Fuse ESB Enterprise 7.0

Getting Started

Developing and deploying projects with Fuse ESB

Edition 1

Legal Notice

1801 Varsity Drive
Raleigh, NC 27606-2072 USA
Phone: +1 919 754 3700
Phone: 888 733 4281
Fax: +1 919 754 3701

Abstract

This manual contains basic concepts for developers, and advice for creating and deploying projects with Fuse ESB.

Table of Contents

Preface

1. Document Conventions

This manual uses several conventions to highlight certain words and phrases and draw attention to specific pieces of information.

In PDF and paper editions, this manual uses typefaces drawn from the Liberation Fonts set. The Liberation Fonts set is also used in HTML editions if the set is installed on your system. If not, alternative but equivalent typefaces are displayed. Note: Red Hat Enterprise Linux 5 and later includes the Liberation Fonts set by default.

1.1. Typographic Conventions

Four typographic conventions are used to call attention to specific words and phrases. These conventions, and the circumstances they apply to, are as follows.

`Mono-spaced Bold`

Used to highlight system input, including shell commands, file names and paths. Also used to highlight keys and key combinations. For example:

> To see the contents of the file **`my_next_bestselling_novel`** in your current working directory, enter the **`cat my_next_bestselling_novel`** command at the shell prompt and press **`Enter`** to execute the command.

The above includes a file name, a shell command and a key, all presented in mono-spaced bold and all distinguishable thanks to context.

Key combinations can be distinguished from an individual key by the plus sign that connects each part of a key combination. For example:

> Press **`Enter`** to execute the command.
>
> Press **`Ctrl`**+**`Alt`**+**`F2`** to switch to a virtual terminal.

The first example highlights a particular key to press. The second example highlights a key combination: a set of three keys pressed simultaneously.

If source code is discussed, class names, methods, functions, variable names and returned values mentioned within a paragraph will be presented as above, in **`mono-spaced bold`**. For example:

> File-related classes include **`filesystem`** for file systems, **`file`** for files, and **`dir`** for directories. Each class has its own associated set of permissions.

Proportional Bold

This denotes words or phrases encountered on a system, including application names; dialog box text; labeled buttons; check-box and radio button labels; menu titles and sub-menu titles. For example:

> Choose **System** → **Preferences** → **Mouse** from the main menu bar to launch **Mouse Preferences**. In the **Buttons** tab, click the **Left-handed mouse** check box and click **Close** to switch the primary mouse button from the left to the right (making the mouse suitable for use in the left hand).
>
> To insert a special character into a **gedit** file, choose **Applications** → **Accessories** →

> **Character Map** from the main menu bar. Next, choose **Search** → **Find…** from the **Character Map** menu bar, type the name of the character in the **Search** field and click **Next**. The character you sought will be highlighted in the **Character Table**. Double-click this highlighted character to place it in the **Text to copy** field and then click the **Copy** button. Now switch back to your document and choose **Edit** → **Paste** from the **gedit** menu bar.

The above text includes application names; system-wide menu names and items; application-specific menu names; and buttons and text found within a GUI interface, all presented in proportional bold and all distinguishable by context.

Mono-spaced Bold Italic or ***Proportional Bold Italic***

Whether mono-spaced bold or proportional bold, the addition of italics indicates replaceable or variable text. Italics denotes text you do not input literally or displayed text that changes depending on circumstance. For example:

> To connect to a remote machine using ssh, type **ssh *username@domain.name*** at a shell prompt. If the remote machine is **example.com** and your username on that machine is john, type **ssh john@example.com**.
>
> The **mount -o remount *file-system*** command remounts the named file system. For example, to remount the **/home** file system, the command is **mount -o remount /home**.
>
> To see the version of a currently installed package, use the **rpm -q *package*** command. It will return a result as follows: ***package-version-release***.

Note the words in bold italics above — username, domain.name, file-system, package, version and release. Each word is a placeholder, either for text you enter when issuing a command or for text displayed by the system.

Aside from standard usage for presenting the title of a work, italics denotes the first use of a new and important term. For example:

> Publican is a *DocBook* publishing system.

1.2. Pull-quote Conventions

Terminal output and source code listings are set off visually from the surrounding text.

Output sent to a terminal is set in **mono-spaced roman** and presented thus:

```
books        Desktop   documentation  drafts  mss    photos   stuff  svn
books_tests  Desktop1  downloads      images  notes  scripts  svgs
```

Source-code listings are also set in **mono-spaced roman** but add syntax highlighting as follows:

```
package org.jboss.book.jca.ex1;

import javax.naming.InitialContext;

public class ExClient
{
   public static void main(String args[])
       throws Exception
   {
      InitialContext iniCtx = new InitialContext();
      Object         ref    = iniCtx.lookup("EchoBean");
      EchoHome       home   = (EchoHome) ref;
      Echo           echo   = home.create();

      System.out.println("Created Echo");

      System.out.println("Echo.echo('Hello') = " + echo.echo("Hello"));
   }
}
```

1.3. Notes and Warnings

Finally, we use three visual styles to draw attention to information that might otherwise be overlooked.

Note

Notes are tips, shortcuts or alternative approaches to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on a trick that makes your life easier.

Important

Important boxes detail things that are easily missed: configuration changes that only apply to the current session, or services that need restarting before an update will apply. Ignoring a box labeled 'Important' will not cause data loss but may cause irritation and frustration.

Warning

Warnings should not be ignored. Ignoring warnings will most likely cause data loss.

Chapter 1. Basic Concepts for Developers

1.1. Development Environment

1.1.1. JDK 1.6

The basic requirement for development with Fuse ESB Enterprise is the Java Development Kit (JDK) from Oracle. Generally, the recommended version is J2SE 1.6—for more platform-specific details, see the Platforms Supported page.

1.1.2. Apache Maven

The recommended build system for developing Fuse ESB Enterprise applications is Apache Maven version 3.0.x. See the *Fuse ESB Enterprise 7.0 Installation Guide* for more details.

Maven is more than a build system, however. Just as importantly, Maven also provides an infrastructure for distributing application components (typically JAR files; formally called *artifacts*). When you build an application, Maven automatically searches repositories on the Internet to find the JAR dependencies needed by your application, and then downloads the needed dependencies. See Section 1.3, “Maven Essentials” for more details.

1.1.3. Fuse IDE

Fuse IDE is an eclipse-based development tool for developing Fuse ESB Enterprise applications and is available either as a standalone binary or as an eclipse plug-in. Using Fuse IDE, you can quickly create new Fuse projects with the built-in project wizard and then edit Apache Camel routes with the drag-and-drop graphical UI. You can download Fuse IDE from the Fuse IDE download page.

Note

The trial version of Fuse IDE is free, but some of the features are available only to subscription customers.

1.1.4. Choice of development tool

You can use other tools to develop Fuse ESB Enterprise projects, but it is recommended that you use a development tool that is compatible with Maven projects (a good example is Intellij IDEA, which recognises Maven projects as a native project format).

1.2. Development Model

1.2.1. Overview

Figure 1.1, “Developing a Fuse ESB Enterprise Project” shows an overview of the development model for building an OSGi bundle or a Fuse Application Bundle that will be deployed into the Fuse ESB Enterprise container.

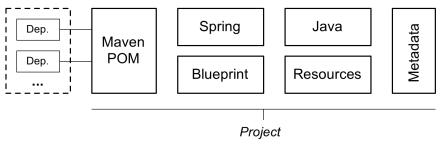

Figure 1.1. Developing a Fuse ESB Enterprise Project

1.2.2. Maven

Apache Maven, which is the recommended build system for Fuse ESB Enterprise, affects the development model in the following important ways:

- *Maven directory layout*—Maven has a standard directory layout that determines where you put your Java code, associated resources, XML configuration files, unit test code, and so on.
- *Accessing dependencies through the Internet*—Maven has the ability to download dependencies automatically through the Internet, by searching through known *Maven repositories*. This implies that you must have access to the Internet, when building with Maven. See Section 1.3.12, “Maven repositories”.

Note

For subscription customers, an offline Maven repository is also available, which makes it possible to build Maven projects without an Internet connection.

1.2.3. Maven archetypes

An easy way to get started with development is by using *Maven archetypes*, which is analogous to a new project wizard (except that it must be invoked from the command line). A Maven archetype typically creates a complete new Maven project, with the correct directory layout and some sample code. For example, see Section 2.1, “Create a Web Services Project” and Section 2.2, “Create a Router Project”.

1.2.4. Maven POM files

The Maven *Project Object Model* (POM) file, `pom.xml`, provides the description of how to build your project. The initial version of a POM is typically generated by a Maven archetype. You can then customise the POM as needed.

For larger Maven projects, there are two special kind of POM files that you might also need:

- *Aggregator POM*—a complete application is typically composed of multiple Maven projects, which must be built in a certain order. To simplify building multi-project applications, Maven enables you to define an *aggregator POM*, which can build all of the sub-projects in a single step. For more details, see Section 2.3, “Create an Aggregate Maven Project”.
- *Parent POM*—in a multi-project application, the POMs for the sub-projects typically contain a lot of the same information. Over the long term, maintaining this information, which is spread across multiple POM files, would time-consuming and error-prone. To make the POMs more manageable, you can define a parent POM, which encapsulates all of the shared information.

1.2.5. Java code and resources

Maven reserves a standard location, `src/main/java`, for your Java code, and for the associated resource files, `src/main/resources`. When Maven builds a JAR file, it automatically compiles all of the Java code and adds it to the JAR package. Likewise, all of the resource files found under `src/main/resources` are copied into the JAR package.

1.2.6. Dependency injection frameworks

Fuse ESB Enterprise has built-in support for two dependency injection frameworks: Spring XML and Blueprint XML. You can use one or the other, or both at the same time. The products underlying Fuse ESB Enterprise (that is, Apache Camel, Apache CXF, Apache ActiveMQ, and Apache ServiceMix) all strongly support XML configuration. In fact, in many cases, it is possible to develop a complete application written in XML, without any Java code whatsoever.

For more details, see Section 1.4, "Dependency Injection Frameworks".

1.2.7. Deployment metadata

Depending on how a project is packaged and deployed (as an OSGi bundle, a FAB, or a WAR), there are a few different files embedded in the JAR package that can be interpreted as deployment metadata, for example:

`META-INF/MANIFEST.MF`

The JAR manifest can be used to provide deployment metadata either for an OSGi bundle (in bundle headers) or for a FAB.

`META-INF/maven/`*`groupId`*`/`*`artifactId`*`/pom.xml`

The POM file—which is normally embedded in any Maven-built JAR file—is the main source of deployment metadata for a FAB.

`WEB-INF/web.xml`

The `web.xml` file is the standard descriptor for an application packaged as a Web ARchive (WAR).

1.2.8. Administrative metadata

The following kinds of metadata are accessible to administrators, who can use them to customize or change the behavior of bundle at run time:

- *Apache Karaf features*—a feature specifies a related collection of packages that can be deployed together. By selecting which features to install (or uninstall), an administrator can easily control which blocks of functionality are deployed in the container.
- *OSGi Config Admin properties*—the OSGi Config Admin service exposes configuration properties to the administrator at run time, making it easy to customize application behavior (for example, by customizing the IP port numbers on a server).

1.3. Maven Essentials

1.3.1. Overview

This section provides a quick introduction to some essential Maven concepts, enabling you to understand the fundamental ideas of the Maven build system.

1.3.2. Build lifecycle phases

Maven defines a standard set of phases in the build lifecycle, where the precise sequence of phases depends on what type of package you are building. For example, a JAR package includes the phases (amongst others): **compile**, **test**, **package**, and **install**.

When running Maven, you normally specify the phase as an argument to the **mvn** command, in order to indicate how far you want the build to proceed. To get started, the following are the most commonly used Maven commands:

- Build the project, run the unit tests, and install the resulting package in the local Maven repository:

  ```
  mvn install
  ```

- Clean the project (deleting temporary and intermediate files):

  ```
  mvn clean
  ```

- Build the project and run the unit tests:

  ```
  mvn test
  ```

- Build and install the project, skipping the unit tests:

  ```
  mvn install -Dmaven.test.skip=true
  ```

- Build the project in offline mode:

  ```
  mvn -o install
  ```

  Offline mode (selected by the **-o** option) is useful in cases where you know that you already have all of the required dependencies in your local repository. It prevents Maven from (unnecessarily) checking for updates to SNAPSHOT dependencies, enabling the build to proceed more quickly.

1.3.3. Maven directory structure

Example 1.1, “Standard Maven Directory Layout” shows the standard Maven directory layout. Most important is the Maven POM file, **pom.xml**, which configures the build for this Maven project.

Example 1.1. Standard Maven Directory Layout

```
ProjectDir/
    pom.xml
    src/
        main/
            java/
                ...
            resources/
                META-INF/
                    spring/
                        *.xml
                OSGI-INF/
                    blueprint/
                        *.xml
        test/
            java/
            resources/
    target/
        ...
```

The project's Java source files must be stored under ***`ProjectDir/src/main/java/`*** and any resource files should be stored under ***`ProjectDir/src/main/resources/`***. In particular, Spring XML files (matching the pattern `*.xml`) should be stored under the following directory:

```
ProjectDir/src/main/resources/META-INF/spring/
```

Blueprint XML files (matching the pattern `*.xml`) should be stored under the following directory:

```
ProjectDir/src/main/resources/OSGI-INF/blueprint/
```

1.3.4. Convention over configuration

An important principle of Maven is that of *convention over configuration*. What this means is that Maven's features and plug-ins are initialized with sensible default conventions, so that the basic functionality of Maven requires little or no configuration.

In particular, the location of the files within Maven's standard directory layout effectively determines how they are processed. For example, if you have a Maven project for building a JAR, all of the Java files under the **`src/main/java`** directory are automatically compiled and added to the JAR. All of the resource files under the **`src/main/resources`** directory are also added to the JAR.

Note

Although it is possible to alter the default Maven conventions, this practice is *strongly discouraged*. Using non-standard Maven conventions makes your projects more difficult to configure and more difficult to understand.

1.3.5. Maven packaging type

Maven defines a variety of packaging types, which determine the basic build behavior. The most common packaging types are as follows:

`jar`

(Default) This packaging type is used for Fuse Application Bundles (FABs).

`bundle`

This packaging type is used for OSGi bundles. To use this packaging type, you must also configure the **`maven-bundle-plugin`** in the POM file.

`war`

This packaging type is used for WAR files. To use this packaging type, you must also configure the **`maven-war-plugin`** in the POM file.

`pom`

When you build with this packaging type, the POM file itself gets installed into the local Maven repository. This packaging type is typically used for parent POM files.

1.3.6. Maven artifacts

The end product of a Maven build is a Maven *artifact* (for example, a JAR file). Maven artifacts are normally installed into a Maven repository, from where they can be accessed and used as building blocks for other Maven projects (by declaring them as dependencies).

1.3.7. Maven coordinates

Artifacts are uniquely identified by a tuple of Maven coordinates, usually consisting of ***`groupId`*****`:`*****`artifactId`*****`:`*****`version`***. For example, when deploying a Maven artifact into the Fuse ESB Enterprise container, you can reference it using a Maven URI of the form, **`mvn:`*****`groupId/artifactId/version`***.

For more details about Maven coordinates, see *Fuse ESB Enterprise 7.0 Deploying into the Container*: "Maven coordinates".

1.3.8. Maven dependencies

The most common modification you will need to make to your project's POM file is adding or removing Maven dependencies. A dependency is simply a reference to a Maven artifact (typically a JAR file) that is needed to build and run your project. In fact, in the context of a Maven build, managing the collection of dependencies in the POM effectively takes the place of managing the collection of JAR files in a Classpath.

The following snippet from a POM file shows how to specify a dependency on the **`camel-blueprint`** artifact:

```
<project ...>
    ...
    <dependencies>
        <dependency>
            <groupId>org.apache.camel</groupId>
            <artifactId>camel-blueprint</artifactId>
            <version>7.0.2.fuse-097</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>
    ...
</project>
```

1.3.9. dependency element

The **`dependency`** element declares a dependency on the Maven artifact with coordinates **`org.apache.camel:camel-blueprint:7.0.2.fuse-097`**. You can add as many **`dependency`** elements as you like inside the **`dependencies`** element.

1.3.10. dependency/scope element

The **`scope`** element is optional and provides some additional information about when this dependency is needed. By default (with the **`scope`** element omitted), it is assumed that the dependency is needed at build time, at unit test time, and at run time. With **`scope`** set to the value, **`provided`**, the effect depends on what kind of artifact you are building:

- *OSGi bundle*—(when the POM's `packaging` element is specified as `bundle`) the `provided` scope setting has no effect.
- *Fuse Application Bundle (FAB)*—(when the POM's `packaging` element is specified as `jar`) the `provided` scope setting implies that this dependency is deployed as a *separate* bundle in the container and is thus shared with other applications at run time.

1.3.11. Transitive dependencies

To simplify the list of dependencies in your POM and to avoid having to list every single dependency explicitly, Maven employs a recursive algorithm to figure out the dependencies needed for your project.

For example, if your project, A, depends on B1 and B2; B1 depends on C1, C2, and C3; and B2 depends on D1 and D2; Maven will automatically pull in *all* of the explicitly and implicitly required dependencies at build time, constructing a classpath that includes the dependencies, B1, B2, C1, C2, C3, D1, and D2. Of these dependencies, only B1 and B2 appear explicitly in A's POM file. The rest of the dependencies—which are figured out by Maven—are known as *transitive dependencies*.

1.3.12. Maven repositories

A Maven repository is a place where Maven can go to search for artifacts. Because Maven repositories can be anywhere—and that includes anywhere on the Internet—the Maven build system is inherently distributed. The following are the main categories of Maven repository:

- *Local repository*—the local repository (by default, located at `~/.m2/repository` on *NIX or `C:\Documents and Settings\UserName\.m2\repository` on Windows) is used by Maven as follows:
  - *First search location*—the local repository is the first place that Maven looks when searching for a dependency.
  - *Cache of downloaded dependencies*—any artifacts that have ever been downloaded from a remote repository are stored permanently in the local repository, so that they can be retrieved quickly next time they are needed.
  - *Store of locally-built artifacts*—any time that you build a local project (using `mvn install`), the resulting artifact gets stored in your local repository.
- *Remote repository*—Maven can also search for and download artifacts from remote repositories. By default, Maven automatically tries to download an artifact from remote repositories, if it cannot find the artifact in the local repository (you can suppress this behavior by specifying the `-o` flag—for example, `mvn -o install`).
- *System repository*—(Fuse ESB Enterprise container only; *not* used by the `mvn` command-line tool) at run time, the Fuse ESB Enterprise container can access artifacts from the Fuse ESB Enterprise system repository, which is located at `ESBInstallDir/system/`.

For more details about Maven repositories, see *Fuse ESB Enterprise 7.0 Deploying into the Container*: "Preparing to use Maven".

1.3.13. Specifying remote repositories

If you need to customise the remote repositories accessible to Maven, you must separately configure the build-time and runtime repository locations, as follows:

- *Build time*—to customize the remote repositories accessible at build time (when running the `mvn` command), edit the Maven `settings.xml` file, at the following location:
  - ***Nix:** default location is `~/.m2/settings.xml`.
  - **Windows:** default location is `C:\Documents and`

Settings\*UserName*\.m2\settings.xml.

- *Run time*—to customize the remote repositories accessible at run time (from within Fuse ESB Enterprise container), edit the relevant property settings in the ***ESBInstallDir*/etc/org.ops4j.pax.url.mvn.cfg**.

1.4. Dependency Injection Frameworks

1.4.1. Overview

Fuse ESB Enterprise offers a choice between the following built-in dependency injection frameworks:

- Section 1.4.4, "Spring XML".
- Section 1.4.7, "Blueprint XML".

1.4.2. Blueprint or Spring?

When trying to decide between the blueprint and Spring dependency injection frameworks, bear in mind that blueprint offers one major advantage over Spring: when new dependencies are introduced in blueprint through XML schema namespaces, blueprint has the capability to resolve these dependencies *automatically* at run time. By contrast, when packaging your project as an OSGi bundle, Spring requires you to add new dependencies explicitly to the **maven-bundle-plugin** configuration.

1.4.3. Bean registries

A fundamental capability of the dependency injection frameworks is the ability to create Java bean instances. Every Java bean created in a dependency injection framework is added to a *bean registry* by default. The bean registry is a map that enables you to look up a bean's object reference using the bean ID. This makes it possible to reference bean instances within the framework's XML configuration file and to reference bean instances from your Java code.

For example, when defining Apache Camel routes, you can use the **bean()** and **beanRef()** DSL commands to access the bean registry of the underlying dependency injection framework (or frameworks).

1.4.4. Spring XML

Spring is fundamentally a dependency injection framework, but it also includes a suite of services and APIs that enable it to act as a fully-fledged container. A Spring XML configuration file can be used in the following ways:

- *An injection framework*—Spring is a classic injection framework, enabling you to instantiate Java objects using the **bean** element and to wire beans together, either explicitly or automatically. For details, see The IoC Container from the *Spring Reference Manual*.
- *A generic XML configuration file*—Spring has an extensibility mechanism that makes it possible to use third-party XML configuration schemas in a Spring XML file. Spring uses the schema namespace as a hook for finding an extension: it searches the classpath for a JAR file that implements that particular namespace extension. In this way, it is possible to embed the following XML configurations inside a Spring XML file:
  - *Apache Camel configuration*—usually introduced by the **camelContext** element in the schema namespace, **http://camel.apache.org/schema/spring**.
  - *Apache CXF configuration*—uses several different schema namespaces, depending on whether you are configuring the Bus, **http://cxf.apache.org/core**, a JAX-WS binding, **http://cxf.apache.org/jaxws**, a JAX-RS binding, **http://cxf.apache.org/jaxrs**, or a

Simple binding, **`http://cxf.apache.org/simple`**.

- *Apache ActiveMQ configuration*—usually introduced by the **`broker`** element in the schema namespace, **`http://activemq.apache.org/schema/core`**.

Note

When packaging your project as an OSGi bundle, the Spring XML extensibility mechanism can introduce additional dependencies. Because the Maven bundle plug-in does *not* have the ability to scan the Spring XML file and automatically discover the dependencies introduced by schema namespaces, *it is generally necessary to add the additional dependencies explicitly to the **`maven-bundle-plugin`** configuration (by specifying the required Java packages)*.

- *An OSGi toolkit*—Spring also has features (provided by Spring Dynamic Modules) to simplify integrating your application with the OSGi container. In particular, Spring DM provides XML elements that make it easy to export and consume OSGi services. For details, see The Service Registry from the Spring DM *Reference Manual*.
- *A provider of container services*—Spring also supports typical container services, such as security, persistence, and transactions. Before using such services, however, you should compare what is available from the Fuse ESB Enterprise container itself. In some cases, the Fuse ESB Enterprise container already layers a service on top of Spring (as with the transaction service, for example). In other cases, the Fuse ESB Enterprise container might provide an alternative implementation of the same service.

1.4.5. Spring XML file location

In your Maven project, Spring XML files must be placed in the following location:

```
ESBInstallDir/src/main/resources/META-INF/spring/*.xml
```

1.4.6. Spring XML sample

The following example shows the bare outline of a Spring XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       >

  <camelContext xmlns="http://camel.apache.org/schema/spring">
    <!-- Define Camel routes here -->
    ...
  </camelContext>

</beans>
```

You can use a Spring XML file like this to configure Apache ActiveMQ, Apache CXF, and Apache Camel applications. For example, the preceding example includes a **`camelContext`** element, which could be used to define Apache Camel routes. For a more detailed example of Spring XML, see Section 2.1.4, “Spring XML configuration”.

1.4.7. Blueprint XML

Blueprint is a dependency injection framework defined in the OSGi specification. Historically, blueprint was originally sponsored by Spring and was based loosely on Spring DM. Consequently, the functionality offered by blueprint is quite similar to Spring XML, but blueprint is a more lightweight framework and it has been specially tailored for the OSGi container.

- *An injection framework*—blueprint is a classic injection framework, enabling you to instantiate Java objects using the **bean** element and to wire beans together, either explicitly or automatically. For details, see *Fuse ESB Enterprise 7.0 Deploying into the Container*: "The Blueprint Container".
- *A generic XML configuration file*—blueprint has an extensibility mechanism that makes it possible to use third-party XML configuration schemas in a blueprint XML file. Blueprint uses the schema namespace as a hook for finding an extension: it searches the classpath for a JAR file that implements that particular namespace extension. In this way, it is possible to embed the following XML configurations inside a blueprint XML file:
  - *Apache Camel configuration*—usually introduced by the **camelContext** element in the schema namespace, **http://camel.apache.org/schema/blueprint**.
  - *Apache CXF configuration*—uses several different schema namespaces, depending on whether you are configuring the Bus, **http://cxf.apache.org/blueprint/core**, a JAX-WS binding, **http://cxf.apache.org/blueprint/jaxws**, a JAX-RS binding, **http://cxf.apache.org/blueprint/jaxrs**, or a Simple binding, **http://cxf.apache.org/blueprint/simple**.
  - *Apache ActiveMQ configuration*—usually introduced by the **broker** element in the schema namespace, **http://activemq.apache.org/schema/core**.

  Note

  When packaging your project as an OSGi bundle, the blueprint XML extensibility mechanism can introduce additional dependencies, through the schema namespaces. *Blueprint automatically resolves the dependencies implied by the schema namespaces at run time.*

- *An OSGi toolkit*—blueprint also has features to simplify integrating your application with the OSGi container. In particular, blueprint provides XML elements that make it easy to export and consume OSGi services. For details, see *Fuse ESB Enterprise 7.0 Deploying into the Container*: "Exporting a Service" and " Importing a Service".

1.4.8. Blueprint XML file location

In your Maven project, blueprint XML files must be placed in the following location:

```
ESBInstallDir/src/main/resources/OSGI-INF/blueprint/*.xml
```

1.4.9. Blueprint XML sample

The following example shows the bare outline of a blueprint XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    >

  <camelContext xmlns="http://camel.apache.org/schema/blueprint">
    <!-- Define Camel routes here -->
    ...
  </camelContext>

</blueprint>
```

You can use a blueprint XML file like this to configure Apache ActiveMQ, Apache CXF, and Apache Camel applications. For example, the preceding example includes a `camelContext` element, which could be used to define Apache Camel routes. For a more detailed example of blueprint XML, see Example 2.3, “Configuring the Port Number in Blueprint XML”.

Note

The schema namespace used for Apache Camel in blueprint, `http://camel.apache.org/schema/blueprint`, is different from the namespace used for Apache Camel in Spring XML. The two schemas are almost identical, however.

Chapter 2. Getting Started with Developing

2.1. Create a Web Services Project

2.1.1. Overview

This section describes how to generate a simple Web services project, which includes complete demonstration code for a server and a test client. The starting point for this project is the **`servicemix-cxf-code-first-osgi-bundle`** Maven archetype, which is a command-line wizard that creates the entire project from scratch. Instructions are then given to build the project, deploy the server to the Fuse ESB Enterprise container, and run the test client.

2.1.2. Prerequisites

In order to access artifacts from the Maven repository, you need to add the **`fusesource`** repository to Maven's **`settings.xml`** file. Maven looks for your **`settings.xml`** file in the following standard location:

- **UNIX:** **`home/`*`User`*`/.m2/settings.xml`**
- **Windows:** **`Documents and Settings\`*`User`*`\.m2\settings.xml`**

If there is currently no **`settings.xml`** file at this location, you need to create a new **`settings.xml`** file. Modify the **`settings.xml`** file by adding the **`repository`** element for **`fusesource`**, as highlighted in the following example:

```
<settings>
    <profiles>
        <profile>
            <id>my-profile</id>
            <activation>
                <activeByDefault>true</activeByDefault>
            </activation>
            <repositories>
 <repository> <id>fusesource</id>
<url>http://repo.fusesource.com/nexus/content/groups/public/</url> <snapshots>
<enabled>false</enabled> </snapshots> <releases> <enabled>true</enabled>
</releases> </repository>
                ...
            </repositories>
        </profile>
    </profiles>
    ...
</settings>
```

2.1.3. Create project from the command line

You can create a Maven project directly from the command line, by invoking the **`archetype:generate`** goal. First of all, create a directory to hold your getting started projects. Open a command prompt, navigate to a convenient location in your file system, and create the **`get-started`** directory, as follows:

```
mkdir get-started
cd get-started
```

You can now use the **`archetype:generate`** goal to invoke the **`servicemix-cxf-code-first-`**

osgi-bundle archetype, which generates a simple Apache CXF demonstration, as follows:

```
mvn archetype:generate
-DarchetypeGroupId=org.apache.servicemix.tooling
-DarchetypeArtifactId=servicemix-cxf-code-first-osgi-bundle
-DarchetypeVersion=2012.01.0.fuse-70-097
-DgroupId=org.fusesource.example
-DartifactId=cxf-basic
-Dversion=1.0-SNAPSHOT
```

Note

The arguments of the preceding command are shown on separate lines for readability, but when you are actually entering the command, the entire command *must* be entered on a single line.

You will be prompted to confirm the project settings, with a message similar to this one:

```
[INFO] Using property: groupId = org.fusesource.example
[INFO] Using property: artifactId = cxf-basic
[INFO] Using property: version = 1.0-SNAPSHOT
[INFO] Using property: package = org.fusesource.example
Confirm properties configuration:
groupId: org.fusesource.example
artifactId: cxf-basic
version: 1.0-SNAPSHOT
package: org.fusesource.example
Y: :
```

Type **Return** to accept the settings and generate the project. When the command finishes, you should find a new Maven project in the **get-started/cxf-basic** directory.

2.1.4. Spring XML configuration

If you look in the **cxf-basic/src/main/resources/META-INF/spring/beans.xml** file, you can see an example of Spring XML configuration, as follows:

Example 2.1. Spring XML for Web Services Endpoint

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Generated by Apache ServiceMix Archetype -->
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jaxws="http://cxf.apache.org/jaxws"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
        http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd
        http://www.springframework.org/schema/osgi
http://www.springframework.org/schema/osgi/spring-osgi.xsd">

    <import resource="classpath:META-INF/cxf/cxf.xml" />

    <jaxws:endpoint id="HTTPEndpoint"
        implementor="org.fusesource.example.PersonImpl"
        address="/PersonServiceCF"/>

</beans>
```

The purpose of this Spring XML file is to create a WS endpoint (that is, an instance of a Web service). The `jaxws:endpoint` element creates the WS endpoint and, in this example, it requires two attributes, as follows:

`implementor`

Specifies the class that implements the Service Endpoint Interface (SEI).

`address`

Specifies the WS endpoint address. In this example, instead of a HTTP URL, the address is specified as a relative path. In the context of Fuse ESB Enterprise, this is taken to mean that the Web service should be installed into the Fuse ESB Enterprise container's default Jetty container. By default, the specified path gets prefixed by `http://localhost:8181/cxf/`, so the actual address of the Web service becomes:

```
http://localhost:8181/cxf/PersonServiceCF
```

2.1.5. Build the Web services project

Build the Web services project and install the generated JAR file into your local Maven repository. From a command prompt, enter the following commands:

```
cd cxf-basic
mvn install
```

2.1.6. Deploy and start the WS server

Start up the Fuse ESB Enterprise container. Open a new command prompt and enter the following commands:

```
cd ESBInstallDir/bin
fuseesb
```

You will see a welcome screen like the following:

```
Please wait while Fuse ESB is loading...

______                    _____  _____ ______
|  ___|                  |  ___|/  ___|| ___ \
| |_  _   _  ___    ___  | |__  \ `--. | |_/ /
|  _|| | | |/ __| / _ \ |  __|  `--. \| ___ \
| |  | |_| |\__ \|  __/ | |___ /\__/ /| |_/ /
\_|   \__,_||___/ \___| \____/ \____/ \____/

  Fuse ESB (7.0.0.fuse-032)
  http://fusesource.org/esb/

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 Fuse ESB.

FuseESB:karaf@root>
```

To install the **`cxf-basic`** Web service as an OSGi bundle, enter the following console command:

```
karaf@root> install mvn:org.fusesource.example/cxf-basic/1.0-SNAPSHOT
```

Note

If your local Maven repository is stored in a non-standard location, you might need to customize the value of the **`org.ops4j.pax.url.mvn.localRepository`** property in the ***`EsbInstallDir`*`/etc/org.ops4j.pax.url.mvn.cfg`** file, before you can use the **`mvn:`** scheme to access Maven artifacts.

If the bundle is successfully resolved and installed, the container responds by giving you the ID of the newly created bundle—for example:

```
Bundle ID: 229
```

You can now start up the Web service using the **`start`** console command, specifying the bundle ID, as follows:

```
karaf@root> start 229
```

2.1.7. Check that the bundle has started

To check that the bundle has started, enter the **`list`** console command, which gives the status of all the bundles installed in the container:

```
karaf@root> list
```

Near the end of the listing, you should see a status line like the following:

```
[ 229] [Active     ] [            ] [Started] [   60]
Apache ServiceMix :: CXF Code First OSGi Bundle (1.0.0.SNAPSHOT)
```

Note

Actually, to avoid clutter, the **list** command only shows the bundles with a start level of 50 or greater (which exludes most of the system bundles).

2.1.8. Run the WS client

The **cxf-basic** project also includes a simple WS client, which you can use to test the deployed Web service. In a command prompt, navigate to the cxf-basic directory and run the simple WS client as follows:

```
cd get-started/cxf-basic
mvn -Pclient
```

If the client runs successfully, you should see output like the following:

```
INFO: Creating Service {http://example.fusesource.org/}PersonService from class
org.fusesource.example.Person
Invoking getPerson...
getPerson._getPerson_personId=Guillaume
getPerson._getPerson_ssn=000-000-0000
getPerson._getPerson_name=Guillaume
```

2.1.9. Troubleshooting

If you have trouble running the client, there is an even simpler way to connect to the Web serivice. Open your favorite Web browser and navigate to the following URL to contact the Fuse ESB Enterprise Jetty container:

```
http://localhost:8181/cxf?wsdl
```

To query the WSDL directly from the PersonService Web service, navigate to the following URL:

```
http://localhost:8181/cxf/PersonServiceCF?wsdl
```

2.2. Create a Router Project

2.2.1. Overview

This section describes how to generate an Apache Camel router project, which acts as a proxy for the WS server described in Section 2.1, “Create a Web Services Project”. The starting point for this project is the **camel-archetype-blueprint** Maven archetype.

2.2.2. Prerequisites

This project depends on the **cxf-basic** project and requires that you have already generated and built the **cxf-basic** project, as described in Section 2.1, “Create a Web Services Project”.

2.2.3. Create project from the command line

Open a command prompt and change directory to the **get-started** directory. You can now use the **archetype:generate** goal to invoke the **camel-archetype-blueprint** archetype, which generates a simple Apache Camel demonstration, as follows:

```
mvn archetype:generate
-DarchetypeGroupId=org.apache.camel.archetypes
-DarchetypeArtifactId=camel-archetype-blueprint
-DarchetypeVersion=2.9.0.fuse-70-097
-DgroupId=org.fusesource.example
-DartifactId=camel-basic
-Dversion=1.0-SNAPSHOT
```

Note

The arguments of the preceding command are shown on separate lines for readability, but when you are actually entering the command, the entire command *must* be entered on a single line.

You will be prompted to confirm the project settings, with a message similar to this one:

```
[INFO] Using property: groupId = org.fusesource.example
[INFO] Using property: artifactId = camel-basic
[INFO] Using property: version = 1.0-SNAPSHOT
[INFO] Using property: package = org.fusesource.example
[INFO] Using property: camel-version = 2.9.0.fuse-7-032
[INFO] Using property: log4j-version = 1.2.16
[INFO] Using property: maven-bundle-plugin-version = 2.3.4
[INFO] Using property: maven-compiler-plugin-version = 2.3.2
[INFO] Using property: maven-surefire-plugin-version = 2.11
[INFO] Using property: slf4j-version = 1.6.1
Confirm properties configuration:
groupId: org.fusesource.example
artifactId: camel-basic
version: 1.0-SNAPSHOT
package: org.fusesource.example
camel-version: 2.9.0.fuse-7-032
log4j-version: 1.2.16
maven-bundle-plugin-version: 2.3.4
maven-compiler-plugin-version: 2.3.2
maven-surefire-plugin-version: 2.11
slf4j-version: 1.6.1
Y: :
```

Type **Return** to accept the settings and generate the project. When the command finishes, you should find a new Maven project in the **get-started/camel-basic** directory.

2.2.4. Modify the route

You are going to modify the default route generated by the archetype and change it into a route that implements a HTTP bridge. This bridge will be interposed between the WS client and Web service, enabling us to apply some routing logic to the WSDL messages that pass through the route.

Using your favorite text editor, open **camel-basic/src/main/resources/OSGI-INF/blueprint/blueprint.xml**. Remove the existing **bean** element and the **camelContext** element and replace them with the **camelContext** element highlighted in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:camel="http://camel.apache.org/schema/blueprint"
       xsi:schemaLocation="
       http://www.osgi.org/xmlns/blueprint/v1.0.0
http://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd
       http://camel.apache.org/schema/blueprint
http://camel.apache.org/schema/blueprint/camel-blueprint.xsd">

 <camelContext id="blueprintContext" trace="false"
xmlns="http://camel.apache.org/schema/blueprint"> <route id="httpBridge">
<from uri="jetty:http://0.0.0.0:8282/cxf/PersonServiceCF?
matchOnUriPrefix=true"/> <delay><constant>5000</constant></delay> <to
uri="jetty:http://localhost:8181/cxf/PersonServiceCF?
bridgeEndpoint=true&amp;throwExceptionOnFailure=false"/> </route>
</camelContext>

</blueprint>
```

The `from` element defines a new HTTP server port, which listens on IP port 8282. The `to` element defines a HTTP client endpoint that attempts to connect to the real Web service, which is listening on IP port 8181. To make the route a little more interesting, we add a `delay` element, which imposes a five second (5000 millisecond) delay on all requests passing through the route.

For a detailed discussion and explanation of the HTTP bridge, see *Web Services and Routing with Camel/CXF*: Proxying with HTTP".

2.2.5. Disable the test

The generated project includes a built-in unit test, which employs the `camel-test` testing toolkit. The Apache Camel testing toolkit is a useful and powerful testing library, but it will not be used in this example. To disable the test, open the `RouteTest.java` file from the `src/test/java/org/fusesource/example` directory using a text editor and look for the `@Test` annotation, as shown in the following snippet:

```
// Java
...
public class RouteTest extends CamelBlueprintTestSupport {
    ...
    @Test
    public void testRoute() throws Exception {
    ...
```

Now comment out the `@Test` annotation, as shown in the following snippet, and save the modified `RouteTest.java` file.

```
// Java
...
public class RouteTest extends CamelBlueprintTestSupport {
    ...
 // @Test // Disable test!
    public void testRoute() throws Exception {
    ...
```

2.2.6. Add the required Maven dependency

Because the route uses the Apache Camel Jetty component, you must add a Maven dependency on the **camel-jetty** artifact, so that the requisite JAR files are added to the classpath. To add the dependency, edit the **camel-basic/pom.xml** file and add the following highlighted dependency as a child of the **dependencies** element:

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
  ...
  <dependencies>
    <dependency>
      <groupId>org.apache.camel</groupId>
      <artifactId>camel-core</artifactId>
      <version>2.9.0.fuse-70-097</version>
    </dependency>
    <dependency>
      <groupId>org.apache.camel</groupId>
      <artifactId>camel-blueprint</artifactId>
      <version>2.9.0.fuse-70-097</version>
    </dependency>
 <dependency> <groupId>org.apache.camel</groupId> <artifactId>camel-
jetty</artifactId> <version>2.9.0.fuse-70-097</version> </dependency>
    ...
  </dependencies>
  ...
</project>
```

2.2.7. Build the router project

Build the router project and install the generated JAR file into your local Maven repository. From a command prompt, enter the following commands:

```
cd camel-basic
mvn install
```

2.2.8. Deploy and start the route

If you have not already started the Fuse ESB Enterprise container and deployed the Web services bundle, you should do so now—see Section 2.1.6, “Deploy and start the WS server”.

To install the **camel-basic** route as an OSGi bundle, enter the following console command:

```
karaf@root> install mvn:org.fusesource.example/camel-basic/1.0-SNAPSHOT
```

If the bundle is successfully resolved and installed, the container responds by giving you the ID of the newly created bundle—for example:

```
Bundle ID: 230
```

You can now start up the Web service using the **start** console command, specifying the bundle ID, as follows:

```
karaf@root> start 230
```

2.2.9. Test the route with the WS client

The **cxf-basic** project includes a simple WS client, which you can use to test the deployed route and Web service. In a command prompt, navigate to the **cxf-basic** directory and run the simple WS client as follows:

```
cd ../cxf-basic
mvn -Pclient -Dexec.args="http://localhost:8282/cxf/PersonServiceCF"
```

If the client runs successfully, you should see output like the following:

```
INFO: Creating Service {http://example.fusesource.org/}PersonService from class
org.fusesource.example.Person
Invoking getPerson...
```

After a five second delay, you will see the following response:

```
getPerson._getPerson_personId=Guillaume
getPerson._getPerson_ssn=000-000-0000
getPerson._getPerson_name=Guillaume
```

2.3. Create an Aggregate Maven Project

2.3.1. Aggregate POM

A complete application typically consists of multiple Maven projects. As the number of projects grows larger, however, it becomes a nuisance to build each project separately. Moreover, it is usually necessary to build the projects in a certain order and the developer must remember to observe the correct build order.

To simplify building multiple projects, you can optionally create an aggregate Maven project. This consists of a single POM file (the *aggregate POM)*, usually in the parent directory of the individual projects. The POM file specifies which sub-projects (or *modules*) to build and builds them in the specified order.

2.3.2. Parent POM

Maven also supports the notion of a *parent POM*. A parent POM enables you to define an inheritance style relationship between POMs. POM files at the bottom of the hierarchy declare that they inherit from a specific parent POM. The parent POM can then be used to share certain properties and details of configuration.

Important

The details of how to define and use a parent POM are beyond the scope of this guide, but it is important to be aware that *a parent POM and an aggregate POM are not the same thing*.

2.3.3. Best practice

Quite often, you will see examples where a POM is used *both* as a parent POM and an aggregate POM. This is acceptable for small, relatively simple applications, but is not recommended as best practice. *Best practice is to define separate POM files for the parent POM and the aggregate POM.*

2.3.4. Create an aggregate POM

To create an aggregate POM for your getting started application, use a text editor to create a **`pom.xml`** file in the **`get-started`** directory and add the following contents to the file:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://maven.apache.org/POM/4.0.0
            http://maven.apache.org/maven-v4_0_0.xsd">

  <groupId>org.fusesource.example</groupId>
  <artifactId>get-started</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>pom</packaging>
  <modelVersion>4.0.0</modelVersion>

  <name>Getting Started :: Aggregate POM</name>
  <description>Getting Started example</description>


  <modules>
    <module>cxf-basic</module>
    <module>camel-basic</module>
  </modules>


</project>
```

As with any other POM, the **`groupId`**, **`artifactId`**, and **`version`** must be defined, in order to identify this artifact uniquely. But the **`packaging`** must be set to **`pom`**. The key portion of the aggregate POM is the **`modules`** element, which defines the list of Maven sub-projects to build *and defines the order in which the projects are built*. The content of each **`module`** element is the relative path of a directory containing a Maven project.

2.3.5. Building with the aggregate POM

Using the aggregate POM you can build *all* of sub-projects in one go, by entering the following at a command prompt:

```
cd get-started
mvn install
```

2.4. Define a Feature for the Application

2.4.1. Why do you need a feature?

An OSGi bundle is *not* a convenient unit of deployment to use with the Fuse ESB Enterprise container. Applications typically consist of multiple OSGi bundles and complex applications may consist of a very large number of bundles. Usually, you want to deploy or undeploy multiple OSGi bundles at the same time and you need a deployment mechanism that supports this.

Apache Karaf features are designed to address this problem. A feature is essentially a way of aggregating multiple OSGi bundles into a single unit of deployment. When defined as a feature, you can simultaneously deploy or undeploy a whole collection of bundles.

2.4.2. What to put in a feature

At a minimum, a feature should contain the basic collection of OSGi bundles that make up your application. In addition, you might need to specify some of the dependencies of your application bundles, in case those bundles are not pre-deployed in the container.

Ultimately, the decision about what to include in your custom feature depends on what bundles and features are pre-deployed in your container. Using a standardised container like Fuse ESB Enterprise makes it easier to decide what to include in your custom feature.

Note

If you decide to use Fuse Application Bundles (FABs) instead of OSGi bundles, your feature definitions can typically be much simpler. FABs are capable of finding and installing most of the dependencies that they need.

2.4.3. Deployment options

You have a few different options for deploying features, as follows:

- *Hot deploy*—the simplest deployment option is to drop the XML features file straight into the hot deploy directory, ***`ESBInstallDir`*****`/deploy`**.
- *Add a repository URL*—you can tell the Fuse ESB Enterprise container where to find your features repository file using the **`features:addUrl`** console command (see *Fuse ESB Enterprise 7.0 Deploying into the Container*: "Add the local repository URL to the features service"). You can then install the feature at any time using the **`features:install`** console command.
- *Through a Fuse Fabric profile*—you can use the Fuse Management Console to deploy a feature inside a Fuse Fabric profile.

For more details about the feature deployment options, see *Fuse ESB Enterprise 7.0 Deploying into the Container*: "Deploying a Feature".

2.4.4. Features and Fuse Fabric

It turns out that a feature is a particularly convenient unit of deployment to use with Fuse Fabric. A Fuse Fabric profile typically consists of a list of features and a collection of related configuration settings. Hence, a Fuse Fabric profile makes it possible to deploy a completely configured application to any container in a single atomic operation.

2.4.5. Create a custom features repository

Create a sub-directory to hold the features repository. Under the **`get-started`** project directory, create all of the directories in the following path:

```
get-started/features/src/main/resources/
```

Under the **`get-started/features/src/main/resources`** directory, use a text editor to create the **`get-started.xml`** file and add the following contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<features name="get-started">
  <feature name="get-started-basic">
    <bundle>mvn:org.fusesource.example/cxf-basic/1.0-SNAPSHOT</bundle>
    <bundle>mvn:org.fusesource.example/camel-basic/1.0-SNAPSHOT</bundle>
  </feature>
  <feature name="get-started-cxf">
    <bundle>mvn:org.fusesource.example/cxf-basic/1.0-SNAPSHOT</bundle>
  </feature>
</features>
```

Under the **`get-started/features/`** directory, use a text editor to create the Maven POM file, **`pom.xml`**, and add the following contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
      http://maven.apache.org/POM/4.0.0
        http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <groupId>org.fusesource.example</groupId>
    <artifactId>get-started</artifactId>
    <packaging>jar</packaging>
    <version>1.0-SNAPSHOT</version>

    <name>Getting Started Feature Repository</name>

    <build>
      <plugins>
      <!-- Attach the generated features file as an artifact,
           and publish to the maven repository -->
        <plugin>
          <groupId>org.codehaus.mojo</groupId>
          <artifactId>build-helper-maven-plugin</artifactId>
          <version>1.5</version>
          <executions>
            <execution>
              <id>attach-artifacts</id>
              <phase>package</phase>
              <goals>
                <goal>attach-artifact</goal>
              </goals>
              <configuration>
                <artifacts>
                  <artifact>
                    <file>target/classes/get-started.xml</file>
                    <type>xml</type>
                    <classifier>features</classifier>
                  </artifact>
                </artifacts>
              </configuration>
            </execution>
          </executions>
        </plugin>
      </plugins>
    </build>

</project>
```

2.4.6. Install the features repository

You need to install the features repository into your local Maven repository, so that it can be located by the Fuse ESB Enterprise container. To install the features repository, open a command prompt, change directory to `get-started/features`, and enter the following command:

```
mvn install
```

2.4.7. Deploy the custom feature

To deploy the `get-started-basic` feature into the container, perform the following steps:

1. If the **`cxf-basic`** and **`camel-basic`** bundles are already installed in the Fuse ESB Enterprise container, you must first uninstall them. At the console prompt, use the **`list`** command to discover the bundle IDs for the **`cxf-basic`** and **`camel-basic`** bundles, and then uninstall them both using the console command, **`uninstall`** ***`BundleID`***.
2. Before you can access features from a features repository, you must tell the container where to find the features repository. Add the features repository URL to the container, by entering the following console command:

   ```
   karaf@root> features:addurl mvn:org.fusesource.example/get-started/1.0-
   SNAPSHOT/xml/features
   ```

   You can check whether the container knows about the new features by entering the console command **`features:list`**. If necessary, you can use the **`features:refreshurl`** console command, which forces the container to re-read its features repositories.
3. To install the **`get-started-basic`** feature, enter the following console command:

   ```
   karaf@root> features:install get-started-basic
   ```

4. After waiting a few seconds for the bundles to start up, you can test the application as described in Section 2.2.9, “Test the route with the WS client”.
5. To uninstall the feature, enter the following console command:

   ```
   karaf@root> features:uninstall get-started-basic
   ```

2.5. Configure the Application

2.5.1. OSGi Config Admin service

The OSGi Config Admin service is a standard OSGi configuration mechanism that enables administrators to modify application configuration at deployment time and at run time. This contrasts the with settings made directly in a Spring XML file or a blueprint XML file, because these XML files are accessible only to the developer.

The OSGi Config Admin service relies on the following basic concepts:

Persistent ID

A persistent ID (PID) identifies a group of related properties. Conventionally, a PID is normally written in the same format as a Java package name. For example, the **`org.ops4j.pax.web`** PID configures the Fuse ESB Enterprise container's default Jetty Web server.

Properties

A property is a name-value pair, which always belongs to a specific PID.

2.5.2. Setting configuration properties

There are two main ways to customise the properties in the OSGi Config Admin service, as follows:

- For a given a PID, ***`PersistentID`***, you can create a text file under the ***`ESBInstallDir/etc`*** directory, which obeys the following naming convention:

```
ESBInstallDir/etc/PersistentID.cfg
```

You can then set the properties belonging to this PID by editing this file and adding entries of the form:

```
Property=Value
```

- Fuse Fabric supports another mechanism for customising OSGi Config Admin properties. In Fuse Fabric, you set OSGi Config Admin properties in a *fabric profile* (where a profile encapsulates the data required to deploy an application). There are two alternative ways of modifying configuration settings in a profile:
  - Using the Fuse Management Console UI tool.
  - Using the **`fabric:profile-edit`** command in a container console (see Section 3.2.2, "Create Fabric Profiles").

2.5.3. Replace IP port with a property placeholder

As an example of how the OSGi Config Admin service might be used in practice, consider the IP port used by the **`PersonService`** Web service from the **`cxf-basic`** project. By modifying the Spring XML file that defines this Web service, you can make the Web service's IP port customisable through the OSGi Config Admin service.

The IP port number in the Spring XML file is replaced by a property placeholder, which resolves the port number at run time by looking up the property in the OSGi Config Admin service.

2.5.4. Spring XML example

In the **`cxf-basic`** project, any XML files from the following location are treated as Spring XML files (the standard Maven location for Spring XML files):

```
cxf-basic/src/main/resources/META-INF/spring/*.xml
```

Edit the **`beans.xml`** file from the preceding directory and add the XML contents shown in Example 2.2, "Configuring the Port Number in Spring XML".

Example 2.2. Configuring the Port Number in Spring XML

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Generated by Apache ServiceMix Archetype -->
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jaxws="http://cxf.apache.org/jaxws"
 xmlns:osgix="http://www.springframework.org/schema/osgi-compendium"
xmlns:ctx="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans.xsd
        http://cxf.apache.org/jaxws
          http://cxf.apache.org/schemas/jaxws.xsd
        http://www.springframework.org/schema/osgi
          http://www.springframework.org/schema/osgi/spring-osgi.xsd
 http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd
http://www.springframework.org/schema/osgi-compendium
http://www.springframework.org/schema/osgi-compendium/spring-osgi-
compendium.xsd">

 <!-- Configuration Admin entry --> <osgix:cm-properties id="cmProps"
persistent-id="org.fusesource.example.get.started"> <prop
key="portNumber">8181</prop> </osgix:cm-properties> <!-- placeholder
configurer --> <ctx:property-placeholder properties-ref="cmProps" />

    <jaxws:endpoint id="HTTPEndpoint"
        implementor="org.fusesource.example.PersonImpl"
        address="http://0.0.0.0:${portNumber}/PersonServiceCF"/>

</beans>
```

The highlighted text shows what is changed from the original **beans.xml** file, in order to integrate the OSGi Config Admin service. Apart from defining the **osgix** and **ctx** namespaces, the main changes are as follows:

1. The **osgix:cm-properties** bean contacts the OSGi Config Admin service and retrieves all of the property settings from the **org.fusesource.example.get.started** PID. The key-value pairs in the **prop** child elements specify default values for the properties (which are overridden, if corresponding settings can be retrieved from the OSGi Config Admin service).
2. The **ctx:property-placehoder** bean makes the properties from the **osgix:cm-properties** bean accessible as *property placeholders*. That is, a placeholder of the form **${*PropName*}** will be replaced by the value of ***PropName*** at run time.
3. The **${portNumber}** placeholder is used to specify the IP port number used by the **PersonService** Web service. Note the value of the address attribute is now specified as a *full* HTTP address (in contrast to the address shown in Example 2.1, “Spring XML for Web Services Endpoint”), which means that the WS endpoint gets deployed into a custom Jetty server (instead of the default Jetty server).

2.5.5. Blueprint XML example

To use blueprint configuration instead of Spring configuration, replace the Spring XML file by a blueprint XML file, where the blueprint file must be added to the following location in the **cxf-basic** project:

```
cxf-basic/src/main/resources/OSGI-INF/blueprint/*.xml
```

Create a **beans.xml** file in the preceding location and add the XML contents shown in Example 2.3, “Configuring the Port Number in Blueprint XML”.

Example 2.3. Configuring the Port Number in Blueprint XML

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xmlns:jaxws="http://cxf.apache.org/blueprint/jaxws"
           xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-
cm/v1.0.0">

 <!-- osgi blueprint property placeholder --> <cm:property-placeholder
id="placeholder" persistent-id="org.fusesource.example.get.started">
<cm:default-properties> <cm:property name="portNumber" value="8181"/>
</cm:default-properties> </cm:property-placeholder>

    <import resource="classpath:META-INF/cxf/cxf.xml" />

    <jaxws:endpoint id="HTTPEndpoint"
        implementor="org.fusesource.example.PersonImpl"
        address="http://0.0.0.0:{{portNumber}}/PersonServiceCF"/>

</blueprint>
```

The highlighted text shows the parts of the blueprint configuration that are relevant to the OSGi Config Admin service. Apart from defining the `cm` namespace, the main changes are as follows:

1. The `cm:property-placeholder` bean contacts the OSGi Config Admin service and retrieves all of the property settings from the `org.fusesource.example.get.started` PID. The key-value pairs in the `cm:default-properties/cm:property` elements specify default values for the properties (which are overridden, if corresponding settings can be retrieved from the OSGi Config Admin service).
2. The `{{portNumber}}` placeholder is used to specify the IP port number used by the `PersonService` Web service.

Note

If you want to try out the blueprint XML configuration, you must ensure that the instructions for the `maven-bundle-plugin` in the project's `pom.xml` file include the wildcard, `*`, in the packages listed in the `Import-Package` element (if the `Import-Package` element is not present, the wildcard is implied by default). Otherwise, you will get the error: `Unresolved references to [org.osgi.service.blueprint] by class(es) on the Bundle-Classpath[Jar:dot]: []`.

2.5.6. Deploying the configurable application

To deploy the configurable Web service from the `cxf-basic` project, perform the following steps:

1. Edit the Spring XML file, **beans.xml**, to integrate the OSGi Config Admin service, as described in Example 2.2, “Configuring the Port Number in Spring XML”.
2. Rebuild the **cxf-basic** project with Maven. Open a command prompt, change directory to the **get-started/cxf-basic** directory, and enter the following Maven command:

   ```
   mvn clean install
   ```

3. Create the following configuration file in the **etc/** directory of your Fuse ESB Enterprise installation:

   ```
   ESBInstallDir/etc/org.fusesource.example.get.started.cfg
   ```

   Edit the **org.fusesource.example.get.started.cfg** file with a text editor and add the following contents:

   ```
   portNumber=8182
   ```

4. If you have previously deployed the **get-started-basic** feature (as described in Section 2.4, “Define a Feature for the Application”), uninstall it now:

   ```
   karaf@root> features:uninstall get-started-basic
   ```

5. Deploy the **get-started-cxf** feature, by entering the following console command:

   ```
   karaf@root> features:install get-started-cxf
   ```

6. After waiting a few seconds for the bundles to start up, you can test the application by opening a command prompt, changing directory to **get-started/cxf-basic**, and entering the following command:

   ```
   mvn -Pclient -Dexec.args="http://localhost:8182/PersonServiceCF"
   ```

   Important

   The URL in this command has a slightly different format from the URLs used in the previous client commands: the path part of the URL is **/PersonServiceCF**, instead of **/cxf/PersonServiceCF**.

7. To uninstall the feature, enter the following console command:

   ```
   features:uninstall get-started-cxf
   ```

2.6. Troubleshooting

2.6.1. Check the status of a deployed bundle

After deploying an OSGi bundle, you can check its status using the **osgi:list** console command. For example:

```
karaf@root> osgi:list
```

The most recently deployed bundles appear at the bottom of the listing. For example, a successfully deployed **`cxf-basic`** bundle has a status line like the following:

```
[ 232] [Active     ] [            ] [Started] [   60]
    Apache ServiceMix :: CXF Code First OSGi Bundle (1.0.0.SNAPSHOT)
```

The second column indicates the status of the OSGi bundle lifecycle (usually **`Installed`**, **`Resolved`**, or **`Active`**). A bundle that is successfully installed and started has the status **`Active`**. If the bundle contains a blueprint XML file, the third column indicates whether the blueprint context has been successfully **`Created`** or not. If the bundle contains a Spring XML file, the fourth column indicates whether the Spring context has been successfully **`Started`** or not.

2.6.2. Logging

If a bundle fails to start up properly, an error message is usually sent to the log. To view the most recent messages from the log, enter the **`log:display`** console command. Usually, you will be able to find a stack trace for the failed bundle in the log.

You can easily change the logging level using the **`log:set`** console command. For example:

```
karaf@root> log:set DEBUG
```

2.6.3. Redeploying bundles with dev:watch

If there is an error in one of your bundles and you need to redeploy it, the best approach is to use the **`dev:watch`** command. For example, given that you have already deployed the **`cxf-basic`** bundle and it has the bundle ID, 232, you can tell the runtime to watch the bundle by entering the following console command:

```
karaf@root> dev:watch 232
Watched URLs/IDs:
232
```

Now, whenever you rebuild the bundle using Maven:

```
cd cxf-basic
mvn clean install
```

The runtime automatically redeploys the bundle, as soon as it notices that the corresponding JAR in the local Maven repository has been updated. In the console window, the following message appears:

```
[Watch] Updating watched bundle: cxf-basic (1.0.0.SNAPSHOT)
```

Chapter 3. Getting Started with Deploying

3.1. Scalable Deployment with Fuse Fabric

3.1.1. Why Fuse Fabric?

A single Fuse ESB Enterprise container deployed on one host provides a flexible and sophisticated environment for deploying your applications, with support for versioning, deployment of various package types (OSGi bundle, FAB, WAR), container services and so on. But when you start to roll out a large-scale deployment of a product based on Fuse ESB Enterprise, where multiple containers are deployed on multiple hosts across a network, you are faced with a whole new set of challenges. Some of the capabilities you typically need for managing a large-scale deployment are, as follows:

- Monitoring the state of all the containers in the network.
- Starting and stopping remote containers.
- Provisioning remote containers to run particular applications.
- Upgrading applications and rolling out patches in a live system.
- Starting up and provisioning new containers quickly—for example, to cope with an increased load on the system.

The Fuse Fabric technology layer has been developed specifically to handle these kinds of challenges in a large-scale production system.

3.1.2. A sample fabric

Figure 3.1, “Containers in a Fabric” shows an example of a distributed collection of containers that belong to a single fabric.

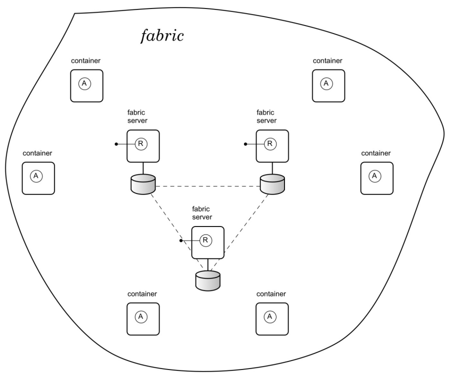

Figure 3.1. Containers in a Fabric

3.1.3. Fabric

Fuse Fabric is a technology layer that supports the scalable deployment of Fuse ESB Enterprise containers across a network. It enables a variety of advanced features, such as remote installation and provisioning of containers; phased rollout of new versions of libraries and applications; load-balancing and failover of deployed endpoints.

A *fabric* is a collection of containers that share a *Fabric Registry*, where the Fabric Registry is a replicated database that stores all information related to provisioning and managing the containers. A fabric is intended to manage a distributed network of containers, where the containers are deployed across multiple hosts.

3.1.4. Fabric Ensemble

A *Fabric Ensemble* is a collection of Fabric Servers that collectively maintain the state of the Fabric Registry. The Fabric Ensemble implements a replicated database and uses a quorum-based voting system to ensure that data in the Fabric Registry remains consistent across all of the Fabric Servers. To guard against network splits in a quorum-based system, it is a requirement that *the number of Fabric Servers in a Fabric Ensemble is always an odd number*.

The number of Fabric Servers in a fabric is typically 1, 3, or 5. A fabric with just one Fabric Server is suitable for experimentation only. A live production system should have at least 3 or 5 Fabric Servers, installed on separate hosts, to provide fault tolerance.

3.1.5. Fabric Server

An Fabric Server has a special status in the fabric, because it is responsible for maintaining a replica of the Fabric Registry. In each Fabric Server, a registry service is installed (shown as R in Figure 3.1, “Containers in a Fabric”). The registry service (based on Apache ZooKeeper) maintains a replica of the registry database and provides a ZooKeeper server, which ordinary agents can connect to in order to retrieve registry data.

3.1.6. Fabric Container

A Fabric Container is aware of the locations of all of the Fabric Servers and is able to retrieve registry data from any of the containers in the Fabric Ensemble. In each Fabric Container, a *Fabric Agent* is installed (shown as A in Figure 3.1, “Containers in a Fabric”). The Fabric Agent actively monitors the fabric registry and, whenever a relevant modification is made to the registry, it immediately updates itself to be consistent with the registry settings.

3.1.7. Profile

A *fabric profile* is an abstract unit of deployment, which is capable of holding all of the data required for deploying an application into a Fabric Container. Profiles are the basic unit of deployment in the context of fabric, and they must be used instead of directly deployed features or bundles.

Important

The presence of a Fabric Agent in a container completely changes the deployment model, *requiring you to use profiles exclusively* as the unit of deployment. Although it is still possible to deploy an individual bundle or feature (using `osgi:install` or `features:install`, respectively), these modifications will not be permanent. As soon as you restart the container or refresh its contents, the Fabric Agent will wipe the existing contents of the container and replace its contents with whatever is specified by the deployed profiles.

3.2. Deploying to a Fabric

3.2.1. Create a Fabric

3.2.1.1. Overview

Figure 3.2, “A Sample Fabric with Child Containers” shows an overview of the sample fabric that will be created. The Fabric Ensemble consists of just one Fabric Server (so this fabric is suitable only for experimental use) and two ordinary child containers.

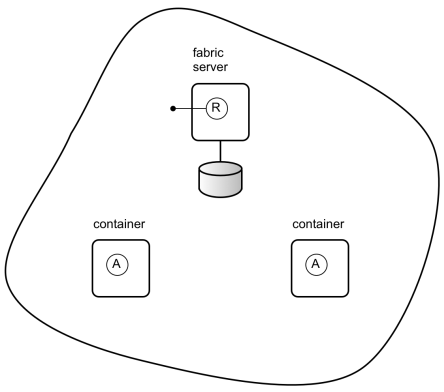

Figure 3.2. A Sample Fabric with Child Containers

3.2.1.2. Fabric Server

An Fabric Server (or servers) forms the backbone of a fabric. It hosts a fabric registry agent, which maintains a replicable database of information about the state of the fabric. Initially, when we create the fabric, there is just a single Fabric Server.

3.2.1.3. Child containers

The simplest way to extend a fabric is to create one or more child containers. As shown in Figure 3.2, "A Sample Fabric with Child Containers", the first container in the fabric is a root container and the child containers are all descended from this root container.

Each child container is an independent Fuse ESB Enterprise container instance, which runs in its own JVM instance. The data files for the child containers are stored under the ***ESBInstallDir*/instances** directory.

3.2.1.4. Steps to create the fabric

To create the basic fabric shown in Figure 3.2, "A Sample Fabric with Child Containers", perform the following steps:

1. *(Optional)* Customise the name of the root container. Edit the ***ESBInstallDir*/etc/system.properties** file and customise the following setting:

   ```
   karaf.name=root
   ```

   Note

   For the first container in your fabric, this step is optional. But if at some later stage you want to join a root container to the fabric, it becomes essential to customise the new container name so that it does not clash with any existing root containers in the fabric.

2. To create the first Fabric Server, which acts as the seed for the new fabric, enter the following console command:

   ```
   karaf@root> fabric:create --clean
   ```

   The current container, named **root** by default, becomes an Fabric Server, with a fabric registry agent installed. Initially, this is the only container in the fabric.

3. Create two new child containers. Assuming that your root container is named **root**, enter the following console command:

```
karaf@root> fabric:container-create-child root child 2
```

4. Invoke the **fabric:container-list** command to see a listing of all the containers in your new fabric. You should see a listing like the following:

```
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
```

3.2.1.5. Shutting down the containers

Because the child containers run in their own JVMs, they are *not* automatically stopped when you shut down the root container. To shut down a container and its children, you must first stop its children using the **fabric:container-stop** command. For example, to shut down the current fabric completely, enter the following console commands:

```
karaf@root> fabric:container-stop child1
karaf@root> fabric:container-stop child2
karaf@root> shutdown -f
```

When you restart the root container, you must then restart the children explicitly using the **fabric:container-start** console command.

3.2.2. Create Fabric Profiles

3.2.2.1. Overview

A profile is the basic unit of deployment in a fabric. You can deploy one or more profiles to a container and the content of those deployed profiles determines exactly what is installed in the container.

3.2.2.2. Contents of a profile

A profile encapsulates the following kinds of information:

- The URL locations of features repositories.
- A list of features to install.
- A list of bundles to install (or, more generally, any suitable JAR package—including OSGi bundles, Fuse Application Bundles, and WAR files).
- A collection of configuration settings for the OSGi Config Admin service.
- Java system properties that affect the Apache Karaf container (analogous to editing **etc/config.properties**).
- Java system properties that affect installed bundles (analogous to editing **etc/system.properties**).

3.2.2.3. Base profile

Profiles support inheritance. This can be useful in cases where you want to deploy a cluster of similar servers—for example, where the servers differ only in the choice of IP port number. In this, you would typically define a *base profile*, which defines all of the deployment data that the servers have in common. Each individual server profile would then inherit from the common base profile and add the specific configuration settings required for that server instance.

3.2.2.4. Create a base profile

To create the **gs-cxf-base** profile, perform the following steps:

1. Create the **gs-cxf-base** profile. Enter the following console command:

   ```
   karaf@root> fabric:profile-create --parents cxf gs-cxf-base
   ```

2. Add the **get-started** features repository (see Section 2.4, “Define a Feature for the Application”) to the **gs-cxf-base** profile. Enter the following console command:

   ```
   karaf@root> profile-edit -r mvn:org.fusesource.example/get-started/1.0-
   SNAPSHOT/xml/features gs-cxf-base
   ```

3. Now add the **get-started-cxf** feature (which provides the Web service example server) to the **gs-cxf-base** profile. Enter the following console command:

   ```
   karaf@root> profile-edit --features get-started-cxf gs-cxf-base
   ```

3.2.2.5. Create the derived profiles

We create two derived profiles, **gs-cxf-01** and **gs-cxf-02**, which configure different IP ports for the Web service. To create the derived profiles, perform the following steps:

1. Create the **gs-cxf-01** profile—which derives from **gs-cxf-base**—by entering the following console command:

   ```
   karaf@root> profile-create --parents gs-cxf-base gs-cxf-01
   ```

2. Create the **gs-cxf-02** profile—which derives from **gs-cxf-base**—by entering the following console command:

   ```
   karaf@root> profile-create --parents gs-cxf-base gs-cxf-02
   ```

3. In the **gs-cxf-01** profile, set the **portNumber** configuration property to 8185, by entering the following console command:

   ```
   karaf@root> profile-edit -p
   org.fusesource.example.get.started/portNumber=8185 gs-cxf-01
   ```

4. In the **gs-cxf-02** profile, set the **portNumber** configuration property to 8186, by entering the following console command:

   ```
   karaf@root> profile-edit -p
   org.fusesource.example.get.started/portNumber=8186 gs-cxf-02
   ```

3.2.3. Deploy the Profiles

3.2.3.1. Deploy profiles to the child containers

Given that you have created the child containers, as described in Section 3.2.1, "Create a Fabric", and you have created the profiles, as described in Section 3.2.2, "Create Fabric Profiles", you can now deploy the profiles by performing the following steps:

1. Deploy the **gs-cxf-01** profile into the **child1** container. Enter the following console command:

   ```
   karaf@root> fabric:container-change-profile child1 gs-cxf-01
   ```

2. Deploy the **gs-cxf-02** profile into the **child2** container. Enter the following console command:

   ```
   karaf@root> fabric:container-change-profile child2 gs-cxf-02
   ```

3.2.3.2. Test the deployed profiles

The deployed profiles can be tested using the WS client from the **cxf-basic** Maven project described in Section 2.1, "Create a Web Services Project".

To test the deployed profiles, open a new command prompt and change directory to **get-started/cxf-basic**. To test the **gs-cxf-01** profile (which deploys a Web service listening on port 8185), enter the following command:

```
mvn -Pclient -Dexec.args="http://localhost:8185/PersonServiceCF"
```

To test the **gs-cxf-02** profile (which deploys a Web service listening on port 8186), enter the following command:

```
mvn -Pclient -Dexec.args="http://localhost:8186/PersonServiceCF"
```

3.2.4. Update a Profile

3.2.4.1. Atomic container upgrades

Normally, when you edit a profile that is already deployed in a container, *the modification takes effect immediately*. The reason for this is that the Fabric Agent in the affected container (or containers) actively monitors the fabric registry in real time.

In practice, however, the immediate propagation of profile modifications is often undesirable. In a production system, you typically want to roll out changes piecemeal: for example, trying out the change initially on just one container and checking for problems, before you go ahead and make the changes to all containers. Moreover, sometimes several edits must be made together, in order to reconfigure an application in a consistent way.

3.2.4.2. Profile versioning

For these reasons, it is typically better to modify profiles *atomically*, where several modifications are applied simultaneously. To support atomic updates, fabric implements profile versioning. Initially, the container points at version 1.0 of a profile. If you now edit a later version of the profile (for example, version 1.1), the changes are invisible to the container. After you are finished editing the profile, you can apply all of the modifications simultaneously by switching the container to use version 1.1 of the profile.

3.2.4.3. Upgrade to a new profile

For example, to modify the **`gs-cxf-01`** profile, when it is already deployed and running in a container, the recommended procedure is as follows:

1. Create a new version, 1.1, to hold the pending changes. Enter the following console command:

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

   The new version is initialised with a copy of all of the profiles from version 1.0.
2. Use the **`fabric:profile-edit`** command to change the **`portNumber`** of **`gs-cxf-01`** to the value, 8187. Remember to specify version **`1.1`** to the **`fabric:profile-edit`** command, so that the modifications are applied to version 1.1 of the **`gs-cxf-01`** profile. Enter the following command:

   ```
   karaf@root> fabric:profile-edit -p
   org.fusesource.example.get.started/portNumber=8187 gs-cxf-01 1.1
   ```

3. You are now ready to upgrade the **`child1`** container to version 1.1, so that the profile changes can take effect. Enter the following console command:

   ```
   karaf@root> fabric:container-upgrade 1.1 child1
   ```

3.2.4.4. Roll back to an old profile

You can easily roll back to the old version of the **`gs-cxf-01`** profile, using the **`fabric:container-rollback`** command, as follows:

```
karaf@root> fabric:container-rollback 1.0 child1
```