Red Hat JBoss Fuse 6.1

Configuring and Running Red Hat JBoss Fuse

Managing the runtime container

Last Updated: 2017-10-12

Red Hat JBoss Fuse 6.1 Configuring and Running Red Hat JBoss Fuse

Managing the runtime container

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

Legal Notice

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

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 the Container".

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

The Red Hat JBoss Fuse installer generates a service wrapper that can be easily configured to install JBoss Fuse as a system service.

Installing Red Hat JBoss Fuse as a system service is a three-step process:

1. Generate the service wrapper for your system.

   See Section 2.1, “Generating the Service Wrapper”.

2. Configure the service wrapper for your system.

   See Section 2.2, “Configuring the Wrapper”.

3. Install the service wrapper as system service.

   See Section 2.3, “Installing and Starting the Service” .

2.1. GENERATING THE SERVICE WRAPPER

Abstract

The service wrapper is generated using an optional feature of the command console. Once installed the feature will generate the appropriate start-up scripts, configuration, an libraries to install a broker as a system service.

Overview

The Red Hat JBoss Fuse console's `wrapper` feature generates a wrapper around the JBoss Fuse runtime that allows you to install a message broker as a system service. The `wrapper` feature does not come preinstalled in the console, so before you can generate the service wrapper you must install the `wrapper` feature.

Once the feature is installed the console gains a `wrapper:install` command. Running this command generates a generic service wrapper in the JBoss Fuse installation.

Procedure

To generate the service wrapper:

1. Start JBoss Fuse in console mode using the `fuse` command.

2. Once the console is started and the command prompt appears, enter `features:install wrapper`.

   The `features:install` command will locate the required libraries to provision the wrapper feature and deploy it into the run time.

3. Generate the wrapper by entering `wrapper:install -n serviceName -d displayName -D description`.

   The `wrapper:install` command has the options described in Table 2.1, "Wrapper Install Options".

   Table 2.1. Wrapper Install Options

Option	Default	Description
`-s`	**AUTO_START**	Specifies the mode in which the service is installed. Valid values are **AUTO_START** or **DEMAND_START**.
`-n`	`karaf`	Specifies the service name that will be used when installing the service.
`-d`		Specifies the display name of the service.
`-D`		Specifies the description of the service.

Generated files

The following files are generated and make up the service wrapper:

- `etc\ServiceName-wrapper(.exe)`—the executable file for the wrapper.
- `bin\ServiceName-service(.bat)`—the script used to install and remove the service.
- `etc\ServiceName-wrapper.conf`—the wrapper's configuration file.
- Three library files required by the service wrapper:
  - `lib\libwrapper.so`
  - `lib\karaf-wrapper.jar`
  - `lib\karaf-wrapper-main.jar`

IMPORTANT

The only generated file you should modify is the configuration file.

2.2. CONFIGURING THE WRAPPER

Abstract

The service wrapper is configured by the ***ServiceName*`-wrapper.conf`** file, which is located under the ***InstallDir*`/etc/`** directory.

Overview

The service wrapper is configured by the ***ServiceName*`-wrapper.conf`** file, which is located under the ***InstallDir*`/etc/`** directory.

There are several kinds of setting you might want to change including:

- Environment variables
- Properties passed to the JVM
- Classpath
- JMX settings
- Logging settings

IMPORTANT

You *must* set the **JAVA_HOME** environment variable.

Specifying the Red Hat JBoss Fuse's environment

A broker's environment is controlled by the following environment variables:

- **JAVA_HOME**—the Java runtime install directory.
- **KARAF_HOME**—the location of the Red Hat JBoss Fuse install directory.
- **KARAF_BASE**—the root directory containing the configuration and OSGi data specific to the broker instance.

  The configuration for the broker instance is stored in the ***KARAF_BASE*`/conf`** directory. Other data relating to the OSGi runtime is also stored beneath the base directory.
- **KARAF_DATA**—the directory containing the logging and persistence data for the broker.

Example 2.1, “Default Environment Settings” shows the default values.

Example 2.1. Default Environment Settings

```
set.default.JAVA_HOME=JavaInstallDir
set.default.KARAF_HOME=InstallDir
set.default.KARAF_BASE=InstallDir
set.default.KARAF_DATA=InstallDir\data
```

NOTE

On Windows, you can set **JAVA_HOME** either as a system variable in the registry (for example, through the system control panel) or in ***ServiceName*-wrapper.conf**, as shown in the preceding example. Setting **JAVA_HOME** as a regular environment variable, however, does not work.

Passing parameters to the JVM

If you want to pass parameters to the JVM, you do so by setting wrapper properties using the form **wrapper.java.additional.<*n*>**. <*n*> is a sequence number that must be distinct for each parameter.

One of the most useful things you can do by passing additional parameters to the JVM is to set Java system properties. The syntax for setting a Java system property is **wrapper.java.additional.<*n*>=-D*PropName*=*PropValue***.

Example 2.2, “Default Java System Properties” shows the default Java properties.

Example 2.2. Default Java System Properties

```
# JVM
# note that n is the parameter number starting from 1.
wrapper.java.additional.1=-Dkaraf.home="%KARAF_HOME%"
wrapper.java.additional.2=-Dkaraf.base="%KARAF_BASE%"
wrapper.java.additional.3=-Dkaraf.data="%KARAF_DATA%"
wrapper.java.additional.4=-Dcom.sun.managment.jmxremote
wrapper.java.additional.5=-Dkaraf.startLocalConsole=false
wrapper.java.additional.6=-Dkaraf.startRemoteShell=true
wrapper.java.additional.7=-
Djava.endorsed.dirs="%JAVA_HOME%/jre/lib/endorsed;%JAVA_HOME%/lib/endors
ed;%KARAF_HOME%/lib/endorsed"
wrapper.java.additional.8=-
Djava.ext.dirs="%JAVA_HOME%/jre/lib/ext;%JAVA_HOME%/lib/ext;%KARAF_HOME%
/lib/ext"
```

Adding classpath entries

You add classpath entries using the syntax **wrapper.java.classpath.<*n*>**. <*n*> is a sequence number that must be distinct for each classpath entry.

Example 2.3, “Default Wrapper Classpath” shows the default classpath entries.

Example 2.3. Default Wrapper Classpath

```
wrapper.java.classpath.1=%KARAF_BASE%/lib/karaf-wrapper.jar
wrapper.java.classpath.2=%KARAF_HOME%/lib/karaf.jar
wrapper.java.classpath.3=%KARAF_HOME%/lib/karaf-jaas-boot.jar
wrapper.java.classpath.4=%KARAF_BASE%/lib/karaf-wrapper-main.jar
```

JMX configuration

The default service wrapper configuration does not enable JMX. It does, however, include template properties for enabling JMX. To enable JMX:

1. Locate the line `# Uncomment to enable jmx`.

   There are three properties, shown in Example 2.4, "Wrapper JMX Properties", that are used to configure JMX.

   Example 2.4. Wrapper JMX Properties

   ```
   # Uncomment to enable jmx
   #wrapper.java.additional.n=-
   Dcom.sun.management.jmxremote.port=1616
   #wrapper.java.additional.n=-
   Dcom.sun.management.jmxremote.authenticate=false
   #wrapper.java.additional.n=-
   Dcom.sun.management.jmxremote.ssl=false
   ```

2. Remove the # from in front of each of the properties.

3. Replace the **n** in each property to a number that fits into the sequence of addition properties established in the configuration.

You can change the settings to use a different port or secure the JMX connection.

For more information about using JMX see Chapter 13, *Configuring JMX*.

Configuring logging

The wrapper's logging in configured using the properties described in Table 2.2, "Wrapper Logging Properties".

Table 2.2. Wrapper Logging Properties

Property	Description
`wrapper.console.format`	Specifies how the logging information sent to the console is formated. The format consists of the following tokens: • **L**—log level • **P**—prefix • **D**—thread name • **T**—time • **Z**—time in milliseconds • **U**—approximate uptime in seconds (based on internal tick counter) • **M**—message

Property	Description
`wrapper.console.loglevel`	Specifies the logging level displayed on the console.
`wrapper.logfile`	Specifies the file used to store the log.
`wrapper.logfile.format`	Specifies how the logging information sent to the log file is formated.
`wrapper.console.loglevel`	Specifies the logging level sent to the log file.
`wrapper.console.maxsize`	Specifies the maximum size, in bytes, that the log file can grow to before the log is archived. The default value of 0 disables log rolling.
`wrapper.console.maxfiles`	Specifies the maximum number of archived log files which will be allowed before old files are deleted. The default value of 0 implies no limit.
`wrapper.syslog.loglevel`	Specifies the logging level for the sys/event log output.

For more information about Red Hat JBoss Fuse logging see Chapter 16, *Logging*.

2.3. INSTALLING AND STARTING THE SERVICE

Overview

The operating system determines the exact steps using to complete the installation of Red Hat JBoss Fuse as a service. The **`wrapper:install`** command provides basic instructions for your operating system.

Windows

To install the service run ***`InstallDir`*`\bin\`*`ServiceName`*`-service.bat install`**. If you used the default start setting, the service will start when Windows is launched. If you specified **`DEMAND_START`**, you will need to start the service manually.

To start the service manually run **`net start "`*`ServiceName`*`"`**. You can also use the Windows service UI.

To manually stop the service run **`net stop "`*`ServiceName`*`"`** You can also use the Windows service UI.

You remove the installed the service by running ***`InstallDir`*`\bin\`*`ServiceName`*`-service.bat remove`**.

Redhat Linux

To install the service and configure it to start when the machine boots, run the following commands:

```
# ln -s InstallDir/bin/ServiceName-service /etc/init.d/
# chkconfig ServiceName-service --add
# chkconfig ServiceName-service on
```

To start the service manually run **service *ServiceName*-service start**.

To manually stop the service run **service *ServiceName*-service stop**.

You remove the installed the service by running the following commands:

```
#service ServiceName-service stop
# chkconfig ServiceName-service --del
# rm /etc/init.d/ServiceName-service
```

Ubuntu Linux

To install the service and configure it to start when the machine boots, run the following commands:

```
# ln -s InstallDir/bin/ServiceName-service /etc/init.d/
# update-rc.d ServiceName-service defaults
```

To start the service manually run **/etc/init.d/*ServiceName*-service start**.

To manually stop the service run **/etc/init.d/*ServiceName*-service stop**.

You remove the installed the service by running the following commands:

```
#/etc/init.d/ServiceName-service stop
# rm /etc/init.d/ServiceName-service
```

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,admin
```

Where ***Username*** and ***Password*** are the new user credentials. The **admin** 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*.

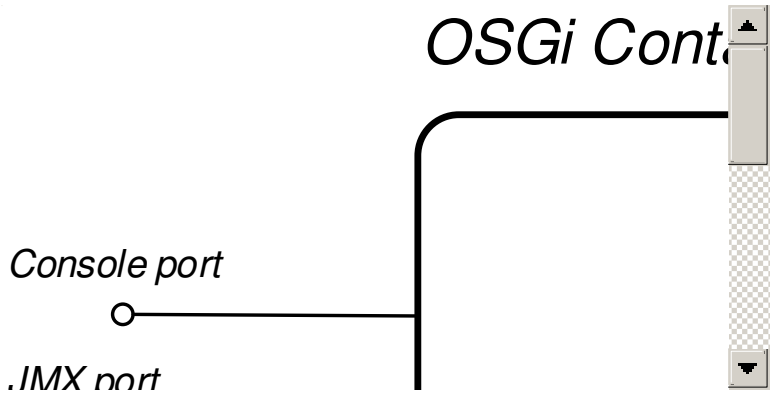
WARNING

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

Ports exposed by the Red Hat JBoss Fuse container

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

Figure 3.1. Ports Exposed by the Red Hat 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 **admin** 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:

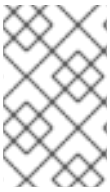
- Make sure that the JAAS user credentials have strong passwords.
- Customize the X.509 certificate (replace the Java keystore file, ***InstallDir***`/etc/host.key`, with a custom key pair).

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/activemq.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 RED HAT JBOSS FUSE

Abstract

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

4.1. STARTING RED HAT 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 from the installation directory without doing any work. 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
```

***NIX**

```
export PATH=$PATH,InstallDir/bin
```

Launching the runtime

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

Windows

***NIX**

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

```
 _ ____ _____ | | _ \ | ____| | | |_) | ___ ___ ___ | |__ _ _ ___ ___ _ |
| _ / _ \/ __/ __| | __| | | / __|/ _ \ | |__| | |_) | (_) \__ \__ \ | | |
|_| \__ \ __/ \____/|____/ \___/|___/___/ |_| \__,_|___/\___| 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:karaf@root>
```

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

***NIX**

```
bin/fuse server
```

Alternatively, you can launch JBoss Fuse in server mode using the **start** script in the ***InstallDir*/bin** directory.

Launching the runtime in client mode

In production environments you may want to have a runtime instance accessible using only a local console. In other words, *you cannot connect to the runtime remotely*. You can do this by launching the runtime in client mode using the following command:

Windows

***NIX**

4.2. STOPPING RED HAT JBOSS FUSE

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 Red Hat JBoss Fuse by running **fuse** or **fuse client**, you can stop it by doing one of the following at the **karaf>** prompt:

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

Stopping an instance running in server mode

If you launched Red Hat JBoss Fuse by running **fuse server** or by running the **start** script, you can stop it remotely, as described in Section 8.3, "Stopping a Remote Container".

Alternatively, you can log on to the host where the instance is running and run one of the following from the ***InstallDir*/bin** directory:

- **./admin stop** *instanceName*

- `./stop`

NOTE

If the sshHost property in **`etc/org.apache.karaf.shell.cfg`** is set to the default value of 0.0.0.0, you can run the **`stop`** script without any arguments. However, if you have configured a different hostname, you must run **`stop -h`** ***`hostname`***.

CHAPTER 5. CREATING A NEW FABRIC

Abstract

When there are no existing fabric's 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.

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 the **admin** role. If the ***InstallDir*/etc/users.properties** already contains users with the **admin** role, 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
```

NOTE

If you want to create a fabric on a remote container, you can log into it using the ***InstallDir*/bin/client** command-line utility.

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

```
JBossFuse:karaf@root> fabric:create --new-user AdminUser --new-user-
```

```
password AdminPass
  --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** and **--new-user-password** 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 "Description" in "Console Reference" section "Description" 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 IP port 2181 (which makes fabric configuration data available to all of the containers in the fabric).
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 "Create Fabric Profiles" in "Getting Started" , section "Description" in "Console Reference" , and section "Description" 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 IP 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 IP 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 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
```

Default settings (shown in Table 8.1) are provided for both the mac (message authentication code) and cipher (ciphers allowed for protocol version 2) properties. You can change these defaults by entering **mac = <macName1>,<macName2>,<macNameN>** and **cipher = <cipherName1>, <cipherName2>,<cipherNameN>** entries in the **etc/org.apache.karaf.shell.cfg** file.

Table 8.1. Default options for mac and cipher properties

Property	Default
mac	**hmac-sha1**
cipher	**aes256-ctr,aes192-ctr,aes128-ctr,arcfour256**

Entries in the **etc/org.apache.karaf.shell.cfg** file override the default settings, so you need to specify all options you want to use. Table 8.2, "Supported options for mac and cipher properties" shows all of the supported mac and cipher options.

For either property, you must enter multiple options in a comma-separated list that contains no white space. The order in which options appear in the list is insignificant, as the client determines which option to use.

Table 8.2. Supported options for mac and cipher properties

Property	Options
mac	**hmac-sha1**, **hmac-sha1-96**, **hmac-md5**, **hmac-md5-96**
cipher	**aes128-ctr**, **aes192-ctr**, **aes256-ctr**, **aes128-cbc**, **aes192-cbc**, **aes256-cbc**, **arcfour128**, **arcfour256**, **blowfish-cbc**, **3des-cbc**

IMPORTANT

Because of vulnerability issues, we recommend that you avoid using 96-bit and MD5-based HMAC algorithms, and use CTR, instead of CBC, mode ciphers.

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 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 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 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.

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 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 8.5. 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
```

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.

8.2.3. 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 IP 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:

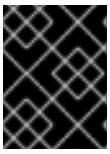
```
Username=PublicKey,Role1,Role2,...
```

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

```
ssh-rsa
AAAAB3NzaC1kc3MAAACBAP1/U4EddRIpUt9KnC7s5Of2EbdSPO9EAMMeP4C2USZpRV1AIlH7WT
2NWPq/xfW6MPbLm1Vs14E7
gB00b/JmYLdrmVClpJ+f6AR7ECLCT7up1/63xhv4O1fnfqimFQ8E+4P208UewwI1VBNaFpEy9n
Xzrith1yrv8iIDGZ3RSAHHAAAAFQCX
YFCPFSMLzLKSuYKi64QL8Fgc9QAAAnEA9+GghdabPd7LvKtcNrhXuXmUr7v6OuqC+VdMCz0Hgm
dRWVeOutRZT+ZxBxCBgLRJFnEj6Ewo
FhO3zwkyjMim4TwWeotifI0o4KOuHiuzpnWRbqN/C/ohNWLx+2J6ASQ7zKTxvqhRkImog9/hWu
WfBpKLZl6Ae1UlZAFMO/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/U4EddRIpUt9KnC7s5Of2EbdSPO9EAMMeP4C2USZpRV1AI
lH7WT2NWPq/xfW6MPbLm1Vs14E7
gB00b/JmYLdrmVClpJ+f6AR7ECLCT7up1/63xhv4O1fnfqimFQ8E+4P208UewwI1VBNaFpEy9n
Xzrith1yrv8iIDGZ3RSAHHAAAAFQCX
YFCPFSMLzLKSuYKi64QL8Fgc9QAAAnEA9+GghdabPd7LvKtcNrhXuXmUr7v6OuqC+VdMCz0Hgm
dRWVeOutRZT+ZxBxCBgLRJFnEj6Ewo
FhO3zwkyjMim4TwWeotifI0o4KOuHiuzpnWRbqN/C/ohNWLx+2J6ASQ7zKTxvqhRkImog9/hWu
WfBpKLZl6Ae1UlZAFMO/7PSSoAAACB
AKKSU2PFl/qOLxIwmBZPPIcJshVe7bVUpFvyl3BbJDow8rXfskl8wO63OzP/qLmcJM0+JbcRU/
53Jj7uyk31drV2qxhIOsLDC9dGCWj4
7Y7TyhPdXh/0dthTRBy6bqGtRPxGa7gJov1xm/UuYYXPIUR/3x9MAZvZ5xvE0kYXO+rx,admin
```



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:

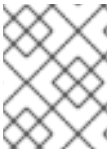
```
Index Realm                Module Class
    1 karaf
org.apache.karaf.jaas.modules.properties.PropertiesLoginModule
    2 karaf
org.apache.karaf.jaas.modules.publickey.PublickeyLoginModule
```

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 (6.0.0.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:fbolton@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 stop script

You can stop a remote container without starting up Red Hat JBoss Fuse on your local host by running the **stop(.bat)** script from the local ***InstallDir*/bin** directory.

> **Example 8.6. stop Script Syntax**
>
> **stop** [-a *port*] { -h *hostname* } { -u *username* } { -p *password* }

-a ***port***

The SSH port of the remote instance. Defaults to 8101.

-h ***hostname***

The hostname of the machine that the remote instance is running on.

-u ***username***

The username used to connect to the remote instance. Use valid JAAS login credentials that have **admin** privileges

-p ***password***

The password used to connect to the remote instance.

Using the remote console

If you have connected to a remote console using the **ssh:ssh** 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.

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 Admin Console Commands.

Creating child containers

You create a new runtime container by typing **admin:create** in the JBoss Fuse console.

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 new container is a direct copy of its parent. The only difference between parent and child is the port number they listen on. The child container is assigned an SSH port number based on an incremental count starting at 8101.

Example 9.1. Creating a Runtime Instance

```
FuseESB@root>admin:create finn
Creating new instance on port 8106 at: /home/fuse/esb4/instances/finn
Creating dir: /home/fuse/esb4/instances/finn/bin Creating dir:
/home/fuse/esb4/instances/finn/etc Creating dir:
/home/fuse/esb4/instances/finn/system Creating dir:
/home/fuse/esb4/instances/finn/deploy Creating dir:
/home/fuse/esb4/instances/finn/data Creating file:
/home/fuse/esb4/instances/finn/etc/config.properties Creating file:
/home/fuse/esb4/instances/finn/etc/java.util.logging.properties Creating
file: /home/fuse/esb4/instances/finn/etc/org.apache.felix.fileinstall-
deploy.cfg Creating file:
/home/fuse/esb4/instances/finn/etc/org.apache.karaf.log.cfg Creating
file: /home/fuse/esb4/instances/finn/etc/org.apache.karaf.features.cfg
Creating file:
/home/fuse/esb4/instances/finn/etc/org.apache.karaf.management.cfg
Creating file:
/home/fuse/esb4/instances/finn/etc/org.ops4j.pax.logging.cfg Creating
file: /home/fuse/esb4/instances/finn/etc/org.ops4j.pax.url.mvn.cfg
Creating file: /home/fuse/esb4/instances/finn/etc/startup.properties
Creating file: /home/fuse/esb4/instances/finn/etc/system.properties
Creating file:
/home/fuse/esb4/instances/finn/etc/org.apache.karaf.shell.cfg Creating
```

```
file: /home/fuse/esb4/instances/finn/bin/karaf Creating file:
/home/fuse/esb4/instances/finn/bin/start Creating file:
/home/fuse/esb4/instances/finn/bin/stop
```

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>
```

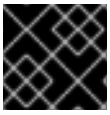
Stopping a child container

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
```

***NIX**

```
./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

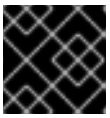
Deploying a new broker instance involves creating a new OSGi broker configuration and deploying it into the container. In a standalone container this can be done from the command console using the **`config`** command shell. For containers deployed in a fabric, you need to either create a profile for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

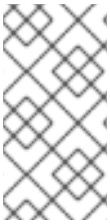
   Example 10.1. Specifying a Broker's Template XML Configuration

   ```
   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 "Details" 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/activemq.xml`**. You will also need to provide values for the data property, the broker-name property, and the openwire-port property.

FABRIC CONTAINERS

To deploy a new broker into a container in a fabric:

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 **`fabric:import`** command to upload the your XML configuration template to the Fabric Ensemble as shown in Example 10.2, "Uploading a Template to a Fabric Ensemble".

   Example 10.2. Uploading a Template to a Fabric Ensemble

   ```
   JBossFuse:karaf@root> fabric:import -t
   /fabric/configs/versions/version/profiles/mq-base/configFile
   configFile
   ```

   version must match the version of the new profile you will create for the new broker.
3. Use the **`fabric:mq-create`** command to create a profile for the new broker and assign it to a container.
   - To deploy the new broker into an existing container use the command shown in Example 10.3, "Creating a New Broker in an Existing Container"

     Example 10.3. Creating a New Broker in an Existing Container

     ```
     JBossFuse:karaf@root> fabric:mq-create --assign-container
     containerName --config configFile profileName
     ```

     This will create a new broker profile that inherits from the **`mq-base`** profile, but uses your XML configuration template, and deploy it to the specified container.
   - To deploy the new broker into an new container use the command shown in Example 10.4, "Creating a New Broker in a New Container"

     Example 10.4. Creating a New Broker in a New Container

     ```
     JBossFuse:karaf@root> fabric:mq-create --create-container
     containerName --config configFile profileName
     ```

This will create a new broker profile that inherits from the **mq-base** profile, but uses your XML configuration template, create a new container named *containerName*, and deploy the broker profile to it.

NOTE

The new container will be a child of the container from which you execute the **fabric:mq-create** command.

NOTE

You can add network configuration settings to the profile as well. See section "Arguments" in "Console Reference".

4. Use the **fabric:profile-edit** command shown in Example 10.5, “Editing a Broker Profile” to set the required properties.

Example 10.5. Editing a Broker Profile

```
JBossFuse:karaf@root> fabric:profile-edit --pid
org.fusesource.mq.fabric.server-profileName/property=value
profileName
```

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 **fabric:profile-edit** see section "Description" in "Console Reference".

NOTE

If you want to simply deploy a second broker that uses the default configuration template skip Step 1 and Step 2.

NOTE

The management console makes this process easier by providing a Web based UI.

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 org.fusesource.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/activemq.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 RED HAT 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 RED HAT 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. Red Hat JBoss Fuse Configuration Files

Filename	Description
`activemq.xml`	Configures the default Apache ActiveMQ broker in a Fabric (used in combination with the `org.fusesource.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.
`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.

Filename	Description
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 16.1, "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".
org.apache.servicemix.jbi.cfg	Configures the shutdown timeout for the JBI container.
org.apache.servicemix.nmr.cfg	Configures the default thread pool settings for JBI. See .
org.apache.servicemix.components.*Name*.cfg	Configures the thread pool settings specifically for the *Name* JBI component. See.
org.fusesource.bai.agent.cfg	Configures the Fuse BAI (Business Activity Insight) feature, if it is installed.
io.fabric8.fab.osgi.url.cfg	Configures the Maven repositories used by the Fuse Application Bundle (FAB) runtime when downloading artifacts. If the properties in this file are not set, FAB defaults to the values in **org.ops4j.pax.url.mvn.cfg**.
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.)
org.fusesource.mq.fabric.server-default.cfg	Configures the default Apache ActiveMQ broker in a Fabric (used in combination with the **activemq.xml** file).
org.jclouds.shell.cfg	Configures options for formatting the output of **jclouds:*** console commands.

Filename	Description
org.ops4j.pax.logging.cfg	Configures the logging system. For more, see Section 16.1, "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*.
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*

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.

JBI component configuration

In addition to the container's configuration files, the ***InstallDir*/etc** folder may contain a number of configuration files for the JBI components that ship with Red Hat JBoss Fuse.

The component configuration files are named using the scheme **org.apache.servicemix.components.*ComponentName*.cfg**. For example, you would configure

the JMS component using a file called `org.apache.servicemix.components.jms.cfg`.

The contents of a component's configuration file is largely component specific. However, each component configuration file contains properties for configuring the thread pool used by the component to process message exchanges. See Chapter 19, *Configuring JBI Component Thread Pools* for details.

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 =
C:\apache\apache-servicemix-6.1.0.redhat-379/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.

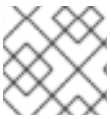
2. Remind yourself of the available properties in a particular configuration by typing **`config:proplist`**.

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

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/smx/jbideploy`** 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/smx/jbideploy
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 IP 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 two JAAS login modules, which are used in parallel:

PropertiesLoginModule

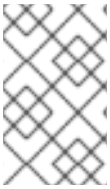
Authenticates username/password credentials and stores the secure user data in the ***InstallDir*/etc/users.properties** file.

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.

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*.

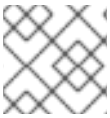
LDAP JAAS realm

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.



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
```

Both 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,Role1,Role2,...
```

Changing the default username and password

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

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

```
Username=Password,admin
```

Where the **admin** 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:

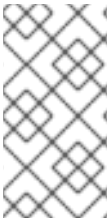
```
JBossFuse:karaf@root> jaas:update
```

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.

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.

16.1. 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
```

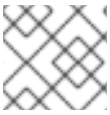
After restarting the container, you can see that each *BundleName* bundle now has its own log file, located at **`data/log/`*`BundleName`*`.log`**.

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`**.

16.2. 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
```

16.3. 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 JBI container.

THE DATA FOLDER

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

`activemq`

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. Typically these are to support deployed JBI artifacts.

`jbi`

Contains a subdirectory for each JBI artifact deployed to the JBoss Fuse runtime. For JBI components the folder's name is generated by the component's name. For JBI service assemblies, the folder's name is the identifier of the bundle generated to support the service assembly.

`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 **servicemix** 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. CONFIGURING JBI COMPONENT THREAD POOLS

Abstract

The JBI components included in Red Hat JBoss Fuse use a thread pool to process message exchanges. You can configure each component's thread pool independently.

OVERVIEW

The JBI components are multi-threaded. Each one maintains a thread pool that it uses to process message exchanges. These thread pools are configured using three properties that control the minimum number of threads in the pool, the maximum number of threads in the pool, and the depth of the component's job queue.

COMPONENT CONFIGURATION PIDS

The thread pool properties can be customised for a particular JBI component, *ComponentName*, by adding the relevant PID to the OSGi Admin service. Each JBI component has a corresponding PID that matches the pattern `org.apache.servicemix.components.`***`ComponentName`***.

The thread pool properties can also be configured using a JMX console.

IMPORTANT

The component needs to be restarted for changes to take effect.

THREAD POOL PROPERTIES

Table 19.1, “Component Thread Pool Properties” lists the properties used to configure component thread properties.

Table 19.1. Component Thread Pool Properties

Property	Default	Description
corePoolSize	8	Specifies the minimum number of threads in a thread pool. If the number of available threads drops below this limit, the runtime will always create a new thread to handle the job.
maximumPoolSize	32	Specifies the maximum number of threads in a thread pool. Setting this property to -1 specifies that it is unbounded.
queueSize	256	Specifies the number of jobs allowed in a component's job queue.

THREAD SELECTION

When a component receives a new message exchange it choose the thread to process the exchange as follows:

1. If the component's thread pool is smaller than the corePoolSize, a new thread is created to process the task.
2. If less than queueSize jobs are in the component's job queue, the task is placed on the queue to wait for a free thread.
3. If the component's job queue is full and the thread pool has less than maximumPoolSize threads instantiated, a new thread is created to process the task.
4. The job is processed by the current thread.

EXAMPLE

Example 19.1, “Component Thread Pool Configuration” shows the configuration for a component whose thread pool can have between 10 and 200 threads.

Example 19.1. Component Thread Pool Configuration

```
corePoolSize = 10
maximumPoolSize = 200
...
```

CHAPTER 20. 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.

Incremental patching allows you apply targets fixes to a container without needing to reinstall an updated version of Red Hat JBoss Fuse. It also allows you to easily back the patch out 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 patch file includes a **`.patch`** file that lists the contained artifacts. 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—the container's command console's **`patch`** shell has commands for managing the patching process
- fabric—patching a fabric requires applying the patch to a profile and then applying the profile to a container. The management console is the recommended way to patch containers in a fabric. See *Using the Management Console* for more information.

20.1. 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 which 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, 6.1.0, from the **`Version`** dropdown menu. A table of downloads now appears, which has three tabs: **`Releases`**, **`Patches`**, and **`Security Advisories`**.

NOTE

Make sure you select the right GA version for your product. A micro version release (for example, 6.1.1) is not the same thing as a patched release.

4. Click the **Patches** tab to view the regular patches (with no security-related fixes).
5. Click the **Security Advisories** tab to view the patches with security-related fixes.

TIP

To see the *complete* set of patches, you must look under both the **Patches** tab *and* the **Security Avisories** tab.

Types of patch

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

- Patches with GA baseline (for example, Patch 1, Patch 2, Patch 3, and so on)
- Rollup patches (for example, Rollup 1, Rollup 2, and so on)
- Patches with rollup baseline (for example, Rollup 1 Patch1, Rollup1 Patch2, and so on)

Patches with GA baseline

Patches with GA baseline (Patch1, Patch2, and so on) are released shortly after the GA date to provide quick fixes for issues identified after GA. These patches can be applied directly to the GA product. These patches are *cumulative*: that is, Patch 2 would contain all of the fixes from Patch 1; and Patch 3 would contain all of the fixes from Patch 1 and Patch 2; and so on.

Rollup patches

A rollup patch (Rollup 1, Rollup 2, and so on) 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.

Patches with rollup baseline

Patches with rollup baseline (Rollup 1 Patch 1, Rollup 1 Patch2, and so on) are patches released *after* a rollup patch, and they are intended to be applied on top of the corresponding rollup patch. For example, Rollup 1 Patch 2 would be applied on top of the Rollup 1 patch; and Rollup 2 Patch 1 would be applied on top of the Rollup 2 patch.

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.

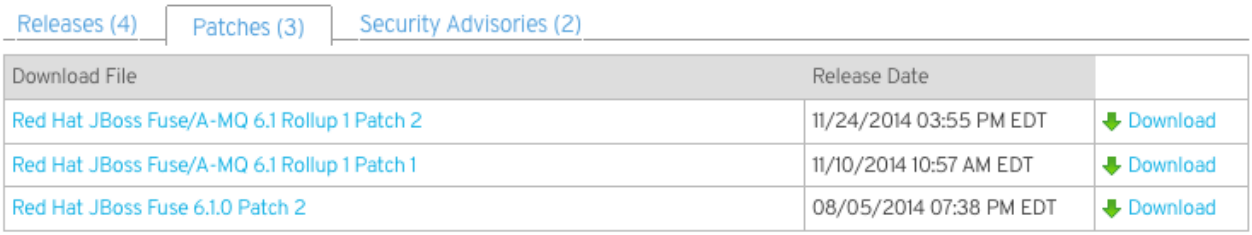
Example of identifying patches to apply

To give a concrete example of how to identify which patches to apply, we take a snapshot of the patches that were available in December 2014 and we discuss which patches you need to apply to get to the latest patch level.

Patches available under the Patches tab

The patches available under the **Patches** tab are shown in Figure 20.1, "Patches Tab".

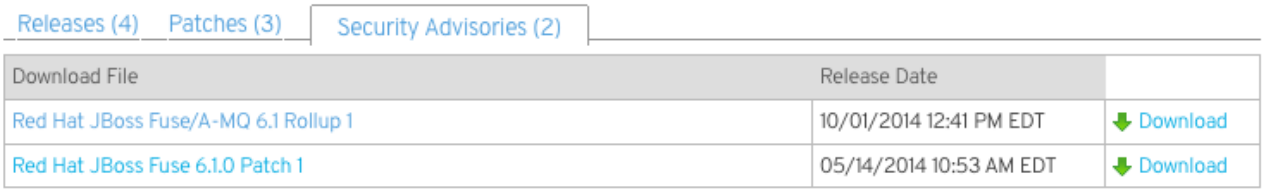
Figure 20.1. Patches Tab



Patches available under the Security Advisories tab

The patches available under the **Security Advisories** tab are shown in Figure 20.2, "Security Advisories Tab".

Figure 20.2. Security Advisories Tab



Complete list of available patches

Taking all of the patches from the **Patches** tab and the **Security Advisories** tab together, we come up with the following list of downloadable patches, in the order they were released:

- Red Hat JBoss Fuse 6.1.0 Patch 1
- Red Hat JBoss Fuse 6.1.0 Patch 2
- Red Hat JBoss Fuse/A-MQ 6.1 Rollup 1

- Red Hat JBoss Fuse/A-MQ 6.1 Rollup 1 Patch 1
- Red Hat JBoss Fuse/A-MQ 6.1 Rollup 1 Patch 2

Patches you would apply to update to the latest patch level

In this case, to update the GA product to the very latest patch level, you would apply the following sequence of patches:

1. Red Hat JBoss Fuse/A-MQ 6.1 Rollup 1
2. Red Hat JBoss Fuse/A-MQ 6.1 Rollup 1 Patch 2

20.2. PATCHING A STANDALONE CONTAINER

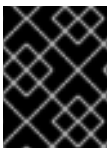
Abstract

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

Overview

Patching a standalone container directs the container to load the patch versions of artifacts instead of the non-patch versions. The **patch** shell provides commands to patch the container's environment, see which bundles are effected by applying the patch, apply the patch to the container, and back the patch out if needed (see chapter "Patch Console Commands" in "Console Reference").

To make sure that a patch can be rolled back Red Hat JBoss Fuse applies the patch in a non-destructive manner. The patching process does not overwrite the artifacts included in the original installation. The patched artifacts are placed in the container's **system** folder. When the patch is applied, the container's configuration is changed so that it points to the patched artifacts instead of the artifacts from the original installation. This makes it easy for the system to be restored to its original state or to selectively back out patches.



IMPORTANT

Patches **do not** persist across installations. If you delete and reinstall a JBoss Fuse instance you will need to download the patches and reapply them.

Applying a patch

To apply a patch to a standalone container:

1. Add the patch to the container's environment using the **patch:add** command.

   Example 20.1, “Adding a Patch to a Broker's Environment” shows the command for adding the patch contained in the patch file **patch.zip** from the local file system.

   Example 20.1. Adding a Patch to a Broker's Environment

   ```
   JBoss Fuse> patch:add file://patch.zip
   ```

This command copies the specified patch file to the container's **system** folder and unpacks it.

2. 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.

NOTE

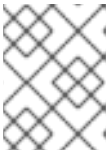
The **patch:list** command displays a list of all patches added to the container's **system** folder.

3. Review the simulation log to understand the changes that will be made to the container.

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

WARNING

Running **patch:install** before the container is fully started and all of the bundles are active will cause the container to hang.



NOTE

The **patch:list** command displays a list of all patches added to the container's **system** folder.

5. Shut down the container that you just applied the patch to.

6. The extracted patch archive contains the **manual_steps** directory. Copy the content of the **manual_steps/xyz** directory to the appropriate directory (**bin**, **etc**, **lib**) in the JBoss Fuse 6.1 installation directory. Copy the content in the **manual_steps/fabric-system-updates/system** directory to the **system** directory in the JBoss Fuse installation directory. This is the system repository that contains some patched artifacts.

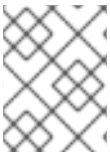
7. Start the container. If you are using a remote console, you will lose the connection to the container. If you are using the container's local console, it will automatically reconnect when the container restarts.

Rolling back a patch

Occasionally a patch will not work or introduce new issues to a container. In these cases you can easily back the patch out of the system and restore it pre-patch behavior using the **patch:rollback** command. As shown in Example 20.2, “Rolling Back a Patch” , the command takes the name of patch to be backed out.

Example 20.2. Rolling Back a Patch

```
JBoss Fuse> patch:rollback patch1
```



NOTE

The `patch:list` command displays a list of all patches added to the container's `system` folder.

The container will need to restart to roll back the patch. If you are using a remote console, you will lose the connection to the container. If you are using the container's local console, it will automatically reconnect when the container restarts.

Adding features to a 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.

20.3. PATCHING A CONTAINER IN A FABRIC

Abstract

In a fabric patches are applied to profiles and the patched version of the profile is applied to the container. The management console is the recommended tool for patching containers in a fabric. The `fabric` shell also has the commands needed to apply a patch and roll it out to running containers.

Overview

The bundles loaded by a container in a fabric are controlled by the container's Fabric Agent. The agent inspects the profiles applied to the container to determine what bundles to load, and the version of each bundle, and then loads the specified version of each bundle for the container.

A patch typically includes a new version of one or more bundles, so to apply the patch to a container in a fabric you need to update the profiles applied to it. This will cause the Fabric Agent to load the patched versions of the bundles.

The management console is the recommended tool for patching containers in a fabric. However, the command console's `fabric` shell also provides the commands needed to patch containers running in a fabric.

Procedure

Patching a container in a fabric involves:

1. Getting a patch file.
    - Customer Support sends you a patch.
    - Customer Support sends you a link to download a patch.
    - You, or your organization, generate a patch file for an internally created application.
2. Uploading one or more patch files to the fabric's Maven repository.
3. Applying the patch(es) to a profile version.

This creates a new profile version that points to the new versions of the patched bundles and repositories.

4. Migrate one or two containers to the patched profile version to ensure that the patch does not introduce any new issues.

5. After you are certain that the patch works, migrate the remaining containers in the fabric to the patched version.

Using the management console

The management console is the easiest and most verbose method of patching containers in a fabric. Its **Patching** tab uploads patches to a fabric's Maven repository and applies the patch to a specified profile version. You can then use the management console to roll the patch out to all of the containers in the fabric.

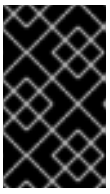
See chapter "Patching a Fabric" in "Management Console User Guide" for more information.

Using the command console

The Red Hat JBoss Fuse command console can also be used to patch containers running in a fabric. To patch a fabric container:

1. 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**).

2. 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
```

3. Upgrade the container using the **fabric:container-upgrade** command, specifying which container you want to upgrade. For example, to upgrade the **root** container, enter the following command:

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

4. 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)
```

CHAPTER 21. CONFIGURING A FABRIC'S MAVEN PROXY

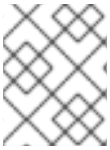
Abstract

The Fabric Ensemble creates a Maven proxy to access the repositories from which artifacts are distributed to the fabric's containers. You can modify the default settings to use a different set of repositories or the make an internal repository accessible.

OVERVIEW

The Fabric Ensemble creates a Maven proxy to facilitate access to the artifacts required by the containers in the fabric. Each Fabric Server deployed in the fabric runs an instance of a Maven proxy. The ensemble aggregates all of the proxies so that it appears to the Fabric Agents as a single Maven proxy.

The Fabric Agents use the fabric's Maven proxy to access the known repositories. This ensures that all of the containers use the same set of repositories and artifacts.



NOTE

Advanced users can configure each Fabric Server to act as a proxy for a different set of repositories. However, this is not a recommended set up.

NOTE

Red Hat JBoss Fuse Tooling provides tooling for uploading bundles using the Maven proxy. You can also add the fabric's Maven Proxy to a POM file so that bundles can be distributed to the ensemble as part of an automated build process.

DEFAULT REPOSITORIES

By default a fabric's Maven proxy is configured to be a proxy for the following Maven repositories:

- Maven Central (http://repo1.maven.org/maven2)
- Fuse Releases (http://repo.fusesource.com/nexus/content/repositories/releases)
- Fuse Early Access (http://repo.fusesource.com/nexus/content/groups/ea)
- SpringSource (http://repository.springsource.com/maven/bundles/release, http://repository.springsource.com/maven/bundles/external)
- Scala Tools (`http://scala-tools.org/repo-releases`)
- User's Local (~/.m2/repository)

CHANGING THE REPOSITORIES

To change the repositories the ensemble proxies:

1. Create a new profile version.

From the command console this is done using the **fabric:version-create** command. See section "Description" in "Console Reference" section "Description" in "Console Reference" for more information.

2. Change the **org.ops4j.pax.url.mvn.repositories** property in the **io.fabric8.agent** PID of the **default** profile. Example 21.1, “Configuring the Maven Proxy URL” shows the console command for editing this property.

Example 21.1. Configuring the Maven Proxy URL

```
JBossFuse:karaf@root> fabric:profile-edit -p
io.fabric8.agent/org.ops4j.pax.url.mvn.repositories= \
    http://repo1.maven.org/maven2, \

http://repo.fusesource.com/nexus/content/repositories/releases, \
    http://repo.fusesource.com/nexus/content/groups/ea, \
    http://repository.springsource.com/maven/bundles/release, \
    http://repository.springsource.com/maven/bundles/external, \
    http://scala-tools.org/repo-releases default
```

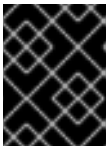
NOTE

The **io.fabric8.agent** PID is refined in all of the fabric profiles. Setting the proxy URL, the **org.ops4j.pax.url.mvn.repositories** property, in the **default** profile ensures that all of the other fabric profiles share the same Maven proxy setting.

IMPORTANT

The **fabric** profile's **io.fabric8.maven** PID, which ultimately controls the Maven proxy, imports its value from the **default** profile's **io.fabric8.agent** PID. You should **not** change the settings of the **io.fabric8.maven** PID.

3. Roll the changes out the fabric by upgrading the containers to the new profile version.



IMPORTANT

You cannot test this configuration change out on a few containers to validate it. The change **must** be made to the entire fabric or it will result in conflicts.

INDEX

I

J

K

L

M

O

S