Red Hat JBoss Fuse

Deploying into a Web Server

Version 6.1
December 2013

Deploying into a Web Server

Version 6.1

Updated: 05 Sep 2013
Copyright © 2011-2013 Red Hat, Inc. and/or its affiliates.

Trademark Disclaimer

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, MetaMatrix, Fedora, the Infinity Logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Apache, ServiceMix, Camel, CXF, and ActiveMQ are trademarks of Apache Software Foundation. Any other names contained herein may be trademarks of their respective owners.

Third Party Acknowledgements

One or more products in the Red Hat JBoss Fuse release includes third party components covered by licenses that require that the following documentation notices be provided:

- JLine (http://jline.sourceforge.net) jline:jline:jar:1.0

  License: BSD (LICENSE.txt) - Copyright (c) 2002-2006, Marc Prud'hommeaux `<mwp1@cornell.edu>`

  All rights reserved.

  Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

  - Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
  - Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
  - Neither the name of JLine nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR

SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- Stax2 API (http://woodstox.codehaus.org/StAX2) org.codehaus.woodstox:stax2-api:jar:3.1.1

  License: The BSD License (http://www.opensource.org/licenses/bsd-license.php)

  Copyright (c) <YEAR>, <OWNER> All rights reserved.

  Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

  - Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
  - Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- jibx-run - JiBX runtime (http://www.jibx.org/main-reactor/jibx-run) org.jibx:jibx-run:bundle:1.2.3

  License: BSD (http://jibx.sourceforge.net/jibx-license.html) Copyright (c) 2003-2010, Dennis M. Sosnoski.

  All rights reserved.

  Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

  - Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
  - Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
  - Neither the name of JiBX nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON

ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- JavaAssist (http://www.jboss.org/javassist) org.jboss.javassist:com.springsource.javassist:jar:3.9.0.GA:compile

  License: MPL (http://www.mozilla.org/MPL/MPL-1.1.html)

- HAPI-OSGI-Base Module (http://hl7api.sourceforge.net/hapi-osgi-base/) ca.uhn.hapi:hapi-osgi-base:bundle:1.2

  License: Mozilla Public License 1.1 (http://www.mozilla.org/MPL/MPL-1.1.txt)

Table of Contents

1. Overview of JBoss Fuse Deployment 11
WAR Build and Deployment Model 12
2. Building a WAR 15
Preparing to use Maven 16
Modifying an Existing Maven Project 20
Bootstrapping a CXF Servlet in a WAR 24
Bootstrapping a Spring Context in a WAR 26
3. Deploying an Apache CXF Web Service 27
Apache CXF Example 28
Deploy the Apache CXF Example 32
4. Deploying an Apache Camel Servlet Endpoint 35
Apache Camel Servlet Example 36
Deploy the Apache Camel Servlet 40
5. Deploying an Apache Camel WS Endpoint 43
Apache Camel CXF Example 44
Deploy the Apache Camel CXF Example 50
6. Deploying the Apache ActiveMQ Resource Adapter 53
Install the ActiveMQ Resource Adapter 54
Install JBoss AS Quickstarts 64
Build and Deploy the helloworld-mdb Example 69
Index 77

List of Figures

1.1. Building the WAR with Maven 12
3.1. Example Web Service Deployed in a Web Server 28
4.1. Camel Servlet Example Deployed in a Web Server 36
5.1. Camel CXF Example Deployed in a Web Server 44

List of Examples

2.1. Adding the Red Hat JBoss Fuse Repositories to Maven 16
3.1. web.xml File for the wsdl_first Example 28
3.2. Spring Configuration for the wsdl_first Example 30
3.3. Address in the WSDL CustomerService Contract 31
4.1. web.xml File for the camel-example-servlet-tomcat Example 36
4.2. Route Definition for the Camel Servlet Example 38
5.1. web.xml File for the camel-example-cxf-tomcat Example 45
5.2. Route Definitions in the CamelRoute Class 48
6.1. ActiveMQ Resource Adapter Configuration in standalone.xml 56
6.2. Message Driven Bean Configuration in standalone.xml 58
6.3. ActiveMQ Broker Configuration 59

Chapter 1. Overview of JBoss Fuse Deployment

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.

WAR Build and Deployment Model 12

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[1] as described in "Preparing to use Maven" on page 16.

Supported Web server platforms

The following Web server platforms are supported by JBoss Fuse:

- *JBoss Enterprise Web Server*
- *JBoss Enterprise Application Platform*

Build model

Figure 1.1 on page 12 shows an overview of the model for building WAR files for JBoss Fuse applications.

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[2] 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.

[1] https://maven.apache.org/
[2] https://maven.apache.org/

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

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

Preparing to use Maven 16
Modifying an Existing Maven Project 20
Bootstrapping a CXF Servlet in a WAR 24
Bootstrapping a Spring Context in a WAR 26

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[1].
- *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 on page 16.

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

```
<settings>
    <profiles>
```

[1] http://maven.apache.org/download.html

```
        <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-snap
shots/</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.

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. "Change the package type to WAR" on page 20.
2. "Customize the JDK compiler version" on page 20.
3. "Store resources under webapp/WEB-INF" on page 21.
4. "Customize the Maven WAR plug-in" on page 21.

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.

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 .

Bootstrapping a Spring Context in a WAR

Overview

You can bootstrap a Spring context in a WAR using Spring's ContextLoaderListener[2] 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 ht
tp://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>
```

[2] http://static.springsource.org/spring/docs/3.0.5.RELEASE/reference/web-integration.html

Chapter 3. Deploying an Apache CXF Web Service

REVISIT -

Apache CXF Example 28
Deploy the Apache CXF Example 32

Apache CXF Example

Overview

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 "Install Apache CXF" on page 32.

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 on page 28 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 on page 30 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.springframe
work.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 on page 31.

Example 3.3. Address in the WSDL CustomerService Contract

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

Deploy the Apache CXF Example

Overview

Prerequisites

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-610025.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-610025.zip
   ```

2. Using a suitable archive utility on your platform, unzip the `apache-cxf-2.7.0.redhat-610025.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://customerser
vice.example.com/" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:tns="http://cus
tomerservice.example.com/" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="ht
tp://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/CustomerService
Port"/>
    </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

Apache Camel Servlet Example 36
Deploy the Apache Camel Servlet 40

Apache Camel Servlet Example

Overview

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 "Install Apache Camel" on page 40.

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 on page 36 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 ht
tp://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</servlet-
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-610025/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 on page 38 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.springframe
work.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-610025/camel/hello
```

Deploy the Apache Camel Servlet

Overview

Prerequisites

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-610025.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-610025.zip
   ```

2. Using a suitable archive utility on your platform, unzip the `apache-camel-2.12.0.redhat-610025.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-610025.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-610025.war` WAR file from the `camel-example-servlet-tomcat/target` directory to the Web server's deployment directory, as follows:

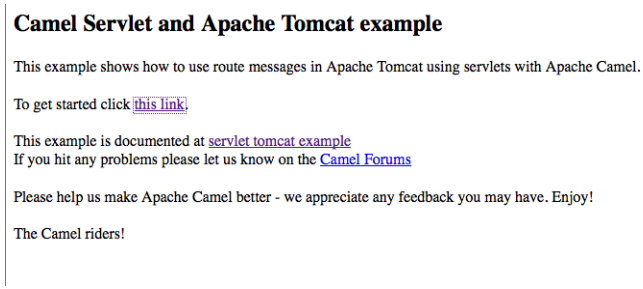
   - *JBoss Enterprise Web Server (EWS)*—copy the `camel-example-servlet-tomcat-2.12.0.redhat-610025.war` WAR file to the `tomcat7/webapps` directory (or `tomcat6/webapps` directory, as appropriate).

   - *JBoss Enterprise Application Platform (EAP)*—copy the `camel-example-servlet-tomcat-2.12.0.redhat-610025.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-610025/
   ```

   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

REVISIT -

Apache Camel CXF Example 44
Deploy the Apache Camel CXF Example 50

Apache Camel CXF Example

Overview

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 "Install Apache Camel" on page 50.

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 on page 45 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 on page 48 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).getIn
cidentId();

                    // 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.

Deploy the Apache Camel CXF Example

Overview

Prerequisites

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-610025.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-610025.zip
   ```

2. Using a suitable archive utility on your platform, unzip the `apache-camel-2.12.0.redhat-610025.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] ------------------------------------------------------------------------
```

Chapter 6. Deploying the Apache ActiveMQ Resource Adapter

This chapter explains how to install the Apache ActiveMQ resource adapter into JBoss Enterprise Application Platform and how to integrate ActiveMQ messaging into your applications, taking the `helloworld-mdb` demonstration as an example.

Install the ActiveMQ Resource Adapter 54
Install JBoss AS Quickstarts 64
Build and Deploy the helloworld-mdb Example 69

Install the ActiveMQ Resource Adapter

Overview

This section describes how to find, install, and configure the ActiveMQ resource adapter into a standalone instance of the JBoss Enterprise Application Platform.

A *resource adapter* is a kind of plug-in for a J2EE container. The J2EE standard defines the resource adapter framework, which makes it possible to expand the core J2EE container, adding new features and functionality. By installing the ActiveMQ resource adapter, you make it possible for message driven beans and servlets to communicate through an embedded Apache ActiveMQ broker instance. The ActiveMQ broker can thus be used as the underlying messaging system in the container.

Resource adapter location

You can find the ActiveMQ resource adapter archive file, `activemq-rar-5.9.0.redhat-610025.rar`, at either of the following locations:

- In the following sub-directory of the JBoss A-MQ installation:

  ```
  InstallDir/extras/lib/optional
  ```

- Directly from the JBoss Fuse Maven repository, at the following URL:

  ```
  http://repo.fusesource.com/nexus/content/groups/public/org/apache/activemq/activemq-rar/
  ```

  Download the `.rar` archive file from the appropriately versioned sub-directory, 5.9.0.redhat-610025.

Configuration files

The following configuration files are needed for the the ActiveMQ resource adapter (when installed in a standalone instance of the JBoss Enterprise Application Platform):

InstallDir`/standalone/configuration/standalone.xml`
: The `standalone.xml` file is the default (bare bones) configuration for the JBoss Enterprise Application Platform container. You must edit this file to complete the installation of the ActiveMQ resource adapter.

> **Note**
> It is assumed that this file does *not* already configure the HornetQ messaging system (which would conflict with the ActiveMQ messaging system).

InstallDir`/standalone/configuration/broker-config.xml`
: The `broker-config.xml` file specifies the detailed configuration of the embedded ActiveMQ broker. You must create this file when you install the ActiveMQ resource adapter.

> **Note**
> JBoss Enterprise Application Platform can be figured either as a standalone container, using `standalone/configuration/standalone.xml`, or as a managed domain, using `domain/configuration/domain.xml`. Throughout this section, we describe explicitly how to configure the standalone container, but it is understood that a similar approach could be used to configure a managed domain.

Steps to install the resource adapter

Perform the following steps to install the Apache ActiveMQ resource adapter into JBoss Enterprise Application Platform (assuming that you will be running the container in standalone mode):

1. The ActiveMQ resource adapter archive file can be found in the `extras/lib/optional` sub-directory of the JBoss A-MQ installation. Make a copy of the ActiveMQ resource adapter archive file, omitting the version number in the filename. For example, on a UNIX or Linux platform, you can rename the `activemq-rar-5.9.0.redhat-610025.rar` archive file as follows:

```
cd InstallDir/extras/lib/optional
cp activemq-rar-5.9.0.redhat-610025.rar activemq-rar.rar
```

> **Note**
> Renaming the resource adapter archive in this way is not strictly necessary. But because the resource adapter file name appears in the resource adapter configuration, using a versionless filename makes it easier to upgrade the resource adapter at a later date.

2. Install the ActiveMQ resource adapter by copying the resource adapter archive, `activemq-rar.rar`, to the relevant JBoss Enterprise Application Platform deployment directory. For example, on a UNIX or Linux platform, you could copy the resource adapter archive to a standalone JBoss Enterprise Application Platform as follows:

```
cp activemq-rar.rar EAPInstallDir/standalone/deployments/
```

3. Add the requisite resource adapter configuration to the `urn:jboss:domain:resource-adapters:1.1` subsystem in the JBoss Enterprise Application Platform configuration, as follows.

   Open the *`EAPInstallDir`*`/standalone/configuration/standalone.xml` file using a text editor and paste the `resource-adapter` element from Example 6.1 on page 56 into the `urn:jboss:domain:resource-adapters:1.1` subsystem, as a child of the `resource-adapters` element.

Example 6.1. ActiveMQ Resource Adapter Configuration in standalone.xml

```
<server xmlns="urn:jboss:domain:1.4">
    ...
    <profile>
        ...
        <subsystem xmlns="urn:jboss:domain:resource-adapters:1.1">
            <resource-adapters>
                <resource-adapter id="activemq-rar.rar">
                    <archive>
                        activemq-rar.rar
                    </archive>
                    <transaction-support>XATransaction</transaction-support>
                    <config-property name="UserName">
                        defaultUser
                    </config-property>
                    <config-property name="Password">
                        defaultPassword
                    </config-property>
                    <config-property name="ServerUrl">
                        vm://localhost
                    </config-property>
                    <config-property name="BrokerXmlConfig">
                        xbean:file:///usr/local/jboss-eap-6.1/standalone/configura
tion/broker-config.xml
                    </config-property>
                    <config-property name="UseInboundSession">
                        false
                    </config-property>
                    <connection-definitions>
                        <connection-definition
                              class-name="org.apache.activemq.ra.ActiveMQManagedConnec
tionFactory"
```

```
                              jndi-name="java:/ConnectionFactory"
                              enabled="true"
                              pool-name="ConnectionFactory">
                            <xa-pool>
                                <min-pool-size>1</min-pool-size>
                                <max-pool-size>20</max-pool-size>
                            </xa-pool>
                        </connection-definition>
                    </connection-definitions>
                    <admin-objects>
                        <admin-object
                              class-name="org.apache.activemq.command.ActiveMQQueue"
                              jndi-name="java:/queue/HELLOWORLDMDBQueue"
                              use-java-context="true"
                              pool-name="HELLOWORLDMDBQueue">
                            <config-property name="PhysicalName">
                                HELLOWORLDMDBQueue
                            </config-property>
                        </admin-object>
                        <admin-object
                              class-name="org.apache.activemq.command.ActiveMQTopic"
                              jndi-name="java:/topic/HELLOWORLDMDBTopic"
                              use-java-context="true"
                              pool-name="HELLOWORLDMDBTopic">
                            <config-property name="PhysicalName">
                                HELLOWORLDMDBTopic
                            </config-property>
                        </admin-object>
                    </admin-objects>
                </resource-adapter>
                ...
            </resource-adapters>
        </subsystem>
        ...
    </profile>
    ...
</server>
```

If your resource adapter archive filename differs from `activemq-rar.rar`, you must change the content of the `archive` element in the preceding configuration to match the name of your archive file.

The value of the `BrokerXmlConfig` property must be customized to point at the *actual* location of the `broker-config.xml` file (see Example 6.3 on page 59 for sample contents of this file). A good place to store the `broker-config.xml` file is under the JBoss Enterprise Application Platform `standalone/configuration/` directory.

> **Note**
> The JMS administrative objects defined in the `admin-objects` element are not strictly necessary. They are meant to serve as examples to show how you can define administrative objects for the ActiveMQ resource adapter (they are used later in the message-driven bean demonstration).

4. Add the requisite message driven bean configuration to the `urn:jboss:domain:ejb3:1.4` subsystem in the JBoss Enterprise Application Platform configuration.

   Open the *`EAPInstallDir`*`/standalone/configuration/standalone.xml` file using a text editor and paste the `mdb` element from Example 6.2 on page 58 into the `urn:jboss:domain:ejb3:1.4` subsystem.

   Example 6.2. Message Driven Bean Configuration in standalone.xml

```
<server xmlns="urn:jboss:domain:1.4">
    ...
    <profile>
        ...
        <subsystem xmlns="urn:jboss:domain:ejb3:1.4">
            ...
            <mdb>
                <resource-adapter-ref resource-adapter-name="activemq-rar"/>
                <bean-instance-pool-ref pool-name="mdb-strict-max-pool"/>
            </mdb>
            <pools>
                <bean-instance-pools>
                    <strict-max-pool name="slsb-strict-max-pool" max-pool-size="20" in
stance-acquisition-timeout="5" instance-acquisition-timeout-unit="MINUTES"/>
                    <strict-max-pool name="mdb-strict-max-pool" max-pool-size="20" in
stance-acquisition-timeout="5" instance-acquisition-timeout-unit="MINUTES"/>
                </bean-instance-pools>
            </pools>
            ...
        </subsystem>
        ...
    </profile>
    ...
</server>
```

5. Create the ActiveMQ broker configuration file at the location you previously specified using the `BrokerXmlConfig` property (typically, *`EAPInstallDir`*`/standalone/configuration/broker-config.xml`). Using a text editor, paste in the sample broker configuration shown in Example 6.3 on page 59.

Example 6.3. ActiveMQ Broker Configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
  xmlns="http://www.springframework.org/schema/beans"
  xmlns:amq="http://activemq.apache.org/schema/core"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframe
work.org/schema/beans/spring-beans-3.0.xsd
  http://activemq.apache.org/schema/core http://activemq.apache.org/schema/core/activemq-
core.xsd">

  <!--  shutdown hook is disabled as RAR classloader may be gone at shutdown -->
  <broker xmlns="http://activemq.apache.org/schema/core" useJmx="true" useShutdown
Hook="false">

     <managementContext>
       <!-- use appserver provided context instead of creating one,
            for jboss use: -Djboss.platform.mbeanserver -->
       <managementContext createConnector="false"/>
     </managementContext>

     <persistenceAdapter>
       <kahaDB directory="activemq-data/kahadb"/>
     </persistenceAdapter>

    <transportConnectors>
      <transportConnector uri="tcp://localhost:61616"/>
    </transportConnectors>

  </broker>
</beans>
```

It is recommended that you customize the location of the KahaDB data directory, by changing the value of the `directory` attribute of the `kahaDB` element in the `broker-config.xml` file. You should specify the KahaDB directory as an *absolute* pathname, otherwise the data directory is created relative to the directory where you launch the JBoss Enterprise Application Platform container (typically, `bin/activemq-data/kahadb`).

Note

Although not explicitly specified with a `transportConnector` element, the broker also implicitly supports a `vm://localhost` endpoint, which enables local connections to the broker through the Java Virtual Machine (this is much more efficient than TCP/IP).

6. Start the standalone instance of JBoss Enterprise Application Platform, For example, on a UNIX or Linux platform, you can start the standalone instance as follows:

```
cd EAPInstallDir/bin
./standalone.sh
```

Verify the installation with JConsole

You can verify the installation of the ActiveMQ resource adapter, using the Java JConsole utility, as follows:

1. Make sure that the standalone instance of JBoss Enterprise Application Platform is running. If necessary, start it from the command line, as follows (UNIX or Linux platform):

```
./standalone.sh
```

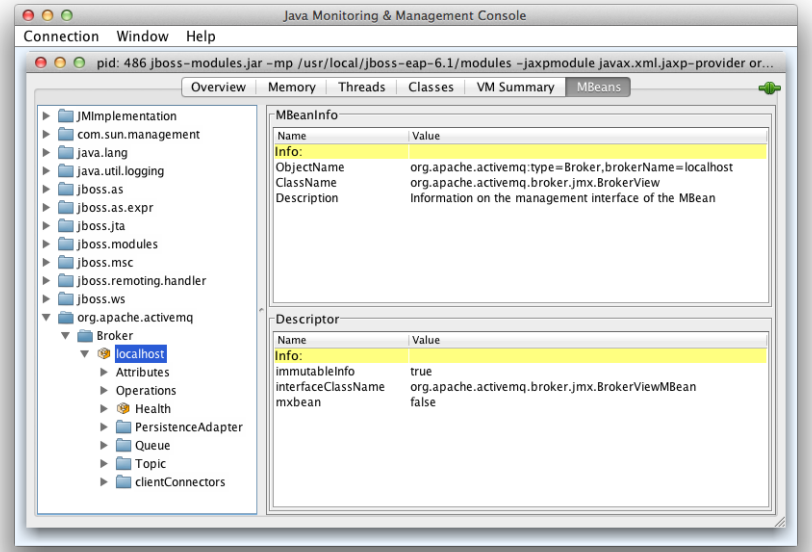
2. Open a new command prompt and start up the Java JConsole utility by entering the following command:

```
jconsole
```

3. The **JConsole: New Connection** dialog opens. In the **Local Process** list, select the `jboss-modules.jar` entry, as shown in the screeen shot, and then click **Connect**.

4. The **Java Monitoring & Management Console** window opens. In this window, select the **MBeans** tab and look for the `org.apache.activemq` folder. If the `org.apache.activemq` folder is *not* visible, this would indicate that the ActiveMQ broker is not running, and therefore there is a problem with the installation of the ActiveMQ resource adapter.

   If the `org.apache.activemq` folder is visible, you can drill down from this node to examine the MBeans in the ActiveMQ broker, as shown in the followin screen shot.



Resource adapter configuration

In the configuration shown in Example 6.1 on page 56, you use the `config-property` element to set resource adapter properties. The ActiveMQ resource adapter supports the following basic properties:

`UserName`
: *(Optional)* Specifies the client username when connecting to the ActiveMQ broker (not required in this example, because the ActiveMQ broker configuration does not enable authentication).

`Password`
: *(Optional)* Specifies the client password when connecting to the ActiveMQ broker (not required in this example, because the ActiveMQ broker configuration does not enable authentication).

`ServerUrl`
: Specifies the URL used to connect to the embedded ActiveMQ broker instance. This value must match one of the endpoints specified by a `transportConnector` element in the ActiveMQ broker configuration. It is strongly recommended that you use the `vm://localhost` URL, because this is the most efficient protocol for connecting to the broker.

`BrokerXmlConfig`
: Specifies the location of the embedded ActiveMQ broker's XML configuration file. To specify a location on the file system, use the format, `xbean:file:`*`//AbsolutePath`*, where the path, *`AbsolutePath`*, should be specified as an absolute pathname.

`UseInboundSession`
: *(Optional)* Sets a flag that specifies whether outbound connections should reuse the inbound connection's session for sending messages (useful for connections going through a firewall). Defaults to `false`.

`Clientid`
: *(Optional)* Specifies a JMS client ID, which the resource adapter uses for the connection from the JBoss Enterprise Application Platform container.

ActiveMQ broker configuration

Most of the options for customizing the ActiveMQ resource adapter are provided by the ActiveMQ broker configuration file, `broker-config.xml` (whose location is specified by the `BrokerXmlConfig` resource adapter property). This configuration file supports a huge range of features and settings which are beyond the scope of this guide. To start learning more about ActiveMQ broker configuration, please consult the following guides from the *JBoss A-MQ* documentation library:

- *Configuring Broker Persistence*
- *Tuning Guide*
- *Security Guide*
- *XML Configuration Reference*

Install JBoss AS Quickstarts

Overview

The JBoss AS Quickstart examples constitute a collection of demonstrations that illustrate a variety of features of the JBoss Enterprise Application Platform. The installation consists of the following parts:

- *JBoss EAP Maven repository*—an offline Maven repository for JBoss Enterprise Application Platform, which contains the dependencies required by the quickstart examples.
- *JBoss AS Quickstart examples*—the quickstart examples themselves.

Prerequisites

To download, install, and build the JBoss AS Quickstart examples, you need the following prerequisites:

- *Subscription*—you must have a Red Hat subscription that includes support for the Red Hat JBoss Enterprise Application Platform product (or ask Red Hat support for access as part of an evaluation).
- *Maven installation*—you must have Apache Maven[1] installed and the version must be 3.0.0 or later. You can get the latest copy of Maven from the Maven download page[2].
- *Internet access*—Maven is a distributed build system, which downloads packages from the Internet on the fly, whenever they are needed during a build. Consequently, you must have access to the Internet while performing a Maven build.

JBoss AS Quickstarts download location

You can download the JBoss AS Quickstart examples from the Quickstarts download page[3] on the Red Hat Customer Portal[4] site. Click the following link to download the `jboss-eap-6.1.0-quickstarts.zip` file:

- Red Hat JBoss Enterprise Application Platform 6.1.0 Quickstarts[5]

[1] https://www.apache.org/
[2] https://maven.apache.org/download.cgi
[3] https://access.redhat.com/jbossnetwork/restricted/softwareDetail.html?softwareId=21733&product=appplatform&version=6.1.0&downloadType=distributions
[4] https://access.redhat.com/downloads/
[5] https://access.redhat.com/jbossnetwork/restricted/softwareDownload.html?softwareId=21733

Note

After following this link, you will be prompted to log on to the Red Hat customer access portal. If you do not have a subscription for JBoss Enterprise Application Platform, you will not be able to access this download, however.

Maven repository download location

The JBoss Enterprise Application Platform Maven repository is required in order to run the quickstart examples.

You can download the Maven repository from the Maven Repository download page[6] on the Red Hat Customer Portal[7] site. Click the following link to download the `jboss-eap-6.1.0-maven-repository.zip` file:

- Red Hat JBoss Enterprise Application Platform 6.1.0 Maven Repository[8]

Steps to install JBoss AS Quickstarts

To install the JBoss AS Quickstart examples, perform the following steps:

1. Download the `jboss-eap-6.1.0-quickstarts.zip` file from the customer portal site. Use an archive utility to unzip the downloaded file at a convenient location on your filesystem, *`QuickInstallDir`*.

2. Download the `jboss-eap-6.1.0-maven-repository.zip` file from the customer portal site. Use an archive utility to unzip the downloaded file at a convenient location on your filesystem, *`MvnRepoInstallDir`*.

Note

It is essential to download and install the Maven repository on your local machine. The quickstart examples require Maven artifacts that are *not* available from any public repositories online. You will not be able to build the quickstart examples unless you download, install, and configure the Maven repository.

3. Configure Maven to use the downloaded Maven repository by editing your local repository's `settings.xml` file (usually located at `~/.m2/settings.xml` on Linux and UNIX systems, or at `C:\Documents and`

[6] https://access.redhat.com/jbossnetwork/restricted/softwareDetail.html?softwareId=21743&product=appplatform&version=6.1.0&downloadType=distributions
[7] https://access.redhat.com/downloads/
[8] https://access.redhat.com/jbossnetwork/restricted/softwareDownload.html?softwareId=21743

`Settings\Username\.m2\settings.xml` on Windows). Open the `settings.xml` file with a text editor and add the following profiles:

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

<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 ht
tp://maven.apache.org/xsd/settings-1.0.0.xsd">

  <profiles>

    <!-- Configure the JBoss EAP Maven repository -->
    <profile>
      <id>jboss-eap-repository</id>
      <repositories>
        <repository>
          <id>jboss-eap-repository</id>
          <url>file:///path/to/jboss-eap-6.1.0.GA-maven-repository</url>
          <releases>
            <enabled>true</enabled>
          </releases>
          <snapshots>
            <enabled>false</enabled>
          </snapshots>
        </repository>
      </repositories>
      <pluginRepositories>
        <pluginRepository>
          <id>jboss-eap-plugin-repository</id>
          <url>file:///path/to/jboss-eap-6.1.0.GA-maven-repository</url>
          <releases>
            <enabled>true</enabled>
          </releases>
          <snapshots>
            <enabled>false</enabled>
          </snapshots>
        </pluginRepository>
      </pluginRepositories>
    </profile>

    <!-- Configure the JBoss Community Maven repository -->
    <profile>
      <id>jboss-community-repository</id>
      <repositories>
        <repository>
          <id>jboss-community-repository</id>
          <url>http://repository.jboss.org/nexus/content/groups/public/</url>
          <releases>
```

```
            <enabled>true</enabled>
          </releases>
          <snapshots>
            <enabled>false</enabled>
          </snapshots>
        </repository>
      </repositories>
      <pluginRepositories>
        <pluginRepository>
          <id>jboss-community-plugin-repository</id>
          <url>http://repository.jboss.org/nexus/content/groups/public/</url>
          <releases>
            <enabled>true</enabled>
          </releases>
          <snapshots>
            <enabled>false</enabled>
          </snapshots>
        </pluginRepository>
      </pluginRepositories>
    </profile>

  </profiles>

  <activeProfiles>
    <!-- Optionally, make the repositories active by default -->
    <activeProfile>jboss-eap-repository</activeProfile>
    <activeProfile>jboss-community-repository</activeProfile>
  </activeProfiles>

</settings>
```

Note

Alternatively, there is a sample settings file provided at *`MvnRepoInstallDir`*`/example-settings.xml` in the downloaded Maven repository, which you can use as a template for defining your `settings.xml` file.

4. Replace all occurrences of `file:///path/to/jboss-eap-6.1.0.GA-maven-repository` in the `settings.xml` file with the actual location of the Maven repository on your filesystem, *`MvnRepoInstallDir`*.

Test the installation

To test the installation of the quickstart examples, try to build the `helloworld-mdb` example using Maven. Open a new command window, change directory to *`QuickInstallDir`*`/helloworld-mdb`, and enter the following command:

```
mvn clean package
```

If the project builds successfully, you should see a `BUILD SUCCESS` status and the generated `jboss-as-helloworld-mdb.war` package will be found under the *`QuickInstallDir`*`/target` directory.

If the project does not build successfully, make sure that you have access to the Internet and check that the Maven `settings.xml` file is correctly configured.

Build and Deploy the helloworld-mdb Example

Overview

In this tutorial, you will customize the `helloworld-mdb` quickstart example so that it works with the ActiveMQ resource adapter. You can then build and deploy the `helloworld-mdb` example into a standalone instance of JBoss Enterprise Application Platform (which already has an ActiveMQ resource adapter installed).

The `helloworld-mdb` example illustrates two kinds of integration with messaging: integration of message-driven beans; and integration of a servlet (which gets access to a JMS queue and a JMS topic using the standard JMS administered objects mechanism).

Prerequisites

The following prerequisites must be satisfied, before you can build and deploy the `helloworld-mdb` example:

- The ActiveMQ resource adapter is installed in the JBoss Enterprise Application Platform (as described in "Install the ActiveMQ Resource Adapter" on page 54), and the installation has been verified.
- The JBoss EAP Maven repository and the JBoss AS Quickstart examples have been installed (as described in "Install JBoss AS Quickstarts" on page 64).
- You have Internet access (for the Maven build).

Customizations

The version of the `helloworld-mdb` demonstration provided in the quickstarts distribution is integrated with the HornetQ messaging platform by default. To refactor the demonstration so that it integrates with Apache ActiveMQ, it is necessary to customize the following aspects of the `helloworld-mdb` code:

- Delete the HornetQ XML configuration file (located in `helloworld-mdb/src/webapp/WEB-INF/hornetq-jms.xml`).
- In `HelloWorldQueueMDB.java`, customize the annotations on the message driven bean to integrate with the ActiveMQ resource adapter.
- In `HelloWorldTopicMDB.java`, customize the annotations on the message driven bean to integrate with the ActiveMQ resource adapter.
- Add additional Maven dependencies.

These customizations are described in more detail in the rest of this section.

Steps to build and deploy the example

To build and deploy the quickstart `helloworld-mdb` example, perform the following steps:

1. Delete the following HornetQ XML configuration file from the `helloworld-mdb` project:

   ```
   helloworld-mdb/src/webapp/WEB-INF/hornetq-jms.xml
   ```

2. Edit the annotations on the `HelloWorldQueueMDB` message driven bean class, so that it integrates with the ActiveMQ resource adapter (instead of HornetQ). Edit the `HelloWorldQueueMDB.java` file at the following location:

   ```
   helloworld-mdb/src/main/java/org/jboss/as/quickstarts/mdb/HelloWorldQueueMDB.java
   ```

   Open the `HelloWorldQueueMDB.java` file using a text editor and make the modifications highlighted in the following extract:

   ```
   import org.jboss.ejb3.annotation.ResourceAdapter;
   ...
   @MessageDriven(name = "HelloWorldQueueMDB", activationConfig = {
     @ActivationConfigProperty(propertyName = "destinationType", propertyValue =
   "javax.jms.Queue"),
     @ActivationConfigProperty(propertyName = "destination", propertyValue = "HELLOWORLDM
   DBQueue"),
     @ActivationConfigProperty(propertyName = "acknowledgeMode", propertyValue = "Auto-ac
   knowledge") })
   @ResourceAdapter(value="activemq-rar.rar")
   public class HelloWorldQueueMDB implements MessageListener {
       ...
   ```

   Where the following changes are made to the code:

   - The `@ResourceAdapter` annotation explicitly associates the message driven bean with the ActiveMQ resource adapter. You *must* include this annotation, if you want to use the ActiveMQ resource adapter.
   - You need to add an `import` statement for the `@ResourceAdapter` annotation.
   - The value of the `destination` property is changed to `HELLOWORLDMDBQueue`, which is the *physical name* of the corresponding ActiveMQ queue that this message driven bean subscribes to. The physical name of the queue is the queue identifier used by the ActiveMQ broker.

> **Note**
> You must specify the queue's physical name here. In contrast to the case of HornetQ, the ActiveMQ messaging integration does *not* allow you to use a JNDI name for the `destination` value.

3. Edit the annotations on the `HelloWorldTopicMDB` message driven bean class, so that it integrates with the ActiveMQ resource adapter (instead of HornetQ). Edit the `HelloWorldTopicMDB.java` file at the following location:

   ```
   helloworld-mdb/src/main/java/org/jboss/as/quickstarts/mdb/HelloWorldTopicMDB.java
   ```

   Open the `HelloWorldTopicMDB.java` file using a text editor and make the modifications highlighted in the following extract:

   ```
   import org.jboss.ejb3.annotation.ResourceAdapter;
   ...
   @MessageDriven(name = "HelloWorldQTopicMDB", activationConfig = {
     @ActivationConfigProperty(propertyName = "destinationType", propertyValue =
   "javax.jms.Topic"),
     @ActivationConfigProperty(propertyName = "destination", propertyValue = "HELLOWORLDM
   DBTopic"),
     @ActivationConfigProperty(propertyName = "acknowledgeMode", propertyValue = "Auto-ac
   knowledge") })
   @ResourceAdapter(value="activemq-rar.rar")
   public class HelloWorldTopicMDB implements MessageListener {
       ...
   ```

4. Add the Maven dependency for the `jboss-ejb3-ext-api` artifact, which is needed for the `@ResourceAdapter` annotation. Open the `helloworld-mdb/pom.xml` file using a text editor and add the following `dependency` element as a child of the `dependencies` element in the POM file:

   ```
   <project ...>
       ...
       <dependencies>
           ...
           <dependency>
               <groupId>org.jboss.ejb3</groupId>
               <artifactId>jboss-ejb3-ext-api</artifactId>
               <!-- to get the right version, look in the EAP offline Maven repo -->
               <version>2.0.0-redhat-2</version>
               <scope>provided</scope>
           </dependency>
       </dependencies>
   ```

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

The `provided` scope value implies there is no need to embed the `jboss-ejb3-ext-api` JAR file in the generated WAR, because this library is already provided by the JBoss Enterprise Application Platform container.

Note

If you ever need to update the version of the `jboss-ejb3-ext-api` artifact, you can discover which version to use by drilling down to the `org/jboss/ejb3/jboss-ejb3-ext-api` sub-directory of the Maven repository you downloaded and installed in "Install JBoss AS Quickstarts" on page 64.

5. Build the `helloworld-mdb` example as follows. Open a new command prompt, change directory to the `helloworld-mdb` directory, and enter the following Maven command:

```
mvn clean package
```

If the build is successful, you should find the `jboss-as-helloworld-mdb.war` WAR file in the `helloworld-mdb/target` directory.

6. If you have not already done so, register the administered objects for the queues and topics used by the example, by editing the JBoss Enterprise Application Platform configuration.

In your JBoss Enterprise Application Platform installation, open the `standalone/configuration/standalone.xml` configuration file with a text editor, and add the following highlighted administered objects to the `activemq-rar.rar` resource adapter:

```
<server xmlns="urn:jboss:domain:1.4">
    ...
    <profile>
        ...
        <subsystem xmlns="urn:jboss:domain:resource-adapters:1.1">
            <resource-adapters>
                <resource-adapter id="activemq-rar.rar">
                    ...
                    <admin-objects>
                        <admin-object
                              class-name="org.apache.activemq.command.ActiveMQQueue"
                              jndi-name="java:/queue/HELLOWORLDMDBQueue"
                              use-java-context="true"
                              pool-name="HELLOWORLDMDBQueue">
                            <config-property name="PhysicalName">
                                HELLOWORLDMDBQueue
```

```
                            </config-property>
                        </admin-object>
                        <admin-object
                              class-name="org.apache.activemq.command.ActiveMQTopic"
                              jndi-name="java:/topic/HELLOWORLDMDBTopic"
                              use-java-context="true"
                              pool-name="HELLOWORLDMDBTopic">
                            <config-property name="PhysicalName">
                                HELLOWORLDMDBTopic
                            </config-property>
                        </admin-object>
                    </admin-objects>
                </resource-adapter>
            </resource-adapters>
        </subsystem>
        ...
    </profile>
    ...
</server>
```

Where the preceding configuration adds the following entries to the JNDI registry:

`java:/queue/HELLOWORLDMDBQueue`
: References a `javax.jms.Queue` object that connects to the `HELLOWORLDMDBQueue` ActiveMQ queue (where the queue name on the ActiveMQ broker is specified by the `PhysicalName` config property).

`java:/queue/HELLOWORLDMDBTopic`
: References a `javax.jms.Topic` object that connects to the `HELLOWORLDMDBTopic` ActiveMQ topic (where the topic name on the ActiveMQ broker is specified by the `PhysicalName` config property).

In the `helloworld-mdb` example, these administered objects are accessed by the servlet class, `HelloWorldMDBServletClient` (but *not* by the message driven bean classes). For example, the `HelloWorldMDBServletClient` class injects these JNDI entries, using the `@Resource` annotation, as follows:

```
import javax.annotation.Resource;
...
import javax.jms.ConnectionFactory;
...
import javax.jms.Queue;
import javax.jms.Topic;
...
public class HelloWorldMDBServletClient extends HttpServlet {
    ...
 @Resource(mappedName = "java:/ConnectionFactory")
 private ConnectionFactory connectionFactory;

 @Resource(mappedName = "java:/queue/HELLOWORLDMDBQueue")
```

```
private Queue queue;

@Resource(mappedName = "java:/topic/HELLOWORLDMDBTopic")
private Topic topic;
   ...
```

7. Deploy the `helloworld-mdb` example to the running Web server. Manually copy the `jboss-as-helloworld-mdb.war` WAR file from the `helloworld-mdb/target` directory to the Web server's deployment directory, `standalone/deployments`.

Access the helloworld-mdb service

You can now test the `helloworld-mdb` service, as follows:

1. If you have not already started the JBoss Enterprise Application Platform standalone container, do so by entering the following commands at the command line:

```
cd EAPInstallDir/bin
./standalone.sh
```

2. You should now be able to access the `helloworld-mdb` service from your browser, by navigating to the following URL:

```
http://localhost:8080/jboss-as-helloworld-mdb/HelloWorldMDBServletClient
```

When you navigate to the preceding URL, the servlet sends five messages to the `HelloWorldQueueMDB` message-driven bean. If you look at the container console window, you should see some output like the following:

```
14:41:20,739 INFO  [class org.jboss.as.quickstarts.mdb.HelloWorldQueueMDB] (default-threads
 - 7) Received Message from queue: This is message 5
14:41:20,739 INFO  [class org.jboss.as.quickstarts.mdb.HelloWorldQueueMDB] (default-threads
 - 5) Received Message from queue: This is message 3
14:41:20,739 INFO  [class org.jboss.as.quickstarts.mdb.HelloWorldQueueMDB] (default-threads
 - 4) Received Message from queue: This is message 2
14:41:20,741 INFO  [class org.jboss.as.quickstarts.mdb.HelloWorldQueueMDB] (default-threads
 - 6) Received Message from queue: This is message 4
14:41:20,742 INFO  [class org.jboss.as.quickstarts.mdb.HelloWorldQueueMDB] (default-threads
 - 3) Received Message from queue: This is message 1
```

These console messages are written by the `HelloWorldQueueMDB` message-driven bean, thus demonstrating that the messages have successfully propagated from the servlet, throught the ActiveMQ broker, to the message-driven bean.

3. To send messages to the `HelloWorldTopicMDB` message-driven bean, navigate to the following URL:

```
http://localhost:8080/jboss-as-helloworld-mdb/HelloWorldMDBServletClient?topic
```

When you navigate to the preceding URL, the servlet sends five messages to the `HelloWorldTopicMDB` message-driven bean. If you look at the container console window, you should see some output like the following:

```
14:53:02,464 INFO  [class org.jboss.as.quickstarts.mdb.HelloWorldTopicMDB] (default-threads
 - 9) Received Message from topic: This is message 2
14:53:02,466 INFO  [class org.jboss.as.quickstarts.mdb.HelloWorldTopicMDB] (default-threads
 - 10) Received Message from topic: This is message 3
14:53:02,468 INFO  [class org.jboss.as.quickstarts.mdb.HelloWorldTopicMDB] (default-threads
 - 8) Received Message from topic: This is message 1
14:53:02,471 INFO  [class org.jboss.as.quickstarts.mdb.HelloWorldTopicMDB] (default-threads
 - 11) Received Message from topic: This is message 4
14:53:02,472 INFO  [class org.jboss.as.quickstarts.mdb.HelloWorldTopicMDB] (default-threads
 - 12) Received Message from topic: This is message 5
```

These console messages are written by the `HelloWorldTopicMDB` message-driven bean, thus demonstrating that the messages have successfully propagated from the servlet, throught the ActiveMQ broker, to the message-driven bean.

Troubleshooting

Index