

Logging in using key-based SSH

You are now ready to login to the container using the key-based SSH utility. For example:

```
$ ssh -p 8101 jdoe@localhost
      _ ____                  ______
     | |  _ \                |  ____|
     | | |_) | ___  ___ ___  | |__ _   _ ___  ___
 _   | |  _   / _ \/ __/ __| |  __| | | / __|/ _ \
| |__| | |_) | (_) \__ \__ \ | |  | |_| \__ \  __/
 \____/|____/ \___/|___/___/ |_|   \__,_|___/\___|


  JBoss Fuse (6.3.0.redhat-187)
  http://www.redhat.com/products/jbossenterprisemiddleware/fuse/

Hit '<tab>' for a list of available commands
and '[cmd] --help' for help on a specific command.
Hit '<ctrl-d>' or 'osgi:shutdown' to shutdown JBoss Fuse.

JBossFuse:karaf@root>
```

Note

If you are using an encrypted private key, the **ssh** utility will prompt you to enter the pass phrase.

8.3. STOPPING A REMOTE CONTAINER

Using the remote console

If you have connected to a remote console using the **ssh:ssh** command, the **fabric:container-connect** command, or the remote client, you can stop the remote instance using the **osgi:shutdown** command.

Note

Pressing **Ctrl**+**D** in a remote console simply closes the remote connection and returns you to the local shell.

Using the fabric:container-stop console command

If your containers are deployed in a fabric, you can stop any container in the fabric using the **`fabric:container-stop`** command. For example, to shut down the container called **`child1`**, you would enter the following console command:

```
JBossFuse:karaf@root> fabric:container-stop child1
```

CHAPTER 9. MANAGING CHILD CONTAINERS

Abstract

A child container is a container that shares a common Red Hat JBoss Fuse runtime with a parent container, but has its own configuration files, runtime information, logs and temporary files. The child container functions as an independent container into which you can deploy bundles.

9.1. STANDALONE CHILD CONTAINERS

Using the admin console commands

The **admin** console commands allow you to create and manage instances of the JBoss Fuse runtime on the same machine. Each new runtime is a child instance of the runtime that created it. You can easily manage the children using names instead of network addresses. For details on the **admin** commands, see chapter "Admin Console Commands" in "Console Reference".

Installing the admin console commands

The **admin** commands are not installed by default. To install the command set, install the **admin** feature with the following command:

```
JBossFuse:karaf@root> features:install admin
```

Cloning a container

When you clone a container using the **admin:clone** command, you create a new child container which is an exact copy of the parent container in its current state. For example, if you clone the root container, the child gets the same configuration as the root container. Note that the child container has the *same port numbers* as the parent by default. After cloning, therefore, it is a good idea to customize the child's port numbers, to avoid clashes.

In the case of the SSH port, it is possible to customize the port when you create the child, by specifying the **-s** option. For example, to create a new child with the SSH port number of 8102:

```
JBossFuse:karaf@root> admin:clone -s 8102 root cloned
```

Creating a Karaf child container

The **admin:create** command creates a new Apache Karaf child container. That is, the new child container is not a full JBoss Fuse container, and is *missing many of the standard bundles, features, and feature repositories* that are normally available in a JBoss Fuse container. What you get is effectively a plain Apache Karaf container with JBoss Fuse branding. Additional feature repositories or features that you require will have to be added to the child manually.

As shown in Example 9.1, “Creating a Runtime Instance”, **`admin:create`** causes the container to create a new child container in the active container's **`instances/containerName`** directory. The child container is assigned an SSH port number based on an incremental count starting at 8101.

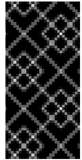
Example 9.1. Creating a Runtime Instance

```
JBossFuse:karaf@root> admin:create finn
Creating new instance on SSH port 8102 and RMI ports 1100/44445 at:
  /home/jdoe/apps/fuse/jboss-fuse-6.3.0.redhat-187/instances/finn
```

Changing a child's SSH port

You can change the SSH port number assigned to a child container using the **`admin:change-port`** command. The syntax for the command is:

`admin:change-port` { *containerName* } { *portNumber* }



Important

You can only use the **`admin:change-port`** command on stopped containers.

Starting child containers

New containers are created in the stopped state. To start a child container and make it ready to host applications, use the **`admin:start`** command. This command takes a single argument, *containerName*, that identifies the child you want started.

Listing all child containers

To see a list of all the JBoss Fuse containers running under a particular installation, use the **`admin:list`** command:

Example 9.2. Listing Instances

```
JBossFuse:karaf@root> admin:list
  Port    State       Pid   Name
[ 8107] [Started ] [10628] harry
[ 8101] [Started ] [20076] root
[ 8106] [Started ] [15924] dick
[ 8105] [Started ] [18224] tom
```

Connecting to a child container

You can connect to a started child container's remote console using the **`admin:connect`** command.As shown in Example 9.3, “Admin connect Command”, this command takes three arguments:

Example 9.3. Admin connect Command

`admin:connect` { *containerName* } { -u *username* } { -p *password* }

containerName

The name of the child to which you want to connect.

`-u` ***username***

The username used to connect to the child's remote console. Use valid JAAS user credentials that have admin privileges (see Chapter 14, *Configuring JAAS Security*).

`-p` ***password***

This argument specifies the password used to connect to the child's remote console.

Once you are connected to the child container., the prompt changes to display the name of the current instance, as shown:

```
JBossFuse:karaf@harry>
```

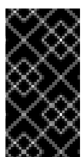
Stopping a child container

To stop a child container, from within the container itself, type **`osgi:shutdown`** or simply **`shutdown`**.

To stop a child container remotely—in other words, from a parent or sibling instance—type **`admin:stop`** ***`containerName`***.

Destroying a child container

You can permanently delete a stopped child container using the **`admin:destroy`** *containerName* command.



Important

You can only remove stopped children.

Changing the JVM options on a child container

To change the Java options in a child instance, use the **admin:change-opts** command. For example, you could change the amamount of memory allocated to the child container's JVM, as follows:

```
JBossFuse:karaf@harry> admin:change-opts tom "-server -Xms128M -Xmx1345m -
Dcom.sun.management.jmxremote"
```

These changes will take effect when you restart the child container.

Using the admin script

You can also use manage a JBoss Fuse container running in server mode without starting a new instance of the runtime. The **admin** script in the ***InstallDir*/bin** directory provides the all of the **admin** console commands except for **admin:connect**.

Example 9.4. The admin Script

```
admin.bat: Ignoring predefined value for KARAF_HOME
Available commands:
  change-port - Changes the port of an existing container instance.
  create - Creates a new container instance.
  destroy - Destroys an existing container instance.
  list - List all existing container instances.
  start - Starts an existing container instance.
  stop - Stops an existing container instance.
Type 'command --help' for more help on the specified command.
```

For example, to list all of the JBoss Fuse containers on your host machine, type:

Windows

```
admin.bat list
```

Linux or UNIX

```
./admin list
```

9.2. FABRIC CHILD CONTAINERS

Creating child containers

You create a new child container using the **fabric:container-create-child** console command, which has the following syntax:

```
karaf@root> fabric:container-create-child parent child [number]
```

Where *parent* is the name of an existing container in the fabric and *child* is the name of the new child container. If you create multiple child containers (by specifying the optional *number* argument), the new child instances are named ***child*1**, ***child*2**, and so on.

For example, assuming the container, root, already belongs to your fabric, you can create two new child containers as follows:

```
karaf@root> fabric:container-create-child root child 2
The following containers have been created successfully:
 child1
 child2
```

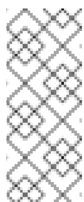
Listing all container instances

To list all of the containers in the current fabric (including child instances), use the **fabric:container-list** console command. For example:

```
JBossFuse:karaf@root> fabric:container-list
[id]                           [version] [alive] [profiles]
[provision status]
root                           1.0       true    fabric, fabric-
ensemble-0000-1
  child1                       1.0       true    default
success
  child2                       1.0       true    default
success
```

Assigning a profile to a child container

By default, a child is assigned the **default** profile when it is created. To assign a new profile (or profiles) to a child container after it has been created, use the **fabric:container-change-profile** console command.



Note

You can assign a profile other than **default** to a newly created container by using the **fabric:container-create-child** command's **--profile** argument.

For example, to assign the **example-camel** profile to the **child1** container, enter the following console command:

```
JBossFuse:karaf@root> fabric:container-change-profile child1 example-camel
```

The command removes the profiles currently assigned to **child1** and replaces them with the specified list of profiles (where in this case, there is just one profile in the list, **example-camel**).

Connecting to a child container

To connect to a child container, use the **fabric:container-connect** console command. For example, to connect to **child1**, enter the following console command:

```
JBossFuse:karaf@root>fabric:container-connect -u admin -p admin child1
```

You should see output like the following in your console window:

```
Connecting to host YourHost on port 8102
Connected
      _ ____                  ______
     | |  _ \                |  ____|
     | | |_) | ___  ___ ___  | |__ _   _ ___  ___
 _   | |  _ < / _ \/ __/ __| |  __| | | / __|/ _ \
| |__| | |_) | (_) \__ \__ \ | |  | |_| \__ \  __/
 \____/|____/ \___/|___/___/ |_|   \__,_|___/\___|

  JBoss Fuse (6.0.0.redhat-xxx)
  http://www.redhat.com/products/jbossenterprisemiddleware/fuse/

Hit '<tab>' for a list of available commands
and '[cmd] --help' for help on a specific command.
Hit '<ctrl-d>' or 'osgi:shutdown' to shutdown JBoss Fuse.

JBossFuse:admin@child1>
```

To terminate the session, enter **Ctrl-D**.

Starting a child container

To start a child container that was previously stopped, use the **fabric:container-start** command, providing the child container's name as the command argument—for example, to restart **child1**:

```
JBossFuse:karaf@root>fabric:container-start child1
```

This command starts up the child in a separate JVM.

Stopping a child container

To stop a child instance, use the **fabric:container-stop** command, providing the child container's name as the command argument—for example, to stop **child1**:

```
JBossFuse:karaf@root>fabric:container-stop child1
```

This command kills the JVM process that hosts the **child1** container.

Destroying a child container

To completely destroy a child container use the **fabric:container-delete** command. For example, to destroy the **child1** container instance, enter the following console command:

```
JBossFuse:karaf@root> fabric:container-delete child1
```

Destroying a child container does the following:

- stops the child's JVM process
- physically removes all files related to the child container

CHAPTER 10. DEPLOYING A NEW BROKER INSTANCE

Abstract

Red Hat JBoss Fuse supports the deployment of multiple JMS brokers in a container. Doing so involves creating a new set of broker configurations and deploying them to the container.

OVERVIEW

Deploying a new broker instance involves creating a new OSGi broker configuration and deploying it into the container. In a standalone container this can be done from the command console using the **`config`** command shell. For containers deployed in a fabric, you need to either create a profile for the new broker, or modify an existing profile to include the new broker configuration.

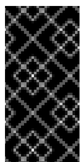
You will also likely want to create a new Apache ActiveMQ template configuration file that allows you to modify the desired settings. This will involve creating a new Apache ActiveMQ XML file and making it accessible to container.

STANDALONE CONTAINERS

To deploy a new broker into a standalone container:

1. Create a template Apache ActiveMQ XML configuration file in a location that is accessible to the container.

2. In the JBoss Fuse command console, use the **`config:edit`** command to create a new OSGi configuration file.

   Important

   The PID must start with **`io.fabric8.mq.fabric.server-`**.

3. Use the **`config:propset`** command to associate your template XML configuration with the broker OSGi configuration as shown in Example 10.1, “Specifying a Broker's Template XML Configuration”.

   Example 10.1. Specifying a Broker's Template XML Configuration

   ```
   JBossFuse:karaf@root> config:propset config configFile
   ```

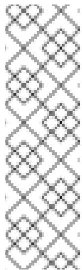
4. Use the **`config:propset`** command to set the required properties.

The properties that need to be set will depend on the properties you specified using property place holders in the template XML configuration and the broker's network settings.

For information on using **config:propset** see section "config:propset, propset" in "Console Reference".

5. Save the new OSGi configuration using the **config:update** command.

Once the new OSGi configuration is saved the new broker instance will start.



Note

If you want to simply deploy a second broker that uses the default configuration template skip Step 1. You will need set the config property to **${karaf.base}/etc/broker.xml**. You will also need to provide values for the data property, the broker-name property, and the openwire-port property.

EXAMPLE

If you wanted to deploy a new instance of the default broker called **myBroker** that stores its data in ***InstallDir*/data/myBroker** and opens a port at 61617, you would do the following:

1. Open the JBoss Fuse command console.

2. In the JBoss Fuse command console, use the **config:edit** command to create a new OSGi configuration file:

```
JBossFuse:karaf@root> config:edit io.fabric8.mq.fabric.server-
myBroker
```

3. Use the **config:propset** command to associate your template XML configuration with the broker OSGi configuration:

```
JBossFuse:karaf@root> config:propset config
${karaf.base}/etc/broker.xml
```

4. Use the **config:propset** command to specify the new broker's data directory:

```
JBossFuse:karaf@root> config:propset data ${karaf.data}/myBroker
```

5. Use the **config:propset** command to specify the new broker's name:

```
JBossFuse:karaf@root> config:propset broker-name myBroker
```

6. Use the **config:propset** command to specify the new broker's openwire port:

```
JBossFuse:karaf@root> config:propset openwire-port 61617
```

7. Save the new OSGi configuration using the **config:update** command.

CHAPTER 11. CONFIGURING JBOSS FUSE

Abstract

Red Hat JBoss Fuse uses the OSGi Configuration Admin service to manage the configuration of OSGi services. How you feed information to the configuration service depends on how the container is deployed.

11.1. INTRODUCING JBOSS FUSE CONFIGURATION

OSGi configuration

The OSGi Configuration Admin service specifies the configuration information for deployed services and ensures that the services receive that data when they are active.

A configuration is a list of name-value pairs read from a `.cfg` file in the ***InstallDir*/etc** directory. The file is interpreted using the Java properties file format. The filename is mapped to the persistent identifier (PID) of the service that is to be configured. In OSGi, a PID is used to identify a service across restarts of the container.

Configuration files

You can configure the Red Hat JBoss Fuse runtime using the following files:

Table 11.1. JBoss Fuse Configuration Files

Filename	Description
`broker.xml`	Configures the default Apache ActiveMQ broker in a Fabric (used in combination with the `io.fabric8.mq.fabric.server-default.cfg` file).
`config.properties`	The main configuration file for the container See Section 11.2, “Setting OSGi Framework and Initial Container Properties” for details.
`keys.properties`	Lists the users who can access the JBoss Fuse runtime using the SSH key-based protocol. The file's contents take the format *username=publicKey,role*
`org.apache.aries.transaction.cfg`	Configures the transaction feature
`org.apache.felix.fileinstall-deploy.cfg`	Configures a watched directory and polling interval for hot deployment.

Filename	Description
`org.apache.karaf.features.cfg`	Configures a list of feature repositories to be registered and a list of features to be installed when JBoss Fuse starts up for the first time.
`org.apache.karaf.features.obr.cfg`	Configures the default values for the features OSGi Bundle Resolver (OBR).
`org.apache.karaf.jaas.cfg`	Configures options for the Karaf JAAS login module. Mainly used for configuring encrypted passwords (disabled by default).
`org.apache.karaf.log.cfg`	Configures the output of the **`log`** console commands. See Section 16.2, “Logging Configuration”.
`org.apache.karaf.management.cfg`	Configures the JMX system. See Chapter 13, *Configuring JMX* for details.
`org.apache.karaf.shell.cfg`	Configures the properties of remote consoles. For more information see Section 8.1, “Configuring a Container for Remote Access”.
`io.fabric8.maven.cfg`	Configures the Maven repositories used by the Fabric Maven Proxy when downloading artifacts, (The Fabric Maven Proxy is used for provisioning new containers on a remote host.)
`io.fabric8.mq.fabric.server-default.cfg`	Configures the default Apache ActiveMQ broker in a Fabric (used in combination with the **`broker.xml`** file).
`org.ops4j.pax.logging.cfg`	Configures the logging system. For more, see Section 16.2, “Logging Configuration”.
`org.ops4j.pax.url.mvn.cfg`	Configures additional URL resolvers.
`org.ops4j.pax.web.cfg`	Configures the default Jetty container (Web server). See Securing the Web Console.
`startup.properties`	Specifies which bundles are started in the container and their start-levels. Entries take the format *bundle=start-level*.

Filename	Description
system.properties	Specifies Java system properties. Any properties set in this file are available at runtime using **System.getProperties()**. See Setting System and Config Properties for more.
users.properties	Lists the users who can access the JBoss Fuse runtime either remotely or via the web console. The file's contents take the format *username*=*password*,*role*
setenv or **setenv.bat**	This file is in the **/bin** directory. It is used to set JVM options. The file's contents take the format JAVA_MIN_MEM=*512M*, where *512M* is the minimum size of Java memory. See Setting Java Options for more information.

Configuration file naming convention

The file naming convention for configuration files depends on whether the configuration is intended for an OSGi Managed Service or for an OSGi Managed Service factory.

The configuration file for an OSGi Managed Service obeys the following naming convention:

```
<PID>.cfg
```

Where **<PID>** is the *persistent ID* of the OSGi Managed Service (as defined in the OSGi Configuration Admin specification). A persistent ID is normally dot-delimited—for example, **org.ops4j.pax.web**.

The configuration file for an OSGi Managed Service Factory obeys the following naming convention:

```
<PID>-<InstanceID>.cfg
```

Where **<PID>** is the *persistent ID* of the OSGi Managed Service Factory. In the case of a managed service factory's **<PID>,** you can append a hyphen followed by an arbitrary instance ID, **<InstanceID>**. The managed service factory then creates a unique service instance for each **<InstanceID>** that it finds.

Setting Java Options

Java Options can be set using the **/bin/setenv** file in Linux, or the **bin/setenv.bat** file for Windows. Use this file to directly set a group of Java options: JAVA_MIN_MEM, JAVA_MAX_MEM, JAVA_PERM_MEM, JAVA_MAX_PERM_MEM. Other Java options can be set using the EXTRA_JAVA_OPTS variable.

For example, to allocate minimum memory for the JVM use

```
JAVA_MIN_MEM=512M # Minimum memory for the JVM
```

To set a Java option other than the direct options, use

```
EXTRA_JAVA_OPTS="Java option"
```

For example,

```
EXTRA_JAVA_OPTS="-XX:+UseG1GC"
```

11.2. SETTING OSGI FRAMEWORK AND INITIAL CONTAINER PROPERTIES

Overview

There are a number of configuration properties that are set when a container is bootstrapped. These properties include the container's name, the default features repository used by the container, the OSGi framework provider, and other settings. These properties are specified in two property files in the **etc** folder:

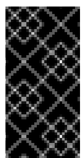
- **config.properties**—specifies the bootstrap properties for the OSGi framework
- **system.properties**—specifies properties to configure container functions

OSGi framework properties

The **etc/config.properties** file contains the properties used to specify which OSGi framework implementation to load and properties for configuring the framework's behaviors. Table 11.2, “Properties for the OSGi Framework” describes the key properties to set.

Table 11.2. Properties for the OSGi Framework

Property	Description
karaf.framework	Specifies the OSGi framework that Red Hat JBoss Fuse uses. The default framework is Apache Felix which is specified using the value **felix**.
karaf.framework.felix	Specifies the path to the Apache Felix JAR on the file system.



Important

JBoss Fuse only supports the Apache Felix OSGi implementation.

Initial container properties

The **etc/system.properties** file contains properties that configure how various aspects of the container behave including:

- the container's name
- the default feature repository used by the container
- the default port used by the OSGi HTTP service
- the initial message broker configuration

Table 11.3, “Container Properties” describes some of the common properties.

Table 11.3. Container Properties

Property	Description
karaf.name	Specifies the name of this container. The default is **root**.
karaf.default.repository	Specifies the location of the feature repository the container will use by default. The default setting is the local feature repository installed with JBoss Fuse.
org.osgi.service.http.port	Specifies the default port for the OSGi HTTP Service.

11.3. CONFIGURING STANDALONE CONTAINERS USING THE COMMAND CONSOLE

Overview

The command console's **config** shell provides commands for editing the configuration of a standalone container. The commands allow you to inspect the container's configuration, add new PIDs, and edit the properties of any PID used by the container. These configuration changes are applied directly to the container and will persist across container restarts.

For more details on the **config** commands see chapter "Config Console Commands" in "Console Reference".

Listing the current configuration

The **config:list** command will show all of the PIDs currently in use by the container. As shown in Example 11.1, "Output of the **config:list** Command", the output from **config:list** contains all of the PIDs and all of the properties for each of the PIDs.

Example 11.1. Output of the config:list Command

```
...
----------------------------------------------------------------
Pid:            org.ops4j.pax.logging
BundleLocation: mvn:org.ops4j.pax.logging/pax-logging-service/1.4
Properties:
   log4j.appender.out.layout.ConversionPattern = %d{ABSOLUTE} | %-
5.5p | %-16.16
t | %-32.32c{1} | %-32.32C %4L | %m%n
   felix.fileinstall.filename = org.ops4j.pax.logging.cfg
   service.pid = org.ops4j.pax.logging
   log4j.appender.stdout.layout.ConversionPattern = %d{ABSOLUTE} |
%-5.5p | %-16
.16t | %-32.32c{1} | %-32.32C %4L | %m%n
   log4j.appender.out.layout = org.apache.log4j.PatternLayout
   log4j.rootLogger = INFO, out, osgi:VmLogAppender
   log4j.appender.stdout.layout = org.apache.log4j.PatternLayout
   log4j.appender.out.file = /home/apache/jboss-fuse-6.3.0.redhat-
187/data/log/karaf.log
   log4j.appender.stdout = org.apache.log4j.ConsoleAppender
   log4j.appender.out.append = true
   log4j.appender.out = org.apache.log4j.FileAppender
----------------------------------------------------------------
Pid:            org.ops4j.pax.web
BundleLocation: mvn:org.ops4j.pax.web/pax-web-runtime/0.7.1
Properties:
   org.apache.karaf.features.configKey = org.ops4j.pax.web
   service.pid = org.ops4j.pax.web
   org.osgi.service.http.port = 8181
----------------------------------------------------------------
...
```

Listing the container's configuration is a good idea before editing a container's configuration. You can use the output to ensure that you know the exact PID to change.

Editing the configuration

Editing a container's configuration involves a number of commands and must be done in the proper sequence. Not following the proper sequence can lead to corrupt configurations or the loss of changes.

To edit a container's configuration:

1. Start an editing session by typing **config:edit *PID***.

PID is the PID for the configuration you are editing. It **must** be entered exactly. If it does not match the desired PID, the container will create a new PID with the specified name.

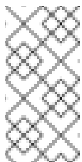
2. Remind yourself of the available properties in a particular configuration by typing **`config:proplist`**.

3. Use one of the editing commands to change the properties in the configuration.

   The editing commands include:

   - **`config:propappend`**—appends a new property to the configuration
   - **`config:propset`**—set the value for a configuration property
   - **`config:propdel`**—delete a property from the configuration

4. Update the configuration in memory and save it to disk by typing **`config:update`**.



Note

To exit the configuration, without saving your changes, type **`config:cancel`**.

Example 11.2, “Editing a Configuration” shows a configuration editing session that changes a container's logging behavior.

Example 11.2. Editing a Configuration

```
JBossFuse:karaf@root> config:edit org.apache.karaf.log
JBossFuse:karaf@root> config:proplist
 service.pid = org.apache.karaf.log size = 500
felix.fileinstall.filename = org.apache.karaf.log.cfg pattern =
%d{ABSOLUTE} | %-5.5p | %-16.16t | %-32.32c{1} | %-32.32C %4L | %m%n
JBossFuse:karaf@root> config:propset size 300
JBossFuse:karaf@root> config:update
```

11.4. CONFIGURING FABRIC CONTAINERS

Overview

When a container is part of a fabric, it does not manage its configuration. The container's configuration is managed by the Fabric Agent. The agent runs along with the container and updates the container's configuration based on information from the fabric's registry.

Because the configuration is managed by the Fabric Agent, any changes to the container's

configuration needs to be done by updating the fabric's registry. In a fabric, container configuration is determined by one or more profiles that are deployed into the container. To change a container's configuration, you must update the profile(s) deployed into the container using either the console's **`fabric:`** shell or the management console.

Profiles

All configuration in a fabric is stored as profiles in the Fabric Registry. One or more profiles are assigned to containers that are part of the fabric. A profile is a collection of configuration that specifies:

- the Apache Karaf features to be deployed
- OSGi bundles to be deployed
- the feature repositories to be scanned for features
- properties that configure the container's runtime behavior

The configuration profiles are collected into versions. Versions are typically used to make updates to an existing profile without effecting deployed containers. When a container is configured it is assigned a profile version from which it draws the profiles. Therefore, when you create a new version and edit the profiles in the new version, the profiles that are in use are not changed. When you are ready to test the changes, you can roll them out incrementally by moving containers to a new version one at a time.

When a container joins a fabric, a Fabric Agent is deployed with the container and takes control of the container's configuration. The agent will ask the Fabric Registry what version and profile(s) are assigned to the container and configure the container based on the profiles. The agent will download and install of the specified bundles and features. It will also set all of the specified configuration properties.

Best practices

Editing a profile makes changes to the copy in the Fabric Registry and all of the Fabric Agents are alerted when changes are made. If a running container is using a profile that is changed, its agent will automatically apply the new settings. If the update is benign having the change rolled out to the entire fabric is not an issue. If, on the other hand, the change causes issues, the entire fabric could become unstable.

To avoid having untested changes infecting an entire fabric, you should always make a new version before editing a profile. This isolates the changes in a version that is not running on any containers and provides a quick backup in case the changes are bad.

Once the profile changes have been made, you should test them out by upgrading only a few containers to the new version to see how they behave. As you become confident that the changes are good, you can then upgrade more containers.

Making changes using the command console

The command console's **`fabric`** shell has commands for managing profiles and versions in a fabric. These commands include:

- **`fabric:version-create`**—create a new version
- **`fabric:profile-create`**—create a new profile
- **`fabric:profile-edit`**—edit the properties in a profile
- **`fabric:container-change-profile`**—change the profiles assigned to a container

Example 11.3, "Editing Fabric Profile" shows a session for updating a profile using the command console.

Example 11.3. Editing Fabric Profile

```
JBossFuse:karaf@root> fabric:version-create
Created version: 1.1 as copy of: 1.0
JBossFuse:karaf@root> fabric:profile-edit -p
org.apache.karaf.log/size=300 NEBroker
```

The change made in Example 11.3, "Editing Fabric Profile" is not applied to any running containers because it is made in a new version. In order to apply the change you need to update one or more containers using the **`fabric:container-upgrade`** command.

See chapter "Fabric Console Commands" in "Console Reference" for more information.

Using the management console

The management console simplifies the process of configuring containers in a fabric by providing an easy to use Web-based interface and reducing the number of steps required to make the changes. For more information on using the management console see *Using the Management Console*.

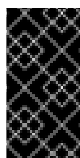
CHAPTER 12. CONFIGURING THE HOT DEPLOYMENT SYSTEM

Abstract

Standalone containers scan a directory for OSGi bundles artifacts to load automatically. You can change the location of this folder and the interval at which the folder is scanned.

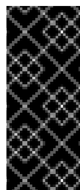
OVERVIEW

Standalone containers will automatically load and deploy OSGi bundles artifacts from a pre-configured folder. It scans the folder once a second for new bundles. You can change the folder a container scans and the scan interval by editing properties in the **`org.apache.felix.fileinstall-deploy`** PID.



Important

The hot deployment system is *not* not enabled for fabric containers.



Important

The hot deployment system works only while the Karaf container is running. In particular, deleting files from the hot deploy directory is not effective while the container is shut down.

SPECIFYING THE HOT DEPLOYMENT FOLDER

By default, a container scans the **`deploy`** folder that is relative to the folder from which you launched the container. You change the folder the container monitors by setting the felix.fileinstall.dir property in the **`rg.apache.felix.fileinstall-deploy`** PID. The value is the absolute path of the folder to monitor. If you set the value to **`/home/joe/deploy`**, the container will monitor a folder in Joe's home directory.

SPECIFYING THE SCAN INTERVAL

By default containers scan the hot deployment folder every 1000 milliseconds. To change the interval between scans of the hot deployment folders, you change the felix.fileinstall.poll property in the **`org.apache.felix.fileinstall-deploy`** PID. The value is specified in milliseconds.

EXAMPLE

Example 12.1, “Configuring the Hot Deployment Folders” shows a configuration editing session that sets **`/home/karaf/hot`** as the hot deployment folder and sets the scan interval to half a second.

Example 12.1. Configuring the Hot Deployment Folders

```
JBossFuse:karaf@root> config:edit org.apache.felix.fileinstall-
deploy
JBossFuse:karaf@root> config:propset felix.fileinstall.dir
/home/karaf/hot
JBossFuse:karaf@root> config:propset felix.fileinstall.poll 500
JBossFuse:karaf@root> config:update
```

CHAPTER 13. CONFIGURING JMX

Abstract

Red Hat JBoss Fuse uses JMX for its underlying management features. You can configure the JMX RMI port, the JMX URL, and the credentials used to access the JMX features.

OVERVIEW

Red Hat JBoss Fuse uses JMX for reporting runtime metrics and providing some limited management capabilities. You can configure how the JMX management features are accessed by changing the properties in the **`org.apache.karaf.management`** PID.

CHANGING THE RMI PORT AND JMX URL

Two of the most commonly changed parts of a container's JMX configuration are the RMI port and the JMX URL. You can set these using the properties described in Table 13.1, "JMX Access Properties".

Table 13.1. JMX Access Properties

Property	Description
rmiRegistryPort	Specifies the RMI registry port. The default value is 1099.
serviceUrl	Specifies the the URL used to connect to the JMX server. The default URL is service:jmx:rmi:///jndi/rmi://localhost:1099/karaf-*KarafName*, where ***`KarafName`*** is the container's name (by default, **`root`**).

SETTING THE JMX USERNAME AND PASSWORD

In a standalone container, use any valid JAAS user credentials (see the section called "Create a secure JAAS user").

In a fabric, the default username is **`admin`** and the default password is **`admin`**.

You can change the username and password used to connect to the JMX server by configuring the JAAS security system as described in Chapter 14, *Configuring JAAS Security*.

TROUBLESHOOTING ON LINUX PLATFORMS

On Linux platforms, if you have trouble getting a remote JConsole instance to connect to the JMX server, check the following points:

- Check that the hostname resolves to the correct IP address. For example, if the **`hostname -i`** command returns 127.0.0.1, JConsole will not be able to connect to the JMX server. To fix this, edit the **`/etc/hosts`** file so that the hostname resolves to the correct IP address.

- Check whether the Linux machine is configured to accept packets from the host where JConsole is running (packet filtering is built in the Linux kernel). You can enter the command, **`/sbin/iptables --list`**, to determine whether an external client is allowed to connect to the JMX server.

  Use the following command to add a rule to allow an external client such as JConsole to connect:

  ```
  /usr/sbin/iptables -I INPUT -s JconsoleHost -p tcp --destination-port
  JMXRemotePort -j ACCEPT
  ```

  Where *JconsoleHost* is either the hostname or the IP address of the host on which JConsole is running and *JMXRemotePort* is the TCP port exposed by the JMX server.

CHAPTER 14. CONFIGURING JAAS SECURITY

14.1. ALTERNATIVE JAAS REALMS

Overview

The Java Authentication and Authorization Service (JAAS) is a pluggable authentication service, which is implemented by a *login module*. A particular instance of a JAAS service is known as a JAAS realm and is identified by a *realm name*.

Applications integrated with JAAS must be configured to use a specific realm, by specifying the realm name.

Default realm

The default realm in Red Hat JBoss Fuse is identified by the **`karaf`** realm name. The standard administration services in JBoss Fuse (SSH remote console, JMX port, and so on) are all configured to use the **`karaf`** realm by default.

Available realm implementations

JBoss Fuse provides the following alternative JAAS realm implementations:

- the section called “Standalone JAAS realm”.
- the section called “Fabric JAAS realm”.
- the section called “LDAP JAAS realm”.

Standalone JAAS realm

In a standalone container, the **`karaf`** realm installs four JAAS login modules, which are used in parallel:

`PropertiesLoginModule`

> Authenticates username/password credentials and stores the secure user data in the ***`InstallDir`*`/etc/users.properties`** file.

`PublickeyLoginModule`

> Authenticates SSH key-based credentials (consisting of a username and a public/private key pair). Secure user data is stored in the ***`InstallDir`*`/etc/keys.properties`** file.

`FileAuditLoginModule`

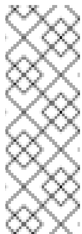
Provides an audit trail of successful/failed login attempts, which are logged to an audit file. Does *not* perform user authentication.

`EventAdminAuditLoginModule`

Provides an audit trail of successful/failed login attempts, which are logged to the OSGi Event Admin service. Does *not* perform user authentication.

Fabric JAAS realm

In a fabric, a **`karaf`** realm based on the **`ZookeeperLoginModule`** login module is automatically installed in every container (the **`fabric-jaas`** feature is included in the default profile) and is responsible for securing the SSH remote console and other administrative services. The Zookeeper login module stores the secure user data in the Fabric Registry.



Note

In containers where the standalone JAAS realm and the Fabric JAAS realm are both installed, the Fabric JAAS realm takes precedence, because it defines a **`karaf`** realm with a *higher rank*.

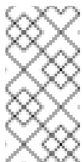
LDAP JAAS realm

It is also possible to configure a container to use an LDAP login module with JAAS. For details of how to set this up, see LDAP Authentication Tutorial.

14.2. JAAS CONSOLE COMMANDS

Editing user data from the console

Red Hat JBoss Fuse provides a set of **`jaas:*`** console commands, which you can use to edit JAAS user data from the console. This works both for standalone JAAS realms and for Fabric JAAS realms.



Note

The **`jaas:*`** console commands are not compatible with the LDAP JAAS module.

Standalone realm configuration

A standalone container (which uses the JAAS **`PropertiesLoginModule`** and the **`PublickeyLoginModule`**) maintains its own database of secure user data, independently of any other containers. To configure the user data for a standalone container, you must log into the specific container (see Connecting and Disconnecting Remotely) whose data you want to modify. Each standalone container must be configured separately.

To start editing the standalone JAAS user data, you must first specify the JAAS realm that you want to modify. To see the available realms, enter the **jaas:realms** command, as follows:

```
JBossFuse:karaf@root> jaas:realms
Index Realm                Module Class
    1 karaf
org.apache.karaf.jaas.modules.properties.PropertiesLoginModule
    2 karaf
org.apache.karaf.jaas.modules.publickey.PublickeyLoginModule
    3 karaf
org.apache.karaf.jaas.modules.audit.FileAuditLoginModule
    4 karaf
org.apache.karaf.jaas.modules.audit.EventAdminAuditLoginModule
```

All of these login modules are active in the default **karaf** JAAS realm. Enter the following console command to start editing the properties login module in the **karaf** realm:

```
JBossFuse:karaf@root> jaas:manage --index 1
```

Fabric realm configuration

A container in a fabric (which uses the JAAS **ZookeeperLoginModule** by default) shares its secure user data with all of the other containers in the fabric and the user data is stored in the Fabric Registry. To configure the user data for a fabric, you can log into any of the containers. Because the user data is shared in the registry, any modifications you make are instantly propagated to all of the containers in the fabric.

To start editing the fabric JAAS user data, you must first specify the JAAS login module you want to modify. In the context of fabric, you must modify the Zookeeper login module. For example, if you enter the **jaas:realms** console command, you might see a listing similar to this:

```
Index Realm                Module Class
    1 karaf                io.fabric8.jaas.ZookeeperLoginModule
    2 karaf
org.apache.karaf.jaas.modules.properties.PropertiesLoginModule
    3 karaf
org.apache.karaf.jaas.modules.publickey.PublickeyLoginModule
```

The **ZookeeperLoginModule** login module has the highest priority and is used by the fabric (you cannot see this from the listing, but its realm is defined to have a higher rank than the other modules). In this example, the **ZookeeperLoginModule** has the index **1**, but it might have a different index number in your container.

Enter the following console command to start editing the fabric's JAAS realm (specifying the index of the **ZookeeperLoginModule**):

```
JBossFuse:karaf@root> jaas:manage --index 1
```

Adding a new user to the JAAS realm

For example, consider how to add a new user, **jdoe**, to the JAAS realm.

First of all, start to manage the relevant JAAS realm as follows:

1. List the available realms and login modules by entering the following command:

```
JBossFuse:karaf@root> jaas:realms
```

2. Choose the login module to edit by specifying its index, *Index*, using a command of the following form:

```
JBossFuse:karaf@root> jaas:manage --index Index
```

Add the user, **jdoe**, with password, **secret**, by entering the following console command:

```
JBossFuse:karaf@root> jaas:useradd jdoe secret
```

Add the **admin** role to **jdoe**, by entering the following console command:

```
JBossFuse:karaf@root> jaas:roleadd jdoe admin
```

As a matter of fact, these changes are *not* applied right away. Initially, the changes are queued in a list of pending operations. To see this list, enter the **jaas:pending** console command, as follows:

```
JBossFuse:karaf@root> jaas:pending
Jaas Realm:karaf Jaas
Module:org.apache.karaf.jaas.modules.properties.PropertiesLoginModule
UserAddCommand{username='jdoe', password='secret'}
RoleAddCommand{username='jdoe', role='admin'}
```

Now you can apply the changes by invoking **jaas:update**, as follows:

```
JBossFuse:karaf@root> jaas:update
```

The new user entry is then persisted (either by writing to the remote container's **etc/users.properties** file, in the case of a standalone container, or by storing the user data in the Fabric Registry, in the case of a fabric).

Canceling pending changes

If you decide that you do *not* want to make the changes permanent after all, instead of invoking the **jaas:update** command, you could abort the pending changes using the **jaas:cancel** command, as follows:

```
JBossFuse:karaf@root> jaas:cancel
```

14.3. STANDALONE REALM PROPERTIES FILE

Overview

The default JAAS realm used by a standalone container is implemented by the **PropertiesLoginModule** JAAS module. This login module stores its user data in a Java properties file in the following location:

```
InstallDir/etc/users.properties
```

Format of users.properties entries

Each entry in the **etc/users.properties** file has the following format (on its own line):

```
Username=Password[,UserGroup|Role][,UserGroup|Role]...
```

Changing the default username and password

The **etc/users.properties** file initially contains a commented out entry for a single user, admin, with password admin and role admin. It is strongly recommended that you create a new user entry that is *different* from the **admin** user example.

For example, you could create a new user in the following format:

```
Username=Password,Administrator
```

Where the **Administrator** role grants full administration privileges to this user.

CHAPTER 15. SECURING FABRIC CONTAINERS

Abstract

By default, fabric containers uses text-based username/password authentication. Setting up a more robust access control system involves creating and deploying a new JAAS realm to the containers in the fabric.

DEFAULT AUTHENTICATION SYSTEM

By default, Fabric uses a simple text-based authentication system (implemented by the JAAS login module, **`io.fabric8.jaas.ZookeeperLoginModule`**). This system allows you to define user accounts and assign passwords and roles to the users. Out of the box, the user credentials are stored in the Fabric registry, unencrypted.

MANAGING USERS

You can manage users in the default authentication system using the **`jaas:*`** family of console commands. First of all you need to attach the **`jaas:*`** commands to the **`ZookeeperLoginModule`** login module, as follows:

```
JBossFuse:karaf@root> jaas:realms
Index Realm                Module Class
    1 karaf
org.apache.karaf.jaas.modules.properties.PropertiesLoginModule
    2 karaf
org.apache.karaf.jaas.modules.publickey.PublickeyLoginModule
    3 karaf                io.fabric8.jaas.ZookeeperLoginModule
JBossFuse:karaf@root> jaas:manage --index 3
```

Which attaches the **`jaas:*`** commands to the **`ZookeeperLoginModule`** login module. You can then add users and roles, using the **`jaas:useradd`** and **`jaas:roleadd`** commands. Finally, when you are finished editing the user data, you must commit the changes by entering the **`jaas:update`** command, as follows:

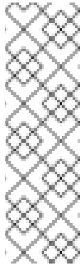
```
JBossFuse:karaf@root> jaas:update
```

Alternatively, you can abort the pending changes by entering **`jaas:cancel`**.

OBFUSCATING STORED PASSWORDS

By default, the JAAS **`ZookeeperLoginModule`** stores passwords in plain text. You can provide additional protection to passwords by storing them in an obfuscated format. This can be done by adding the appropriate configuration properties to the **`io.fabric8.jaas`** PID and ensuring that they are applied to *all* of the containers in the fabric.

For more details, see section "Using Encrypted Property Placeholders" in "Security Guide".



Note

Although message digest algorithms are not easy to crack, they are not invulnerable to attack (for example, see the Wikipedia article on cryptographic hash functions). Always use file permissions to protect files containing passwords, in addition to using password encryption.

ENABLING LDAP AUTHENTICATION

Fabric supports LDAP authentication (implemented by the Apache Karaf **`LDAPLoginModule`**), which you can enable by adding the requisite configuration to the default profile.

For details of how to enable LDAP authentication in a fabric, see chapter "LDAP Authentication Tutorial" in "Security Guide".

CHAPTER 16. LOGGING

Abstract

The Red Hat JBoss Fuse runtime uses OPS4j Pax Logging as its logging mechanism. It is easily configured using the standard OSGi Admin mechanism and can be easily integrated with applications deployed in a container. The command console provides commands to manage the logs.

16.1. LOGGING OVERVIEW

Red Hat JBoss Fuse uses the *OPS4j Pax Logging* system. Pax Logging is an open source OSGi logging service that extends the standard OSGi logging service to make it more appropriate for use in enterprise applications. It uses Apache Log4j as the back-end logging service. Pax Logging has its own API, but it also supports the following APIs:

- Apache Log4j
- Apache Commons Logging
- SLF4J
- Java Util Logging

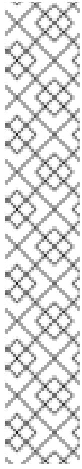
For more information on OPS4j Pax Logging see http://team.ops4j.org/wiki/display/paxlogging/Pax+Logging.

16.2. LOGGING CONFIGURATION

Overview

The logging system is configured by a combination of two OSGi Admin PIDs and one configuration file:

- **`etc/system.properties`**—the configuration file that sets the logging level during the container's boot process. The file contains a single property, org.ops4j.pax.logging.DefaultServiceLog.level, that is set to **`ERROR`** by default.
- **`org.ops4j.pax.logging`**—the PID used to configure the logging back end service. It sets the logging levels for all of the defined loggers and defines the appenders used to generate log output. It uses standard Log4j configuration. By default, it sets the root logger's level to **`INFO`** and defines two appenders: one for the console and one for the log file.



Note

The console's appender is disabled by default. To enable it, add **`log4j.appender.stdout.append=true`** to the configuration For example, to enable the console appender in a standalone container, you would use the following commands:

```
JBossFuse:karaf@root> config:edit org.ops4j.pax.logging
JBossFuse:karaf@root> config:propappend
log4j.appender.stdout.append true
JBossFuse:karaf@root> config:update
```

- **`org.apache.karaf.log.cfg`**—configures the output of the **`log`** console commands.

The most common configuration changes you will make are changing the logging levels, changing the threshold for which an appender writes out log messages, and activating per bundle logging.

Changing the log levels

The default logging configuration sets the logging levels so that the log file will provide enough information to monitor the behavior of the runtime and provide clues about what caused a problem. However, the default configuration will not provide enough information to debug most problems.

The most useful logger to change when trying to debug an issue with Red Hat JBoss Fuse is the root logger. You will want to set its logging level to generate more fine grained messages. To do so you change the value of the **`org.ops4j.pax.logging`** PID's **`log4j.rootLogger`** property so that the logging level is one of the following:

- **`TRACE`**
- **`DEBUG`**
- **`INFO`**
- **`WARN`**
- **`ERROR`**
- **`FATAL`**
- **`NONE`**

Example 16.1, “Changing Logging Levels” shows the commands for setting the root loggers log level in a standalone container.

Example 16.1. Changing Logging Levels

```
JBossFuse:karaf@root> config:edit org.ops4j.pax.logging
JBossFuse:karaf@root> config:propset log4j.rootLogger "DEBUG, out, osgi:VmLogAppender"
JBossFuse:karaf@root> config:update
```

Changing the appenders' thresholds

When debugging a problem in JBoss Fuse you may want to limit the amount of logging information that is displayed on the console, but not the amount written to the log file. This is controlled by setting the thresholds for each of the appenders to a different level. Each appender can have a **log4j.appender.*appenderName*.threshold** property that controls what level of messages are written to the appender. The appender threshold values are the same as the log level values.

Example 16.2, "Changing the Log Information Displayed on the Console" shows an example of setting the root logger to **DEBUG** but limiting the information displayed on the console to **WARN**.

Example 16.2. Changing the Log Information Displayed on the Console

```
JBossFuse:karaf@root> config:edit org.ops4j.pax.logging
JBossFuse:karaf@root> config:propset log4j.rootLogger "DEBUG, out, osgi:VmLogAppender"
JBossFuse:karaf@root> config:propappend log4j.appender.stdout.threshold WARN
JBossFuse:karaf@root> config:update
```

Logging per bundle

It is possible to reconfigure JBoss Fuse logging so that it writes one log file for each bundle, instead of writing all of the log messages into a single log file. This feature is enabled by adding the Log4j **sift** appender to the Log4j root logger as shown in Example 16.3, "Enabling Per Bundle Logging".

Example 16.3. Enabling Per Bundle Logging

```
JBossFuse:karaf@root> config:edit org.ops4j.pax.logging
JBossFuse:karaf@root> config:propset log4j.rootLogger "INFO, out, sift, osgi:VmLogAppender"
JBossFuse:karaf@root> config:update
```

After restarting the container, you can see that each *BundleName* bundle now has its own log file, located at **data/log/*BundleName*.log**.

This is the behavior you will see with the default sift appender settings. You can edit this behavior using the sift appender configuration settings in **`org.ops4j.pax.logging.cfg`**.

Logging History

When JBoss Fuse is stopped, the Karaf log is saved in **`~/.karaf/karaf.history`**. JBoss Fuse can be configured to prevent the history from being saved each time.

Procedure 16.1. Stop JBoss Fuse saving Logging History on shutdown.

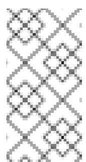
1. Stop JBoss Fuse.
2. Edit **`$FUSE_HOME/etc/system.properties`**.
3. Uncomment the line that says **`# karaf.shell.history.maxSize = 0`** .
4. Restart JBoss Fuse

`~/.karaf/karaf.history` will still be created, but it will always have a size of 0, and will be empty.

16.3. LOGGING PER APPLICATION

Overview

Using Mapped Diagnostic Context (MDC) logging, you create a separate log file for each of your applications. The basic idea of MDC logging is that you associate each logging message with a particular context (for example, by associating it with a set of key-value pairs). Later on, when it comes to writing the log stream, you can use the context data to sort or filter the logging messages in various ways.



Note

MDC logging is supported only by log4j and slf4j.

Application key

To use MDC logging, you must define a unique MDC key for each of your applications. The MDC key is a string that is associated with one application or logging context. At runtime, you can then use the application key to sort logging messages and write them into separate files for each application key.

Enabling per application logging

To enable per application logging:

1. In each of your applications, edit the Java source code to define a unique application key.

   If you are using slf4j, add the following static method call to your application:

   ```
   org.slf4j.MDC.put("app.name","MyFooApp");
   ```

   If you are using log4j, add the following static method call to your application:

   ```
   org.apache.log4j.MDC.put("app.name","MyFooApp");
   ```

2. Edit the **etc/org.ops4j.pax.logging** PID to customize the sift appender.

   a. Set **log4j.appender.sift.key** to **app.name**.

   b. Set **log4j.appender.sift.appender.file** to **=${karaf.data}/log/$\\{app.name\\}.log**.

3. Edit the **etc/org.ops4j.pax.logging** PID to add the sift appender to the root logger.

   ```
   JBossFuse:karaf@root> config:edit org.ops4j.pax.logging
   JBossFuse:karaf@root> config:propset log4j.rootLogger "INFO, out,
   sift, osgi:VmLogAppender"
   JBossFuse:karaf@root> config:update
   ```

16.4. LOG COMMANDS

The Red Hat JBoss Fuse console provides the following commands for managing logging output:

log:display

Displays the most recent log entries. By default, the number of entries returned and the pattern of the output depends on the size and pattern properties in the **org.apache.karaf.log.cfg** file. You can override these using the **-p** and **-d** arguments.

log:display-exception

Displays the most recently logged exception.

log:get

Displays the current log level.

log:set

Sets the log level.

`log:tail`

Continuously display log entries .

`log:clear`

Clear log entries.

CHAPTER 17. PERSISTENCE

Abstract

The Red Hat JBoss Fuse container caches information about its state and the artifacts deployed to it. It uses this data to make startup faster. You can configure how this information is stored on your file system.

OVERVIEW

Red Hat JBoss Fuse containers store all of their persistent caches relative to its start location. It will create a **`data`** folder in the directory from which you launch the container. This folder is populated by folders storing information about the message broker used by the container, the OSGi framework, and the Karaf container.

THE DATA FOLDER

The **`data`** folder is used by the JBoss Fuse runtime to store persistent state information. It contains the following folders:

`amq`

Contains persistent data needed by any Apache ActiveMQ brokers that are started by the container.

`cache`

The OSGi bundle cache. The cache contains a directory for each bundle, where the directory name corresponds to the bundle identifier number.

`generated-bundles`

Contains bundles that are generated by the container.

`log`

Contains the log files.

`maven`

A temporary directory used by the Fabric Maven Proxy when uploading files.

`txlog`

Contains the log files used by the transaction management system. You can set the location of this directory in the **`org.apache.aries.transaction.cfg`** file

CHANGING THE BUNDLE CACHE LOCATION

By default, the bundle cache is stored in ***InstallDir*/data/cache**.

To specify an alternative location, modify the org.osgi.framework.storage property in **config.properties**.

If you use a relative path, the cache location is added to the root of the JBoss Fuse installation directory.

FLUSHING THE BUNDLE CACHE

You can configure JBoss Fuse to flush the bundle cache every time the runtime starts by setting the org.osgi.framework.storage.clean property to onFirstInit in **config.properties**. This property is set to none by default.

CHANGING THE GENERATED-BUNDLE CACHE LOCATION

The generated-bundle cache is where the container caches bundles it creates to support JARs that are not supplied as OSGi bundles.

You can configure the location of this cache by changing the felix.fileinstall.tmpdir property in the **org.apache.felix.fileinstall-deploy.cfg** file.

ADJUSTING THE BUNDLE CACHE BUFFER

The felix.cache.bufsize property controls the size of the buffer used to copy bundles into the bundle cache. Its default value is 4096 bytes.

You can adjust this property by editing its value in the **config.properties** configuration file. The value is specified in bytes.

CHAPTER 18. FAILOVER DEPLOYMENTS

Abstract

Red Hat JBoss Fuse provides failover capability using either a simple lock file system or a JDBC locking mechanism. In both cases, a container-level lock system allows bundles to be preloaded into the slave kernel instance in order to provide faster failover performance.

18.1. USING A SIMPLE LOCK FILE SYSTEM

Overview

When you first start Red Hat JBoss Fuse a lock file is created at the root of the installation directory. You can set up a master/slave system whereby if the master instance fails, the lock is passed to a slave instance that resides on the same host machine.

Configuring a lock file system

To configure a lock file failover deployment, edit the **`etc/system.properties`** file on both the master and the slave installation to include the properties in Example 18.1, “Lock File Failover Configuration”.

Example 18.1. Lock File Failover Configuration

```
karaf.lock=true
karaf.lock.class=org.apache.karaf.main.SimpleFileLock
karaf.lock.dir=PathToLockFileDirectory
karaf.lock.delay=10000
```

- karaf.lock—specifies whether the lock file is written.
- karaf.lock.class—specifies the Java class implementing the lock. For a simple file lock it should always be **`org.apache.karaf.main.SimpleFileLock`**.
- karaf.lock.dir—specifies the directory into which the lock file is written. This **must** be the same for both the master and the slave installation.
- karaf.lock.delay—specifies, in milliseconds, the delay between attempts to reaquire the lock.

18.2. USING A JDBC LOCK SYSTEM

Overview

The JDBC locking mechanism is intended for failover deployments where Red Hat JBoss Fuse instances exist on separate machines.

In this scenario, the master instance holds a lock on a locking table hosted on a database. If the master loses the lock, a waiting slave process gains access to the locking table and fully starts its container.

Adding the JDBC driver to the classpath

In a JDBC locking system, the JDBC driver needs to be on the classpath for each instance in the master/slave setup. Add the JDBC driver to the classpath as follows:

1. Copy the JDBC driver JAR file to the ***ESBInstallDir*/lib** directory for each Red Hat JBoss Fuse instance.

2. Modify the **bin/karaf** start script so that it includes the JDBC driver JAR in its **CLASSPATH** variable.

   For example, given the JDBC JAR file, ***JDBCJarFile*.jar**, you could modify the start script as follows (on a *NIX operating system):

```
    ...
    # Add the jars in the lib dir
    for file in "$KARAF_HOME"/lib/karaf*.jar
    do
        if [ -z "$CLASSPATH" ]; then
            CLASSPATH="$file"
        else
            CLASSPATH="$CLASSPATH:$file"
        fi
    done
CLASSPATH="$CLASSPATH:$KARAF_HOME/lib/JDBCJarFile.jar"
```

Note

If you are adding a MySQL driver JAR or a PostgreSQL driver JAR, you must rename the driver JAR by prefixing it with the **karaf-** prefix. Otherwise, Apache Karaf will hang and the log will tell you that Apache Karaf was unable to find the driver.

Configuring a JDBC lock system

To configure a JDBC lock system, update the **etc/system.properties** file for each instance in the master/slave deployment as shown

Example 18.2. JDBC Lock File Configuration

```
karaf.lock=true
karaf.lock.class=org.apache.karaf.main.DefaultJDBCLock
karaf.lock.level=50
karaf.lock.delay=10000
karaf.lock.jdbc.url=jdbc:derby://dbserver:1527/sample
karaf.lock.jdbc.driver=org.apache.derby.jdbc.ClientDriver
karaf.lock.jdbc.user=user
karaf.lock.jdbc.password=password
karaf.lock.jdbc.table=KARAF_LOCK
karaf.lock.jdbc.clustername=karaf
karaf.lock.jdbc.timeout=30
```

In the example, a database named sample will be created if it does not already exist. The first Red Hat JBoss Fuse instance to acquire the locking table is the master instance. If the connection to the database is lost, the master instance tries to gracefully shutdown, allowing a slave instance to become master when the database service is restored. The former master will require manual restart.

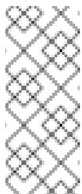
Configuring JDBC locking on Oracle

If you are using Oracle as your database in a JDBC locking scenario, the karaf.lock.class property in the **`etc/system.properties`** file must point to org.apache.karaf.main.OracleJDBCLock.

Otherwise, configure the **`system.properties`** file as normal for your setup, as shown:

Example 18.3. JDBC Lock File Configuration for Oracle

```
karaf.lock=true
karaf.lock.class=org.apache.karaf.main.OracleJDBCLock
karaf.lock.jdbc.url=jdbc:oracle:thin:@hostname:1521:XE
karaf.lock.jdbc.driver=oracle.jdbc.OracleDriver
karaf.lock.jdbc.user=user
karaf.lock.jdbc.password=password
karaf.lock.jdbc.table=KARAF_LOCK
karaf.lock.jdbc.clustername=karaf
karaf.lock.jdbc.timeout=30
```



Note

The karaf.lock.jdbc.url requires an active Oracle system ID (SID). This means you must manually create a database instance before using this particular lock.

Configuring JDBC locking on Derby

If you are using Derby as your database in a JDBC locking scenario, the karaf.lock.class property in the **etc/system.properties** file should point to org.apache.karaf.main.DerbyJDBCLock. For example, you could configure the **system.properties** file as shown:

Example 18.4. JDBC Lock File Configuration for Derby

```
karaf.lock=true
karaf.lock.class=org.apache.karaf.main.DerbyJDBCLock
karaf.lock.jdbc.url=jdbc:derby://127.0.0.1:1527/dbname
karaf.lock.jdbc.driver=org.apache.derby.jdbc.ClientDriver
karaf.lock.jdbc.user=user
karaf.lock.jdbc.password=password
karaf.lock.jdbc.table=KARAF_LOCK
karaf.lock.jdbc.clustername=karaf
karaf.lock.jdbc.timeout=30
```

Configuring JDBC locking on MySQL

If you are using MySQL as your database in a JDBC locking scenario, the karaf.lock.class property in the **etc/system.properties** file must point to org.apache.karaf.main.MySQLJDBCLock. For example, you could configure the **system.properties** file as shown:

Example 18.5. JDBC Lock File Configuration for MySQL

```
karaf.lock=true
karaf.lock.class=org.apache.karaf.main.MySQLJDBCLock
karaf.lock.jdbc.url=jdbc:mysql://127.0.0.1:3306/dbname
karaf.lock.jdbc.driver=com.mysql.jdbc.Driver
karaf.lock.jdbc.user=user
karaf.lock.jdbc.password=password
karaf.lock.jdbc.table=KARAF_LOCK
karaf.lock.jdbc.clustername=karaf
karaf.lock.jdbc.timeout=30
```

Configuring JDBC locking on PostgreSQL

If you are using PostgreSQL as your database in a JDBC locking scenario, the karaf.lock.class property in the **etc/system.properties** file must point to org.apache.karaf.main.PostgreSQLJDBCLock. For example, you could configure the **system.properties** file as shown:

Example 18.6. JDBC Lock File Configuration for PostgreSQL

```
karaf.lock=true
karaf.lock.class=org.apache.karaf.main.PostgreSQLJDBCLock
```

```
karaf.lock.jdbc.url=jdbc:postgresql://127.0.0.1:5432/dbname
karaf.lock.jdbc.driver=org.postgresql.Driver
karaf.lock.jdbc.user=user
karaf.lock.jdbc.password=password
karaf.lock.jdbc.table=KARAF_LOCK
karaf.lock.jdbc.clustername=karaf
karaf.lock.jdbc.timeout=0
```

JDBC lock classes

The following JDBC lock classes are currently provided by Apache Karaf:

```
org.apache.karaf.main.DefaultJDBCLock
org.apache.karaf.main.DerbyJDBCLock
org.apache.karaf.main.MySQLJDBCLock
org.apache.karaf.main.OracleJDBCLock
org.apache.karaf.main.PostgreSQLJDBCLock
```

18.3. CONTAINER-LEVEL LOCKING

Overview

Container-level locking allows bundles to be preloaded into the slave kernel instance in order to provide faster failover performance. Container-level locking is supported in both the simple file and JDBC locking mechanisms.

Configuring container-level locking

To implement container-level locking, add the following to the **`etc/system.properties`** file on each system in the master/slave setup:

Example 18.7. Container-level Locking Configuration

```
karaf.lock=true
karaf.lock.level=50
karaf.lock.delay=10000
```

The karaf.lock.level property tells the Red Hat JBoss Fuse instance how far up the boot process to bring the OSGi container. Bundles assigned the same start level or lower will then also be started in that JBoss Fuse instance.

Bundle start levels are specified in **`etc/startup.properties`**, in the format *BundleName*.jar=level. The core system bundles have levels below 50, where as user bundles have levels greater than 50.

Table 18.1. Bundle Start Levels

Start Level	Behavior
1	A 'cold' standby instance. Core bundles are not loaded into container. Slaves will wait until lock acquired to start server.
<50	A 'hot' standby instance. Core bundles are loaded into the container. Slaves will wait until lock acquired to start user level bundles. The console will be accessible for each slave instance at this level.
>50	This setting is not recommended as user bundles will be started.

Avoiding port conflicts

When using a 'hot' spare on the same host you need to set the JMX remote port to a unique value to avoid bind conflicts. You can edit the **`fuse`** start script (or the **`karaf`** script on a child instance) to include the following:

```
DEFAULT_JAVA_OPTS="-server $DEFAULT_JAVA_OPTS -
Dcom.sun.management.jmxremote.port=1100 -
Dcom.sun.management.jmxremote.authenticate=false"
```

CHAPTER 19. APPLYING PATCHES

Abstract

Red Hat JBoss Fuse supports incremental patching. FuseSource will supply you with easy to install patches that only make targeted changes to a deployed container.

19.1. PATCHING OVERVIEW

Patching enables you apply fixes to a broker without needing to reinstall an updated version of Red Hat JBoss Fuse. It also allows you to back out the patch, if it causes problems with your deployed applications.

Patches are ZIP files that contain the artifacts needed to update a targeted set of bundles in a container. The artifacts are typically one or more bundles. They can, however, include configuration files and feature descriptors.

You get a patch file in one of the following ways:

- Customer Support sends you a patch.
- Customer Support sends you a link to download a patch.
- Download a patch directly from the Red Hat customer portal.

The process of applying a patch to a container depends on how the container is deployed:

- *Standalone (standard process)*—using commands from the Karaf console's **`patch`** shell. This approach is non-destructive and reversible.
- *Fabric*—patching a fabric requires applying the patch to a profile and then applying the profile to a container.

19.2. FINDING THE RIGHT PATCHES TO APPLY

Abstract

This section explains how to find the patches for a specific version of JBoss Fuse on the Red Hat Customer Portal and how to figure out which patches to apply, and in what order.

Locate the patches on the customer portal

If you have a subscription for JBoss Fuse, you can download the latest patches directly from the Red Hat Customer Portal. Locate the patches as follows:

1. Login to the Red Hat Customer Portal using your customer account. This account *must* be associated with an appropriate Red Hat software subscription, otherwise you will not be able to see the patch downloads for JBoss Fuse.

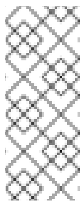
2. Navigate to the customer portal Software Downloads page.

3. In the **`Product`** dropdown menu, select the appropriate product (for example, **`A-MQ`** or **`Fuse`**), and then select the version, 6.3, from the **`Version`** dropdown menu. A table of downloads now appears, which has three tabs: **`Releases`**, **`Patches`**, and **`Security Advisories`**.

4. Click the **`Releases`** tab to view the GA product releases.

5. Click the **`Patches`** tab the rollup patches, and the regular incremental patches (with no security-related fixes).

6. Click the **`Security Advisories`** tab to view the incremental patches with security-related fixes.



Note

To see the *complete* set of patches, you must look under the **`Releases`** tab, the **`Patches`** tab *and* the **`Security Advisories`** tab.

Types of patch

The following types of patch can be made available for download:

- Rollup patches

- Incremental patches

Rollup patches

A rollup patch is a cumulative patch that incorporates *all* of the fixes from the preceding patches. Moreover, each rollup patch is regression tested and establishes a new baseline for the application of future patches.

Since JBoss Fuse 6.2.1, a rollup patch file is dual-purpose, as follows:

- Each rollup patch file is a complete new build of the official target distribution. This means you can unzip the rollup patch file to obtain a completely new installation of JBoss Fuse, just as if it was a fresh download of the product (which, in fact, it is). See Section 19.3, “Installing a Rollup Patch as a New Installation”.

- You can also treat the rollup patch as a regular patch, using it to upgrade an existing installation.

That is, you can provide the rollup patch file as an argument to the standalone patch console commands (for example, **`patch:add`** and **`patch:install`**) or the Fabric patch console command, **`patch:fabric-install`**.

Incremental patches

Incremental patches are patches released either directly after GA or after a rollup patch, and they are intended to be applied on top of the corresponding build of JBoss Fuse. The main purpose of an incremental patch is to update some of the bundles in an existing distribution.

Which patches are needed to update the GA product to the latest patch level?

To figure out which patches are needed to update the GA product to the latest patch level, you need to pay attention to the type of patches that have been released so far:

1. If the only patches released so far are patches with GA baseline (Patch 1, Patch 2, and so on), apply the *latest* of these patches directly to the GA product.

2. If a rollup patch has been released and no patches have been released after the latest rollup patch, simply apply the latest rollup patch to the GA product.

3. If the latest patch is a patch with a rollup baseline, you must apply two patches to the GA product, as follows:

    a. Apply the latest rollup patch, and then

    b. Apply the latest patch with a rollup baseline.

Which patches to apply, if you only want to install regression-tested patches?

If you prefer to install only patches that have been regression tested, install the latest rollup patch.

19.3. INSTALLING A ROLLUP PATCH AS A NEW INSTALLATION

A rollup patch is a new build

Since JBoss Fuse 6.2.1, a rollup patch file is a complete new build of the official target distribution. In other words, it is just like the original GA distribution, except that it includes later build artifacts.

Installing the new build

To install a new build, corresponding to a rollup patch level, perform the following steps:

1. Identify which rollup patch you need to install and download it from the Customer Portal. For more details, see Section 19.2, “Finding the Right Patches to Apply”.

2. Unzip the rollup patch file to a convenient location, just as you would with a regular GA distribution. This is your new installation of JBoss Fuse.

Comparison with patch process

Compared with the conventional patch process, installing a new build has the following advantages and limitations:

- This approach is only for creating a completely new installation of JBoss Fuse. If your existing installation already has a lot of custom configuration, this might not be the most convenient approach to use.
- The new build includes only the artifacts and configuration for the new patch level. There is thus no concept of *rolling back* to the GA version.
- If you create a new fabric (using **`fabric:create`**), the default fabric profiles are already at the new patch level (same as the standalone container). It is therefore not necessary to patch the fabric.

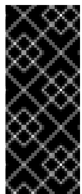
19.4. PATCHING A STANDALONE CONTAINER

Abstract

You apply patches to a standalone container using the command console's **`patch`** shell. You can apply and roll back patches as needed.

Overview

When patching a standalone container, you can apply either an incremental patch or a rollup patch. There are very significant differences between the two kinds of patch and the way they are applied. Although the same commands are used in both cases, the internal processes are different (the patch commands auto-detect the patch type).



Important

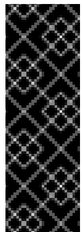
The instructions in this section apply only to JBoss Fuse versions 6.2.1 and later, which support the new patching mechanism.

Incremental patch

An incremental patch is used mainly to update the *bundle JARs* in the container. This type of patch is suitable for delivering hot fixes to the JBoss Fuse installation, but it has its limitations. An incremental patch:

- Updates bundle JARs.

- Patches only the current container instance (under the **data/** directory). Hence, patches are *not* preserved after deleting a container instance.
- Updates any feature dependencies installed in the current container instance, but does not update the feature files themselves.
- Might update some configuration files, but is *not* suitable for updating most configuration files.
- Supports patch rollback.
- After applying the patch, and creating a new fabric using **fabric:create**, the new Fabric reverts to the unpatched configuration.



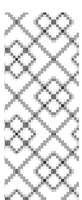
Important

Incremental patches are *not* preserved after deleting a container instance. That is, if you delete a container instance by deleting the **data/** directory (for the root container), you will need to re-apply the patch after restarting the container (cold start).

Rollup patch

A rollup patch can make updates to *any* installation files including *bundle JARs* and *static files* (including, for example, configuration files under the **etc/** directory). A rollup patch:

- Updates any files, including bundle JARs, configuration files, and any static files.
- Patches both the current container instance (under the **data/** directory) and the underlying installation. Hence, patches are preserved after deleting a container instance.
- Updates all of the files related to Karaf features, including the features repository files and the features themselves. Hence, any features installed after the rollup patch will reference the correct patched dependencies.
- If necessary, updates configuration files (for example, files under **etc/**), automatically merging any configuration changes you have made with the configuration changes made by the patch. If merge conflicts occur, see the patch log for details of how they are handled.
- Tracks *all* of the changes made to the installation (including to static files), so that it is possible to roll back the patch.

  Note

  The rollup patching mechanism uses an internal git repository (located under **patches/.management/history**) to track the changes made.

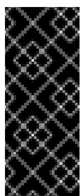
- After applying the patch, and creating a new fabric using **`fabric:create`**, the new Fabric is created with the patched configuration.

Patching the patch mechanism

(Optional) Before upgrading JBoss Fuse with a rollup patch, you can optionally patch the patch mechanism to a higher level. Since the original GA version of JBoss Fuse 6.3 was released, significant improvements have been made to the patch mechanism. If you upgrade to the latest rollup patch version of JBoss Fuse, the improved patch mechanism will become available *after* you have completed the upgrade. But at that stage, it will be too late to benefit from the improvements in the patch mechanism.

To circumvent this bootstrap problem, the improved patch mechanism is made available as a separate download, so that you can patch the patch mechanism itself, *before* you upgrade to the new patch level. This step is optional, but recommended. To patch the patch mechanism, proceed as follows:

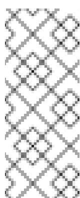
1. Download the appropriate patch management package. From the JBoss Fuse 6.3.0 Software Downloads page, select a package named **`Red Hat JBoss Fuse 6.3.0 Rollup N on Karaf Update Installer`**, where *N* is the number of the particular rollup patch you are about to install.

   Important

   The rollup number, *N*, of the downloaded patch management package *must* match the rollup number of the rollup patch you are about to install.

2. Install the patch management package, **`patch-management-for-fuse-630-TargetVersion.zip`**, on top of your 6.3.0 installation. Use an archive utility to extract the contents on top of the existing Karaf container installation (installing files under the **`system/`** and **`patches/`** subdirectories).

   Note

   It does not matter whether the container is running or not when you extract these files.

3. Start the container, if it is not already running.

4. Uninstall the existing patch commands from the container as follows. Remove the patch features as follows:

   ```
   JBossFuse:karaf@root> features:uninstall patch patch-core
   ```

   But this is not sufficient to remove all of the patch bundles. Check for any remaining patch bundles as follows:

   ```
   JBossFuse:karaf@root> list -t 0 -l | grep patch
   ```

```
[   1] [Active     ] [            ] [       ] [    2]
mvn:io.fabric8.patch/patch-management/1.2.0.redhat-630187
```

You can remove this system bundle, as follows:

```
JBossFuse:karaf@root> uninstall 1
You are about to access system bundle 1.  Do you wish to continue
(yes/no): yes
JBossFuse:karaf@root> list -t 0 -l | grep patch
```

Finally, you should remove the features URL for the old patch mechanism version, as follows:

```
JBossFuse:karaf@root> features:listurl | grep patch
  true    mvn:io.fabric8.patch/patch-features/1.2.0.redhat-
630187/xml/features
JBossFuse:karaf@root> features:removeurl
mvn:io.fabric8.patch/patch-features/1.2.0.redhat-
630187/xml/features
```

Check the version of **`patch-features`** that you have, because it might be different from **`1.2.0.redhat-630187`**.

5. Install the new patch commands as follows. Add the features URL for the new patch commands, as follows:

```
JBossFuse:karaf@root> features:addurl mvn:io.fabric8.patch/patch-
features/1.2.0.redhat-630xxx/xml/features
```

Where you must replace **`1.2.0.redhat-630xxx`** with the actual build version of the patch commands you are installing (for example, the build version **`xxx`** can be taken from the last three digits of the ***`TargetVersion`*** in the downloaded patch management package file name).

Install the new patch features, as follows:

```
JBossFuse:karaf@root> features:install patch-core patch
```

Check that the requisite patch bundles are now installed:

```
JBossFuse:karaf@root> list -t 0 -l | grep patch
[ 265] [Active     ] [            ] [       ] [   80]
mvn:io.fabric8.patch/patch-core/1.2.0.redhat-630xxx
[ 266] [Active     ] [            ] [       ] [    2]
mvn:io.fabric8.patch/patch-management/1.2.0.redhat-630xxx
[ 267] [Active     ] [            ] [       ] [   80]
mvn:io.fabric8.patch/patch-commands/1.2.0.redhat-630xxx
```

Applying a patch

To apply a patch to a standalone container:

1. Make a full backup of your JBoss Fuse installation before attempting to apply the patch.

2. *(Incremental patch only)* Before you proceed to install the patch, make sure to read the text of the **README** file that comes with the patch, as there might be additional *manual steps* required to install a particular patch.

3. Start the container, if it is not already running. If the container is running in the background (or remotely), connect to the container using the SSH console client, **/bin/client**.

4. Add the patch to the container's environment using the **patch:add** command. For example, to add the ***patch*.zip** patch file:

   ```
   patch:add file://patch.zip
   ```

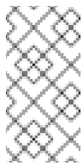
5. Simulate installing the patch using the **patch:simulate** command.

   This will generate a log of the changes that will be made to the container when the patch is installed, but will not make any actual changes to the container. Review the simulation log to understand the changes that will be made to the container.

6. Invoke the **patch:list** command to display a list of all added patches. From this list, you can get the ID, ***PatchID***, of the patch you want to install.

7. Apply the patch to the container using the **patch:install** command:

   ```
   patch:install PatchID
   ```

   Note

   Make sure that the container has fully started before you run **patch:install**.

   In some cases the container will need to restart to apply the patch (for example, if static files are patched). In these cases, the container restarts automatically.

8. Validate the patch, by searching for one of the patched artifacts. For example, if you had just upgraded JBoss Fuse 6.2.1 to the patch with build number **621423**, you could search for bundles with this build number, as follows:

   ```
   JBoss Fuse:karaf@root> osgi:list -s -t 0 | grep -i 630187
   [   6] [Active     ] [            ] [       ] [   10]
   org.apache.felix.configadmin    (1.2.0.redhat-630187)
   ```

After applying a rollup patch, you will also see the new version and build number in the Welcome banner when you restart the container.

Rolling back a patch

Occasionally a patch will not work or will introduce new issues to a container. In these cases, you can easily back the patch out of the system and restore it to pre-patch behaviour using the **`patch:rollback`** command, as follows:

1. Invoke the **`patch:list`** command to obtain the patch ID, ***`PatchID`***, of the most recently installed patch.

2. Invoke the **`patch:rollback`** command, as follows:

   ```
   patch:rollback PatchID
   ```

   In some cases the container will need to restart to roll back the patch. In these cases, the container restarts automatically. Due to the highly dynamic nature of the OSGi runtime, during the restart you might see some occasional errors related to incompatible classes. These are related to OSGi services that have just started or stopped. These errors can be safely ignored.

Adding features to an incrementally patched container

Since JBoss Fuse 6.1, it is possible to add Karaf features to an already patched standalone container without performing any special steps.

19.5. PATCHING A CUSTOM ASSEMBLY

Overview

Red Hat does not provide patches specifically for a custom assembly. Nevertheless, it *is* possible to generate a patched custom assembly by following the instructions in this section.

Custom assembly

A *custom assembly* is a custom distribution of the Karaf container based on JBoss Fuse, which can be generated using the **`quickstart/custom`** template. The custom assembly deploys a customized set of Karaf features and can be used to cut down the size of a JBoss Fuse deployment.

Inverting the patching procedure

Red Hat does *not* release patches for specific custom assemblies. Because there is an endless variety of possible custom assemblies, it is not practical to release patches for specific custom assemblies. Nevertheless, it is possible to produce a patched custom assembly. The trick is to *invert* the procedure for producing a patch: instead of starting with a custom assembly and attempting to apply a rollup patch to this assembly, what you must do is start with the rollup patch (which is also a

product distribution) and follow the procedure for generating a custom assembly in the unpacked rollup patch.

A rollup patch is a JBoss Fuse distribution

It is crucial to understand that a rollup patch is *also* a JBoss Fuse distribution (under the new patching system). In other words, if you unpack a rollup patch archive file, you will discover that it has the identical directory layout to a full release. Not only that, but it has all of the files you would expect to find in a full release. In other words, the rollup patch is equivalent to a *pre-patched* container.

Generating a patched custom assembly

To generate a patched custom assembly, you need the latest rollup patch from the Red Hat Customer Portal (for details of how to find the right patch, see Section 19.2, “Finding the Right Patches to Apply”). Follow the steps for generating a patched custom assembly, as described in appendix "Generating a Custom Assembly or an Offline Repository" in "Installation on Apache Karaf".

19.6. PATCHING A FABRIC CONTAINER WITH A ROLLUP PATCH

Abstract

Follow the procedures described in this section to patch a Fabric container with a *rollup patch*.

Overview

A rollup patch updates *bundle JARs*, other *Maven artifacts*, *libraries*, and *static files* in a Fabric. The following aspects of the fabric are affected:

- Distribution of patched artifacts
- Profiles
- Configuration of the underlying container

Distribution of patch artifacts

When patching an entire fabric of containers, you need to consider how the patch artifacts are distributed to the containers in the fabric. You can adopt one of the following approaches:

- *Through the Maven proxy* (default approach)—when you add a rollup patch to your root container (using the **`patch:add`** command), the patch artifacts are installed into the root container's **`system/`** directory, whose directory structure is laid out like a Maven repository. The root container can then serve up these patch artifacts to remote containers by behaving as a Maven proxy, enabling remote containers to download the required Maven artifacts (this process is managed by the Fabric agent running on each Fabric container). Alternatively, if you have

installed the rollup patch to a container that is *not* hosting the Maven proxy, you can ensure that the patch artifacts are uploaded to the Maven proxy by invoking the **patch:fabric-install** command with the **--upload** option.

There is a limitation to the Maven proxy approach, however, if the Fabric ensemble consists of multiple containers. In this case, it can happen that the Maven proxy fails over to a different ensemble container (not the original root container). This can result in the patch artifacts suddenly becoming unavailable to other containers in the fabric. If this occurs during the patching procedure, it will cause problems.

For more details, see chapter "Fabric Maven Proxies" in "Fabric Guide".

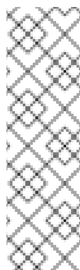
- *Through a local repository* (recommended approach)—to overcome the limitations of the Maven proxy approach, we recommend that you make the patch artifacts available directly to all of the containers in the Fabric by setting up a *local repository* on the file system. Assuming that you have a networked file system, all containers will be able to access the patch artifacts directly.

  For example, you might set up a local repository of patch artifacts, as follows:

  - Given a rollup patch file, extract the contents of the **system/** directory from the rollup patch file into the **repositories/** subdirectory of a local Maven repository (which could be **~/.m2/repositories** or any other location).
  - Configure the Fabric agent and the Maven proxy to pick up artifacts from the local repository by editing the current version of the **default** profile, as follows:

    ```
    profile-edit --append --pid
    io.fabric8.agent/org.ops4j.pax.url.mvn.defaultRepositories
    file:///PathToRepository default
    ```

    Replace ***PathToRepository*** by the actual location of the local repository on your file system.

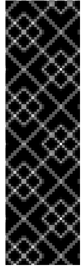
    

    Note

    Make sure that you make the edits to the **default** profile for all relevant profile versions. If some of your containers are using a non-default profile version, repeat the **profile-edit** commands while specifying the profile version explicitly as the last parameter.

Profiles

The rollup patching process updates all of the standard profiles, so that they reference the patched dependencies. Any custom profiles that you created yourself remain unaffected by these updates. However, in cases where you have already made some changes directly to the *standard profiles* (such as **default**, **fabric**, **karaf**, and so on), the patching mechanism attempts to merge your changes with the changes introduced by the patch.



Important

In the case where you have modified standard profiles, it is recommended that you verify your custom changes are preserved after patching. This is particularly important with respect to any changes made to the location of Maven repositories (which are usually configured in the **`default`** profile).

Configuration of the underlying container

If required, the rollup patching mechanism is capable of patching the underlying container (that is, files located under **`etc/`**, **`lib/`**, and so on). When a Fabric container is upgraded to a patched version (for example, using the **`fabric:container-upgrade`** command), the container's Fabric agent checks whether the underlying container must be patched. If yes, the Fabric agent triggers the patching mechanism to update the underlying container. Moreover, if certain critical files are updated (for example, **`lib/karaf.jar`**), the container status changes to **`requires full restart`** after the container is upgraded. This status indicates that a full*manual* restart is required (an automatic restart is not possible in this case).

io.fabric.version in the default profile

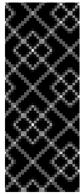
The **`io.fabric.version`** resource in the **`default`** profile plays a key role in the patching mechanism. This resource defines the version and build of JBoss Fuse and of all of its main components. When upgrading (or rolling back) a Fabric container to a new version, the Fabric agent checks the version and build of JBoss Fuse as defined in the **`io.fabric.version`** resource. If the JBoss Fuse version changes between the original profile version and the upgraded profile version, the Fabric agent knows that an upgrade of the underlying container is required when upgrading to this profile version.

Patching the patch mechanism

(Optional) Before upgrading JBoss Fuse with a rollup patch, you can optionally patch the patch mechanism to a higher level. Since the original GA version of JBoss Fuse 6.3 was released, significant improvements have been made to the patch mechanism. If you upgrade to the latest rollup patch version of JBoss Fuse, the improved patch mechanism will become available *after* you have completed the upgrade. But at that stage, it will be too late to benefit from the improvements in the patch mechanism.

To circumvent this bootstrap problem, the improved patch mechanism is made available as a separate download, so that you can patch the patch mechanism itself, *before* you upgrade to the new patch level. This step is optional, but recommended. To patch the patch mechanism, proceed as follows:

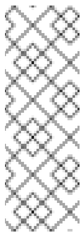
1. Download the appropriate patch management package. From the JBoss Fuse 6.3.0 Software Downloads page, select a package named **`Red Hat JBoss Fuse 6.3.0 Rollup N on Karaf Update Installer`**, where *N* is the number of the particular rollup patch you are about to install.



Important

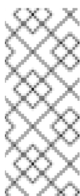
The rollup number, *N*, of the downloaded patch management package *must* match the rollup number of the rollup patch you are about to install.

2. Install the patch management package, **`patch-management-for-fuse-630-TargetVersion.zip`**, on top of your root container installation. Use an archive utility to extract the contents on top of the existing root container installation (installing files under the **`system/`** and **`patches/`** subdirectories).



Note

The *root container* is the ensemble container in your fabric that is currently serving Maven artifacts through the Maven proxy to other containers in the fabric. See chapter "Fabric Maven Proxies" in "Fabric Guide".



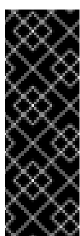
Note

It does not matter whether the root container is running or not when you extract these files.

3. Start the root container, if it is not already running.

4. Create a new version, using the **`fabric:version-create`** command (where we assume that the current profile version is **`1.0`**):

```
JBossFuse:karaf@root> fabric:version-create --parent 1.0 1.0.1
Created version: 1.0.1 as copy of: 1.0
```



Important

The version name must be a pure *numeric* string, such as **`1.1`**, **`1.2`**, **`2.1`**, or **`2.2`**. You cannot incorporate alphabetic characters in the version name (such as **`1.0.patch`**).

5. Update the **`patch`** property in the **`io.fabric8.version`** PID in the version **`1.0.1`** of the **`default`** profile, by entering the following Karaf console command:

```
profile-edit --pid io.fabric8.version/patch=1.2.0.redhat-630xxx
default 1.0.1
```

Where you must replace **`1.2.0.redhat-630xxx`** with the actual build version of the patch commands you are installing (for example, the build version **`xxx`** can be taken from the last three digits of the ***`TargetVersion`*** in the downloaded patch management package file name).

6. Upgrade the root container to use the new patching mechanism, as follows:

```
container-upgrade 1.0.1 root
```

7. Likewise, for any other containers in your fabric that need to be patched, you can provision them with the new patching mechanism by upgrading them to profile version **1.0.1**. For example:

```
container-upgrade 1.0.1 container1 container2 container3
```

Applying a rollup patch

To apply a rollup patch to a Fabric container:

1. Add the patch to the root container's environment using the **patch:add** command. For example, to add the ***patch*.zip** patch file:

```
JBossFuse:karaf@root> patch:add file://patch.zip
[name]              [installed] [description]
PatchID            false        Description
```

Important

If you have decided to use a local repository to distribute the patch artifacts *(recommended)*, set up the local repository now—see the section called “Distribution of patch artifacts”.

2. Create a new version, using the **fabric:version-create** command:

```
JBossFuse:karaf@root> fabric:version-create 1.1
Created version: 1.1 as copy of: 1.0
```

Important

The version name must be a pure *numeric* string, such as **1.1**, **1.2**, **2.1**, or **2.2**. You cannot incorporate alphabetic characters in the version name (such as **1.0.patch**).

Note

If you followed the optional steps to *patch the patch mechanism*, the profile version **1.1** would be created as a copy of version **1.0.1**, not **1.0**, in the above example.

3. Apply the patch to the new version, using the **patch:fabric-install** command. Note that in order to run this command you *must* provide the credentials, ***Username*** and

Password, of a user with **Administrator** privileges. For example, to apply the ***PatchID*** patch to version **1.1**:

```
patch:fabric-install --username Username --password Password --
upload --version 1.1 PatchID
```

Note

When you invoke the **patch:fabric-install** command with the **--upload** option, Fabric looks up the ZooKeeper registry to discover the URL of the currently active Maven proxy, and uploads all of the patch artifacts to this URL. Using this approach it is possible to make the patch artifacts available through the Maven proxy, even if the container you are currently logged into is not hosting the Maven proxy.

4. Synchronize the patch information across the fabric, to ensure that the profile changes in version **1.1** are propagated to all containers in the fabric (particularly remote SSH containers). Enter the following console command:

```
patch:fabric-synchronize
```

5. Upgrade each existing container in the fabric using the **fabric:container-upgrade** command (but leaving the root container, where you installed the patch, until last). For example, to upgrade a container named **remote**, enter the following command:

```
JBossFuse:karaf@root> fabric:container-upgrade 1.1 remote
Upgraded container remote from version 1.0 to 1.1
```

At this point, not only does the Fabric agent download and install the patched bundles into the specified container, but *the agent also applies the patch to the underlying container* (updating any static files in the container, if necessary). If necessary, the agent will then restart the target container automatically or set the container status to **requires full restart** (if an automatic restart is not possible), so that any changes made to the static files are applied to the running container.

Important

It is recommended that you upgrade only one or two containers to the patched profile version, to ensure that the patch does not introduce any new issues.

6. If the current status of the upgraded container is **requires full restart**, you must now use one of the standard mechanisms to stop and restart the container manually. In some cases, it will be possible to do this using Fabric commands from the console of the root container.

For example, you could stop the **remote** container as follows:

```
fabric:container-stop remote
```

And restart the **remote** container as follows:

```
fabric:container-start remote
```

7. Upgrade the root container last (that is, the container that you originally installed the patch on):

```
fabric:container-upgrade 1.1 root
```

Rolling back a rollup patch

To roll back a rollup patch on a Fabric container, use the **fabric:container-rollback** command. For example, assuming that **1.0** is an unpatched profile version, you can roll the **remote** container back to the unpatched version **1.0** as follows:

```
fabric:container-rollback 1.0 remote
```

At this point, not only does the Fabric agent roll back the installed profiles to an earlier version, but *the agent also rolls back the patch on the underlying container* (restoring any static files to the state they were in before the patch was applied, if necessary). If necessary, the agent will then restart the target container automatically or set the container status to **requires full restart** (if an automatic restart is not possible), so that any changes made to the static files are applied to the running container.

19.7. PATCHING A FABRIC CONTAINER WITH AN INCREMENTAL PATCH

Abstract

Follow the procedures described in this section to patch a Fabric container with an *incremental patch*.

Overview

An incremental patch makes updates only to the *bundle JARs* in a Fabric. The following aspects of the fabric are affected:

- Distribution of patched artifacts through Maven proxy
- Profiles

Distribution of patched artifacts through Maven proxy

When you install the incremental patch on your local container, the patch artifacts are installed into the local **system/** directory, whose directory structure is laid out like a Maven repository. The local container distributes these patch artifacts to remote containers by behaving as a Maven proxy,

enabling remote containers to upload bundle JARs as needed (this process is managed by the Fabric agent running on each Fabric container). For more details, see chapter "Fabric Maven Proxies" in "Fabric Guide".

Profiles

The incremental patching process defines bundle overrides, so that profiles switch to use the patched dependencies (bundle JARs). This mechanism works as follows:

1. The patch mechanism creates a new profile, **patch-*PatchProfileID***, which defines bundle overrides for all of the patched bundles.

2. The new patch profile, **patch-*PatchProfileID***, is inserted as the parent of the **default** profile (at the base of the entire profile tree).

3. All of the profiles that inherit from default now use the bundle versions defined by the overrides in **patch-*PatchProfileID***. The contents of the existing profiles themselves *are not modified* in any way.

Is it necessary to patch the underlying container?

Usually, when patching a fabric with an incremental patch, it is *not* necessary to patch the underlying container as well. Fabric has its own mechanisms for distributing patch artifacts (for example, using a git repository for the profile data, and Apache Maven for the OSGi bundles), which are independent of the underlying container installation.

In exceptional cases, however, it might be necessary to patch the underlying container (for example, if there was an issue with the **fabric:create** command). Always read the patch **README** file to find out whether there are any special steps required to install a particular patch. In these cases, however, it is more likely that the patch would be distributed in the form of a rollup patch, which has the capability to patch the underlying container automatically—see Section 19.6, “Patching a Fabric Container with a Rollup Patch”.

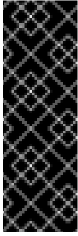
Applying an incremental patch

To apply an incremental patch to a Fabric container:

1. Before you proceed to install the incremental patch, make sure to read the text of the **README** file that comes with the patch, as there might be additional *manual steps* required to install a particular incremental patch.

2. Create a new version, using the **fabric:version-create** command:

```
JBossFuse:karaf@root> fabric:version-create 1.1
Created version: 1.1 as copy of: 1.0
```



Important

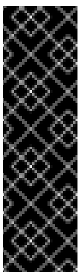
The version name must be a pure *numeric* string, such as **1.1**, **1.2**, **2.1**, or **2.2**. You cannot incorporate alphabetic characters in the version name (such as **1.0.patch**).

3. Apply the patch to the new version, using the **fabric:patch-apply** command. For example, to apply the **activemq.zip** patch file to version **1.1**:

```
JBossFuse:karaf@root> fabric:patch-apply --version 1.1
file:///patches/activemq.zip
```

4. Upgrade a container using the **fabric:container-upgrade** command, specifying which container you want to upgrade. For example, to upgrade the **child1** container, enter the following command:

```
JBossFuse:karaf@root> fabric:container-upgrade 1.1 child1
Upgraded container child1 from version 1.0 to 1.1
```



Important

It is recommended that you upgrade only one or two containers to the patched profile version, to ensure that the patch does not introduce any new issues. Upgrade the **root** container (the one that you applied the patch to, using the **fabric:patch-apply** command) last.

5. You can check that the new patch profile has been created using the **fabric:profile-list** command, as follows:

```
BossFuse:karaf@root> fabric:profile-list --version 1.1 | grep patch
default                                  0              patch-
activemq-patch
patch-activemq-patch
```

Where we presume that the patch was applied to profile version 1.1.

Tip

If you want to avoid specifying the profile version (with **--version**) every time you invoke a profile command, you can change the default profile version using the **fabric:version-set-default *Version*** command.

You can also check whether specific JARs are included in the patch, for example:

```
JBossFuse:karaf@root> list | grep -i activemq
[ 131] [Active     ] [Created     ] [       ] [   50] activemq-osgi
(5.9.0.redhat-61037X)
```

```
[ 139] [Active     ] [Created     ] [       ] [   50] activemq-
karaf (5.9.0.redhat-61037X)
[ 207] [Active     ] [            ] [       ] [   60] activemq-
camel (5.9.0.redhat-61037X)
```

Rolling back an incremental patch

To roll back an incremental patch on a Fabric container, use the **`fabric:container-rollback`** command. For example, assuming that **`1.0`** is an unpatched profile version, you can roll the **`child1`** container back to the unpatched version **`1.0`** as follows:

```
fabric:container-rollback 1.0 child1
```

CHAPTER 20. FABRIC MAVEN PROXIES

Abstract

Container hosts often have limited or no access to the Internet, which can make it difficult for Fabric containers to download and install Maven artifacts. This problem can be mitigated using a *Maven proxy*, which serves as a central cache of Maven artifacts for the Fabric containers. Managed containers try to download from the Maven proxy, before trying to download from the Internet. This chapter explains how the Maven proxy works and how to customize the configuration of the Maven proxy to suit your network environment.

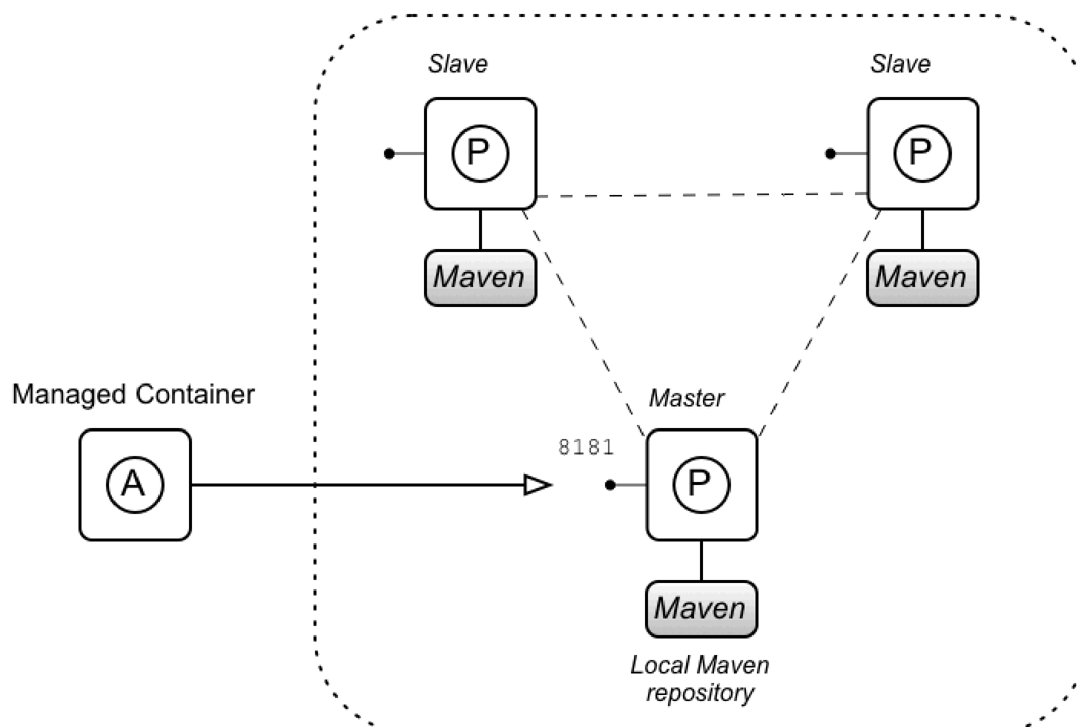
20.1. CLUSTER OF FABRIC MAVEN PROXIES

Overview

Fabric Maven proxies are deployed *only* on Fabric servers (ensemble members), not on regular managed containers. So, if there is just a single Fabric server in your fabric, there will be just one Maven proxy. But if your Fabric ensemble consists of multiple servers (for example, three or five), a Maven proxy is deployed on each server, and this cluster of Maven proxies is configured automatically as a *master-slave cluster*.

Figure 20.1, “Maven Proxy Cluster” shows the outline of a Maven proxy cluster consisting of three Fabric servers (which constitute the Fabric ensemble).

Figure 20.1. Maven Proxy Cluster



Master-slave cluster

Each Maven proxy is deployed inside a Fabric server (a container that belongs to the Fabric ensemble) and the Maven proxies together are organized as a *master-slave cluster*. This means that one of the Maven proxies in the cluster is elected to be the *master*, while all of the other Maven proxies remain as *slaves*. Only the master proxy is available to serve up Maven artifacts, while the slave proxies remain in a suspended state.

The master-slave architecture is implemented with the help of Apache Zookeeper distributed locking. At start-up time, each of the Maven proxies attempts to acquire a Zookeeper lock: the proxy that succeeds becomes the master, while the remaining proxies remain as slaves.

Maven proxy

A Maven proxy is a HTTP Web server that behaves very much like a standard Maven repository, such as Maven Central.

The purpose of the Maven proxy is to serve Maven artifacts on the local network. It has its own local cache of Maven artifacts, which it can serve up quickly. But if necessary, the Maven proxy can also download artifacts from remote repositories (in a proxy role). This architecture offers a number of advantages:

- The Maven proxy builds up a large cache over time, which can be served up quickly to other containers in the Fabric.
- It is not necessary for every container to download Maven artifacts from remote repositories—the Maven proxy performs this service for the other containers.
- In a network with limited Internet access, you can arrange to deploy the Maven proxy on a host with Internet access, while the other containers in the fabric are deployed on hosts without Internet access.

Managed container

A *managed container* is a regular Fabric container (not part of the Fabric ensemble), whose contents are managed by a Fabric8 agent. The Fabric8 agent is responsible for ensuring that the bundles deployed in the container are consistent with what is specified in this container's Fabric profiles. Whenever necessary, the Fabric8 agent will contact the Maven proxy to download new Maven artifacts for deploying inside the container.

Resolving a Maven artifact

The Fabric8 agent attempts to locate a Maven artifact roughly as follows:

1. The Fabric8 agent searches its local Maven repository for the artifact.
2. If that fails, the Fabric8 agent contacts the Maven proxy to request the artifact.
3. If that fails, the Fabric8 agent attempts to contact remote Maven repositories directly to request the artifact.

For a more detailed outline of this process, see Section 20.2, “How a Managed Container Resolves Artifacts”.

Endpoint discovery

Before the Fabric8 agent can connect to the Maven proxy, it needs to discover the HTTP address of the current master instance (only the master instance is usable, because the slave instances are dormant). The discovery mechanism is based on the Apache Zookeeper registry: by querying Zookeeper, the Fabric8 agent can discover the URL of the current master instance.

Which Fabric server is the current master?

You can query Zookeeper manually (using console commands) to discover the URL of the current Maven proxy master instance. To discover the URLs for the current Maven proxy master, invoke the **`fabric:cluster-list`** console command, as follows:

```
JBossFuse:karaf@root> cluster-list servlets/io.fabric8.fabric-maven-proxy
[cluster]                           [masters]  [slaves]  [services]
[]
1.2.0.redhat-621084/maven/download
   root                             root       -
http://127.0.0.1:8181/maven/download
1.2.0.redhat-621084/maven/upload
   root                             root       -
http://127.0.0.1:8181/maven/upload
```

The preceding example is trivial, because there is only one Fabric server (the **`root`** container) in this Fabric ensemble. This command returns two URLs: one for downloading artifacts (**`http://127.0.0.1:8181/maven/download`**), and another for uploading artifacts (**`http://127.0.0.1:8181/maven/upload`**). For more details about uploading artifacts, see Section 20.6, “Automated Deployment”.

What happens during failover?

Normally, the master instance remains the master instance for as long as the Maven proxy is deployed and running in its container. However, if the container hosting the master Maven proxy gets shut down (for whatever reason), the master instance releases the Zookeeper lock, and one of the slave instances has the opportunity to be promoted to master. Each of the slave instances retries the Zookeeper lock at regular time intervals and the first slave that retries the lock will acquire the lock and become the new master.

When the cluster fails over and a former slave becomes the new master, this has important consequences:

- The URLs for the master Maven proxy are changed. Clients must now connect to a *different* URL to connect to the Maven proxy. For Fabric8 agents, this failover is transparent, because the Fabric8 agent automatically rediscovers the new URLs.

- If you have been automatically uploading artifacts to the Maven proxy as part of your build process (see Section 20.6, “Automated Deployment”), you will need to reconfigure the upload URL. In this case, failover is *not* transparent.
- It is likely that the new master has a much smaller cache of Maven artifacts than the old master. This could result in noticeable delays, because many previously cached artifacts have to be downloaded again.

No replication

Within the Maven proxy cluster, there is *no automatic replication* of artifacts between different Maven proxies in the cluster. You will probably notice the effects of this, when the cluster fails over to a new Maven proxy.

Managing the Maven artifact data

Although Fabric does not support replication of the local Maven caches, there are some strategies you can adopt to compensate for this. The Maven proxy caches its artifacts in the *local Maven repository* (normally in ***UserHome*/.m2/repository**). You could simply do a manual copy of the contents of the local Maven repository from one Maven proxy host to another. Or for a more sophisticated approach, you can try storing the local Maven repository on a networked file system.

20.2. HOW A MANAGED CONTAINER RESOLVES ARTIFACTS

Overview

Maven proxies play a critically important role in the way managed containers resolve Maven artifacts. When a managed container fails to locate a needed artifact locally (in its **system/** directory or in its local Maven repository) it tries to download the missing artifact from the fabric's Maven proxy server (master instance). In other words, downloading from the Maven proxy is the primary mechanism for managed containers to obtain new artifacts.

The process for resolving artifacts in a managed container is controlled by the Fabric8 agent, which detects when new artifacts need to be deployed (for example, as a result of editing a Fabric profile) and then calls into the Eclipse Aether layer to resolve the artifacts.

Fabric profiles drive bundle provisioning

In the context of Fabric, it is the Fabric profiles that drive provisioning of OSGi bundles and other resource. Provisioning is triggered whenever you edit and save properties from a current bundle—for example by adding a **bundle.*BundleName*** entry to the profile's agent properties. Provisioning can also be triggered when you edit other resources (not directly associated with OSGi Config Admin) in a profile—for example, by referencing a resource through a checksum property resolver (see ???).

In some cases, you might not want provisioning to be triggered right away. A more controlled way to roll out profile updates is to take advantage of profile versioning—see ??? for details.

Fabric8 agent

After provisioning has been triggered in a managed container, the Fabric8 agent automatically scans the changed profiles to check for any OSGi bundles or Karaf features that were added to (or deleted from) the profile. If there are any new bundles referenced using the **mvn** URL scheme, the Fabric8 agent is responsible for locating these new bundles through Maven. In the context of Fabric, the Fabric8 agent effectively plays the same role that the Pax URL Aether component plays in a standalone (non-Fabric) container.

In order to locate a Maven artifact, the Fabric8 agent parses the **mvn** URL, reads the relevant Maven configuration properties, and calls directly into the Eclipse Aether layer to resolve the referenced artifact.

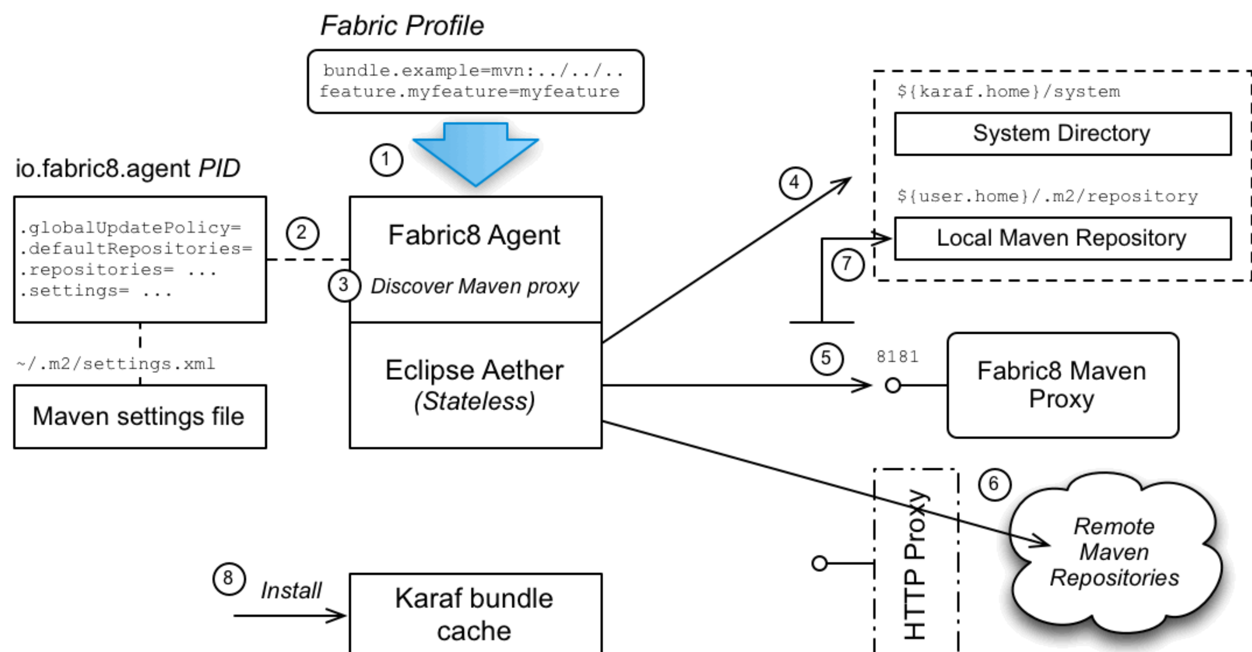
Eclipse Aether layer

The Eclipse Aether layer is fundamental to Maven artifact resolution in Apache Karaf. Ultimately, resolution of Maven artifacts for the Karaf container is *always* performed by the Aether layer. Note that the Aether layer itself is *stateless*: the parameters required to perform resolution of a Maven artifact are passed to the Aether layer with every invocation.

Provisioning a managed container

Figure 20.2, “Provisioning a Managed Container” shows an outline of the process for resolving a Maven URL at run time in a managed container.

Figure 20.2. Provisioning a Managed Container



Provisioning steps

The steps followed to locate the required Maven artifacts are:

1. Provisioning of a profile is triggered when the properties of a current profile are updated. In particular, whenever new bundles or features are added to a profile, the Fabric8 agent is responsible for resolving the new Maven artifacts (referenced through the **mvn** URL protocol).

2. The Fabric8 agent reads its Maven configuration from the **io.fabric8.agent** PID in the **default** profile (and possibly also from a Maven **settings.xml** file, if so configured).

3. The Fabric8 agent contacts Zookeeper to discover the repository URL of the Fabric8 Maven proxy (master instance)—see Section 20.1, “Cluster of Fabric Maven Proxies”. *The Fabric8 agent then inserts the discovered Maven proxy URL at the head of the list of remote Maven repositories.*

   The Fabric8 agent parses the requested Maven URL and combines this information with the specified configuration—including the discovered Maven proxy URL—in order to invoke the Eclipse Aether library.

4. When the Aether library is invoked, the first step is to look up any local Maven repositories to try and find the Maven artifact. The following local repositories are configured by default:

   ***InstallDir*/system**

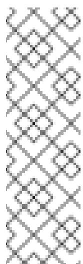
   The JBoss Fuse system directory, which contains all of the Maven artifacts that are bundled with the JBoss Fuse distribution.

   ***UserHome*/.m2/repository**

   The user's own local Maven repository in the user's home directory, ***UserHome***.

   If the Maven artifact is found locally, skip straight to step 8.

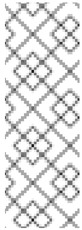
5. If the Maven artifact cannot be found in one of the local repositories, Aether next tries to download the artifact from the Maven proxy. *If the Maven artifact is found in the Maven proxy, skip straight to step 7.*

   Note

   If you configure the Fabric8 agent to use a HTTP proxy, the Maven proxy would also be accessed through the HTTP proxy. To bypass the HTTP proxy in this case, you could configure the Maven proxy host to be a HTTP non-proxy host—see the section called “Configuring a HTTP proxy”.

6. Aether next tries to look up the specified remote repositories (using the list of remote repositories specified in the Fabric8 agent configuration). Because the remote repositories are located on the Internet (accessed through the HTTP protocol), it is necessary to have Internet access in order for this step to succeed.



Note

If your local network requires you to use a HTTP proxy to access the Internet, it is possible to configure Fabric8 to use a HTTP proxy. For example, see the section called "Configuring a HTTP proxy" for details.

7. If the Maven artifact is found in the Maven proxy or in a remote repository, Aether automatically installs the artifact into the user's local Maven repository, so that another remote lookup will not be required.

8. Finally, assuming that the Maven artifact has been successfully resolved, Karaf installs the artifact in the *Karaf bundle cache*, ***InstallDir*/data/cache**, and loads the artifact (usually, an OSGi bundle) into the container runtime. At this point, the artifact is effectively installed in the container.

io.fabric8.agent configuration

The resolution of Maven artifacts in a managed container is configured by setting properties from the **io.fabric8.agent** PID (also known as *agent properties*). The Maven properties are normally set in the **default** profile, which ensures that the same settings are used throughout the entire fabric (recommended).

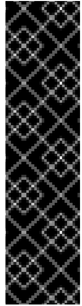
For example, you can see how the Maven properties are set in the **default** profile using the **fabric:profile-display** command, as follows:

```
JBossFuse:karaf@root> profile-display default
...
Agent Properties :
    ...
    org.ops4j.pax.url.mvn.globalUpdatePolicy = daily
    org.ops4j.pax.url.mvn.defaultRepositories =
file:${runtime.home}/${karaf.default.repository}@snapshots@id=karaf-
default,
      file:${runtime.data}/maven/upload@snapshots@id=fabric-upload,
      file:${user.home}/.m2/repository@snapshots@id=local
    ...
    org.ops4j.pax.url.mvn.globalChecksumPolicy = warn
    org.ops4j.pax.url.mvn.settings = ${karaf.etc}/maven-settings.xml
    ...
    org.ops4j.pax.url.mvn.localRepository = ${karaf.data}/repository-
agent
    ...
    org.ops4j.pax.url.mvn.repositories =
http://repo1.maven.org/maven2@id=maven.central.repo,
      https://maven.repository.redhat.com/ga@id=redhat.ga.repo,

https://maven.repository.redhat.com/earlyaccess/all@id=redhat.ea.repo,

https://repository.jboss.org/nexus/content/groups/ea@id=fuseearlyaccess
    ...
```

The properties prefixed by **org.ops4j.pax.url.mvn.*** are the Maven properties used by the Fabric8 agent.



Important

The **org.ops4j.pax.url.mvn.*** properties are *not* related to the Pax URL Aether component. There is some potential for confusion here, because the Fabric8 agent uses the same property names as Pax URL Aether. These properties are read by the Fabric8 agent, however, *not* by Pax URL Aether (and are associated with the **io.fabric8.agent** PID, not the **org.ops4j.pax.url.mvn** PID).

20.3. HOW A MAVEN PROXY RESOLVES ARTIFACTS

Overview

A Maven proxy is essentially a Web server that is configured to behave like a standard Maven repository server. Remember that the purpose of the Maven proxy is to serve artifacts to remote HTTP clients, *not* to install artifacts locally. So, although Maven proxy configuration properties have similar names to the managed container case, they ultimately serve quite a different purpose.

Fabric8 Maven proxy server

The Fabric8 Maven proxy server is a HTTP server, implemented as a servlet inside the container's default Jetty container. Hence, the Maven proxy server shares the same port number, 8181, as many of the other Karaf container services. On a given host, ***Host***, the Maven proxy can be accessed through the following URL:

```
http://Host:8181/maven/download
```

The Fabric8 Maven proxy server is configured by setting properties from the **io.fabric8.maven.proxy** PID. By default, some of these properties are set in the **default** profile and some are set in the **fabric** profile.

io.fabric8.maven bundle layer

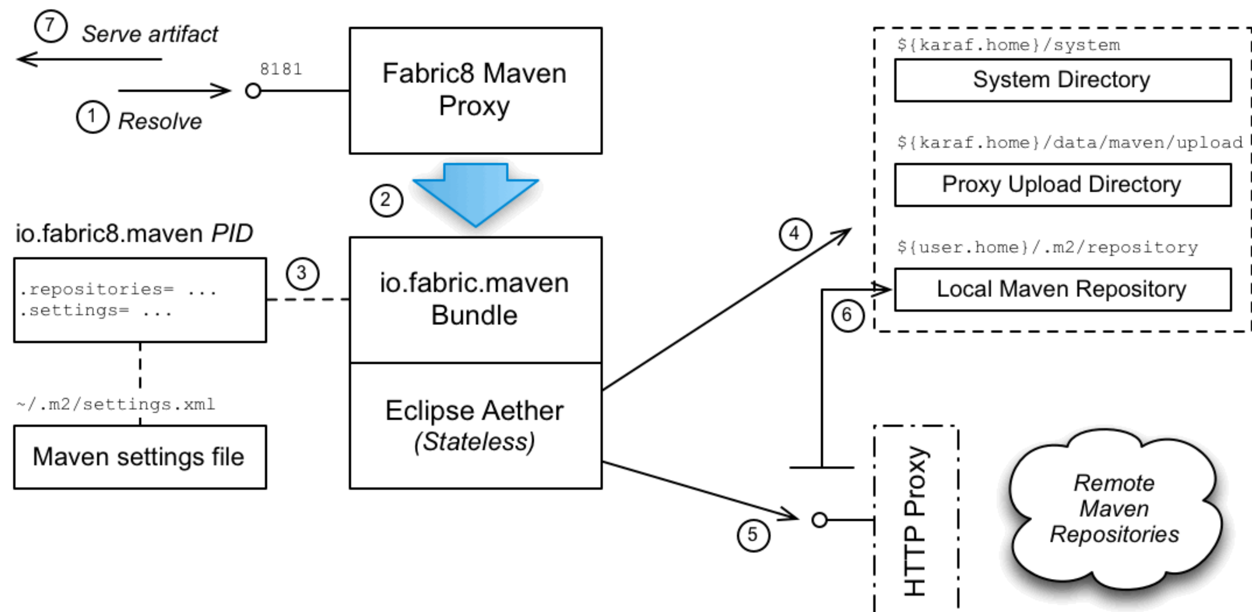
The **io.fabric8.maven** bundle layer offers similar functionality to the Pax URL Aether component (from a standalone Karaf container).

The **io.fabric8.maven** bundle is configured by setting properties from the **io.fabric8.maven** PID and these properties are normally set in the **default** profile (recommended).

Serving artifacts through the Maven proxy

Figure 20.3, “Maven Proxy Serving an Artifact” shows how a Maven proxy processes a HTTP download request, by locating the requested Maven artifact and then returning it to the client.

Figure 20.3. Maven Proxy Serving an Artifact



Steps to serve artifacts

The steps to serve the required Maven artifacts are, as follows:

1. Resolution of a Maven artifact is triggered when a managed container sends a request to the Maven proxy server.

2. The Maven proxy server parses the incoming HTTP request and then makes a call to the **`io.fabric8.maven`** layer, asking it to resolve the requested Maven artifact.

3. The **`io.fabric8.maven`** layer reads its Maven configuration from the **`io.fabric8.maven`** PID in the **`default`** profile (and possibly also from a Maven **`settings.xml`** file, if so configured).

4. When the Aether library is invoked, the first step is to look up any local Maven repositories to try and find the Maven artifact. The following local repositories are configured by default:

   ***InstallDir*/system**

   The JBoss Fuse system directory, which contains all of the Maven artifacts that are bundled with the JBoss Fuse distribution.

   ***InstallDir*/data/maven/upload**

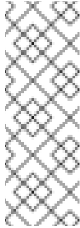
   The Maven proxy's upload directory, which is used to store artifacts that have been directly uploaded to the Maven proxy—see Section 20.6, “Automated Deployment”.

   ***UserHome*/.m2/repository**

The user's own local Maven repository in the user's home directory, ***UserHome***.

If the Maven artifact is found locally, skip straight to step 7.

5. Aether next tries to look up the specified remote repositories. Because the remote repositories are located on the Internet (accessed through the HTTP protocol), it is necessary to have Internet access in order for this step to succeed.



Note

If your local network requires you to use a HTTP proxy to access the Internet, it is possible to configure Fabric8 to use a HTTP proxy. For example, see the section called “Configuring a HTTP proxy” for details.

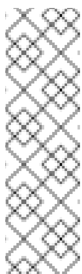
6. If the Maven artifact is found in a remote repository, Aether automatically installs the artifact into the local Maven repository, ***UserHome*/.m2/repository**, so that another remote lookup will not be required.

7. The Maven proxy server returns the successfully located Maven artifact to the client (or an error message, if the artifact could not be found).

20.4. CONFIGURING MAVEN PROXIES DIRECTLY

Overview

The default approach to configuring the Maven proxy settings is to edit the properties from the **io.fabric8.agent** PID and the **io.fabric8.maven** PID. Because these properties are set in a profile, they are immediately available to all containers in a fabric.



Note

In order to use the direct configuration approach, you must at least set the **org.ops4j.pax.url.mvn.repositories** property in the **io.fabric8.agent** PID. If this property is not set, the Fabric8 agent falls back to reading configuration from the Maven **settings.xml** file.

Tools for editing configuration

The examples in the following sections show how to modify Maven proxy configuration using Karaf console commands (for example, by invoking **fabric:profile-edit**). It is worth recalling, however, that there are several different tools you can use to modify the settings in a fabric:

- *Karaf console*—use the **fabric:*** family of commands (for example, **fabric:profile-edit**).

- *Fuse Management Console (Hawtio)*—you can edit profile settings through the **Profile** tab or the **Wiki** tab in the **Fabric** perspective of the Hawtio console, http://localhost:8181/hawtio/login.
- *Git configuration*—you can edit profile settings by cloning the Git profile repository. See ??? for details.

Rolling out configuration changes

The examples in the following sections show the form of command for editing the *current version* of the profile, which causes the changes to take effect immediately in the current fabric. If you prefer to have a more controlled rollout of configuration changes, however, you should use profile versioning to roll out the changes (see ???).

For example, instead of adding a remote repository to the current version of the **default** profile, as follows:

```
profile-edit --pid
io.fabric8.agent/org.ops4j.pax.url.mvn.repositories='http://foo/bar@id=my
foo' --append default
```

You could implement a phased rollout using versions, as follows (assuming the current version is **1.0**):

```
version-create 1.1
profile-edit --pid
io.fabric8.agent/org.ops4j.pax.url.mvn.repositories='http://foo/bar@id=my
foo' --append default 1.1
```

You can now upgrade a specific container to version **1.1**, using the following command:

```
container-upgrade 1.1 mycontainer
```

Adding a remote Maven repository

To add another remote Maven repository to the list of remote repositories used by the Maven proxy, add the relevant repository URL to the comma-separated list of repository URLs in the **org.ops4j.pax.url.mvn.repositories** property of the **io.fabric8.agent** PID in the **default** profile (not forgetting to specify the mandatory **@id** suffix in the repository URL).

For example, to add the **http://foo/bar** Maven repository to the list of remote repositories, enter the following console command:

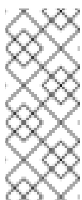
```
profile-edit --pid
io.fabric8.agent/org.ops4j.pax.url.mvn.repositories='http://foo/bar@id=my
foo' --append default
```

Note the following points about this configuration approach:

- The preceding setting simultaneously updates the **`io.fabric8.maven/io.fabric8.maven.repositories`** property (which, by default, is configured to copy the contents of the **`io.fabric8.agent/org.ops4j.pax.url.mvn.repositories`** property). This is the property that actually configures the Maven proxy.

- By editing this property in the **`default`** profile (which is normally the base profile of every profile), you ensure that this setting is propagated to all containers and to all Maven proxies in the Fabric.

- The preceding command immediately changes the configuration of all containers at the current version. If you prefer to implement a phased rollout of the new configuration, use profile versions, as described in ???.



Note

The **`@id`** option specifies the name of the repository and is *required*. You can choose an arbitrary value for this ID.

Configuring a HTTP proxy

To configure a HTTP proxy (which will be used when connecting to remote Maven repositories), set the **`io.fabric8.maven.proxies`** property of the **`io.fabric8.maven`** PID in the **`default`** profile. This property can be used to configure HTTP proxies for both the HTTP and HTTPS protocols, for example:

```
profile-edit --pid
io.fabric8.maven/io.fabric8.maven.proxies='http:host=gateway,port=8080;ht
tps:host=sslgateway,port=8453' default
```

For a detailed description of the syntax, see the section called “io.fabric8.maven PID”.

Note the following points about this configuration approach:

- The **`io.fabric8.maven.proxies`** property configures a HTTP proxy *only* for the Maven proxy servers. In particular, the Fabric8 agents are not affected by this setting, so that the Fabric8 agents continue to connect directly to the Maven proxy server on the internal network (*not* attempting to go through the HTTP proxy, which would be an error).

- Because the HTTP proxy is configured in a Fabric profile, this configuration automatically becomes available to any new container that you create in your fabric.

- If you have configured the HTTP proxy and you also want to configure the Maven proxies to connect to a Maven repository hosted on your *internal* network, you must add this internally hosted Maven repository to the list of non-proxy hosts. For details of how to do this, see the section called “io.fabric8.maven PID”.

Alternative approaches to configuring a HTTP proxy

There are some alternative approaches you can use to configure a HTTP proxy, but generally these other approaches are *not* as convenient as setting the **io.fabric8.maven.proxies** property. For example, you could take the approach of setting the **http.proxyHost** and **http.proxyPort** system properties in the ***InstallDir*/etc/system.properties** file (just like the approach for a standalone, non-Fabric container):

```
http.proxyHost=192.0.2.0
http.proxyPort=8080
```

This configuration suffers from the disadvantage that it also affects the Fabric8 agents, preventing them from accessing the Maven proxy server directly (on the internal network). In order to compensate for this, you would need to configure the list of non-proxy hosts to include the hosts where the Fabric servers (ensemble servers) are running, for example:

```
http.nonProxyHosts=ensemblehost1|ensemblehost2|ensemblehost3
```

20.5. CONFIGURING MAVEN PROXIES THROUGH SETTINGS.XML

Overview

You can optionally configure the Maven proxy using a standard Maven **settings.xml** file. For example, this approach is particularly convenient in a *development environment*, because it makes it possible to store your build time settings and your run time settings in one place.

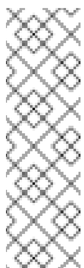
Enabling the settings.xml configuration approach

To configure Fabric to read its Maven configuration from a Maven **settings.xml** file, perform the following steps:

1. Delete the **org.ops4j.pax.url.mvn.repositories** property setting from the **io.fabric8.agent** PID in the **default** profile, using the following console command:

   ```
   profile-edit --delete --pid
   io.fabric8.agent/org.ops4j.pax.url.mvn.repositories default
   ```

   When the repositories setting is absent, Fabric implicitly switches to the **settings.xml** configuration approach.

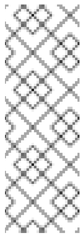
   

   Note

   This step simultaneously clears both the repositories list used by the Fabric8 agent and the repositories list used by the Maven proxy server (because the Maven proxy's repository list is normally copied straight from the Fabric8 agent's repository list).

2. Set the **io.fabric8.maven.settings** property from the **io.fabric8.maven** PID in the **default** profile to the location of the Maven **settings.xml** file. For example, if your **settings.xml** file is stored in the location, **/home/fuse/settings.xml**, you would set the **io.fabric8.maven.settings** property as follows:

```
profile-edit --pid
io.fabric8.maven/io.fabric8.maven.settings='/home/fuse/settings.xml
' default
```

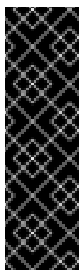


Note

The effect of this setting is to configure the Maven proxy server to takes its Maven configuration from the specified **settings.xml** file. The Fabric8 agent is *not* affected by this setting.

3. In order to configure the Fabric8 agent with the Maven **settings.xml**, set the **org.ops4j.pax.url.mvn.settings** property from the **io.fabric8.agent** PID in the **default** profile to the location of the Maven **settings.xml** file. For example, if your **settings.xml** file is stored in the location, **/home/fuse/settings.xml**, you would set the **org.ops4j.pax.url.mvn.settings** property as follows:

```
profile-edit --pid
io.fabric8.agent/org.ops4j.pax.url.mvn.settings='/home/fuse/setting
s.xml' default
```



Important

If you configure a HTTP proxy in your **settings.xml** file, it is essential to configure the ensemble hosts (where the Maven proxies are running) as HTTP non-proxy hosts. Otherwise, the Fabric8 agent tries to connect to the local Maven proxy through the HTTP proxy (which is an error).

Adding a remote Maven repository

To add a new remote Maven repository to your **settings.xml** file, open the **settings.xml** file in a text editor and add a new **repository** XML element. For example, to create an entry for the JBoss Fuse public Maven repository, add a **repository** element as shown:

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
http://maven.apache.org/xsd/settings-1.0.0.xsd">
  ...
  <profiles>
    <profile>
      <id>my-fuse-profile</id>
      <activation>
        <activeByDefault>true</activeByDefault>
      </activation>
```

```
      <repositories>
        <!--
         | Add new remote Maven repositories here
          -->
<repository>
            <id>redhat-ga-repository</id>
            <url>https://maven.repository.redhat.com/ga</url>
            <releases>
                <enabled>true</enabled>
            </releases>
            <snapshots>
                <enabled>false</enabled>
            </snapshots>
        </repository>
        ...
      </repositories>
  </profiles>
  ...
</settings>
```

The preceding example additionally specifies that release artifacts can be downloaded, but snapshot artifacts cannot be downloaded from the repository.

Configuring a HTTP proxy

To configure a HTTP proxy (which will be used when connecting to remote Maven repositories), open the **settings.xml** file in a text editor and add a new **proxy** XML element as a child of the **proxies** XML element. The definition of the proxy follows the standard Maven syntax. For example, to create a proxy for the HTTP (insecure) protocol with host, **192.0.2.0**, and port, **8080**, add a **proxy** element as follows:

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
http://maven.apache.org/xsd/settings-1.0.0.xsd">
  ...
  <proxies>
    <proxy>
      <id>fuse-proxy-1</id>
      <active>true</active>
      <protocol>http</protocol>
      <host>192.0.2.0</host>
      <port>8080</port>
      <nonProxyHosts>ensemble1|ensemble2|ensemble3</nonProxyHosts>
    </proxy>
  </proxies>
  ...
</settings>
```

You must remember to add the ensemble hosts (where the Maven proxy servers are running) to the list of HTTP non-proxy hosts in the **nonProxyHosts** element. This ensures that the Fabric8 agents do not attempt to connect to the Maven proxies through the HTTP proxy, but make a direct connection instead. In the preceding example, the ensemble host names are **ensemble1**,

ensemble2, and **ensemble3**.

You can also add a proxy for secure HTTPS connections by adding a **proxy** element configured with the **https** protocol.

Reference

For a detailed description of the syntax of the Maven **settings.xml** file, see the Maven Settings Reference. But please note that *not all of the features* documented there are necessarily supported by Fabric.

20.6. AUTOMATED DEPLOYMENT

Overview

The Maven proxy supports not just downloading artifacts, but also *uploading* artifacts. Hence, if you want to make an artifact available to all of the containers in the fabric, a simple way of doing this is to upload the artifact to the Maven proxy. For ultimate convenience in a development environment, you can automate the deployment step by installing the Fabric8 Maven plug-in in your project POM file.

Discover the upload URL of the current master

To discover the upload URL of the current Maven proxy master instance, invoke the **fabric:cluster-list** command, as follows:

```
JBossFuse:karaf@root> cluster-list servlets/io.fabric8.fabric-maven-proxy
[cluster]                           [masters]  [slaves]  [services]
[]
1.2.0.redhat-621084/maven/download
   root                             root       -
http://127.0.0.1:8181/maven/download
1.2.0.redhat-621084/maven/upload
   root                             root       -
http://127.0.0.1:8181/maven/upload
```

In this example, the upload URL of the current master is **http://127.0.0.1:8181/maven/upload**.

Manually deploy a Maven project

You can build a Maven project and upload the resulting artifact directly to the Maven proxy server, by invoking **mvn deploy** with the **altDeploymentRepository** command-line option. The value of **altDeploymentRepository** is specified in the following format:

```
ID::Layout::RepositoryURL
```

Where the format segments can be explained as follows:

ID

Can be used to pick up the relevant credentials from the **settings.xml** file (from the matching **settings/servers/server/id element**). Otherwise, the credentials must be specified in the repository URL. If necessary, you can simply specify a dummy value for the ID.

Layout

Can be either **default** (for Maven3 or Maven2) or **legacy** (for Maven1, which is not compatible with JBoss Fuse).

RepositoryURL

The Maven proxy upload URL. For example, **http://*User*:*Password*@localhost:8181/maven/upload/**.

For example, to deploy a Maven project to a Maven proxy server running on the localhost (**127.0.0.1**), authenticating with the admin/admin credentials, enter a command like the following:

```
mvn deploy -
DaltDeploymentRepository=releaseRepository::default::http://admin:admin@1
27.0.0.1:8181/maven/upload/
```

Automatically deploy a Maven project

When working in a build environment, the most convenient way to interact with the Maven proxy server is to configure the Fabric8 Maven plug-in. The Fabric8 Maven plug-in can automatically deploy your project to the local Maven proxy and, in addition, has the capability to create or update a Fabric profile for your application. For more details, see ???.

20.7. FABRIC MAVEN CONFIGURATION REFERENCE

Overview

This section provides a configuration reference for the Maven proxy configuration settings, which includes properties from the **io.fabric8.agent** PID, the **io.fabric8.maven** PID, and the **io.fabric8.maven.proxy** PID.

Repository URL syntax

You can specify a repository location using a URL with a **file:**, **http:**, or **https:** scheme, optionally appending one or more of the following suffixes:

@snapshots

Allow snapshot versions to be read from the repository.

@noreleases

Do not allow release versions to be read from the repository.

@id=*RepoName*

(Required) Specifies the repository name. This setting is required by the Aether handler.

@multi

Marks the path as a parent directory of multiple repository directories. At run time the parent directory is scanned for subdirectories and each subdirectory is used as a remote repository.

@update=*UpdatePolicy*

Specifies the Maven **updatePolicy**, overriding the value of **org.ops4j.pax.url.mvn.globalUpdatePolicy**.

@releasesUpdate=*UpdatePolicy*

Specifies the Maven **updatePolicy** specifically for release artifacts (overriding the value of **@update**).

@snapshotsUpdate=*UpdatePolicy*

Specifies the Maven **updatePolicy** specifically for snapshot artifacts (overriding the value of **@update**).

@checksum=*ChecksumPolicy*

Specifies the Maven **checksumPolicy**, which specifies how to react if a downloaded Maven artifact has a missing or incorrect checksum. The policy value can be: **ignore**, **fail**, or **warn**.

@releasesChecksum=*ChecksumPolicy*

Specifies the Maven **checksumPolicy** specifically for release artifacts (overriding the value of **@checksum**).

@snapshotsChecksum=*ChecksumPolicy*

Specifies the Maven **checksumPolicy** specifically for snapshot artifacts (overriding the value of **@checksum**).

For example:

```
https://repo.example.org/maven/repository@id=example.repo
```

io.fabric8.agent PID

The **io.fabric8.agent** PID configures the Fabric8 agent. The **io.fabric8.agent** PID supports the following properties relating specifically to Maven configuration:

org.ops4j.pax.url.mvn.defaultRepositories

Specifies a list of default (local) Maven repositories that are checked *before* looking up the remote repositories. Specified as a comma-separated list of **file:** repository URLs, where each repository URL has the syntax defined in the section called “Repository URL syntax”.

org.ops4j.pax.url.mvn.globalUpdatePolicy

Specifies the Maven **updatePolicy**, which determines how often Aether attempts to update local Maven artifacts from remote repositories. Can take the following values:

- **always**—always resolve the latest SNAPSHOT from remote Maven repositories.
- **never**—never check for newer remote SNAPSHOTS.
- **daily**—check on the first run of the day (local time).
- **interval:*Mins***—check every ***Mins*** minutes.

The **default** profile sets this property to **always**. If not set, default is **daily**.

org.ops4j.pax.url.mvn.repositories

Specifies a list of remote Maven repositories that can be searched for Maven artifacts. This property can be used in any of the following ways:

- *Use this property and disable **settings.xml***

  Normally, the **org.ops4j.pax.url.mvn.repositories** property is set as a comma-separated list of repository URLs, where the \ character can be used for line continuation. In this case, any Maven **settings.xml** file is ignored (that is, the **org.ops4j.pax.url.mvn.settings** property setting is ignored). For example, this property is set as follows in the **default** profile:

  ```
  org.ops4j.pax.url.mvn.repositories =
  http://repo1.maven.org/maven2@id=maven.central.repo,
        https://maven.repository.redhat.com/ga@id=redhat.ga.repo,
  ```

```
https://maven.repository.redhat.com/earlyaccess/all@id=redhat.ea.
repo,

https://repository.jboss.org/nexus/content/groups/ea@id=fuseearly
access
```

- *Use **settings.xml** and disable this property*

  If you want to use a Maven **settings.xml** file to configure the list of remote repositories *instead* of this property, you must remove the **org.ops4j.pax.url.mvn.repositories** property settings from the profile. For example, assuming that this property is set in the default profile, you can delete it with the following command:

  ```
  profile-edit --delete --pid
  io.fabric8.agent/org.ops4j.pax.url.mvn.repositories default
  ```

- *Use both this property* and ***settings.xml***

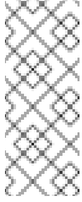
  You can combine the remote repositories specified in this setting *and* the remote repositories configured in a **settings.xml** file by using a special syntax for the list of repository URLs. In this case, you must specify a space-separated list of repository URLs, where each repository URL is prefixed by the **+** character, and the repository URLs are listed on a single line (the **** line continuation character is not supported in this syntax). For example:

  ```
  org.ops4j.pax.url.mvn.repositories =
  +file://${runtime.data}/maven/upload@snapshots@id=fabric-upload
  +file://${runtime.home}/${karaf.default.repository}@snapshots@id=
  karaf-default
  ```

org.ops4j.pax.url.mvn.settings

Specifies a path on the file system to override the default location of the Maven **settings.xml** file. The Fabric8 agent resolves the location of the Maven **settings.xml** file in the following order:

1. The location specified by **org.ops4j.pax.url.mvn.settings**.
2. **${user.home}/.m2/settings.xml**
3. **${maven.home}/conf/settings.xml**
4. ***M2_HOME*/conf/settings.xml**



Note

All **settings.xml** files are ignored, if the **org.ops4j.pax.url.mvn.repositories** property is set.

io.fabric8.maven PID

The **io.fabric8.maven** PID configures the **io.fabric8.maven** bundle (which is used by the Maven proxy server) and supports the following properties:

io.fabric8.maven.proxies

Can be used to specify a HTTP proxy, a HTTPS proxy, and non-proxy host lists. The value consists of a semicolon, **;**, separated list of entries. You can include the following entries (where each entry is optional):

- **http:host=*Host*,port=*Port*[,nonProxyHosts=*NonProxyHosts*]**
- **https:host=*Host*,port=*Port*[,nonProxyHosts=*NonProxyHosts*]**

The **nonProxyHosts** setting is optional and, if included, specifies a list of hosts that can be reached without going through the proxy. The list of non-proxy hosts has its entries separated by a pipe, **|**, symbol—for example, **nonProxyHosts=localhost|example.org**.

Here is a sample **io.fabric8.maven.proxies** setting:

```
io.fabric8.maven.proxies=
http:host=gateway,port=8080;https:host=sslgateway,port=8453
```

io.fabric8.maven.repositories

Specifies a list of remote Maven repositories that can be searched for Maven artifacts. This setting is normally copied from **org.ops4j.pax.url.mvn.repositories**.

io.fabric8.maven.useFallbackRepositories

This option is *deprecated* and should always be set to **false**.

The **default** profile sets this property to **false**.

io.fabric8.maven.proxy PID

The **io.fabric8.maven.proxy** PID configures the Fabric8 Maven proxy server and supports the following properties:

appendSystemRepos

The **fabric** profile sets this property to **false**.

role

Specifies a comma-separated list of security roles that are allowed to access the Maven proxy server. For details of role-based access control, see section "Role-Based Access Control" in "Security Guide".

The **default** profile sets this property to the following list:

```
admin,manager,viewer,Monitor,Operator,Maintainer,Deployer,Auditor,A
dministrator,SuperUser
```

updatePolicy

Specifies the Maven **updatePolicy**.

The **fabric** profile sets this property to **always**.

uploadRepository

Specifies the location of the directory used to store artifacts uploaded to the Maven proxy server.

The **fabric** profile sets this property to **${runtime.data}/maven/upload**.

CHAPTER 21. MAVEN INDEXER PLUGIN

The Maven Indexer Plugin is required for the Maven plugin to enable it to quickly search Maven Central for artifacts.

To Deploy the Maven Indexer plugin use the following commands:

Procedure 21.1. Deploy the Maven Indexer Plugin

1. In the Container perspective go to the Karaf console and enter the following command to install the Maven Indexer plugin:

```
features:install hawtio-maven-indexer
```

   In the Fabric perspective go to the Karaf console and add add the feature to a profile:

```
fabric:profile-edit --features hawtio-maven-indexer jboss-fuse-
full
```

2. For both perspectives, the rest of the commands are the same. Enter the following commands to configure the Maven Indexer plugin:

```
config:edit io.hawt.maven.indexer
config:proplist
config:propset repositories 'https://maven.oracle.com'
config:proplist
config:update
```

3. Wait for the Maven Indexer plugin to be deployed. This may take a few minutes. Look out for messages like those shown below to appear on the log tab.

INFO	org.apache.felix.fileinstall	Creating configuration from io.hawt.maven.indexer.cfg
INFO	io.fabric8.internal.ProfileServiceImpl	updateProfile: Profile[ver=1.0,id=fabric,atts={parents=karaf hawtio}]
INFO	io.fabric8.internal.ProfileServiceImpl	updateProfile: Profile[ver=1.0,id=fabric,atts={parents=karaf hawtio}]

When the Maven Indexer plugin has been deployed, use the following commands to add further external Maven repositories to the Maven Indexer plugin configuration:

```
config:edit io.hawt.maven.indexer
config:proplist
config:propset repositories external repository
config:proplist
config:update
```

CHAPTER 22. WELCOME BANNER

A banner is available on the JBoss Fuse console which can be used to display extra information. This banner is only visible when logging in using SSH.

To enable the **`welcome banner`**, edit ***`Fuse install dir`*****`/etc/org.apache.karaf.shell.cfg`**. Uncomment **`welcomBanner =`**

```
# Specify an additional welcome banner to be displayed when a user logs
into the server.
#
welcomeBanner =
```

Add your text to the welcome banner.

```
welcomeBanner = \
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ \n\
Hello and welcome to my secure server. \n\
More information here ...\n\
  ... \n\
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@\n
```

The banner will appear after the login credentials have been entered.

Figure 22.1. Log in screen for the Management Console

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
Hello and welcome to my secure server.
More information here ...
...
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

      _ ____                  ______
     | |  _ \                |  ____|
     | | |_) | ___  ___ ___  | |__ _   _ ___  ___
 _   | |  _ < / _ \/ __/ __| |  __| | | / __|/ _ \
| |__| | |_) | (_) \__ \__ \ | |  | |_| \__ \  __/
 \____/|____/ \___/|___/___/ |_|   \__,_|___/\___|

  JBoss Fuse (6.2.0.redhat-118)
  http://www.redhat.com/products/jbossenterprisemiddleware/fuse/

Hit '<tab>' for a list of available commands
and '[cmd] --help' for help on a specific command.

Open a browser to http://localhost:8181 to access the management console

Create a new Fabric via 'fabric:create'
or join an existing Fabric via 'fabric:join [someUrls]'

Hit '<ctrl-d>' or 'osgi:shutdown' to shutdown JBoss Fuse.

JBossFuse:admin@root>
```

CHAPTER 23. BRANDING JBOSS FUSE CONSOLE

When you start JBoss Fuse, you are welcomed by a default message on JBoss Fuse console. You can customize the JBoss Fuse console with your own brand. You can define your own welcome message to be displayed when you start the console and also the prompt displayed to the users. There are two ways to customize:

- Adding a **`branding.properties`** file to ***`Fuse install dir`*****`/etc`** directory

  Procedure 23.1. Adding a `branding.properties` file to *`Fuse install dir`*`/etc` directory

  - Create a **`branding.properties`** file with your message. A sample file is given below:

    ```
    {
    }
    welcome = \
    \u001B[31m      _ ____                  ______\u001B[0m\n\
    \u001B[31m     | |  _ \\                |  ____|
    \u001B[0m\n\
    \u001B[31m     | | |_) | ___  ___ ___  | |__ _   _ ___
    ___\u001B[0m\n\
    \u001B[31m _   | |  _ < / _ \\/ __/ __| |  __| | | / __|/ _
    \\\u001B[0m\n\
    \u001B[31m| |__| | |_) | (_) \\__ \\__ \\ | |  | |_| \\__ \\
    __/\u001B[0m\n\
    \u001B[31m \\____/|____/ \\___/|___/___/ |_|
    \\__,_|___/\\___|\u001B[0m\n\
    \n\
    \u001B[1m  JBoss Fuse\u001B[0m (6.2.0.redhat-133)\n\
    \n\
    Open a browser to http://localhost:8181 to access the management
    console\n\
    \n\
    Hit '<ctrl-d>' or 'osgi:shutdown' to shutdown JBoss Fuse.\n
    prompt =
    \u001B[31mJBossFuse\u001B[0m:\u001B[34m${USER}\u001B[0m\u001B[1m@
    \u001B[0m\u001B[36m${APPLICATION}\u001B[0m>\u0020
    {
    }
    ```

  - Copy the **`branding.properties`** file to ***`Fuse install dir`*****`/etc`** directory.

  - Navigate to ***`Fuse install dir`*****`/bin`** directory. Open the terminal and enter the command **`./fuse`** to start the JBoss Fuse server. You will see your branded message on the JBoss Fuse console.

- Creating a branding bundle

At the startup, JBoss Fuse is looking for a bundle which exports the **org.apache.karaf.branding** package, containing a **branding.properties** file. This branding bundle contains a file which stores your customized brand.

You can create a simple branding bundle using Maven. Copy your **branding.properties** file to the maven project resources directory, for example, **src/main/resources/org/apache/karaf/branding/** directory. Then using your project's **pom.xml** file you can generate the branding bundle as per the steps given below:

Procedure 23.2. Creating a branding bundle

- Create **branding.properties** file as shown above. Copy this file to project resources directory, for example, **src/main/resources/org/apache/karaf/branding/branding.properties** directory.

- A sample **pom.xml** file can be as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <groupId>branding.console</groupId>
    <artifactId>my.brand</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>bundle</packaging>
    <name>My Brand</name>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.felix</groupId>
                <artifactId>maven-bundle-plugin</artifactId>
                <version>2.4.0</version>
                <extensions>true</extensions>
                <configuration>
                    <instructions>
                        <Bundle-SymbolicName>manual</bundle-
SymbolicName>
                        <Import-Package>*</Import-Package>
                        <Private-Package>!*</Private-Package>
                        <Export-Package>
                            org.apache.karaf.branding
                        </Export-Package>
                        <Spring-Context>*;public-
context:=false</Spring-Context>
                    </instructions>
                </configuration>
```

```
            </plugin>
        </plugins>
    </build>
</project>
```

- Open a terminal and navigate to directory where you have saved the **pom.xml** file. Run **mvn clean install** command to create a branding bundle.
- Copy generated branded bundle from the project's **/target** directory to ***Fuse install dir*/lib** directory.
- In order for JBoss Fuse to use this branding bundle instead of default one, add the following line to ***Fuse install dir*/etc/custom.properties** file:

  ```
  org.osgi.framework.system.packages.extra = \
  org.apache.karaf.branding
  ```

- Save your changes. Navigate to ***Fuse install dir*/bin**directory and run **./fuse** to start JBoss Fuse server. You can see your branded console after the startup.

INDEX

A

admin commands, Using the admin console commands

B

broker

deploying

standalone container, Standalone containers

bundle cache, Changing the bundle cache location

C

child containers, Managing Child Containers

config shell, Standalone containers

config.properties, OSGi framework properties, Overview

config:list, Listing the current configuration

configuration

Java Options

setenv, Setting Java Options

OSGi, Introducing JBoss Fuse Configuration

F

fabric:container-stop, Shutting Down a Fabric

fabric:container-upgrade, Applying a rollup patch, Applying an incremental patch

fabric:join, Joining a Fabric

failover, Failover Deployments

featureRepositories, Modifying the default set of feature URLs

featuresBoot, Modifying the default installed features

felix.cache.bufsize, Adjusting the bundle cache buffer

felix.fileinstall.dir, Specifying the hot deployment folder

felix.fileinstall.poll, Specifying the scan interval

felix.fileinstall.tmpdir, Changing the generated-bundle cache location

G

generated bundle cache, Changing the generated-bundle cache location

H

hot deployment

folder, Specifying the hot deployment folder

monitor interval, Specifying the scan interval

J

Java Options

configuration

setenv, Setting Java Options

JDBC lock, Using a JDBC Lock System

JMX configuration

url, Changing the RMI port and JMX URL

K

karaf.default.repository, Initial container properties

karaf.framework, OSGi framework properties

karaf.framework.felix, OSGi framework properties

karaf.name, Initial container properties

L

launching

client mode, Launching the runtime in client mode

default mode, Launching the runtime in console mode

server mode, Launching the runtime in server mode

lock file, Using a Simple Lock File System

logging

commands, Log Commands

O

org.apache.felix.fileinstall-deploy, Overview

org.apache.karaf.log, Overview

org.ops4j.pax.logging, Overview

org.ops4j.pax.logging.DefaultServiceLog.level, Overview

org.osgi.framework.storage, Changing the bundle cache location

org.osgi.framework.storage.clean, Flushing the bundle cache

org.osgi.service.http.port, Initial container properties

OSGi

configuration, Introducing JBoss Fuse Configuration

OSGi configuration

creating, Standalone containers

OSGi framework

configuring, OSGi framework properties

P

patch:add, Applying a patch

patch:install, Applying a patch

patch:list, Applying a patch, Rolling back a patch

patch:rollback, Rolling back a patch

patch:simulate, Applying a patch

patching

fabric

command console, Applying a rollup patch, Applying an incremental patch

standalone, Applying a patch

rollback, Rolling back a patch

R

remote client, Using the remote client

remote console

address, Configuring a standalone container for remote access

container-connect, Using the fabric:container-connect command

ssh, Using the ssh:ssh console command

remoteShellLocation, Configuring a standalone container for remote access

RMI port, Changing the RMI port and JMX URL

RMI registry

port number, Changing the RMI port and JMX URL

rmiRegistryPort, Changing the RMI port and JMX URL

S

security, Configuring JAAS Security

serviceUrl, Changing the RMI port and JMX URL

standalone

- **initial features**, Configuring the Initial Features in a Standalone Container

starting, Starting JBoss Fuse

stopping, Stopping JBoss Fuse

- **remote container**, Stopping a Remote Container

system.properties, Initial container properties