Red Hat JBoss Fuse 6.1

Deploying into a Web Server

Deploying Apache CXF and Apache Camel applications into an Enterprise Web Services container or an Enterprise Application Platform container

Last Updated: 2017-10-12

Red Hat JBoss Fuse 6.1 Deploying into a Web Server

Deploying Apache CXF and Apache Camel applications into an Enterprise Web Services container or an Enterprise Application Platform container

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

Legal Notice

Copyright © 2013 Red Hat.

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

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

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

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

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

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

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

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

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

All other trademarks are the property of their respective owners.

Abstract

The guide describes the options for deploying applications into a Red Hat JBoss Fuse container.

Table of Contents

CHAPTER 1. OVERVIEW OF JBOSS FUSE DEPLOYMENT **3**
1.1. SUPPORTED WEB SERVER PLATFORMS 3
1.2. WAR BUILD AND DEPLOYMENT MODEL 3

CHAPTER 2. BUILDING A WAR **5**
2.1. PREPARING TO USE MAVEN 5
2.2. MODIFYING AN EXISTING MAVEN PROJECT 8
2.3. BOOTSTRAPPING A CXF SERVLET IN A WAR 11
2.4. BOOTSTRAPPING A SPRING CONTEXT IN A WAR 12

CHAPTER 3. DEPLOYING AN APACHE CXF WEB SERVICE **14**
3.1. APACHE CXF EXAMPLE 14
3.2. DEPLOY THE APACHE CXF EXAMPLE 17

CHAPTER 4. DEPLOYING AN APACHE CAMEL SERVLET ENDPOINT **21**
4.1. APACHE CAMEL SERVLET EXAMPLE 21
4.2. DEPLOY THE APACHE CAMEL SERVLET 24

CHAPTER 5. DEPLOYING AN APACHE CAMEL WS ENDPOINT **27**
5.1. APACHE CAMEL CXF EXAMPLE 27
5.2. DEPLOY THE APACHE CAMEL CXF EXAMPLE 32

INDEX **34**

CHAPTER 1. OVERVIEW OF JBOSS FUSE DEPLOYMENT

Abstract

You have the option of deploying JBoss Fuse applications into various Web server products, such as Red Hat JBoss Enterprise Web Server and Red Hat JBoss Enterprise Application Platform. This mode of deployment is based on WAR files, which you can generate using the Apache Maven build tool.

1.1. SUPPORTED WEB SERVER PLATFORMS

Overview

The following Web server platforms are supported by JBoss Fuse 6.1:

- *JBoss Enterprise Web Server (JBoss EWS)*
- *JBoss Enterprise Application Platform (JBoss EAP)*

Supported product versions

To see which versions of JBoss EWS and JBoss EAP are supported with JBoss Fuse 6.1, please consult the JBoss Fuse 6.1 Supported Configurations page.

1.2. WAR BUILD AND DEPLOYMENT MODEL

How to install Fuse libraries

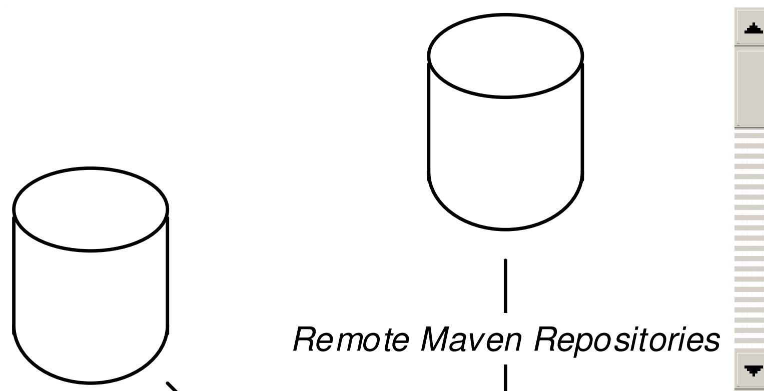
There is no need to install the JBoss Fuse libraries directly into a Web server installation. Under the WAR deployment model, all of the requisite JBoss Fuse libraries are packaged into your application's WAR file.

Before you can build a WAR file, however, you need to install and configure Apache Maven as described in Section 2.1, “Preparing to use Maven”.

Build model

Figure 1.1, “Building the WAR with Maven” shows an overview of the model for building WAR files for JBoss Fuse applications. The WAR file is generated using a Maven project, which compiles the Java source code belonging to the project and also downloads any required dependencies from remote Maven repositories. The WAR configuration, compiled classes, and downloaded (or locally cached) dependencies are then packaged into the WAR file.

Figure 1.1. Building the WAR with Maven



Maven POM file

The normal approach to building JBoss Fuse applications is to use the Apache Maven build system. The Maven build system is configured by a POM file, `pom.xml`, which is typically used to configure the following aspects of the build:

- The packaging type to be `war` (which instructs Maven to build a WAR file).
- Dependent JAR files, which will be bundled with the WAR (including the requisite JBoss Fuse libraries).
- The name of the WAR file.

Build process

Apache Maven is fundamentally a distributed build system. In its normal mode of operation (online), any dependencies it cannot find in its own cache (or *local repository*) will be downloaded from remote repositories on the Internet. Assuming that you have configured Maven with the Maven repository URLs for JBoss Fuse, Maven will download all of the required JBoss Fuse dependencies and *transitive dependencies* (that is, dependencies of dependencies), and embed these JBoss Fuse JAR libraries into the generated WAR file.

Deployment

The generated WAR file encapsulates all of the code and resources required to deploy your JBoss Fuse application to a Web server. After building the WAR, you can deploy it into your Web server in the usual way (for example, by manually copying it into a particular hot deploy directory for your Web server).

For more details, see the tutorials.

CHAPTER 2. BUILDING A WAR

Abstract

This chapter describes how to build and package a WAR using Maven.

2.1. PREPARING TO USE MAVEN

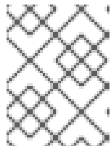
Overview

This section gives a brief overview of how to prepare Maven for building Red Hat JBoss Fuse projects and introduces the concept of Maven coordinates, which are used to locate Maven artifacts.

Prerequisites

In order to build a project using Maven, you must have the following prerequisites:

- *Maven installation*—Maven is a free, open source build tool from Apache. You can download the latest version from the Maven download page.
- *Network connection*—whilst performing a build, Maven dynamically searches external repositories and downloads the required artifacts on the fly. By default, Maven looks for repositories that are accessed over the Internet. You can change this behavior so that Maven will prefer searching repositories that are on a local network.



NOTE

Maven can run in an offline mode. In offline mode Maven will only look for artifacts in its local repository.

Adding the Red Hat JBoss Fuse repository

In order to access artifacts from the Red Hat JBoss Fuse Maven repository, you need to add it to Maven's **`settings.xml`** file. Maven looks for your **`settings.xml`** file in the **`.m2`** directory of the user's home directory. If there is not a user specified **`settings.xml`** file, Maven will use the system-level **`settings.xml`** file at **`M2_HOME/conf/settings.xml`**.

To add the JBoss Fuse repository to Maven's list of repositories, you can either create a new **`.m2/settings.xml`** file or modify the system-level settings. In the **`settings.xml`** file, add the **`repository`** element for the JBoss Fuse repository as shown in bold text in Example 2.1, “Adding the Red Hat JBoss Fuse Repositories to Maven”.

Example 2.1. Adding the Red Hat JBoss Fuse Repositories to Maven

```
<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>               <repository>
                    <id>fusesource.snapshot</id>

<url>http://repo.fusesource.com/nexus/content/groups/public-
snapshots/</url>
                    <snapshots>
                        <enabled>true</enabled>
                    </snapshots>
                    <releases>
                        <enabled>false</enabled>
                    </releases>
                </repository>
                <repository>
                    <id>apache-public</id>

<url>https://repository.apache.org/content/groups/public/</url>
                    <snapshots>
                        <enabled>true</enabled>
                    </snapshots>
                    <releases>
                        <enabled>true</enabled>
                    </releases>
                </repository>
                ...
            </repositories>
        </profile>
    </profiles>
    ...
</settings>
```

The preceding example also shows repository element for the following repositories:

- **`fusesource-snapshot`** repository—if you want to experiment with building your application using an Red Hat JBoss Fuse snapshot kit, you can include this repository.
- **`apache-public`** repository—you might not always need this repository, but it is often useful to include it, because JBoss Fuse depends on many of the artifacts from Apache.

Artifacts

The basic building block in the Maven build system is an *artifact*. The output of an artifact, after performing a Maven build, is typically an archive, such as a JAR or a WAR.

Maven coordinates

A key aspect of Maven functionality is the ability to locate artifacts and manage the dependencies between them. Maven defines the location of an artifact using the system of *Maven coordinates*, which uniquely define the location of a particular artifact. A basic coordinate tuple has the form, ***{groupId, artifactId, version}***. Sometimes Maven augments the basic set of coordinates with the additional coordinates, *packaging* and *classifier*. A tuple can be written with the basic coordinates, or with the additional *packaging* coordinate, or with the addition of both the *packaging* and *classifier* coordinates, as follows:

```
groupdId:artifactId:version
groupdId:artifactId:packaging:version
groupdId:artifactId:packaging:classifier:version
```

Each coordinate can be explained as follows:

groupdId

Defines a scope for the name of the artifact. You would typically use all or part of a package name as a group ID—for example, **org.fusesource.example**.

artifactId

Defines the artifact name (relative to the group ID).

version

Specifies the artifact's version. A version number can have up to four parts: **n.n.n.n**, where the last part of the version number can contain non-numeric characters (for example, the last part of **1.0-SNAPSHOT** is the alphanumeric substring, **0-SNAPSHOT**).

packaging

Defines the packaged entity that is produced when you build the project. For OSGi projects, the packaging is **bundle**. The default value is **jar**.

classifier

Enables you to distinguish between artifacts that were built from the same POM, but have different content.

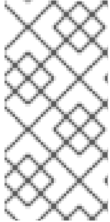
The group ID, artifact ID, packaging, and version are defined by the corresponding elements in an artifact's POM file. For example:

```
<project ... >
  ...
  <groupId>org.fusesource.example</groupId>
  <artifactId>bundle-demo</artifactId>
  <packaging>bundle</packaging>
  <version>1.0-SNAPSHOT</version>
  ...
</project>
```

For example, to define a dependency on the preceding artifact, you could add the following **dependency** element to a POM:

```
<project ... >
  ...
```

```
  <dependencies>
    <dependency>
      <groupId>org.fusesource.example</groupId>
      <artifactId>bundle-demo</artifactId>
      <version>1.0-SNAPSHOT</version>
    </dependency>
  </dependencies>
  ...
</project>
```



NOTE

It is *not* necessary to specify the **bundle** package type in the preceding dependency, because a bundle is just a particular kind of JAR file and **jar** is the default Maven package type. If you do need to specify the packaging type explicitly in a dependency, however, you can use the **type** element.

2.2. MODIFYING AN EXISTING MAVEN PROJECT

Overview

If you already have a Maven project and you want to modify it so that it generates a WAR, perform the following steps:

1. the section called “Change the package type to WAR”.
2. the section called “Customize the JDK compiler version” .
3. the section called “Store resources under webapp/WEB-INF”.
4. the section called “Customize the Maven WAR plug-in” .

Change the package type to WAR

Configure Maven to generate a WAR by changing the package type to **war** in your project's **pom.xml** file. Change the contents of the **packaging** element to **war**, as shown in the following example:

```
<project ... >
  ...
  <packaging>war</packaging>
  ...
</project>
```

The effect of this setting is to select the Maven WAR plug-in, **maven-war-plugin**, to perform packaging for this project.

Customize the JDK compiler version

It is almost always necessary to specify the JDK version in your POM file. If your code uses any modern features of the Java language—such as generics, static imports, and so on—and you have not customized the JDK version in the POM, Maven will fail to compile your source code. It is *not* sufficient to set the **JAVA_HOME** and the **PATH** environment variables to the correct values for your JDK, you must also modify the POM file.

To configure your POM file, so that it accepts the Java language features introduced in JDK 1.6, add the following **maven-compiler-plugin** plug-in settings to your POM (if they are not already present):

```
<project ... >
  ...
  <build>
    <defaultGoal>install</defaultGoal>
    <plugins>
      ...
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <source>1.6</source>
          <target>1.6</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

Store resources under webapp/WEB-INF

Resource files for the Web application are stored under the **/WEB-INF** directory in the standard WAR directory layout. In order to ensure that these resources are copied into the root of the generated WAR package, store the **WEB-INF** directory under ***ProjectDir*/src/main/webapp** in the Maven directory tree, as follows:

```
ProjectDir/
    pom.xml
    src/
        main/
            webapp/
                WEB-INF/
 web.xml
                    classes/
                    lib/
```

In particular, note that the **web.xml** file is stored at ***ProjectDir*/src/main/webapp/WEB-INF/web.xml**.

Customize the Maven WAR plug-in

It is possible to customize the Maven WAR plug-in by adding an entry to the **plugins** section of the **pom.xml** file. Most of the configuration options are concerned with adding additonal resources to the WAR package. For example, to include all of the resources under the **src/main/resources** directory (specified relative to the location of **pom.xml**) in the WAR package, you could add the following WAR plug-in configuration to your POM:

```
<project ...>
  ...
  <build>
```

```
    ...
    <plugins>
      <plugin>
        <artifactId>maven-war-plugin</artifactId>
     <version>2.1.1</version>
        <configuration>
          <!-- Optionally specify where the web.xml file comes from -->
          <webXml>src/main/webapp/WEB-INF/web.xml</webXml>
          <!-- Optionally specify extra resources to include -->
          <webResources>
            <resource>
              <directory>src/main/resources</directory>
              <targetPath>WEB-INF</targetPath>
              <includes>
                <include>**/*</include>
              </includes>
            </resource>
          </webResources>
        </configuration>
      </plugin>
      ...
    </plugins>
  </build>
</project>
```

The preceding plug-in configuration customizes the following settings:

webXml

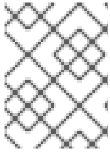
Specifies where to find the **web.xml** file in the current Maven project, relative to the location of **pom.xml**. The default is **src/main/webapp/WEB-INF/web.xml**.

webResources

Specifies additional resource files that are to be included in the generated WAR package. It can contain the following sub-elements:

- **webResources/resource**—each resource elements specifies a set of resource files to include in the WAR.
- **webResources/resource/directory**—specifies the base directory from which to copy resource files, where this directory is specified relative to the location of **pom.xml**.
- **webResources/resource/targetPath**—specifies where to put the resource files in the generated WAR package.
- **webResources/resource/includes**—uses an Ant-style wildcard pattern to specify explicitly which resources should be *included* in the WAR.
- **webResources/resource/excludes**—uses an Ant-style wildcard pattern to specify explicitly which resources should be *excluded* from the WAR (exclusions have priority over inclusions).

For complete details of how to configure the Maven WAR plug-in, see http://maven.apache.org/plugins/maven-war-plugin/index.html.



NOTE

Do not use version 2.1 of the **maven-war-plugin** plug-in, which has a bug that causes two copies of the **web.xml** file to be inserted into the generated **.war** file.

Building the WAR

To build the WAR defined by the Maven project, open a command prompt, go to the project directory (that is, the directory containing the **pom.xml** file), and enter the following Maven command:

```
mvn install
```

The effect of this command is to compile all of the Java source files, to generate a WAR under the ***ProjectDir*/target** directory, and then to install the generated WAR in the local Maven repository.

2.3. BOOTSTRAPPING A CXF SERVLET IN A WAR

Overview

A simple way to bootstrap Apache CXF in a WAR is to configure **web.xml** to use the standard CXF servlet, **org.apache.cxf.transport.servlet.CXFServlet**.

Example

For example, the following **web.xml** file shows how to configure the CXF servlet, where all Web service addresses accessed through this servlet would be prefixed by **/services/** (as specified by the value of **servlet-mapping/url-pattern**):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC
  "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
  "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <display-name>cxf</display-name>
  <description>cxf</description>

  <servlet>
    <servlet-name>cxf</servlet-name>
    <display-name>cxf</display-name>
    <description>Apache CXF Endpoint</description>
    <servlet-class>org.apache.cxf.transport.servlet.CXFServlet</servlet-
class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>cxf</servlet-name>
    <url-pattern>/services/*</url-pattern>
  </servlet-mapping>

  <session-config>
    <session-timeout>60</session-timeout>
```

```
  </session-config>

</web-app>
```

cxf-servlet.xml file

In addition to configuring the **web.xml** file, it is also necessary to configure your Web services by defining a **cxf-servlet.xml** file, which must be copied into the root of the generated WAR.

Alternatively, if you do not want to put **cxf-servlet.xml** in the default location, you can customize its name and location, by setting the **contextConfigLocation** context parameter in the **web.xml** file. For example, to specify that Apache CXF configuration is located in **WEB-INF/cxf-servlet.xml**, set the following context parameter in **web.xml**:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC
  "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
  "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
    ...
 <context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>WEB-INF/cxf-servlet.xml</param-value>
 </context-param>
    ...
</web-app>
```

Reference

For full details of how to configure the CXF servlet, see .

2.4. BOOTSTRAPPING A SPRING CONTEXT IN A WAR

Overview

You can bootstrap a Spring context in a WAR using Spring's ContextLoaderListener class.

Bootstrapping a Spring context in a WAR

For example, the following **web.xml** file shows how to boot up a Spring application context that is initialized by the XML file, **/WEB-INF/applicationContext.xml** (which is the location of the context file in the generated WAR package):

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

    <display-name>Camel Routes</display-name>
```

```
    <!-- location of spring xml files -->
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/applicationContext.xml</param-value>
    </context-param>

    <!-- the listener that kick-starts Spring -->
    <listener>
        <listener-
class>org.springframework.web.context.ContextLoaderListener</listener-
class>
    </listener>

</web-app>
```

Maven dependency

In order to access the **ContextLoaderListener** class from the Spring framework, you *must* add the following dependency to your project's **pom.xml** file:

```
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-web</artifactId>
            <version>${spring-version}</version>
        </dependency>
```

Where the **spring-version** property specifies the version of the Spring framework you are using.

CHAPTER 3. DEPLOYING AN APACHE CXF WEB SERVICE

Abstract

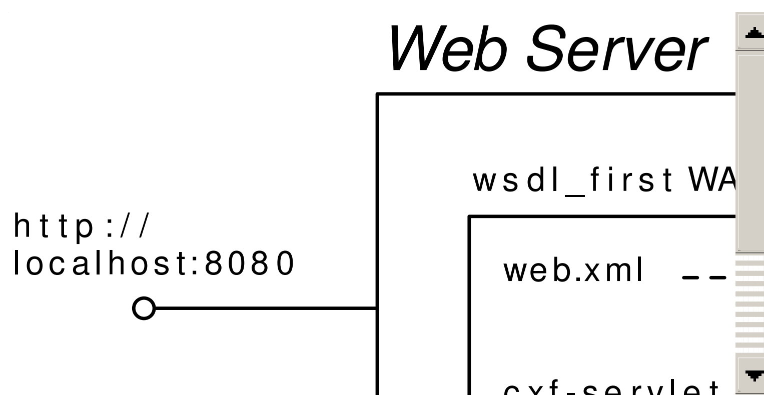
This tutorial describes how to deploy an Apache CXF Web services endpoint in a WAR file, where the Web service endpoint is implemented by binding directly to a Java class with the JAX-WS mapping.

3.1. APACHE CXF EXAMPLE

Overview

Figure 3.1, "Example Web Service Deployed in a Web Server" gives an overview of the Apache CXF example deployed in a Web server, which lets you see how the Web service's URL is constructed from settings at different configuration layers. The Web server's host and port, the WAR file name, the **url-pattern** setting from **web.xml**, and the **address** attribute of the Web services endpoint are combined to give the URL, **http://localhost:8080/wsdl_first/services/CustomerServicePort**.

Figure 3.1. Example Web Service Deployed in a Web Server



wsdl_first sample

The code for this example is available from the standard Apache CXF distribution, under the **samples/wsdl_first** directory. For details of how to install the Apache CXF distribution, see the section called "Install Apache CXF".

web.xml file

To deploy the example Web service as a servlet, you must provide a properly configured **web.xml** file. In the **wsdl_first** project, the **web.xml** file is stored at the following location:

```
wsdl_first/src/main/webapp/WEB-INF/web.xml
```

Example 3.1, "web.xml File for the wsdl_first Example" shows the contents of the **web.xml** file.

Example 3.1. web.xml File for the wsdl_first Example

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
         http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
    <display-name>cxf</display-name>

    <servlet>
        <servlet-name>cxf</servlet-name>
        <display-name>cxf</display-name>
        <description>Apache CXF Endpoint</description>
        <servlet-
class>org.apache.cxf.transport.servlet.CXFServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>cxf</servlet-name>
        <url-pattern>/services/*</url-pattern>
    </servlet-mapping>

    <session-config>
        <session-timeout>60</session-timeout>
    </session-config>

</web-app>
```

The key settings in the preceding **web.xml** file are:

- *Servlet class*—specifies the **org.apache.cxf.transport.servlet.CXFServlet** class, which implements a special servlet that integrates with Web services.
- *URL pattern*—determines which URLs are routed to this servlet. In general, the servlet URL has the following form:

  ```
  http://Host:Port/WARFileName/URLPattern
  ```

  Where the base URL, **http://*Host*:*Port***, is determined by the configuration of the Web server, the ***WARFileName*** is the root of the ***WARFileName*.war** WAR file, and the ***URLPattern*** is specified by the contents of the **url-pattern** element.

  Assuming that the Web server port is set to 8080, the **wsdl_first** example servlet will match URLs of the following form:

  ```
  http://localhost:8080/wsdl_first/services/*
  ```

Implied Spring container

The **CXFServlet** automatically creates and starts up a Spring container, which you can then use for defining Web service endpoints. By default, this Spring container automatically loads the following XML file in the WAR:

```
WEB-INF/cxf-servlet.xml
```

In the **wsdl_first** example project, this file is stored at the following location:

```
wsdl_first/src/main/webapp/WEB-INF/cxf-servlet.xml
```

cxf-servlet.xml file

The **cxf-servlet.xml** file is primarily used to create Web service endpoints, which represent the Web services exposed through the Web server. Apache CXF provides a convenient and flexible syntax for defining Web service endpoints in XML and you can use this flexible syntax to define endpoints in **cxf-servlet.xml**.

Example 3.2, “Spring Configuration for the wsdl_first Example” shows the contents of the **cxf-servlet.xml** file, which creates a single **CustomerService** endpoint, using the **jaxws:endpoint** element.

Example 3.2. Spring Configuration for the wsdl_first Example

```
<?xml version="1.0" encoding="UTF-8"?>
<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:soap="http://cxf.apache.org/bindings/soap"
      xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://cxf.apache.org/bindings/soap
http://cxf.apache.org/schemas/configuration/soap.xsd
http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd">

    <jaxws:endpoint
            xmlns:customer="http://customerservice.example.com/"
            id="CustomerServiceHTTP"
            address="/CustomerServicePort"
            serviceName="customer:CustomerServiceService"
            endpointName="customer:CustomerServicePort"

implementor="com.example.customerservice.server.CustomerServiceImpl">
 <!--jaxws:properties>
     <entry key="schema-validation-enabled" value="true" />
 </jaxws:properties-->
    </jaxws:endpoint>

</beans>
```

Note that the **address** attribute of the **jaxws:endpoint** specifies the final segment of the Web service's URL. When you put together all of the settings from the Web server, the **web.xml** file, and the **cxf-server.xml** file, you obtain the following URL for this Web service endpoint:

```
http://localhost:8080/wsdl_first/services/CustomerServicePort
```

WSDL address configuration

In addition to defining the servlet descriptor, **web.xml**, and the Spring configuration, **cxf-servlet.xml**, it is also necessary to ensure that the SOAP address in the WSDL contract is correctly specified, so that it matches the URL for this Web service.

In the **wsdl_first** example, the WSDL contract is located in the following file:

```
wsdl_first/src/main/resources/CustomerService.wsdl
```

In the WSDL contract, the **location** attribute of the **soap:address** element must be set to the correct Web service URL, as shown in Example 3.3, “Address in the WSDL CustomerService Contract” .

Example 3.3. Address in the WSDL CustomerService Contract

```
<?xml version="1.0" encoding="UTF-8"?>
...
<wsdl:definitions name="CustomerServiceService"
targetNamespace="http://customerservice.example.com/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:tns="http://customerservice.example.com/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">
  ...
  <wsdl:service name="CustomerServiceService">
    <wsdl:port name="CustomerServicePort"
binding="tns:CustomerServiceServiceSoapBinding">
      <!-- embedded deployment -->
      <!--soap:address
location="http://localhost:9090/CustomerServicePort"/-->
      <!-- standalone Tomcat deployment -->
      <soap:address
location="http://localhost:8080/wsdl_first/services/CustomerServicePort"
/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

3.2. DEPLOY THE APACHE CXF EXAMPLE

Overview

This tutorial takes a standard Apache CXF example (the **wsdl_first** example) and shows you how to deploy it into a Web server, by packaging the application as a WAR. In this example, the Web service is implemented by binding the service to a Java class with the JAX-WS mapping.

Prerequisites

The following prerequisites are needed to build and run this example:

- Either of the following Web servers are installed:
  - JBoss Enterprise Web Server, or

    - JBoss Enterprise Application Platform

- Java version 1.6 or later is installed.
- Apache Maven 3.0.0 or later is installed.
- Maven is configured to access the JBoss Fuse repositories, as described in Section 2.1, “Preparing to use Maven”.
- You have access to the Internet, so that Maven can download dependencies from remote repositories.

Install Apache CXF

To obtain the code for the **wsdl_first** example, you need to install the Apache CXF kit, **apache-cxf-2.7.0.redhat-610379.zip**, provided in the **extras/** directory of the JBoss Fuse installation.

Install the Apache CXF kit as follows:

1. Find the Apache CXF kit at the following location:

   ```
   InstallDir/extras/apache-cxf-2.7.0.redhat-610379.zip
   ```

2. Using a suitable archive utility on your platform, unzip the **apache-cxf-2.7.0.redhat-610379.zip** file and extract it to a convenient location, ***CXFInstallDir***.

The wsdl_first example

The **wsdl_first** example is located under the following sub-directory of the Apache CXF installation:

```
CXFInstallDir/samples/wsdl_first/
```

Build and run the example

To build and run the **wsdl_first** example, perform the following steps:

1. Using your favorite text editor, open the **CustomerService.wsdl** file, which can be found in the following location in the **wsdl_first** project:

   ```
   wsdl_first/src/main/resources/CustomerService.wsdl
   ```

   Edit the **soap:address** element in the WSDL contract, removing comments around the element labeled **standalone Tomcat deployment** and inserting comments around the element labeled **embedded deployment**. When you are finished editing, the **soap:address** element should be look as follows:

   ```
   <?xml version="1.0" encoding="UTF-8"?>
   ...
   <wsdl:definitions name="CustomerServiceService"
   targetNamespace="http://customerservice.example.com/"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
   xmlns:tns="http://customerservice.example.com/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   ```

```
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">
  ...
  <wsdl:service name="CustomerServiceService">
    <wsdl:port name="CustomerServicePort"
binding="tns:CustomerServiceServiceSoapBinding">
      <!-- embedded deployment -->
      <!--soap:address
location="http://localhost:9090/CustomerServicePort"/-->
      <!-- standalone Tomcat deployment -->
 <soap:address
location="http://localhost:8080/wsdl_first/services/CustomerServiceP
ort"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

2. Build the **wsdl_first** example using Maven. Change directory to the ***CXFInstallDir*/samples/wsdl_first** directory, open a command prompt, and enter the following command at the command line:

```
mvn clean package
```

If this command executes successfully, you should be able to find the WAR file, **wsdl_first.war**, under the **wsdl_first/target** sub-directory.

3. Make sure that the Web server is already running (a simple way to test this is to enter the URL, **http://localhost:8080**, into your browser). If you need to start the Web server, you can typically do this from the command line. The command to start the Web server depends on the particular product you are using, as follows:

   - *JBoss Enterprise Web Server (EWS)*—open a new command prompt and execute the **startup.sh** script from the **tomcat7/bin/** directory (or the **tomcat6/bin/** directory, as appropriate). For more details about how to configure and launch the EWS, see the *Installation Guide* from the JBoss Enterprise Web Server library.

   - *JBoss Enterprise Application Platform (EAP)*—for a standalone instance, open a new command prompt and execute the **bin/standalone.sh** script. For more details about how to configure and launch the EAP, see the *Administration and Configuration Guide* from the JBoss Enterprise Application Platform library.

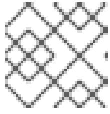
4. Deploy the **wsdl_first** example to the running Web server. Manually copy the **wsdl_first.war** WAR file from the **wsdl_first/target** directory to the Web server's deployment directory, as follows:

   - *JBoss Enterprise Web Server (EWS)*—copy the **wsdl_first.war** WAR file to the **tomcat7/webapps** directory (or **tomcat6/webapps** directory, as appropriate).

   - *JBoss Enterprise Application Platform (EAP)*—copy the **wsdl_first.war** WAR file to the **standalone/deployments** directory.

5. Use a Web browser to query the WSDL contract from the newly deployed Web service. Navigate to the following URL in your browser:

```
http://localhost:8080/wsdl_first/services/CustomerServicePort?wsdl
```



NOTE

This step might not work in the Safari browser.

6. Run the test client against the deployed Web service. Change directory to the ***CXFInstallDir*/samples/wsdl_first** directory, open a command prompt, and enter the following command at the command line:

```
mvn -Pclient
```

If the client runs successfully, you should see some output like the following in your command window:

```
...
Sending request for customers named Smith
Response received
Did not find any matching customer for name=None
NoSuchCustomer exception was received as expected
All calls were successful
```

CHAPTER 4. DEPLOYING AN APACHE CAMEL SERVLET ENDPOINT

Abstract

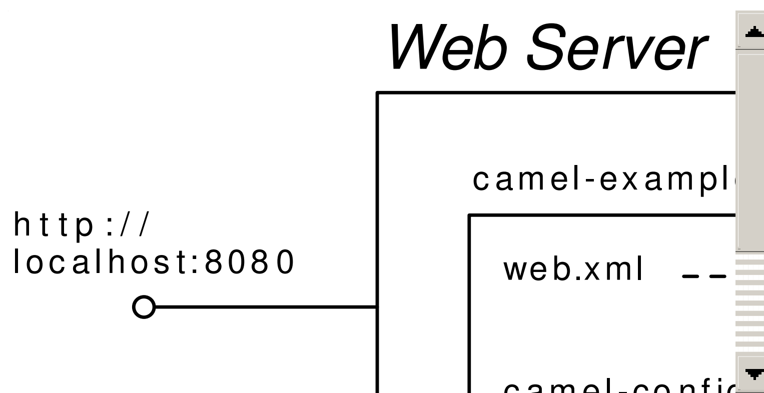
This tutorial describes how to deploy a Camel application, which is implemented using the Camel servlet component. The Camel application gets installed into the Web server as a servlet, receiving messages through the servlet endpoint which are then processed in a Camel route.

4.1. APACHE CAMEL SERVLET EXAMPLE

Overview

Figure 4.1, “Camel Servlet Example Deployed in a Web Server” gives an overview of the Camel servlet example deployed in a Web server, which lets you see how the servlet's URL is constructed from settings at different configuration layers. The Web server's host and port, the WAR file name, the **`url-pattern`** setting from **`web.xml`**, and the endpoint URI of the Camel servlet endpoint are combined to give the URL, **`http://localhost:8080/camel-example-servlet-tomcat-2.12.0.redhat-610379/camel/hello`**.

Figure 4.1. Camel Servlet Example Deployed in a Web Server



camel-example-servlet-tomcat example

The code for this example is available from the standard Apache Camel distribution, under the **`examples/camel-example-servlet-tomcat`** directory. For details of how to install the Apache Camel distribution, see the section called “Install Apache Camel”.

Camel servlet component

The Camel servlet component is used to process incoming HTTP requests, where the HTTP endpoint is bound to a published servlet. The servlet component is implemented by the following servlet class:

```
org.apache.camel.component.servlet.CamelHttpTransportServlet
```

To create a Camel servlet endpoint in a Camel route, define a servlet endpoint URI with the following syntax:

```
servlet://RelativePath[?Options]
```

Where ***RelativePath*** specifies the tail segment of the HTTP URL path for this servlet.

web.xml file

To deploy the Apache Camel servlet example, you must provide a properly configured **web.xml** file. In the **camel-example-servlet-tomcat** project, the **web.xml** file is stored at the following location:

```
camel-example-servlet-tomcat/src/main/webapp/WEB-INF/web.xml
```

Example 4.1, "web.xml File for the camel-example-servlet-tomcat Example" shows the contents of the **web.xml** file.

Example 4.1. web.xml File for the camel-example-servlet-tomcat Example

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

  <display-name>My Web Application</display-name>

  <!-- location of spring xml files -->
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:camel-config.xml</param-value>
  </context-param>

  <!-- the listener that kick-starts Spring -->
  <listener>
    <listener-
class>org.springframework.web.context.ContextLoaderListener</listener-
class>
  </listener>

  <!-- Camel servlet -->
  <servlet>
    <servlet-name>CamelServlet</servlet-name>
    <servlet-
class>org.apache.camel.component.servlet.CamelHttpTransportServlet</serv
let-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <!-- Camel servlet mapping -->
  <servlet-mapping>
    <servlet-name>CamelServlet</servlet-name>
    <url-pattern>/camel/*</url-pattern>
  </servlet-mapping>

</web-app>
```

The key settings in the preceding **web.xml** file are:

`servlet/servlet-class`

Specifies the **`org.apache.camel.component.servlet.CamelHttpTransportServlet`** class, which implements the Camel servlet component.

`servlet-mapping/url-pattern`

Determines which URLs are routed to this servlet. In general, the servlet URL has the following form:

```
http://Host:Port/WARFileName/URLPattern
```

Where the base URL, **`http://Host:Port`**, is determined by the configuration of the Web server, the ***`WARFileName`*** is the root of the ***`WARFileName`*`.war`** WAR file, and the ***`URLPattern`*** is specified by the contents of the **`url-pattern`** element.

Assuming that the Web server port is set to 8080, the **`camel-example-servlet-tomcat`** example servlet will match URLs of the following form:

```
http://localhost:8080/camel-example-servlet-tomcat-2.12.0.redhat-
610379/camel/*
```

`listener/listener-class`

This element launches the Spring container.

`context-param`

This element specifies the location of the Spring XML file, **`camel-config.xml`**, in the WAR. The Spring container will read this parameter and load the specified Spring XML file, which contains the definition of the Camel route.

Example Camel route

Example 4.2, "Route Definition for the Camel Servlet Example" shows the Camel route for this example, defined in a Spring XML file, using Camel's XML DSL syntax.

Example 4.2. Route Definition for the Camel Servlet Example

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

  <camelContext xmlns="http://camel.apache.org/schema/spring">

    <route>
      <!-- incoming requests from the servlet is routed -->
      <from uri="servlet:///hello"/>
      <choice>
```

```
        <when>
          <!-- is there a header with the key name? -->
          <header>name</header>
          <!-- yes so return back a message to the user -->
          <transform>
            <simple>Hello ${header.name} how are you?</simple>
          </transform>
        </when>
        <otherwise>
          <!-- if no name parameter then output a syntax to the user --
>
          <transform>
            <constant>Add a name parameter to uri, eg ?
name=foo</constant>
          </transform>
        </otherwise>
      </choice>
    </route>

  </camelContext>

</beans>
```

Because the servlet URL, **servlet:///hello**, specifies the relative path, **/hello**, the complete URL to access this servlet is the following:

```
http://localhost:8080/camel-example-servlet-tomcat-2.12.0.redhat-
610379/camel/hello
```

4.2. DEPLOY THE APACHE CAMEL SERVLET

Overview

This tutorial takes a standard Apache Camel example (the **camel-example-servlet-tomcat** example) and shows you how to deploy it into a Web server, by packaging the application as a WAR.

Prerequisites

The following prerequisites are needed to build and run this example:

- Either of the following Web servers are installed:
  - JBoss Enterprise Web Server, or
  - JBoss Enterprise Application Platform
- Java version 1.6 or later is installed.
- Apache Maven 3.0.0 or later is installed.
- Maven is configured to access the JBoss Fuse repositories, as described in Section 2.1, “Preparing to use Maven”.

- You have access to the Internet, so that Maven can download dependencies from remote repositories.

Install Apache Camel

To obtain the code for the **camel-example-servlet-tomcat** example, you need to install the Apache Camel kit, **apache-camel-2.12.0.redhat-610379.zip**, provided in the **extras/** directory of the JBoss Fuse installation.

Install the Apache Camel kit as follows:

1. Find the Apache Camel kit at the following location:

```
InstallDir/extras/apache-camel-2.12.0.redhat-610379.zip
```

2. Using a suitable archive utility on your platform, unzip the **apache-camel-2.12.0.redhat-610379.zip** file and extract it to a convenient location, ***CamelInstallDir***.

The camel-example-servlet-tomcat example

The **camel-example-servlet-tomcat** example is located under the following sub-directory of the Apache Camel installation:

```
CamelInstallDir/examples/camel-example-servlet-tomcat/
```

Build and run the example

To build and run the **camel-example-servlet-tomcat** example, perform the following steps:

1. Build the **camel-example-servlet-tomcat** example using Maven. Change directory to the ***CamelInstallDir*/examples/camel-example-servlet-tomcat/** directory, open a command prompt, and enter the following command at the command line:

```
mvn package
```

   If this command executes successfully, you should be able to find the WAR file, **camel-example-servlet-tomcat-2.12.0.redhat-610379.war**, under the **camel-example-servlet-tomcat/target** sub-directory.

2. Make sure that the Web server is already running (a simple way to test this is to enter the URL, **http://localhost:8080**, into your browser). If you need to start the Web server, you can typically do this from the command line. The command to start the Web server depends on the particular product you are using, as follows:

   - *JBoss Enterprise Web Server (EWS)*—open a new command prompt and execute the **startup.sh** script from the **tomcat7/bin/** directory (or the **tomcat6/bin/** directory, as appropriate). For more details about how to configure and launch the EWS, see the *Installation Guide* from the JBoss Enterprise Web Server library.

   - *JBoss Enterprise Application Platform (EAP)*—for a standalone instance, open a new command prompt and execute the **bin/standalone.sh** script. For more details about how to configure and launch the EAP, see the *Administration and Configuration Guide* from the JBoss Enterprise Application Platform library.

3. Deploy the **camel-example-servlet-tomcat** example to the running Web server. Manually copy the **camel-example-servlet-tomcat-2.12.0.redhat-610379.war** WAR file from the **camel-example-servlet-tomcat/target** directory to the Web server's deployment directory, as follows:

   - *JBoss Enterprise Web Server (EWS)*—copy the **camel-example-servlet-tomcat-2.12.0.redhat-610379.war** WAR file to the **tomcat7/webapps** directory (or **tomcat6/webapps** directory, as appropriate).

   - *JBoss Enterprise Application Platform (EAP)*—copy the **camel-example-servlet-tomcat-2.12.0.redhat-610379.war** WAR file to the **standalone/deployments** directory.

4. Navigate to the following URL in your browser:

   ```
   http://localhost:8080/camel-example-servlet-tomcat-2.12.0.redhat-610379/
   ```

   When the page loads, you should see the following text in your browser window:

   Camel Servlet and Apache Tomcat example

   This example shows how to use route messages in Apache Tomcat using servlets with Apache Camel.

   To get started click this link.

   This example is documented at servlet tomcat example
   If you hit any problems please let us know on the Camel Forums

   Please help us make Apache Camel better - we appreciate any feedback you may have. Enjoy!

   The Camel riders!

5. Click the highlighted link in the line **To get started click this link.** and follow the on-screen instructions to test the servlet.

CHAPTER 5. DEPLOYING AN APACHE CAMEL WS ENDPOINT

Abstract

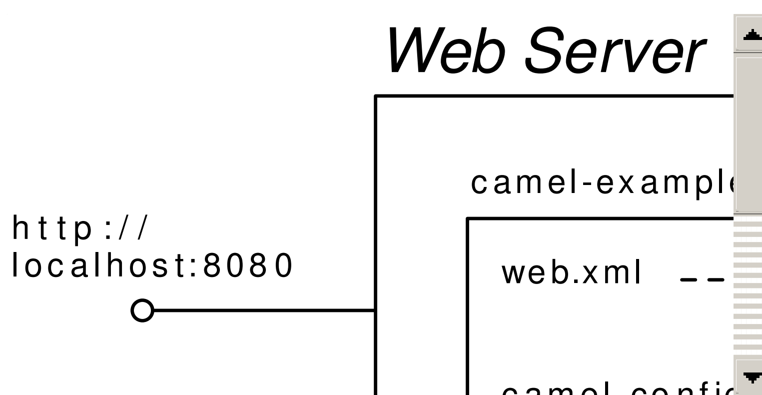
This tutorial describes how to deploy an Apache CXF Web services endpoint in a WAR file, where the Web service endpoint is implemented by binding to an Apache Camel route using the Camel CXF component.

5.1. APACHE CAMEL CXF EXAMPLE

Overview

Figure 5.1, “Camel CXF Example Deployed in a Web Server” gives an overview of the Camel CXF example deployed in a Web server, which lets you see how the Web service's URL is constructed from settings at different configuration layers. The Web server's host and port, the WAR file name, the **`url-pattern`** setting from **`web.xml`**, and the URI of the Camel CXF endpoint are combined to give the URL, **`http://localhost:8080/camel-example-cxf-tomcat/webservices/incident`**.

Figure 5.1. Camel CXF Example Deployed in a Web Server



camel-example-cxf-tomcat example

The code for this example is available from the standard Apache Camel distribution, under the **`examples/camel-example-cxf-tomcat`** directory. For details of how to install the Apache Camel distribution, see the section called “Install Apache Camel”.

Camel CXF component

The Camel CXF component binds an Apache CXF endpoint to a Camel route. That is, the Camel CXF endpoint itself is a fully-fledged Apache CXF Web service with all of the (potentially very complex) configuration options that are available from Apache CXF (including SSL security, WS-Security, and other WS-* standards). In contrast to the usual case, where you would bind the WS endpoint to a Java class (for example, using the JAX-WS binding), the Camel CXF component *binds the WS endpoint to a Camel route*, so that incoming SOAP message are encapsulated in a Camel **`Exchange`** object, and can then propagate through the route.

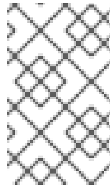
To create a Camel CXF endpoint in a Camel route, define a CXF endpoint URI with either of the following syntaxes:

```
cxf:Address[?Options]
```

Specifies the WSDL endpoint address and a (potentially large) number of options to configure the endpoint.

cxf:bean:*BeanID*[?*Options*]

References a bean with the ID, ***BeanID***, defined using the **cxf:cxfEndpoint** element (where the **cxf** prefix is bound to the **http://camel.apache.org/schema/cxf** namespace). The advantage of this approach is that all of the configuration complexity is encapsulated in the bean. Typically, this means that very few options (or none) need to be specified on the endpoint URI.



NOTE

The **cxf:cxfEndpoint** element, which binds a WS endpoint to a Camel route, should not be confused with the **jaxws:endpoint** element, which binds a WS endpoint directly to a Java class.

More about the Camel CXF component

For more details about the Camel CXF component, please consult the following documents from the JBoss Fuse library:

- *Web Services and Routing with Camel CXF*
- The *CXF* chapter from the *EIP Component Reference*.

web.xml file

To deploy the Apache Camel CXF example, you must provide a properly configured **web.xml** file. In the **camel-example-cxf-tomcat** project, the **web.xml** file is stored at the following location:

```
camel-example-cxf-tomcat/src/main/webapp/WEB-INF/web.xml
```

Example 5.1, “web.xml File for the camel-example-cxf-tomcat Example” shows the contents of the **web.xml** file.

Example 5.1. web.xml File for the camel-example-cxf-tomcat Example

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

 <display-name>My Web Application</display-name>

 <!-- location of spring xml files -->
 <context-param>
  <param-name>contextConfigLocation</param-name>
        <param-value>classpath:camel-config.xml</param-value>
 </context-param>

 <!-- the listener that kick-starts Spring -->
 <listener>
```

```
  <listener-
class>org.springframework.web.context.ContextLoaderListener</listener-
class>
 </listener>

 <!-- CXF servlet -->
 <servlet>
  <servlet-name>CXFServlet</servlet-name>
  <servlet-class>org.apache.cxf.transport.servlet.CXFServlet</servlet-
class>
  <load-on-startup>1</load-on-startup>
  <!-- If you want to leverage the Servlet3's async feature in Tomcat,
   please enable this feature
  <async-supported>true</async-supported>
  -->
 </servlet>

 <!-- all our webservices are mapped under this URI pattern -->
 <servlet-mapping>
  <servlet-name>CXFServlet</servlet-name>
  <url-pattern>/webservices/*</url-pattern>
 </servlet-mapping>

</web-app>
```

The key settings in the preceding **web.xml** file are:

servlet/servlet-class

Specifies the **org.apache.cxf.transport.servlet.CXFServlet** class, which implements a special servlet that enables you to deploy Apache CXF WS endpoints.

servlet-mapping/url-pattern

Determines which URLs are routed to this servlet. In general, the servlet URL has the following form:

```
http://Host:Port/WARFileName/URLPattern
```

Where the base URL, **http://*Host*:*Port***, is determined by the configuration of the Web server, the ***WARFileName*** is the root of the ***WARFileName*.war** WAR file, and the ***URLPattern*** is specified by the contents of the **url-pattern** element.

Assuming that the Web server port is set to 8080, the **camel-example-cxf-tomcat** example servlet will match URLs of the following form:

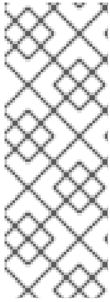
```
http://localhost:8080/camel-example-cxf-tomcat/webservices/*
```

listener/listener-class

This element launches a Spring container.

context-param

This element specifies the location of the Spring XML file, **camel-config.xml**, in the WAR. The Spring container will read this parameter and load the specified Spring XML file, which contains the definition of the Camel route.



NOTE

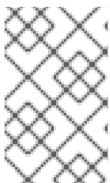
Strictly speaking, it is not absolutely necessary to create a Spring container explicitly using the **listener-class** element here, because the **CXFServlet** class already creates its own Spring container. If you put the Spring XML file in the location expected by the **CXFServlet** class (that is, **WEB-INF/cxf-servlet.xml**) instead of the location used by this example (that is, **WEB-INF/classes/camel-config.xml**), you could remove the Spring container settings from this **web.xml** file.

Spring XML file

The Spring XML file for this example, **camel-config.xml**, contains the following XML code:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>

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

  <bean id="myRoutes" class="org.apache.camel.example.cxf.CamelRoute"/>

  <camelContext xmlns="http://camel.apache.org/schema/spring">
    <routeBuilder ref="myRoutes"/>
  </camelContext>

</beans>
```

In this example, the Spring XML file is used just as a convenient mechanism to bootstrap the Camel context. The XML syntax is used to create the Camel context, but the code for the **RouteBuilder** class is defined in the Java class, **org.apache.camel.example.cxf.CamelRoute**.



NOTE

The resource import in this file was required in earlier versions of Apache CXF, in order to import some standard, boilerplate definitions. But in recent versions of Apache CXF, this import is not required, and you can safely remove it from the Spring XML file.

Camel route class

Example 5.2, “Route Definitions in the CamelRoute Class” shows the Camel routes for this example, defined using the Java DSL.

Example 5.2. Route Definitions in the CamelRoute Class

```
// Java
package org.apache.camel.example.cxf;

import org.apache.camel.Exchange;
import org.apache.camel.Processor;
```

```
import org.apache.camel.builder.RouteBuilder;
import org.apache.camel.example.cxf.incident.IncidentService;
import org.apache.camel.example.cxf.incident.InputReportIncident;
import org.apache.camel.example.cxf.incident.OutputReportIncident;
import org.apache.camel.example.cxf.incident.OutputStatusIncident;

// this static import is needed for older versions of Camel than 2.5
// import static org.apache.camel.language.simple.SimpleLanguage.simple;

public class CamelRoute extends RouteBuilder {

    // CXF webservice using code first approach
    private String uri = "cxf:/incident?serviceClass=" +
IncidentService.class.getName();

    @Override
    public void configure() throws Exception {
        from(uri)
            .to("log:input")
            // send the request to the route to handle the operation
            // the name of the operation is in that header
            .recipientList(simple("direct:${header.operationName}"));

        // report incident
        from("direct:reportIncident")
            .process(new Processor() {
                public void process(Exchange exchange) throws Exception
{
                    // get the id of the input
                    String id =
exchange.getIn().getBody(InputReportIncident.class).getIncidentId();

                    // set reply including the id
                    OutputReportIncident output = new
OutputReportIncident();
                    output.setCode("OK;" + id);
                    exchange.getOut().setBody(output);
                }
            })
            .to("log:output");

        // status incident
        from("direct:statusIncident")
            .process(new Processor() {
                public void process(Exchange exchange) throws Exception
{
                    // set reply
                    OutputStatusIncident output = new
OutputStatusIncident();
                    output.setStatus("IN PROGRESS");
                    exchange.getOut().setBody(output);
                }
            })
            .to("log:output");
    }
}
```

The most important feature of this class is the Camel CXF endpoint URI, which appears at the start of the first route (in the `from(uri)` DSL command). The Camel CXF endpoint is defined using the following endpoint URI:

```
cxf:/incident?
serviceClass=org.apache.camel.example.cxf.incident.IncidentService
```

Where we have substituted the literal name of the **IncidentService** class in this URI. The relative path, **/incident**, defines the tail of the servlet URL for this Web service. Hence, the full servlet URL for the Web service is the following:

```
http://localhost:8080/camel-example-cxf-tomcat/webservices/incident
```

The **serviceClass** option specifies the name of the Service Endpoint Interface (SEI) for this Web service. By default, the CXF endpoint is set up to use the POJO mode, using the SEI to check the syntax of incoming messages.

5.2. DEPLOY THE APACHE CAMEL CXF EXAMPLE

Overview

This tutorial takes a standard Camel CXF example (the **camel-example-cxf-tomcat** example) and shows you how to deploy it into a Web server, by packaging the application as a WAR. In this example, the Web service is implemented by binding the service to a Camel route using the Camel CXF component.

Prerequisites

The following prerequisites are needed to build and run this example:

- Either of the following Web servers are installed:
  - JBoss Enterprise Web Server, or
  - JBoss Enterprise Application Platform
- Java version 1.6 or later is installed.
- Apache Maven 3.0.0 or later is installed.
- Maven is configured to access the JBoss Fuse repositories, as described in Section 2.1, “Preparing to use Maven”.
- You have access to the Internet, so that Maven can download dependencies from remote repositories.

Install Apache Camel

To obtain the code for the **camel-example-cxf-tomcat** example, you need to install the Apache Camel kit, **apache-camel-2.12.0.redhat-610379.zip**, provided in the **extras/** directory of the JBoss Fuse installation.

Install the Apache Camel kit as follows:

1. Find the Apache Camel kit at the following location:

   ```
   InstallDir/extras/apache-camel-2.12.0.redhat-610379.zip
   ```

2. Using a suitable archive utility on your platform, unzip the **apache-camel-2.12.0.redhat-610379.zip** file and extract it to a convenient location, ***CamelInstallDir***.

The camel-example-cxf-tomcat example

The **camel-example-cxf-tomcat** example is located under the following sub-directory of the Apache Camel installation:

```
CamelInstallDir/examples/camel-example-cxf-tomcat/
```

Build and run the example

To build and run the **camel-example-cxf-tomcat** example, perform the following steps:

1. Build the **camel-example-cxf-tomcat** example using Maven. Change directory to the ***CamelInstallDir*/examples/camel-example-cxf-tomcat/** directory, open a command prompt, and enter the following command at the command line:

   ```
   mvn clean package
   ```

   If this command executes successfully, you should be able to find the WAR file, **camel-example-cxf-tomcat.war**, under the **camel-example-cxf-tomcat/target** sub-directory.

2. Make sure that the Web server is already running (a simple way to test this is to enter the URL, **http://localhost:8080**, into your browser). If you need to start the Web server, you can typically do this from the command line. The command to start the Web server depends on the particular product you are using, as follows:

   - *JBoss Enterprise Web Server (EWS)*—open a new command prompt and execute the **startup.sh** script from the **tomcat7/bin/** directory (or the **tomcat6/bin/** directory, as appropriate). For more details about how to configure and launch the EWS, see the *Installation Guide* from the JBoss Enterprise Web Server library.

   - *JBoss Enterprise Application Platform (EAP)*—for a standalone instance, open a new command prompt and execute the **bin/standalone.sh** script. For more details about how to configure and launch the EAP, see the *Administration and Configuration Guide* from the JBoss Enterprise Application Platform library.

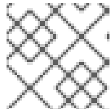
3. Deploy the **camel-example-cxf-tomcat** example to the running Web server. Manually copy the **camel-example-cxf-tomcat.war** WAR file from the **camel-example-cxf-tomcat/target** directory to the Web server's deployment directory, as follows:

   - *JBoss Enterprise Web Server (EWS)*—copy the **camel-example-cxf-tomcat.war** WAR file to the **tomcat7/webapps** directory (or **tomcat6/webapps** directory, as appropriate).

- *JBoss Enterprise Application Platform (EAP)*—copy the **`camel-example-cxf-tomcat.war`** WAR file to the **`standalone/deployments`** directory.

4. Use a Web browser to query the WSDL contract from the newly deployed Web service. Navigate to the following URL in your browser:

```
http://localhost:8080/camel-example-cxf-tomcat/webservices/incident?
wsdl
```



NOTE

This step might not work in the Safari browser.

5. Run the test client against the deployed Web service. Change directory to the ***`CamelInstallDir`*`/examples/camel-example-cxf-tomcat/`** directory, open a command prompt, and enter the following command at the command line:

```
mvn exec:java
```

If the client runs successfully, you should see some output like the following in your command window:

```
...
[INFO] --- exec-maven-plugin:1.1.1:java (default-cli) @ camel-
example-cxf-tomcat ---
2013-07-24 13:59:16,829 [teClient.main()] INFO
ReflectionServiceFactoryBean
- Creating Service
{http://incident.cxf.example.camel.apache.org/}IncidentService
from class org.apache.camel.example.cxf.incident.IncidentService
OK;123
IN PROGRESS
[INFO] ---------------------------------------------------------
--------------
[INFO] BUILD SUCCESS
[INFO] ---------------------------------------------------------
--------------
[INFO] Total time: 7.445s
[INFO] Finished at: Wed Jul 24 13:59:17 CEST 2013
[INFO] Final Memory: 10M/81M
[INFO] ---------------------------------------------------------
--------------
```

INDEX