Red Hat JBoss Fuse 6.3

Deploying into a Web Server

Deploying Apache CXF and Apache Camel applications into a JBoss Web Server or a JBoss Enterprise Application Platform container

Last Updated: 2018-07-16

Red Hat JBoss Fuse 6.3 Deploying into a Web Server

Deploying Apache CXF and Apache Camel applications into a JBoss Web Server or a JBoss Enterprise Application Platform container

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

Legal Notice

Copyright © 2016 Red Hat.

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

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

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

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

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

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

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

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

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

All other trademarks are the property of their respective owners.

Abstract

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

Abstract

You have the option of deploying JBoss Fuse applications into various Web server products, such as Red Hat JBoss Web Server and Red Hat JBoss Enterprise Application Platform. There are two deployment models available: the Camel on EAP subsystem (specifically for Apache Camel applications on JBoss EAP); and WAR files.

1.1. SUPPORTED WEB SERVER PLATFORMS

Overview

The following Web server platforms are supported by JBoss Fuse 6.3:

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

Supported product versions

To see which versions of JBoss WS and JBoss EAP are supported with JBoss Fuse 6.3, please consult the Supported Configurations page.

1.2. CAMEL ON EAP SUBSYSTEM

Overview

The *Camel on EAP* subsystem integrates Apache Camel directly into the JBoss EAP container. This subsystem is available after you install the *Fuse on EAP* package into the JBoss EAP container. This subsystem offers many advantages for Camel deployment, including simplified deployment of Camel components and tighter integration with the underlying JBoss EAP container. For deployment of Apache Camel applications on JBoss EAP, this deployment model is recommended over the WAR deployment model.

For details of this approach, see Chapter 2, *Apache Camel on JBoss EAP*.

1.3. 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.

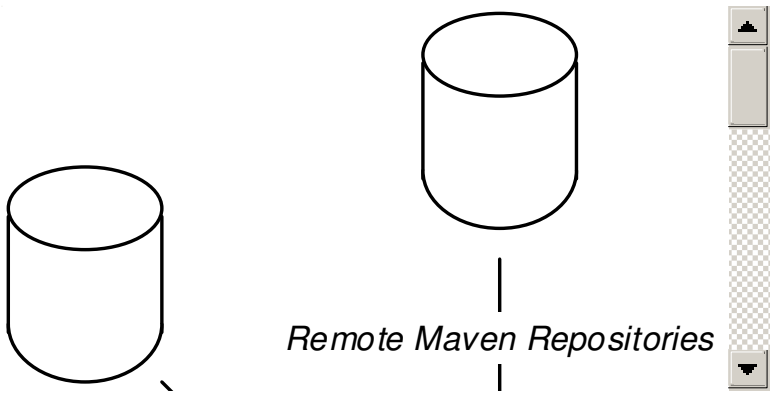
Before you can build a WAR file, however, you need to install and configure Apache Maven as described in Section 3.1, “Preparing to use Maven”.

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. APACHE CAMEL ON JBOSS EAP

Abstract

Apache Camel in JBoss Fuse enables you to select your own way to develop an integrated application. Also, It allows you to select the container to run. This chapter describes how to install JBoss Fuse on JBoss EAP.

2.1. OVERVIEW OF USING JBOSS FUSE ON JBOSS EAP

Installing JBoss Fuse on JBoss EAP enables you to add Camel routes as a part of the JBoss EAP configuration. You can deploy routes as a part of JavaEE applications.

The main goal of JBoss Fuse on JBoss EAP is to provide Camel features as a direct usable option within JBoss EAP. You do not need to configure or deploy anything. The other advantages of packaging Camel on JBoss EAP as a global library are:

- Less bloated war deployments by using the EAP Camel Subsystem instead of packaging Fuse libraries in a WAR file.
- Patch the application server instead of individual deployments
- No need to ship camel dependencies with applications
- Use supported versions of component libraries

2.2. INSTALL JBOSS FUSE ON JBOSS EAP

JBoss Fuse 6.3 must be installed on JBoss EAP 6.4. Download the latest version of JBoss EAP 6.4 from JBoss EAP 6.4 Installer Download. See JBoss EAP 6.4 Installation Guide for installation instructions.

IMPORTANT

We recommend that you apply the latest patch to JBoss EAP 6.4. The JBoss EAP patch level should be at least 6.4.10, to avoid known issues. Patches are available from the JBoss EAP 6.4 Patches tab of the download page.

Procedure 2.1. Downlad and Install JBoss Fuse

1. Download the Red Hat JBoss Fuse 6.3.0 on EAP Installer:
    1. Browse to the JBoss Fuse Software Downloads page on the Red Hat Customer Portal and, when prompted, login to your customer account.
    2. Select version **`6.3.0`** from the **`Version`** dropdown menu and click the **`Download`** link for the *Red Hat JBoss Fuse 6.3.0 on EAP Installer*package. Download the installer package to a temporary location.
2. Navigate to *EAP_HOME* of a clean instance of JBoss EAP.
3. Run the downloaded installer with the following command:

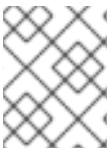
```
java -jar TEMP_LOCATION/fuse-eap-installer-6.3.0.redhat-187.jar
```

NOTE

Once a datastore has been has been selected at installation, it can not be changed.

2.3. WORKING WITH CAMEL ON JBOSS EAP

Here are some basic examples that describe how the Camel subsystem interacts with JBoss EAP configuration files.



NOTE

For more working examples of the Camel subsystem and JBoss EAP, see the quickstarts provided with JBoss EAP.

2.3.1. Using a Camel Context

The **camelContext** represents a single Camel routing rulebase. It contains all the routes of your application. You can have as many camelContexts as necessary, provided they have different names.

Camel on EAP allows you to:

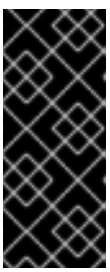
- define a **camelContext** as a part of the subsystem definition in the **standalone.xml** and **domain.xml** files
- deploy them in a supported deployment artifact that includes the **-camel-context.xml** suffixed file
- provide camelContexts along with their routes via a RouteBuilder and the CDI integration

You can configure a **camelContext** as a part of the subsystem definition this way:

```
<route>
       <from uri="direct:start"/>
       <transform>
         <simple>Hello #{body}</simple>
       </transform>
</route>
```

Also, you can consume a defined **camelContext** two ways:

- using annotation **@Inject** via **camel-cdi**
- via JNDI tree



IMPORTANT

To inject by CDI a deployed **camelContext** defined in Spring XML, you need to use the Java **@Resource** annotation, instead of the **@Inject @ContextName** annotations in the Camel CDI extension. Using the **@Inject @ContextName** annotations can result in the creation of a new **camelContext** instead of injecting the named context, which later causes endpoint lookups to fail.

2.3.1.1. Example of a Context and a Route

The following example, describes a context along with an associated route provided via CDI and a RouteBuilder. It displays an application scoped bean that starts automatically, when you start an application. The **@ContextName** annotation provides a specific name to the CamelContext.

```
@ApplicationScoped
@Startup
@ContextName("cdi-context")
public class HelloRouteBuilder extends RouteBuilder {

    @Inject
    HelloBean helloBean;

    @Override
    public void configure() throws Exception {

from("direct:start").transform(body().prepend(helloBean.sayHello()).append
(" user."));
    }
}
```

2.3.1.2. Configuring Camel Context using CDI Mechanism

Camel CDI automatically deploys and configures a **CamelContext** bean. After you initialise the CDI container, a CamelContext bean starts and instantiates automatically.

You can inject a **CamelContext** bean into the application as:

```
    @Inject
    @ContextName("cdi-context")
    private CamelContext context;
```

2.3.1.3. Configuring Camel Routes using CDI Mechanism

After you initialise the CDI container, Apache Camel CDI automatically collects all the RouteBuilder beans in the application, instantiates and adds them to the CamelContext bean instance.

For example, you can add a camel route and declare a class in the following way:

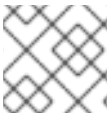
```
class MyRouteBean extends RouteBuilder {

    @Override
    public void configure() {
        from("jms:invoices").to("file:/invoices");
   }
  }
```

2.3.1.4. Customizing Camel Context

Apache Camel CDI provides **@ContextName** qualifier that allows you to change the name of the default CamelContext bean. For example:

```
@ApplicationScoped
class CustomCamelContext extends DefaultCamelContext {
```

```
    @PostConstruct
    void customize() {
        // Set the Camel context name
        setName("custom");
        // Disable JMX
        disableJMX();
    }

    @PreDestroy
    void cleanUp() {
        // ...
    }
}
```



NOTE

You can use any **CamelContext** class to declare a custom camel context bean.

2.3.1.5. Supporting Multiple CamelContexts

You can declare any number of CamelContext beans in your application. The CDI qualifiers declared on these CamelContext beans are used to bind the Camel routes and other Camel primitives to the corresponding Camel contexts.

The CDI qualifiers declared on the CamelContext beans are also used to bind the corresponding Camel primitives. For example:

```
@Inject
@ContextName("foo")
@Uri("direct:inbound")
ProducerTemplate producerTemplate;

@Inject
@BarContextQualifier
MockEndpoint outbound; // URI defaults to the member name, i.e.
mock:outbound

@Inject
@ContextName("baz")
@Uri("direct:inbound")
Endpoint endpoint;
```

2.3.2. Camel Context Deployment

You can deploy a Camel context to JBoss EAP two ways:

- Use the ***-camel-context.xml*** suffix as a part of another supported deployment, such as a JAR, WAR, or EAR deployment

  This deployment may contain multiple **-camel-context.xml** files.

- Use the ***-camel-context.xml*** suffix in a standalone **XML** file deployment by dropping the file into the EAP deployment directory

A deployed Spring-based Camel context is CDI injectable as:

```
@Resource(name = "java:jboss/camel/context/mycontext")
CamelContext camelContext;
```

In this example, the string **`java:jboss/camel/context/mycontext`** is the name assigned the deployed camel context in the JNDI registry. **`mycontext`** is the **`xml:id`** of the **`camelContext`** element in the Camel context XML file.

2.3.3. Hawtio Web Console

HawtIO is a web application that runs in a JVM. You can start Hawtio on your machine:

- deploy HawtIO as a WAR file
- add some users to your management and application realms by using the following command: **`$ bin/add-user.sh`**
- navigate to the **`http://localhost:8080/hawtio`**, the HawtIO login page appears
- Click **Camel** in the top navigation bar to view all the running Camel Contexts

Apache Camel plugin allows you to browse all the running Camel applications in the current JVM. You can also view the following details:

- list of all the running camel applications
- detail information of each CamelContext such as Camel version number, runtime statics
- list of all the routes and their runtime statistics in each camel application
- manage the lifecycle of all camel applications and their routes
- graphical representation of the running routes along with real time metrics
- live tracing and debugging of running routes
- profile the running routes with real time runtime statics
- browse and send messages to camel endpoint

2.3.4. Selecting Components

If you add nested component or component-module XML elements, then instead of the default list of Camel components, only the specified elements will be added to your deployment.

For example:

```
<jboss umlns="urn:jboss:1.0">
    <jboss-camel xmlns="urn:jboss:jboss-camel:1.0">
        <component name="camel-ftp"/>
        <component-module name="org.apache.camel.component.rss"/>
    </jboss-camel>
</jboss>
```

2.3.5. Configuring Camel Subsystem

The Camel subsystem configuration may contain static system routes. However, these routes are started automatically.

```
<route>
    <from uri="direct:start"/>
    <transform>
        <simple>Hello #{body}</simple>
    </transform>
</route>
```

2.3.6. Configuring Camel Deployment

To make changes in the default configuration of your Camel deployment, you can edit either **`WEB-INF/jboss-all.xml`** or **`META-INF/jboss-all.xml`** configuration file.

Use a jboss-camel XML element within the **`jboss-all.xml`** file, to control the camel configuration.

2.4. INTEGRATION WITH JMS

The **`camel-jms`** component provides messaging support. It integrates with the EAP Messaging (HornetQ) subsystem. Integration with other JMS implementations is possible by using the JBoss generic JMS Resource Adapter.

2.4.1. Configuring EAP JMS

With the help of standard EAP XML configuration files, you can configure the EAP messaging subsystem. The following example displays the configuration of a new JMS queue on the messaging subsystem, by adding the XML configuration to the **`jms-destinations`** section.

```
<jms-queue name="WildFlyCamelQueue">
  <entry name="java:/jms/queue/WildFlyCamelQueue"/>
</jms-queue>
```

However, you can also use a CLI script to add the queue.

```
jms-queue add --queue-address=WildFlyCamelQueue --
entries=queue/WildFlyCamelQueue,java:/jms/queue/WildFlyCamelQueue
```

2.4.2. Configuring Camel Route

The following examples of JMS producer and consumer illustrates the use of EAP embedded HornetQ server to publish and consume messages, to and from destinations.

However, it also displays the use of CDI in conjunction with the **`camel-cdi`** component. You can inject the JMS ConnectionFactory instances into the Camel RouteBuilder via JNDI lookups.

2.4.2.1. JMS Producer

You can start the ***`RouteBuilder`*** by injecting the ***`DefaultJMSConnectionFactory`*** connection factory from JNDI. However, the connection factory is defined within the messaging subsystem. Also, a timer endpoint runs after every 10 seconds to share an XML payload to the ***`EAPCamelQueue`*** destination.

```
@Startup
@ApplicationScoped
@ContextName("jms-camel-context")
public class JmsRouteBuilder extends RouteBuilder {

  @Resource(mappedName = "java:jboss/DefaultJMSConnectionFactory")
  private ConnectionFactory connectionFactory;

  @Override
  public void configure() throws Exception {
    JmsComponent component = new JmsComponent();
    component.setConnectionFactory(connectionFactory);

    getContext().addComponent("jms", component);

    from("timer://sendJMSMessage?fixedRate=true&period=10000")
    .transform(constant("<?xml version='1.0><message><greeting>hello
world</greeting></message>"))
    .to("jms:queue:WildFlyCamelQueue")
    .log("JMS Message sent");
  }
}
```

When you add a JMS message to the ***EAPCamelQueue*** destination, a log message appears. Use the EAP Administration console to verify that the messages are placed onto the queue.

2.4.2.2. JMS Consumer

To consume JMS messages, inject and set the connection factory from JNDI on the JMSComponent instance. When the JMS endpoint consumes messages from the EAPCamelQueue destination, the content is logged to the console.

```
@Override
public void configure() throws Exception {
  JmsComponent component = new JmsComponent();
  component.setConnectionFactory(connectionFactory);

  getContext().addComponent("jms", component);

  from("jms:queue:WildFlyCamelQueue")
  .to("log:jms?showAll=true");
}
```

2.4.2.3. JMS Transactions

To enable the Apache Camel JMS routes to participate in JMS transactions, you need to configure some spring classes to enable them to work with the EAP transaction manager and connection factory. The following example illustrates how to use CDI to configure a transactional JMS Camel route.

The **`camel-jms`** component requires a transaction manager of type ***`org.springframework.transaction.PlatformTransactionManager`***. Therefore, you can create a bean extending JtaTransactionManager.

NOTE

Use the annotation ***@Named*** to make the bean available to Camel. It enables you to register the bean within the camel bean registry. Also, inject the EAP transaction manager and user transaction instances by using CDI.

```
@Named("transactionManager")
public class CdiTransactionManager extends JtaTransactionManager {

  @Resource(mappedName = "java:/TransactionManager")
  private TransactionManager transactionManager;

  @Resource
  private UserTransaction userTransaction;

  @PostConstruct
  public void initTransactionManager() {
    setTransactionManager(transactionManager);
    setUserTransaction(userTransaction);
  }
}
```

Declare the transaction policy that you want to implement. Use the annotation to make the bean available to Camel. Inject the transaction manager, so that you can create a ***`TransactionalTemplate`*** using the desired transaction policy. For example, ***`PROPAGATION_REQUIRED`*** in this instance.

```
@Named("PROPAGATION_REQUIRED")
public class CdiRequiredPolicy extends SpringTransactionPolicy {
  @Inject
  public CdiRequiredPolicy(CdiTransactionManager cdiTransactionManager) {
    super(new TransactionTemplate(cdiTransactionManager,
      new
DefaultTransactionDefinition(TransactionDefinition.PROPAGATION_REQUIRED)))
;
  }
}
```

Also, you can configure the Camel RouteBuilder class and inject the dependencies you need for the Camel JMS component as shown in the given example:

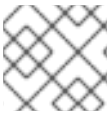
```
@Startup
@ApplicationScoped
@ContextName("jms-camel-context")
public class JMSRouteBuilder extends RouteBuilder {

  @Resource(mappedName = "java:/JmsXA")
  private ConnectionFactory connectionFactory;

  @Inject
  CdiTransactionManager transactionManager;

  @Override
  public void configure() throws Exception {
    // Creates a JMS component which supports transactions
```

```
    JmsComponent jmsComponent =
JmsComponent.jmsComponentTransacted(connectionFactory,
transactionManager);
    getContext().addComponent("jms", jmsComponent);

    from("jms:queue:queue1")
      .transacted("PROPAGATION_REQUIRED")
      .to("jms:queue:queue2");

       from("jms:queue:queue2")
      .to("log:end")
      .rollback();
  }
```

2.4.2.4. Remote JMS destinations

You can send messages from one EAP instance to HornetQ destinations configured on an other EAP instance, through remote JNDI. To send messages, you need to configure an exported JMS queue. Only JNDI names bound in the **`java:jboss/export`** namespace are appropriate for remote clients.

```
<jms-queue name="RemoteQueue">
  <entry name="java:jboss/exported/jms/queues/RemoteQueue"/>
</jms-queue>
```



NOTE

Configure the queue on the EAP client application server and EAP remote server.

CHAPTER 3. BUILDING A WAR

Abstract

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

3.1. PREPARING TO USE MAVEN

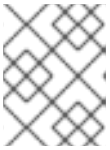
Overview

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 3.1, “Adding the Red Hat JBoss Fuse Repositories to Maven”.

Example 3.1. Adding the Red Hat JBoss Fuse Repositories to Maven

```
<?xml version="1.0"?>
<settings>

  <profiles>
    <profile>
      <id>extra-repos</id>
      <activation>
        <activeByDefault>true</activeByDefault>
```

```
      </activation>
      <repositories>
       <repository>
            <id>redhat-ga-repository</id>
            <url>https://maven.repository.redhat.com/ga</url>
            <releases>
                <enabled>true</enabled>
            </releases>
            <snapshots>
                <enabled>false</enabled>
            </snapshots>
        </repository>
        <repository>
            <id>redhat-ea-repository</id>

<url>https://maven.repository.redhat.com/earlyaccess/all</url>
            <releases>
                <enabled>true</enabled>
            </releases>
            <snapshots>
                <enabled>false</enabled>
            </snapshots>
        </repository>
        <repository>
          <id>jboss-public</id>
          <name>JBoss Public Repository Group</name>

<url>https://repository.jboss.org/nexus/content/groups/public/</url>
        </repository>
      </repositories>
      <pluginRepositories>
        <pluginRepository>
            <id>redhat-ga-repository</id>
            <url>https://maven.repository.redhat.com/ga</url>
            <releases>
                <enabled>true</enabled>
            </releases>
            <snapshots>
                <enabled>false</enabled>
            </snapshots>
        </pluginRepository>
        <pluginRepository>
            <id>redhat-ea-repository</id>

<url>https://maven.repository.redhat.com/earlyaccess/all</url>
            <releases>
                <enabled>true</enabled>
            </releases>
            <snapshots>
                <enabled>false</enabled>
            </snapshots>
        </pluginRepository>
        <pluginRepository>
          <id>jboss-public</id>
          <name>JBoss Public Repository Group</name>
```

```
<url>https://repository.jboss.org/nexus/content/groups/public</url>
        </pluginRepository>
      </pluginRepositories>
    </profile>
  </profiles>

  <activeProfiles>
    <activeProfile>extra-repos</activeProfile>
  </activeProfiles>

</settings>
```

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:

```
groupId:artifactId:version
groupId:artifactId:packaging:version
groupId:artifactId:packaging:classifier:version
```

Each coordinate can be explained as follows:

groupId

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

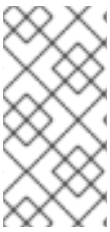
Enables you to distinguish between artifacts that were built from the same POM, but have different content.

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.

3.2. 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 1.7, 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>1.7</source>
          <target>1.7</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.

3.3. 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>
```

3.4. 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 4. DEPLOYING AN APACHE CXF WEB SERVICE

Abstract

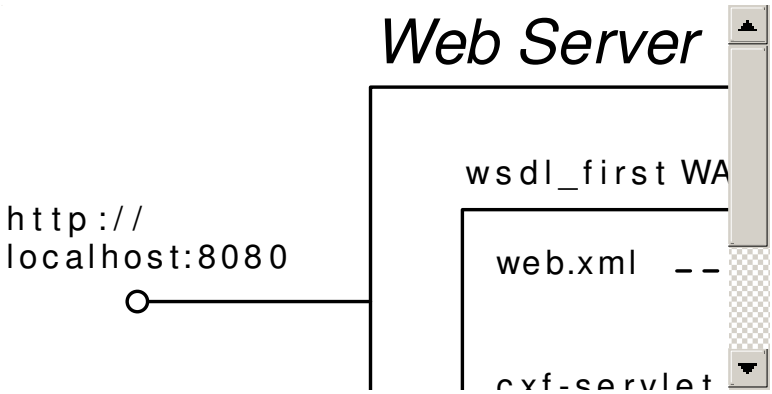
This tutorial describes how to deploy an Apache CXF Web services endpoint in a WAR file, where the Web service endpoint is implemented by binding directly to a Java class with the JAX-WS mapping.

4.1. APACHE CXF EXAMPLE

Overview

Figure 4.1, "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 4.1. 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 4.1, "web.xml File for the wsdl_first Example" shows the contents of the **web.xml** file.

Example 4.1. 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 4.2, "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 4.2. 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 4.3, “Address in the WSDL CustomerService Contract” .

Example 4.3. 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>
```

4.2. 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 Web Server, or

- JBoss Enterprise Application Platform

- Java version 1.7 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 3.1, "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-3.1.5.redhat-630187.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-3.1.5.redhat-630187.zip
   ```

2. Using a suitable archive utility on your platform, unzip the **apache-cxf-3.1.5.redhat-630187.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 Web Server (WS)*—open a new command prompt and execute the **startup.sh** script from the **tomcat8/bin/** directory (or the **tomcat7/bin/** directory, as appropriate). For more details about how to configure and launch the Web server, see the *Installation Guide* from the JBoss 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 Web Server (WS)*—copy the **wsdl_first.war** WAR file to the **tomcat8/webapps** directory (or **tomcat7/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 5. 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.

5.1. APACHE CAMEL SERVLET EXAMPLE

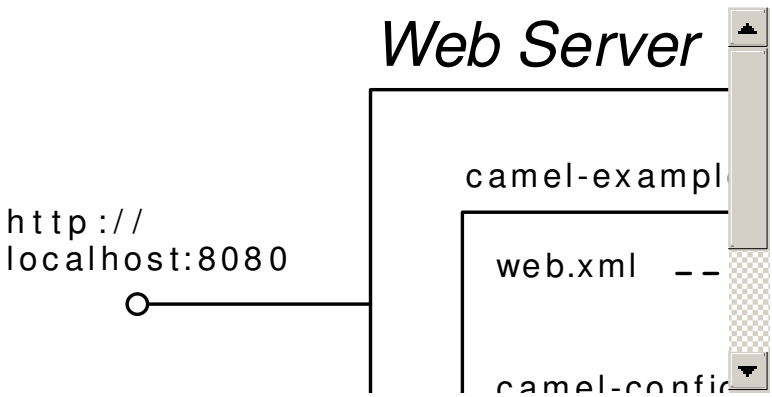
Overview

IMPORTANT

For deploying Apache Camel applications in JBoss EAP, consider using the *Camel on EAP* subsystem instead. The Camel on EAP subsystem is easier to use and provides tighter integration with the JBoss EAP container. For details, see Chapter 2, *Apache Camel on JBoss EAP*.

Figure 5.1, “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-2.17.0.redhat-630187/camel/hello`**.

Figure 5.1. 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 5.1, "web.xml File for the camel-example-servlet-tomcat Example" shows the contents of the **web.xml** file.

Example 5.1. 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-2.17.0.redhat-
630187/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 5.2, “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 5.2. 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-2.17.0.redhat-
630187/camel/hello
```

5.2. 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 Web Server, or

- JBoss Enterprise Application Platform

- Java version 1.7 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 3.1, "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-2.17.0.redhat-630187.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-2.17.0.redhat-630187.zip
   ```

2. Using a suitable archive utility on your platform, unzip the **apache-camel-2.17.0.redhat-630187.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-2.17.0.redhat-630187.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 Web Server (WS)*—open a new command prompt and execute the **`startup.sh`** script from the **`tomcat8/bin/`** directory (or the **`tomcat7/bin/`** directory, as appropriate). For more details about how to configure and launch the WS, see the *Installation Guide* from the JBoss 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-2.17.0.redhat-630187.war`** WAR file from the **`camel-example-servlet-tomcat/target`** directory to the Web server's deployment directory, as follows:

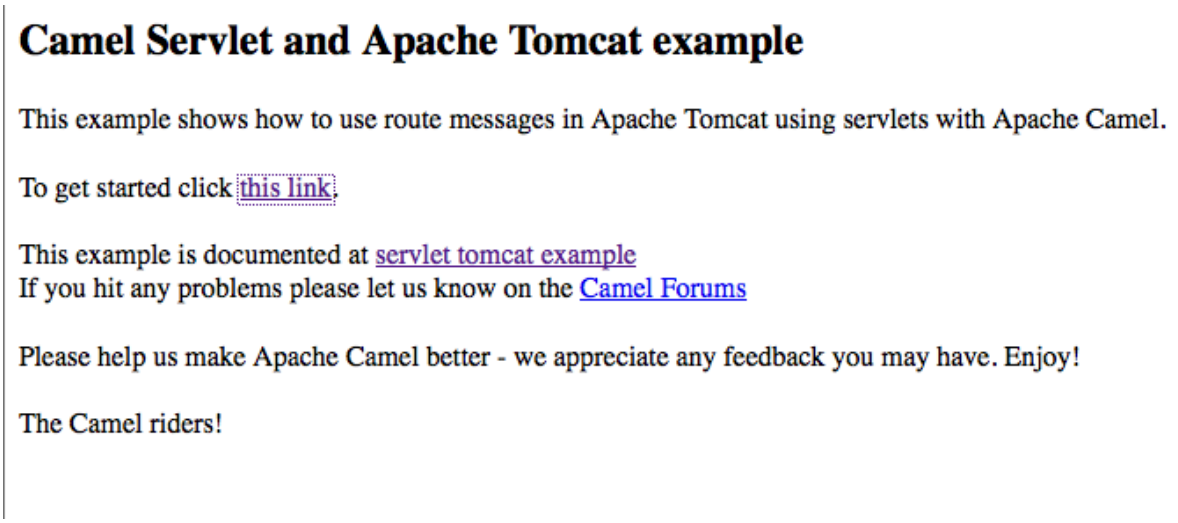
   - *JBoss Web Server (WS)*—copy the **`camel-example-servlet-tomcat-2.17.0.redhat-630187.war`** WAR file to the **`tomcat8/webapps`** directory (or **`tomcat7/webapps`** directory, as appropriate).

   - *JBoss Enterprise Application Platform (EAP)*—copy the **`camel-example-servlet-tomcat-2.17.0.redhat-630187.war`** WAR file to the **`standalone/deployments`** directory.

4. Navigate to the following URL in your browser:

   ```
   http://localhost:8080/camel-example-servlet-tomcat-2.17.0.redhat-630187/
   ```

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

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