Community Experience Distilled

Advanced Java® EE Development with WildFly®

Your one-stop guide to developing Java® EE applications with the Eclipse IDE, Maven, and WildFly® 8.1

Deepak Vohra

[PACKT] PUBLISHING open source* community experience distilled

Advanced Java® EE Development with WildFly®

Your one-stop guide to developing Java® EE applications with the Eclipse IDE, Maven, and WildFly® 8.1

Deepak Vohra

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

Advanced Java® EE Development with WildFly®

First published: March 2015

Production reference: 1230315

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78328-890-8

`www.packtpub.com`

Credits

Author
Deepak Vohra

Reviewers
Miro Kopecky
Konstantinos A. Margaritis
Manu PK
Alexandre Arcanjo de Queiroz
Muhammad Rosli

Commissioning Editor
Kartikey Pandey

Acquisition Editor
Vinay Argekar

Content Development Editor
Rahul Nair

Technical Editors
Vijin Boricha
Parag Topre

Copy Editors
Sarang Chari
Janbal Dharmaraj

Project Coordinator
Suzanne Coutinho

Proofreaders
Paul Hindle
Clyde Jenkins

Indexer
Hemangini Bari

Production Coordinator
Aparna Bhagat

Cover Work
Aparna Bhagat

About the Author

Deepak Vohra is a consultant and a principal member of the NuBean software company. He is a Sun Certified Java Programmer (SCJP) and Web Component Developer (SCWCD) and has worked in the fields of XML and Java programming and J2EE for over 5 years. Deepak is the coauthor of the Apress book *Pro XML Development with Java Technology* and is the technical reviewer for the O'Reilly book *WebLogic: The Definitive Guide*.

Deepak was also the technical reviewer for the Course Technology PTR book *Ruby Programming for the Absolute Beginner*, and the technical editor for the Manning Publications book *Prototype and Scriptaculous in Action*. He is also the author of the Packt Publishing books *JDBC 4.0 and Oracle JDeveloper for J2EE Development*, *Processing XML documents with Oracle JDeveloper 11g*, *EJB 3.0 Database Persistence with Oracle Fusion Middleware 11g*, *Java 7 JAX-WS Web Services*, and *Java EE Development with Eclipse*.

About the Reviewers

Miro Kopecky has been a passionate JVM enthusiast since the moment he joined Sun Microsystems in 2002. Miro truly believes in distributed system design, concurrency, and parallel computing, which push the system performance to its limit without losing reliability and stability.

I would like to thank my family and my girlfriend, Tanja, for her big support when I was reviewing this book.

Konstantinos A. Margaritis has worked as a contractor since 2003 and has tackled all sorts of projects, from the very low level (C, C++, and ASM) to the very high level (Java EE/Enterprise). He is highly proficient in both C/C++ and Java, having written hundreds of thousands of lines in either language, and has used Jboss/Wildfly for many years in corporate projects. He is also taking interest in new languages, such as D, C#, Go, and Rust. His favorite subject is software optimization and also SIMD architectures (SSE/AVX, ARM NEON, PowerPC Altivec/VSX, and many more). He has been an official Debian developer since 1999 and has boostrapped the Debian armhf port. Among others, he has worked for Genesi, Linaro, and Collabora. He has written a user book for Debian (in Greek) and several smaller guides and is currently in the process of writing an SIMD book.

I would like to thank my wife, Chryssoula, and my amazing kids, Yiannis and Evangelos, who have shown incredible patience and love in everything I attempt.

Manu PK is a software architect at Schneider Electric, where he designs and develops applications using Java and related technologies. He blogs at `http://blog.manupk.com` on his experiments with technology and is a guest author on DZone. Manu contributes to the developer community by participating in technical discussion forums and speaks at various community events. His current interests include modern technical architectures, Polyglot Persistence, JVM, and Agile practices. You can reach him at `manu.pk@outlook.com` or via his LinkedIn profile at `http://in.linkedin.com/in/manupk`.

I would like to thank my family and my wife, Lakshmi, for her support and patience when I was reviewing this book.

Alexandre Arcanjo de Queiroz is a Brazilian software developer graduated from the Faculty of Technology of São Paulo, a renowned institution of his country. He has experience in developing applications using Java EE platform in the UNIX environment.

He is currently working for Geofusion. Geofusion is the leader in geomarketing in Brazil and offers an online platform called OnMaps, indispensable for companies seeking expansion and assistance in more accurate decision making.

I would like to thank my family who supported me at every moment of my life and my friends who believed in my potential.

Muhammad Rosli is a freelance hardware designer who completed his bachelor's degree in electrical and electronic engineering from the University of Canterbury. He has worked for different software contractors in telecommunication, construction, and education, serving the government and private sectors for over 2 years. His primary job involved interfacing data from hardware to the end user using web technology. Most of the designs utilize Java EE and Wildfly for mission-critical projects involving large data, such as e-commerce databases and scientific databases. He is currently active in checking documentation for open source projects and testing the example application provided.

Disclaimer

Table of Contents

Preface v

Chapter 1: Getting Started with EJB 3.x 1

Setting up the Environment 2
Creating a WildFly runtime 3
Creating a Java EE project 6
Configuring a data source with MySQL database 10
Creating entities 14
Creating a JPA persistence configuration file 19
Creating a session bean facade 20
Creating a JSP client 22
Configuring the jboss-ejb3-ejb subproject 25
Configuring the jboss-ejb3-web subproject 27
Configuring the jboss-ejb3-ear subproject 28
Deploying the EAR module 30
Running the JSP client 34
Configuring a Java EE 7 Maven project 34
Summary 36

Chapter 2: Developing Object/Relational Mapping with Hibernate 4 37

Setting up the environment 38
Creating a Java EE web project 39
Creating a Hibernate XML Mapping file 42
Creating a properties file 48
Creating a Hibernate configuration file 51
Creating JSPs for CRUD 56
Creating the JavaBean class 58
Exporting schema 61
Creating table data 63

Retrieving table data 65
Updating a table row 67
Deleting a table row 68
Installing the Maven project 70
Running a schema export 73
Creating table rows 76
Retrieving table data 76
Updating the table 77
Deleting the table row 78
Summary 79

Chapter 3: Developing JSF 2.x Facelets 81

Setting up the environment 82
Creating a Java EE web project 84
Creating a managed bean 89
Creating a Facelets template 97
Creating header and footer 103
Creating input and output Facelets composition pages 106
Creating a web descriptor 110
Deploying the web project with Maven 114
Running the Facelets application 120
Summary 121

Chapter 4: Using Ajax 123

Setting up the environment 123
Creating a Java EE web project 125
Creating a user interface 129
Creating a servlet 139
Deploying the Ajax application with Maven 147
Running the Ajax application 154
Summary 159

Chapter 5: Using GWT 161

Setting up the environment 162
Running the starter project on WildFly 8.1 165
Creating a GWT project in Eclipse 167
Deploying the starter project to WildFly 8.1 172
Running the starter project on WildFly 8.1 176
Creating a new GWT project 177
Creating a GWT module 183
Creating an entry-point class 187
Creating an HTML host page 195

Deploying the GWT project with Maven 199
Running the GWT project 203
Summary 208
Chapter 6: Developing a JAX-WS 2.2 Web Service 209
Setting up the environment 210
Creating a Java EE web project 211
Creating a web descriptor 215
Creating a JAX-WS web service 217
Creating a web service client 224
Deploying the JAX-WS application with Maven 228
Running the JAX-WS application 238
Summary 238
Chapter 7: Developing a JAX-RS 1.1 Web Service 239
Setting up the environment 240
Creating a Java EE web project 241
Creating a sample RESTful web service 244
Deploying the RESTful web service 252
Running the RESTful web service 261
Creating a Java client 263
Running the Java client 266
Creating a JSP client 269
Running the JSP client 272
Summary 273
Chapter 8: Using Spring MVC 4.1 275
Setting up the environment 276
Creating a Spring MVC project 277
Creating a JPA configuration file 281
Creating the Model 282
Creating the Data Access Object design pattern 288
Creating a web descriptor 291
Creating a request handler 292
Creating the View 295
Creating a Spring MVC context 299
Deploying the Spring project with Maven 300
Creating deployment structure and infrastructure deployment descriptors 308
Installing the Spring MVC Maven project 312
Running the Spring MVC application 314
Summary 315

Chapter 9: Using JAX-RS 2.0 in Java EE 7 with RESTEasy **317**

Setting up the environment **318**
Creating a Java EE web project **318**
The Client API **329**
Creating a client instance 329
Accessing a resource 330
Setting a query parameter 335
Setting a template parameter 335
Filters and interceptors **336**
Creating a client filter 337
Creating a container filter **342**
Asynchronous processing **345**
Suspended response 347
Resuming request processing 349
Resuming a request with a suspend timeout handler 350
Cancelling a request 354
Session bean EJB resource **355**
Making an asynchronous call from the client **355**
Fixing a common issue **357**
Summary **358**

Chapter 10: Processing JSON with Java EE 7 **359**

Setting up the environment **359**
Creating a Maven project **360**
Creating JSON **366**
Parsing JSON **373**
Processing JSON in a RESTful web service **377**
JsonArray as a return type in the resource method for a GET request 378
Encoding a JsonObject object for a GET request 381
JsonObject as a parameter type in the resource method for a POST request 384
Summary **388**

Index **389**

Preface

WildFly is the new name for JBoss Application Server (JBoss AS) starting with version 8.0.0. WildFly provides high startup speed, low memory overhead, a high-performance, scalable web server, customizable runtime based on pluggable subsystems, and support for Java EE 7 and the latest web development standards. In past references to the server, "JBoss" is still used, but for subsequent references, "WildFly" is used. According to the Developer Productivity Report 2012, referred to subsequently as the "2012 report", JBoss is the most commonly used application server, with 28 percent of Java developers or organizations using the application server, more than any other application server. The same report indicated that 67 percent of Java developers use the Maven build tool, more than any other build tool. Eclipse is used by 68 percent of Java developers. The Java Tools and Technologies Landscape for 2014 report (referred to subsequently as the "2014 report") indicates that JBoss is the most commonly used application server in both development (16 percent) and production (17 percent). In the book, we discuss developing Java EE applications using WildFly 8.1.0, Maven 3, and Eclipse IDE. The book combines the most commonly used tools for Java EE development: WildFly, Maven, and Eclipse IDE.

The book is based on the Java EE standards 5, 6, and 7. We will discuss the commonly used technologies and frameworks JAX-RS 1.1, JSF 2.0, JPA 2.0, JAX-WS 2.2, EJB 3.0, Hibernate 4, Ajax, GWT 2.4, and Spring 3.1. The new Java EE7 support for JAX-RS 2.0 is discussed with RESTEasy. The new Java EE 7 feature for processing JSON is also discussed.

While several books on WildFly administration are available, none on Java EE application development with WildFly are available. WildFly is the most commonly used application server with support for all the commonly used Java EE technologies and frameworks. WildFly is efficient, lightweight, and modular, and provides a flexible deployment structure. JBoss Tools provides a set of plugins with support for WildFly, Maven, and Java EE frameworks such as JSF. Maven is the most commonly used build tool for compiling and packaging a Java EE application based on a project-object model (POM). Maven provides dependency management. The Eclipse IDE for Java EE developers is the most commonly used Java EE IDE.

The objective of the book is to discuss how a Java EE developer would develop applications with WildFly using Maven as the build tool and Eclipse IDE as the development environment. The book covers all aspects of application development, including the following topics:

- Setting the environment for an application
- Creating sample data
- Running a sample application

What this book covers

In *Chapter 1*, *Getting Started with EJB 3.x*, we discuss developing an EJB 3.0/JPA-based application with WildFly 8.1.0. According to the 2012 report, JPA (at 44 percent) and EJB 3.0 (at 23 percent) are the two most commonly used Java EE standards.

In *Chapter 2*, *Developing Object/Relational Mapping with Hibernate 4*, we discuss using Hibernate 4 with WildFly 8.1.0. According to the 2012 report, Hibernate (at 54 percent) is one of the most commonly used application frameworks. According to the 2014 report, Hibernate (at 67.5 percent) is the top object/relational mapping framework.

In *Chapter 3*, *Developing JSF 2.x Facelets*, we discuss using JSF 2.0 with WildFly 8.1.0. According to the 2012 report, JSF (at 23 percent) is the second most commonly used web framework. According to the 2014 report also, JSF is ranked second (at 21 percent) among "web frameworks in use".

In *Chapter 4*, *Using Ajax*, we discuss developing an Ajax application with WildFly 8.1.0. Ajax is a trend started in 2004-2005 that makes use of a web technique to transfer data between a browser and a server asynchronously.

In *Chapter 5*, *Using GWT*, we use Google Web Toolkit to develop an application with WildFly. According to both, the 2012 report and the 2014 report, GWT is one of the top four web frameworks.

In *Chapter 6*, *Developing a JAX-WS 2.2 Web Service*, we discuss developing an application based on the JAX-WS 2.2 standard in the Eclipse IDE using the Maven build tool. We deploy and run the application on WildFly 8.1.0.

In *Chapter 7*, *Developing a JAX-RS 1.1 Web Service*, we discuss developing a web service based on the JAX-RS 1.1 standard in the Eclipse IDE using Maven as the build tool. We deploy and run the application on WildFly 8.1.0.

In *Chapter 8*, *Using Spring MVC 4.1*, we discuss using Spring MVC with WildFly 8.1.0. According to the 2012 report, Spring MVC (at 30 percent) is the most commonly used web framework. In the 2014 report also, Spring MVC (at 40 percent) is the most commonly used web framework.

In *Chapter 9*, *Using JAX-RS 2.0 in Java EE 7 with RESTEasy*, we introduce the support for JAX-RS 2.0: the Java API for RESTful Web Services added to Java EE 7. We discuss the new Client API introduced in JAX-RS 2.0. We also discuss the asynchronous processing feature of JAX-RS 2.0.

In *Chapter 10*, *Processing JSON with Java* EE *7*, we introduce another new feature in Java EE 7, that is, the support for JSR 353: Java API for JSON Processing.

What you need for this book

We have used WildFly 8.1.0 in the book. Download WildFly 8.1.0 Final from `http://wildfly.org/downloads/`. In some of the chapters, we have used MySQL 5.6 Database-Community Edition, which can be downloaded from `http://dev.mysql.com/downloads/mysql/`. You also need to download and install the Eclipse IDE for Java EE Developers from `http://www.eclipse.org/downloads/`. Eclipse Luna 4.4.1 is used, but a later version can also be used. Also, install JBoss Tools (version 4.2.0 used) as a plugin to Eclipse. Apache Maven (version 3.05 or later) is also required to be installed and can be downloaded from `http://maven.apache.org/download.cgi`. We have used Windows OS, but if you have Linux installed, the book can still be used (though the source code and samples have not been tested with Linux). Slight modifications may be required with the Linux install; for example, the directory paths on Linux would be different than the Windows directory paths. You also need to install Java for Java-based chapters; Java SE 7 is used in the book.

Who this book is for

The target audience of the book is Java EE application developers. You might already be using JBoss or WildFly, but don't use the Eclipse IDE or Maven for development. The book introduces you to how the Eclipse IDE and Maven facilitate the development of Java EE applications with WildFly 8.1.0. This book is suitable for professional Java EE developers as well as beginners. The book is also suitable for an intermediate/advanced-level course in Java EE development with WildFly 8.1.0. The target audience is expected to have prior, albeit beginner-intermediate level, knowledge about the Java language and the Java EE technologies used in the book. The book also requires some familiarity with the Eclipse IDE.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Configuring the `jboss-ejb3-ejb` subproject."

A block of code is set as follows:

```
<module xmlns="urn:jboss:module:1.1" name="mysql" slot="main">
  <resources>
    <resource-root path="mysql-connector-java-5.1.33-bin.jar"/>
  </resources>
  <dependencies>
    <module name="javax.api"/>
  </dependencies>
</module>
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Select **Window** | **Preferences** in Eclipse. In **Preferences**, select **Server** | **Runtime Environment**."

[Warnings or important notes appear in a box like this.]

[Tips and tricks appear like this.]

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message. If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Getting Started with EJB 3.x

The objective of the EJB 3.x specification is to simplify its development by improving the EJB architecture. This simplification is achieved by providing metadata annotations to replace XML configuration. It also provides default configuration values by making entity and session beans **POJOs** (**Plain Old Java Objects**) and by making component and home interfaces redundant. The EJB 2.x entity beans is replaced with EJB 3.x entities. EJB 3.0 also introduced the **Java Persistence API** (**JPA**) for object-relational mapping of Java objects.

WildFly 8.x supports EJB 3.2 and the JPA 2.1 specifications from Java EE 7. While EJB 3.2 is supported, the sample application in this chapter does not make use of the new features of EJB 3.2 (such as the new TimerService API and the ability to disable passivation of stateful session beans). The sample application is based on Java EE 6 and EJB 3.1. The configuration of EJB 3.x with Java EE 7 is also discussed, and the sample application can be used or modified to run on a Java EE 7 project. We have used a Hibernate 4.3 persistence provider. Unlike some of the other persistence providers, the Hibernate persistence provider supports automatic generation of relational database tables, including the joining of tables.

In this chapter, we will create an EJB 3.x project and build and deploy this project to WildFly 8.1 using Maven. This chapter has the following sections:

- Setting up the environment
- Creating a WildFly runtime
- Creating a Java EE project
- Configuring a data source with MySQL database
- Creating entities
- Creating a JPA persistence configuration file
- Creating a Session Bean Facade
- Creating a JSP client

- Configuring the `jboss-ejb3-ejb` subproject
- Configuring the `jboss-ejb3-web` subproject
- Configuring the `jboss-ejb3-ear` subproject
- Deploying the EAR Module
- Running the JSP Client
- Configuring a Java EE 7 Maven Project

Setting up the Environment

We need to download and install the following software:

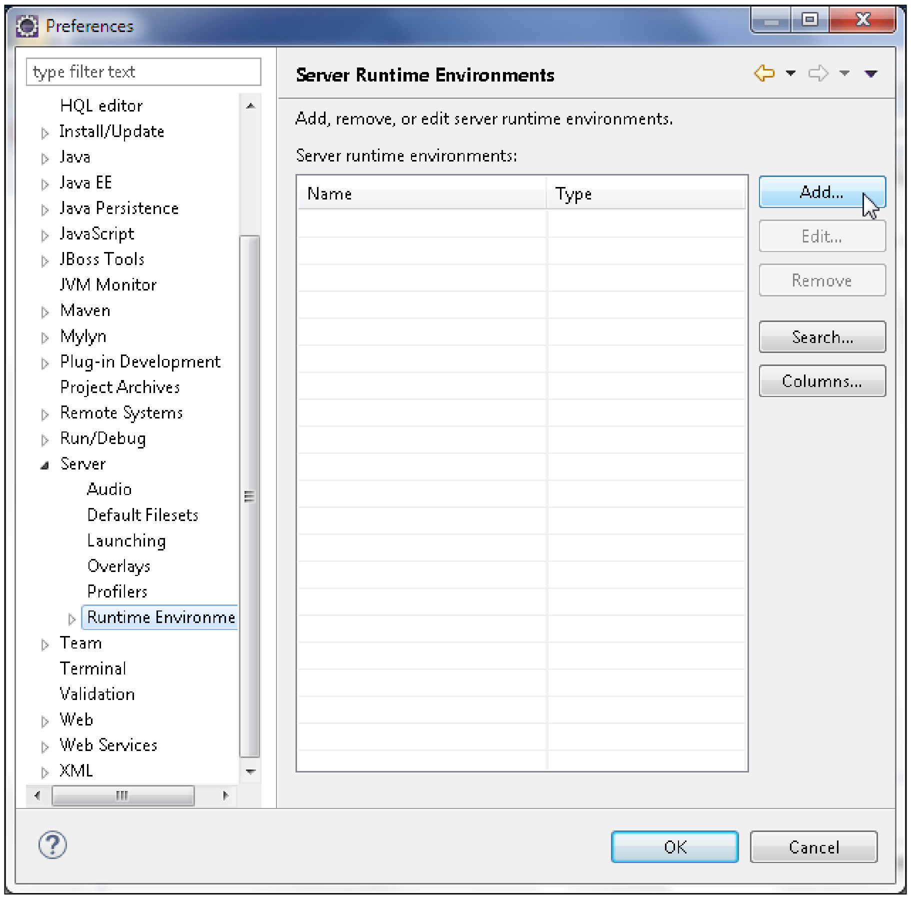
- **WildFly 8.1.0.Final**: Download `wildfly-8.1.0.Final.zip` from `http://wildfly.org/downloads/`.
- **MySQL 5.6 Database-Community Edition**: Download this edition from `http://dev.mysql.com/downloads/mysql/`. When installing MySQL, also install **Connector/J**.
- **Eclipse IDE for Java EE Developers**: Download Eclipse Luna from `https://www.eclipse.org/downloads/packages/release/Luna/SR1`.
- **JBoss Tools (Luna) 4.2.0.Final**: Install this as a plug-in to Eclipse from the Eclipse Marketplace (`http://tools.jboss.org/downloads/installation.html`). The latest version from Eclipse Marketplace is likely to be different than 4.2.0.
- **Apache Maven**: Download version 3.05 or higher from `http://maven.apache.org/download.cgi`.
- **Java 7**: Download Java 7 from `http://www.oracle.com/technetwork/java/javase/downloads/index.html?ssSourceSiteId=ocomcn`.

Set the environment variables: `JAVA_HOME`, `JBOSS_HOME`, `MAVEN_HOME`, and `MYSQL_HOME`. Add `%JAVA_HOME%/bin`, `%MAVEN_HOME%/bin`, `%JBOSS_HOME%/bin`, and `%MYSQL_HOME%/bin` to the `PATH` environment variable. The environment settings used are `C:\wildfly-8.1.0.Final` for `JBOSS_HOME`, `C:\Program Files\MySQL\MySQL Server 5.6.21` for `MYSQL_HOME`, `C:\maven\apache-maven-3.0.5` for `MAVEN_HOME`, and `C:\Program Files\Java\jdk1.7.0_51` for `JAVA_HOME`. Run the `add-user.bat` script from the `%JBOSS_HOME%/bin` directory to create a user for the WildFly administrator console. When prompted **What type of user do you wish to add?**, select **a) Management User**. The other option is **b) Application User**.

Management User is used to log in to **Administration Console**, and **Application User** is used to access applications. Subsequently, specify the **Username** and **Password** for the new user. When prompted with the question, **Is this user going to be used for one AS process to connect to another AS..?**, enter the answer as `no`. When installing and configuring the MySQL database, specify a password for the root user (the password `mysql` is used in the sample application).

Creating a WildFly runtime

As the application is run on WildFly 8.1, we need to create a runtime environment for WildFly 8.1 in Eclipse. Select Window | Preferences in Eclipse. In Preferences, select Server | Runtime Environment. Click on the Add button to add a new runtime environment, as shown in the following screenshot:



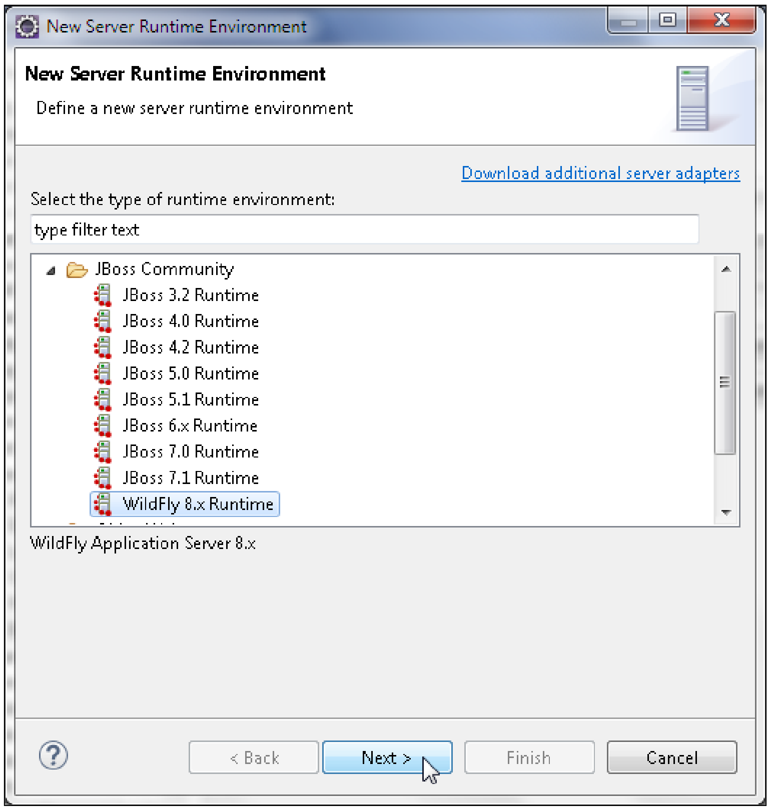
Server Runtime Environments

Add, remove, or edit server runtime environments.

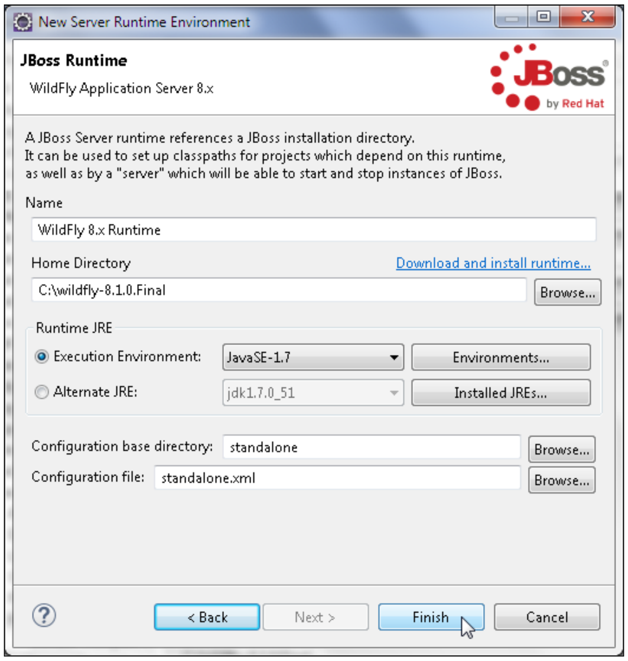
Server runtime environments:

Name	Type

In **New Server Runtime Environment**, select **JBoss Community | WildFly 8.x Runtime**. Click on **Next**:



In **WildFly Application Server 8.x**, which appears below **New Server Runtime Environment**, specify a Name for the new runtime or choose the default name, which is `WildFly 8.x Runtime`. Select the Home Directory for the WildFly 8.x server using the Browse button. The Home Directory is the directory where WildFly 8.1 is installed. The default path is C:\wildfly-8.1.0.Final for this chapter and subsequent chapters. Select the Runtime JRE as `JavaSE-1.7`. If the JDK location is not added to the runtime list, first add it from the JRE preferences screen in Eclipse. In **Configuration base directory**, select `standalone` as the default setting. In Configuration file, select `standalone.xml` as the default setting. Click on Finish:


New Server Runtime Environment

JBoss Runtime

WildFly Application Server 8.x

A JBoss Server runtime references a JBoss installation directory.
It can be used to set up classpaths for projects which depend on this runtime,
as well as by a "server" which will be able to start and stop instances of JBoss.

Name: WildFly 8.x Runtime

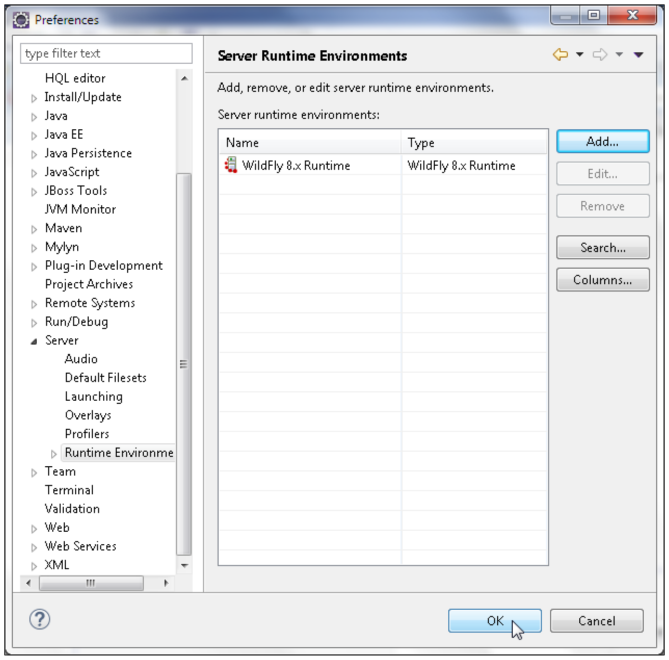
Home Directory: C:\wildfly-8.1.0.Final

Runtime JRE
- Execution Environment: JavaSE-1.7
- Alternate JRE: jdk1.7.0_51

Configuration base directory: standalone

Configuration file: standalone.xml

A new server runtime environment for WildFly 8.x Runtime gets created, as shown in the following screenshot. Click on **OK**:


Preferences

Server Runtime Environments

Add, remove, or edit server runtime environments.

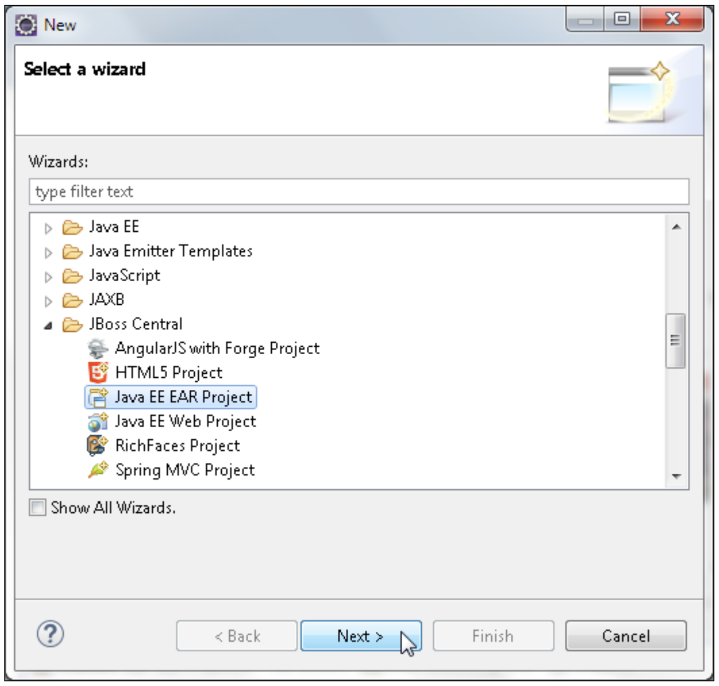
Server runtime environments:

Name	Type
WildFly 8.x Runtime	WildFly 8.x Runtime

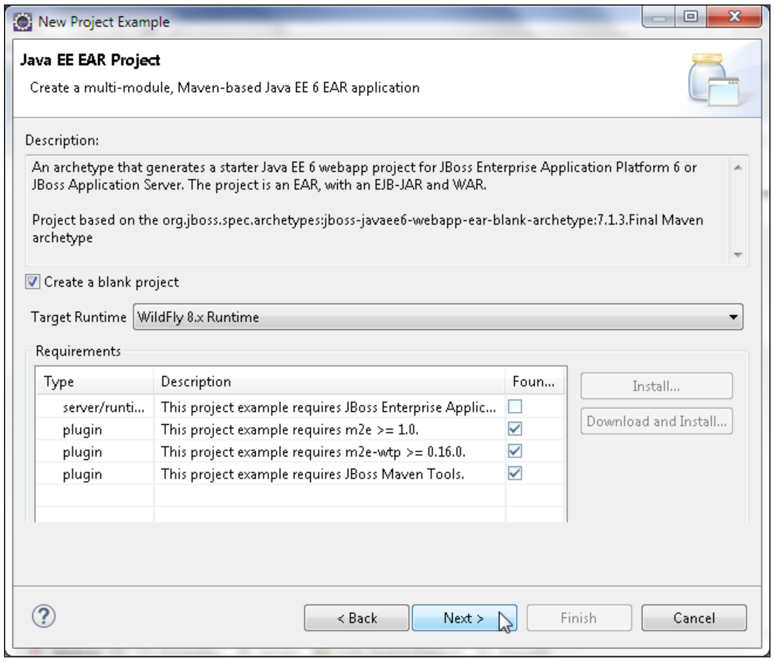
Creating a **Server Runtime Environment** for WildFly 8.x is not a prerequisite for creating a Java EE project in Eclipse. In the next section, we will create a new Java EE project for an EJB 3.x application.

Creating a Java EE project

JBoss Tools provides project templates for different types of JBoss projects. In this section, we will create a Java EE project for an EJB 3.x application. Select **File | New | Other** in Eclipse IDE. In the **New** wizard, select the **JBoss Central | Java EE EAR Project** wizard. Click on the **Next** button:



The **Java EE EAR Project** wizard gets started. By default, a Java EE 6 project is created. A Java EE EAR Project is a Maven project. The **New Project Example** window lists the requirements and runs a test for the requirements. The JBoss AS runtime is required and some plugins (including the JBoss Maven Tools plugin) are required for a Java EE project. Select **Target Runtime** as `WildFly 8.x Runtime`, which was created in the preceding section. Then, check the **Create a blank project** checkbox. Click on the **Next** button:



Java EE EAR Project

Create a multi-module, Maven-based Java EE 6 EAR application

Description:

An archetype that generates a starter Java EE 6 webapp project for JBoss Enterprise Application Platform 6 or JBoss Application Server. The project is an EAR, with an EJB-JAR and WAR.

Project based on the org.jboss.spec.archetypes:jboss-javaee6-webapp-ear-blank-archetype:7.1.3.Final Maven archetype

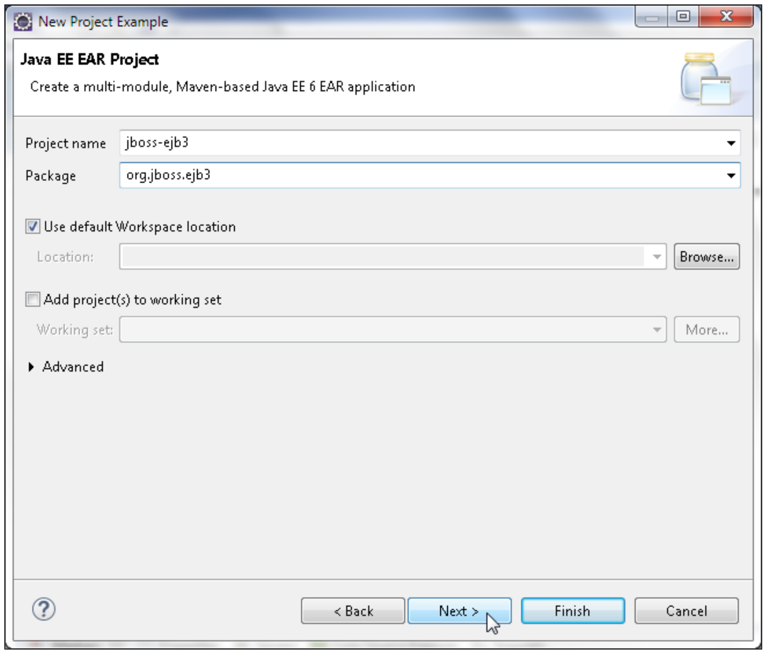
☑ Create a blank project

Target Runtime: WildFly 8.x Runtime

Requirements

Type	Description	Foun...
server/runti...	This project example requires JBoss Enterprise Applic...	☐
plugin	This project example requires m2e >= 1.0.	☑
plugin	This project example requires m2e-wtp >= 0.16.0.	☑
plugin	This project example requires JBoss Maven Tools.	☑

Specify **Project name** as `jboss-ejb3`, **Package** as `org.jboss.ejb3`, and tick the **Use default Workspace location** box. Click on the **Next** button:



Java EE EAR Project

Create a multi-module, Maven-based Java EE 6 EAR application

Project name: jboss-ejb3

Package: org.jboss.ejb3

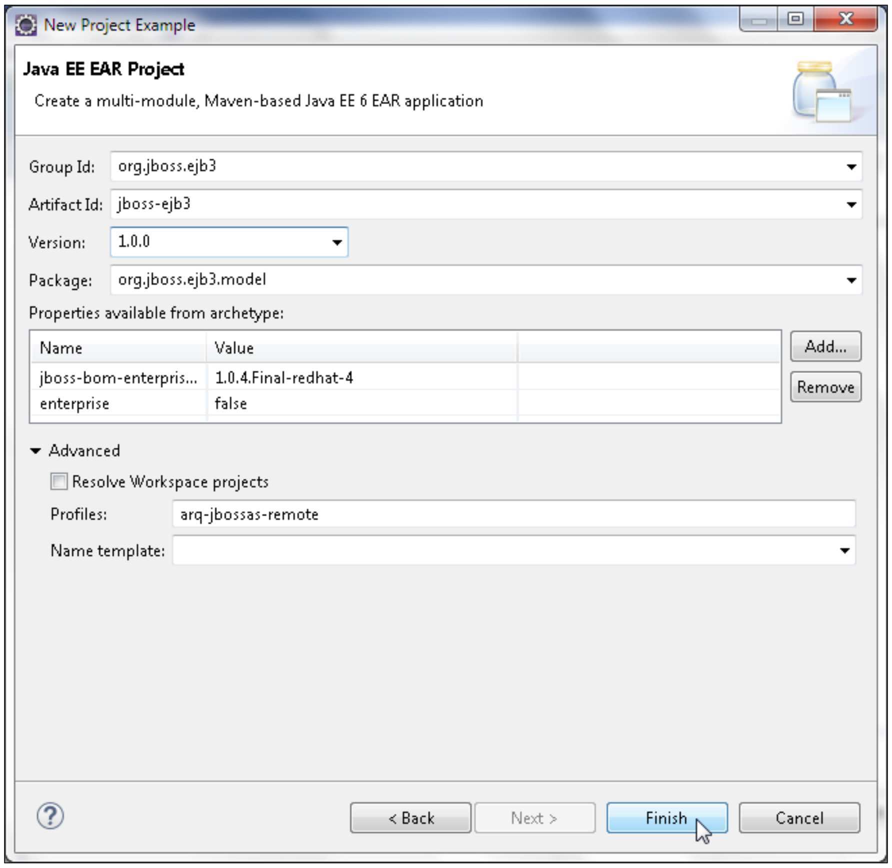
☑ Use default Workspace location

Location:

☐ Add project(s) to working set

Working set:

Specify Group Id as `org.jboss.ejb3`, Artifact Id as `jboss-ejb3`, Version as `1.0.0`, and Package as `org.jboss.ejb3.model`. Click on Finish:

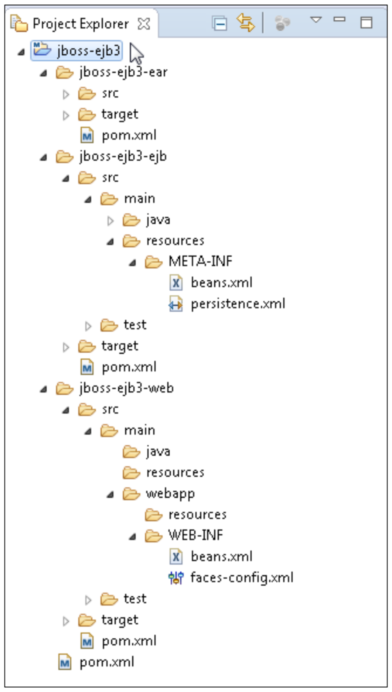


Group Id: org.jboss.ejb3
Artifact Id: jboss-ejb3
Version: 1.0.0
Package: org.jboss.ejb3.model
Properties available from archetype:

Name	Value
jboss-bom-enterpris...	1.0.4.Final-redhat-4
enterprise	false

Resolve Workspace projects [ ]
Profiles: arq-jbossas-remote
Name template:

A Java EE project gets created, as shown in the following Project Explorer window. Delete the `jboss-ejb3/jboss-ejb3-ear/src/main/application/META-INF/jboss-ejb3-ds.xml` configuration file. The jboss-ejb3 project consists of three subprojects: `jboss-ejb3-ear`, `jboss-ejb3-ejb`, and `jboss-ejb3-web`. Each subproject consists of a pom.xml file for Maven. Initially the subprojects indicate errors with red error markers, but these would get fixed when the main project is built later in the chapter. Initially the subprojects might indicate errors with red error markers, but these would get fixed when the main project is built later in the chapter. We will configure a data source with the MySQL database in a later section. The `jboss-ejb3-ejb` subproject consists of a META-INF/persistence.xml file within the src/main/resources source folder for the JPA database persistence configuration.



```
jboss-ejb3
  jboss-ejb3-ear
    src
    target
    pom.xml
  jboss-ejb3-ejb
    src
      main
        java
        resources
          META-INF
            beans.xml
            persistence.xml
      test
    target
    pom.xml
  jboss-ejb3-web
    src
      main
        java
        resources
        webapp
          resources
          WEB-INF
            beans.xml
            faces-config.xml
      test
    target
    pom.xml
  pom.xml
```

We will use MySQL as the database for data for the EJB application. In the next section, we will create a data source in the MySQL database.

Configuring a data source with MySQL database

The default data source in WildFly 8.1 is configured with the H2 database engine. There are several options available for a database. The top four most commonly used relational databases are Oracle database, MySQL database, SQL Server, and PostgreSQL Server. Oracle database and SQL Server are designed for enterprise level applications and are not open source. Oracle database offers more features to facilitate system and data maintenance. It also offers features to prevent system and data failure as compared to SQL Server. MySQL and PostgreSQL are open source databases with comparable features and designed primarily for small scale applications. We will use MySQL database. Some of the reasons to choose MySQL are discussed at `http://www.mysql.com/why-mysql/topreasons.html`.

We will configure a datasource with the MySQL database for use in the EJB 3.x application for object/relational mapping. Use the following steps to configure a datasource:

1. First, we need to create a module for MySQL database. For the MySQL module, create a module.xml file in the `%JBOSS_HOME%/modules/mysql/main` directory; the `mysql/main` subdirectory is also to be created. The module.xml file is listed in the following code snippet:

   ```
   <module xmlns="urn:jboss:module:1.1" name="mysql" slot="main">
     <resources>
       <resource-root path="mysql-connector-java-5.1.33-bin.jar"/>
     </resources>
     <dependencies>
       <module name="javax.api"/>
     </dependencies>
   </module>
   ```

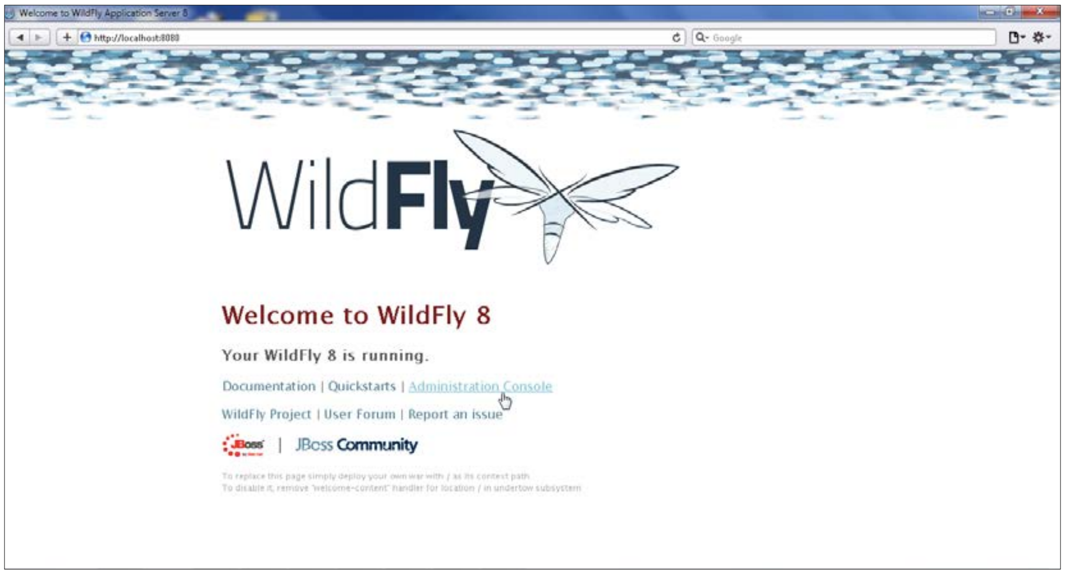
2. Copy the `mysql-connector-java-5.1.33-bin.jar` (MySQL JDBC JAR) file from `C:\Program Files (x86)\MySQL\Connector.J 5.1` to the `%JBOSS_HOME%/modules/mysql/main` directory. The MySQL `mysql-connector-java` JAR file version specified in `module.xml` must be the same as the version of the JAR file copied to the `/modules/mysql/main` directory.

3. Add a `<datasource/>` definition for the MySQL database to the `<datasources/>` element and a `<driver/>` definition to the `<drivers/>` element in the `%JBOSS_HOME%/standalone/configuration/standalone.xml` file within the `<subsystem xmlns="urn:jboss:domain:datasources:2.0"> </subsystem>` element. The `<password/>` tag in the `<datasource/>` configuration tag is the password configured when the MySQL database is installed. The datasource class for the MySQL driver is a `XA` datasource, which is used for distributed transactions:

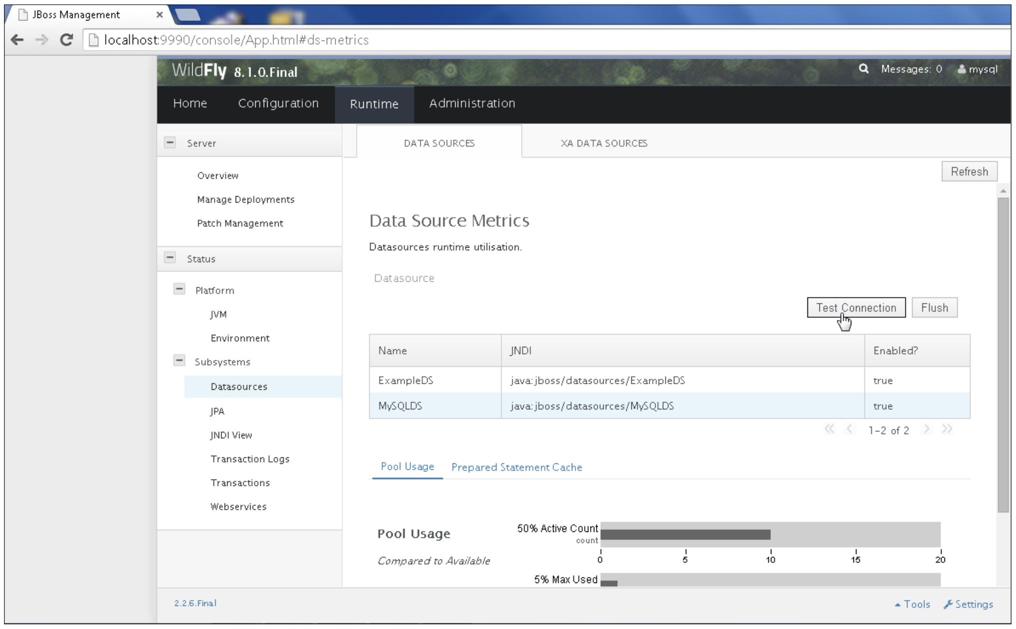
```
<subsystem xmlns="urn:jboss:domain:datasources:2.0">
  <datasources>
    <datasource jndi-name="java:jboss/datasources/MySQLDS" pool-
name="MySQLDS" enabled="true" use-java-context="true">
      <connection-url>jdbc:mysql://localhost:3306/test</
connection-url>
      <driver>mysql</driver>
      <pool>
        <min-pool-size>10</min-pool-size>
        <max-pool-size>20</max-pool-size>
        <prefill>true</prefill>
      </pool>
      <security>
        <user-name>root</user-name>
        <password>mysql</password>
      </security>
    </datasource>
    <drivers>
      <driver name="mysql" module="mysql">
        <driver-class>com.mysql.jdbc.Driver</driver-class>
        <xa-datasource-class>com.mysql.jdbc.jdbc2.optional.
MysqlXADataSource</xa-datasource-class>
      </driver>
    </drivers>
  </datasources>
</subsystem>
```

4. If the server is running after modifying the standalone.xml configuration file, restart WildFly 8.x server. The MySQL datasource gets deployed. To start or restart the WildFly server, double-click on the `C:\wildfly-8.1.0.Final\bin\standalone` batch file.

5. Log in to the WildFly 8 Administration Console with the URL: `http://localhost:8080`. Click on **Administration Console**, as shown in the following screenshot:


Welcome to WildFly Application Server 8
http://localhost:8080
WildFly
Welcome to WildFly 8
Your WildFly 8 is running.
Documentation | Quickstarts | Administration Console
WildFly Project | User Forum | Report an issue
JBoss | JBoss Community
To replace this page simply deploy your own war with / as its context path.
To disable it, remove "welcome-content" handler for location / in undertow subsystem

6. In the login dialog box, specify the username and password for the user added with the add-user.bat script.
7. Select the **Runtime** tab in Administration Console. The MySQL datasource is listed as deployed in **Datasources Subsystems**, as shown in the following screenshot. Click on **Test Connection** to test the connection:



Data Source Metrics

Datasources runtime utilisation.

Datasource

Name	JNDI	Enabled?
ExampleDS	java:jboss/datasources/ExampleDS	true
MySQLDS	java:jboss/datasources/MySQLDS	true

1-2 of 2

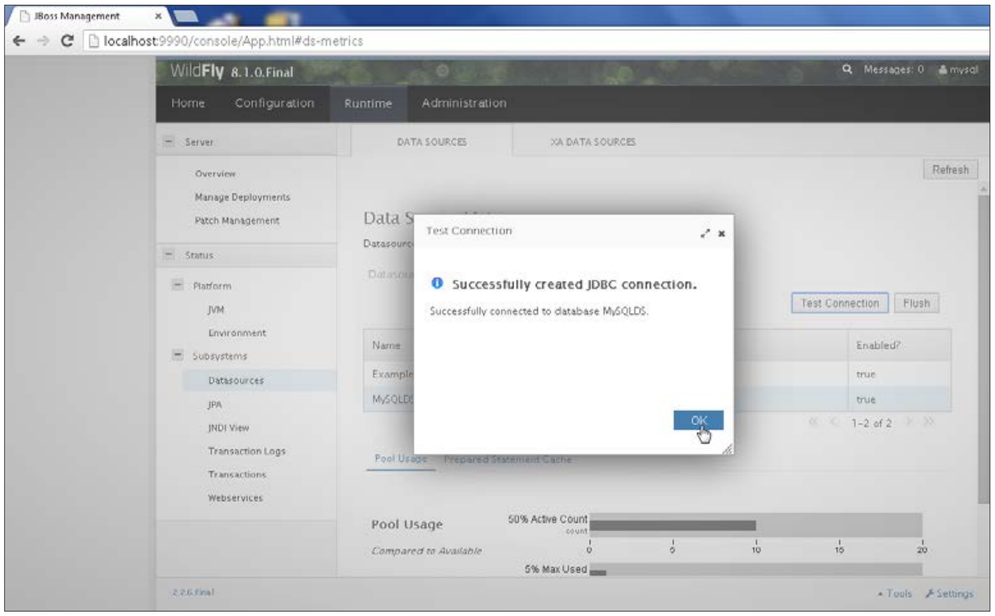
Pool Usage

Compared to Available

50% Active Count

5% Max Used

8. If a connection with the MySQL database is established, a **Successfully created JDBC connection** message will get displayed:



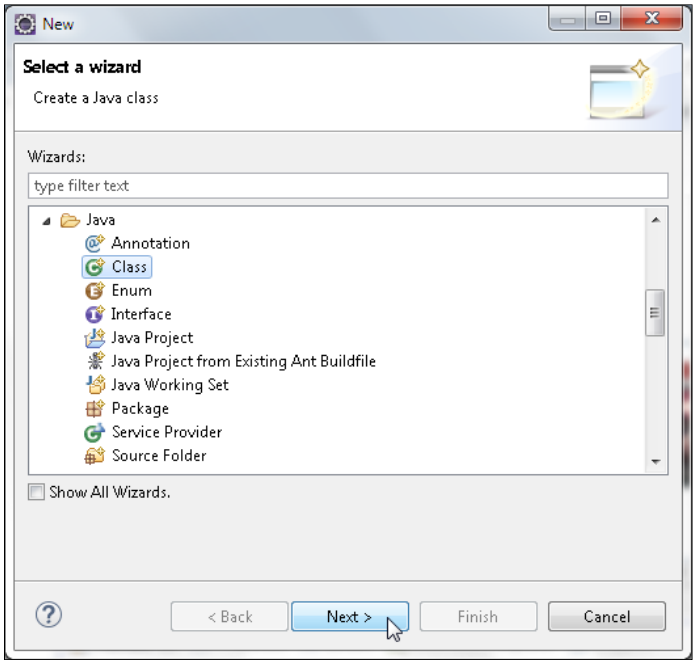
Successfully created JDBC connection.

Successfully connected to database MySQLDS.

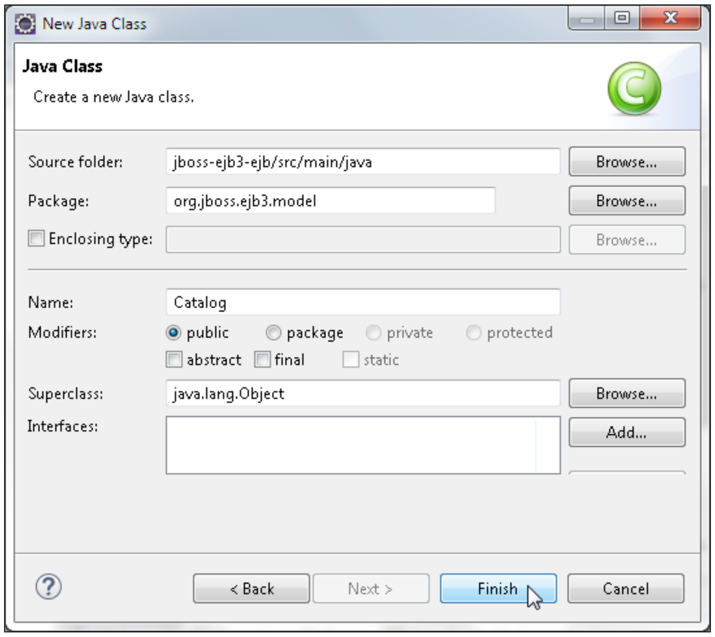
In the next section, we will create entities for the EJB 3.x application.

Creating entities

In EJB 3.x, an entity is a **POJO** (**Plain Old Java Object**) persistent domain object that represents a database table row. As an entity is a Java class, create a Java class in the `jboss-ejb3-ejb` subproject of the jboss-ejb3 project. Select **File** | **New**. In the **New** window, select **Java** | **Class** and click on **Next**:

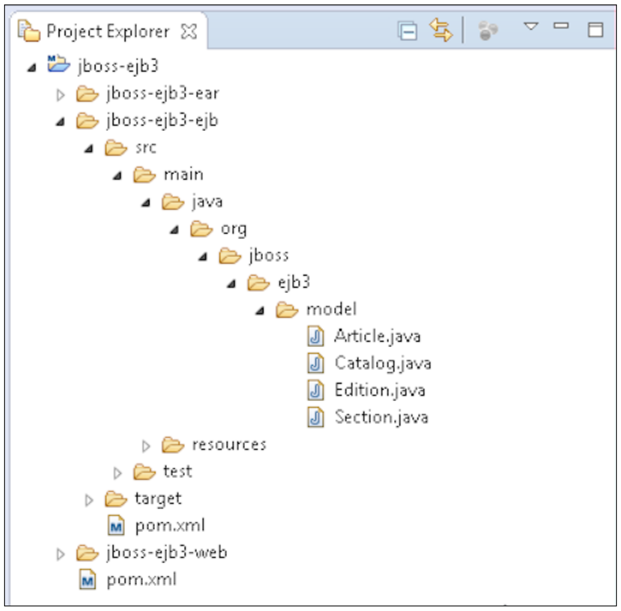


Select/specify `jboss-ejb3/jboss-ejb3-ejb/src/main/java` as the Java **Source folder**, `org.jboss.ejb3.model` as the **Package**, and `Catalog` as the class **Name**. Click on **Finish**:



Source folder: jboss-ejb3-ejb/src/main/java
Package: org.jboss.ejb3.model
Enclosing type: [ ]
Name: Catalog
Modifiers: [✓ public] package private protected
[ ] abstract [ ] final [ ] static
Superclass: java.lang.Object
Interfaces:

Similarly, add Java classes for the `Edition.java`, `Section.java`, and `Article.java` entities, as shown in the following **Project Explorer**:

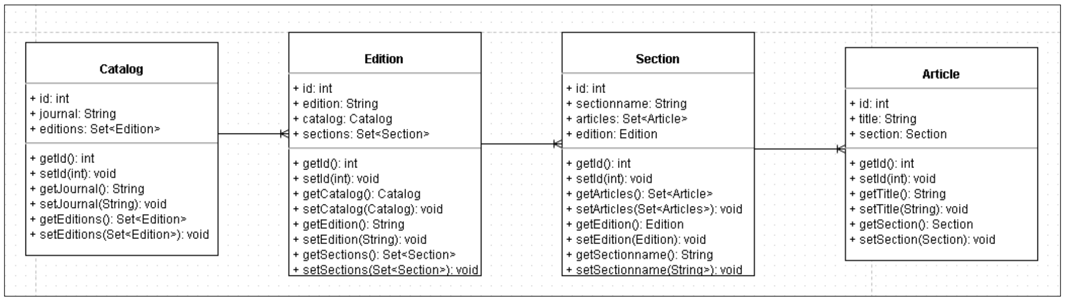


```
jboss-ejb3
  jboss-ejb3-ear
  jboss-ejb3-ejb
    src
      main
        java
          org
            jboss
              ejb3
                model
                  Article.java
                  Catalog.java
                  Edition.java
                  Section.java
        resources
      test
    target
    pom.xml
  jboss-ejb3-web
  pom.xml
```

Next, we develop the EJB 3.x entities. A JPA persistence provider is required for the EJB entities, and we will use the Hibernate persistence provider. The Hibernate persistence provider has some peculiarities that need to be mentioned, as follows:

- If an entity has more than one non-lazy association of the following types, Hibernate fails to fetch the entity:
    - The `java.util.List, java.util.Collection` properties annotated with `@org.hibernate.annotations.CollectionOfElements`
    - The `@OneToMany` or `@ManyToMany` associations not annotated with `@org.hibernate.annotations.IndexColumn`
    - Associations marked as mappedBy must not define database mappings (such as @JoinTable or @JoinColumn)

We will develop the Catalog, Edition, Section, and Article class with one-to-many relationship between the Catalog and Edition class, the Edition and Section class, and the Section and Article class, as shown in the following UML class diagram:



Annotate the Catalog entity class with the @Entity annotation and the @Table annotation. If the @Table annotation is not used, then the entity name is used as the table name by default. In the @Table annotation, specify the table name as CATALOG and uniqueConstraints, using the @UniqueConstraint annotation for the `id` column. Specify the named queries as findCatalogAll, which selects all Catalog and findCatalogByJournal entities. This selects a Catalog entity by Journal, using the @NamedQueries and @NamedQuery annotations:

```
@Entity
@Table(name = "CATALOG", uniqueConstraints = @
UniqueConstraint(columnNames = "ID"))
@NamedQueries({
  @NamedQuery(name="findCatalogAll", query="SELECT c FROM Catalog c"),
  @NamedQuery(name="findCatalogByJournal",
    query="SELECT c FROM Catalog c WHERE c.journal = :journal")
})
public class Catalog implements Serializable {
}
```

Specify the no-argument constructor, which is required in an entity class. The Catalog entity class implements the Serializable interface to serialize a cache-enabled entity to a cache when persisted to a database. To associate a version number with a serializable class for a serialization runtime, specify a serialVersionUID variable. Declare String variables for id and journal bean properties and for a collection of Set<Edition> type, as the Catalog entity has a bi-directional one-to-many association to Edition. The collection is chosen as Set for the reason mentioned earlier. Hibernate does not support more than one EAGER association of the java.util.List type. Add get/set methods for the bean properties. The @Id annotation specifies the identifier property. The @Column annotation specifies the column name associated with the property. The nullable element is set to false as the primary key cannot be null.

If we were using the Oracle database, we would have specified the primary key generator to be of the sequence type, using the @SequenceGenerator annotation. The generation strategy is specified with the @GeneratedValue annotation. For the Oracle database, the generation strategy would be strategy=GenerationType.SEQUENCE, but as MySQL database supports auto increment of primary key column values by generating a sequence, we have set the generation strategy to GenerationType.AUTO.

Specify the bi-directional one-to-many association to Edition using the @OneToMany annotation. The mappedBy element is specified on the non-owning side of the relationship, which is the Catalog entity. The cascade element is set to ALL. Cascading is used to cascade database table operations to associated tables. The fetch element is set to EAGER. With EAGER fetching the associated entity, collection is immediately fetched when an entity is retrieved:

```
    // bi-directional many-to-one association to Edition
    @OneToMany(mappedBy = "catalog", targetEntity=org.jboss.ejb3.model.
    Edition.class, cascade = { CascadeType.ALL }, fetch = FetchType.EAGER)
      public Set<Edition> getEditions() {
        return this.editions;
      }
    }
```

As mentioned earlier, associations marked with `mappedBy` must not specify `@JoinTable` or `@JoinColumn`. The get and set methods for the `Edition` collection are also specified. The `Catalog.java` entity class is available in the code download for the chapter at `http://www.packtpub.com/support`.

Next, develop the entity class for the `EDITION` database table: `Edition.java`. Specify the `@Entity`, `@Table`, `@Id` , `@Column`, and `@GeneratedValue` annotations, as discussed for the `Catalog` entity. Specify the `findEditionAll` and `findEditionByEdition` named queries to find Edition collections. Specify the bean properties and associated get/set methods for `id` and `edition`. Also, specify the one-to-many association to the `Section` entity using a collection of the `Set` type. The bi-directional many-to-one association to the `Catalog` relationship is specified using the `@ManyToOne` annotation, and with cascade of type `PERSIST`, `MERGE`, and `REFRESH`. The `Edition` entity is the owning side of the relationship. Using the `@JoinTable` annotation, a join table is included on the owning side to initiate cascade operations. The join columns are specified using the `@JoinColumn` annotation. The `Edition.java` entity class is available in the code download for the chapter.

Develop the entity class for the `SECTION` table: `Section.java`. Specify the `findSectionAll` and `findSectionBySectionName` named queries to find `Section` entities. Specify the `id` and `sectionname` bean properties. Specify the bi-directional many-to-one association to `Edition` using the `@ManyToOne` annotation and the bi-directional one-to-many association to `Article` using `@OneToMany`. The `@JoinTable` and `@JoinColumn` are specified only for the `@ManyToOne` association for which `Section` is the owning side. The `Section.java` entity class is available in the code download for the chapter.

Specify the entity class for the `ARTICLE` table: `Article.java`. The `Article` entity is mapped to the `ARTICLE` database table using the `@TABLE` annotation. Add the `findArticleAll` and `findArticleByTitle` named queries to find `Article` entities. Specify `id` and `sectionname` bean properties and the associated `get/set` methods. The `Article` entity is the owning side of the bi-directional many-to-one association to `Section`. Therefore, the `@JoinTable` and `@JoinColumn` are specified. The `Article.java` class is available in the code downloaded for the chapter.

Creating a JPA persistence configuration file

The META-INF/persistence.xml configuration file in the ejb/src/main/resources folder in the jboss-ejb3-ejb subproject was created when we created the Java EE project. The persistence.xml specifies a persistence provider to be used to map object/relational entities to the database. Specify that, the persistence unit is using the persistence-unit element. Set the transaction-type to JTA (the default value). Specify the persistence provider as the Hibernate persistence provider: org.hibernate.ejb.HibernatePersistence. Set the jta-data-source element value to the java:jboss/datasources/MySQLDS data source, which we created earlier. Specify the entity classes using the class element. The DDL generation strategy is set to create-drop using the hibernate.hbm2ddl.auto property, which automatically validates or exports the DDL schema to the database, when the SessionFactory class is created. With the create-drop strategy, the required tables are created and dropped when the SessionFactory is closed. The hibernate.show_sql property is set to false. Setting it to true implies that all SQL statements be the output, which is an alternative method to debug. The hibernate.dialect property is set to org.hibernate.dialect.MySQLDialect for MySQL Database. Other Hibernate properties (http://docs.jboss.org/hibernate/orm/3.3/reference/en/html/session-configuration.html) can also be specified as required. The persistence.xml configuration file is listed in the following code:

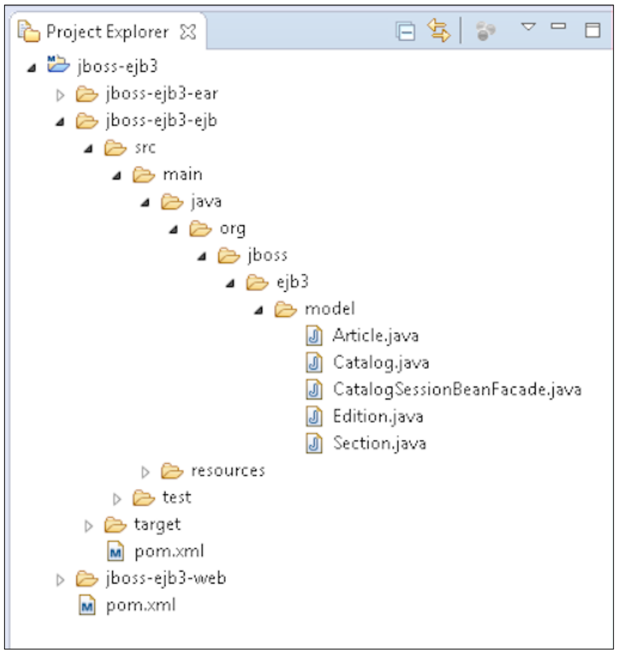
```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="2.0"
xsi:schemaLocation="         http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
  <persistence-unit name="em" transaction-type="JTA">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <!-- If you are running in a production environment, add a managed
data source, the example data source is just for development and
testing! -->
    <jta-data-source>java:jboss/datasources/MySQLDS</jta-data-source>
    <class>org.jboss.ejb3.model.Article</class>
    <class>org.jboss.ejb3.model.Catalog</class>
    <class>org.jboss.ejb3.model.Edition</class>
    <class>org.jboss.ejb3.model.Section</class>
    <properties>
      <!-- Properties for Hibernate -->
      <property name="hibernate.hbm2ddl.auto" value="create-drop" />
      <property name="hibernate.show_sql" value="false" />
```

```
        <property name="hibernate.dialect" value="org.hibernate.dialect.
MySQLDialect" />
    </properties>
  </persistence-unit>
</persistence>
```

The JPA specification does not mandate a persistence provider to create tables with the `hibernate.hbm2ddl.auto` property set to `create-drop` or `create`. Hibernate persistence provider supports creating tables. In addition to the entity tables, some additional tables (such as the join tables and the sequence table) are created by the Hibernate persistence provider.

Creating a session bean facade

One of the best practices of developing entities for separation of concerns and maintainable code and as a result better performance is to wrap the entities in a session bean facade. With a Session Facade, fewer remote method calls are required, and an outer transaction context is created with which each get method invocation does not start a new transaction. Session Facade is one of the core Java EE design patterns (`http://www.oracle.com/technetwork/java/sessionfacade-141285.html`). Create a `CatalogSessionBeanFacade` session bean class in the `org.jboss.ejb3.model` package, as shown in the following screenshot. The Session Facade class can also be created in a different package (such as `org.jboss.ejb3.view`):



The session bean class is annotated with the `@Stateless` annotation:

```
@Stateless
public class CatalogSessionBeanFacade {}
```

In the bean session, we use an `EntityManager` to create, remove, find, and query persistence entity instances. Inject a `EntityManager` using the `@PersistenceContext` annotation. Specify the `unitName` as the `unitName` configured in `persistence.xml`. Next, specify the `getAllEditions`, `getAllSections`, `getAllArticles`, `getAllCatalogs` get methods to fetch the collection of entities. The `get` methods get all entities' collections with the named queries specified in the entities. The `createNamedQuery` method of `EntityManager` is used to create a `Query` object from a named query. Specify the `TransactionAttribute` annotation's `TransactionAttributeType` enumeration to `REQUIRES_NEW`, which has the advantage that if a transaction is rolled back due to an error in a different transaction context from which the session bean is invoked, it does not affect the session bean.

To demonstrate the use of the entities, create the test data with the `createTestData` convenience method in the session bean. Alternatively, a unit test or an extension class can also be used. Create a `Catalog` entity and set the journal using the `setJournal` method. We do not set the id for the `Catalog` entity as we use the `GenerationType.AUTO` generation strategy for the `ID` column. Persist the entity using the `persist` method of the `EntityManager` object. However, the `persist` method does not persist the entity to the database. It only makes the entity instance managed and adds it to the persistence context. The `EntityManager.flush()` method is not required to be invoked to synchronize the entity with the database as `EntityManager` is configured with `FlushModeType` as `AUTO` (the other setting being `COMMIT`) and a flush will be done automatically when the EntityManager.persist() is invoked:

```
Catalog catalog1 = new Catalog();
catalog1.setJournal("Oracle Magazine");
em.persist(catalog1);
```

Similarly, create and persist an `Edition` entity object. Add the `Catalog` object: `catalog1` using the `setCatalog` method of the `Edition` entity class:

```
Edition edition = new Edition();
edition.setEdition("January/February 2009");
edition.setCatalog(catalog1);
em.persist(edition);
```

Likewise add the `Section` and `Article` entity instances. Add another `Catalog` object, but without any associated `Edition`, `Section`, or `Article` entities:

```
Catalog catalog2 = new Catalog();
catalog2.setJournal("Linux Magazine");
em.persist(catalog2);
```

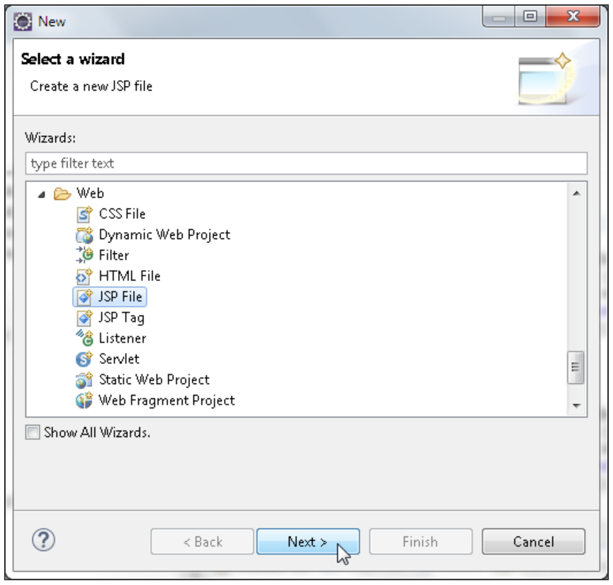
Next, we will delete data with the `deleteSomeData` method, wherein we first create a `Query` object using the named query `findCatalogByJournal`. Specify the journal to delete with the `setParameter` method of the Query object. Get the `List` result with the `getResultList` method of the `Query` object. Iterate the `List` result and remove the `Catalog` objects with the `remove` method of the `EntityManager` object. The `remove` method only removes the `Catalog` object from the persistence context:

```
public void deleteSomeData() {
  // remove a catalog
  Query q = em.createNamedQuery("findCatalogByJournal");
  //q.setParameter("journal", "Linux Magazine");
  q.setParameter("journal", "Oracle Magazine");
  List<Catalog> catalogs = q.getResultList();
  for (Catalog catalog : catalogs) {
    em.remove(catalog);
  }
}
```

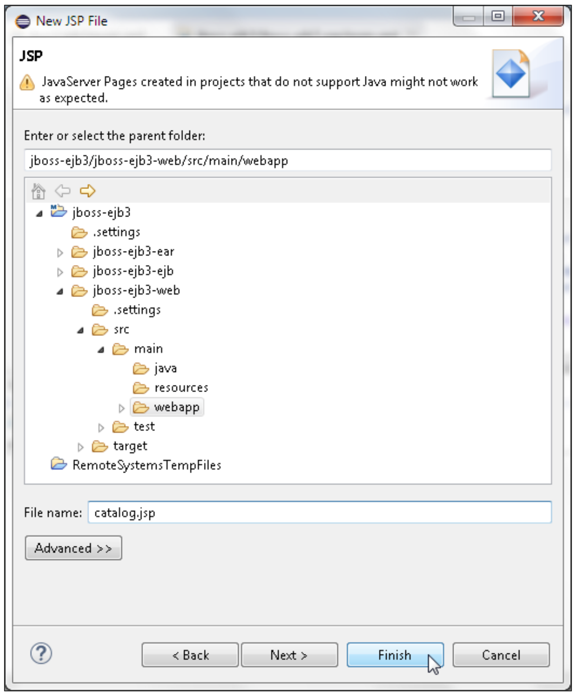
The `CatalogSessionBeanFacade` session bean class is available in the code downloaded for the chapter.

Creating a JSP client

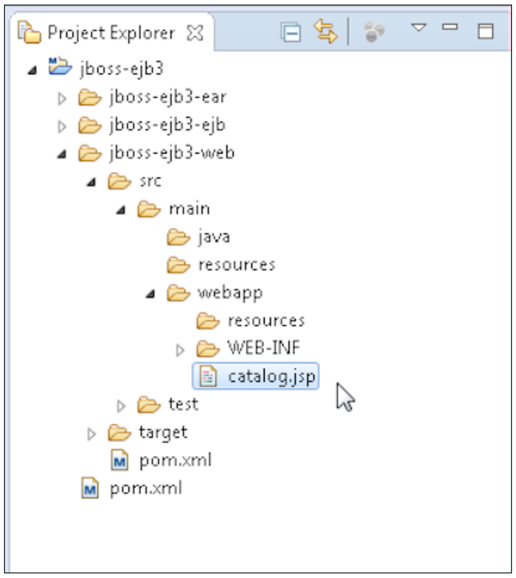
Next, we will create a JSP client to test the EJB entities. We will look up the session bean using a local JNDI name. Subsequently, we will invoke the testData method of the session bean to test database persistence using these entities. First create a JSP file. Select **File** | **New** | **Other**, and in the **New** wizard, select **Web** | **JSP File** and click on **Next**, as in the following screenshot:



In the **New JSP File** wizard, select the **jboss-ejb3/web/src/main/webapp** folder in the **jboss-ejb3-web** subproject. Specify **catalog.jsp** as as **File name** and click on **Next**. Then click on **Finish**:



The `catalog.jsp` file gets added to the **jboss-ejb3-web** subproject:



We need to retrieve the `CatalogSessionBeanFacade` component from the JSP client. WildFly 8 provides the local **JNDI** (**Java Naming and Directory Interface**) namespace: Java, and the following JNDI contexts:

JNDI Context	Description
`java:comp`	This is the namespace that is scoped to the current component, the EJB.
`java:module`	This namespace is scoped to the current module.
`java:app`	This namespace is scoped to the current application.
`java:global`	This namespace is scoped to the application server.

When the `jboss-ejb3` application is deployed, the JNDI bindings in the namespaces (discussed in the preceding table) are created as indicated by the server message:

```
JNDI bindings for session bean named CatalogSessionBeanFacade in
deployment unit subdeployment "jboss-ejb3-ejb.jar" of deployment
"jboss-ejb3-ear.ear" are as follows:
  java:global/jboss-ejb3-ear/jboss-ejb3-ejb/
CatalogSessionBeanFacade!org.jboss.ejb3.model.CatalogSessionBeanFacade
  java:app/jboss-ejb3-ejb/CatalogSessionBeanFacade!org.jboss.ejb3.
model.CatalogSessionBeanFacade
  java:module/CatalogSessionBeanFacade!org.jboss.ejb3.model.
    CatalogSession
```

```
BeanFacade
  java:global/jboss-ejb3-ear/jboss-ejb3-ejb/CatalogSessionBeanFacade
  java:app/jboss-ejb3-ejb/CatalogSessionBeanFacade
  java:module/CatalogSessionBeanFacade
```

Next we will retrieve the session bean façade: `CatalogSessionBeanFacade` using the standard Java SE JNDI API, which does not require any additional configuration, using the local JNDI lookup in the `java:app` namespace. For the local JNDI lookup, we need to create an `InitialContext` object:

```
Context context = new InitialContext();
```

Using the local JNDI name lookup in the `java:app` namespace ,retrieve the `CatalogSessionBeanFacade` component:

```
CatalogSessionBeanFacade bean = (CatalogSessionBeanFacade) context
.lookup("java:app/jboss-ejb3-ejb/CatalogSessionBeanFacade!org.jboss.
ejb3.model.CatalogSessionBeanFacade");
```

Invoke the `createTestData` method and retrieve the `List Catalog` entities. Iterate over the `Catalog` entities and output the catalog ID as the journal name:

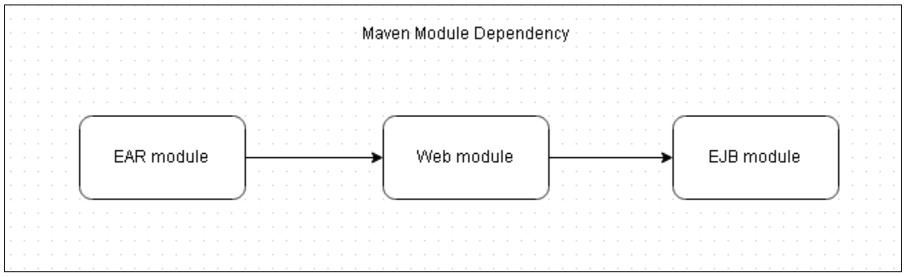
```
bean.createTestData();
List<Catalog> catalogs = beanRemote.getAllCatalogs();
out.println("<br/>" + "List of Catalogs" + "<br/>");
for (Catalog catalog : catalogs) {
  out.println("Catalog Id:");
  out.println("<br/>" + catalog.getId() + "<br/>");
  out.println("Catalog Journal:");
  out.println(catalog.getJournal() + "<br/>");
}
```

Similarly, obtain the `Entity`, `Section`, and `Article` entities and output the entity property values. The `catalog.jsp` file is available in the code downloaded for the chapter.

Configuring the jboss-ejb3-ejb subproject

We will generate an EAR file using the Maven project: `jboss-ejb3`, which includes the `jboss-ejb3-ejb`, `jboss-ejb-web` and `jboss-ejb3-ear` subproject/artifacts. We will use the Maven build tool to compile, package, and deploy the EAR application. The `jboss-ejb3-ear` module to be deployed to WildFly has two submodules: `jboss-ejb3-web` and `jboss-ejb3-ejb`.

The `jboss-ejb3-ear`, `jboss-ejb3-web` and `jboss-ejb3-ejb` modules may be referred to as `ear`, `web`, and `ejb` modules respectively. The `ear` module has dependency on the `web` module, and the `web` module has dependency on the `ejb` module, as shown in the following diagram:



The `ejb`, `web`, and `ear` modules can be built and installed individually using subproject-specific `pom.xml`, or these can be built together using the `pom.xml` file in the `jboss-ejb3` project. If built individually, the `ejb` module has to be built and installed before the `web` module, as the `web` module has a dependency on the `ejb` module. The `ear` module is to be built after the `web` and `ejb` modules have been built and installed. We will build and install the top level project using the `pom.xml` file in the `jboss-ejb3` project, which has dependency specified on the `jboss-ejb3-web` and `jboss-ejb3-ejb` artifacts. The `pom.xml` file for the `jboss-ejb3-ejb` subproject specifies packaging as `ejb`. The WildFly 8.x provides most of the APIs required for an EJB 3.x application. The provided APIs are specified with `scope` set to `provided` in `pom.xml`. Dependencies for the EJB 3.1 API and the JPA 2.0 API are pre-specified. Add the following dependency for the **Hibernate Annotations API**:

```
<dependency>
  <groupId>org.jboss.spec.javax.ejb</groupId>
  <artifactId>jboss-ejb-api_3.1_spec</artifactId>
  <version>1.0.0.Final</version>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>org.hibernate.javax.persistence</groupId>
  <artifactId>hibernate-jpa-2.0-api</artifactId>
  <version>1.0.0.Final</version>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-annotations</artifactId>
  <version>3.5.6-Final</version>
</dependency>
```

The Hibernate Validator API dependency is also preconfigured in `pom.xml`. The build is preconfigured with the Maven EJB plugin, which is required to package the subproject into an EJB module. The EJB version in the Maven EJB plugin is 3.1:

```
<build>
  <finalName>${project.artifactId}</finalName>
  <plugins>
    <plugin>
      <artifactId>maven-ejb-plugin</artifactId>
      <version>${version.ejb.plugin}</version>
      <configuration>
        <!-- Tell Maven we are using EJB 3.1 -->
        <ejbVersion>3.1</ejbVersion>
      </configuration>
    </plugin>
  </plugins>
</build>
```

The Maven `POM.xml` file for the EJB subproject is available in the code downloaded for the chapter.

Configuring the jboss-ejb3-web subproject

Most of the required configuration for the `jboss-ejb3-web` subproject is pre-specified. The packaging for the `jboss-ejb3-web` artifacts is set to `war`:

```
<artifactId>jboss-ejb3-web</artifactId>
<packaging>war</packaging>
<name>jboss-ejb3 Web module</name>
```

The `pom.xml` file for the subproject pre-specifies most of the required dependencies. It also specifies dependency on the `jboss-ejb3-ejb` artifact:

```
<dependency>
  <groupId>org.jboss.ejb3</groupId>
  <artifactId>jboss-ejb3-ejb</artifactId>
  <type>ejb</type>
  <version>1.0.0</version>
  <scope>provided</scope>
</dependency>
```

The EJB 3.1 API, the JPA 2.0 API, the JSF 2.1 API, and the JAX-RS 1.1 API are provided by the WildFly 8.x server, as indicated by the `provided` scope in the `dependency` declarations. Add the dependency on the `hibernate-annotations` artifact. The build is preconfigured with the Maven `WAR` plugin, which is required to package the subproject into an `WAR` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<build>
  <finalName>${project.artifactId}</finalName>
  <plugins>
    <plugin>
      <artifactId>maven-war-plugin</artifactId>
      <version>${version.war.plugin}</version>
      <configuration>
        <!-- Java EE 6 doesn't require web.xml, Maven needs to catch
up! -->
        <failOnMissingWebXml>false</failOnMissingWebXml>
      </configuration>
    </plugin>
  </plugins>
</build>
```

The `pom.xml` file for the `jboss-ejb3-web` subproject is available in the code downloaded for the chapter.

Configuring the jboss-ejb3-ear subproject

In `pom.xml` for the `jboss-ejb3-ear` subproject, the packaging for the `jboss-ejb3-ear` artifact is specified as `ear`:

```
<artifactId>jboss-ejb3-ear</artifactId>
<packaging>ear</packaging>
```

The `pom.xml` file specifies dependency on the ejb and web modules:

```
<dependencies>
  <!-- Depend on the ejb module and war so that we can package them
-->
  <dependency>
  <groupId>org.jboss.ejb3</groupId>
```

```
    <artifactId>jboss-ejb3-web</artifactId>
    <version>1.0.0</version>
    <type>war</type>
  </dependency>
    <dependency>
    <groupId>org.jboss.ejb3</groupId>
    <artifactId>jboss-ejb3-web</artifactId>
    <version>1.0.0</version>
    <type>war</type>
  </dependency>
  </dependencies>
```

The `build` tag in the `pom.xml` file specifies the configuration for the `maven-ear-plugin` plugin with output directory as the `deployments` directory in the WildFly 8.x standalone server. The `EAR` file generated from the Maven project is deployed to the directory specified in the `<outputDirectory/>` element. Specify the `<outputDirectory/>` element as the `C:\wildfly-8.1.0.Final\standalone\deployments` directory. The `outputDirectory` might need to be modified based on the installation directory of WildFly 8.1. The `EAR`, `WAR`, and `JAR` modules in the deployments directory get deployed to the WildFly automatically, if the server is running:

```
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-ear-plugin</artifactId>
    <version>2.8</version>
    <configuration>
      <!-- Tell Maven we are using Java EE 6 -->
      <version>6</version>
      <!-- Use Java EE ear libraries as needed. Java EE ear libraries
  are in easy way to package any libraries needed in the ear, and
  automatically have any modules (EJB-JARs and WARs) use them -->
      <defaultLibBundleDir>lib</defaultLibBundleDir>
      <fileNameMapping>no-version</fileNameMapping>
      <outputDirectory>C:\wildfly-8.1.0.Final\standalone\deployments</
  outputDirectory>
    </configuration>
  </plugin>
```

Deploying the EAR module

In this section, we will build and deploy the application EAR module to the WildFly 8.x server. The `pom.xml` for the `jboss-ejb3` Maven project specifies three modules: `jboss-ejb3-ejb`, `jboss-ejb3-web`, and `jboss-ejb3-ear`:

```
<modules>
  <module>jboss-ejb3-ejb</module>
  <module>jboss-ejb3-web</module>
  <module>jboss-ejb3-ear</module>
</modules>
```

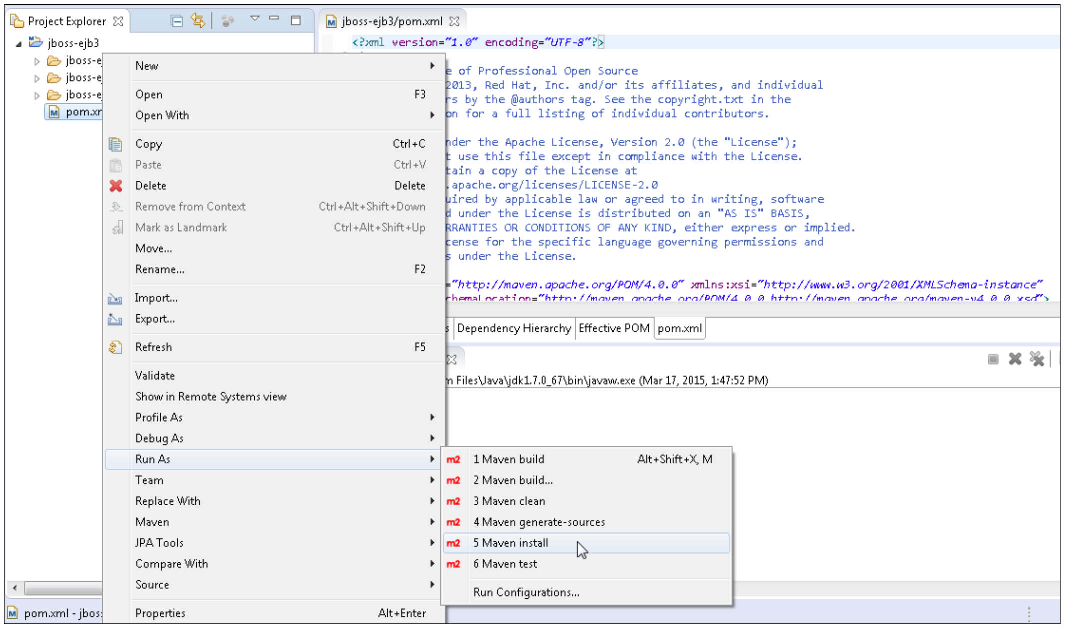
Specify the JBoss AS version as `8.1.0.Final`:

```
<version.jboss.as>8.1.0.Final</version.jboss.as>
```

The `pom.xml` for the `jboss-ejb3` project specifies dependency on the `jboss-ejb3-web` and `jboss-ejb3-ejb` artifacts:

```
<dependency>
  <groupId>org.jboss.ejb3</groupId>
  <artifactId>jboss-ejb3-ejb</artifactId>
  <version>${project.version}</version>
  <type>ejb</type>
</dependency>
<dependency>
  <groupId>org.jboss.ejb3</groupId>
  <artifactId>jboss-ejb3-web</artifactId>
  <version>${project.version}</version>
  <type>war</type>
  <scope>compile</scope>
</dependency>
```

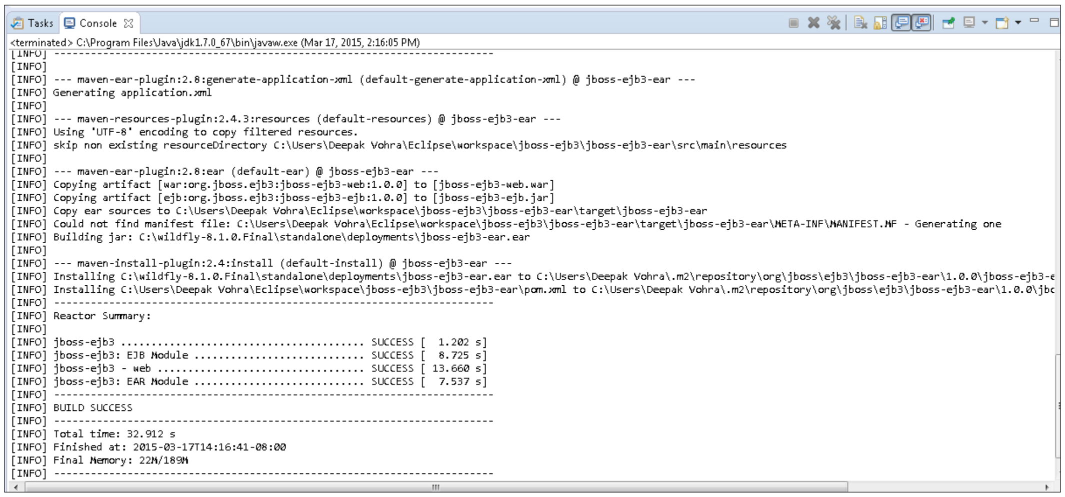
Next, we will build and deploy the EAR module to WildFly 8.x while the server is running. Right-click on `pom.xml` for the `jboss-ejb3` Maven project and select **Run As | Maven install**, as shown in the following screenshot:



```
<?xml version="1.0" encoding="UTF-8"?>
e of Professional Open Source
2013, Red Hat, Inc. and/or its affiliates, and individual
rs by the @authors tag. See the copyright.txt in the
on for a full listing of individual contributors.

nder the Apache License, Version 2.0 (the "License");
t use this file except in compliance with the License.
tain a copy of the License at
.apache.org/licenses/LICENSE-2.0
uired by applicable law or agreed to in writing, software
d under the License is distributed on an "AS IS" BASIS,
RRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
cense for the specific language governing permissions and
s under the License.

="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

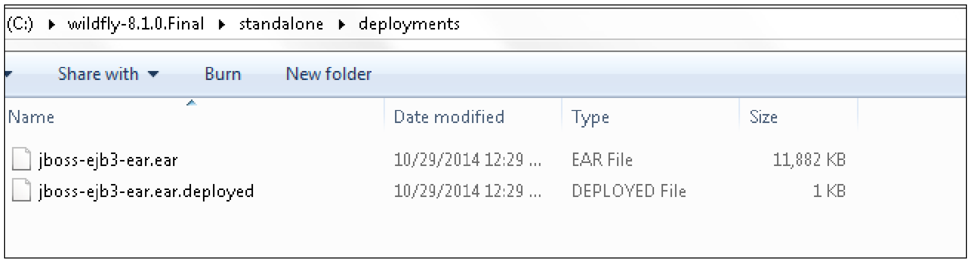
```
n Files\Java\jdk1.7.0_67\bin\javaw.exe (Mar 17, 2015, 1:47:52 PM)
```

As the output from the `pom.xml` indicates all the three modules: `ejb`, `web`, and `ear` get built. The `ear` module gets copied to the `deployments` directory in WildFly 8.x:



```
<terminated> C:\Program Files\Java\jdk1.7.0_67\bin\javaw.exe (Mar 17, 2015, 2:16:05 PM)
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-ear-plugin:2.8:generate-application-xml (default-generate-application-xml) @ jboss-ejb3-ear ---
[INFO] Generating application.xml
[INFO]
[INFO] --- maven-resources-plugin:2.4.3:resources (default-resources) @ jboss-ejb3-ear ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory C:\Users\Deepak Vohra\Eclipse\workspace\jboss-ejb3\jboss-ejb3-ear\src\main\resources
[INFO]
[INFO] --- maven-ear-plugin:2.8:ear (default-ear) @ jboss-ejb3-ear ---
[INFO] Copying artifact [war:org.jboss.ejb3:jboss-ejb3-web:1.0.0] to [jboss-ejb3-web.war]
[INFO] Copying artifact [ejb:org.jboss.ejb3:jboss-ejb3-ejb:1.0.0] to [jboss-ejb3-ejb.jar]
[INFO] Copy ear sources to C:\Users\Deepak Vohra\Eclipse\workspace\jboss-ejb3\jboss-ejb3-ear\target\jboss-ejb3-ear
[INFO] Could not find manifest file: C:\Users\Deepak Vohra\Eclipse\workspace\jboss-ejb3\jboss-ejb3-ear\target\jboss-ejb3-ear\META-INF\MANIFEST.MF - Generating one
[INFO] Building jar: C:\wildfly-8.1.0.Final\standalone\deployments\jboss-ejb3-ear.ear
[INFO]
[INFO] --- maven-install-plugin:2.4:install (default-install) @ jboss-ejb3-ear ---
[INFO] Installing C:\wildfly-8.1.0.Final\standalone\deployments\jboss-ejb3-ear.ear to C:\Users\Deepak Vohra\.m2\repository\org\jboss\ejb3\jboss-ejb3-ear\1.0.0\jboss-ejb3-e
[INFO] Installing C:\Users\Deepak Vohra\Eclipse\workspace\jboss-ejb3\jboss-ejb3-ear\pom.xml to C:\Users\Deepak Vohra\.m2\repository\org\jboss\ejb3\jboss-ejb3-ear\1.0.0\jbo
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Summary:
[INFO]
[INFO] jboss-ejb3 ........................................ SUCCESS [  1.202 s]
[INFO] jboss-ejb3: EJB Module ............................ SUCCESS [  8.725 s]
[INFO] jboss-ejb3 - web .................................. SUCCESS [ 13.660 s]
[INFO] jboss-ejb3: EAR Module ............................ SUCCESS [  7.537 s]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 32.912 s
[INFO] Finished at: 2015-03-17T14:16:41-08:00
[INFO] Final Memory: 22M/189M
[INFO] ------------------------------------------------------------------------
```

Start the WildFly 8.x server if not already started. The `jboss-ejb3.ear` file gets deployed to the WildFly 8.x server and the `jboss-ejb3-web` context gets registered. The `jboss-ejb3.ear.deployed` file gets generated in the `deployments` directory, as shown in the following screenshot:

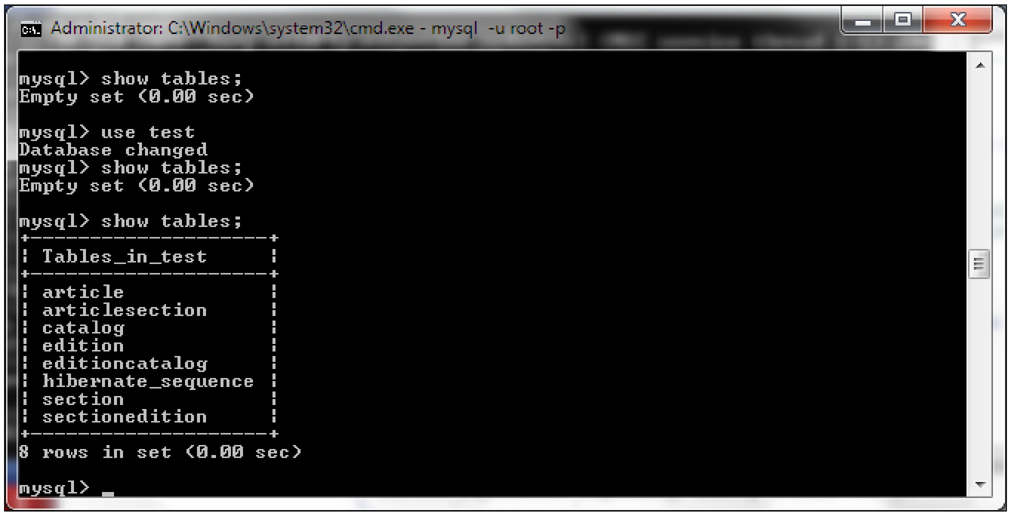


Name	Date modified	Type	Size
jboss-ejb3-ear.ear	10/29/2014 12:29 ...	EAR File	11,882 KB
jboss-ejb3-ear.ear.deployed	10/29/2014 12:29 ...	DEPLOYED File	1 KB

The `EntityManager em` persistence unit gets registered and the JNDI bindings for the `CatalogSessionBeanFacade` session bean gets generated:

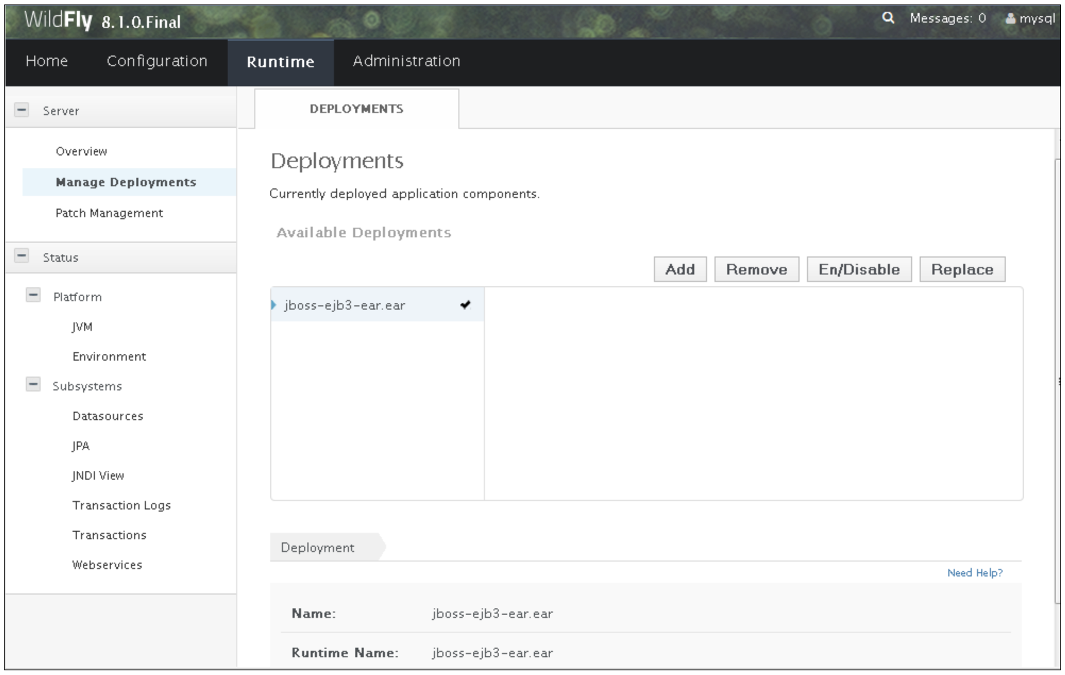
```
Starting Persistence Unit (phase 1 of 2) Service 'jboss-ejb3-ear.ear/
jboss-e
jb3-ejb.jar#em'
12:30:32,047 INFO  [org.hibernate.jpa.internal.util.LogHelper]
(ServerService Th
read Pool -- 50) HHH000204: Processing PersistenceUnitInfo [
  name: em
...]
```

The MySQL database tables for the entities get created, as shown in the following screenshot:


Administrator: C:\Windows\system32\cmd.exe - mysql -u root -p

```
mysql> show tables;
Empty set (0.00 sec)

mysql> use test
Database changed
mysql> show tables;
Empty set (0.00 sec)

mysql> show tables;
+---------------------+
| Tables_in_test      |
+---------------------+
| article             |
| articlesection      |
| catalog             |
| edition             |
| editioncatalog      |
| hibernate_sequence  |
| section             |
| sectionedition      |
+---------------------+
8 rows in set (0.00 sec)

mysql> _
```

To log in to the WildFly 8 administration console, open `http://localhost:8080` in any web browser. Click on the **Administration Console** link. Specify **User Name** and **Password** and click on **Log In**. Select the **Runtime** tab. The `jboss-ejb3.ear` application is listed as deployed in the **Deployments | Manage Deployments** section:



Deployments

Currently deployed application components.

Available Deployments

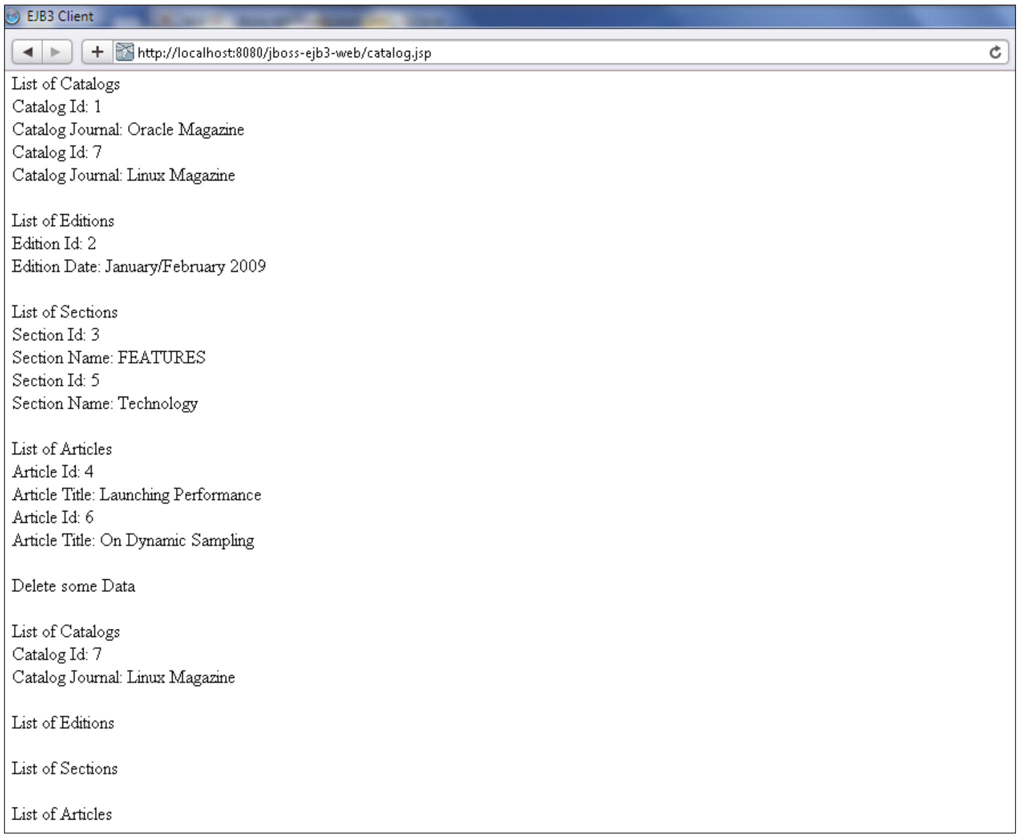
jboss-ejb3-ear.ear ✓

Name: jboss-ejb3-ear.ear

Runtime Name: jboss-ejb3-ear.ear

Running the JSP client

Next, open `http://localhost:8080/jboss-ejb3-web/catalog.jsp` and run the JSP client. The **List of Catalogs** gets displayed. The `deleteSomeData` method deletes `Catalog` for `Oracle Magazine`. As the `Linux Magazine` catalog does not have any data, the empty list gets displayed, as shown in the following screenshot:



```
List of Catalogs
Catalog Id: 1
Catalog Journal: Oracle Magazine
Catalog Id: 7
Catalog Journal: Linux Magazine

List of Editions
Edition Id: 2
Edition Date: January/February 2009

List of Sections
Section Id: 3
Section Name: FEATURES
Section Id: 5
Section Name: Technology

List of Articles
Article Id: 4
Article Title: Launching Performance
Article Id: 6
Article Title: On Dynamic Sampling

Delete some Data

List of Catalogs
Catalog Id: 7
Catalog Journal: Linux Magazine

List of Editions

List of Sections

List of Articles
```

Configuring a Java EE 7 Maven project

The default JBoss Java EE EAR project created is a Java EE 6 project. If a Java EE 7 project is required to avail of the EJB 3.2, Servlet 3.1, JSF 2.2, and Hibernate JPA 2.1 APIs, the `pom.xml` for the `ejb` module and the `web` module subprojects should include the **BOM** (**Bill of Materials**) for Java EE 7 and the Nexus repository:

```
<repositories>
  <repository>
    <id>JBoss Repository</id>
```

```
        <url>https://repository.jboss.org/nexus/content/groups/public/
    </url>
      </repository>
    </repositories>
    <dependencyManagement>
      <dependencies>
          <dependency>
            <groupId>org.jboss.spec</groupId>
            <artifactId>jboss-javaee-7.0</artifactId>
            <version>1.0.0.Final</version>
            <type>pom</type>
            <scope>import</scope>
          </dependency>
      </dependencies>
    </dependencyManagement>
```

In addition the `pom.xml` for the `ejb` module and `web` module subprojects should specify the dependencies for the EJB 3.2, JSF 2.2, Servlet 3.1, and Hibernate JPA 2.1 specifications, as required, instead of the dependencies for the EJB 3.1, JSF 2.1, Servlet 3.0, and Hibernate JPA 2.0:

```
    <dependency>
      <groupId>org.jboss.spec.javax.ejb</groupId>
      <artifactId>jboss-ejb-api_3.2_spec</artifactId>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>org.hibernate.javax.persistence</groupId>
      <artifactId>hibernate-jpa-2.1-api</artifactId>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>org.jboss.spec.javax.servlet</groupId>
      <artifactId>jboss-servlet-api_3.1_spec</artifactId>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>org.jboss.spec.javax.faces</groupId>
      <artifactId>jboss-jsf-api_2.2_spec</artifactId>
      <scope>provided</scope>
    </dependency>
```

Summary

In this chapter, we used the JBoss Tools plugin 4.2 in Eclipse Luna to generate a Java EE project for an EJB 3.x application in Eclipse IDE for Java EE Developers. We created entities to create a `Catalog` and used the Hibernate persistence provider to map the entities to the MySQL 5.6 database. Subsequently, we created a session bean façade for the entities. In the session bean, we created a catalog using the `EntityManager` API. We also created a JSP client to invoke the session bean facade using the local JNDI lookup and subsequently invoke the session bean methods to display database data. We used Maven to build the `EJB`, `Web`, and `EAR` modules and deploy the `EAR` module to WildFly 8.1. We ran the JSP client in a browser to fetch and display the data from the MySQL database. In the next chapter, we will discuss another database persistence technology: **Hibernate**.

2
Developing Object/Relational Mapping with Hibernate 4

Hibernate is an object/relational mapping Java framework with which POJO domain objects can be mapped to a relational database. Though Hibernate has evolved beyond just being a object/relational framework, we will discuss only its object/relational mapping aspect. Hibernate's advantage over traditional **Java Database Connectivity** (**JDBC**) is that it provides the mapping of Java objects to relational database tables and the mapping of Java data types to SQL data types without a developer to provide the mapping, which implies having to make fewer API calls and the elimination of SQL statements. Hibernate provides loose coupling with the database with vendor-specific mapping using dialect configuration. Hibernate implements features such as caching, query tuning, and connection pooling, which have to be implemented by a developer in JDBC.

The chapter has the following sections:

- Creating a Java EE web project
- Creating a Hibernate XML Mapping file
- Creating a properties file
- Creating a Hibernate configuration file
- Creating JSPs for CRUD
- Creating the JavaBean class
- Exporting schema
- Creating table data
- Retrieving table data
- Updating a table row

- Deleting a table row
- Installing the Maven project
- Running schema export
- Creating table rows
- Retrieving table data
- Updating table
- Deleting the table row

Setting up the environment

We need to install the following software (the same as in *Chapter 1, Getting Started with EJB 3.x*):

- **WildFly 8.1.0.Final**: Download `wildfly-8.1.0.Final.zip` from `http://wildfly.org/downloads/`.
- **MySQL 5.6 Database-Community Edition**: Download this edition from `http://dev.mysql.com/downloads/mysql/`. When installing MySQL, also install **Connector/J**.
- **Eclipse IDE for Java EE Developers**: Download Eclipse Luna from `https://www.eclipse.org/downloads/packages/release/Luna/SR1`.
- **JBoss Tools (Luna) 4.2.0.Final (or the latest version)**: Install this as a plugin to Eclipse from the Eclipse Marketplace (`http://tools.jboss.org/downloads/installation.html`).
- **Apache Maven**: Download version 3.05 or higher from `http://maven.apache.org/download.cgi`.
- **Java 7**: Download Java 7 from `http://www.oracle.com/technetwork/java/javase/downloads/index.html?ssSourceSiteId=ocomcn`.

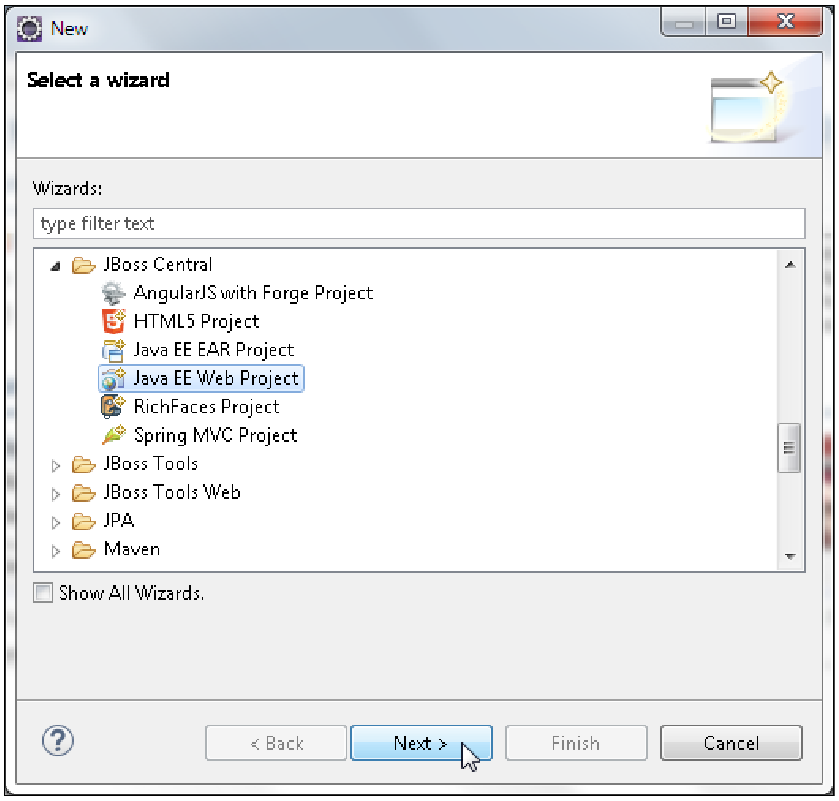
Set the same environment variables as in *Chapter 1, Getting Started with EJB 3.x*: `JAVA_HOME`, `JBOSS_HOME`, `MAVEN_HOME`, and `MYSQL_HOME`. Add `%JAVA_HOME%/bin`, `%MAVEN_HOME%/bin`, `%JBOSS_HOME%/bin`, and `%MYSQL_HOME%/bin` to the `PATH` environment variable.

Create a WildFly 8.1.0 runtime as discussed in *Chapter 1, Getting Started with EJB 3.x*.

Creating a Java EE web project

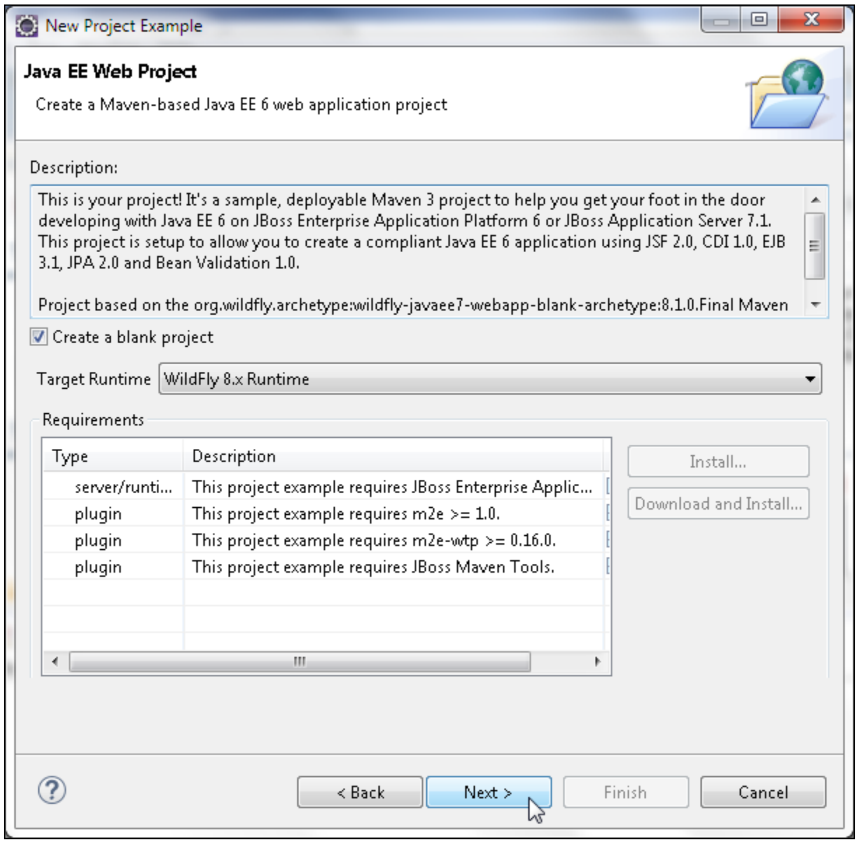
In this section, we will create **Java EE Web Project** in Eclipse IDE. Perform the following steps to accomplish this task:

1. Select **File** | **New** | **Other**. In the **New** window, select **JBoss Central** | **Java EE Web Project** and click on **Next**, as shown in the following screenshot:



The **Java EE Web Project** wizard gets started. A test is run for the requirements, which includes a JBoss server runtime, the JBoss Tools runtime, and the **m2e** and **m2eclipes-wtp** plugins.

2. Select the **Create a blank project** checkbox and **Target Runtime WildFly 8.x Runtime** and click on **Next**, as shown in the following screenshot. Even though Java EE Web Project indicates the Java EE version as Java EE 6, a Java EE 7 web project is actually created.



Java EE Web Project

Create a Maven-based Java EE 6 web application project

Description:

This is your project! It's a sample, deployable Maven 3 project to help you get your foot in the door developing with Java EE 6 on JBoss Enterprise Application Platform 6 or JBoss Application Server 7.1. This project is setup to allow you to create a compliant Java EE 6 application using JSF 2.0, CDI 1.0, EJB 3.1, JPA 2.0 and Bean Validation 1.0.

Project based on the org.wildfly.archetype:wildfly-javaee7-webapp-blank-archetype:8.1.0.Final Maven

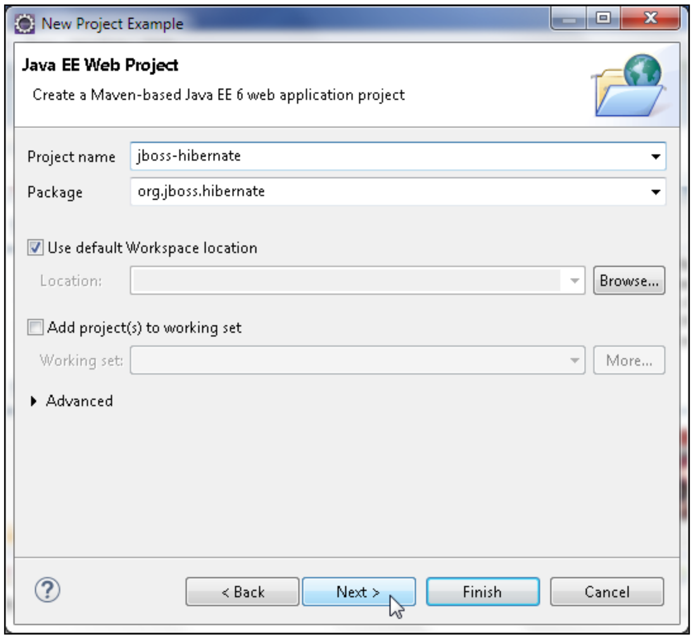
☑ Create a blank project

Target Runtime: WildFly 8.x Runtime

Requirements

Type	Description
server/runti...	This project example requires JBoss Enterprise Applic...
plugin	This project example requires m2e >= 1.0.
plugin	This project example requires m2e-wtp >= 0.16.0.
plugin	This project example requires JBoss Maven Tools.

3. Specify **Project Name** (`jboss-hibernate`) and **Package** (`org.jboss.hibernate`), and click on **Next**, as shown in the following screenshot:


New Project Example

Java EE Web Project

Create a Maven-based Java EE 6 web application project

Project name: jboss-hibernate

Package: org.jboss.hibernate

☑ Use default Workspace location

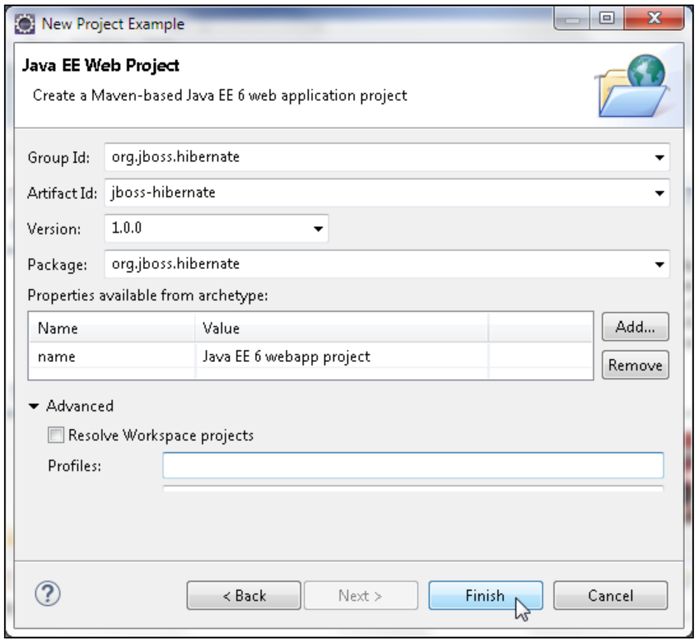
Location:

☐ Add project(s) to working set

Working set:

Advanced

4. Specify **Group Id** (`org.jboss.hibernate`), **Artifact Id** (`jboss-hibernate`), **Version** (`1.0.0`), and **Package** (`org.jboss.hibernate`), as shown in the following screenshot. Click on the **Finish** button.


New Project Example

Java EE Web Project

Create a Maven-based Java EE 6 web application project

Group Id: org.jboss.hibernate

Artifact Id: jboss-hibernate

Version: 1.0.0

Package: org.jboss.hibernate

Properties available from archetype:

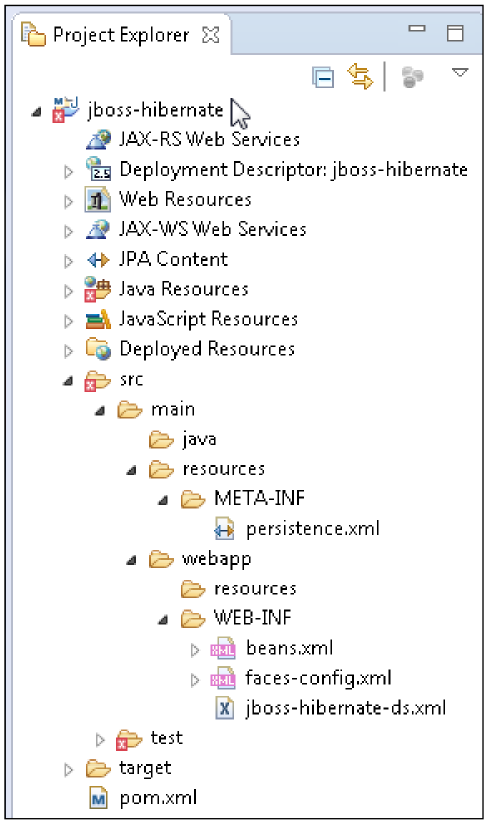
Name	Value
name	Java EE 6 webapp project

Advanced

☐ Resolve Workspace projects

Profiles:

5. The `jboss-hibernate` project gets created in Eclipse and gets added to **Project Explorer**, as shown in the following screenshot:

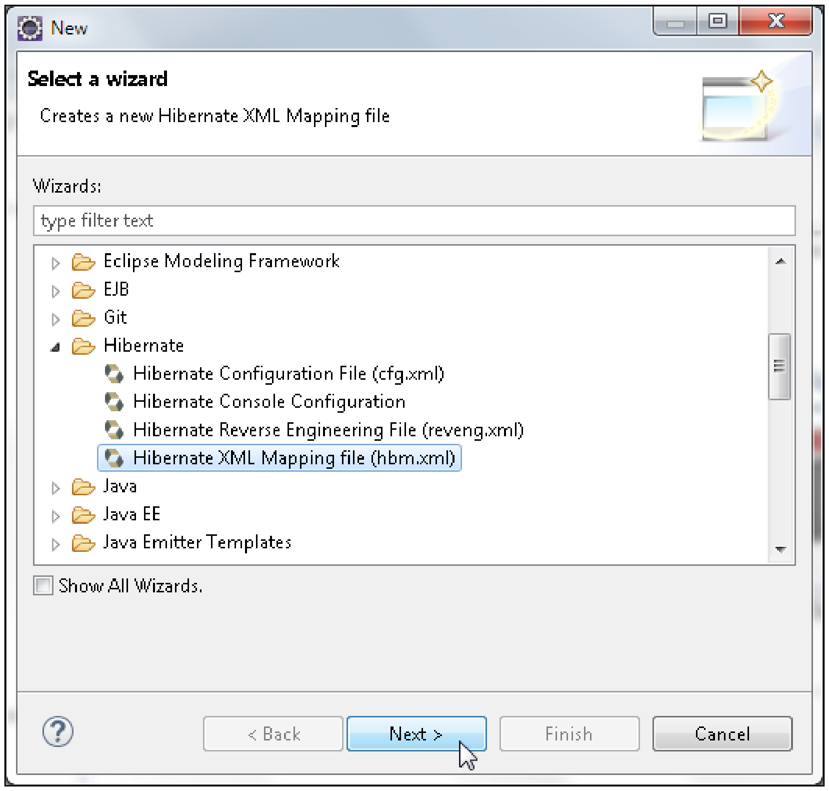


Creating a Hibernate XML Mapping file

Hibernate provides transparent mapping between a persistence class and a relational database using an XML mapping file. The actual storing and loading of objects of the persistence class is based on the mapping metadata. Perform the following steps to accomplish this:

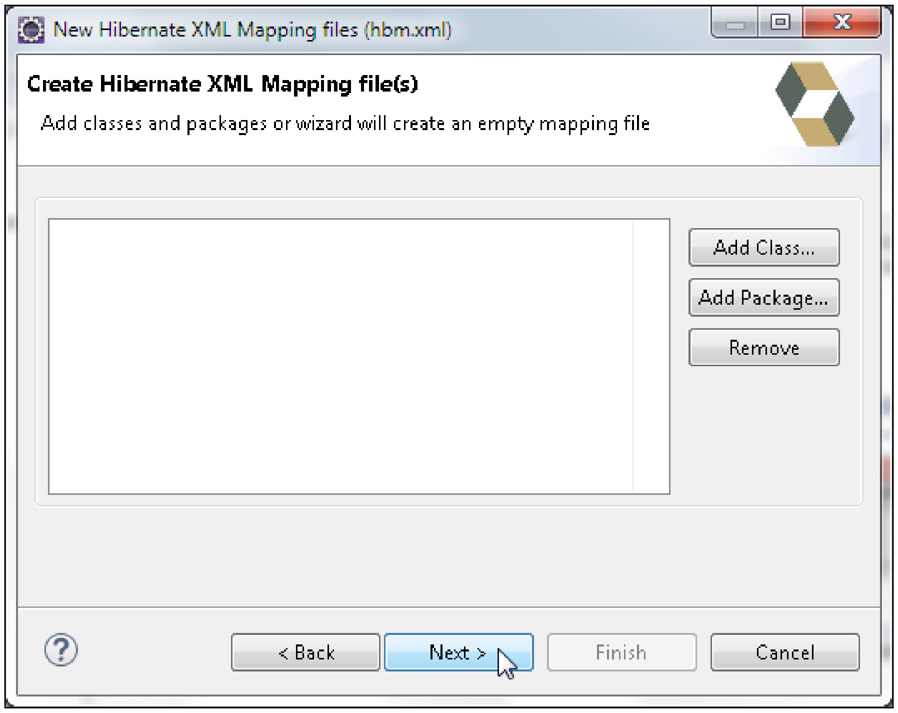
1. To create a Hibernate XML Mapping file, select **File** | **New** | **Other**.

2. In the **New** window, select **Hibernate | Hibernate XML Mapping File (hbm.xml)** and click on **Next**, as shown in the following screenshot:


New
Select a wizard
Creates a new Hibernate XML Mapping file
Wizards:
type filter text
Eclipse Modeling Framework
EJB
Git
Hibernate
Hibernate Configuration File (cfg.xml)
Hibernate Console Configuration
Hibernate Reverse Engineering File (reveng.xml)
Hibernate XML Mapping file (hbm.xml)
Java
Java EE
Java Emitter Templates
Show All Wizards.
< Back
Next >
Finish
Cancel

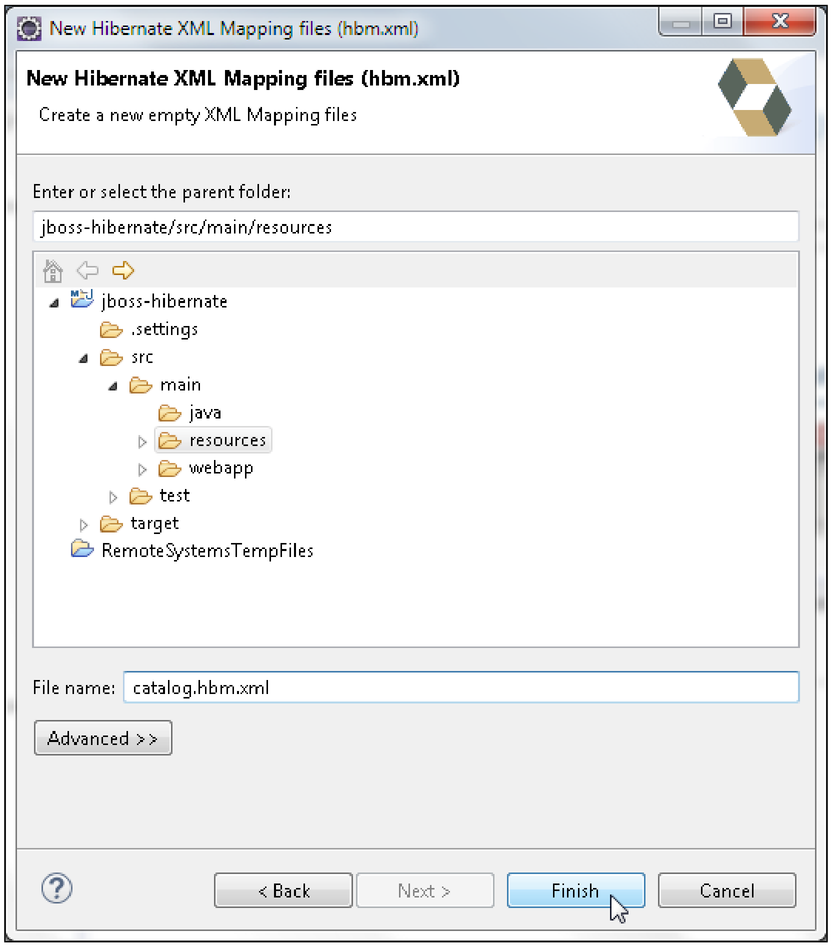
The **New Hibernate XML Mapping files** wizard gets started. As we have not yet defined any persistence classes to map, we will first create an empty XML mapping file.

3. In **Create Hibernate XML Mapping file(s)**, click on **Next**, as shown in the following screenshot:


New Hibernate XML Mapping files (hbm.xml)
Create Hibernate XML Mapping file(s)
Add classes and packages or wizard will create an empty mapping file
Add Class...
Add Package...
Remove
< Back
Next >
Finish
Cancel

The resources in the `src/main/resources` directory are in the classpath of a Hibernate application.

4. Select the `jboss-hibernate` | `src` | `main` | `resources` folder and specify **File name** as `catalog.hbm.xml`, as shown in the following screenshot. Click on **Finish**.


New Hibernate XML Mapping files (hbm.xml)
New Hibernate XML Mapping files (hbm.xml)
Create a new empty XML Mapping files
Enter or select the parent folder:
jboss-hibernate/src/main/resources
jboss-hibernate
.settings
src
main
java
resources
webapp
test
target
RemoteSystemsTempFiles
File name: catalog.hbm.xml
Advanced >>
< Back
Next >
Finish
Cancel

The `catalog.hbm.xml` mapping file gets added to the `resources` directory. The root element of the mapping file is `hibernate-mapping`. The persistence classes are configured using the `<class/>` element. Add a `<class/>` element to the `org.jboss.hibernate.model.Catalog` class specified with the `name` attribute in the `<class/>` element. We have yet to create the persistence class, which would not be required if we were just exporting a schema to a relational database but is required to store or load any POJO objects. Specify a table that the class is to be mapped to with the `table` attribute of the `<class/>` element. With the specified mapping, an instance of the `Catalog` class is mapped to a row in the `CATALOG` database table. A mapped persistence class is required to specify the primary key column of the table it is mapped to. The primary key column is mapped to an identifier property in the persistence class. The primary key column and the identifier property are specified using the `<id/>` element. The column attribute specifies the primary key column; the name attribute specifies the identifier property in the persistence class being mapped; and the type attribute specifies the Hibernate type. The `<generator/>` subelement of the `<id/>` element specifies the primary key generation strategy. Some built-in generation strategies are available and different relational databases support different ID generation strategies.

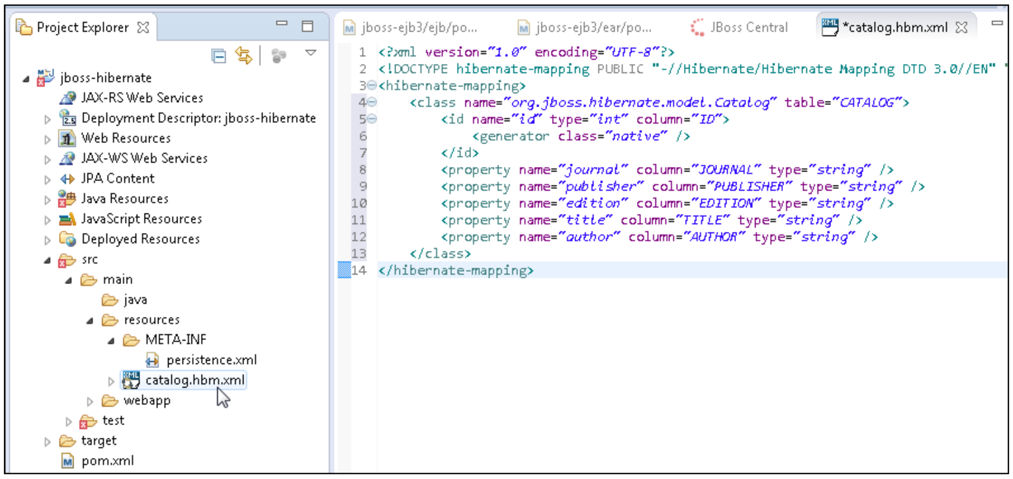
As we are using the MySQL database, which supports identity columns using `AUTO_INCREMENT`, we can use the generation strategy as `identity` or `native`. An identity column is a table column of the `INTEGER` type, with `AUTO_INCREMENT` and `PRIMARY KEY` or `UNIQUE KEY` specified, such as `id INTEGER AUTO_INCREMENT PRIMARY KEY` or `id INTEGER AUTO_INCREMENT UNIQUE KEY`.

Add JavaBean properties using the `<property/>` element. The JavaBean properties in the persistence class are mapped to the columns of the database table. The `name` attribute of the `<property/>` element specifies the property name and is the only required attribute. The `column` attribute specifies the database table column name; the default column name is the property name. The `type` attribute specifies the Hibernate type. If the `type` attribute is not specified, Hibernate finds the type, which might not be exactly the same as the actual type specified in the JavaBean class. To distinguish between similar Hibernate types, it is recommended that you specify the type in the `property` element. Add the `<property/>` elements `journal`, `publisher`, `edition`, `title`, and `author` of the type `string` and mapped to the columns `JOURNAL`, `PUBLISHER`, `EDITION`, `TITLE`, and `AUTHOR` respectively. The `catalog.hbm.xml` mapping file is listed in the following code:

```
<?xml version="1.0"?><!DOCTYPE hibernate-mapping PUBLIC"-//Hibernate/
Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
```

```
    <class name="org.jboss.hibernate.model.Catalog" table="CATALOG">
      <id name="id" type="int" column="ID">
        <generator class="native" />
      </id>
      <property name="journal" column="JOURNAL" type="string" />
      <property name="publisher" column="PUBLISHER" type="string" />
      <property name="edition" column="EDITION" type="string" />
      <property name="title" column="TITLE" type="string" />
      <property name="author" column="AUTHOR" type="string" />
    </class>
  </hibernate-mapping>
```

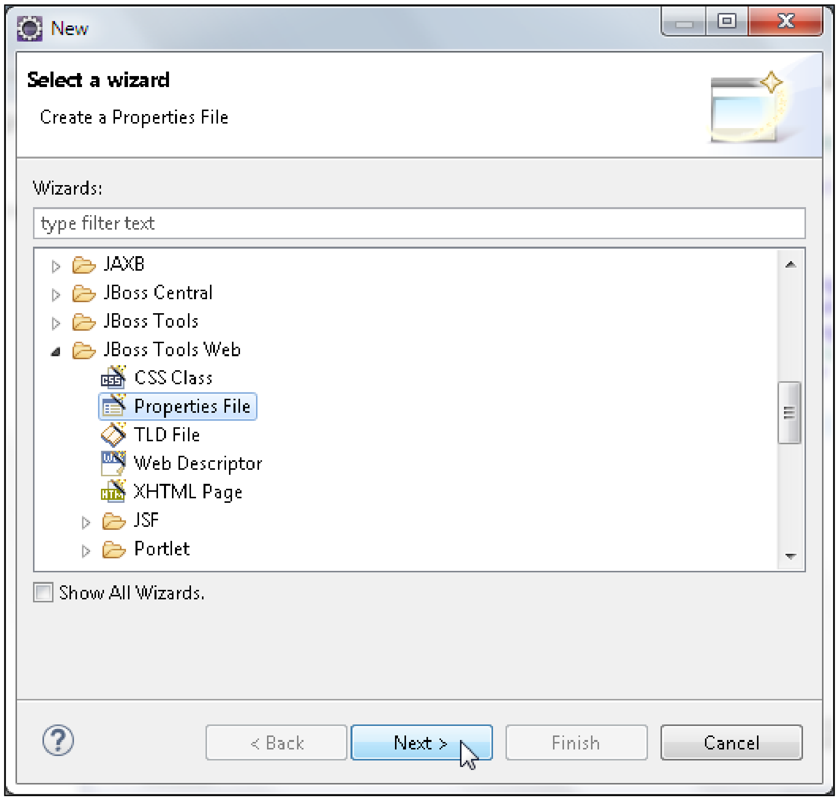
The mapping file is shown in the `jboss-hibernate` project, as shown in the following screenshot:



```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN" "
<hibernate-mapping>
    <class name="org.jboss.hibernate.model.Catalog" table="CATALOG">
        <id name="id" type="int" column="ID">
            <generator class="native" />
        </id>
        <property name="journal" column="JOURNAL" type="string" />
        <property name="publisher" column="PUBLISHER" type="string" />
        <property name="edition" column="EDITION" type="string" />
        <property name="title" column="TITLE" type="string" />
        <property name="author" column="AUTHOR" type="string" />
    </class>
</hibernate-mapping>
```

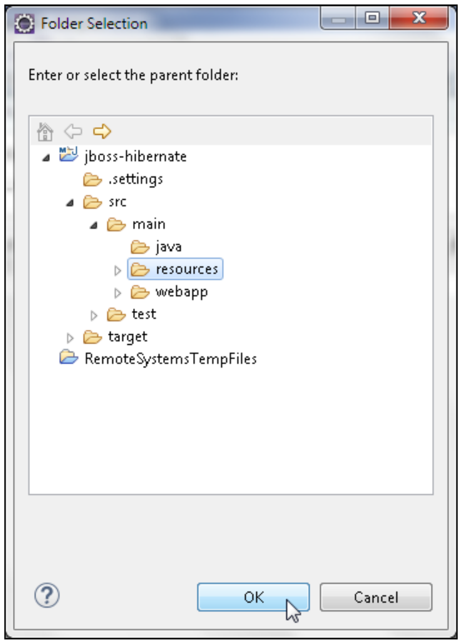
Creating a properties file

1. The Hibernate XML Mapping file defines the mapping of the persistence or class or classes with the relational database. The connection parameters used to connect to the database can be configured in a properties file or an XML configuration file, or both. To create a properties file, select **File** | **New** | **Other**. In the **New** wizard, select **JBoss Tools Web** | **Properties File** and click on **Next**, as shown in the following screenshot:

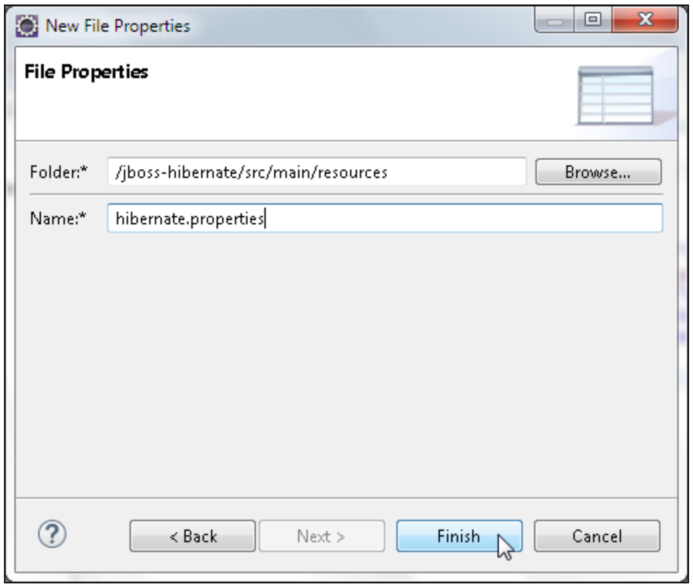


The **New File Properties** wizard gets started.

2. Click on **Browser** for the **Folder** field to select a folder. In **Folder Selection**, select the `jboss-hibernate` | `src` | `main` | `resources` folder and click on **OK**, as shown in the following screenshot:



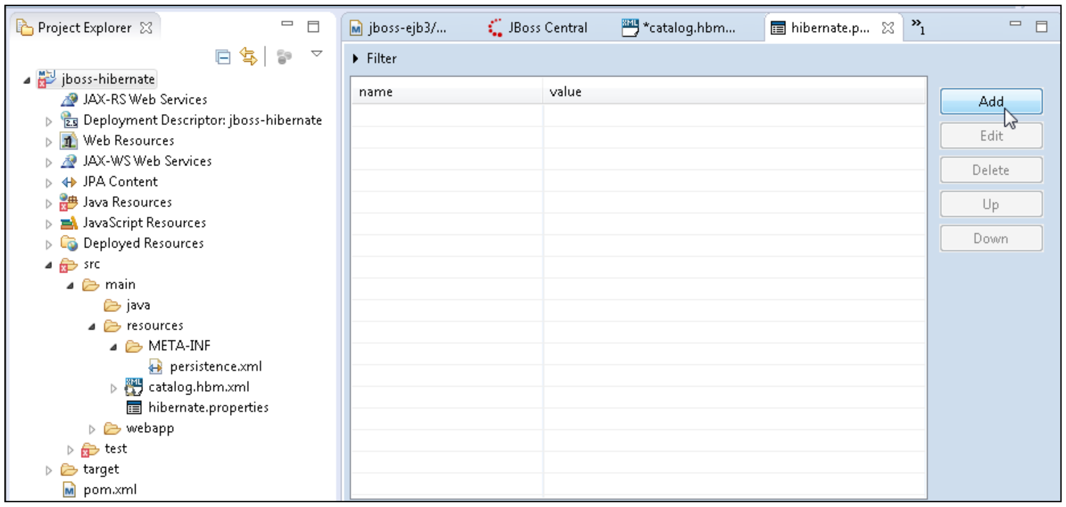
3. Specify **Name*** as `hibernate.properties` and click on **Finish**, as shown in the following screenshot:



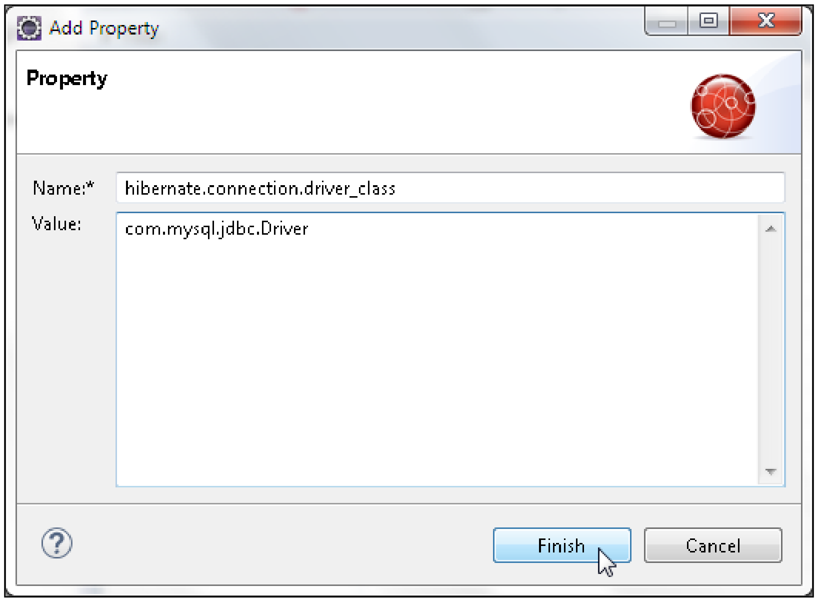
Folder:* /jboss-hibernate/src/main/resources
Name:* hibernate.properties

The `hibernate.properties` file gets added to the `resources` folder.

4. In the `hibernate.properties` table, click on the **Add** button to add a property, as shown in the following screenshot:

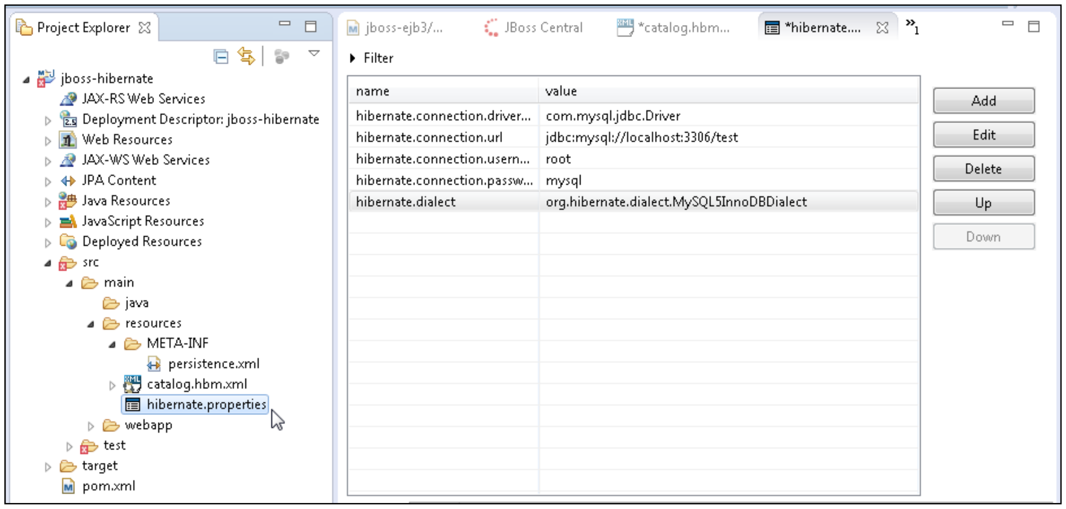


5. Add the `hibernate.connection.driver_class` property with **Value** as `com.mysql.jdbc.Driver` and click on **Finish**, as shown in the following screenshot:



Name:* hibernate.connection.driver_class
Value: com.mysql.jdbc.Driver

6. Similarly, add other properties as shown in the `hibernate.properties` table. The `hibernate.connection.url` property specifies the connection URL, and the `hibernate.dialect` property specifies the database dialect to be used as `MySQL5InnoDBDialect`, as shown in the following screenshot:



name	value
hibernate.connection.driver...	com.mysql.jdbc.Driver
hibernate.connection.url	jdbc:mysql://localhost:3306/test
hibernate.connection.usern...	root
hibernate.connection.passw...	mysql
hibernate.dialect	org.hibernate.dialect.MySQL5InnoDBDialect

The `hibernate.properties` file is listed in the following code. The `username` and `password` attributes can be different than the ones listed:

```
hibernate.connection.driver_class=com.mysql.jdbc.Driver
hibernate.connection.url=jdbc:mysql://localhost:3306/test
hibernate.connection.username=root
hibernate.connection.password=mysql16
hibernate.dialect=org.hibernate.dialect.MySQL5InnoDBDialect
```

Creating a Hibernate configuration file

The Hibernate configuration can be specified in the **Hibernate Configuration File (cfg.xml)**, which has more configuration parameters than the properties file.

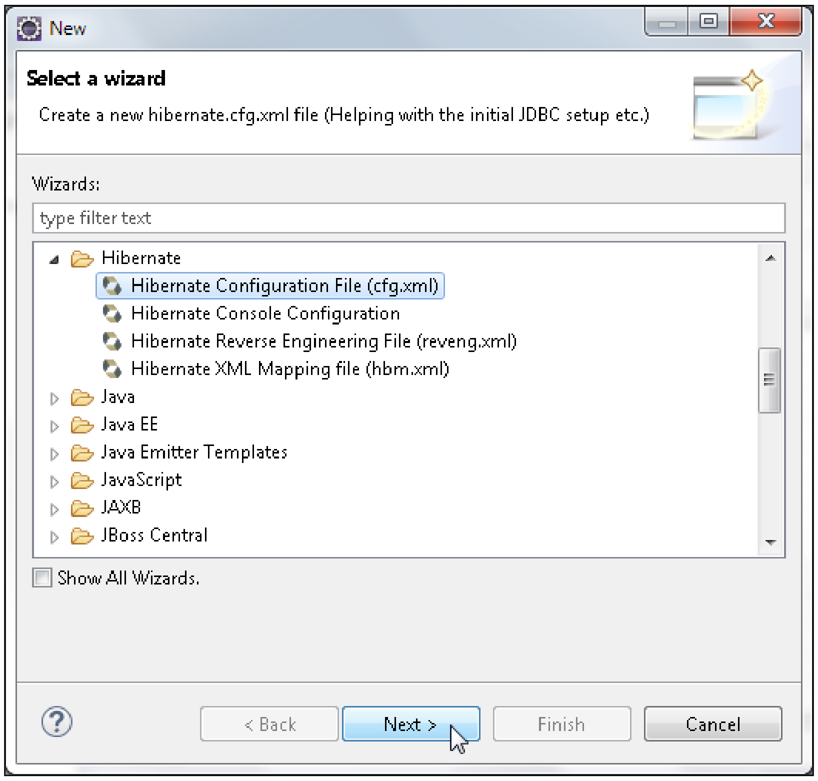
Either the properties file or the configuration file can be used to specify the configuration, or both can be used. If both are provided, the configuration file overrides the properties file for the configuration parameters specified in both.

The Hibernate XML configuration file has the following advantages over the properties file:

- The Hibernate configuration file is more convenient when tuning the Hibernate cache. The Hibernate configuration file has the provision to configure the Hibernate XML Mapping files.
- For exporting a schema to a database using the `SchemaExport` tool, just the properties file would suffice, but for object/relational mapping of a persistence class, the Hibernate XML configuration file is a better option.

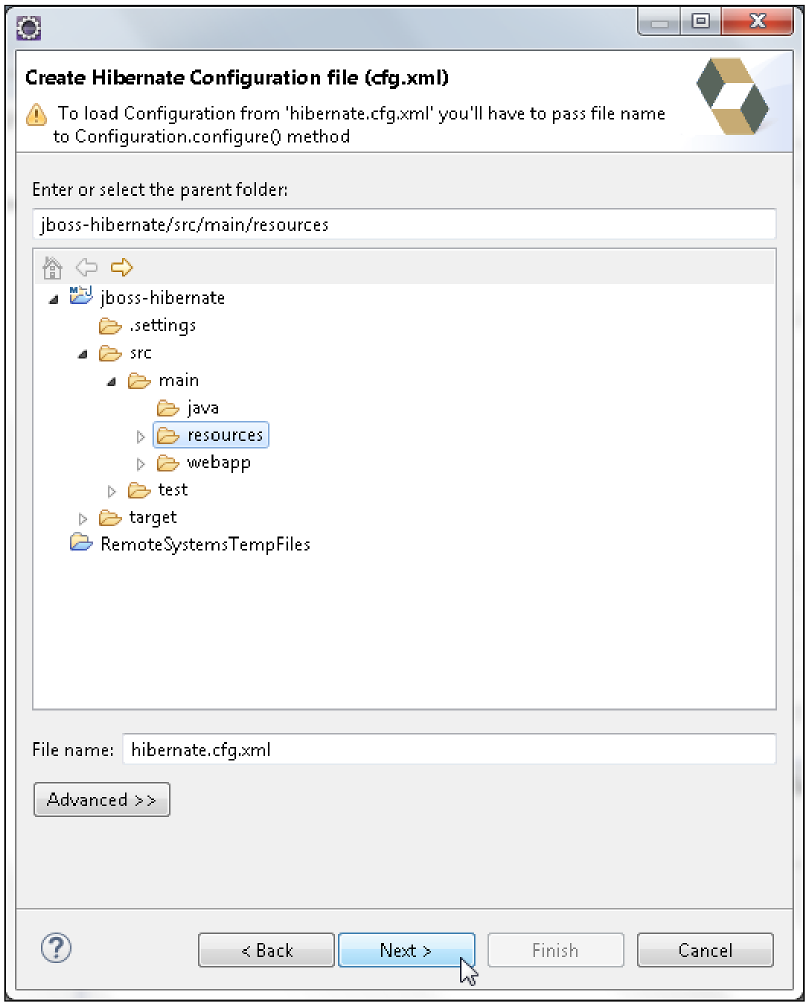
The following are the steps to create a Hibernate configuration file:

1. Select **File** | **New** | **Other**. In **New**, select **Hibernate** | **Hibernate Configuration File (cfg.xml)** and click on **Next**, as shown in the following screenshot:


New
Select a wizard
Create a new hibernate.cfg.xml file (Helping with the initial JDBC setup etc.)
Wizards:
type filter text
Hibernate
Hibernate Configuration File (cfg.xml)
Hibernate Console Configuration
Hibernate Reverse Engineering File (reveng.xml)
Hibernate XML Mapping file (hbm.xml)
Java
Java EE
Java Emitter Templates
JavaScript
JAXB
JBoss Central
Show All Wizards.
< Back
Next >
Finish
Cancel

The **Create Hibernate Configuration file** wizard gets started.

2. Select the `jboss-hibernate | src | main | resources` folder, specify **File name** as `hibernate.cfg.xml`, and click on `Next`, as shown in the following screenshot:



Create Hibernate Configuration file (cfg.xml)

To load Configuration from 'hibernate.cfg.xml' you'll have to pass file name to Configuration.configure() method

Enter or select the parent folder:

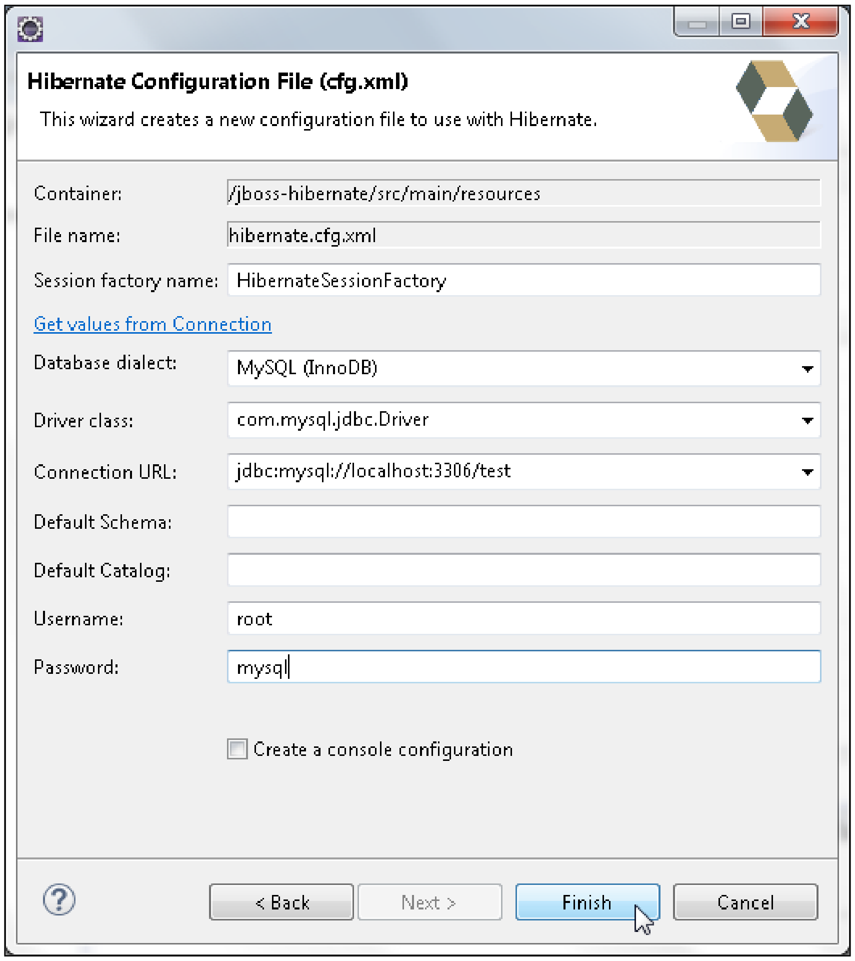
jboss-hibernate/src/main/resources

```
jboss-hibernate
    .settings
    src
        main
            java
            resources
            webapp
        test
    target
RemoteSystemsTempFiles
```

File name: hibernate.cfg.xml

3. In the **Hibernate Configuration File** wizard, specify **Session factory name** (`HibernateSessionFactory`). A session factory is a factory that is used to generate client sessions to Hibernate. A session factory stores the metadata for the object/relational mapping.
4. Select **Database dialect** as `MySQL 5 (InnoDB)`. Select **Driver class** as `com.mysql.jdbc.Driver`.
5. Specify **Connection URL** as `jdbc:mysql://localhost:3306/test`.

6. Specify the **Username** and **Password** and click on **Finish**, as shown in the following screenshot:



Hibernate Configuration File (cfg.xml)

This wizard creates a new configuration file to use with Hibernate.

Container:	/jboss-hibernate/src/main/resources
File name:	hibernate.cfg.xml
Session factory name:	HibernateSessionFactory
Get values from Connection	
Database dialect:	MySQL (InnoDB)
Driver class:	com.mysql.jdbc.Driver
Connection URL:	jdbc:mysql://localhost:3306/test
Default Schema:	
Default Catalog:	
Username:	root
Password:	mysql

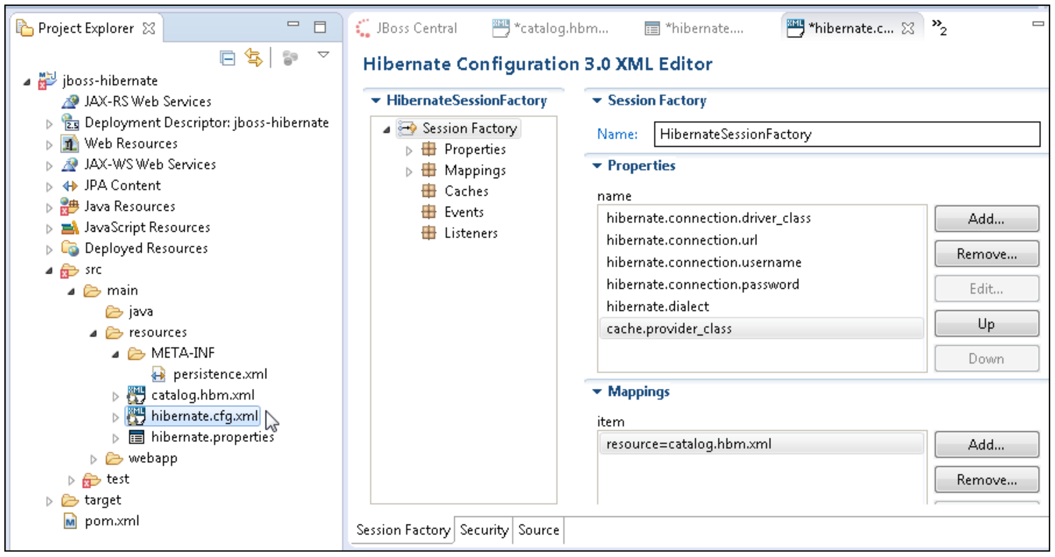
☐ Create a console configuration

Hibernate provides transaction-level caching of persistence data in a session by default. Hibernate has the provision for a query-level cache, which is turned off by default, to frequently run queries. Hibernate also has the provision for a second-level cache on the `SessionFactory` level or on the cluster level. The second-level cache is configured in the `hibernate.cfg.xml` file using the `hibernate.cache.provider_class` property. Classes that specify `<cache/>` mapping have the second-level cache enabled by default. The second-level cache can be turned off by setting the `cache.provider_class` property to `org.hibernate.cache.NoCacheProvider`. Specify the Hibernate XML Mapping file using the `<mapping/>` element with the `resource` attribute set to `catalog.hbm.xml`.

The `hibernate.cfg.xml` file is listed in the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC"-//Hibernate/Hibernate
Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory name="HibernateSessionFactory">
    <property name="hibernate.connection.driver_class">com.mysql.jdbc.
Driver</property>
    <property name="hibernate.connection.url">jdbc:mysql://
localhost:3306/test</property>
    <property name="hibernate.connection.username">root</property>
    <property name="hibernate.connection.password">mysql16</property>
    <property name="hibernate.dialect">org.hibernate.dialect.
MySQL5InnoDBDialect</property>
    <property name="cache.provider_class">org.hibernate.cache.
NoCacheProvider</property>
    <mapping resource="catalog.hbm.xml" />
  </session-factory>
</hibernate-configuration>
```

The `hibernate.cfg.xml` file is shown in the `jboss-hibernate` project folder, as follows:



Session Factory
Name: HibernateSessionFactory

Properties

name
hibernate.connection.driver_class
hibernate.connection.url
hibernate.connection.username
hibernate.connection.password
hibernate.dialect
cache.provider_class

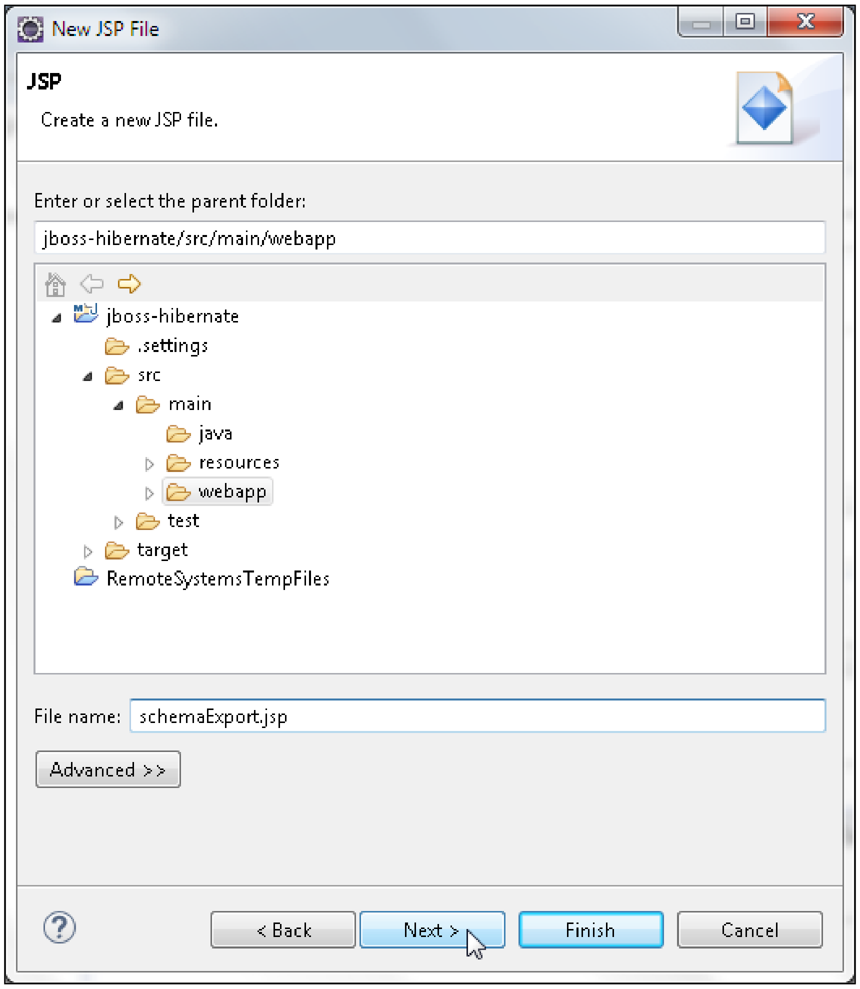
Mappings

item
resource=catalog.hbm.xml

Creating JSPs for CRUD

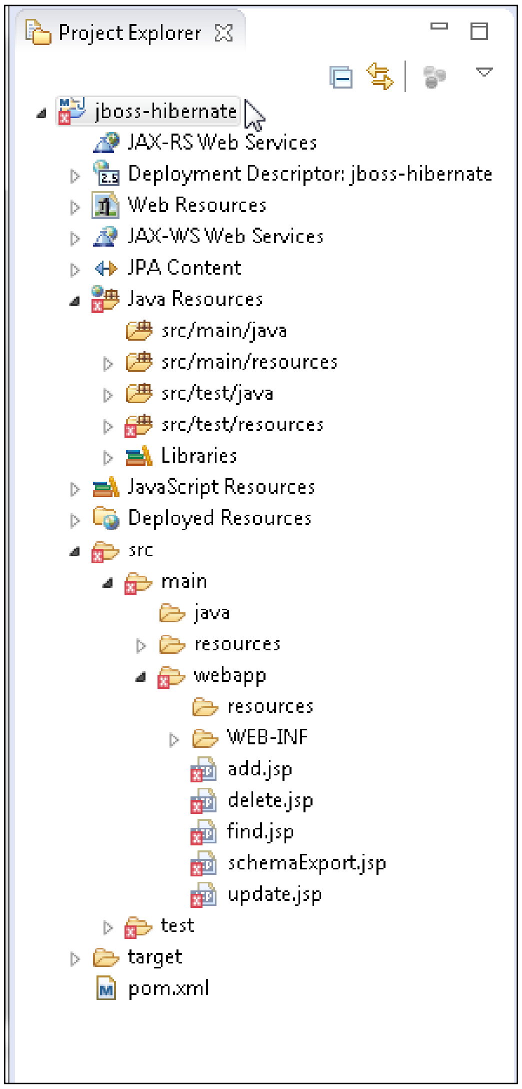
We have created the required configuration files for Hibernate. Next, we will create the JSPs to persist, load, update, and delete POJO domain objects, which are also referred to as **create, read, update, delete** (**CRUD**). Perform the following steps to accomplish this:

1. Select **File** | **New** | **Other**, and in **New**, select **Web** | **JSP** File and click on **Next**.
2. In **New JSP File**, select the `webapp` folder and specify **File name** as `schemaExport.jsp`. Click on **Next**, as shown in the following screenshot:



3. Select the **New JSP file (html)** template, which is also the default, and click on **Finish**. The `schemaExport.jsp` file gets added to the `webapp` folder.
4. Similarly, use `add.jsp` (to add table data), `find.jsp` (to find table data), `update.jsp` (to update a table row), and `delete.jsp` (to delete a table row).

The directory structure of the `jboss-hibernate` project is shown in the following screenshot. The JSP files might indicate an error, which will get fixed as the application is developed and the Maven dependencies are added.

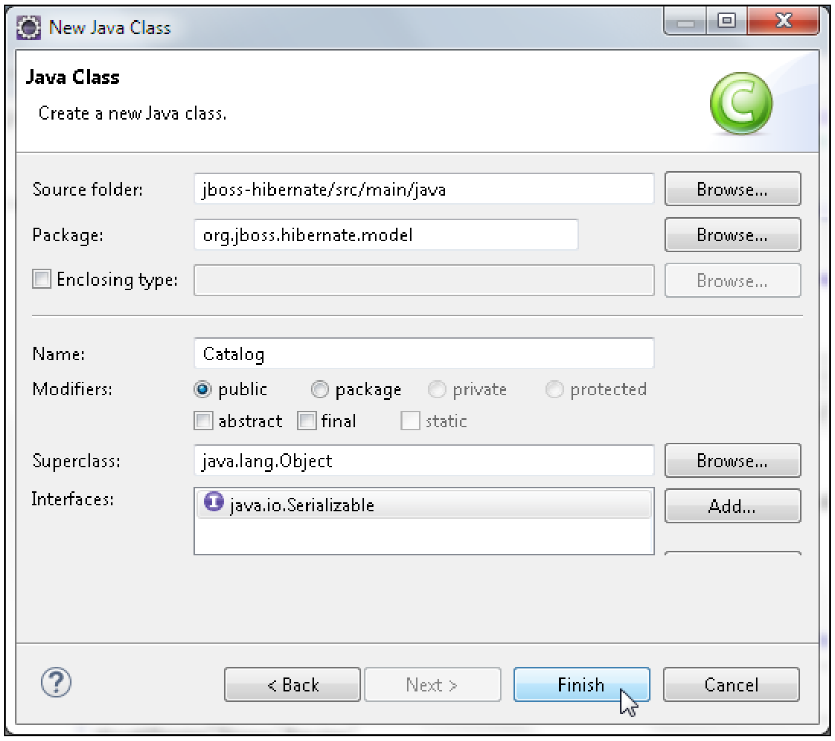


```
Project Explorer
jboss-hibernate
    JAX-RS Web Services
    Deployment Descriptor: jboss-hibernate
    Web Resources
    JAX-WS Web Services
    JPA Content
    Java Resources
        src/main/java
        src/main/resources
        src/test/java
        src/test/resources
        Libraries
    JavaScript Resources
    Deployed Resources
    src
        main
            java
            resources
            webapp
                resources
                WEB-INF
                add.jsp
                delete.jsp
                find.jsp
                schemaExport.jsp
                update.jsp
        test
    target
    pom.xml
```

Creating the JavaBean class

In this section, we create the JavaBean class to be persisted to the database. To accomplish this, perform the following steps:

1. Select **File** | **New** | **Other**, and in the **New** wizard, select **Class** and click on **Next**.
2. In the **New Java Class** wizard, specify **Source folder** as `jboss-hibernate/src/main/java` and specify **Package** as `org.jboss.hibernate.model`. Specify **Name** as `Catalog` and in **Interfaces**, add `java.io.Serializable`. Click on **Finish**, as shown in the following screenshot:



Source folder: jboss-hibernate/src/main/java
Package: org.jboss.hibernate.model
Enclosing type:
Name: Catalog
Modifiers: [✓ public] package private protected
abstract final static
Superclass: java.lang.Object
Interfaces: java.io.Serializable

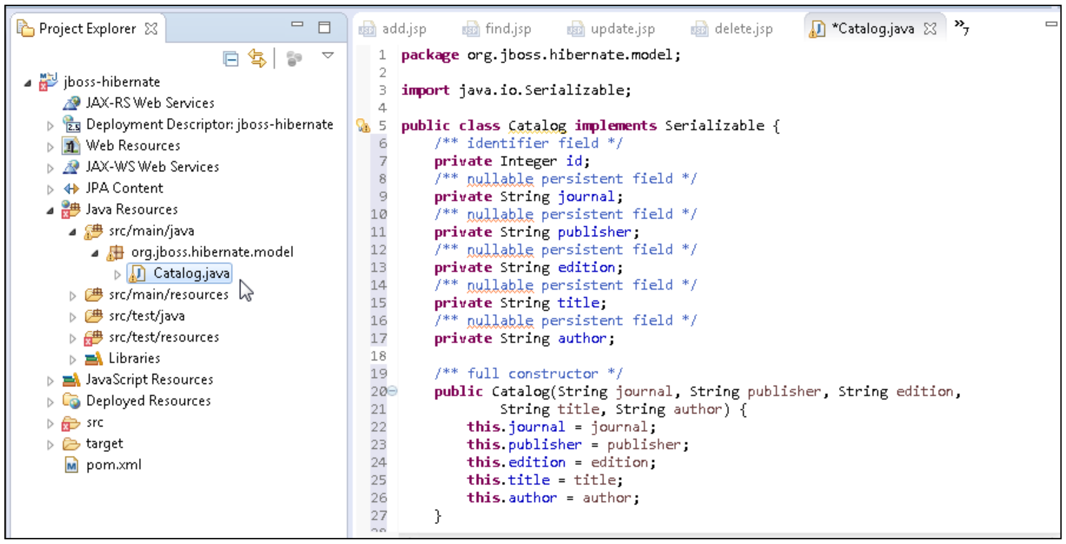
The `org.jboss.hibernate.model.Catalog` class is added to the `jboss-hibernate` project. In the `Catalog` class, declare the `id` property of the type `Integer`. The `id` property is mapped to the `ID` column in the `CATALOG` table as specified in the `catalog.hbm.xml` file. Add the `journal`, `publisher`, `edition`, `title`, and `author` properties of the `String` type. Add the no-argument constructor and a constructor with all properties as parameters. Add getter/setter methods for the properties.

The `Catalog` persistence class is listed in the following code:

```
package org.jboss.hibernate.model;
import java.io.Serializable;

public class Catalog implements Serializable {
  /** identifier field */
  private Integer id;
  /** nullable persistent field */
  private String journal;
  /** nullable persistent field */
  private String publisher;
  /** nullable persistent field */
  private String edition;
  /** nullable persistent field */
  private String title;
  /** nullable persistent field */
  private String author;

  /** full constructor */
  public Catalog(String journal, String publisher, String edition,
  String title, String author) {
    this.journal = journal;
    this.publisher = publisher;
    this.edition = edition;
    this.title = title;
    this.author = author;
  }

  /** default constructor */
  public Catalog() {
  }

  public Integer getId() {
    return this.id;
  }

  public void setId(Integer id) {
    this.id = id;
  }

  public String getJournal() {
    return this.journal;
  }
```

```
    public void setJournal(String journal) {
      this.journal = journal;
    }

    public String getPublisher() {
      return this.publisher;
    }

    public void setPublisher(String publisher) {
      this.publisher = publisher;
    }

    public String getEdition() {
      return this.edition;
    }

    public void setEdition(String edition) {
      this.edition = edition;
    }

    public String getTitle() {
      return this.title;
    }

    public void setTitle(String title) {
      this.title = title;
    }

    public String getAuthor() {
      return this.author;
    }

    public void setAuthor(String author) {
      this.author = author;
    }
  }
```

The `org.jboss.hibernate.model.Catalog` class is shown in the `jboss-hibernate` project in the following screenshot:



```
package org.jboss.hibernate.model;

import java.io.Serializable;

public class Catalog implements Serializable {
    /** identifier field */
    private Integer id;
    /** nullable persistent field */
    private String journal;
    /** nullable persistent field */
    private String publisher;
    /** nullable persistent field */
    private String edition;
    /** nullable persistent field */
    private String title;
    /** nullable persistent field */
    private String author;

    /** full constructor */
    public Catalog(String journal, String publisher, String edition,
            String title, String author) {
        this.journal = journal;
        this.publisher = publisher;
        this.edition = edition;
        this.title = title;
        this.author = author;
    }
```

Exporting schema

In this section, we export the schema in `schemaExport.jsp JSP`. We will run `schemaExport.jsp` in a later section. Import the `org.hibernate.cfg.Configuration` and `org.hibernate.tool.hbm2ddl.SchemaExport` Hibernate classes. An `org.hibernate.cfg.Configuration` object is an initialization-time-only object to configure properties and mapping files. Create an instance of the `Configuration` class with the no-argument constructor and configure the `hibernate.cfg.xml` Hibernate XML configuration file using the `configure` method in the manner shown in the following code:

```
Configuration cfg=new Configuration();
cfg.configure("hibernate.cfg.xml");
```

The `org.hibernate.tool.hbm2ddl.SchemaExport` class is a command-line tool to export a table schema to a database and can also be invoked from an application. Create an instance of `SchemaExport` using the constructor that takes a `Configuration` object as an argument. Specify the `Configuration` object we created using the `hibernate.cfg.xml` file. The following is the line of code to accomplish this:

```
SchemaExport schemaExport =new  SchemaExport(cfg);
```

Set the output file for the **DDL** script used to create the database table. Use the following line of code to accomplish this:

```
schemaExport.setOutputFile("hbd2ddl.sql");
```

The output file gets generated in the `bin` directory of the WildFly installation. Export the schema to the database using the `create(boolean script,boolean export)` method. The `script` parameter specifies whether the DDL script used to create the database table is to be output to the console. The `export` parameter specifies whether the schema is to be exported. The `create` method can be run with export set to `false` to test the DDL script. Here's the code that encapsulates the discussion in this paragraph:

```
schemaExport.create(true, true);
```

Optionally, add an `out` statement to output a message that the schema has been exported. The `schemaExport.jsp` file is listed in the following code:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<%@ page import="org.hibernate.*,org.hibernate.cfg.Configuration,org.
hibernate.tool.hbm2ddl.SchemaExport"%>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/xml;
charset=windows-1252" />
    <title>Export Schema</title>
  </head>
  <body>
    <%
      Configuration cfg=new Configuration();
      cfg.configure("hibernate.cfg.xml");
      SchemaExport schemaExport =new  SchemaExport(cfg);
      schemaExport.setOutputFile("hbd2ddl.sql");
      schemaExport.create(true, true);
      out.println("Schema Exported");
  %>
    </body>
</html>
```

Creating table data

Having exported the schema to the database, in `add.jsp`, persist the `Catalog` POJO domain model to the database. We will run `schemaExport.jsp` and the other JSPs in a later section after discussing the JSPs:

1. Import the classes in the `org.jboss.hibernate.model` and `org.hibernate` packages and the `org.hibernate.cfg.Configuration` class. Create an instance of the `Configuration` object and configure the `hibernate.cfg.xml` file as in the `schemaExport.jsp`.
2. Create instances of the `Catalog` class using either the no-argument constructor with the setter methods for the properties or the argument constructor that takes all properties in the manner shown in the following code:

   ```
   Catalog catalog = new Catalog();
   catalog.setId(1);
   catalog.setJournal("Oracle Magazine");
   catalog.setPublisher("Oracle Publishing");
   catalog.setEdition("Jan-Feb 2004");
   catalog.setTitle("Understanding Optimization");
   catalog.setAuthor("Kimberly Floss");
   ```

3. Create `SessionFactory` from the `Configuration` object using the `buildSessionFactory()` method. `SessionFactory` has all the metadata from the mapping and properties files in `Configuration`. The `Configuration` object is not used after `SessionFactory` has been created. Create a `Session` object from the `SessionFactory` object using the `openSession()` method. The `openSession` method implements JDBC transparently. JDBC connections are obtained from `ConnectionProvider` internally by Hibernate. We made the `Catalog` persistent class serializable because a `Session` object is serializable only if the persistent class is serializable. A `Session` object is a client interface to Hibernate. The actual persistence to the database is made using a `Transaction` object. Refer to the following line of code, which puts into action the discussion in this paragraph:

   ```
   Session sess = sessionFactory.openSession();
   ```

4. Begin a client session using the `beginTransaction()` method that returns a `Transaction` object. A `Transaction` object represents a global transaction. Refer to the following line of code that summarizes the discussion in this step:

   ```
   Transaction tx = sess.beginTransaction();
   ```

5. The `beginTransaction()` method starts a new underlying transaction only if required; otherwise it uses an existing transaction. Make the `Catalog` instances associate with the `Session` object using the `save()` method, as follows:

   ```
   sess.save(catalog);
   sess.save(catalog2);
   ```

6. The `save()` method does not store the `Catalog` instances to the database but only adds the POJOs to `Session`. To store the `Catalog` instances, invoke the `commit()` method of the `Transaction` object in the manner shown in the following code:

   ```
   tx.commit();
   ```

7. Optionally output a message to indicate that the data has been added to the database. The `add.jsp` file is listed in the following code:

   ```
   <%@ page language="java" contentType="text/html;
   charset=ISO-8859-1" pageEncoding="ISO-8859-1"%><!DOCTYPE HTML
   PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">
   <%@ page import="org.jboss.hibernate.model.*,org.hibernate.*,java.
   util.List,org.hibernate.cfg.Configuration"%>
   <html>
     <head>
       <meta http-equiv="Content-Type" content="text/xml;
   charset=windows-1252" />
       <title>Export Schema</title>
     </head>
     <body>
       <%
         Configuration cfg = new Configuration();
         cfg.configure("hibernate.cfg.xml");
         Catalog catalog = new Catalog();
         catalog.setId(1);
         catalog.setJournal("Oracle Magazine");
         catalog.setPublisher("Oracle Publishing");
         catalog.setEdition("Jan-Feb 2004");
         catalog.setTitle("Understanding Optimization");
         catalog.setAuthor("Kimberly Floss");

         Catalog catalog2 = new Catalog();
         catalog2.setId(2);
         catalog2.setJournal("Oracle Magazine");
         catalog2.setPublisher("Oracle Publishing");
         catalog2.setEdition("March-April 2005");
   ```

```
            catalog2.setTitle("Starting with Oracle ADF");
            catalog2.setAuthor("Steve Muench");
            SessionFactory sessionFactory = cfg.buildSessionFactory();
            Session sess = sessionFactory.openSession();
            Transaction tx = sess.beginTransaction();
            sess.save(catalog);
            sess.save(catalog2);
            tx.commit();
            out.println("Added");
          %>
        </body>
      </html>
```

Retrieving table data

In this section, we query the database to find all instances of a persistent object. Hibernate provides HQL, a query language, which has syntax similar to SQL but is object oriented. A reference to all HQL commands is available at `https://docs.jboss.org/hibernate/orm/3.3/reference/en/html/queryhql.html`. The query is made on the persistent object, which is the `Catalog` class instance, and not on the `CATALOG` database table. To query the database to find all instances of the persistent object, perform the following steps:

1. Create a `String` HQL query to get all instances of the `Catalog` class:

   ```
   String hqlQuery = "from Catalog";
   ```

2. Create and configure a `Configuration` object, create a `SessionFactory` object, and obtain a `Session` object as discussed for `add.jsp`
3. Create a `Query` object from the string HQL query using the `createQuery(String)` method of the `Session` object, as follows:

   ```
   Query query = sess.createQuery(hqlQuery);
   ```

4. A `Query` object is an object-oriented representation of a Hibernate query. Obtain the result of the Hibernate query using the `list()` method, which returns `List`. The SQL used to query the database is implemented internally by Hibernate. A `Transaction` object is not required for a Hibernate query. A `Transaction` object is required only to add, update, or delete a table row.
5. Iterate over `List` to output the query result. The `find.jsp` file is listed in the following code:

   ```
   <%@ page language="java" contentType="text/html;
   charset=ISO-8859-1" pageEncoding="ISO-8859-1"%><!DOCTYPE HTML
   PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   ```

```
"http://www.w3.org/TR/html4/loose.dtd">
<%@ page import="org.jboss.hibernate.model.*,org.hibernate.*,java.
util.List,org.hibernate.cfg.Configuration"%>
<html>
  <head>
    <meta name="generator" content="HTML Tidy for Linux/x86 (vers
25 March 2009), see www.w3.org" />
    <meta http-equiv="Content-Type" content="text/xml; charset=us-
ascii" />
    <title>
      Export Schema
    </title>
  </head>
  <body>
    <%
      String hqlQuery = "from Catalog";
      Configuration cfg = new Configuration();
      cfg.configure("hibernate.cfg.xml");
      SessionFactory sessionFactory = cfg.buildSessionFactory();
      Session sess = sessionFactory.openSession();
      Query query = sess.createQuery(hqlQuery);
      List list = query.list();
      for (int i = 0; i < list.size(); i++) {
        Catalog catalog = (Catalog) list.get(i);
        out.println("<br>");
        out.println("CatalogId " + catalog.getId() + " Journal: "+
catalog.getJournal());
        out.println("<br>");
        out.println("CatalogId " + catalog.getId() + " Publisher:
" + catalog.getPublisher());
        out.println("<br>");
        out.println("CatalogId " + catalog.getId() + " Edition: "+
catalog.getEdition());
        out.println("<br>");
        out.println("CatalogId " + catalog.getId() + " Title "+
catalog.getTitle());
        out.println("<br>");
         out.println("CatalogId " + catalog.getId() + " Author: "+
catalog.getAuthor());
      }
      sess.close();
    %>
  </body>
</html>
```

Updating a table row

In this section, we will update a table row. Perform the following steps to accomplish this:

1. Create an HQL query `String` and create a `Query` object to generate `List` as discussed for `find.jsp`.
2. Obtain the first item in `List` and modify the `publisher` value with the `setPublisher()` method, as follows:

   ```
   Catalog catalog = (Catalog) list.get(0);
   catalog.setPublisher("Oracle Magazine");
   ```

3. Create a `Transaction` object, which represents a transaction with the database, with the `beginTransaction()` method. Save or update the persistent state of the `Catalog` object in `Session` with the `saveOrUpdate` method. Invoke the `commit()` method of the `Transaction` object to save the `Catalog` instance in the database.
4. Optionally, output a message to indicate that the update was completed:

   ```
   Transaction tx = sess.beginTransaction();
   sess.saveOrUpdate(catalog);
   tx.commit();
   ```

   The `update.jsp` is listed in the following code:

   ```
   <%@ page language="java" contentType="text/html;
   charset=ISO-8859-1"        pageEncoding="ISO-8859-1"%><!DOCTYPE
   HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">
   <%@ page import="org.jboss.hibernate.model.*,org.hibernate.*,java.
   util.List,org.hibernate.cfg.Configuration"%>
   <html>
     <head>
       <meta http-equiv="Content-Type" content="text/xml;
   charset=windows-1252" />
       <title>Export Schema</title>
     </head>
     <body>
       <%
         String hqlQuery = "from Catalog";
         Configuration cfg = new Configuration();
   ```

```
            cfg.configure("hibernate.cfg.xml");
            SessionFactory sessionFactory = cfg.buildSessionFactory();
            Session sess = sessionFactory.openSession();
            Query query = sess.createQuery(hqlQuery);
            List list = query.list();
            Catalog catalog = (Catalog) list.get(0);
            catalog.setPublisher("Oracle Magazine");
            Transaction tx = sess.beginTransaction();
            sess.saveOrUpdate(catalog);
            tx.commit();
            out.println("Updated");
          %>
        </body>
      </html>
```

Deleting a table row

In this section, we will delete a table row from the `CATALOG` table. Perform the following steps to accomplish this:

1. Create a HQL query `String` for the `Catalog` instance to delete the table row using the following line of code:

   ```
   String hqlQuery = "from Catalog as catalog WHERE catalog.
   edition='March-April 2005'";
   ```

2. As in `find.jsp` and `update.jsp`, get `List` for `Catalog` instances. As only one `Catalog` instance has `edition` set to `March-April 2005`, we only need to get the first `Catalog` instance from `List`. To do so, use the following code:

   ```
   Catalog catalog = (Catalog) list.get(0);
   ```

3. Create a `Transaction` object with `beginTransaction()`.
4. Delete the `Catalog` instance from the `Session` with the `delete` method, which doesn't delete the `Catalog` instance from the database. Invoke the `commit()` method of the `Transaction` object to save the `Session` state in the database, which deletes the corresponding table row from the `CATALOG` table.

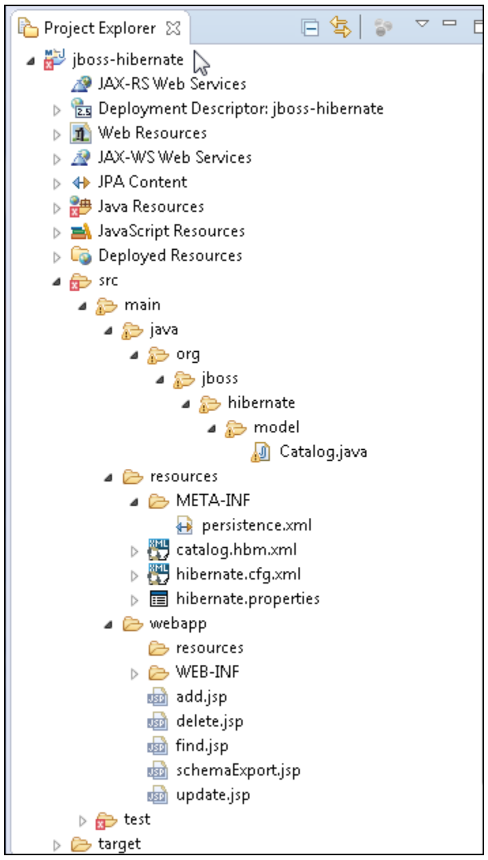
5. Optionally, using the following code, output a message to indicate deletion:

```
sess.delete(catalog);
tx.commit();
```

The `delete.jsp` file is listed in the following code:

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%><!DOCTYPE HTML
PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<%@ page import="org.jboss.hibernate.model.*,org.hibernate.*,java.
util.List,org.hibernate.cfg.Configuration"%>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/xml;
charset=windows-1252" />
    <title>Export Schema</title>
  </head>
  <body>
    <%
     String hqlQuery = "from Catalog as catalog WHERE catalog.
edition='March-April 2005'";
      Configuration cfg = new Configuration();
     cfg.configure("hibernate.cfg.xml");
     SessionFactory sessionFactory = cfg.buildSessionFactory();
     Session sess = sessionFactory.openSession();
     Query query = sess.createQuery(hqlQuery);
     List list = query.list();
     Catalog catalog = (Catalog) list.get(0);
     Transaction tx = sess.beginTransaction();
     sess.delete(catalog);
     tx.commit();
      out.println("Deleted");
    %>
    </body>
</html>
```

The directory structure of the Maven project is shown in the following screenshot:



```
Project Explorer
jboss-hibernate
  JAX-RS Web Services
  Deployment Descriptor: jboss-hibernate
  Web Resources
  JAX-WS Web Services
  JPA Content
  Java Resources
  JavaScript Resources
  Deployed Resources
  src
    main
      java
        org
          jboss
            hibernate
              model
                Catalog.java
      resources
        META-INF
          persistence.xml
        catalog.hbm.xml
        hibernate.cfg.xml
        hibernate.properties
      webapp
        resources
        WEB-INF
        add.jsp
        delete.jsp
        find.jsp
        schemaExport.jsp
        update.jsp
    test
  target
```

Installing the Maven project

In this section, we will compile and package the Hibernate web application using the Maven build tool. Some APIs, such as the Common Annotations API, Hibernate validator API, and CDI API, are provided by WildFly 8. We need to add the MySQL JDBC connector dependency to `pom.xml` inside the `<dependencies/>` element. To acomplish this, use the following code:

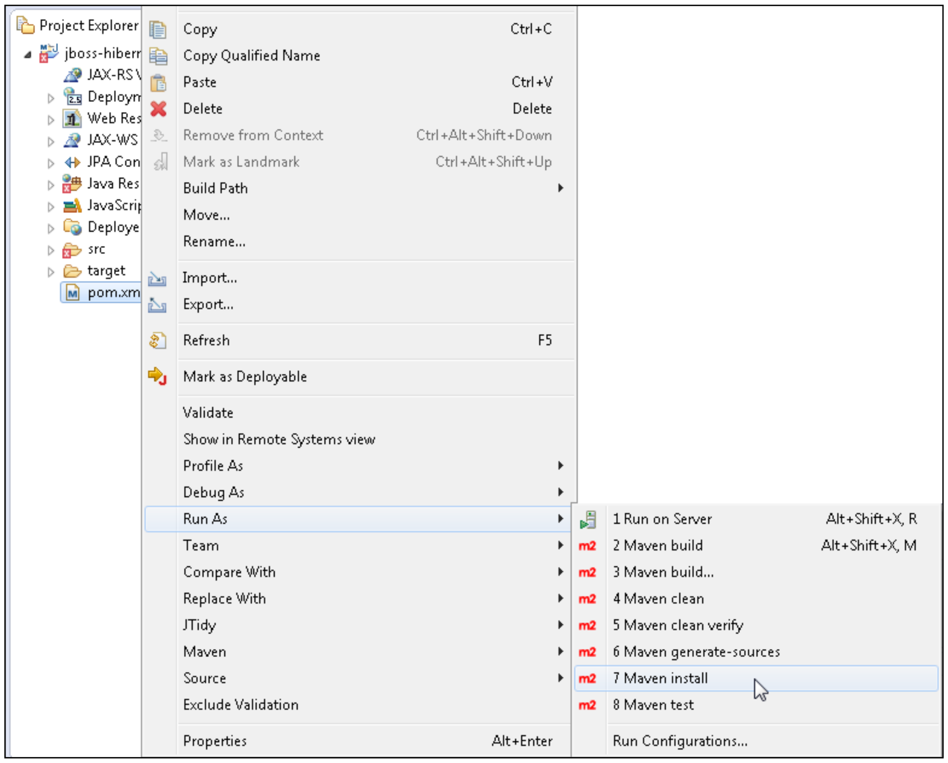
```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>5.1.28</version>
</dependency>
```

Hibernate provides Hibernate-related artifacts in the group ID `org.hibernate`. Include the Hibernate artifacts listed in the following table:

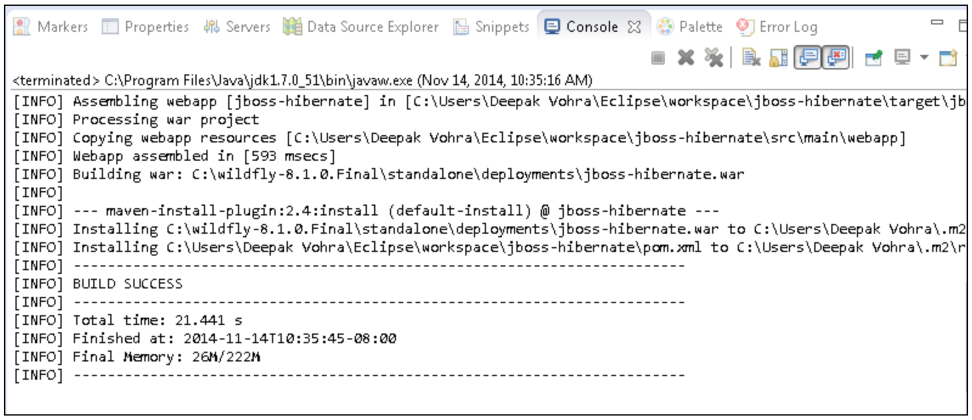
Artifact	**Description**
`hibernate-core`	This is the main artifact for the Hibernate API.
`hibernate-annotations`	This is for the annotations metadata.
`hibernate-commons-annotations`	This pertains to the EJB 3 style annotations for Hibernate.
`hibernate-ehcache`	This provides the cache for second-level cache.
`hibernate-c3p0`	This is the C3P0 connection-pooling library.
`antlr`	This is the parser generator.
`hibernate-cglib-repack`	This is the CGLIB code generation library and also signifies ASM dependencies.
`hibernate-tools`	This provides tools to generate various Hibernate source artifacts, such as mapping files and Java entities.
`hibernate-envers`	This is used for auditing and versioning of persistent classes.
`hibernate-jpamodelgen`	This is an annotation processor to generate JPA 2 static meta-model classes. The Catalog entity in *Chapter 1, Getting Started with EJB 3.x*, is an example of a JPA 2 meta-model class.

The Maven compiler plugin is used to compile the project sources, and the Maven WAR plugin collects all the dependencies, classes, and resources and generates a WAR archive. In the configuration for `maven-war-plugin`, specify the directory to output the `WAR` file with the `outputDirectory` element as `C:\JBossAS8\wildfly-8.0.0.CR1\standalone\deployments`. The `EAR`, `WAR`, and `JAR` files in the `deployments` directory get deployed to the WildFly application server. The `pom.xml` file for the `jboss-hibernate` project is available in the code download for this chapter.

Next, we install the Maven project. Right-click on `pom.xml` and select **Run As** | **Maven install**, as shown in the following screenshot:

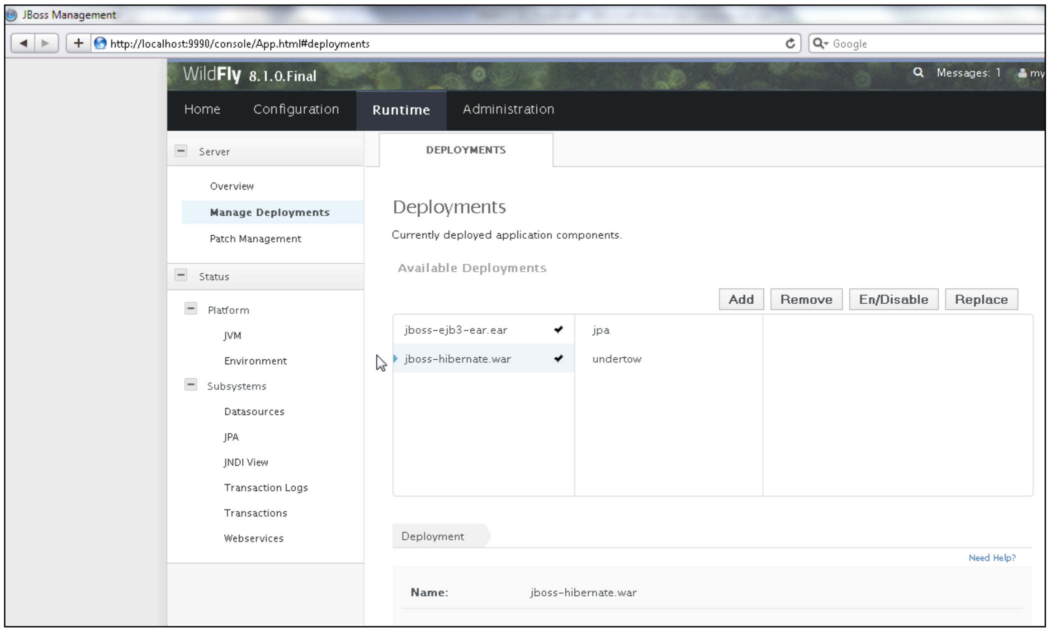


The Maven project is compiled and the `jboss-hibernate.war` archive gets generated and output to the `deployments` directory of WildFly 8, as shown in the following screenshot:



```
<terminated> C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Nov 14, 2014, 10:35:16 AM)
[INFO] Assembling webapp [jboss-hibernate] in [C:\Users\Deepak Vohra\Eclipse\workspace\jboss-hibernate\target\jb
[INFO] Processing war project
[INFO] Copying webapp resources [C:\Users\Deepak Vohra\Eclipse\workspace\jboss-hibernate\src\main\webapp]
[INFO] Webapp assembled in [593 msecs]
[INFO] Building war: C:\wildfly-8.1.0.Final\standalone\deployments\jboss-hibernate.war
[INFO]
[INFO] --- maven-install-plugin:2.4:install (default-install) @ jboss-hibernate ---
[INFO] Installing C:\wildfly-8.1.0.Final\standalone\deployments\jboss-hibernate.war to C:\Users\Deepak Vohra\.m2
[INFO] Installing C:\Users\Deepak Vohra\Eclipse\workspace\jboss-hibernate\pom.xml to C:\Users\Deepak Vohra\.m2\r
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 21.441 s
[INFO] Finished at: 2014-11-14T10:35:45-08:00
[INFO] Final Memory: 26M/222M
[INFO] ------------------------------------------------------------------------
```

Start the WildFly 8 server. The `jboss-hibernate.war` file gets deployed to the server and the MySQL data source also gets deployed. The persistence unit gets started. The `jboss-hibernate.war` is shown deployed in the WildFly **Administration Console** in the following screenshot:



Deployments

Currently deployed application components.

Available Deployments

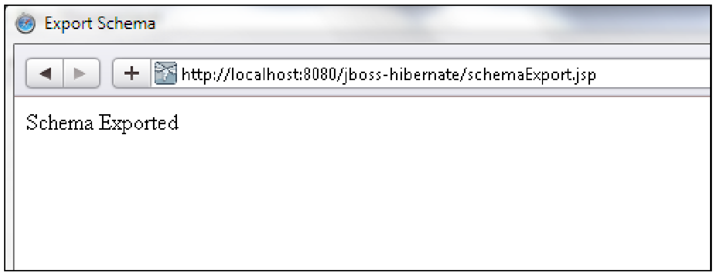
jboss-ejb3-ear.ear ✓	jpa
jboss-hibernate.war ✓	undertow

Name: jboss-hibernate.war

Running a schema export

In this section, we will run `schemaExport.jsp` on the WildFly application server to export the schema to the MySQL database. To accomplish this, perform the following steps:

1. Invoke the URL `http://localhost:8080/jboss-hibernate/schemaExport.jsp` in a browser, as shown in the following screenshot. The schema gets exported to the MySQL database.

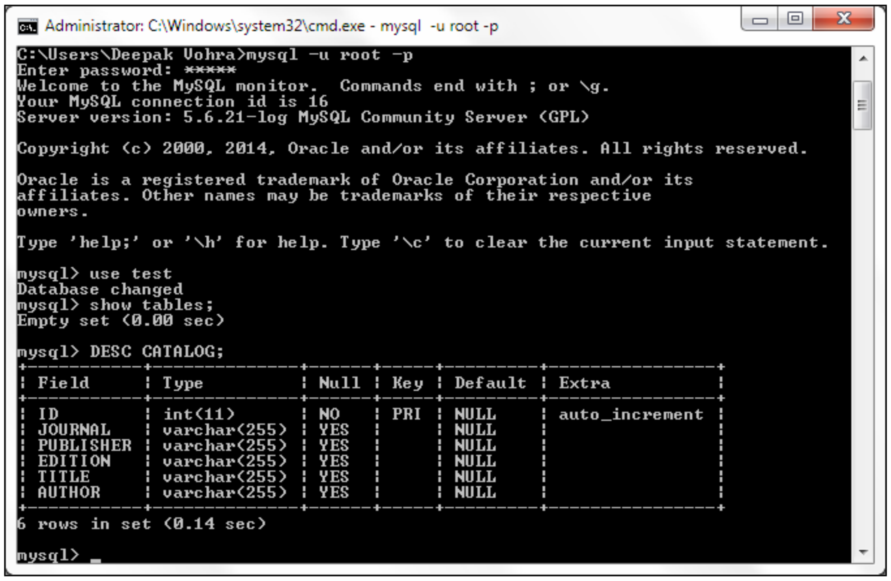


Schema Exported

2. The output from `schemaExport.jsp` on the server is shown in the following code:

```
10:57:23,589 INFO  [org.hibernate.cfg.Configuration] (default
task-8) HHH000041:
 Configured SessionFactory: HibernateSessionFactory
10:57:23,590 INFO  [org.hibernate.dialect.Dialect] (default task-
8) HHH000400: U
sing dialect: org.hibernate.dialect.MySQL5InnoDBDialect
10:57:23,592 INFO  [org.hibernate.tool.hbm2ddl.SchemaExport]
(default task-8) HH
H000227: Running hbm2ddl schema export
10:57:23,761 WARN  [org.hibernate.engine.jdbc.connections.
internal.DriverManager
ConnectionProviderImpl] (default task-8) HHH000402: Using
Hibernate built-in con
nection pool (not for production use!)
10:57:23,841 INFO  [org.hibernate.engine.jdbc.connections.
internal.DriverManager
ConnectionProviderImpl] (default task-8) HHH000401: using driver
[com.mysql.jdbc
.Driver] at URL [jdbc:mysql://localhost:3306/test]
10:57:23,842 INFO  [org.hibernate.engine.jdbc.connections.
internal.DriverManager
ConnectionProviderImpl] (default task-8) HHH000046: Connection
properties: {user
=root, password=****}
10:57:23,843 INFO  [org.hibernate.engine.jdbc.connections.
internal.DriverManager
ConnectionProviderImpl] (default task-8) HHH000006: Autocommit
mode: false
10:57:23,843 INFO  [org.hibernate.engine.jdbc.connections.
internal.DriverManager
ConnectionProviderImpl] (default task-8) HHH000115: Hibernate
connection pool si
ze: 20 (min=1)
10:57:24,035 INFO  [stdout] (default task-8)
10:57:24,036 INFO  [stdout] (default task-8)     drop table if
exists CATALOG
10:57:24,038 INFO  [stdout] (default task-8)
10:57:24,038 INFO  [stdout] (default task-8)     create table
CATALOG (
10:57:24,038 INFO  [stdout] (default task-8)         ID integer
not null auto_in
crement,
10:57:24,039 INFO  [stdout] (default task-8)         JOURNAL
varchar(255),
10:57:24,039 INFO  [stdout] (default task-8)          PUBLISHER
varchar(255),
10:57:24,039 INFO  [stdout] (default task-8)          EDITION
varchar(255),
10:57:24,039 INFO  [stdout] (default task-8)          TITLE
varchar(255),
10:57:24,039 INFO  [stdout] (default task-8)          AUTHOR
varchar(255),
10:57:24,040 INFO  [stdout] (default task-8)          primary key
(ID)
10:57:24,040 INFO  [stdout] (default task-8)      ) ENGINE=InnoDB
10:57:27,224 INFO  [org.hibernate.engine.jdbc.connections.
internal.DriverManager
ConnectionProviderImpl] (default task-8) HHH000030: Cleaning up
connection pool
[jdbc:mysql://localhost:3306/test]
10:57:27,225 INFO  [org.hibernate.tool.hbm2ddl.SchemaExport]
(default task-8) HHH000230: Schema export complete
```

3. Run the `DESC CATALOG` command in the MySQL command line for the structure of the `CATALOG` table, as shown in the following screenshot:

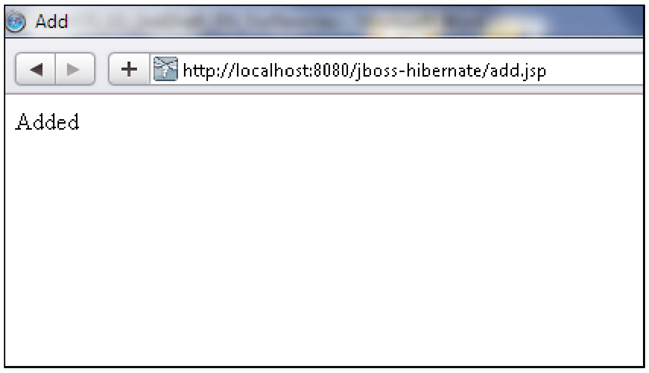


Administrator: C:\Windows\system32\cmd.exe - mysql -u root -p

```
C:\Users\Deepak Vohra>mysql -u root -p
Enter password: *****
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 16
Server version: 5.6.21-log MySQL Community Server (GPL)

Copyright (c) 2000, 2014, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> use test
Database changed
mysql> show tables;
Empty set (0.00 sec)

mysql> DESC CATALOG;
+-----------+--------------+------+-----+---------+----------------+
| Field     | Type         | Null | Key | Default | Extra          |
+-----------+--------------+------+-----+---------+----------------+
| ID        | int(11)      | NO   | PRI | NULL    | auto_increment |
| JOURNAL   | varchar(255) | YES  |     | NULL    |                |
| PUBLISHER | varchar(255) | YES  |     | NULL    |                |
| EDITION   | varchar(255) | YES  |     | NULL    |                |
| TITLE     | varchar(255) | YES  |     | NULL    |                |
| AUTHOR    | varchar(255) | YES  |     | NULL    |                |
+-----------+--------------+------+-----+---------+----------------+
6 rows in set (0.14 sec)

mysql> _
```

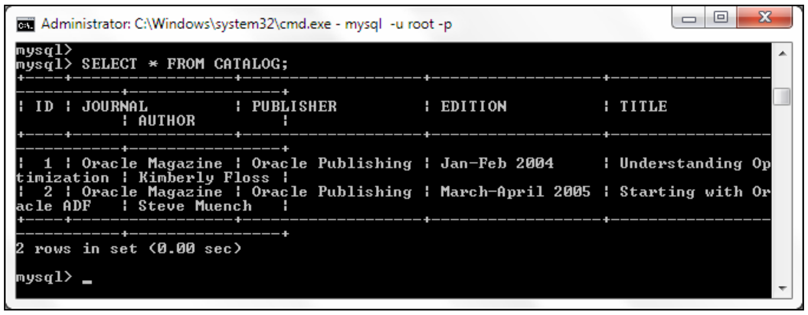
Creating table rows

In this section, we will add the table data to the `CATALOG` table. Invoke the URL `http://localhost:8080/jboss-hibernate/add.jsp`, as shown in the following screenshot. The table data gets added.



```
Added
```

Data gets added to the `CATALOG` table. A `SELECT` query in the MySQL command line lists the `CATALOG` table in the manner shown in the following screenshot:

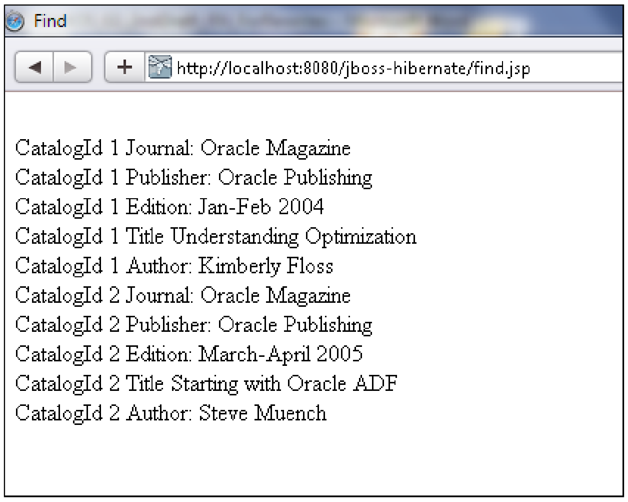


Administrator: C:\Windows\system32\cmd.exe - mysql -u root -p

```
mysql>
mysql> SELECT * FROM CATALOG;
+----+-----------------+-------------------+--------------------+----------------
-----------+-----------------+
| ID | JOURNAL         | PUBLISHER         | EDITION            | TITLE
           | AUTHOR          |
+----+-----------------+-------------------+--------------------+----------------
-----------+-----------------+
|  1 | Oracle Magazine | Oracle Publishing | Jan-Feb 2004       | Understanding Op
timization | Kimberly Floss |
|  2 | Oracle Magazine | Oracle Publishing | March-April 2005   | Starting with Or
acle ADF   | Steve Muench    |
+----+-----------------+-------------------+--------------------+----------------
-----------+-----------------+
2 rows in set (0.00 sec)

mysql> _
```

Retrieving table data

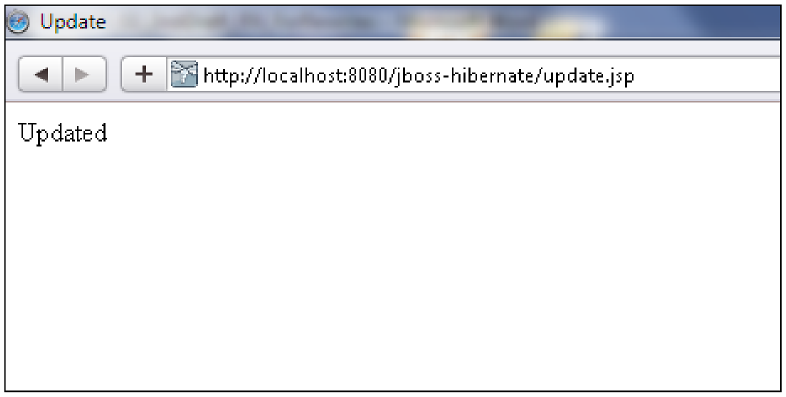
In this section, we will run `find.jsp` to get and display the `CATALOG` table data. Invoke the URL `http://localhost:8080/jboss-hibernate/find.jsp`, as shown in the following screenshot. The `CATALOG` table data gets output in the browser.



```
CatalogId 1 Journal: Oracle Magazine
CatalogId 1 Publisher: Oracle Publishing
CatalogId 1 Edition: Jan-Feb 2004
CatalogId 1 Title Understanding Optimization
CatalogId 1 Author: Kimberly Floss
CatalogId 2 Journal: Oracle Magazine
CatalogId 2 Publisher: Oracle Publishing
CatalogId 2 Edition: March-April 2005
CatalogId 2 Title Starting with Oracle ADF
CatalogId 2 Author: Steve Muench
```

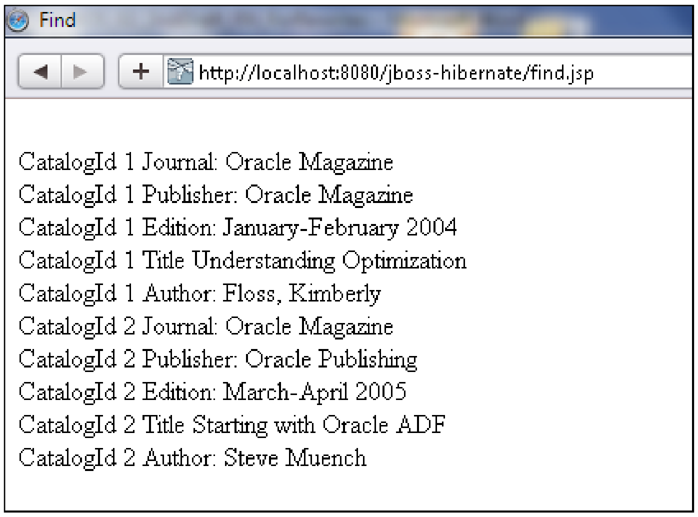
Updating the table

In this section, we will update the `CATALOG` table. Invoke the URL `http://localhost:8080/jboss-hibernate/update.jsp`, as shown in the following screenshot. The `CATALOG` table gets updated.



```
Updated
```

Run `find.jsp` to get and display the updated `CATALOG` table data. Invoke the URL `http://localhost:8080/jboss-hibernate/find.jsp`, as shown in the following screenshot:



Find

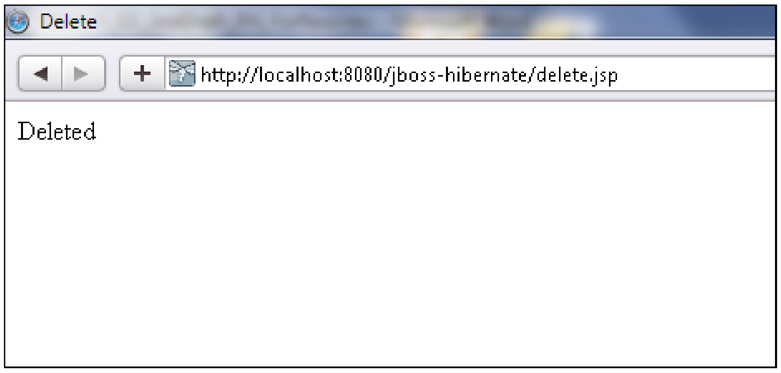
http://localhost:8080/jboss-hibernate/find.jsp

```
CatalogId 1 Journal: Oracle Magazine
CatalogId 1 Publisher: Oracle Magazine
CatalogId 1 Edition: January-February 2004
CatalogId 1 Title Understanding Optimization
CatalogId 1 Author: Floss, Kimberly
CatalogId 2 Journal: Oracle Magazine
CatalogId 2 Publisher: Oracle Publishing
CatalogId 2 Edition: March-April 2005
CatalogId 2 Title Starting with Oracle ADF
CatalogId 2 Author: Steve Muench
```

Deleting the table row

In this section, we will delete a table row with the `delete.jsp` file. To accomplish this, perform the following steps:

1. Invoke the URL `http://localhost:8080/jboss-hibernate/delete.jsp`, as shown in the following screenshot. A table row gets deleted from `CATALOG`.

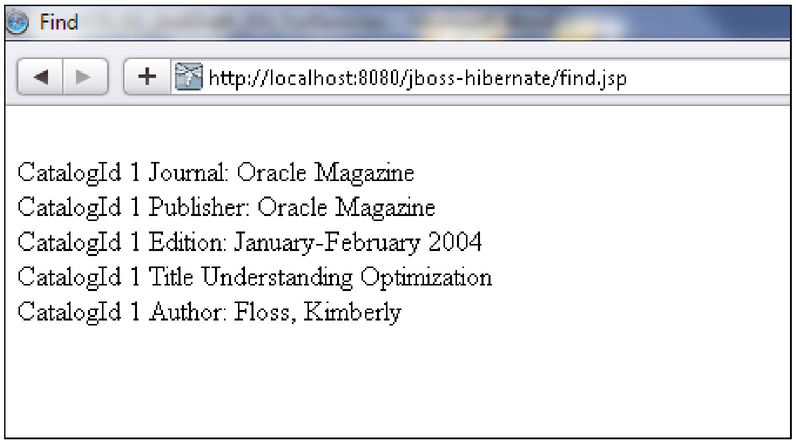


Delete

http://localhost:8080/jboss-hibernate/delete.jsp

```
Deleted
```

2. Run `find.jsp` again to list the updated `CATALOG` table with a row deleted. The output in the browser is shown in the following screenshot:



```
CatalogId 1 Journal: Oracle Magazine
CatalogId 1 Publisher: Oracle Magazine
CatalogId 1 Edition: January-February 2004
CatalogId 1 Title Understanding Optimization
CatalogId 1 Author: Floss, Kimberly
```

Summary

In this chapter, we created a CRUD application with the Hibernate API. We configured Hibernate using `hibernate.cfg.xml`. We mapped the persistence class `Catalog` to a MySQL database table with mapping specified in `catalog.hbm.xml`. We compiled and packaged the Hibernate web application with the Maven build tool. We ran the web application on the WildFly 8 server to export a schema to the MySQL database and created, retrieved, updated, and deleted table data. We used hardcoded `set`, `get`, `update`, and `delete` operations, but a more dynamic CRUD application can be created with user interfaces. In the next chapter, we discuss how to develop **Java Server Faces** (JSF) in WildFly 8.

3
Developing JSF 2.x Facelets

JavaServer Faces (JSF) 2.x has introduced several new features, such as integrated Facelets, implicit navigation, conditional navigation, preemptive navigation, bean validation, view parameters, client behaviors, new scopes (view, flash, and custom), configuration annotations, composite components, Resource Handler API, support for Ajax, and new event handling and exception handling features.

WildFly 8.x supports JSF 2.2, the latest version of JSF. JSF 2.2 has introduced new features of convenient HTML5 markup, Resource Library Contracts, Face Flows, and stateless views. JSF 2.2 support is added to a project with a Maven dependency. In this chapter, we will develop a JSF 2.x Facelets application in Eclipse, build and package the application with Maven, and deploy the application to WildFly 8.1. We will run the application in WildFly 8.1 to demonstrate the use of Facelets in a web application. Facelets is the default **View Declaration Language** (**VDL**) in JSF 2.x, replacing JSP as the default VDL. This chapter has the following sections:

- Setting the environment
- Creating a Java EE web project
- Creating a managed bean
- Creating the Facelets template
- Creating a header and footer
- Creating input and output Facelets pages
- Creating a web descriptor
- Installing the web project with Maven
- Running the Facelets application

Setting up the environment

We need to install the following software:

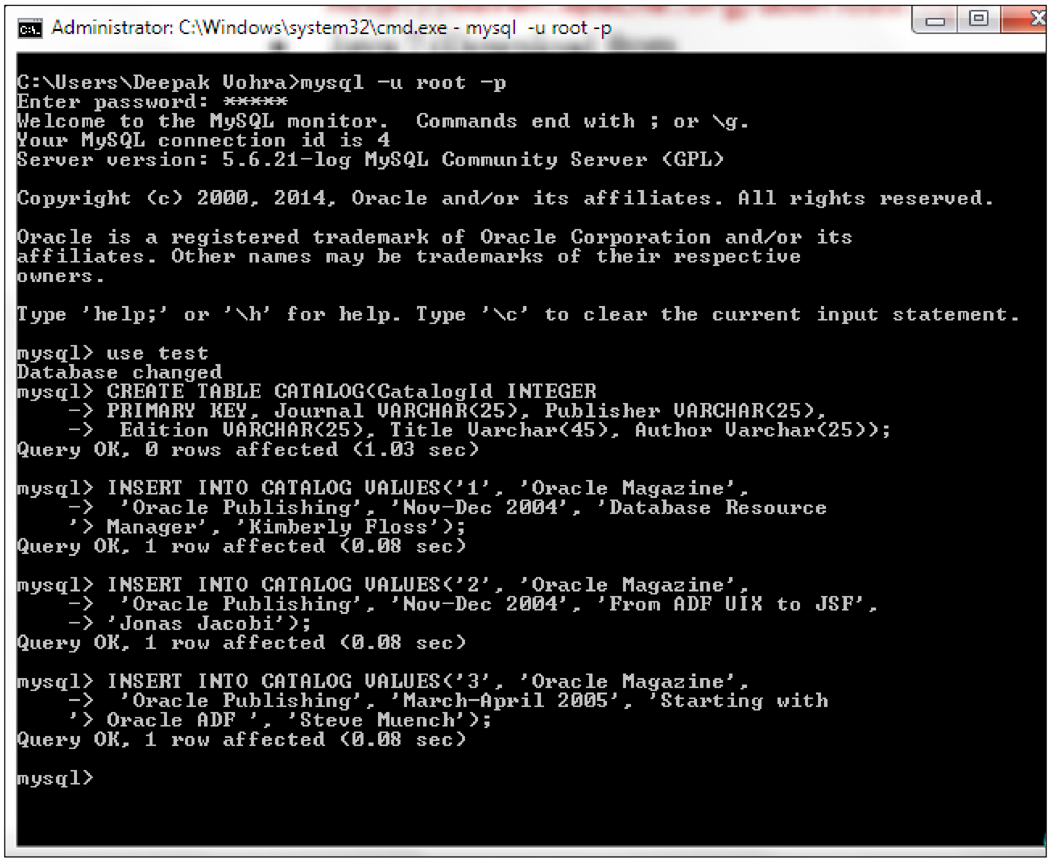
- **WildFly 8.1.0.Final**: Download `wildfly-8.1.0.Final.zip` from `http://wildfly.org/downloads/`
- **MySQL 5.6 Database-Community Edition**: Download this edition from `http://dev.mysql.com/downloads/mysql/`. When installing MySQL, also install **Connector/J**
- **Eclipse IDE for Java EE Developers**: Download Eclipse Luna from `https://www.eclipse.org/downloads/packages/release/Luna/SR1`
- **JBoss Tools (Luna) 4.2.0.Final**: Install this as a plugin to Eclipse from the Eclipse Marketplace (`http://tools.jboss.org/downloads/installation.html`)
- **Apache Maven**: Download version 3.05 or higher from `http://maven.apache.org/download.cgi`
- **Java 7**: Download Java 7 from `http://www.oracle.com/technetwork/java/javase/downloads/index.html?ssSourceSiteId=ocomcn`

Set the environment variables `JAVA_HOME`, `JBOSS_HOME`, `MAVEN_HOME`, and `MYSQL_HOME`. Add `%JAVA_HOME%/bin`, `%MAVEN_HOME%/bin`, `%JBOSS_HOME%/bin`, and `%MYSQL_HOME%/bin` to the `PATH` environment variable.

Create a WildFly 8.1.0 runtime as discussed in *Chapter 1, Getting Started with EJB 3.x*. Create a MySQL database `CATALOG` with the following SQL script:

```
CREATE TABLE CATALOG(CatalogId INTEGER
PRIMARY KEY, Journal VARCHAR(25), Publisher VARCHAR(25), Edition
VARCHAR(25), Title Varchar(45), Author Varchar(25));
INSERT INTO CATALOG VALUES('1', 'Oracle Magazine', 'Oracle
Publishing', 'Nov-Dec 2004', 'Database Resource Manager', 'Kimberly
Floss');
INSERT INTO CATALOG VALUES('2', 'Oracle Magazine', 'Oracle
Publishing', 'Nov-Dec 2004', 'From ADF UIX to JSF', 'Jonas Jacobi');
INSERT INTO CATALOG VALUES('3', 'Oracle Magazine', 'Oracle
Publishing', 'March-April 2005', 'Starting with Oracle ADF ', 'Steve
Muench');
```

Run the script in the MySQL 5.6 command-line client. The database table, `Catalog`, gets created. The output from the preceding script is shown in the following screenshot:

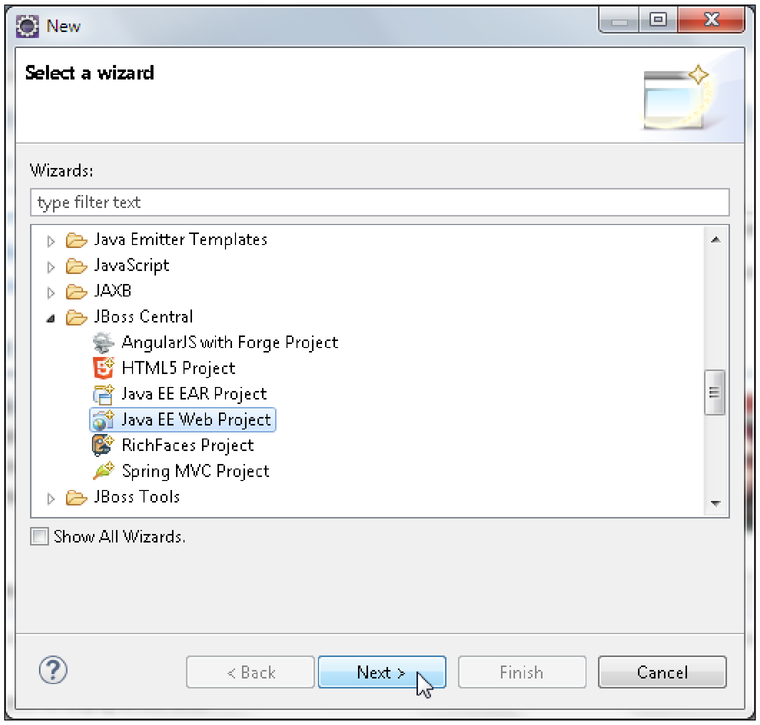

Administrator: C:\Windows\system32\cmd.exe - mysql -u root -p

```
C:\Users\Deepak Vohra>mysql -u root -p
Enter password: *****
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 4
Server version: 5.6.21-log MySQL Community Server (GPL)

Copyright (c) 2000, 2014, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> use test
Database changed
mysql> CREATE TABLE CATALOG(CatalogId INTEGER
    -> PRIMARY KEY, Journal VARCHAR(25), Publisher VARCHAR(25),
    ->  Edition VARCHAR(25), Title Varchar(45), Author Varchar(25));
Query OK, 0 rows affected (1.03 sec)

mysql> INSERT INTO CATALOG VALUES('1', 'Oracle Magazine',
    ->  'Oracle Publishing', 'Nov-Dec 2004', 'Database Resource
    '> Manager', 'Kimberly Floss');
Query OK, 1 row affected (0.08 sec)

mysql> INSERT INTO CATALOG VALUES('2', 'Oracle Magazine',
    ->  'Oracle Publishing', 'Nov-Dec 2004', 'From ADF UIX to JSF',
    -> 'Jonas Jacobi');
Query OK, 1 row affected (0.08 sec)

mysql> INSERT INTO CATALOG VALUES('3', 'Oracle Magazine',
    ->  'Oracle Publishing', 'March-April 2005', 'Starting with
    '> Oracle ADF ', 'Steve Muench');
Query OK, 1 row affected (0.08 sec)

mysql>
```

We also need to configure a data source for MySQL database. The procedure to configure a MySQL data source was discussed in *Chapter 1, Getting Started with EJB 3.x*, and will not be repeated in this chapter.

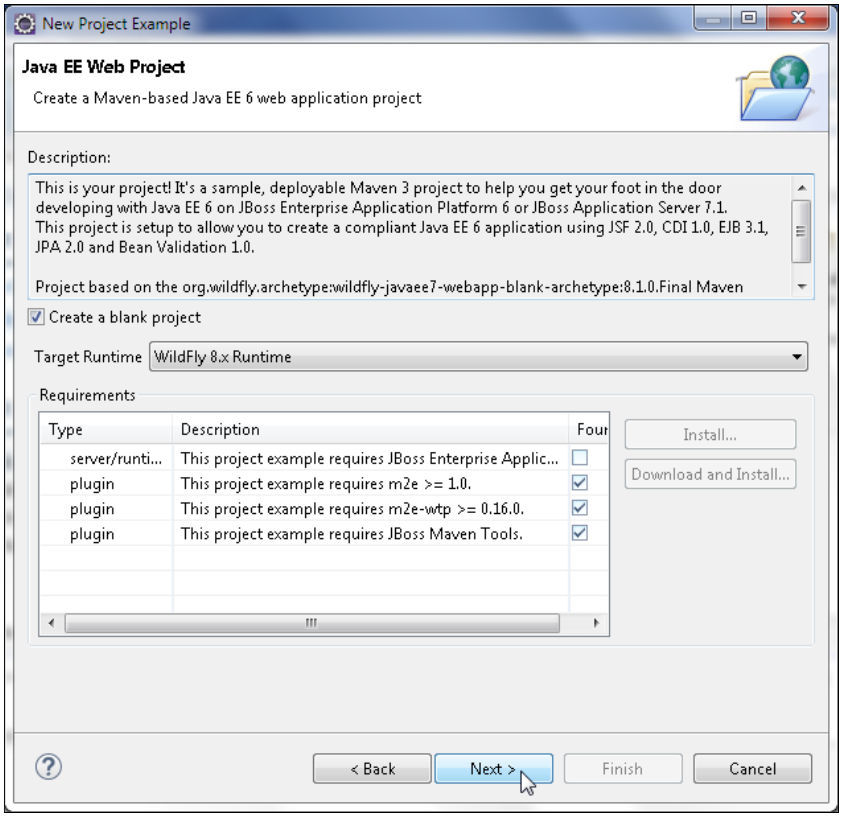
Creating a Java EE web project

In this section, we will create a Eclipse project for JSF 2.x. Select **File** | **New** | **Other** in Eclipse. Select **JBoss Central** | **Java EE Web Project** and click on **Next**, as shown in the following screenshot:



The **Java EE Web Project** wizard gets started. Though the wizard indicates that a Java EE 6 web application project will be created, a Java EE 7 web application project gets created.

A test is run for the requirements, which include JBoss AS runtime, the **m2e** and **m2eclipse-wtp** plugins, and the JBoss Tools plugin, as shown in the following screenshot. Select **Create a blank project** checkbox and select **Target Runtime** as **WildFly 8.x Runtime**. Click on **Next**.



New Project Example

Java EE Web Project

Create a Maven-based Java EE 6 web application project

Description:

This is your project! It's a sample, deployable Maven 3 project to help you get your foot in the door developing with Java EE 6 on JBoss Enterprise Application Platform 6 or JBoss Application Server 7.1.
This project is setup to allow you to create a compliant Java EE 6 application using JSF 2.0, CDI 1.0, EJB 3.1, JPA 2.0 and Bean Validation 1.0.

Project based on the org.wildfly.archetype:wildfly-javaee7-webapp-blank-archetype:8.1.0.Final Maven

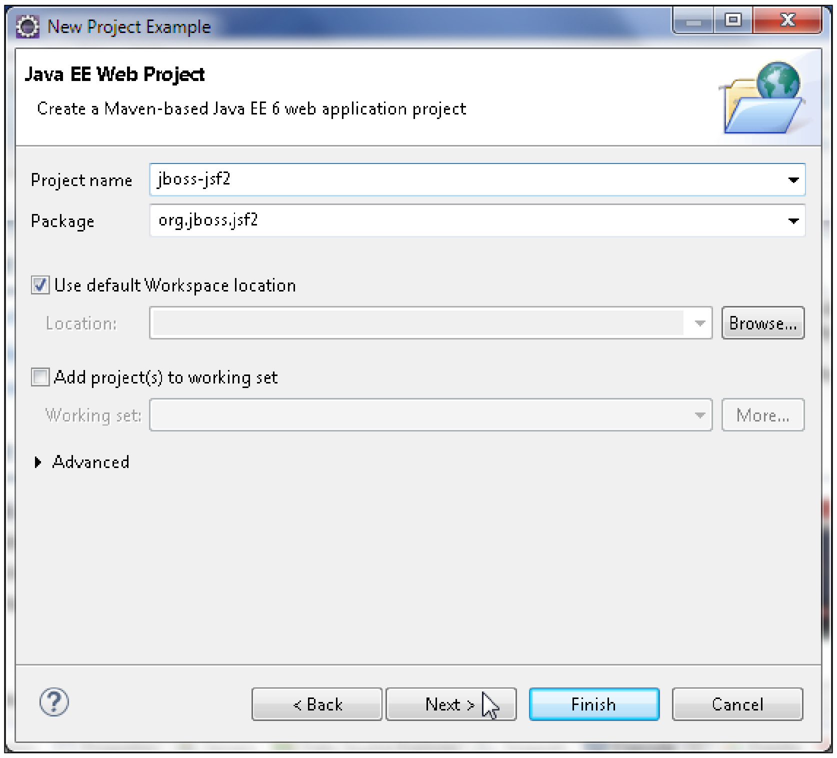
☑ Create a blank project

Target Runtime: WildFly 8.x Runtime

Requirements

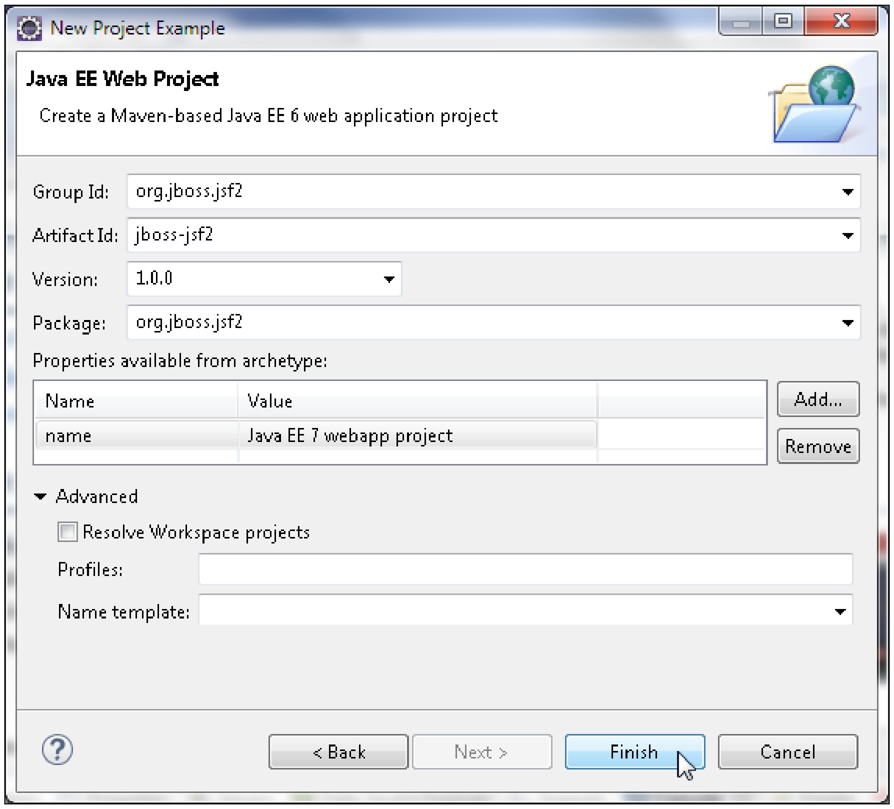
Type	Description	Four
server/runti...	This project example requires JBoss Enterprise Applic...	☐
plugin	This project example requires m2e >= 1.0.	☑
plugin	This project example requires m2e-wtp >= 0.16.0.	☑
plugin	This project example requires JBoss Maven Tools.	☑

Specify **Project name** (`jboss-jsf2`) and **Package** (`org.jboss.jsf2`), and select the checkbox **Use default Workspace location**. Click on **Next** as shown in the following screenshot:



The Maven building tool is used for the example project, and therefore, it is necessary to specify the Maven modules: **Group Id** (`org.jboss.jsf2`), **Artifact Id** (`jboss-jsf2`), **Version** (`1.0.0`), and **Package** (`org.jboss.jsf2`), as shown in the following screenshot. Other than the `name` property, the properties listed in **Properties available from archetype** may be deleted as these are not required for the application.

The default `name` property is set to **Java EE 6 webapp project**, which should be modified to **Java EE 7 webapp project**. Click on **Finish**.



Create a Maven-based Java EE 6 web application project

Group Id: org.jboss.jsf2

Artifact Id: jboss-jsf2

Version: 1.0.0

Package: org.jboss.jsf2

Properties available from archetype:

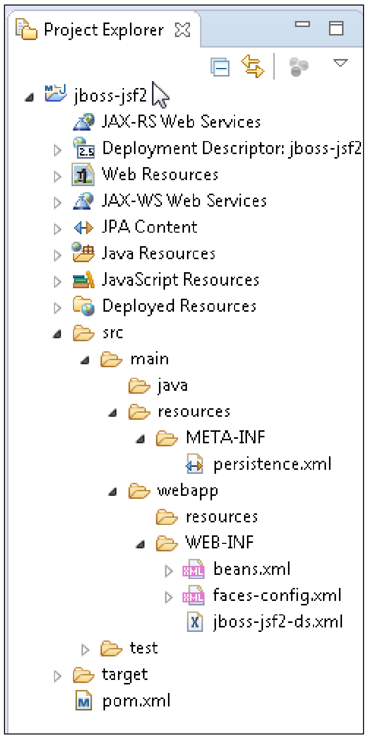
Name	Value
name	Java EE 7 webapp project

[ ] Resolve Workspace projects

Profiles:

Name template:

A **Java EE Web Project** gets created, as shown in **Project Explorer** in the following screenshot. Delete the `//jboss-jsf2/src/main/resources/META-INF/persistence.xml` configuration file as it is not used in the JSF application. The web project includes a `default.xhtml` file in the `WEB-INF/templates` directory. We will need a Facelets template, but not the default template.



Facelets is a set of tags in the `http://java.sun.com/jsf/facelets` namespace. Facelets tags are used in conjunction with JSF Core and JSF HTML tag libraries to develop a JSF Facelets application. The default suffix for a Facelets page is `.xhtml`. A Facelets application consists of the following configuration and templating files:

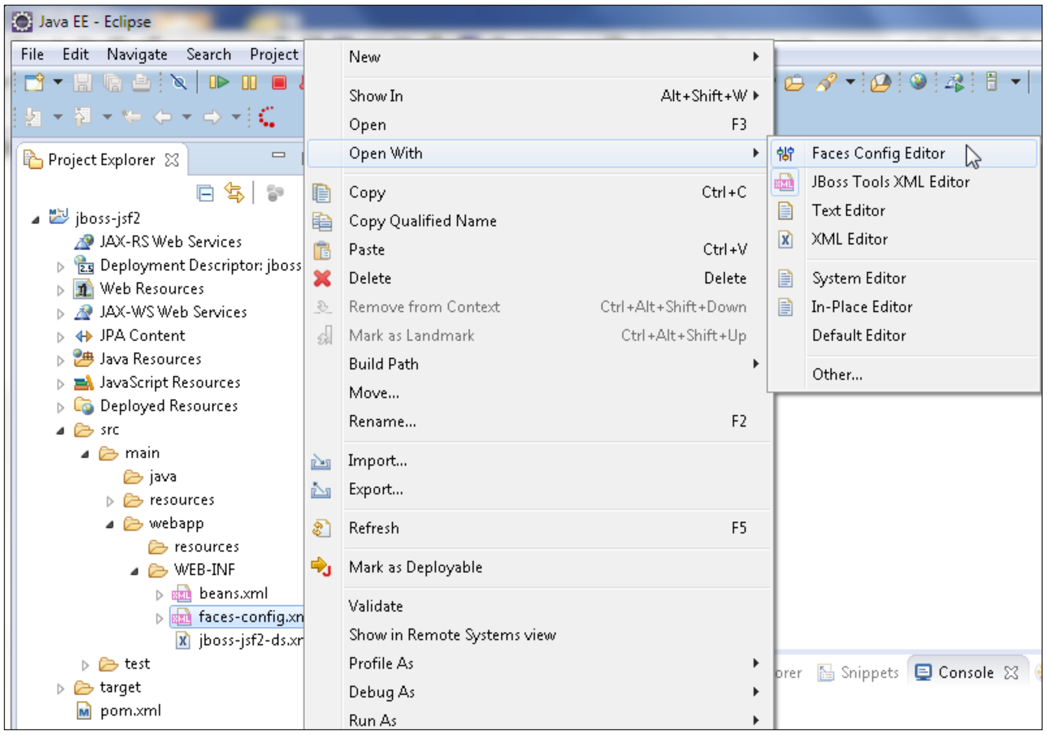
1. **A Facelets template page**: A template may be reused in several Facelets composition pages.
2. **Facelets header and footer pages**: These pages are included in the Facelets template page for common sections of a Facelets application.
3. **A configuration file**: This is the `faces-config.xml` file, which is included by default in a Java EE web project.
4. **Facelets composition page or pages**: These pages are run on the WildFly.
5. **A managed bean**: This is used for the Facelets composition pages.

We will create a Facelets application with an input Facelets composition page and an output Facelets composition page. A common header and footer are included in the input and output pages. In the input page, a SQL query may be specified in an input field. The SQL query is used to get data from the database and create a JSF data table and demonstrate templating. A command button sends the input request parameters to a managed bean's action method. In the action method, a connection is established with MySQL database and a result set generated for the SQL query. A JSF data table is generated from the result set and displayed in the output Facelets composition page.

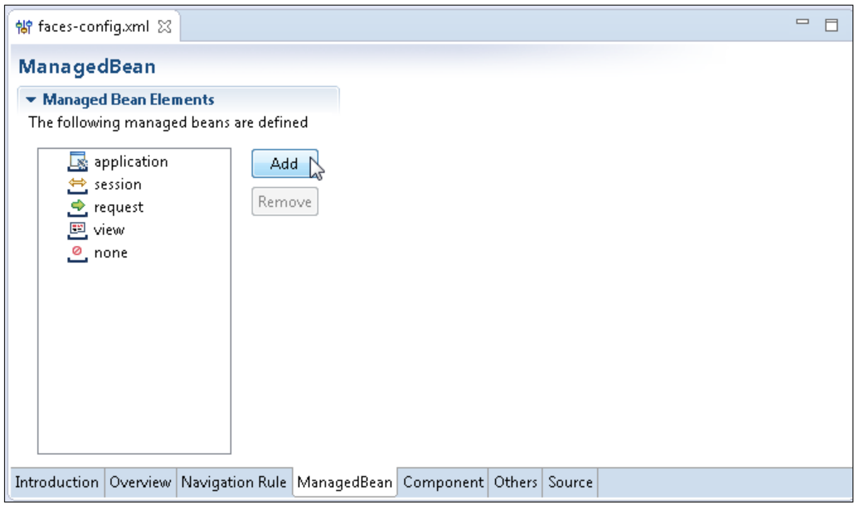
In the subsequent sections, we will create the different components of the Facelets application; first, the managed bean.

Creating a managed bean

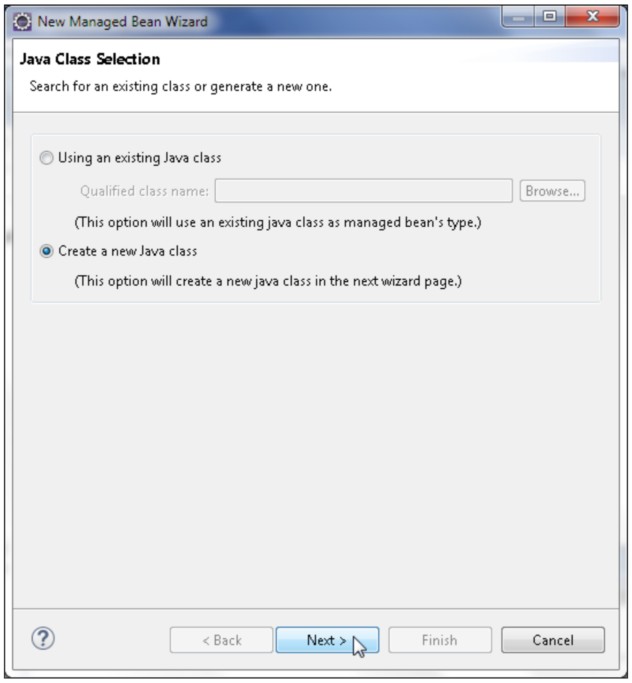
A managed bean is a Java bean managed by the **JSF Managed Bean Facility**. A managed bean is registered with JSF and instantiated when first invoked. In this section, we will create a JSF managed bean for the Facelets composition pages. Right-click on `faces-config.xml` and select **Open With** | **Faces Config Editor**, as shown in the following screenshot:



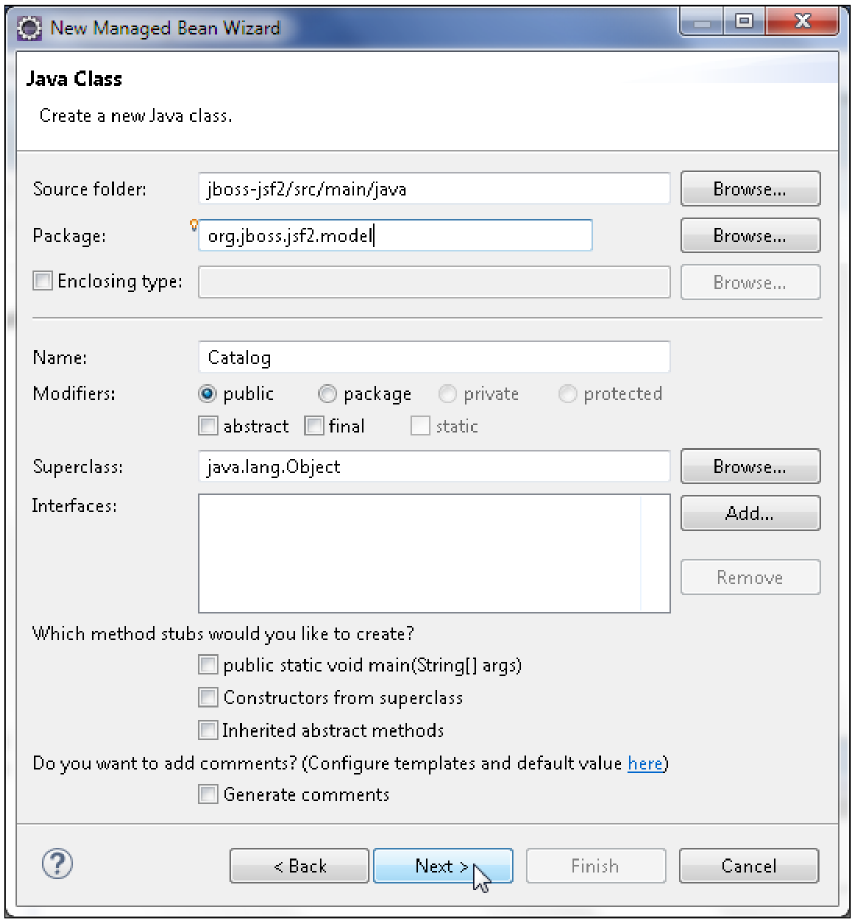
The **Faces Config Editor** gets started. Click on **Add** to add a managed bean, as shown in the following screenshot:



In the **New Managed Bean Wizard**, select **Create a new Java class** in **Java Class Selection** and click on **Next**, as shown in the following screenshot:


New Managed Bean Wizard
Java Class Selection
Search for an existing class or generate a new one.
Using an existing Java class
Qualified class name:
Browse...
(This option will use an existing java class as managed bean's type.)
Create a new Java class
(This option will create a new java class in the next wizard page.)
< Back
Next >
Finish
Cancel

The **Java Class** wizard gets started. The **Source Folder** (`jboss-jsf2/src/main/java`), **Package** (`org.jboss.jsf2.model`), and **Name** (`Catalog`) should be specified, as shown in the following screenshot. Click on **Next**.



Source folder: jboss-jsf2/src/main/java

Package: org.jboss.jsf2.model

☐ Enclosing type:

Name: Catalog

Modifiers: [✓ public] package private protected

☐ abstract ☐ final ☐ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

☐ public static void main(String[] args)

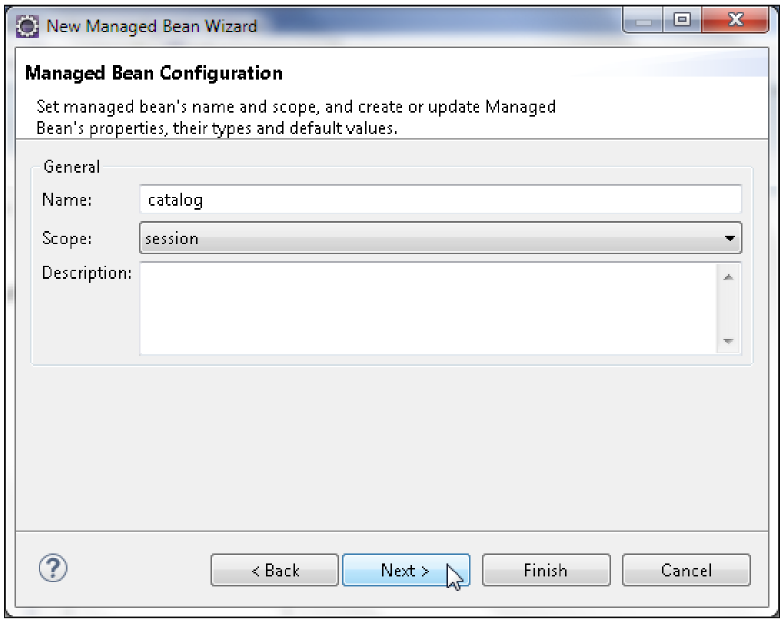
☐ Constructors from superclass

☐ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

☐ Generate comments

In the **Managed Bean Configuration** window, the **Name** textbox is specified as `catalog`, and **Scope** as `session`. Click on **Next**, as shown in the following screenshot:



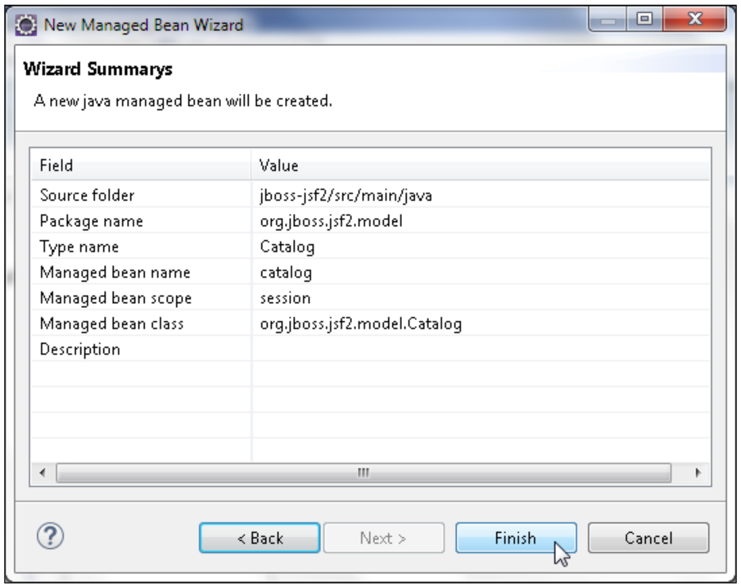
Managed Bean Configuration

Set managed bean's name and scope, and create or update Managed Bean's properties, their types and default values.

General

Name:	catalog
Scope:	session
Description:	

A summary of the managed bean to be created gets displayed in the **Wizards Summarys** window. Click on **Finish**, as shown in the following screenshot:

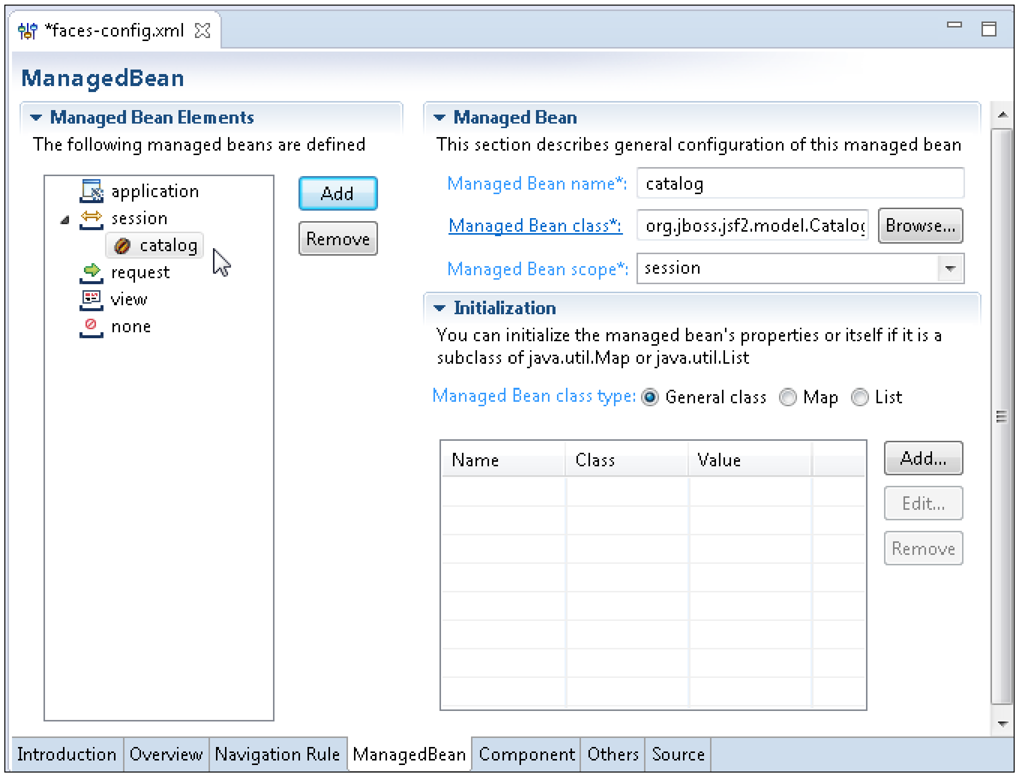


Wizard Summarys

A new java managed bean will be created.

Field	Value
Source folder	jboss-jsf2/src/main/java
Package name	org.jboss.jsf2.model
Type name	Catalog
Managed bean name	catalog
Managed bean scope	session
Managed bean class	org.jboss.jsf2.model.Catalog
Description	

A new managed bean catalog gets created, as shown in the following screenshot:



The following managed beans are defined

- application
- session
  - catalog
- request
- view
- none

This section describes general configuration of this managed bean

Managed Bean name*: catalog

Managed Bean class*: org.jboss.jsf2.model.Catalog

Managed Bean scope*: session

You can initialize the managed bean's properties or itself if it is a subclass of java.util.Map or java.util.List

Managed Bean class type: [✓ General class] Map List

Name	Class	Value

In the managed bean, import the required classes and annotate the `Catalog` class with the `@ManagedBean` annotation. Declare Java bean properties for an input form (of the type `HtmlForm`), input the text field (of the type `HtmlInputText`), a label for the input text field (of the type `HtmlOutputLabel`), an command button (of the type `HtmlCommandButton`), a data table (of the type `HtmlDataTable`), the columns of the data table (of the type `UIColumn`), and an error message (of the type `HtmlOutputText`). Add getter/setter methods for the properties as required by the Java bean convention. Also, declare variables for a `Connection` object, a `Statement` object, and a `ResultSet` object.

Add an action method-a method that has no parameters and returns a `String` value and `commandButton1_action`. In the action method, create a connection to MySQL database either with the JDBC API or using a MySQL data source. If the JDBC API is used, the `Connection` object is obtained as follows:

```
Class.forName("com.mysql.jdbc.Driver");
String url = "jdbc:mysql://localhost:3306/test";
Connection connection = DriverManager.getConnection(url, "root",
"mysql");
```

If a MySQL data source is used, the `Connection` object is obtained as follows:

```
InitialContext initialContext = new InitialContext();
DataSource ds = (DataSource) initialContext.lookup("java:jboss/
datasources/MySQLDS");
java.sql.Connection conn = ds.getConnection();
```

Create a `Statement` object from the `Connection` object using the `createStatement` method. Run the SQL query input in the input field using the `executeQuery` method to obtain a `ResultSet`:

```
Statement stmt = connection.createStatement(ResultSet.TYPE_SCROLL_
INSENSITIVE, ResultSet.CONCUR_READ_ONLY);
ResultSet rs = stmt.executeQuery((String) inputText1.getValue());
```

Create a `HtmlDataTable` object, set its border using the `setBorder` method, set the cell padding using the `setCellpadding` method, and set the iteration variable for a data collection using the `setVar` method:

```
HtmlDataTable dataTable1 = new HtmlDataTable();
dataTable1.setBorder(5);
dataTable1.setCellpadding("1");
dataTable1.setVar("journalcatalog");
```

Create a `ResultSetDataModel` object, which encapsulates a data collection represented by a `ResultSet`. Set the `ResultSet` generated from the SQL query as the data collection for the `ResultSetDataModel` object using the `setWrappedData` method:

```
ResultSetDataModel dataModel = new ResultSetDataModel();
dataModel.setWrappedData(rs);
```

Create columns for the data table using UIColumn class constructors, and set the columns on the data table using the setColumn method of the HtmlDataTable object. Create a data table column header using the HtmlOutputText type variable for each column, and set a header on a column using the setHeader method of the UIColumn object. The data table values are also of the type HtmlOutputText. The result set data is bound to the data table using value expressions. A ValueExpression object is used to set values. An ExpressionFactory object is required to create a ValueExpression object. Obtain a ExpressionFactory object from a FacesContext object using getApplication().getExpressionFactory(). First create a FacesContext object using the getCurrentInstance() method. Create a ValueExpression object from the ExpressionFactory method using the createValueExpression(ELContext context, java.lang.String expression, java.lang.Class<?> expectedType) method. The ELContext object for the createValueExpression method is created from the FacesContext object using the getELContext() method. The expression for the createValueExpression method is an EL expression, and the expectedType method is String.class. A ValueExpression value is set on a HtmlOutputText type column using the setValueExpression method. Add the HtmlOutputText object to a UIColumn object using getChildren().add(). The result set values are not bound on each data table cell individually, but an EL expression consisting of an iteration variable is used to bind the result set. For example, column1, which is the column for the catalog ID, is set as follows:

```
HtmlOutputText column1Text = new HtmlOutputText();
FacesContext fCtx = FacesContext.getCurrentInstance();
ELContext elCtx = fCtx.getELContext();
ExpressionFactory ef = fCtx.getApplication().getExpressionFactory();
ValueExpression ve = ef.createValueExpression(elCtx,"#{journalcatalog.
catalogid}", String.class);
column1Text.setValueExpression("value", ve);
column1.getChildren().add(column1Text);
```

Similarly, set the other data table columns. Set the ResultSetDataModel data collection on the HtmlDataTable object using the setValue() method:

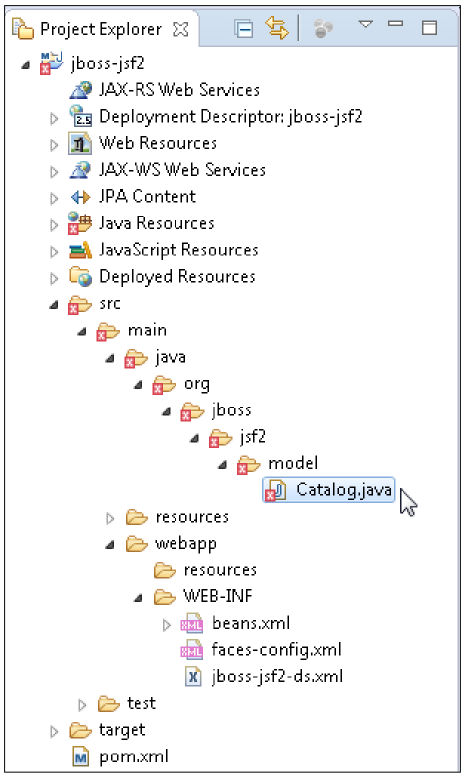
```
dataTable1.setValue(dataModel);
```

If an error is generated, return the error from the action method, which navigates the Facelets application to error.jsp using implicit navigation. If implicit navigation is used, the to-URL is the same name as the String value returned by the action method. For a more detailed discussion on implicit navigation and other JSF 2 features, refer to *JavaServer Faces 2.0, Essential Guide for Developers, Cengage Learning*. If an error is not generated, return the output that navigates to output.jsp to display the data table. The Catalog.java managed bean is available in this chapter download.

As we have used the `@ManagedBean` annotation, the `faces-config.xml` file is an empty file, which is an advantage in terms of having to specify less configuration:

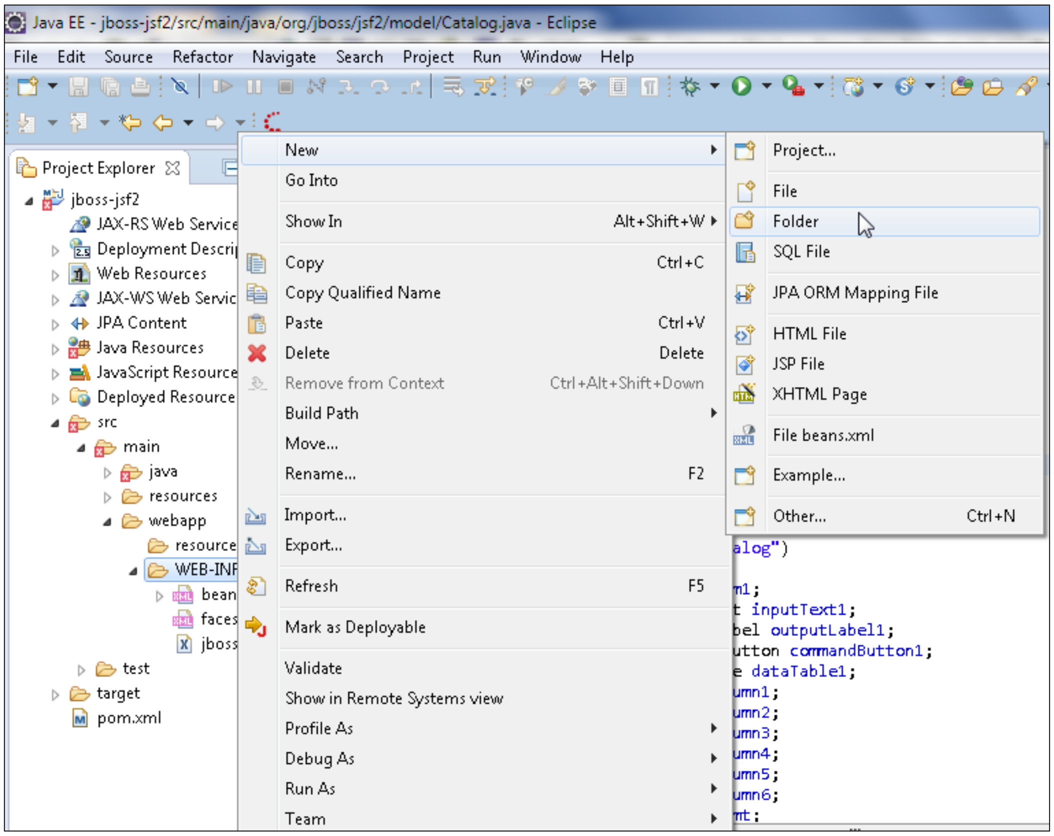
```
<?xml version="1.0" encoding="UTF-8"?>
<!-- This file is not required if you don't need any extra
configuration. -->
<faces-config version="2.0" xmlns="http://java.sun.com/xml/ns/javaee"
 xmlns:xi="http://www.w3.org/2001/XInclude"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="         http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd">
</faces-config>
```

The managed bean is shown in the `jboss-jsf2` project in the next screenshot. The `Catalog.java` Java class may indicate some errors, which are due to some required Maven dependencies not being added in the `pom.xml` file. We will add the required Maven dependencies in a later section.

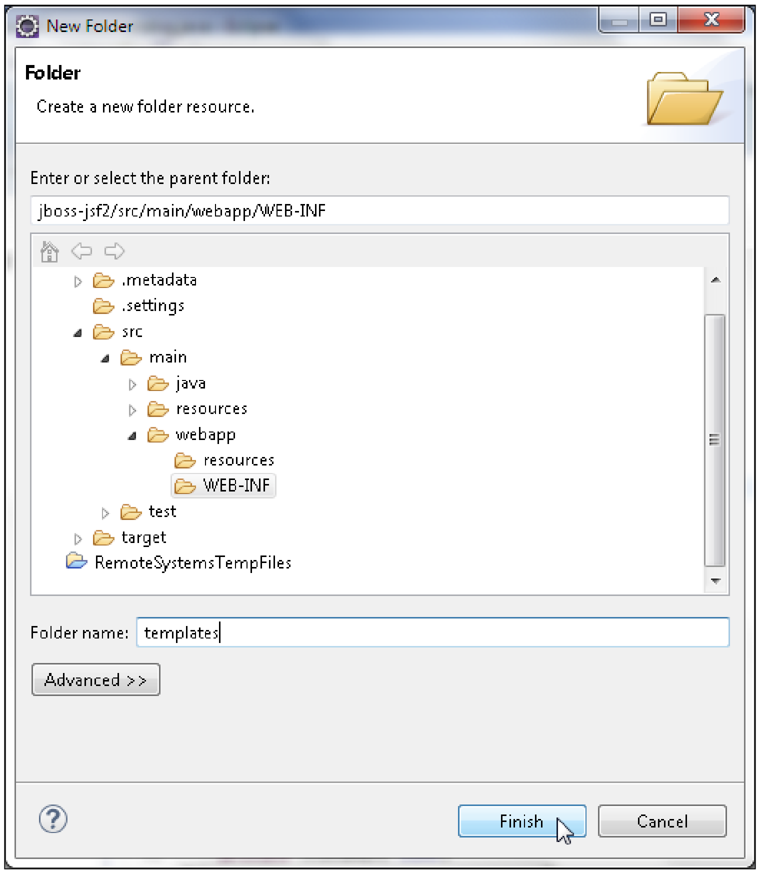


Creating a Facelets template

A Facelets template is a reusable component, which includes one or more JSF pages as different, sections of the template that may be used in multiple Facelets composition pages thus precluding the inclusion of each of the JSF pages that comprise the common sections of Facelets composition pages separately. We will use a Facelets template to include a header JSF page and a footer JSF page. The Facelets templates are to be created in the `WEB-INF/templates` directory for which you need to add a `templates` directory. Right-click on the `WEB-INF` folder in **Project Explorer** and select **New | Folder**, as shown in the following screenshot:



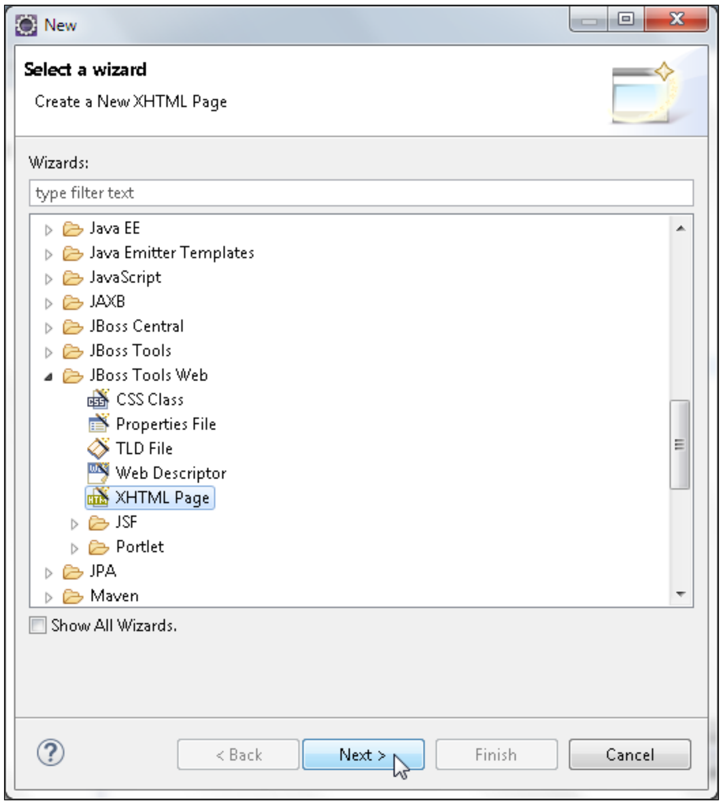
In the **New Folder** wizard, select the `webapp` | `WEB-INF` folder and specify **Folder name** as `templates`, as shown in the following screenshot:



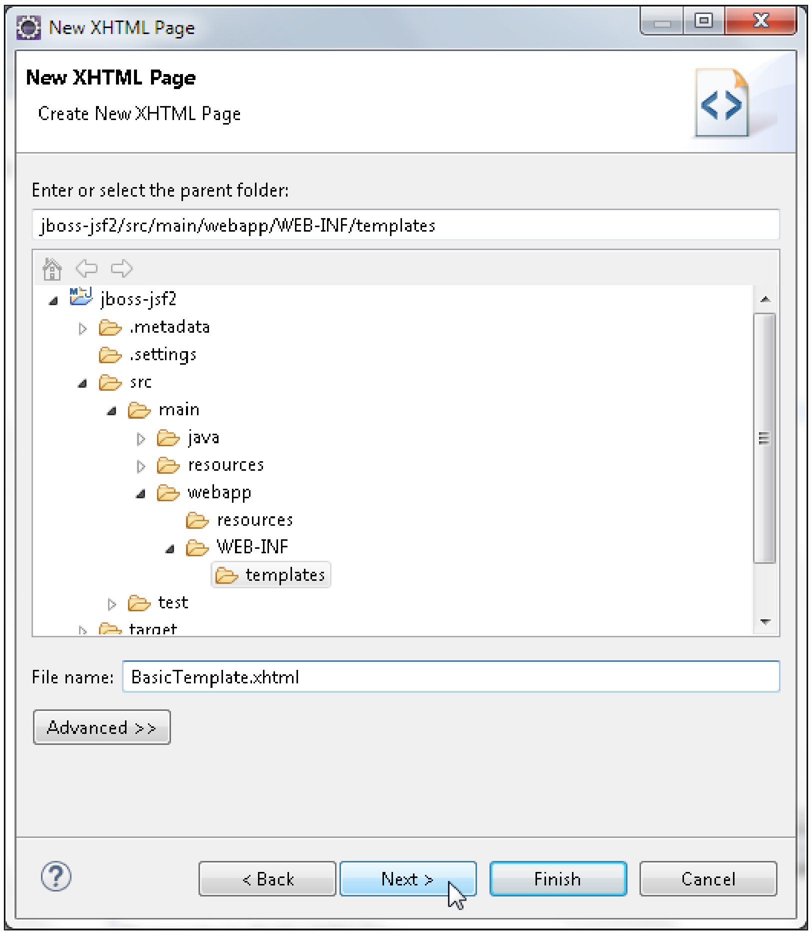
Enter or select the parent folder:
jboss-jsf2/src/main/webapp/WEB-INF

Folder name: templates

To create a Facelets template, select **File** | **New** | **Other**. In **New**, select **JBoss Tools Web** | **XHTML Page** and click on **Next**, as shown in the following screenshot:



In **New XHTML Page** wizard, select the folder as `webapp/WEB-INF/templates` and specify **File name** as `BasicTemplate.xhtml`. Click on **Next**, as shown in the following screenshot:


New XHTML Page

New XHTML Page

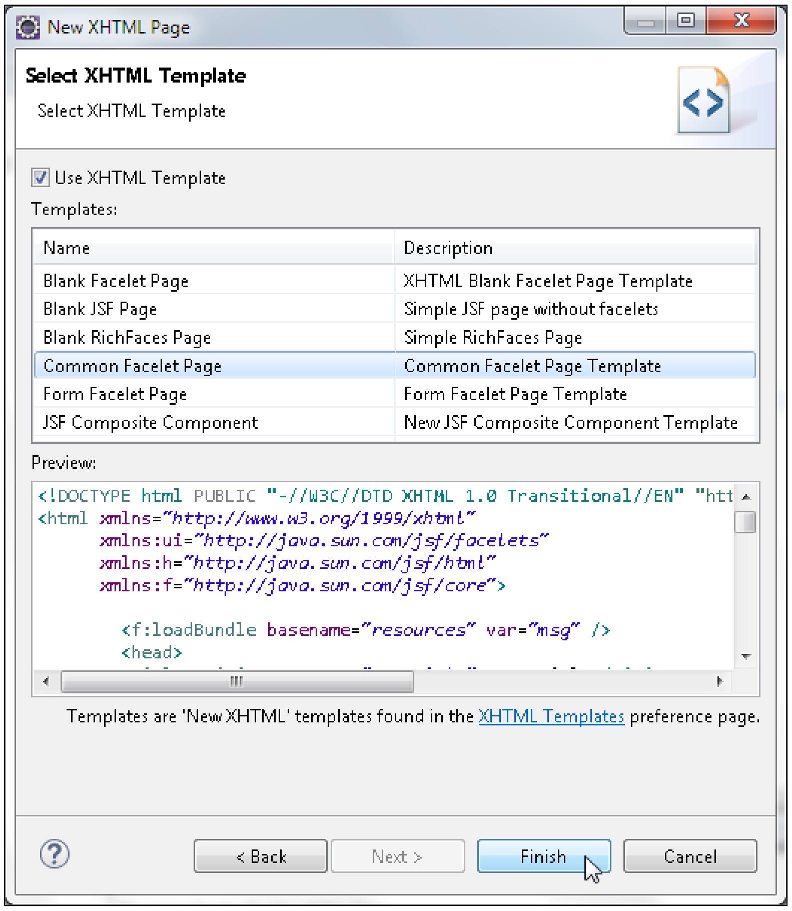
Create New XHTML Page

Enter or select the parent folder:

jboss-jsf2/src/main/webapp/WEB-INF/templates

File name: BasicTemplate.xhtml

In **Select XHTML Template**, select the **Common Facelet Page** template and click on **Finish**, as shown in the following screenshot:



Name	Description
Blank Facelet Page	XHTML Blank Facelet Page Template
Blank JSF Page	Simple JSF page without facelets
Blank RichFaces Page	Simple RichFaces Page
Common Facelet Page	Common Facelet Page Template
Form Facelet Page	Form Facelet Page Template
JSF Composite Component	New JSF Composite Component Template

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "htt
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">

        <f:loadBundle basename="resources" var="msg" />
        <head>
```

Templates are 'New XHTML' templates found in the XHTML Templates preference page.

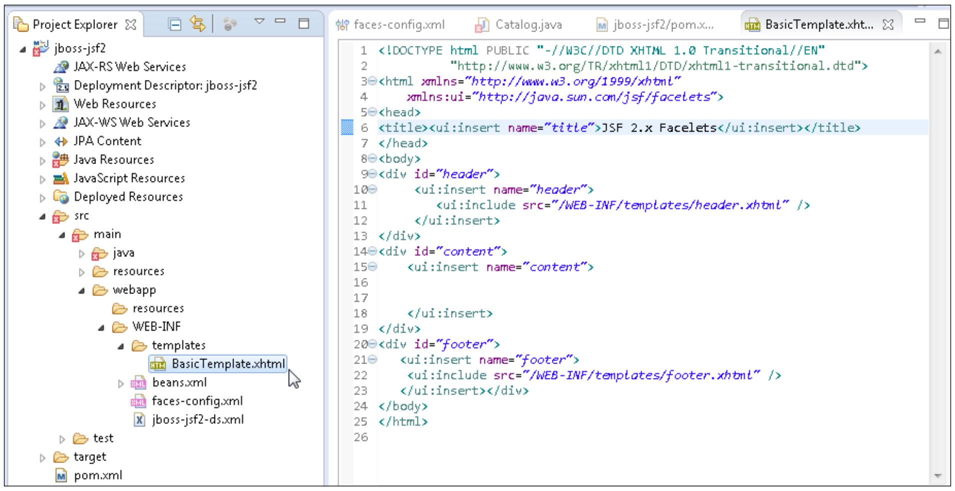
The `BasicTemplate.xhtml` Facelet template gets created in the `WEB-INF/templates` folder. In the template, create `<div/>` elements for header, content, and footer sections of a Facelet composition page. The `<ui:insert/>` Facelets tag is used as a placeholder for a composition page section. As common header and footer sections are required in the input and output composition pages, you need to include a header JSF page in the header `div` and a footer JSF page in the footer `<div>` using the `<ui:include/>` tag. Keep the `<ui:insert/>` element for the content `div` empty for the composition page to include the page section. For example, include a `header.xhtml` JSF page as follows:

```
<ui:insert name="header">
  <ui:include src="/WEB-INF/templates/header.xhtml" />
</ui:insert>
```

We will create the `header.xhtml` and `footer.xhtml` JSF pages in the next section. The `BasicTemplate.xhtml` template is listed as follows:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:ui="http://java.sun.
com/jsf/facelets">
  <head>
    <title><ui:insert name="title">JSF 2.0 Facelets</ui:insert></
title>
  </head>
  <body>
    <div id="header">
      <ui:insert name="header">
        <ui:include src="/WEB-INF/templates/header.xhtml" />
      </ui:insert>
    </div>
      <div id="content">
        <ui:insert name="content">
      </ui:insert>
    </div>
    <div id="footer">
      <ui:insert name="footer">
        <ui:include src="/WEB-INF/templates/footer.xhtml" />
      </ui:insert>
    </div>
  </body>
</html>
```

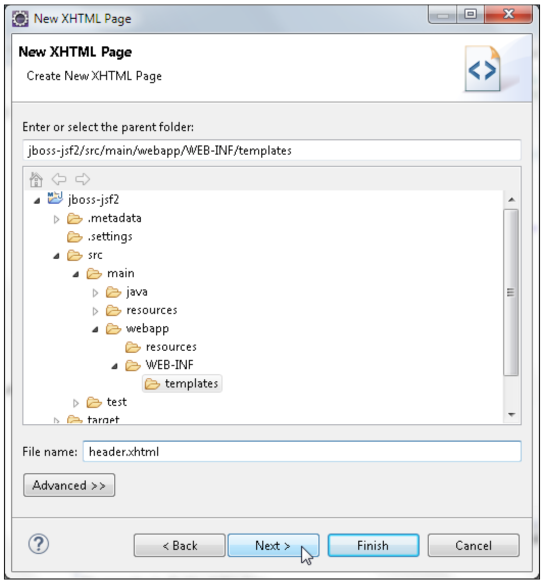
Copy the listing to the `BasicTemplate.xhtml` file in Eclipse IDE. The `BasicTemplate.xhtml` file is shown in the following screenshot:



```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
          "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:ui="http://java.sun.com/jsf/facelets">
<head>
<title><ui:insert name="title">JSF 2.x Facelets</ui:insert></title>
</head>
<body>
<div id="header">
     <ui:insert name="header">
        <ui:include src="/WEB-INF/templates/header.xhtml" />
     </ui:insert>
</div>
<div id="content">
    <ui:insert name="content">


    </ui:insert>
</div>
<div id="footer">
   <ui:insert name="footer">
    <ui:include src="/WEB-INF/templates/footer.xhtml" />
   </ui:insert></div>
</body>
</html>
```

Creating header and footer

In this section, we will create header and footer JSF pages. Similar to creating the `BasicTemplate.xhtml` Facelet template, create `header.xhtml` and `footer.xhtml` in the `WEB-INF/templates` folder. For example, for the header JSF page, specify `header.xhtml`, as shown here:



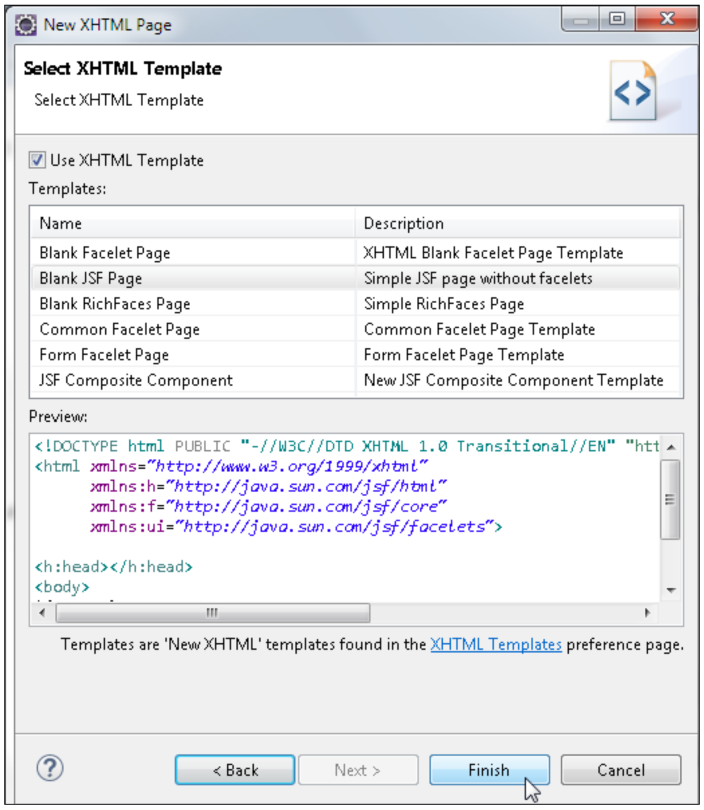
New XHTML Page

Create New XHTML Page

Enter or select the parent folder: jboss-jsf2/src/main/webapp/WEB-INF/templates

File name: header.xhtml

The difference in creating the header and footer pages that in **Select XHTML Template**, select the **Blank JSF Page** template shown as follows:



Name	Description
Blank Facelet Page	XHTML Blank Facelet Page Template
Blank JSF Page	Simple JSF page without facelets
Blank RichFaces Page	Simple RichFaces Page
Common Facelet Page	Common Facelet Page Template
Form Facelet Page	Form Facelet Page Template
JSF Composite Component	New JSF Composite Component Template

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "htt
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:ui="http://java.sun.com/jsf/facelets">

<h:head></h:head>
<body>
```

Templates are 'New XHTML' templates found in the XHTML Templates preference page.

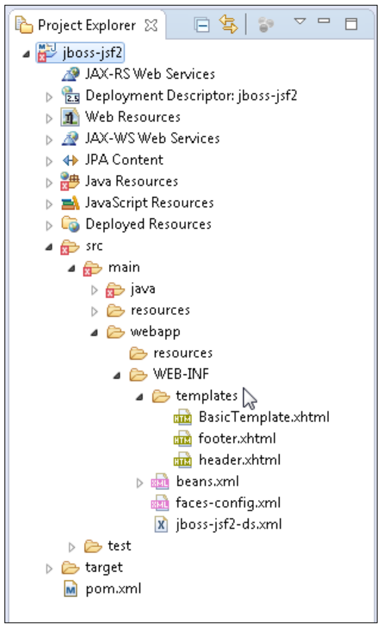
We will include graphic `JPEG` files for the header and footer sections. In the `header.xhtml`, include a `JPEG` file using the `h:graphicImage` tag enclosed in a `h:panelGrid` tag. First, copy the graphic JPEG files `FaceletsHeader.jpg` and `FaceletsFooter.jpg` to the `//jboss-jsf2/src/main/webapp` directory. The `header.xhtml` JSF page is listed as follows:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:f="http://java.sun.com/jsf/core">
  <f:view>
    <h:form>
      <h:panelGrid columns="1">
        <h:graphicImage value="FaceletsHeader.jpg" />
      </h:panelGrid>
    </h:form>
  </f:view>
</html>
```

Similarly, in the `footer.xhtml` JSF page, include a `FaceletsFooter.jpg` image file with the `h:graphicImage` tag enclosed in the `h:panelGrid` tag. The `footer.xhtml` page is listed as follows:

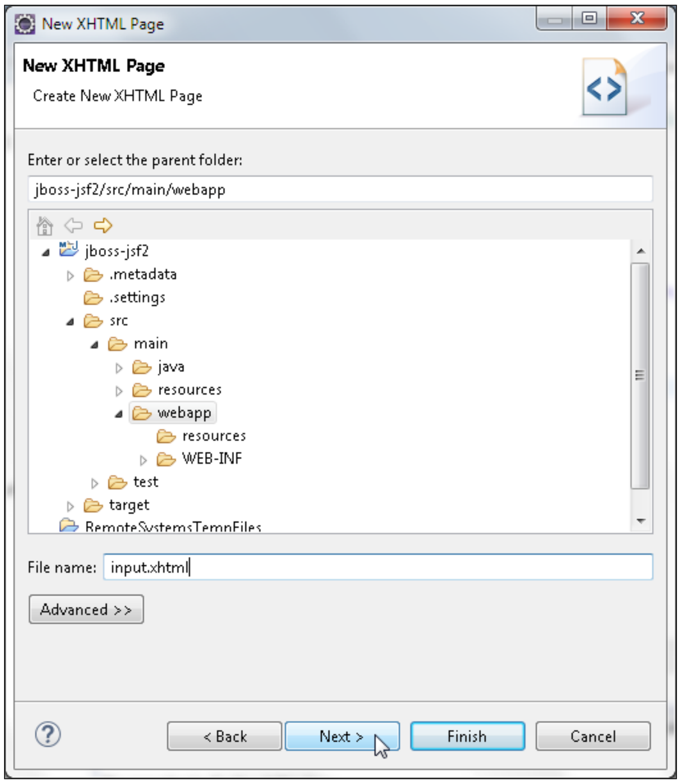
```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:f="http://java.sun.com/jsf/core">
  <f:view>
    <h:form>
      <h:panelGrid columns="1">
        <h:graphicImage value="FaceletsFooter.jpg" />
      </h:panelGrid>
    </h:form>
  </f:view>
</html>
```

The directory structure of the `jboss-jsf2` project is shown in the following screenshot:

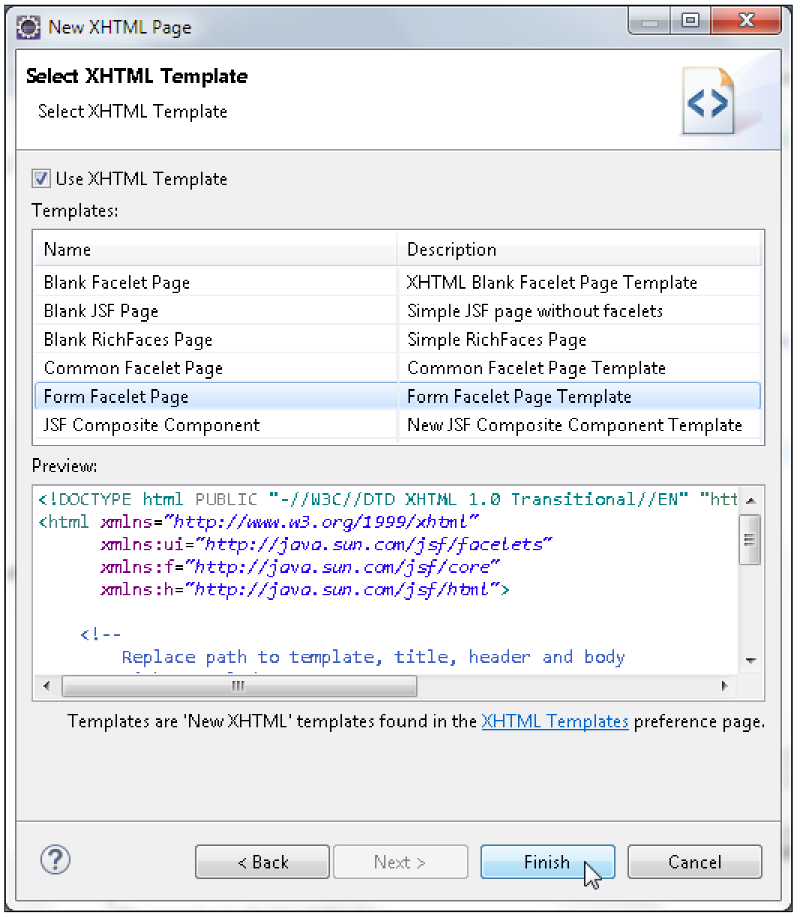


Creating input and output Facelets composition pages

In this section, we will create input and output Facelets composition pages, `input.xhtml` and `output.xhtml`. Create the Facelets pages similar to creating the `BasicTemplate.xhtml`, `header.xhtml`, and `footer.xhtml` to that select the folder to create `input.xhtml` and `output.xhtml` as `webapp`. For example, specify `input.xhtml` in **Create New XHTML Page**, as shown here:



And in **Select XHTML Template**, select **Form Facelet Page Template** and click on **Finish**, as shown in the following screenshot:



Name	Description
Blank Facelet Page	XHTML Blank Facelet Page Template
Blank JSF Page	Simple JSF page without facelets
Blank RichFaces Page	Simple RichFaces Page
Common Facelet Page	Common Facelet Page Template
Form Facelet Page	Form Facelet Page Template
JSF Composite Component	New JSF Composite Component Template

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "htt
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html">

    <!--
        Replace path to template, title, header and body
```

Templates are 'New XHTML' templates found in the XHTML Templates preference page.

In the `BasicTemplate.xhtml` file, we defined the structure of a template that may be reused in composition pages. The header and footer `div` tags are included in the `header.xhtml` and `footer.xhtml` files respectively. In the `input.xhtml` file, include the `BasicTemplate.xhtml` file using the `ui:composition` tag's `template` attribute. Specify the relative path to the template. We only need to define the content section of the `input.xhtml` composition page. The placeholder in the `BasicTemplate.xhtml` file is specified using `<ui:insert name="content"/>`. Specify the actual definition in the `input.xhtml` with `<ui:define name="content"/>`. Within the `ui:define` tag, add the JSF components for an input text with a corresponding output label, and a command button to invoke the action method in the managed bean `catalog`. The components have binding with corresponding managed bean properties using EL expression (`http://docs.oracle.com/javaee/6/tutorial/doc/gjddd.html`). Enclose the JSF components within `h:panelGrid`, which is enclosed within a `h:form` tag. The Facelet composition page is listed as follows:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:f="http://java.sun.com/jsf/core">
  <ui:composition template="/WEB-INF/templates/BasicTemplate.xhtml">
    <ui:define name="content">
      <h:form>
        <h:panelGrid columns="2">
          <h:outputLabel binding="#{catalog.outputLabel1}" value="SQL
Query:" />
          <h:inputText binding="#{catalog.inputText1}" />
        <h:commandButton value="Submit" binding="#{catalog.
commandButton1}" action="#{catalog.commandButton1_action}" />
        </h:panelGrid>
      </h:form>
    </ui:define>
  </ui:composition>
</html>
```

In the `output.xhtml` file, include `BasicTemplate.xhtml` with the `template` attribute of the `ui:composition` tag. Define the content section using the `ui:define` tag. Within the `ui:define` tag, add a `h:dataTable` tag for a data table. Specify binding to the managed bean property `dataTable1` using EL expression. Set the data table border with the border attribute of `h:dataTable` and set the number of rows to 5 using the `rows` attribute. Within the `h:dataTable` tag, add six `h:column` tags for the data table columns. Specify binding of the `h:column` tags to the managed bean properties.

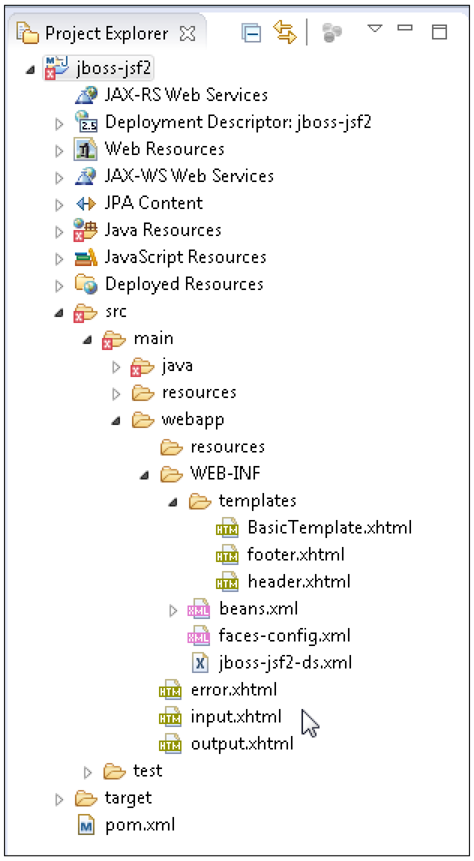
The `output.xhtml` page is listed as follows:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:f="http://java.sun.com/jsf/core">
  <ui:composition template="/WEB-INF/templates/BasicTemplate.xhtml">
    <ui:define name="content">
      <h:form>
        <h:dataTable  binding="#{catalog.dataTable1}" border="1"
rows="5">
          <h:column binding="#{catalog.column1}"></h:column>
          <h:column binding="#{catalog.column2}"></h:column>
          <h:column binding="#{catalog.column3}"></h:column>
          <h:column binding="#{catalog.column4}"></h:column>
          <h:column binding="#{catalog.column5}"></h:column>
          <h:column binding="#{catalog.column6}"></h:column>
        </h:dataTable>
      </h:form>
    </ui:define>
  </ui:composition>
</html>
```

For a more detailed discussion on JSF 2 features, refer to *JavaServer Faces 2.0, Essential Guide for Developers, Cengage Learning*. Add an `error.xhtml` JSF page for displaying an error message. The `error.xhtml` page is not a Facelets composition page and just has a `h:outputLabel` tag with binding to the `errorMsg` managed bean property:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:ui="http://java.sun.com/jsf/facelets">
  <head>
    <title>Error Page</title>
  </head>
  <body>Error Page<h:outputLabel binding="#{catalog.errorMsg}"
value="#{catalog.errorMsg}" />
  </body>
</html>
```

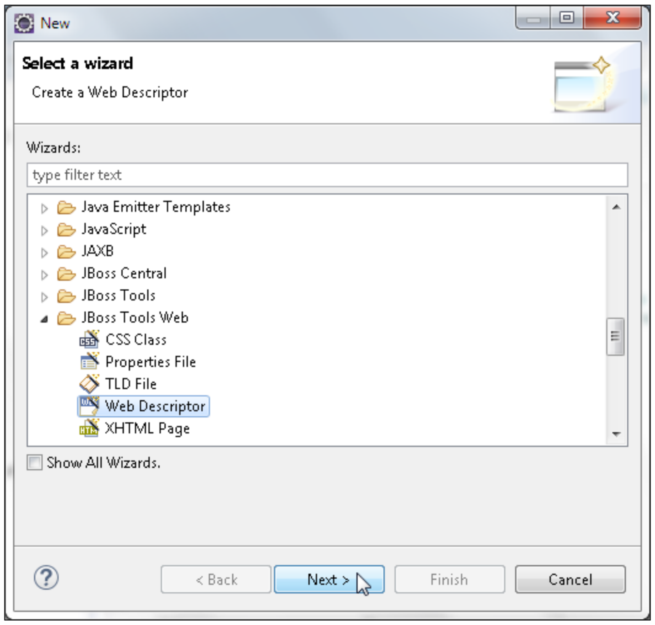
The template `BasicTemplate.xhtml`, the header `header.xhtml`, the footer `footer.xhtml` and the `input.xhtml` and `output.xhtml` composition pages are shown in the **Project Explorer** tab:



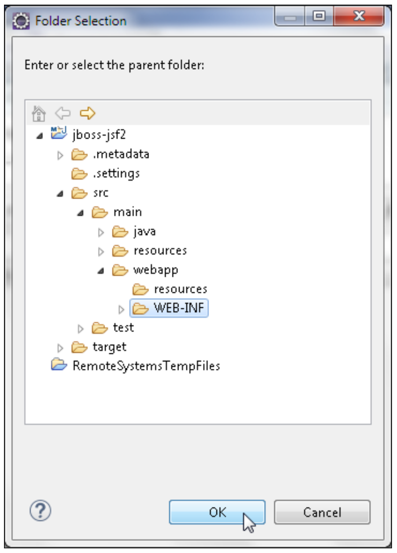
Creating a web descriptor

A web descriptor (`web.xml`) is not a requirement in Java EE 7, but for a JSF application, we need to configure the Facelets servlet. To create a web descriptor, select **File** | **New** | **Other**.

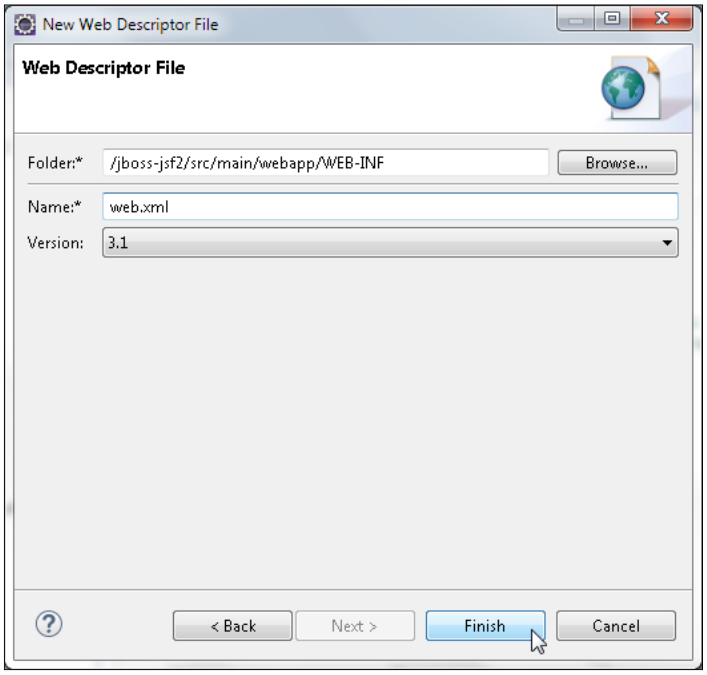
In **New**, select **JBoss Tools Web | Web Descriptor** and click on **Next**, as shown in the following screenshot:



In the **Web Descriptor File** wizard for **Folder**, click on **Browse** to select a folder. In **Folder Selection**, select the **WEB-INF** folder and click on **OK**, as shown in the following screenshot:



Select the **WEB-INF** folder and specify **Name** as `web.xml`, **Version** as `3.1` and click on **Finish**, as shown in the following screenshot:



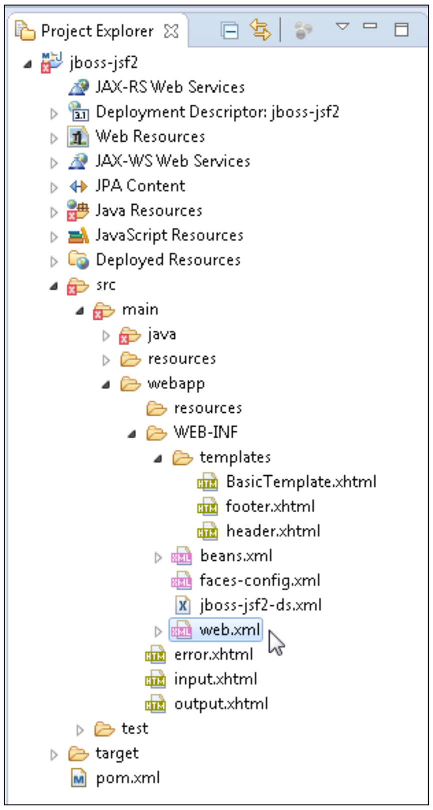
Folder:* /jboss-jsf2/src/main/webapp/WEB-INF
Name:* web.xml
Version: 3.1

A `web.xml` file gets created in the `WEB-INF` folder. In `web.xml`, specify the Faces servlet and its servlet mappings. The `web.xml` web descriptor is listed as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/
javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.
jcp.org/xml/ns/javaee/web-app_3_1.xsd">
<display-name>JSF 2.x Facelets</display-name>

  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.jsf</url-pattern>
  </servlet-mapping>
```

```
    <servlet-mapping>
      <servlet-name>Faces Servlet</servlet-name>
      <url-pattern>*.faces</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
      <servlet-name>Faces Servlet</servlet-name>
      <url-pattern>*.xhtml</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
      <servlet-name>Faces Servlet</servlet-name>
      <url-pattern>/faces/*</url-pattern>
    </servlet-mapping>
  </web-app>
```

The `web.xml` is shown in the following `jboss-jsf2` web project:



Deploying the web project with Maven

The Java EE web project we created is based on Maven. It includes `pom.xml` to build, compile, and package the web application. The default project in WildFly 8.1 is a Java EE 7 version. As the Java EE project is a web project, the packaging gets specified as `war` in `pom.xml`. The `Group Id` and `Artifact Id` attributes that we specified in creating a Java EE web project get configured in `pom.xml` as well. As we are using MySQL database, we need to add the dependency on the MySQL JDBC driver:

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>5.1.22</version>
</dependency>
```

The dependency on the Java EE 7 JSF 2.2 API, which is provided by WildFly 8.1, is included in `pom.xml` by default.:

```
<dependency>
  <groupId>org.jboss.spec.javax.faces</groupId>
  <artifactId>jboss-jsf-api_2.2_spec</artifactId>
  <scope>provided</scope>
</dependency>
```

As the managed bean and the Facelets composition pages use EL expressions, include a dependency on `el-api`, which is not provided by default by WildFly 8.1:

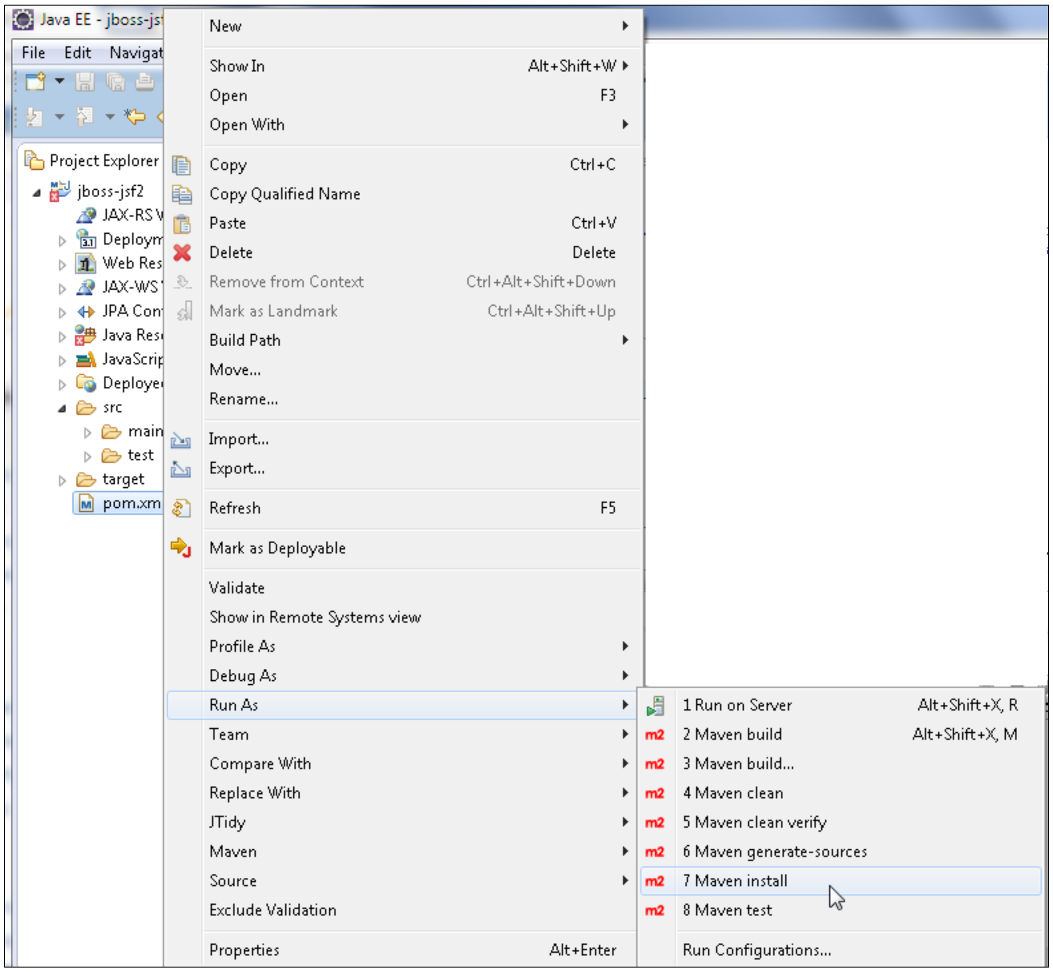
```
<dependency>
  <groupId>javax.el</groupId>
  <artifactId>javax.el-api</artifactId>
  <version>3.0.0</version>
</dependency>
```

The **CDI** (**Context and Dependency Injection**) API, Common Annotations API, JAX-RS API, **JPA** (**Java Persistence API**), **EJB** (**Enterprise JavaBeans**) API, Hibernate Validator API, annotation processor to generate the JPA metamodel classes, Hibernate validator annotation processor, and JUnit API are provided by WildFly 8.1 by default and can be removed if not being used in the sample application in this chapter. In the `build` element, the `Compiler` plugin and the `maven-war-plugin` get configured by default. In the configuration for the `maven-war-plugin` plugin, specify the output directory for the `WAR` archive as the `deployments` directory of the WildFly 8.1 installation.

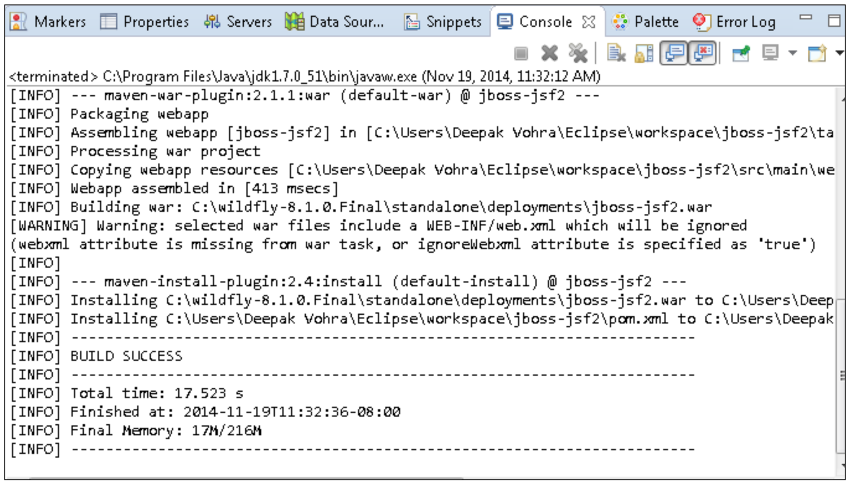
The pom.xml file is listed as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.
apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.jboss.jsf2</groupId>
  <artifactId>jboss-jsf2</artifactId>
  <version>1.0.0</version>
  <packaging>war</packaging>
  <name>WildFly JSF 2.x</name>
  <description>A starter Java EE 7 webapp project for use on JBoss
WildFly / WildFly, generated from the jboss-javaee6-webapp archetype</
description>
  <url>http://wildfly.org</url>
  <licenses>
    <license>
      <name>Apache License, Version 2.0</name>
      <distribution>repo</distribution>
      <url>http://www.apache.org/licenses/LICENSE-2.0.html</url>
    </license>
  </licenses>
  <properties>
    <!-- Explicitly declaring the source encoding eliminates the
following message: -->
    <!-- [WARNING] Using platform encoding (UTF-8 actually) to copy
filtered resources, i.e. build is platform dependent! -->
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <!-- JBoss dependency versions -->
    <version.wildfly.maven.plugin>1.0.2.Final</version.wildfly.maven.
plugin>
    <!-- Define the version of the JBoss BOMs we want to import to
specify tested stacks. -->
    <version.jboss.bom>8.1.0.Final</version.jboss.bom>
    <version.arquillian.container>8.1.0.Final</version.arquillian.
container>
    <!-- other plugin versions -->
    <version.compiler.plugin>3.1</version.compiler.plugin>
    <version.surefire.plugin>2.16</version.surefire.plugin>
    <version.war.plugin>2.1.1</version.war.plugin>
    <!-- maven-compiler-plugin -->
    <maven.compiler.target>1.7</maven.compiler.target>
    <maven.compiler.source>1.7</maven.compiler.source>
  </properties>
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.wildfly.bom</groupId>
        <artifactId>jboss-javaee-7.0-with-tools</artifactId>
        <version>${version.jboss.bom}</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
      <dependency>
        <groupId>org.wildfly.bom</groupId>
        <artifactId>jboss-javaee-7.0-with-hibernate</artifactId>
        <version>${version.jboss.bom}</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>5.1.22</version>
    </dependency>
    <dependency>
      <groupId>javax.el</groupId>
      <artifactId>javax.el-api</artifactId>
      <version>3.0.0</version>
    </dependency>
    <!-- Import the JSF API, we use provided scope as the API is
included in JBoss WildFly -->
    <dependency>
      <groupId>org.jboss.spec.javax.faces</groupId>
      <artifactId>jboss-jsf-api_2.2_spec</artifactId>
      <scope>provided</scope>
    </dependency>
  </dependencies>
  <build>
    <!-- Maven will append the version to the finalName (which is the 
name given to the generated war, and hence the context root) -->
    <finalName>${project.artifactId}</finalName>
    <plugins>
      <!-- Compiler plugin enforces Java 1.6 compatibility and 
activates annotation processors -->
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>${version.compiler.plugin}</version>
        <configuration>
          <source>${maven.compiler.source}</source>
          <target>${maven.compiler.target}</target>
        </configuration>
      </plugin>
      <plugin>
        <artifactId>maven-war-plugin</artifactId>
        <version>${version.war.plugin}</version>
        <configuration>
          <outputDirectory>C:\wildfly-8.1.0.Final\standalone\
deployments</outputDirectory>
          <!-- Java EE 7 doesn't require web.xml, Maven needs to catch 
up! -->
          <failOnMissingWebXml>false</failOnMissingWebXml>
        </configuration>
      </plugin>
      <!-- The WildFly plugin deploys your war to a local WildFly 
container -->
      <!-- To use, run: mvn package wildfly:deploy -->
      <plugin>
        <groupId>org.wildfly.plugins</groupId>
        <artifactId>wildfly-maven-plugin</artifactId>
        <version>${version.wildfly.maven.plugin}</version>
      </plugin>
    </plugins>
  </build>
</project>
```

Right-click on the `pom.xml` file and select **Run As | Maven install**, as shown in the following screenshot:

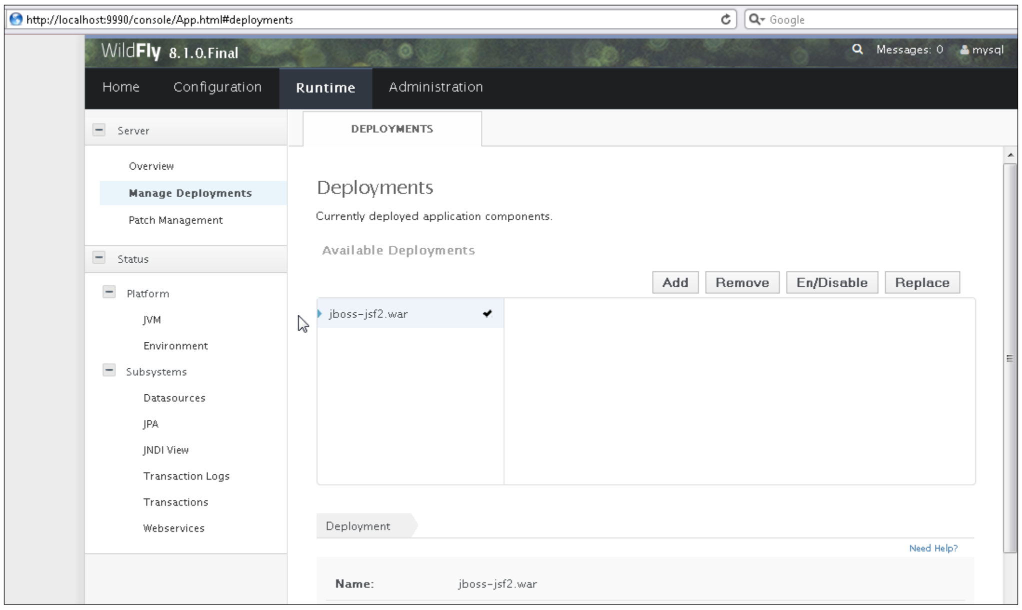


The Maven `pom.xml` build completes with a `BUILD SUCCESS` message, as shown in the following screenshot. The `jboss-jsf2` application gets compiled, packaged, and copied to the `deployments` folder, as all `WAR` archives copied to the `deployments` directory get installed automatically and `jboss-js2.war` gets deployed to WildFly 8.1.



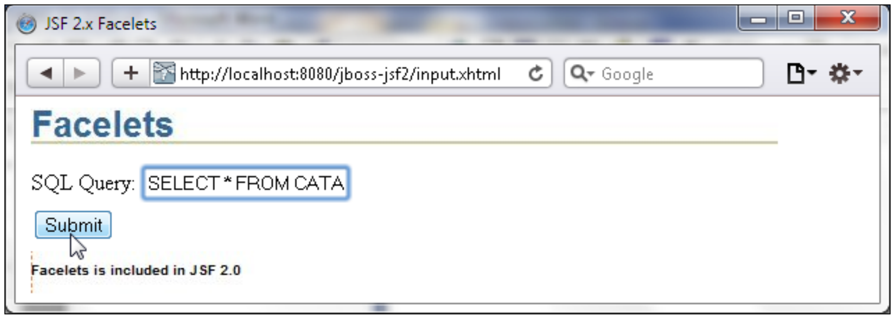
```
<terminated> C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Nov 19, 2014, 11:32:12 AM)
[INFO] --- maven-war-plugin:2.1.1:war (default-war) @ jboss-jsf2 ---
[INFO] Packaging webapp
[INFO] Assembling webapp [jboss-jsf2] in [C:\Users\Deepak Vohra\Eclipse\workspace\jboss-jsf2\ta
[INFO] Processing war project
[INFO] Copying webapp resources [C:\Users\Deepak Vohra\Eclipse\workspace\jboss-jsf2\src\main\we
[INFO] Webapp assembled in [413 msecs]
[INFO] Building war: C:\wildfly-8.1.0.Final\standalone\deployments\jboss-jsf2.war
[WARNING] Warning: selected war files include a WEB-INF/web.xml which will be ignored
(webxml attribute is missing from war task, or ignoreWebxml attribute is specified as 'true')
[INFO]
[INFO] --- maven-install-plugin:2.4:install (default-install) @ jboss-jsf2 ---
[INFO] Installing C:\wildfly-8.1.0.Final\standalone\deployments\jboss-jsf2.war to C:\Users\Deep
[INFO] Installing C:\Users\Deepak Vohra\Eclipse\workspace\jboss-jsf2\pom.xml to C:\Users\Deepak
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 17.523 s
[INFO] Finished at: 2014-11-19T11:32:36-08:00
[INFO] Final Memory: 17M/216M
[INFO] ------------------------------------------------------------------------
```

Navigate to the **Adminstration Console** in WildFly 8.1 using the URL `http://localhost:8080/`. The `jboss-jsf2.war` archive is listed as deployed, as shown in the following screenshot:

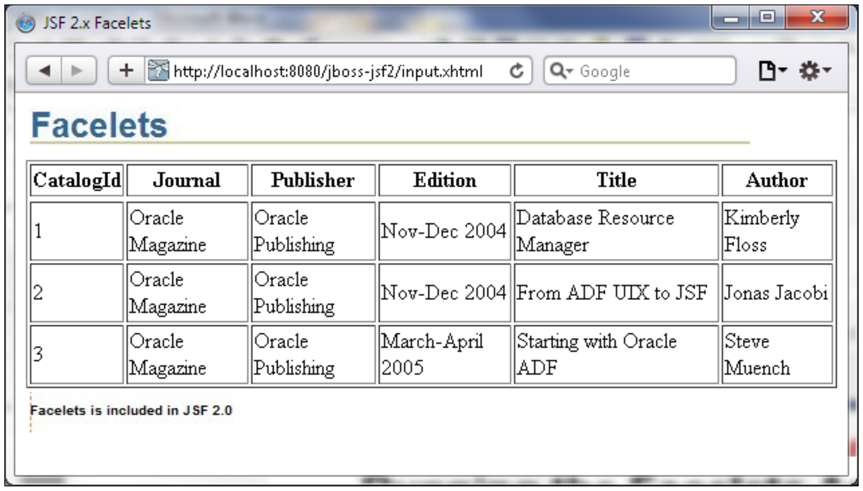


Running the Facelets application

In this section, we will run the JSF 2 application on WildFly 8. Invoke the `input.xhtml` file using the URL `http://localhost:8080/jboss-jsf2/input.xhtml`. The `jboss-jsf2` is included as it is the context root for the `jboss-jsf2.war` application. The header and footer graphics JPEG files are included in the Facelets composition page using templating. Specify an SQL query, for example, `SELECT * FROM CATALOG` and click on **Submit**, as shown in the following screenshot:


JSF 2.x Facelets
http://localhost:8080/jboss-jsf2/input.xhtml
Google
Facelets
SQL Query:
SELECT * FROM CATA
Submit
Facelets is included in JSF 2.0

The `input.xhtml` page invokes the Facelets Servlet as `.xhtml` is specified in the servlet mapping. The `action` method of the managed bean generates a JSF data table and returns the output, which renders `output.xhtml`. The URL displayed in the browser stays the same because the request dispatcher sends a request forward, which does not start a new request. To display the `output.xhtml` file in the browser URL, a redirect will be required, which starts a new request. The same header and footer are included in the `output.xhml` file using templating, as shown in the following screenshot:



CatalogId	Journal	Publisher	Edition	Title	Author
1	Oracle Magazine	Oracle Publishing	Nov-Dec 2004	Database Resource Manager	Kimberly Floss
2	Oracle Magazine	Oracle Publishing	Nov-Dec 2004	From ADF UIX to JSF	Jonas Jacobi
3	Oracle Magazine	Oracle Publishing	March-April 2005	Starting with Oracle ADF	Steve Muench

Facelets is included in JSF 2.0

Summary

In this chapter, we created a JSF 2.x application to create a data table from an SQL query. We used Facelets based templating to include the same header and footer. The JSF 2.0 application is compiled and packaged using the Maven build tool and deployed to WildFly 8.1. The Facelets application is run on a browser to generate a data table.

In the next chapter, we will discuss using Ajax with WildFly 8.1.

4
Using Ajax

Asynchronous JavaScript and XML (**AJAX** or **Ajax**) is a technique for transferring data between a browser and a server asynchronously. "Asynchronously" implies that the web page continues to be processed while the request is sent from the browser to the server and a response is received. Ajax is based on JavaScript, **Document Object Model** (**DOM**), and `XMLHttpRequest`. Ajax provides dynamic interaction between a browser and a server and can be used in several types of applications, such as for validating a form that requires a unique identifier without submitting the form, autocompleting input fields based on partial input, and refreshing information on a web page periodically without having to reload the web page, thus incurring less bandwidth usage.

In this chapter, we will create an Ajax application in the Eclipse IDE, compile and package the application using **Maven**, and run the web application on WildFly 8.1 with MySQL database. In this chapter, we will cover the following topics:

- Setting up the environment
- Creating a Java EE web project
- Creating a user interface
- Creating a servlet
- Deploying the Ajax application with Maven
- Running the Ajax application

Setting up the environment

We need to install the following software:

- **WildFly 8.1.0.Final**: Download `wildfly-8.1.0.Final.zip` from `http://wildfly.org/downloads/`.

- **MySQL 5.6 Database-Community Edition**: Download this edition from `http://dev.mysql.com/downloads/mysql/`. When installing MySQL, also install **Connector/J**.
- **Eclipse IDE for Java EE Developers**: Download Eclipse Luna from `https://www.eclipse.org/downloads/packages/release/Luna/SR1`.
- **JBoss Tools (Luna) 4.2.0.Final (or the latest version)**: Install this as a plugin to Eclipse from Eclipse Marketplace (`http://tools.jboss.org/downloads/installation.html`).
- **Apache Maven**: Download version 3.05 or higher from `http://maven.apache.org/download.cgi`.
- **Java 7**: Download Java 7 from `http://www.oracle.com/technetwork/java/javase/downloads/index.html?ssSourceSiteId=ocomcn`.

Set the `JAVA_HOME`, `JBOSS_HOME`, `MAVEN_HOME`, and `MYSQL_HOME` environment variables. Add `%JAVA_HOME%/bin`, `%MAVEN_HOME%/bin`, `%JBOSS_HOME%/bin`, and `%MYSQL_HOME%/bin` to the `PATH` environment variable.

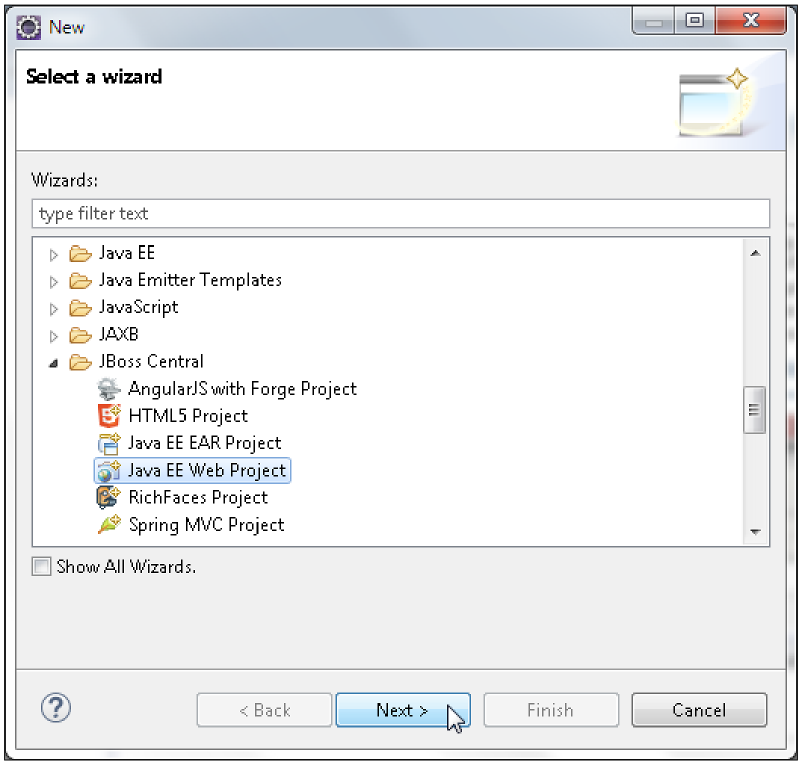
Create a WildFly 8.1.0 runtime as discussed in *Chapter 1, Getting Started with EJB 3.x*.

```
Create a MySQL database CATALOG with the following SQL script:CREATE
TABLE Catalog(CatalogId VARCHAR(255), Journal VARCHAR(255), Publisher
Varchar(255), Edition VARCHAR(255), Title Varchar(255), Author
Varchar(255));

INSERT INTO Catalog VALUES('catalog1', 'Oracle Magazine',  'Oracle
Publishing', 'September-October 2010', 'Using Oracle Essbase Release
11.1.2 Aggregate Storage Option Databases', 'Mark Rittman and
VenkatakrishnanJanakiraman');

INSERT INTO Catalog VALUES('catalog2', 'Oracle Magazine',   'Oracle
Publishing', 'July-August 2010', 'Infrastructure Software and
Virtualization', 'David Baum');
```

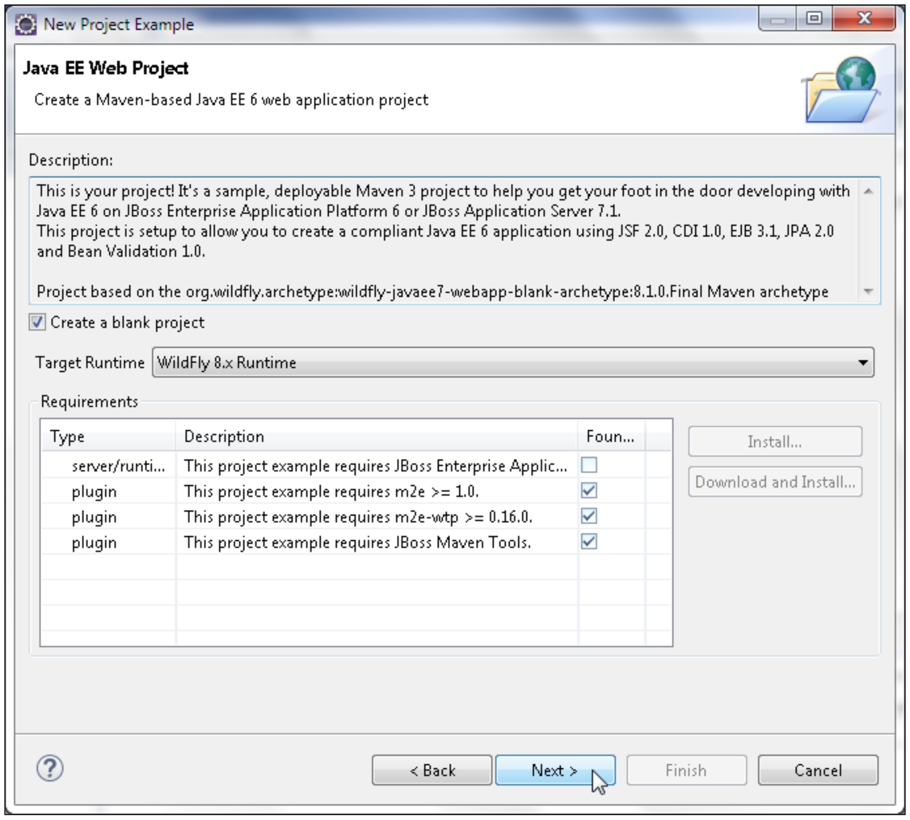
We will use the same MySQL data source we used in earlier chapters. The procedure to create a MySQL module, define a MySQL driver, and configure a data source is discussed in *Chapter 1, Getting Started with EJB 3.x*.

Creating a Java EE web project

In this section, we will create a Java EE web project in Eclipse. Select **File** | **New** | **Other**. In **New**, select **JBoss Central** | **Java EE Web Project** and click on **Next**, as shown in the following screenshot:



The **Java EE Web Project** wizard gets started. A test gets run for the requirements, which include the **m2e**, **m2eclipse-wtp**, and **JBoss Maven Tools** plugins. Select the **Create a blank project** checkbox and **Target Runtime** as `WildFly 8.x Runtime`, as shown in the following screenshot. Then click on **Next**.



This is your project! It's a sample, deployable Maven 3 project to help you get your foot in the door developing with Java EE 6 on JBoss Enterprise Application Platform 6 or JBoss Application Server 7.1.
This project is setup to allow you to create a compliant Java EE 6 application using JSF 2.0, CDI 1.0, EJB 3.1, JPA 2.0 and Bean Validation 1.0.

Project based on the org.wildfly.archetype:wildfly-javaee7-webapp-blank-archetype:8.1.0.Final Maven archetype

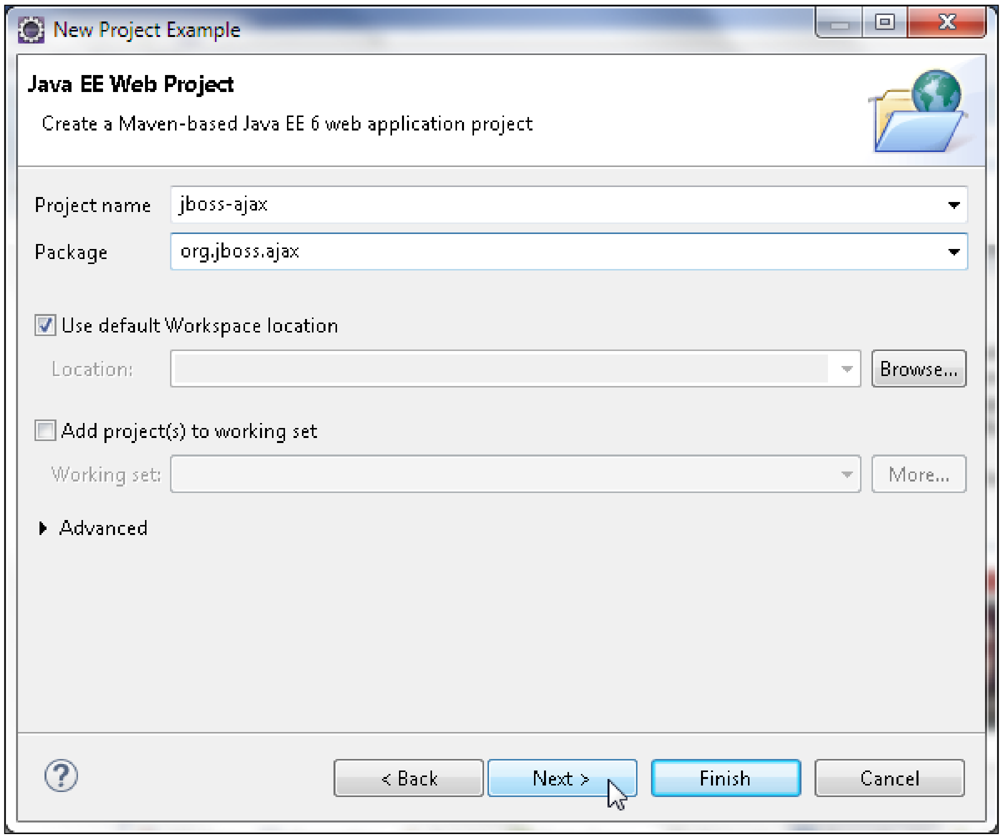
☑ Create a blank project

Target Runtime: WildFly 8.x Runtime

Requirements

Type	Description	Foun...
server/runti...	This project example requires JBoss Enterprise Applic...	☐
plugin	This project example requires m2e >= 1.0.	☑
plugin	This project example requires m2e-wtp >= 0.16.0.	☑
plugin	This project example requires JBoss Maven Tools.	☑

Specify **Project name** (`jboss-ajax`), **Package** (`org.jboss.ajax`), and click on **Next**, as shown in the following screenshot:



Java EE Web Project

Create a Maven-based Java EE 6 web application project

Project name: jboss-ajax

Package: org.jboss.ajax

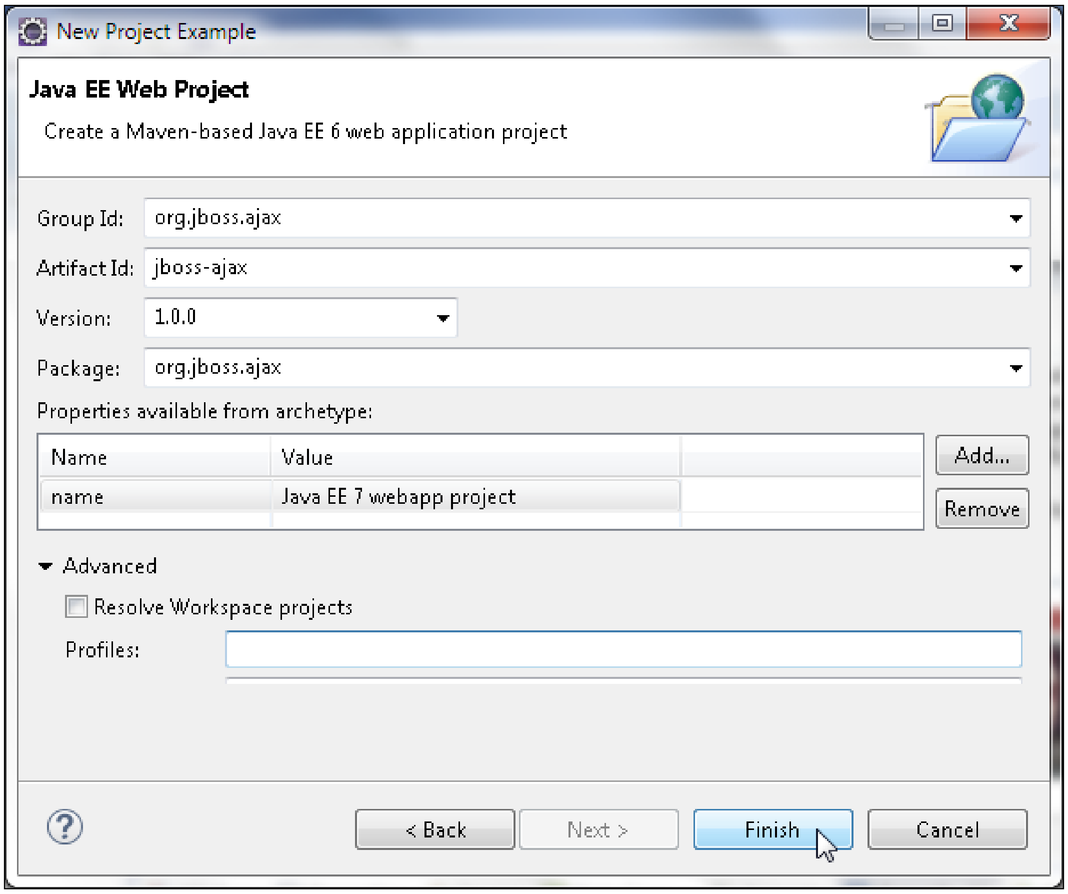
[✓] Use default Workspace location

Location:

[ ] Add project(s) to working set

Working set:

Specify **Group Id** (`org.jboss.ajax`), **Artifact Id** (`jboss-ajax`), **Version** (`1.0.0`), and **Package** (`org.jboss.ajax`), as shown in the following screenshot. After this, click on **Finish**.



Java EE Web Project

Create a Maven-based Java EE 6 web application project

Group Id: org.jboss.ajax

Artifact Id: jboss-ajax

Version: 1.0.0

Package: org.jboss.ajax

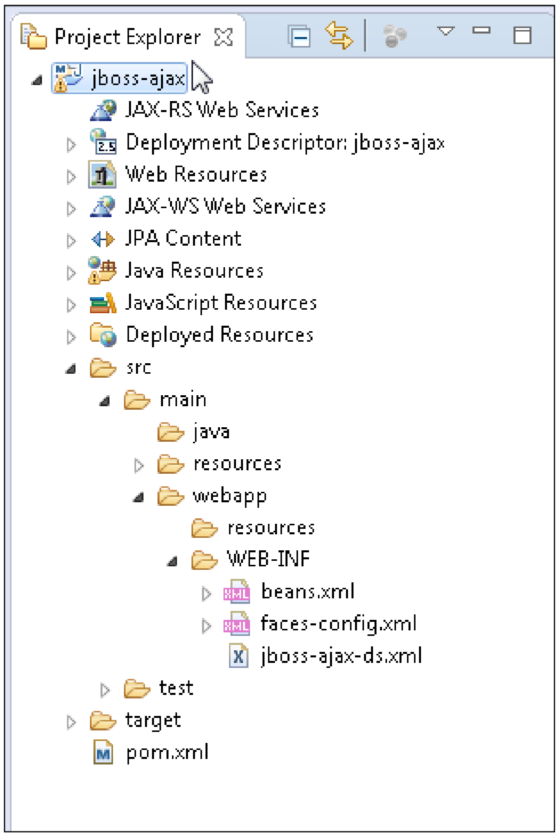
Properties available from archetype:

Name	Value
name	Java EE 7 webapp project

[ ] Resolve Workspace projects

Profiles:

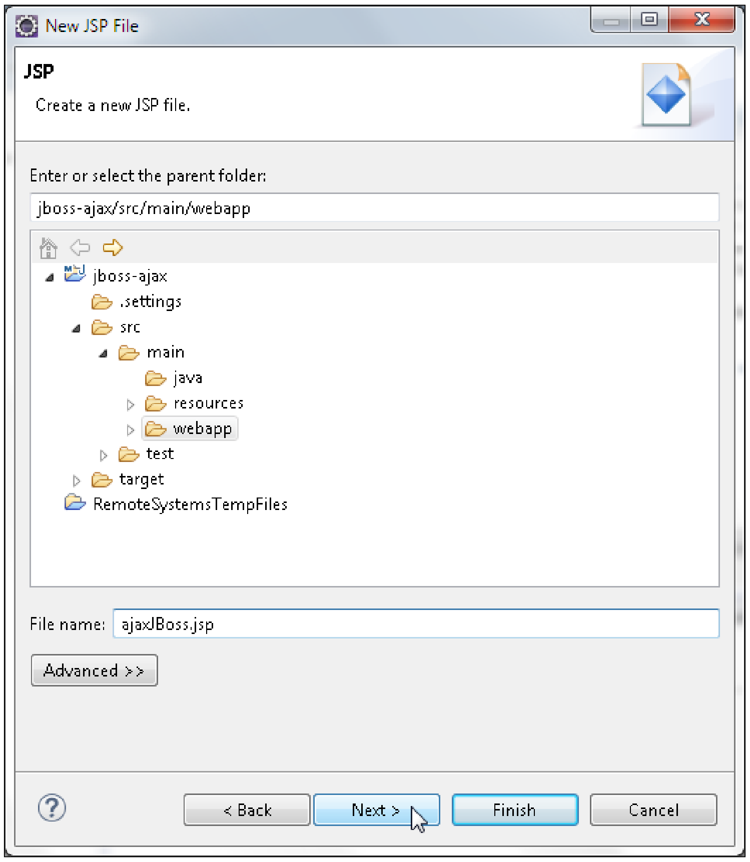
The `jboss-ajax` Java EE web project gets created, as shown in **Project Explorer** in the following screenshot. Delete the `//jboss-ajax/src/main/resources/META-INF/persistence.xml` configuration file as it is not used in the Ajax application:



Creating a user interface

An Ajax request is initiated in a browser from a web page. In this section, we will create the user interface for the Ajax application. To initiate an Ajax request, JavaScript is required, for which we will create a JSP page. We have used JSP, but another user interface technology, such as JSF, can be used instead. Select **File** | **New** | **Other**, and in **New**, select **Web** | **JSP File** and click on **Next**.

In the **New JSP File** wizard, select the `webapp` folder, specify **File name** (`ajaxJBoss.jsp`), and click on **Next,** as shown in the following screenshot:



In **Select JSP Template**, select the **New JSP File (html)** template, and click on **Finish**. The `ajaxJBoss.jsp` file gets added to the `webapp` folder. In `ajaxJBoss.jsp`, add an HTML form to create a catalog entry. The input form consists of input fields for Catalog ID, journal, publisher, edition, title, and author. The **Catalog ID** field requires a unique field value. In a form without Ajax, we would specify a Catalog ID value and the other field values and submit the form with a **Submit** button. If the Catalog ID is unique, a new catalog entry would get created, but if the Catalog ID already exists in the database, an error message would get displayed and the form would be required to be refilled and resubmitted. With Ajax, the Catalog ID value can be validated as the value is specified, thus preempting the need to resubmit the form. In the Ajax application, the input Catalog ID value is sent to the server as the value is specified using an Ajax request and an HTTP servlet immediately returns an XML message about the validity of the input data.

To send `XMLHttpRequest` and receive a response, we will use the following procedure:

1. Invoke a JavaScript function from an HTML event, such as `onkeyup`.
2. Create an `XMLHttpRequest` object in the JavaScript function.
3. Open an `XMLHttpRequest` request.
4. Register a callback event handler, which gets invoked when the request is complete, with the `XMLHttpRequest` object.
5. Send an `XMLHttpRequest` request to the server asynchronously.
6. Process the request on the server; for the server, the asynchronous request is just like any other HTTP request.
7. Send an XML message response back to the browser.
8. Receive the XML response and display a message on the web page without reloading the web page.

In `ajaxJBoss.jsp`, add an HTML form with input fields for a catalog entry. Set the action for the form as `AjaxFormServlet`, which is mapped to invoke a servlet, as discussed later. Add `<table/>` within `<form/>` and add a `<input/>` field for a Catalog ID. Add a **Submit** button to submit the form to the server using the HTTP `POST` method. In the `<input/>` field, set the `onkeyup` event handler to a `validateCatalogId()` JavaScript function, which we will add to `ajaxJBoss.jsp`. Include `<div/>` in the table row for the Catalog ID input field; `<div/>` will be used to display a message about the validity of the Catalog ID. Here's the code that encapsulates the discussion in this paragraph:

```
<table>
  <tr>
    <td>Catalog Id:</td><td><input type="text" size="20"
id="catalogId" name="catalogId" onkeyup="validateCatalogId()">
    </td>
    <td>
<div id="validationMessage"></div>
    </td>
  </tr>
…
</table>
```

Similarly, add the other input fields for a catalog. Each time the `onkeyup` event is generated, the `validateCatalogId` function gets invoked.

Next, create a `validateCatalogId` JavaScript function. In the `validateCatalogId()` JavaScript function, create a new `XMLHttpRequest` object. If a browser supports the `XMLHttpRequest` object as an ActiveX object (as in IE 6), the procedure to create an `XMLHttpRequest` object is different than the procedure if the `XMLHttpRequest` object is a native object; a `window` object property (as in IE 7 and later, and other browsers). Create an `init()` function within the `validateCatalogId` function and create an `XMLHttpRequest` object for both types of browsers (those supporting / not supporting `XMLHttpRequest` as a native object). The following is the code for the discussion in this paragraph:

```
function validateCatalogId(){
  var xmlHttpRequest=init();
  function init(){
    if (window.XMLHttpRequest) {
      return new XMLHttpRequest();
    } else if (window.ActiveXObject) {
      return new ActiveXObject("Microsoft.XMLHTTP");
    }
  }
}
```

Next, open `XMLHttpRequest` using the `open()` method, `url [, async = true [, user = null [, password = null]]])`. Set the HTTP method to `GET` for the browser to receive a response from the server. The server URL to which `XMLHttpRequest` is to be sent consists of a servlet mapping to invoke a servlet to process the request and the `CatalogId` request parameter. In the example application, we will invoke `AjaxFormServlet`, which is mapped to `/AjaxFormServlet` in `web.xml`. Encode the request parameter `CatalogId` using the `encodeURIComponent(string)` method, which encodes the `CatalogId` value to UTF-8 (`https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/encodeURIComponent`). By default, the user and password are set an empty string. The following code shows this action performed on the example application:

```
varcatalogId=document.getElementById("catalogId");
xmlHttpRequest.open("GET", "AjaxFormServlet?catalogId="+
encodeURIComponent(catalogId.value), true);
```

We need to know when a request has been completed so that we can process the response. Register a callback event handler, `processRequest`, with the `XMLHttpRequest` object using the `onreadystatechange` property. The JavaScript callback function `processRequest`, which we will add later, gets invoked whenever the value of the `readyState` property changes, as shown in the following line of code:

```
xmlHttpRequest.onreadystatechange=processRequest;
```

Send an Ajax request using the `send()` method (as the HTTP method is `GET`, data sent with the `send` method is set to `null`), as shown in the following line of code:

```
xmlHttpRequest.send(null);
```

With an asynchronous request, the `send()` method returns immediately. The `processRequest` function gets invoked each time the value of the `readyState` property changes. In the `processRequest` function, retrieve the `readyState` property value. When the request has loaded completely corresponding to the `readyState` property value `4` and the HTTP status `OK`, invoke the `processResponse` JavaScript function to process the response from the server:

```
function processRequest(){
  if(xmlHttpRequest.readyState==4){
    if(xmlHttpRequest.status==200){
      processResponse();
    }
  }
}
```

The `XMLHttpRequest` request invokes `AjaxFormServlet`. The servlet processes the request and returns a response as an XML message of the following format:

```
<catalog>
  <valid></valid>
  <journal></journal>
  <publisher></publisher>
  <edition></edition>
  <title></title>
  <author></author>
</catalog>
```

We will discuss the server-side processing of the request in the *Creating a Servlet* section. The `XMLHttpRequest` request's response from the server is processed, and if the instructions indicate that the `CatalogId` input is valid, a message **Catalog Id is Valid** is displayed. `XMLHttpRequest` will be sent to the server and a response is received with each modification in the `catalogId` input field.

Next, add the JavaScript function to process the response `processResponse()`. In the `processRequest()` JavaScript function, when the HTTP request has loaded completely, which corresponds to the `readyState` property value `4` and the HTTP status `OK`, which in turn corresponds to the status property value `200`, the `processResponse()` JavaScript function gets invoked. In the `processResponse()` function, obtain the value of the `responseXML` property, which contains the XML string returned from the server. This is shown in the following line of code:

```
Var xmlMessage=xmlHttpRequest.responseXML;
```

The `responseXML` property contains the `<valid/>` element, which indicates the validity of the `CatalogId` value specified in the input form. Obtain the value of the `<valid/>` element using the `getElementsByTagName(string)` method. This is shown in the following lines of code:

```
var
valid=xmlMessage.getElementsByTagName("valid")[0].firstChild.
nodeValue;
```

If the `<valid/>` element value is `true`, set the `validationMessage div` tag to `Catalog Id is Valid`, and enable the submit button in the input form. Also, set the value of the form fields to an empty string so that new input values can be specified:

```
if(valid=="true"){
  varvalidationMessage=document.getElementById
    ("validationMessage");
  validationMessage.innerHTML = "Catalog Id is Valid";
  document.getElementById("submitForm").disabled = false;
......
}
```

If the Catalog ID value is valid, a new catalog entry can be created by adding values for the different values of a catalog entry. Submit the form with the **Submit** button. If the `<valid/>` element value is `false`, set the `validationMessage` div tag in the `CatalogID` field row to `Catalog Id is not Valid`, and disable the **Submit** button. Set the values of other input fields to the values returned in the XML message from the server. For example, the `journal` field value is set as follows, setting the values of the other fields as an example of autocompletion with Ajax:

```
if(valid=="false"){

  var validationMessage=document.getElementById("validationMessage");
  validationMessage.innerHTML = "Catalog Id is not Valid";
  document.getElementById("submitForm").disabled = true;

  var journal=xmlMessage.getElementsByTagName
    ("journal")[0].firstChild.nodeValue;
...
  var journalElement=document.getElementById("journal");
  journalElement.value = journal;

}
```

The `ajaxJBoss.jsp` file is listed as follows:

```
<html>
  <head>
    <script type="text/javascript">
      function validateCatalogId(){
        var xmlHttpRequest=init();
      function init(){

      if (window.XMLHttpRequest) {
      return new XMLHttpRequest();
           } else if (window.ActiveXObject) {
      return new ActiveXObject("Microsoft.XMLHTTP");
           }

      }
      var catalogId=document.getElementById("catalogId");
      xmlHttpRequest.open("GET", "AjaxFormServlet?catalogId="+
encodeURIComponent(catalogId.value), true);
      xmlHttpRequest.onreadystatechange=processRequest;
      xmlHttpRequest.send(null);

      function processRequest(){
      if(xmlHttpRequest.readyState==4){
      if(xmlHttpRequest.status==200){
      processResponse();
        }
      }
      }
      function processResponse(){
      var xmlMessage=xmlHttpRequest.responseXML;
      var valid=xmlMessage.getElementsByTagName("valid")[0].
firstChild.nodeValue;
      if(valid=="true"){
      var validationMessage=document.getElementById("validationMessa
ge");
      validationMessage.innerHTML = "Catalog Id is Valid";
      document.getElementById("submitForm").disabled = false;
      var journalElement=document.getElementById("journal");
      journalElement.value = "";
      var publisherElement=document.getElementById("publisher");
      publisherElement.value = "";
      var editionElement=document.getElementById("edition");
      editionElement.value = "";
      var titleElement=document.getElementById("title");
```

```
        titleElement.value = "";
        var authorElement=document.getElementById("author");
        authorElement.value = "";
        }
        if(valid=="false"){
        var validationMessage=document.getElementById("validationMessa
ge");
        validationMessage.innerHTML = "Catalog Id is not Valid";
        document.getElementById("submitForm").disabled = true;

        var journal=xmlMessage.getElementsByTagName("journal")[0].
firstChild.nodeValue;
        var publisher=xmlMessage.getElementsByTagName("publisher")[0].
firstChild.nodeValue;
        var edition=xmlMessage.getElementsByTagName("edition")[0].
firstChild.nodeValue;
        var title=xmlMessage.getElementsByTagName("title")[0].
firstChild.nodeValue;
        var author=xmlMessage.getElementsByTagName("author")[0].
firstChild.nodeValue;

        var journalElement=document.getElementById("journal");
        journalElement.value = journal;

        var publisherElement=document.getElementById("publisher");
        publisherElement.value = publisher;

        var editionElement=document.getElementById("edition");
        editionElement.value = edition;

        var titleElement=document.getElementById("title");
        titleElement.value = title;

        var authorElement=document.getElementById("author");
        authorElement.value = author;
        }
        }
        }

      </script>
    </head>
    <body>
      <h1>Form for Catalog Entry</h1>
      <form name="AjaxFormServlet" action="AjaxFormServlet"
method="post">
```

```
<table>
  <tr>
    <td>Catalog Id:</td>
    <td><input    type="text"
      size="20"
      id="catalogId"
      name="catalogId"
      onkeyup="validateCatalogId()"></td>
    <td>
      <div id="validationMessage"></div>
    </td>
  </tr>
  <tr>
    <td>Journal:</td>
    <td><input    type="text"
      size="20"
      id="journal"
      name="journal"></td>
  </tr>
  <tr>
    <td>Publisher:</td>
    <td><input    type="text"
      size="20"
      id="publisher"
      name="publisher"></td>
  </tr>
  <tr>
    <td>Edition:</td>
    <td><input    type="text"
      size="20"
      id="edition"
      name="edition"></td>
  </tr>
  <tr>
    <td>Title:</td>
    <td><input    type="text"
      size="20"
      id="title"
      name="title"></td>
  </tr>
  <tr>
    <td>Author:</td>
    <td><input    type="text"
      size="20"
```

```
              id="author"
              name="author"></td>
          </tr>
          <tr>
            <td><input    type="submit"
              value="Create Catalog"
              id="submitForm"
              name="submitForm"></td>
          </tr>
        </table>
      </form>
    </body>
  </html>
```

Add a `catalog.jsp` JSP to output a message that a catalog entry has been created without error, as follows:

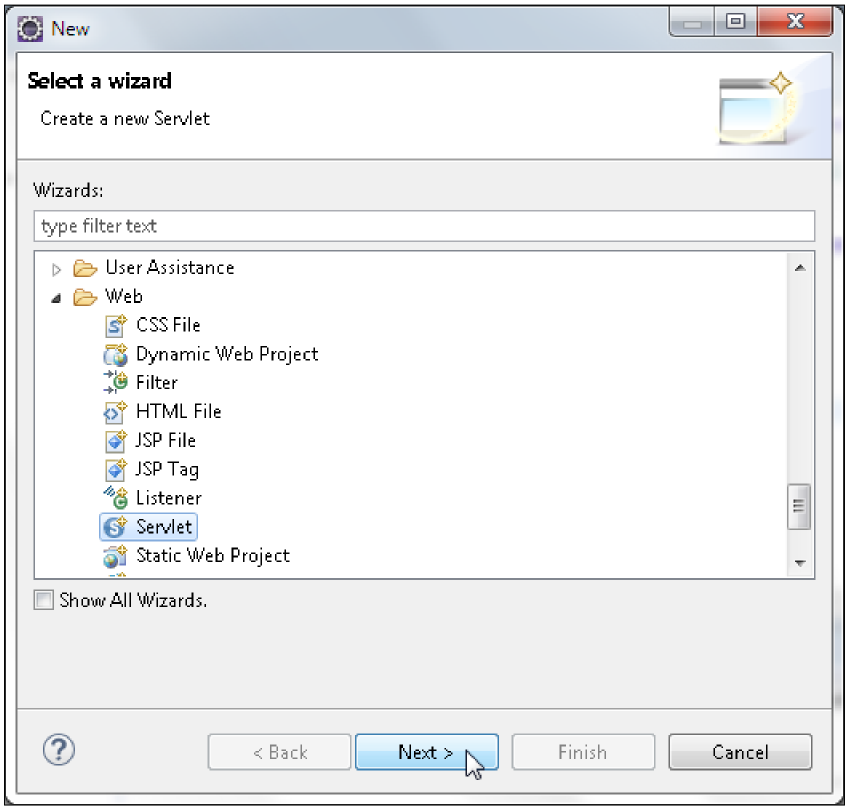
```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
  pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
    <title>Catalog entry created without error</title>
  </head>
  <body>
    Catalog entry created without error
  </body>
</html>
```

Add another `error.jsp` JSP to indicate an error message, as follows:

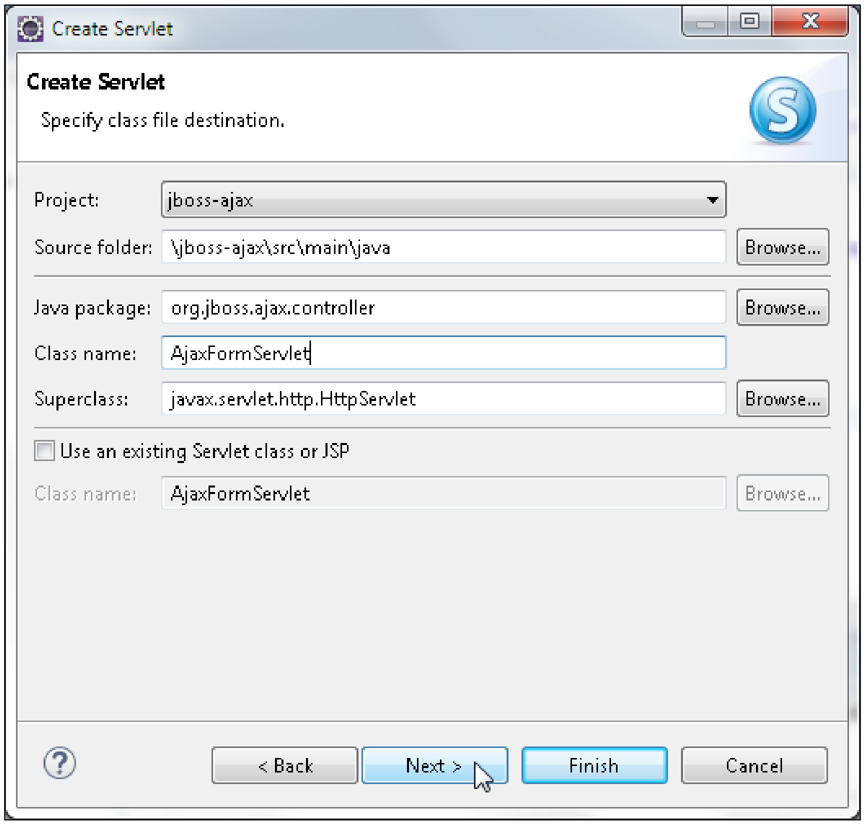
```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
  pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
    <title>Error page</title>
  </head>
  <body>
    Error Page
  </body>
</html>
```

Creating a servlet

In this section, we will create a servlet to process an Ajax request. Select **File** | **New** | **Other**, and in **New**, select **Web** | **Servlet**, which is shown as follows. Then, click on **Next**.

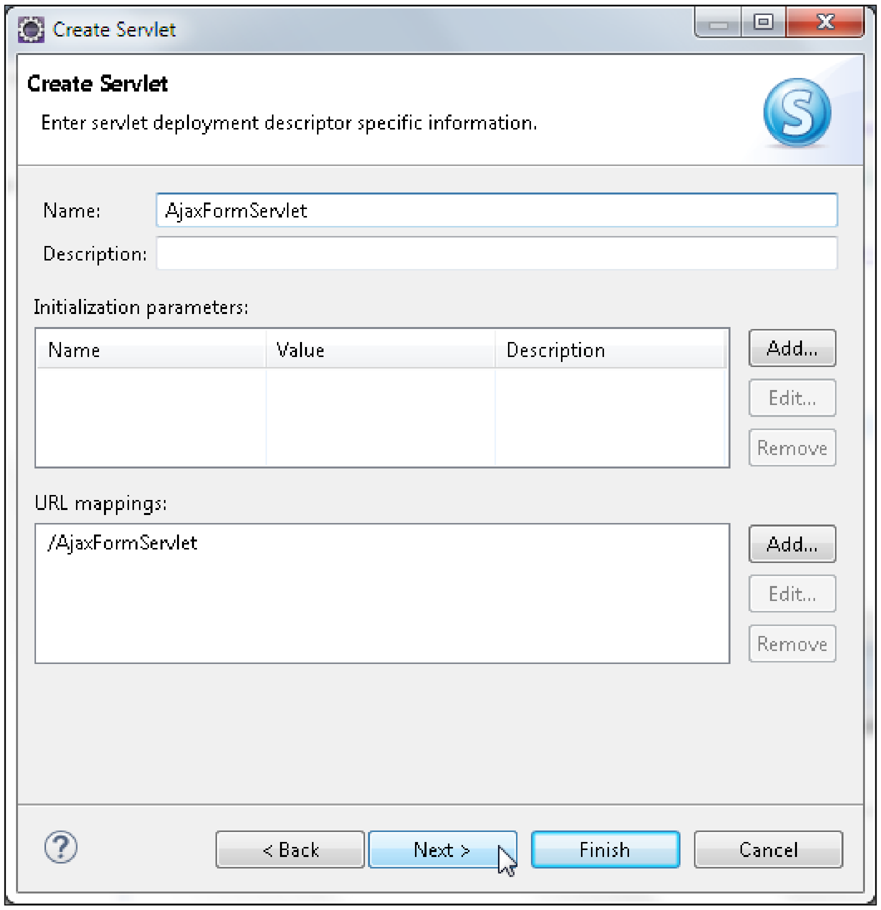


The **Create Servlet** wizard gets started. Select **Project** as `jboss-ajax`, **Source folder** as `src\main\java`, **Java package** as `org.jboss.ajax.controller`, **Class name** as `AjaxFormServlet`, and **Superclass** as `javax.servlet.http.HttpServlet`, as shown in the following screenshot. Then click on **Next**.



Project: jboss-ajax
Source folder: \jboss-ajax\src\main\java
Java package: org.jboss.ajax.controller
Class name: AjaxFormServlet
Superclass: javax.servlet.http.HttpServlet
☐ Use an existing Servlet class or JSP
Class name: AjaxFormServlet

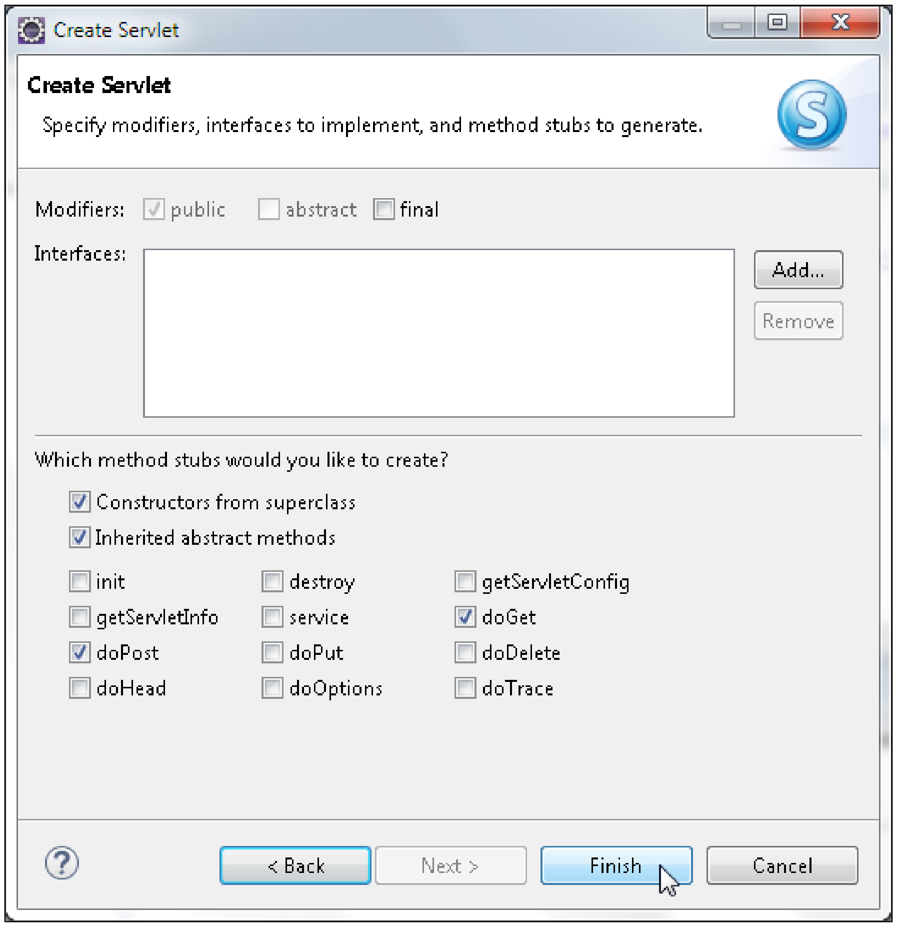
Specify URL mappings as `AjaxFormServlet` and click on **Next**, as shown in the following screenshot:



Name: AjaxFormServlet

URL mappings: /AjaxFormServlet

Select the `doGet` and `doPost` methods to create the servlet, as shown in the following screenshot. Once this is done, click on **Finish**.


Create Servlet
Create Servlet
Specify modifiers, interfaces to implement, and method stubs to generate.
Modifiers: ✓ public ☐ abstract ☐ final
Interfaces:
Add...
Remove
Which method stubs would you like to create?
✓ Constructors from superclass
✓ Inherited abstract methods
☐ init ☐ destroy ☐ getServletConfig
☐ getServletInfo ☐ service ✓ doGet
✓ doPost ☐ doPut ☐ doDelete
☐ doHead ☐ doOptions ☐ doTrace
< Back
Next >
Finish
Cancel

`AjaxFormServlet` gets created and the servlet gets configured in `web.xml`, including a URL mapping to `/AjaxFormServlet`. We included `/AjaxFormServlet` in the URL to send an `XMLHttpRequest` request to invoke `AjaxFormServlet`. The `web.xml` file is listed as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="..."
version="3.1">
  <display-name>jboss-ajax</display-name>
  <servlet>
    <description></description>
    <display-name>AjaxFormServlet</display-name>
    <servlet-name>AjaxFormServlet</servlet-name>
    <servlet-class>org.jboss.ajax.controller.AjaxFormServlet</servlet-
class>
  </servlet>
  <servlet-mapping>
    <servlet-name>AjaxFormServlet</servlet-name>
    <url-pattern>/AjaxFormServlet</url-pattern>
  </servlet-mapping>
</web-app>
```

As the `HTTP` method is `GET`, the `doGet()` method of the servlet gets invoked. In the `doGet` method, retrieve the value of the `catalogId` parameter, as shown in the following line of code:

```
String catalogId = request.getParameter("catalogId");
```

Apply the business logic on the `catalogId` value to validate the value. We have used the business logic that the value must be unique to be valid, which implies the same value must not already be in the database. Create a `DataSource` object using a JNDI lookup with an `InitialContext` object on the `java:jboss/datasources/MySQLDS` data source.

Create a `Connection` object from the `DataSource` object using the `getConnection()` method. Using the `CatalogId` value specified in the input form, create a SQL query to retrieve the data from the database. Create a `PreparedStatement` object from the `Connection` object using the `prepareStatement(String)` method. Run the SQL query using the `executeQuery()` method to obtain a `ResultSet` object. If the `ResultSet` object is empty, it implies that the `CatalogId` field value is not defined in the `Catalog` database table; the `CatalogId` field value is valid. If the `ResultSet` object contains data, it implies that the `CatalogId` value already exists in the database; the `CatalogId` field value is not valid.

Next, construct an XML string to return to the server. If `CatalogId` is not valid, construct an XML string that includes the different field values for the catalog entry as XML elements. The XML string is required to have a root element, `catalog`, for example. Include a `<valid> </valid>` element that specifies the validity of the `CatalogId` field value with a `boolean` value. If the `CatalogId` value is valid, add only the `<valid> </valid>` element to the XML string, as shown in the following code snippet (the variable `rs` represents `ResultSet`):

```
if (rs.next()) {
  out.println("<catalog>" + "<valid>false</valid>" + "<journal>" +
  rs.getString(2) + "</journal>" + "<publisher>" +
  rs.getString(3) + "</publisher>" + "<edition>" +
  rs.getString(4) + "</edition>" + "<title>" +
  rs.getString(5) + "</title>" + "<author>" +
  rs.getString(6) + "</author>" + "</catalog>");
} else {
  out.println("<valid>true</valid>");
}
```

Set the content type of `HttpServletResponse` to `text/xml` because the response to an Ajax request is in the XML format, and set the `Cache-Control` header to `no-cache` to prevent JSPs and servlets from being cached. As the Ajax response is updated with each request, caching must be disabled to prevent a cached response from being reserved, as follows:

```
response.setContentType("text/xml");
response.setHeader("Cache-Control", "no-cache");
```

If the `CatalogId` field value does not exist in the database, the input form with field values for a new catalog entry can be submitted using the `POST` method. In the `doPost` method in the servlet, create a JDBC connection to the MySQL database as in the `doGet` method, and add a catalog entry with an `INSERT` SQL statement.

The `FormServlet.java` Ajax is listed as follows:

```
package org.jboss.ajax.controller;

import java.io.*;
import java.sql.*;
import javax.naming.InitialContext;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.sql.DataSource;

public class AjaxFormServlet extends HttpServlet {
```

The `doGet` method is invoked with an asynchronous request sent using the HTTP `GET` method. Run a SQL query using Catalog Id, which is specified in the input form to generate a result set. Set headers for the `HttpServletResponse` object, and create a `PrintWriter` object from the `HttpServletResponse` object. Construct an output as an XML response, as shown in the following code:

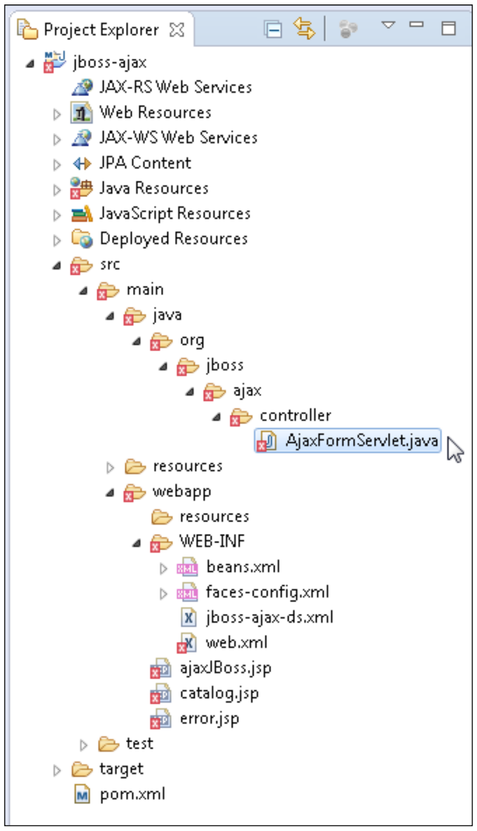
```
public void doGet(HttpServletRequest request, HttpServletResponse
response)
  throws ServletException, IOException {
    try {
      // Obtain value of Catalog Id field to be validated.
      String catalogId = request.getParameter("catalogId");

      // Obtain Connection
      InitialContext initialContext = new InitialContext();
      DataSource ds = (DataSource)initialContext.lookup("java:jboss/
datasources/MySQLDS");
      java.sql.Connection conn = ds.getConnection();

      // Obtain result set
      PreparedStatement pstmt = conn.prepareStatement("SELECT * from
CATALOG WHERE CatalogId = ?");
      pstmt.setString(1, catalogId);

      ResultSet rs = pstmt.executeQuery();

      // set headers before accessing the Writer
      response.setContentType("text/xml");
      response.setHeader("Cache-Control", "no-cache");

      PrintWriter out = response.getWriter();

      // then send the response
      // If result set is empty set valid element to true
      if (rs.next()) {
        out.println("<catalog>" + "<valid>false</valid>" + "<journal>"
        + rs.getString(2) + "</journal>" + "<publisher>"
        + rs.getString(3) + "</publisher>" + "<edition>"
        + rs.getString(4) + "</edition>" + "<title>"
        + rs.getString(5) + "</title>" + "<author>"
        + rs.getString(6) + "</author>" + "</catalog>");
      } else {
        out.println("<valid>true</valid>");
      }

      rs.close();
      stmt.close();
      conn.close();

    } catch (javax.naming.NamingException e) {System.err.println(e.
getMessage());
    } catch (SQLException e) {System.err.println(e.getMessage());
    }
  }
```

The `doPost()` method is used to create a new catalog entry. Create an `InitialContext` object. With a JNDI lookup, create a `DataSource` object. Obtain a `Connection` object from the `DataSource` object using the `getConnection()` method. Create a `Statement` object using the `createStatement()` method of the `Connection` class. `PreparedStatement` can be used instead of `Statement`. Create a SQL string from values retrieved from the input form. Run the SQL statement using the `execute()` method. If the SQL statement runs without error, redirect the response to `catalogentrycreated.jsp`. If an error is generated, redirect the response to `error.jsp`, as shown in the following code:

```
public void doPost(HttpServletRequest request, HttpServletResponse
response)
  throws ServletException, IOException {
    try {
      // Obtain Connection
      InitialContext initialContext = new InitialContext();
      DataSource ds = (DataSource) initialContext.lookup("java:jboss/
datasources/MySQLDS");
      java.sql.Connection conn = ds.getConnection();

      String catalogId = request.getParameter("catalogId");
      String journal = request.getParameter("journal");
      String publisher = request.getParameter("publisher");
      String edition = request.getParameter("edition");
      String title = request.getParameter("title");
      String author = request.getParameter("author");

      Statement stmt = conn.createStatement();
      String sql = "INSERT INTO Catalog VALUES(" + "\'" + catalogId
      + "\'" + "," + "\'" + journal + "\'" + "," + "\'"
      + publisher + "\'" + "," + "\'" + edition + "\'" + ","
      + "\'" + title + "\'" + "," + "\'" + author + "\'" + ")";

      stmt.execute(sql);

      response.sendRedirect("catalogentrycreated.jsp");

      stmt.close();
      conn.close();

    } catch (javax.naming.NamingException e) {
      response.sendRedirect("error.jsp");
    } catch (SQLException e) {
      response.sendRedirect("error.jsp");
```

```
            }
        }
    }
```

The `AjaxFormServlet` class is shown in **Package Explorer** in the following screenshot. The errors shown in the listing will be removed in the next section once the dependencies are satisfied through Maven.



Deploying the Ajax application with Maven

In this section, we will compile, package, and deploy the Ajax application to WildFly 8.1 using the Maven build tool. The information about the project and the configuration details are specified in `pom.xml` in the root directory of the Ajax application.

As we are using the MySQL database, add a dependency on the MySQL JDBC Java Connector, as follows:

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>5.1.22</version>
</dependency>
```

Add the dependency for the Servlet 3.1 API, as follows:

```
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>javax.servlet-api</artifactId>
  <version>3.1.0</version>
</dependency>
```

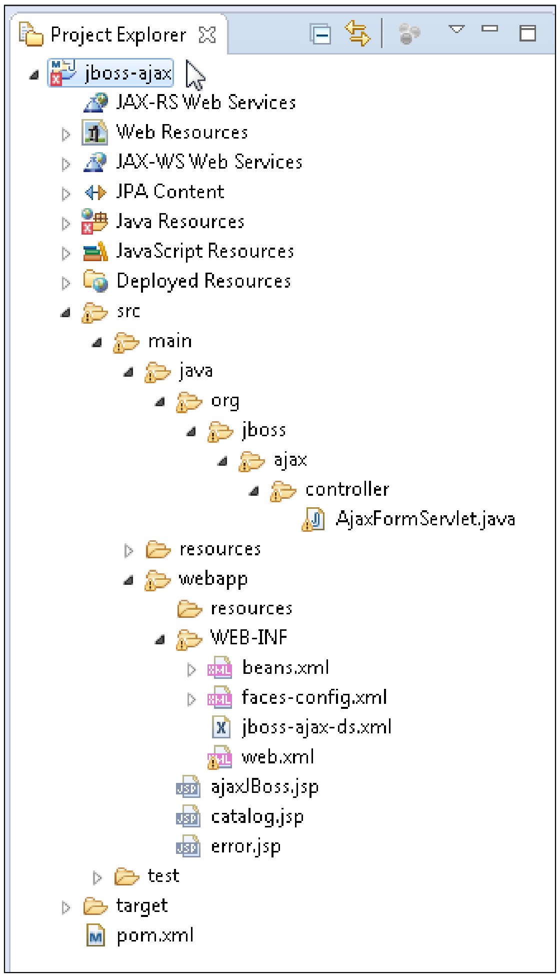
Add the Maven compiler plugin and the Maven `WAR` plugin in the `build` element. In the configuration for the Maven `WAR` plugin, specify the output directory as the `deployments` directory of WildFly 8.1. The `pom.xml` file is listed as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.
apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.jboss.ajax</groupId>
  <artifactId>jboss-ajax</artifactId>
  <version>1.0.0</version>
  <packaging>war</packaging>
  <name>WildFly Ajax</name>
  <description>A starter Java EE 7 webapp project for use on JBoss
WildFly / WildFly, generated from the jboss-javaee6-webapp archetype</
description>
  <url>http://wildfly.org</url>
  <licenses>
    <license>
      <name>Apache License, Version 2.0</name>
      <distribution>repo</distribution>
      <url>http://www.apache.org/licenses/LICENSE-2.0.html</url>
    </license>
  </licenses>
  <properties>
    <!-- Explicitly declaring the source encoding eliminates the
following message: -->
```

```
    <!-- [WARNING] Using platform encoding (UTF-8 actually) to copy 
filtered resources, i.e. build is platform dependent! -->
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <!-- JBoss dependency versions -->
    <version.wildfly.maven.plugin>1.0.2.Final</version.wildfly.maven.
plugin>
    <!-- Define the version of the JBoss BOMs we want to import to 
specify tested stacks. -->
    <version.jboss.bom>8.1.0.Final</version.jboss.bom>
    <version.arquillian.container>8.0.0.Final</version.arquillian.
container>
    <!-- other plugin versions -->
    <version.compiler.plugin>3.1</version.compiler.plugin>
    <version.surefire.plugin>2.16</version.surefire.plugin>
    <version.war.plugin>2.1.1</version.war.plugin>
    <!-- maven-compiler-plugin -->
    <maven.compiler.target>1.7</maven.compiler.target>
    <maven.compiler.source>1.7</maven.compiler.source>
  </properties>
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.wildfly.bom</groupId>
        <artifactId>jboss-javaee-7.0-with-tools</artifactId>
        <version>${version.jboss.bom}</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
      <dependency>
        <groupId>org.wildfly.bom</groupId>
        <artifactId>jboss-javaee-7.0-with-hibernate</artifactId>
        <version>${version.jboss.bom}</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
  <dependencies>
    <!-- First declare the APIs we depend on and need for compilation. 
All of them are provided by JBoss WildFly -->
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>javax.servlet-api</artifactId>
      <version>3.1.0</version>
    </dependency>
```

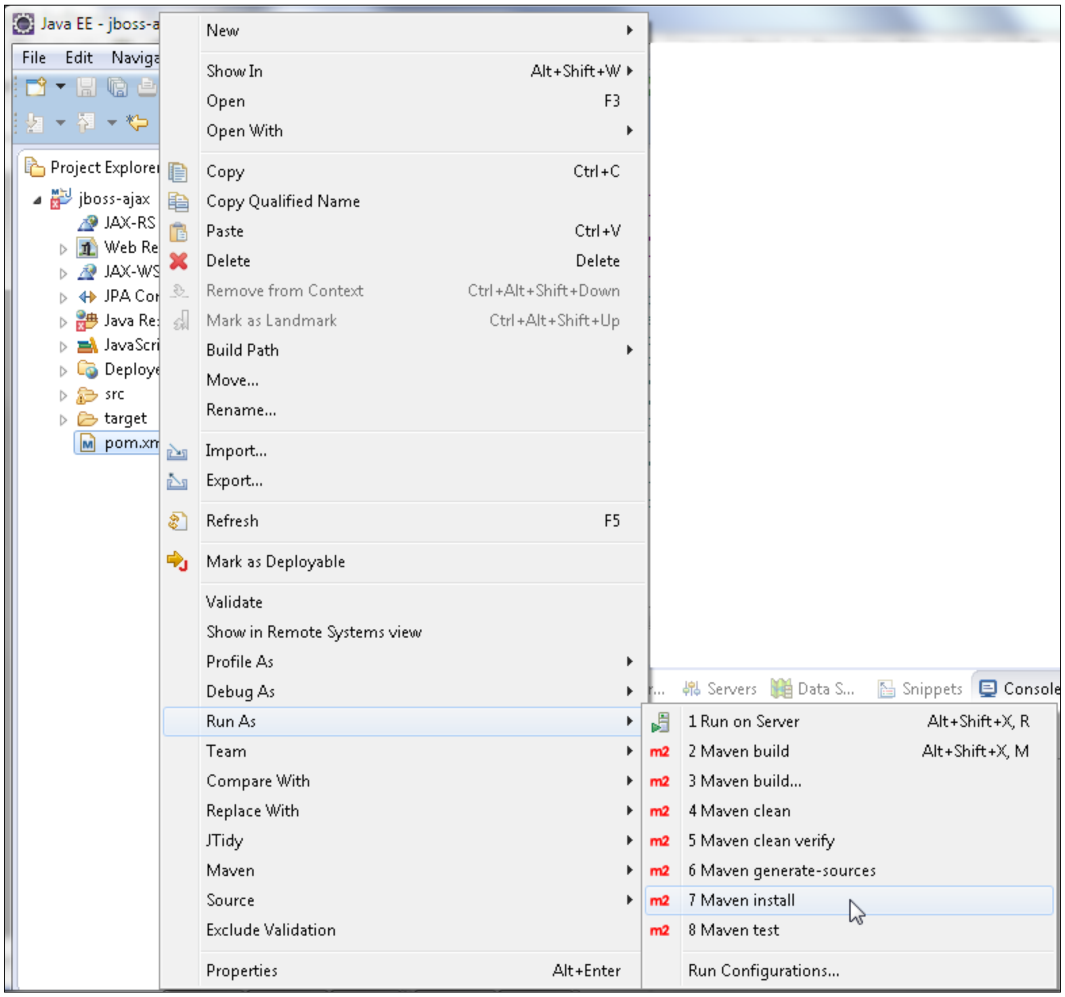
```
      <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.22</version>
      </dependency>
    </dependencies>
    <build>
      <!-- Maven will append the version to the finalName (which is the
name      given to the generated war, and hence the context root) -->
      <finalName>${project.artifactId}</finalName>
      <plugins>
        <!-- Compiler plugin enforces Java 1.6 compatibility and
activates annotation processors -->
        <plugin>
          <artifactId>maven-compiler-plugin</artifactId>
          <version>${version.compiler.plugin}</version>
          <configuration>
            <source>${maven.compiler.source}</source>
            <target>${maven.compiler.target}</target>
          </configuration>
        </plugin>
        <plugin>
          <artifactId>maven-war-plugin</artifactId>
          <version>${version.war.plugin}</version>
          <configuration>
            <outputDirectory>C:\wildfly-8.1.0.Final\standalone\
deployments</outputDirectory>
            <!-- Java EE 7 doesn't require web.xml, Maven needs to catch
up! -->
            <failOnMissingWebXml>false</failOnMissingWebXml>
          </configuration>
        </plugin>
        <!-- The WildFly plugin deploys your war to a local WildFly
container -->
        <!-- To use, run: mvn package wildfly:deploy -->
        <plugin>
          <groupId>org.wildfly.plugins</groupId>
          <artifactId>wildfly-maven-plugin</artifactId>
          <version>${version.wildfly.maven.plugin}</version>
        </plugin>
      </plugins>
    </build>
  </project>
```

After the dependencies have been added to `pom.xml`, the errors in the JSPs and the servlet get removed, as shown in the following screenshot:

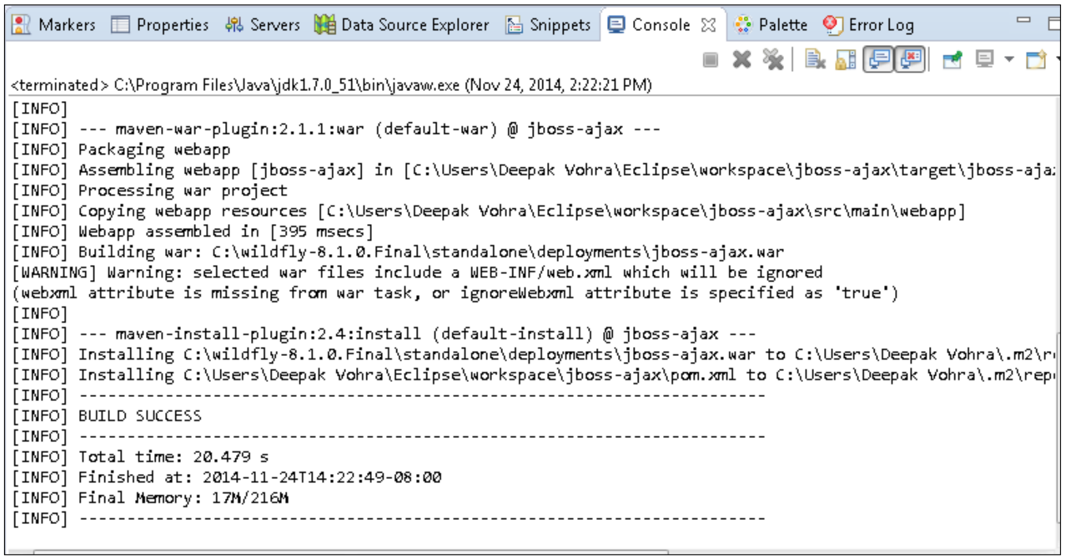


```
jboss-ajax
    JAX-RS Web Services
    Web Resources
    JAX-WS Web Services
    JPA Content
    Java Resources
    JavaScript Resources
    Deployed Resources
    src
        main
            java
                org
                    jboss
                        ajax
                            controller
                                AjaxFormServlet.java
            resources
            webapp
                resources
                WEB-INF
                    beans.xml
                    faces-config.xml
                    jboss-ajax-ds.xml
                    web.xml
                ajaxJBoss.jsp
                catalog.jsp
                error.jsp
        test
    target
    pom.xml
```

Right-click on `pom.xml` and select **Run As** | **Maven install**, as shown in the following screenshot:

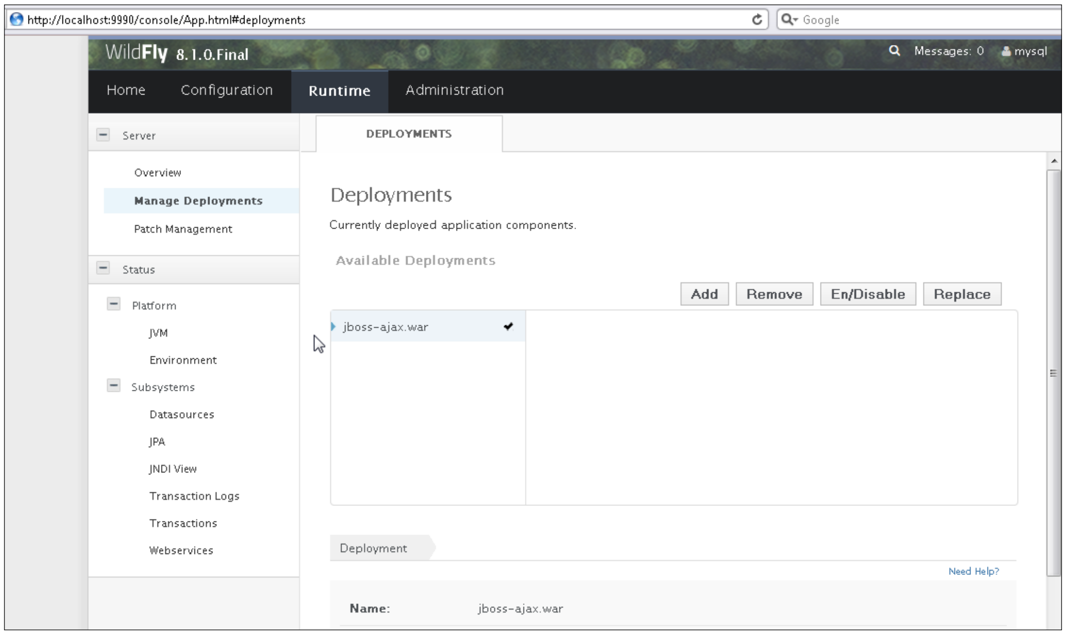


The `jboss-ajax` application gets compiled and packaged into `jboss-ajax.war`, which gets the output to the `deployments` directory. The Maven build outputs the message `BUILD SUCCESS`, as shown in the following screenshot:



```
<terminated> C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Nov 24, 2014, 2:22:21 PM)
[INFO]
[INFO] --- maven-war-plugin:2.1.1:war (default-war) @ jboss-ajax ---
[INFO] Packaging webapp
[INFO] Assembling webapp [jboss-ajax] in [C:\Users\Deepak Vohra\Eclipse\workspace\jboss-ajax\target\jboss-aja
[INFO] Processing war project
[INFO] Copying webapp resources [C:\Users\Deepak Vohra\Eclipse\workspace\jboss-ajax\src\main\webapp]
[INFO] Webapp assembled in [395 msecs]
[INFO] Building war: C:\wildfly-8.1.0.Final\standalone\deployments\jboss-ajax.war
[WARNING] Warning: selected war files include a WEB-INF/web.xml which will be ignored
(webxml attribute is missing from war task, or ignoreWebxml attribute is specified as 'true')
[INFO]
[INFO] --- maven-install-plugin:2.4:install (default-install) @ jboss-ajax ---
[INFO] Installing C:\wildfly-8.1.0.Final\standalone\deployments\jboss-ajax.war to C:\Users\Deepak Vohra\.m2\r
[INFO] Installing C:\Users\Deepak Vohra\Eclipse\workspace\jboss-ajax\pom.xml to C:\Users\Deepak Vohra\.m2\rep
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 20.479 s
[INFO] Finished at: 2014-11-24T14:22:49-08:00
[INFO] Final Memory: 17M/216M
[INFO] ------------------------------------------------------------------------
```

Start WildFly 8.1 if it is not already started. The `jboss-ajax.war` gets deployed to WildFly 8.1 and the web context root `|jboss-ajax` gets registered. The `jboss-ajax.war` gets deployed and gets listed in WildFly 8.1. **Administration Console**, as shown in the following screenshot:



Deployments

Currently deployed application components.

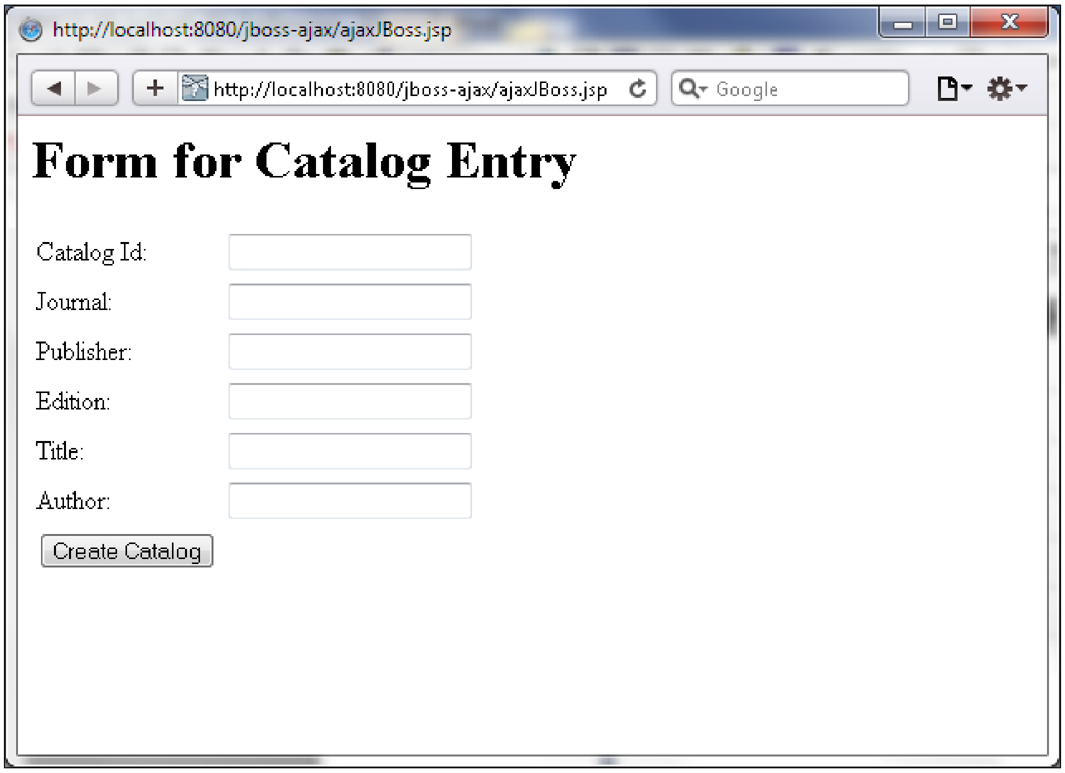
Available Deployments

jboss-ajax.war ✓

Name: jboss-ajax.war

Running the Ajax application

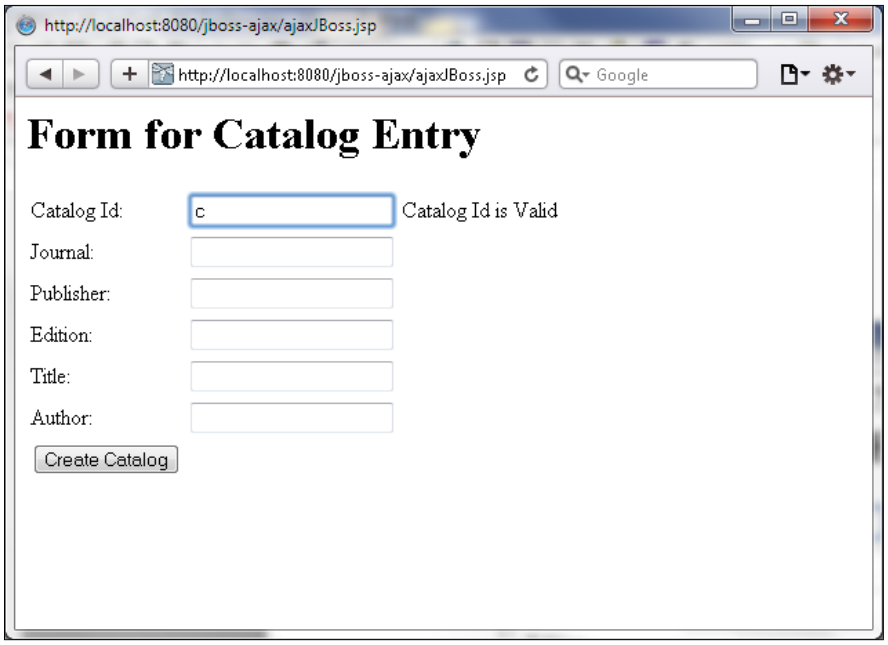
In this section, we will run the Ajax application. Run `ajaxBoss.jsp` using the URL `http://localhost:8080/jboss-ajax/ajaxJBoss.jsp`. The input form for a catalog entry gets displayed, as shown in the following screenshot:



Form for Catalog Entry

Catalog Id:	
Journal:	
Publisher:	
Edition:	
Title:	
Author:	

Start to specify a **Catalog Id** value. As an Ajax request is sent to the server, with each modification to the **Catalog Id** value, a response is returned from the server and a message gets displayed about the validity of **Catalog Id**, which is shown in the following screenshot:



http://localhost:8080/jboss-ajax/ajaxJBoss.jsp

Form for Catalog Entry

Catalog Id: c Catalog Id is Valid

Journal:

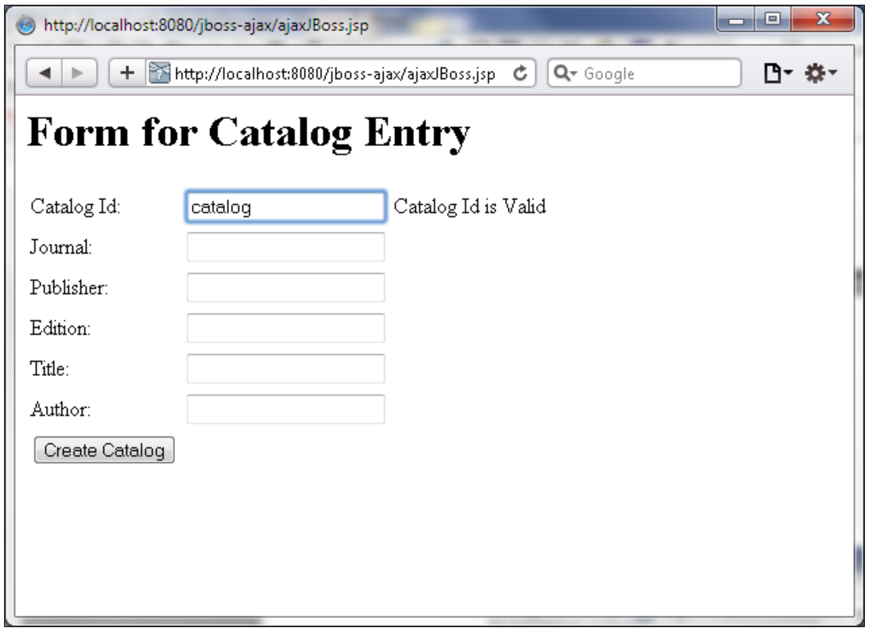
Publisher:

Edition:

Title:

Author:

The business logic used for **Catalog Id** to be valid is that the value should be unique, but another logic can be used instead. **Catalog Id** (catalog) is still valid, as shown in the following screenshot:



http://localhost:8080/jboss-ajax/ajaxJBoss.jsp

Form for Catalog Entry

Catalog Id: catalog Catalog Id is Valid

Journal:

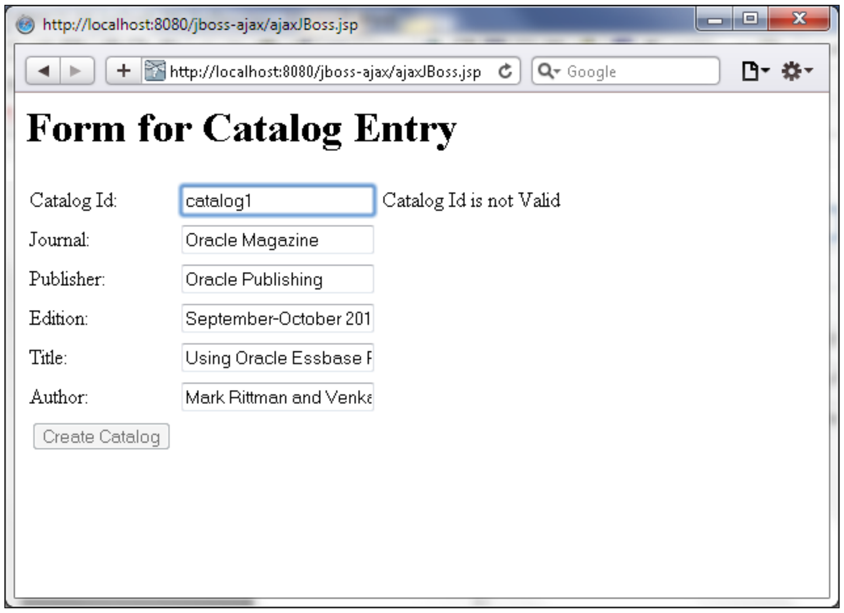
Publisher:

Edition:

Title:

Author:

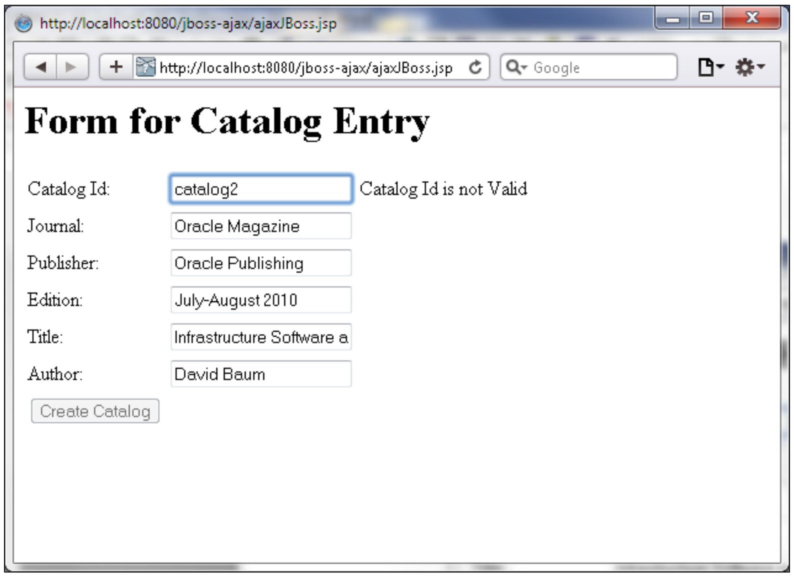
Specify a **Catalog Id** value that is already in the MySQL database, `catalog1` for example. A message **Catalog Id is not Valid** gets displayed, and the input fields get filled with the catalog entry's field values, as follows:



Form for Catalog Entry

Catalog Id:	catalog1	Catalog Id is not Valid
Journal:	Oracle Magazine	
Publisher:	Oracle Publishing	
Edition:	September-October 201	
Title:	Using Oracle Essbase F	
Author:	Mark Rittman and Venka	

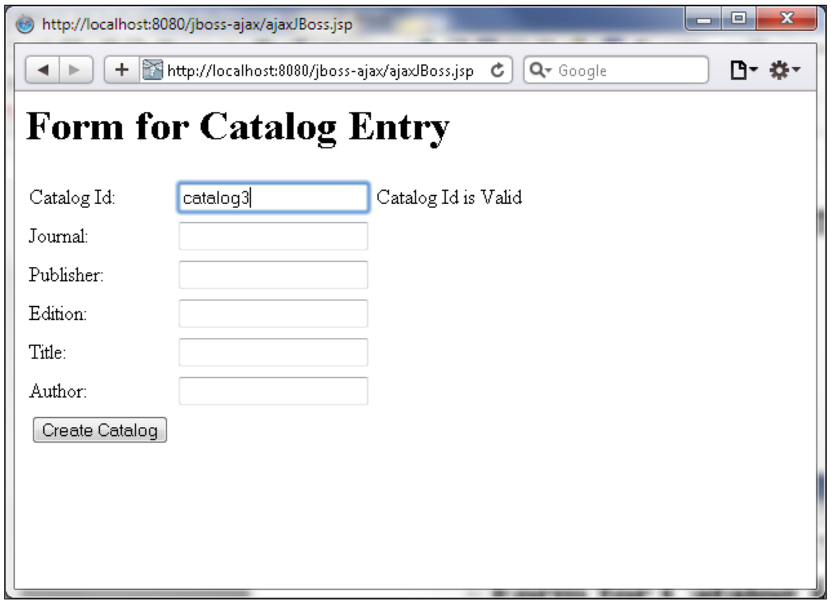
Specify `catalog2` as **Catalog Id**, which is also not valid, as shown in the following screenshot:



Form for Catalog Entry

Catalog Id:	catalog2	Catalog Id is not Valid
Journal:	Oracle Magazine	
Publisher:	Oracle Publishing	
Edition:	July-August 2010	
Title:	Infrastructure Software a	
Author:	David Baum	

Specify `catalog3` as **Catalog Id**. As the `catalog3` value is not already in the database, **Catalog Id is valid**, as shown in the following screenshot:

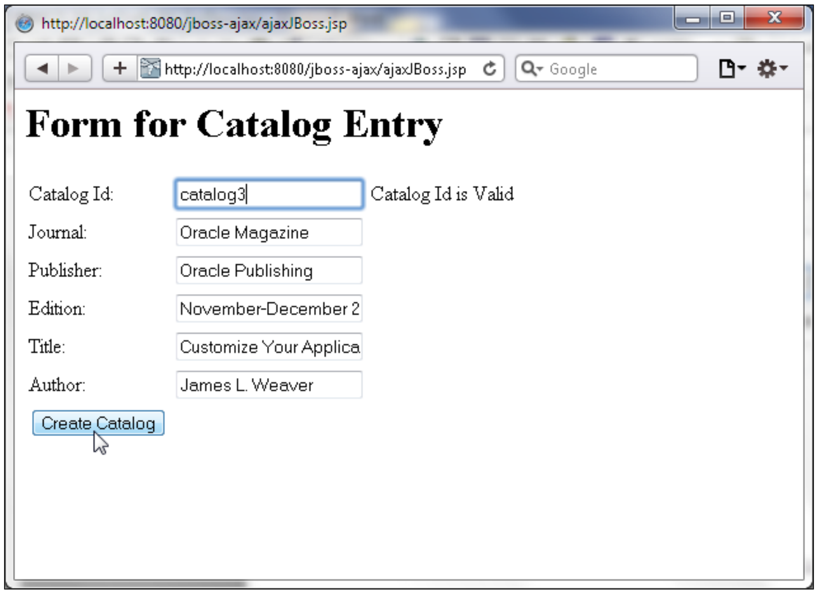


http://localhost:8080/jboss-ajax/ajaxJBoss.jsp

Form for Catalog Entry

Catalog Id:	catalog3	Catalog Id is Valid
Journal:		
Publisher:		
Edition:		
Title:		
Author:		

Add field values for a new catalog entry and click on **Submit**, as shown in the following screenshot:

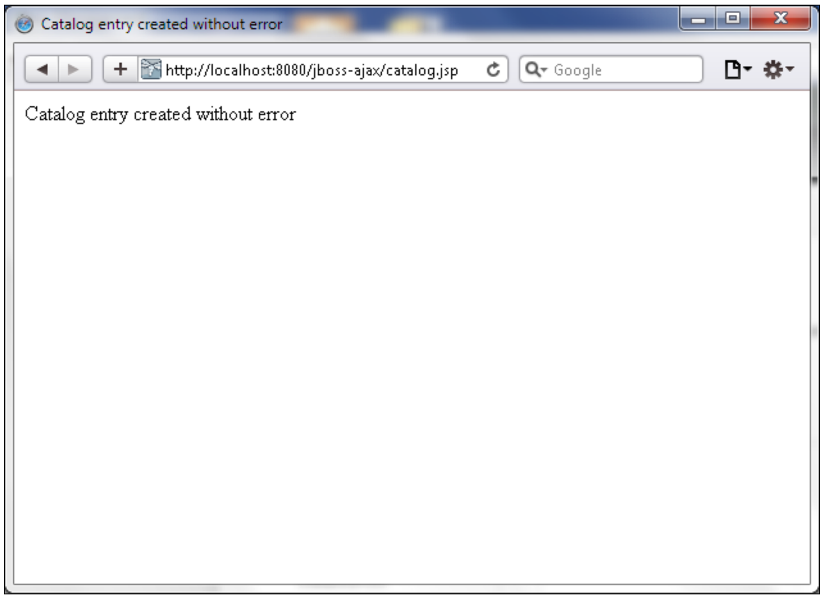


http://localhost:8080/jboss-ajax/ajaxJBoss.jsp

Form for Catalog Entry

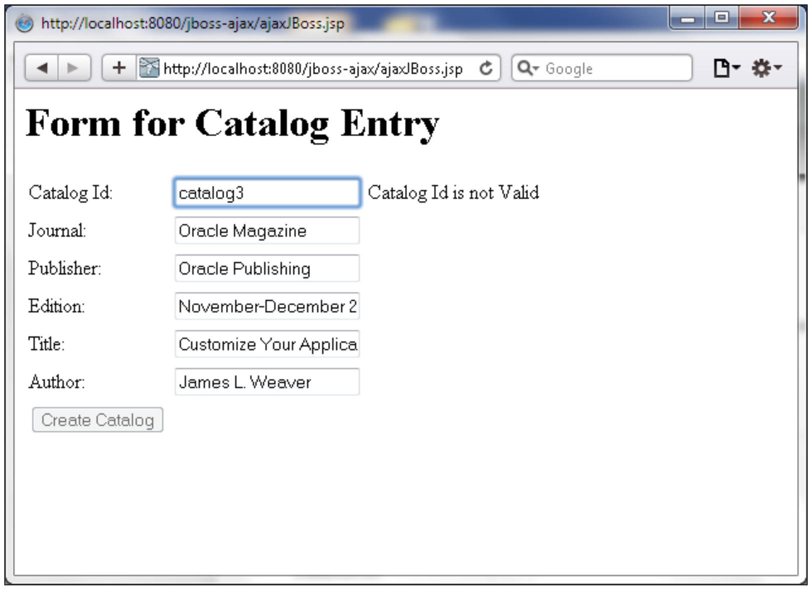
Catalog Id:	catalog3	Catalog Id is Valid
Journal:	Oracle Magazine	
Publisher:	Oracle Publishing	
Edition:	November-December 2	
Title:	Customize Your Applica	
Author:	James L. Weaver	

A new catalog entry gets created as indicated by the message shown in the following screenshot:



Catalog entry created without error

If the `catalog3` value is re-added to the **Catalog Id** field, the **Catalog Id is not Valid** message gets displayed, as shown in the following screenshot:



Form for Catalog Entry

Catalog Id:	catalog3	Catalog Id is not Valid
Journal:	Oracle Magazine	
Publisher:	Oracle Publishing	
Edition:	November-December 2	
Title:	Customize Your Applica	
Author:	James L. Weaver	

Summary

In this chapter, we learned how to develop an Ajax application in Eclipse. We compiled and packaged the application with the Maven build tool and deployed the application to WildFly 8.1. We ran the Ajax application in a browser with the MySQL database. We discussed only some of the methods and attributes of the `XMLHttpRequest` object. For complete information on `XMLHttpRequest`, refer to `http://www.w3.org/TR/XMLHttpRequest/` and `https://developer.mozilla.org/en-US/docs/DOM/XMLHttpRequest?redirectlocale=en-US&redirectslug=XMLHttpRequest`.

In the next chapter, we will discuss the GWT web framework, which provides Ajax support in UI components as a built-in feature.

5
Using GWT

Google Web Toolkit (GWT) is a toolkit for developing complex, Model-View-Presenter model, browser-based web applications. A client-side application is developed as Java using the GWT SDK and deployed as compiled JavaScript. In addition to a Java API, GWT provides widgets, which are integrated with support for Ajax. In *Chapter 4*, *Using Ajax*, we developed an Ajax application using the `XMLHttpRequest` object, which involved creating, sending, and processing an Ajax request. The GWT SDK compiler compiles Java into JavaScript, which includes the `XMLHttpRequest` optimized to run on all browsers.

This chapter consists of the following sections:

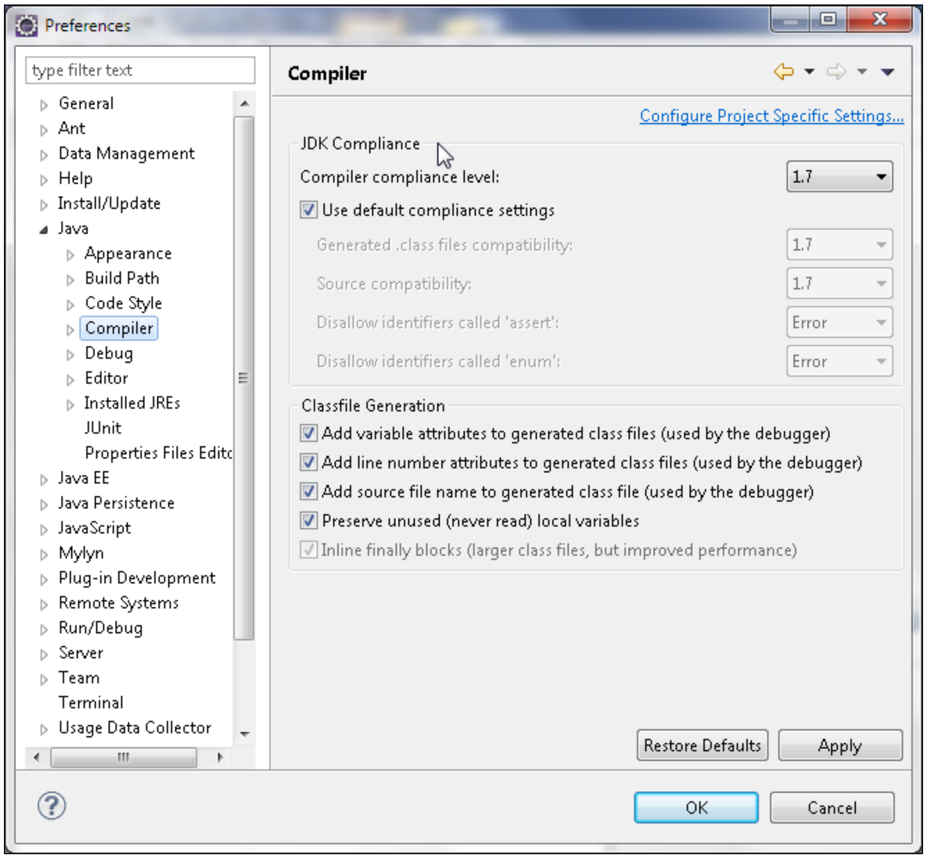
- Setting up the environment
- Running the starter project on WildFly 8.1
- Creating a GWT project
- Creating a module
- Create an entry-point class
- Creating an HTML page
- Deploying the GWT project with Maven
- Running the GWT project

Setting up the environment

We need to install the following software:

- **WildFly 8.1.0.Final**: Download `wildfly-8.1.0.Final.zip` from `http://wildfly.org/downloads/`.
- **MySQL 5.6 Database-Community Edition**: Download this edition from `http://dev.mysql.com/downloads/mysql/`. When installing MySQL, also install **Connector/J**.
- **Eclipse IDE for Java EE Developers**: Download Eclipse Luna from `https://www.eclipse.org/downloads/packages/release/Luna/SR1`.
- **JBoss Tools (Luna) 4.2.0.Final (or the latest version)**: Install this as a plugin to Eclipse from Eclipse Marketplace (`http://tools.jboss.org/downloads/installation.html`).
- **Apache Maven**: Download version 3.05 or higher from `http://maven.apache.org/download.cgi`.
- **Java 7**: Download Java 7 from `http://www.oracle.com/technetwork/java/javase/downloads/index.html?ssSourceSiteId=ocomcn`.

The GWT version must be 2.6. The reason is that the Google plugin for Eclipse supports the 2.6 version and not 2.7. Extract the GWT `ZIP` file to the `C:` drive. Set the environment variables `JAVA_HOME`, `JBOSS_HOME`, and `MAVEN_HOME`. Add `%JAVA_HOME%/bin`, `%MAVEN_HOME%/bin`, and `%JBOSS_HOME%/bin` to the `PATH` environment variable. Create a WildFly 8.1 runtime as discussed in *Chapter. 1, Getting Started With EJB 3.x*. Install JDK 7 and set the JDK compliance level to 1.7 in the Eclipse IDE. To set the JDK compliance level, select **Windows** | **Preferences**. In **Preferences**, select **Java** | **Compiler**. Select **Compiler compliance level** as `1.7`, as shown in the following screenshot:



Compiler

JDK Compliance

Compiler compliance level: 1.7

[✓] Use default compliance settings

Generated .class files compatibility: 1.7

Source compatibility: 1.7

Disallow identifiers called 'assert': Error

Disallow identifiers called 'enum': Error

Classfile Generation

[✓] Add variable attributes to generated class files (used by the debugger)

[✓] Add line number attributes to generated class files (used by the debugger)

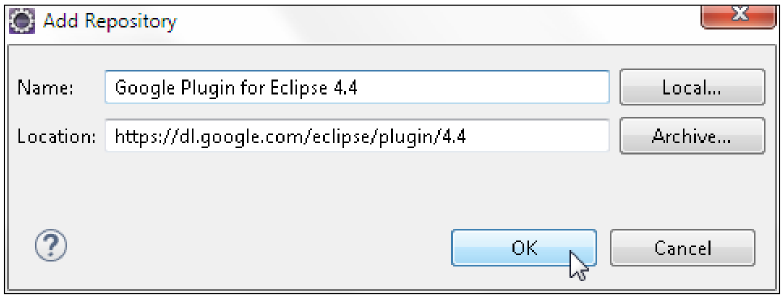
[✓] Add source file name to generated class file (used by the debugger)

[✓] Preserve unused (never read) local variables

[✓] Inline finally blocks (larger class files, but improved performance)

We also need to install Google Plugin for Eclipse by performing the following steps:

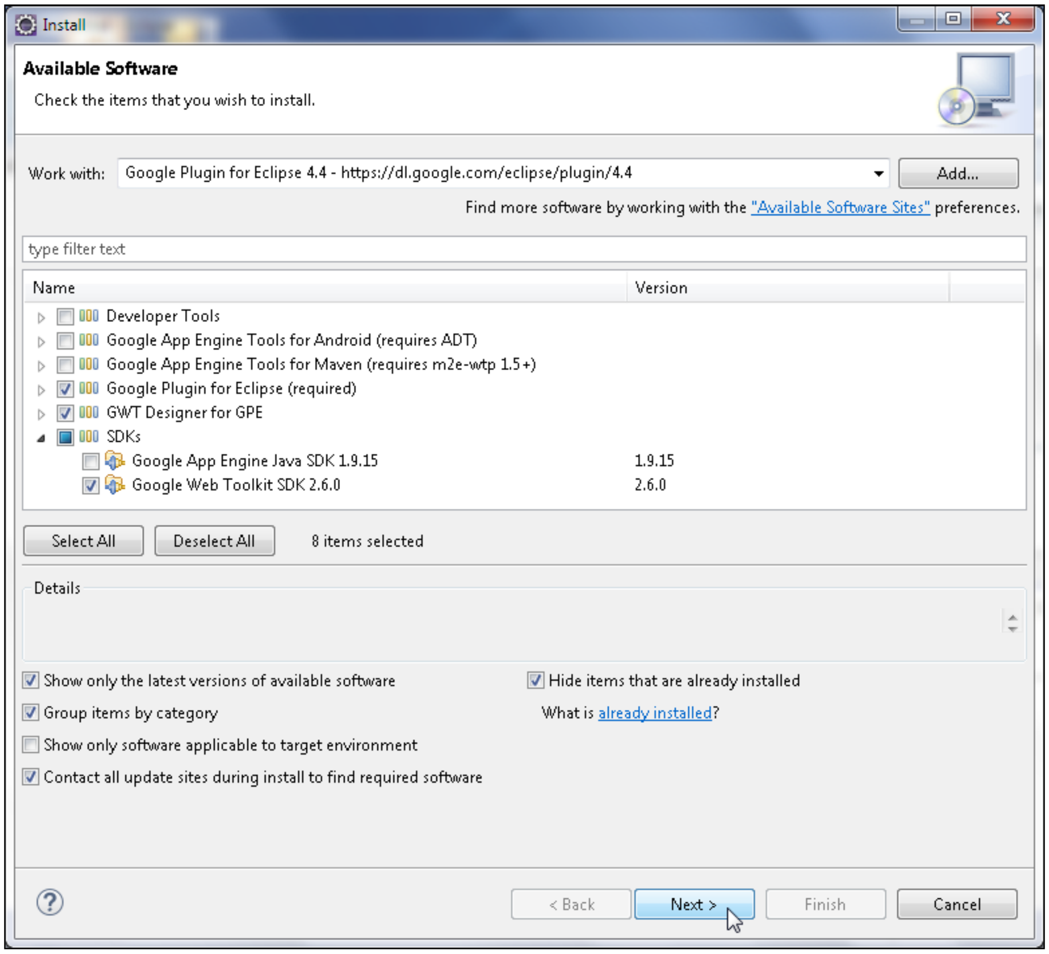
1. In Eclipse, select **Help** | **Install New Software**. In the **Install (Available Software)** wizard, click on **Add** to add a GWT repository. In **Add Repository**, specify **Name** (`Google Plugin for Eclipse 4.4`) and specify **Location** as `https://dl.google.com/eclipse/plugin/4.4`, as shown in the following screenshot. After this, click on **OK**.



Name: Google Plugin for Eclipse 4.4

Location: https://dl.google.com/eclipse/plugin/4.4

2. In **Work With**, select the Google Plugin for Eclipse 4.4 repository. From the items listed, select the **Google Plugin for Eclipse (required)**, **GWT Designer for GPE**, and **SDK | Google Web Toolkit SDK 2.6.0**, as shown in the following screenshot. Then, click on **Next**.

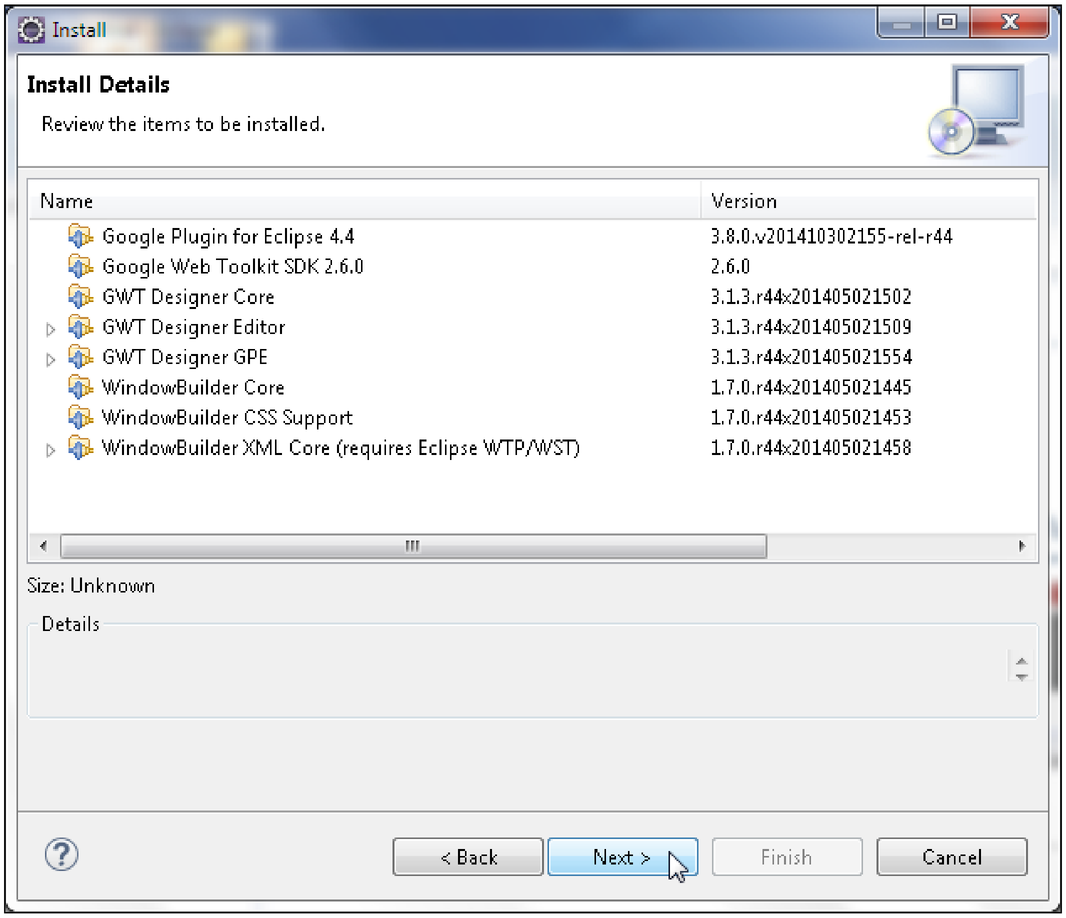

Install
Available Software
Check the items that you wish to install.
Work with: Google Plugin for Eclipse 4.4 - https://dl.google.com/eclipse/plugin/4.4
Add...
Find more software by working with the "Available Software Sites" preferences.
type filter text
Name
Version
Select All
Deselect All
8 items selected
Details
< Back
Next >
Finish
Cancel

Name	Version
☐ Developer Tools	
☐ Google App Engine Tools for Android (requires ADT)	
☐ Google App Engine Tools for Maven (requires m2e-wtp 1.5+)	
☑ Google Plugin for Eclipse (required)	
☑ GWT Designer for GPE	
■ SDKs	
☐ Google App Engine Java SDK 1.9.15	1.9.15
☑ Google Web Toolkit SDK 2.6.0	2.6.0

- ☑ Show only the latest versions of available software
- ☑ Group items by category
- ☐ Show only software applicable to target environment
- ☑ Contact all update sites during install to find required software
- ☑ Hide items that are already installed

What is already installed?

3. In **Install Details**, the different items to be installed are listed, including **Google Plugin for Eclipse 4.4** and **Google Web Toolkit SDK 2.6.0** , as shown in the following screenshot. Now, click on **Next**.



Name	Version
Google Plugin for Eclipse 4.4	3.8.0.v201410302155-rel-r44
Google Web Toolkit SDK 2.6.0	2.6.0
GWT Designer Core	3.1.3.r44x201405021502
GWT Designer Editor	3.1.3.r44x201405021509
GWT Designer GPE	3.1.3.r44x201405021554
WindowBuilder Core	1.7.0.r44x201405021445
WindowBuilder CSS Support	1.7.0.r44x201405021453
WindowBuilder XML Core (requires Eclipse WTP/WST)	1.7.0.r44x201405021458

Size: Unknown

Details

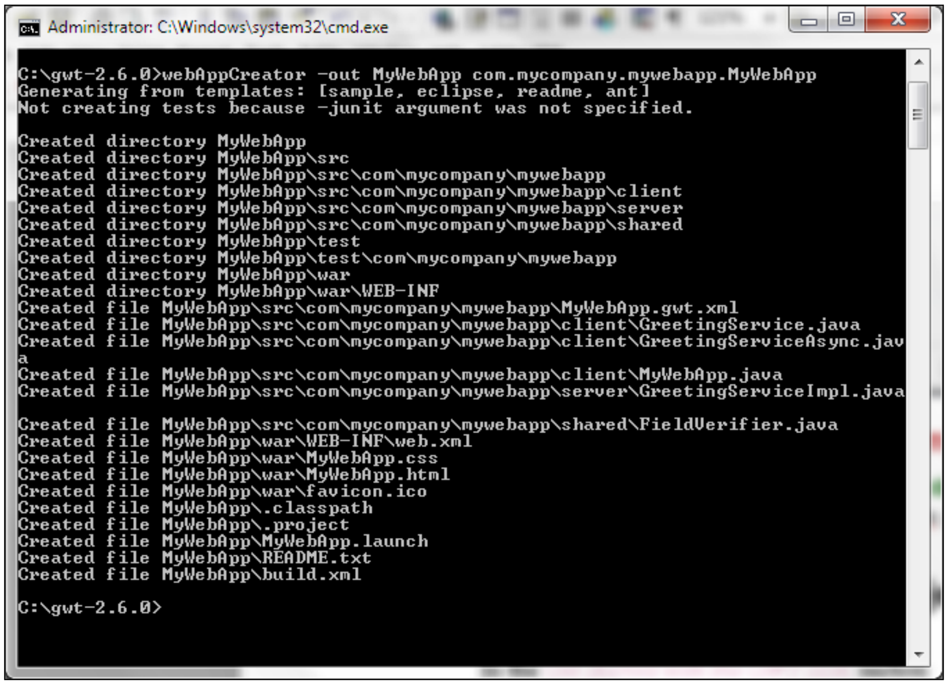
4. In **Review Licenses**, select **I accept the terms of the license agreement** and click on **Finish**. Google Plugin for Eclipse gets installed.

Running the starter project on WildFly 8.1

In the *Get Started with the GWT SDK* section (`https://developers.google.com/web-toolkit/gettingstarted`), creating a first web application is discussed. In this section, we will create the starter application and run the application on WildFly 8.1. Create the application with `webAppCreator`. Change directory (with the `cd` command) to the GWT installation directory and run the following command to create the starter web application.

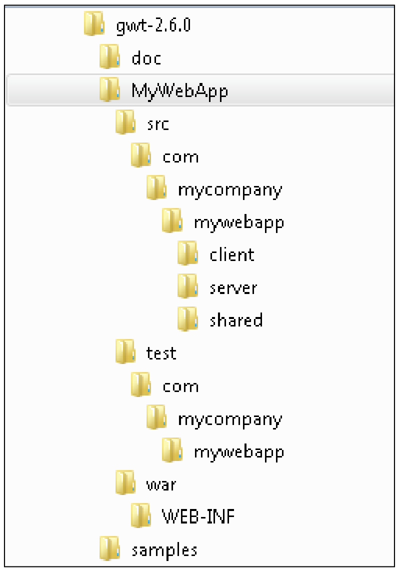
```
>cd gwt-2.6.0
>webAppCreator -out MyWebApp com.mycompany.mywebapp.MyWebApp
```

The application gets created and looks like the following screenshot:



```
C:\gwt-2.6.0>webAppCreator -out MyWebApp com.mycompany.mywebapp.MyWebApp
Generating from templates: [sample, eclipse, readme, ant]
Not creating tests because -junit argument was not specified.

Created directory MyWebApp
Created directory MyWebApp\src
Created directory MyWebApp\src\com\mycompany\mywebapp
Created directory MyWebApp\src\com\mycompany\mywebapp\client
Created directory MyWebApp\src\com\mycompany\mywebapp\server
Created directory MyWebApp\src\com\mycompany\mywebapp\shared
Created directory MyWebApp\test
Created directory MyWebApp\test\com\mycompany\mywebapp
Created directory MyWebApp\war
Created directory MyWebApp\war\WEB-INF
Created file MyWebApp\src\com\mycompany\mywebapp\MyWebApp.gwt.xml
Created file MyWebApp\src\com\mycompany\mywebapp\client\GreetingService.java
Created file MyWebApp\src\com\mycompany\mywebapp\client\GreetingServiceAsync.jav
a
Created file MyWebApp\src\com\mycompany\mywebapp\client\MyWebApp.java
Created file MyWebApp\src\com\mycompany\mywebapp\server\GreetingServiceImpl.java

Created file MyWebApp\src\com\mycompany\mywebapp\shared\FieldVerifier.java
Created file MyWebApp\war\WEB-INF\web.xml
Created file MyWebApp\war\MyWebApp.css
Created file MyWebApp\war\MyWebApp.html
Created file MyWebApp\war\favicon.ico
Created file MyWebApp\.classpath
Created file MyWebApp\.project
Created file MyWebApp\MyWebApp.launch
Created file MyWebApp\README.txt
Created file MyWebApp\build.xml

C:\gwt-2.6.0>
```

The directory structure of the the GWT 2.6 application is shown in the following screenshot:

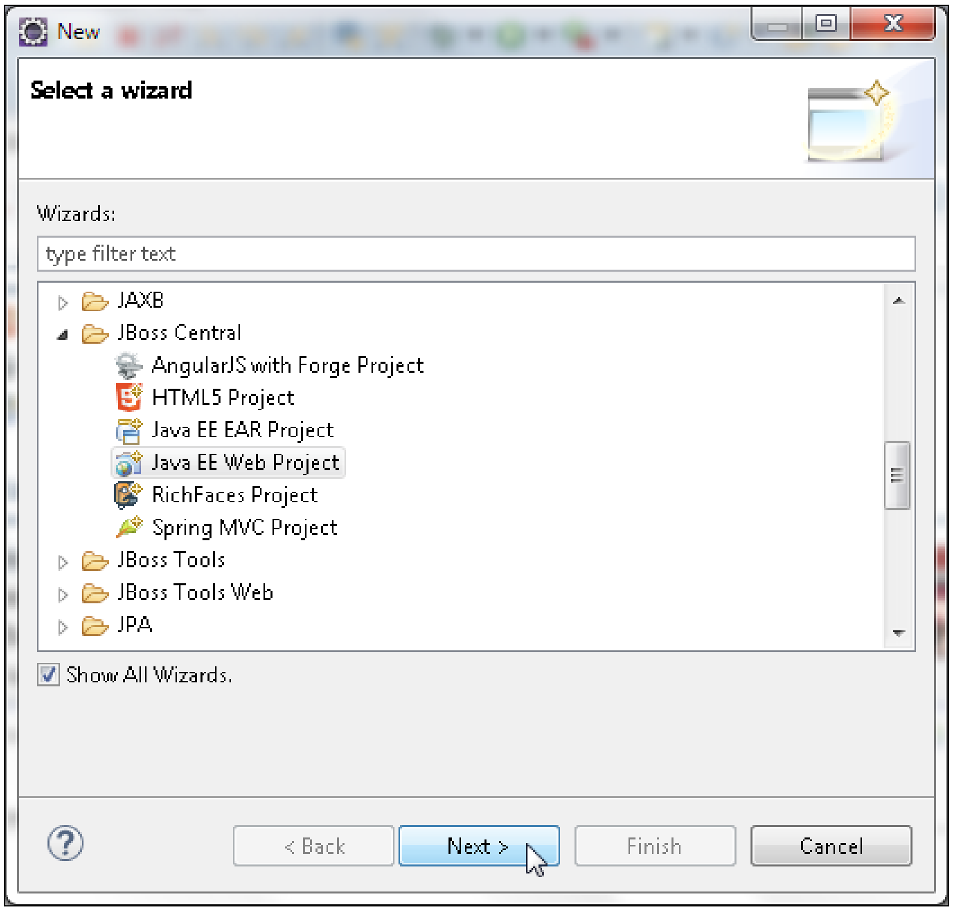


```
gwt-2.6.0
  doc
  MyWebApp
    src
      com
        mycompany
          mywebapp
            client
            server
            shared
    test
      com
        mycompany
          mywebapp
    war
      WEB-INF
  samples
```

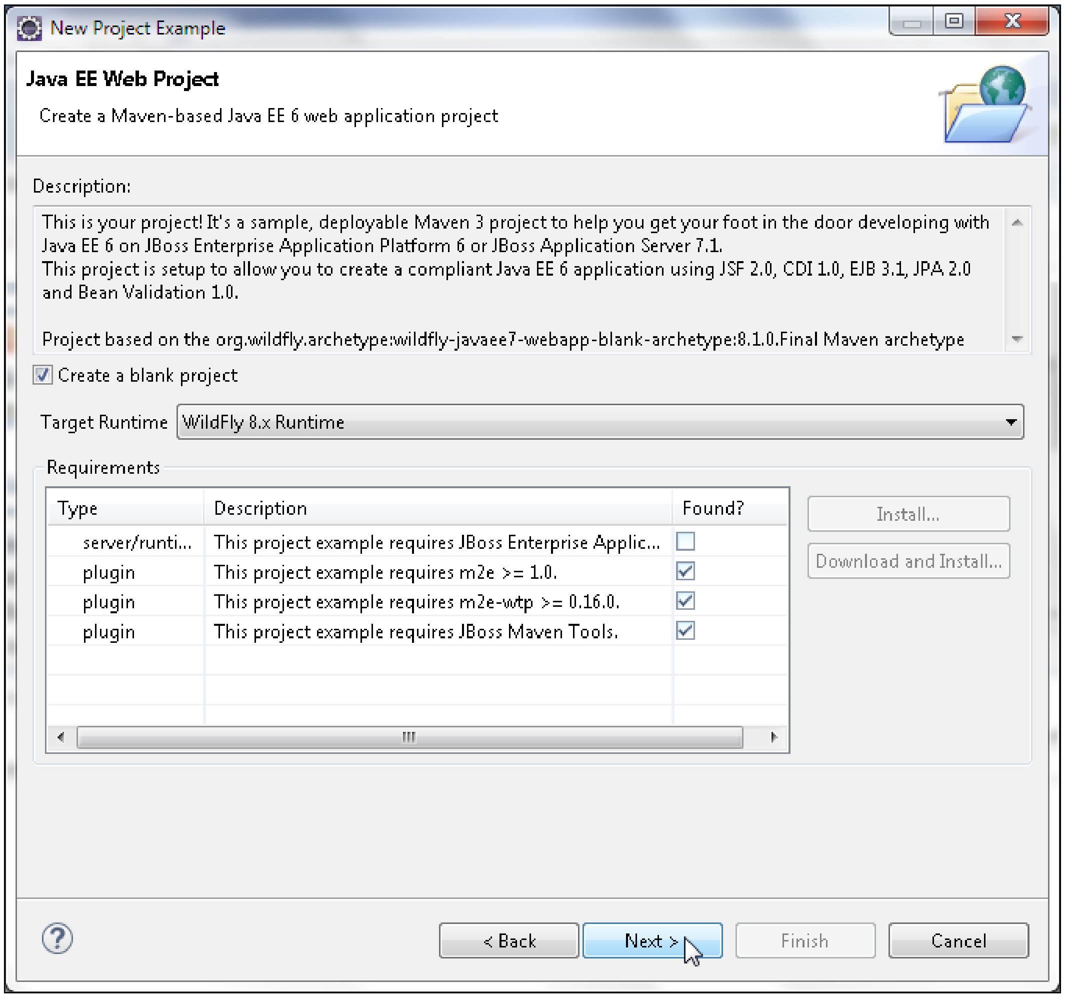
Creating a GWT project in Eclipse

In this section, we will run the starter application on WildFly. We will compile and package the application with Maven in Eclipse. First we need to import the GWT application to Eclipse.

In Eclipse, select **File** | **New** | **Other**. In **New**, select **JBoss Central** | **Java EE Web Project** in the manner shown in the following screenshot. Then, click on **Next**.



A test gets run for the requirements, which include the **m2e** and **m2eclipse-wtp** plugins, the JBoss Maven Tools plugin, and GWT Plugin, shown in the following screenshot. Click in the **Create a blank project** checkbox and select **WildFly 8.x Runtime** as **Target Runtime**. Now, click on **Next**.



Java EE Web Project

Create a Maven-based Java EE 6 web application project

Description:

This is your project! It's a sample, deployable Maven 3 project to help you get your foot in the door developing with Java EE 6 on JBoss Enterprise Application Platform 6 or JBoss Application Server 7.1.
This project is setup to allow you to create a compliant Java EE 6 application using JSF 2.0, CDI 1.0, EJB 3.1, JPA 2.0 and Bean Validation 1.0.

Project based on the org.wildfly.archetype:wildfly-javaee7-webapp-blank-archetype:8.1.0.Final Maven archetype

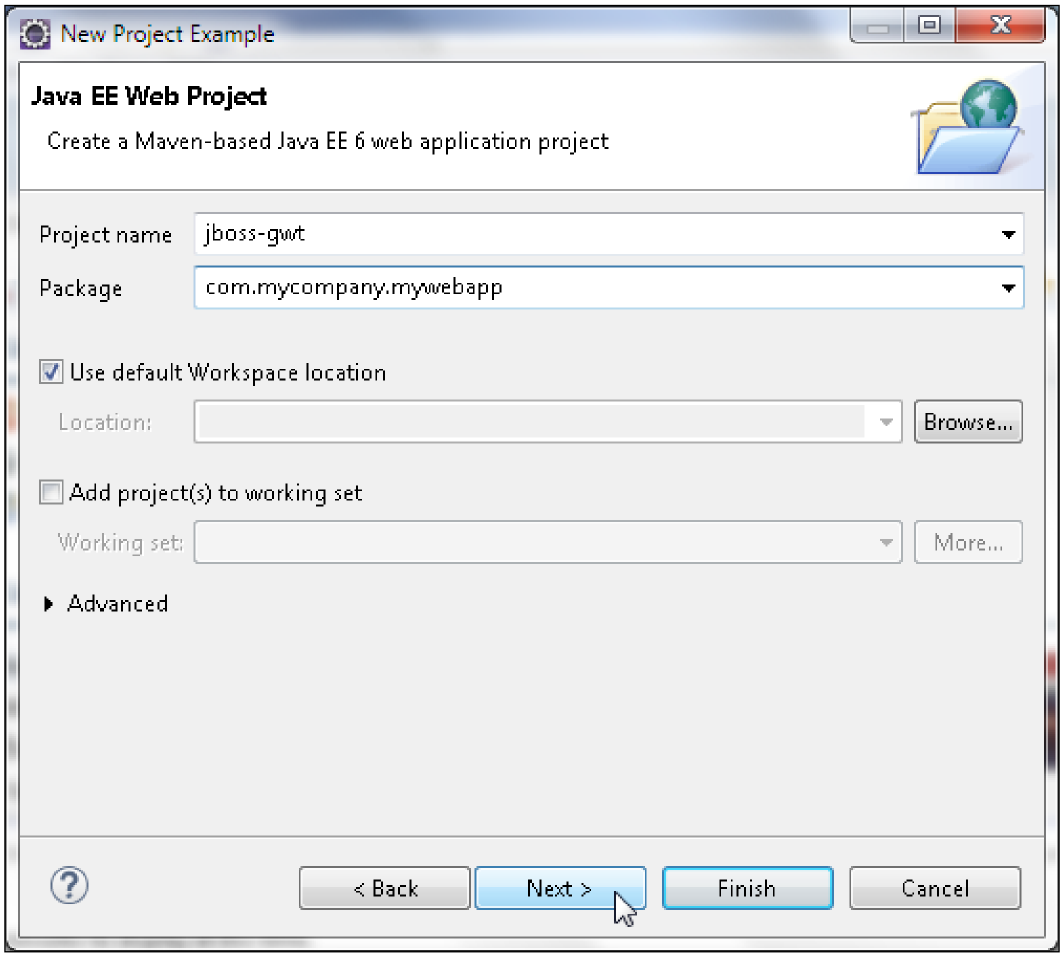
[✓] Create a blank project

Target Runtime: WildFly 8.x Runtime

Requirements

Type	Description	Found?
server/runti...	This project example requires JBoss Enterprise Applic...	☐
plugin	This project example requires m2e >= 1.0.	✓
plugin	This project example requires m2e-wtp >= 0.16.0.	✓
plugin	This project example requires JBoss Maven Tools.	✓

In the **New Project Example** wizard to create a Java EE Web Project, specify **Project name** (`jboss-gwt`) and **Package** (`com.mycompany.mywebapp`), as shown in the following screenshot, and click on **Next**.


New Project Example

Java EE Web Project

Create a Maven-based Java EE 6 web application project

Project name: jboss-gwt

Package: com.mycompany.mywebapp

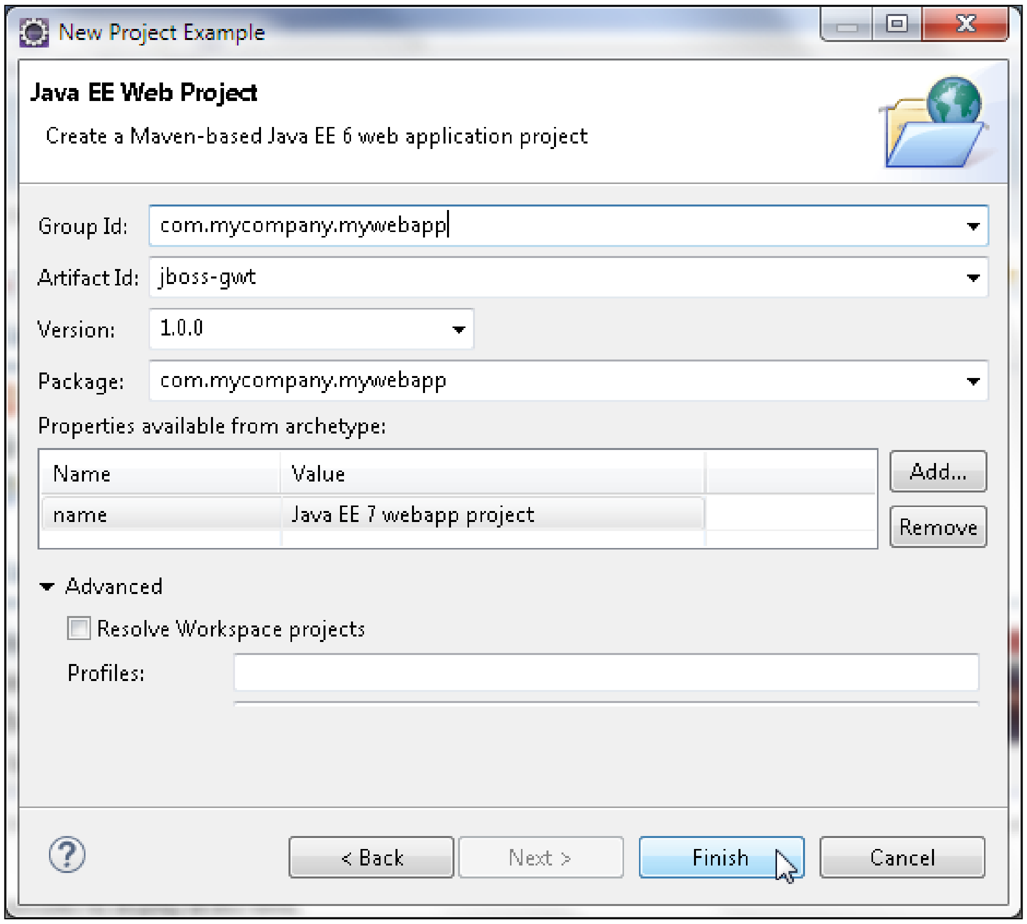
[✓] Use default Workspace location

Location:

[ ] Add project(s) to working set

Working set:

Specify **Group Id** (`com.mycompany.mywebapp`), **Artifact Id** (`jboss-gwt`), **Version** (`1.0.0`), and **Package** (`com.mycompnay.mywebapp`), as shown in the following screenshot . After this, click on **Finish**.

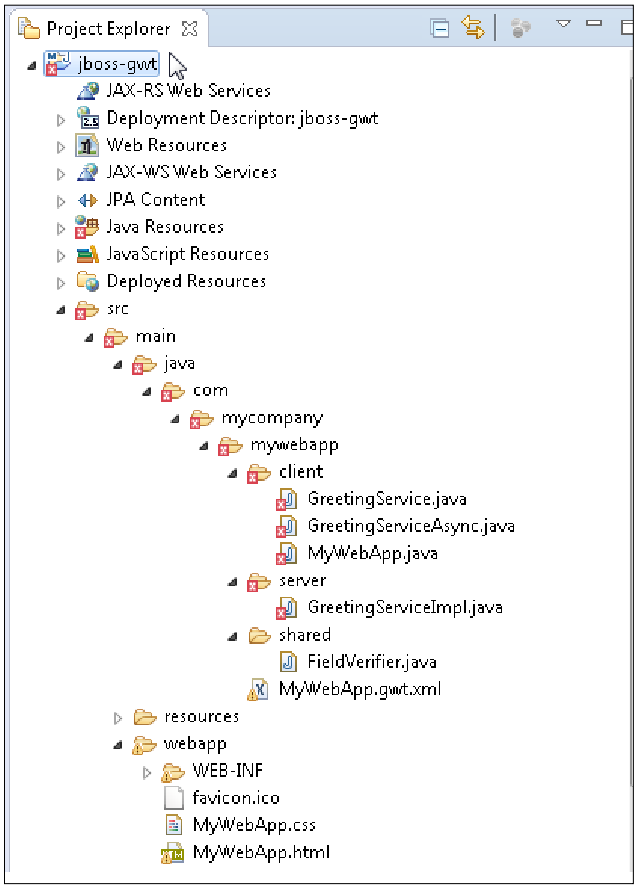


Group Id: com.mycompany.mywebapp
Artifact Id: jboss-gwt
Version: 1.0.0
Package: com.mycompany.mywebapp
Properties available from archetype:

Name	Value
name	Java EE 7 webapp project

Resolve Workspace projects [ ]
Profiles:

The `jboss-gwt` project gets created. But, the GWT project generated is not the one we will run. Remove the files in the `jboss-gwt` project's `\\jboss-gwt\src\main\java` folder and the `\\jboss-gwt\src\main\webapp` folder. Add the `C:\gwt-2.6.0\MyWebApp\src` folder files generated for the starter GWT web application on the command line to the `jboss-gwt` project's `\\jboss-gwt\src\main\java` folder. Then, copy the files of the `C:\GWT\gwt-2.6.0\MyWebApp\war` directory to the `\\jboss-gwt\src\main\webapp` directory. Select **File** | **Refresh** to refresh the application folder. The starter GWT web application is shown in **Project Explorer** as follows:



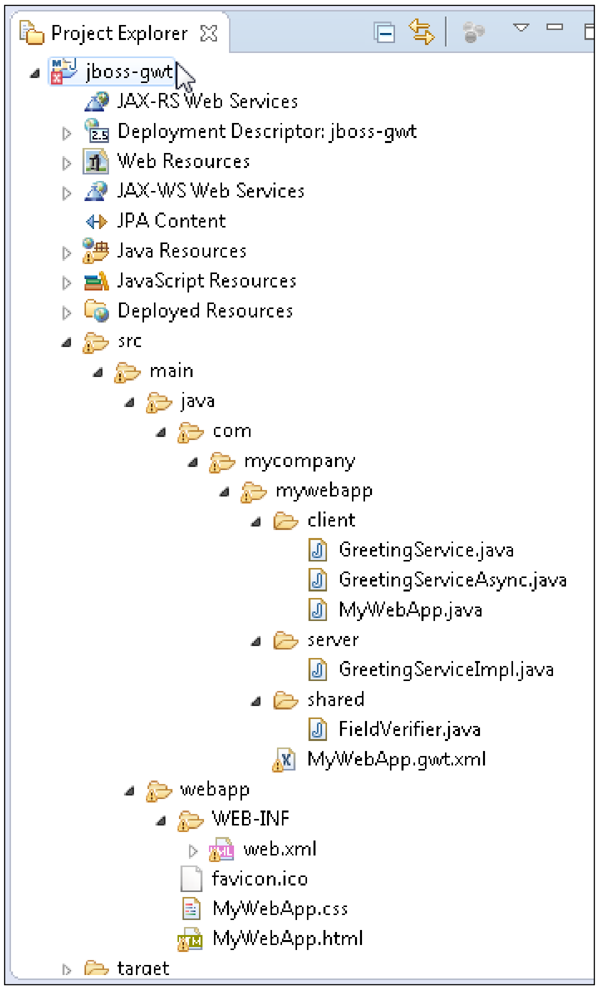
Deploying the starter project to WildFly 8.1

In this section, we will compile, package, and deploy the GWT starter web application to WildFly 8.1. The JDK compliance level for the `jboss-gwt` project should be set to **1.7**. The global Java compiler, JDK compliance, was set to **1.7** earlier—a global setting that applies to every project. If a project-level JDK compliance is required to be set, right-click on the `jboss-gwt` project and select **Properties**. In **Properties**, select the **Java Compiler** node. Select **Compiler compliance level** as **1.7** and click on **OK**. If the global JDK compliance is set to **1.7**, the project-level JDK compliance is not required to be set separately.

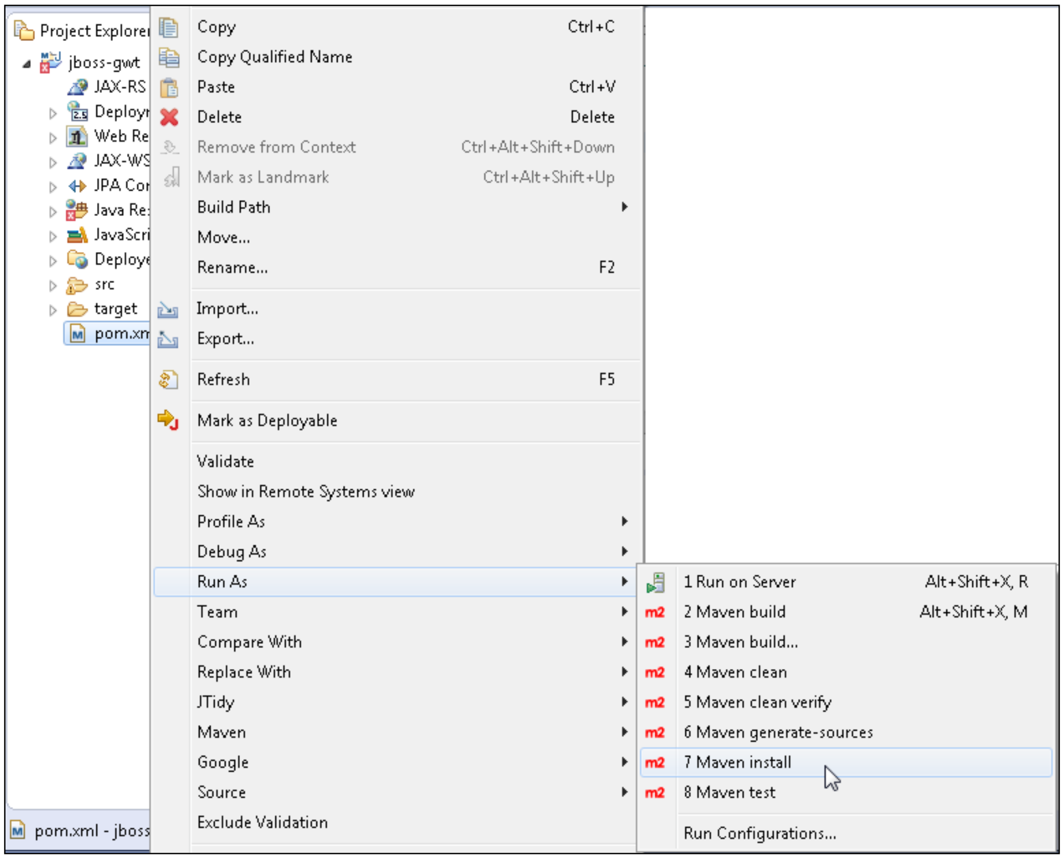
To the Maven WAR plugin configuration in `pom.xml`, specify the output directory as the `deployments` directory for WildFly 8.1, as follows:

```
<plugin>
  <artifactId>maven-war-plugin</artifactId>
  <version>2.1.1</version>
  <configuration>
    <outputDirectory>C:\wildfly-8.1.0.Final\standalone\deployments</
outputDirectory>
  </configuration>
</plugin>
```

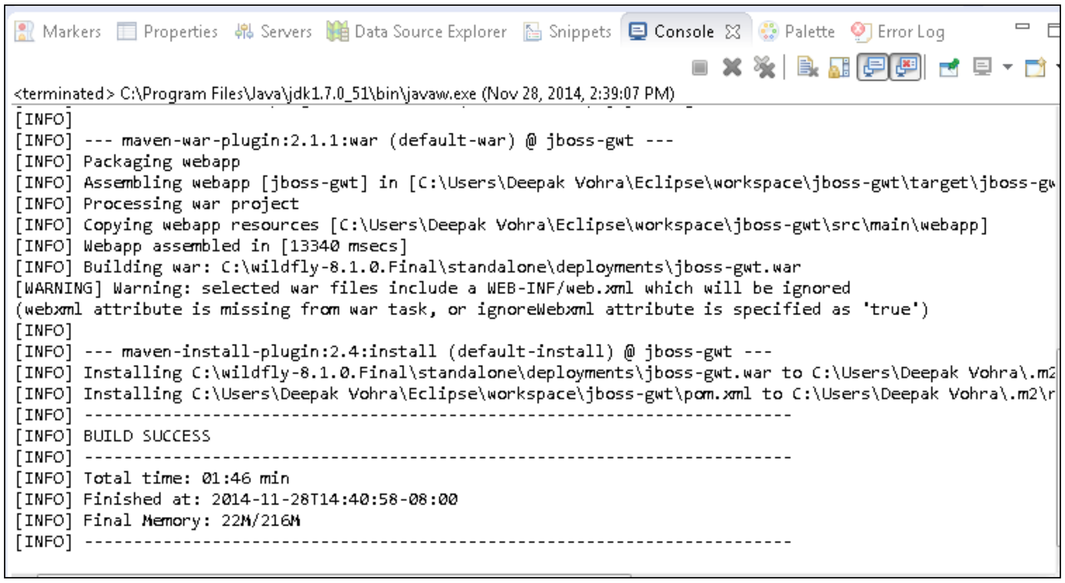
The complete `pom.xml` file is available in the downloadable code zip. The errors in the GWT application files get removed after the project dependencies have been added to `pom.xml`, as shown in the following screenshot:



Start the WildFly 8.1 server. In **Project Explorer**, right-click on `pom.xml` and select **Run As** | **Maven install**, as shown in the following screenshot:



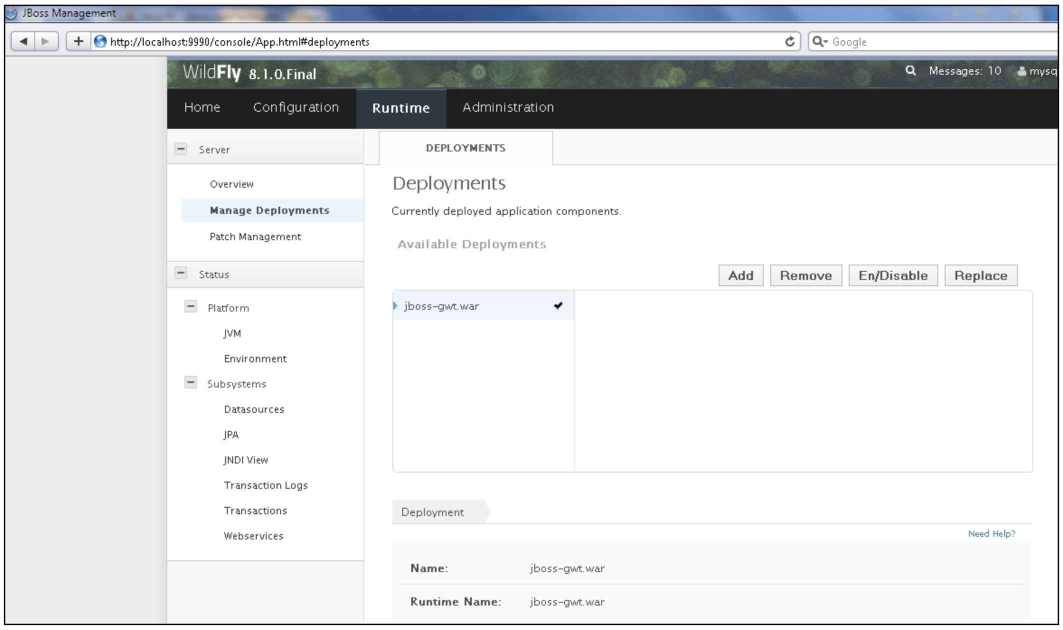
The `jboss-gwt` application gets compiled, packaged, and deployed to WildFly 8.1. The Maven `pom.xml` build and configuration script outputs the message **BUILD SUCCESS**, as shown in the following screenshot:



Markers Properties Servers Data Source Explorer Snippets Console Palette Error Log

```
<terminated> C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Nov 28, 2014, 2:39:07 PM)
[INFO]
[INFO] --- maven-war-plugin:2.1.1:war (default-war) @ jboss-gwt ---
[INFO] Packaging webapp
[INFO] Assembling webapp [jboss-gwt] in [C:\Users\Deepak Vohra\Eclipse\workspace\jboss-gwt\target\jboss-gw
[INFO] Processing war project
[INFO] Copying webapp resources [C:\Users\Deepak Vohra\Eclipse\workspace\jboss-gwt\src\main\webapp]
[INFO] Webapp assembled in [13340 msecs]
[INFO] Building war: C:\wildfly-8.1.0.Final\standalone\deployments\jboss-gwt.war
[WARNING] Warning: selected war files include a WEB-INF/web.xml which will be ignored
(webxml attribute is missing from war task, or ignoreWebxml attribute is specified as 'true')
[INFO]
[INFO] --- maven-install-plugin:2.4:install (default-install) @ jboss-gwt ---
[INFO] Installing C:\wildfly-8.1.0.Final\standalone\deployments\jboss-gwt.war to C:\Users\Deepak Vohra\.m2
[INFO] Installing C:\Users\Deepak Vohra\Eclipse\workspace\jboss-gwt\pom.xml to C:\Users\Deepak Vohra\.m2\r
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 01:46 min
[INFO] Finished at: 2014-11-28T14:40:58-08:00
[INFO] Final Memory: 22M/216M
[INFO] ------------------------------------------------------------------------
```

The `jboss-gwt.war` application gets deployed to WildFly 8.1. Now, log in to the WildFly 8.1 **Administration Console** using the URL `http://localhost:8080`, and click on **Manage Deployments**. The `jboss-gwt.war` file should be listed as deployed, as shown in the following screenshot:



Deployments

Currently deployed application components.

Available Deployments

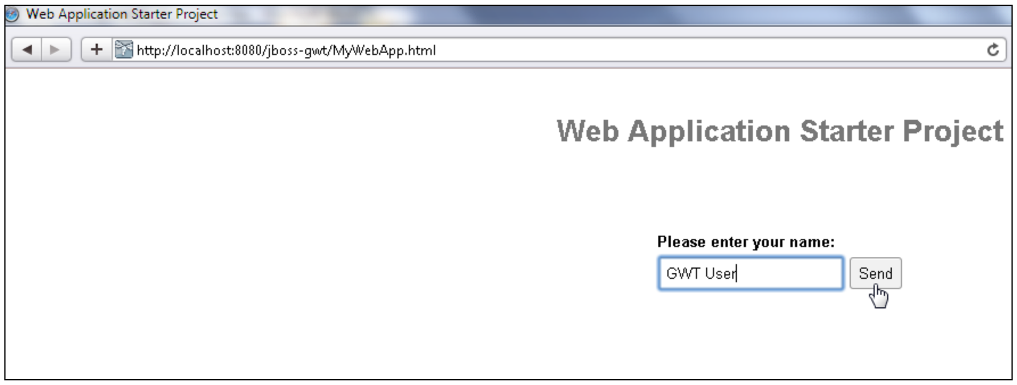
jboss-gwt.war ✓

Name: jboss-gwt.war

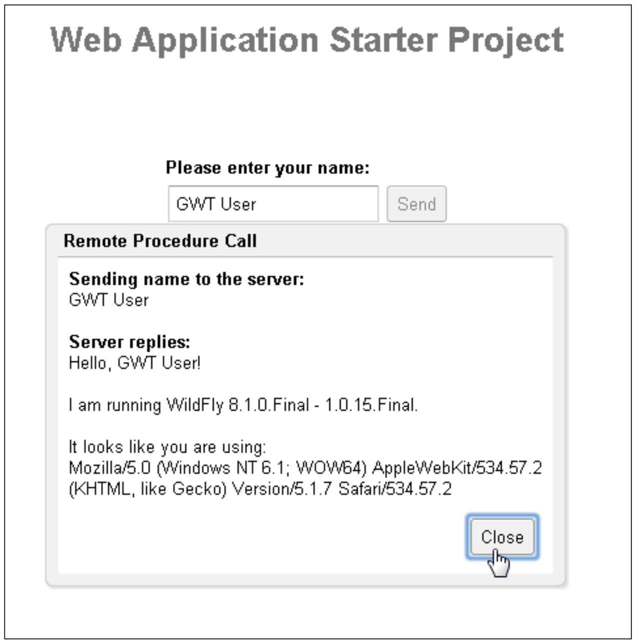
Runtime Name: jboss-gwt.war

Running the starter project on WildFly 8.1

In this section, we will run the starter GWT web application generated on the command line and added to a Java EE web project in Eclipse. Invoke the GWT HTML page using the URL `http://localhost:8080/jboss-gwt/MyWebApp.html`. The **Web Application Starter Project** user interface gets displayed in the browser. Specify a name and click on **Send**, as shown in the following screenshot:


Web Application Starter Project
http://localhost:8080/jboss-gwt/MyWebApp.html
Web Application Starter Project
Please enter your name:
GWT User
Send

The user name gets sent to the WildFly 8.1 server and a reply from the server is received and displayed in the browser, as shown in the following screenshot. Then, click on **Close**.



Sending name to the server:
GWT User

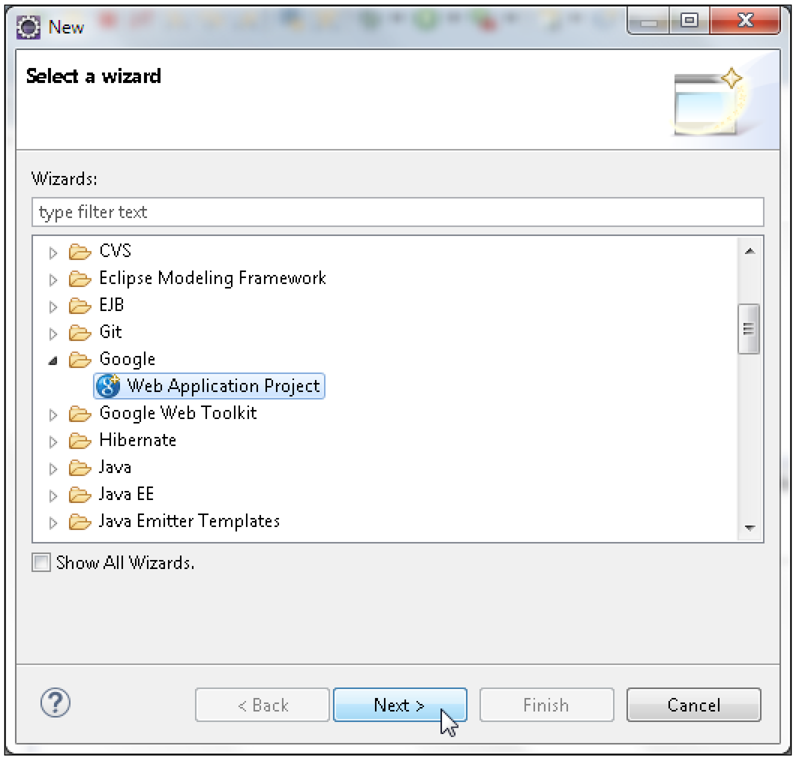
Server replies:
Hello, GWT User!

I am running WildFly 8.1.0.Final - 1.0.15.Final.

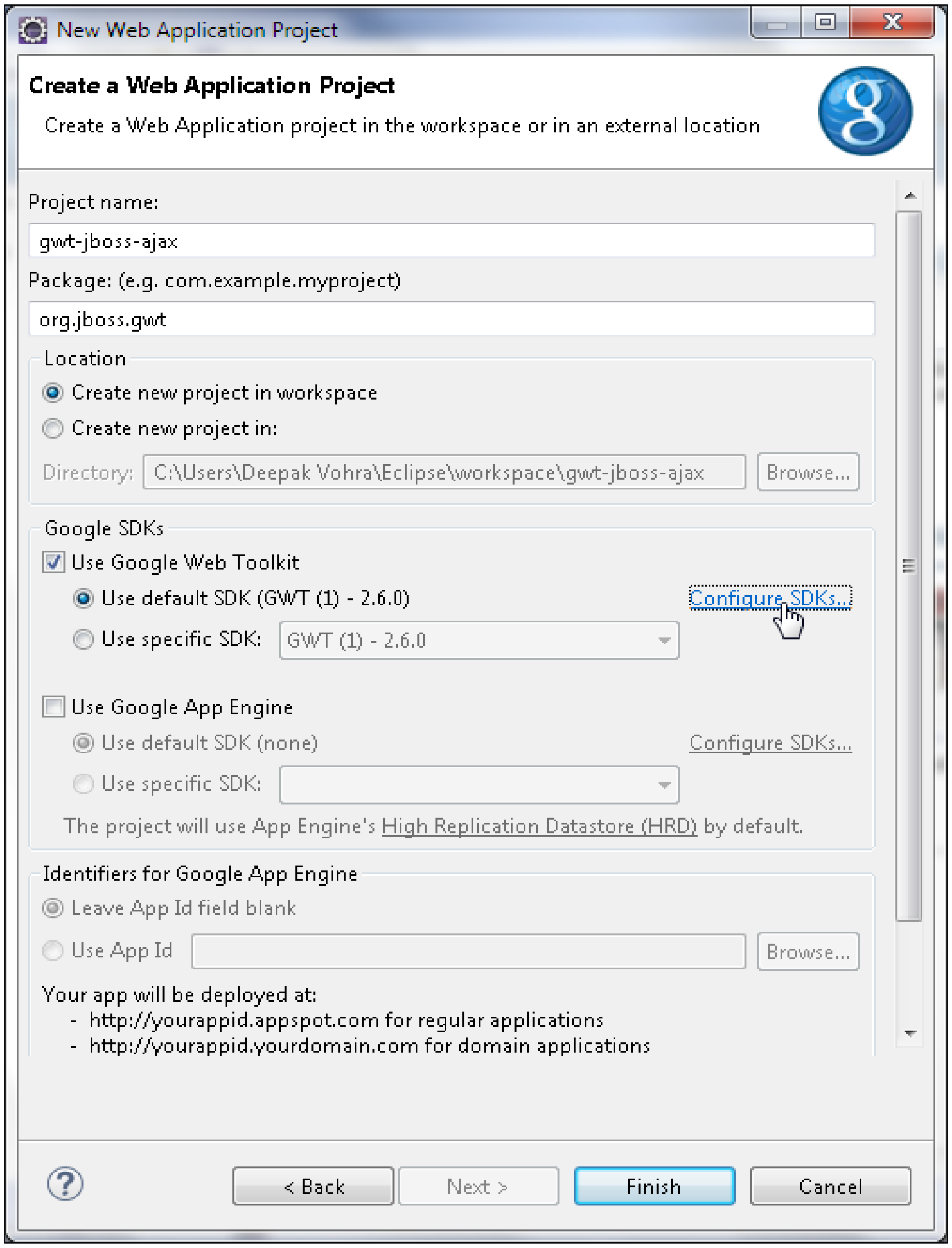
It looks like you are using:
Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/534.57.2 (KHTML, like Gecko) Version/5.1.7 Safari/534.57.2

Creating a new GWT project

Having discussed the starter GWT web project, we will create a new GWT web application. Select **File** | **New** | **Other** in Eclipse. In **New**, select **Google** | **Web Application Project** and click on **Next**, as shown in the following screenshot:



The **New Web Application Project** wizard gets started. Specify **Project name** (`gwt-jboss-ajax`) and **Package** (`org.jboss.gwt`). Select the default setting for **Location**. In **Google SDKs**, click on the **Use Google Web Toolkit** checkbox and click on **Configure SDKs**, as shown in the following screenshot:



New Web Application Project

Create a Web Application Project

Create a Web Application project in the workspace or in an external location

Project name: gwt-jboss-ajax

Package: (e.g. com.example.myproject) org.jboss.gwt

Location

[● Create new project in workspace]

○ Create new project in:

Directory: C:\Users\Deepak Vohra\Eclipse\workspace\gwt-jboss-ajax

Google SDKs

[✓ Use Google Web Toolkit]

[● Use default SDK (GWT (1) - 2.6.0)]

○ Use specific SDK: GWT (1) - 2.6.0

☐ Use Google App Engine

● Use default SDK (none)

○ Use specific SDK:

The project will use App Engine's High Replication Datastore (HRD) by default.

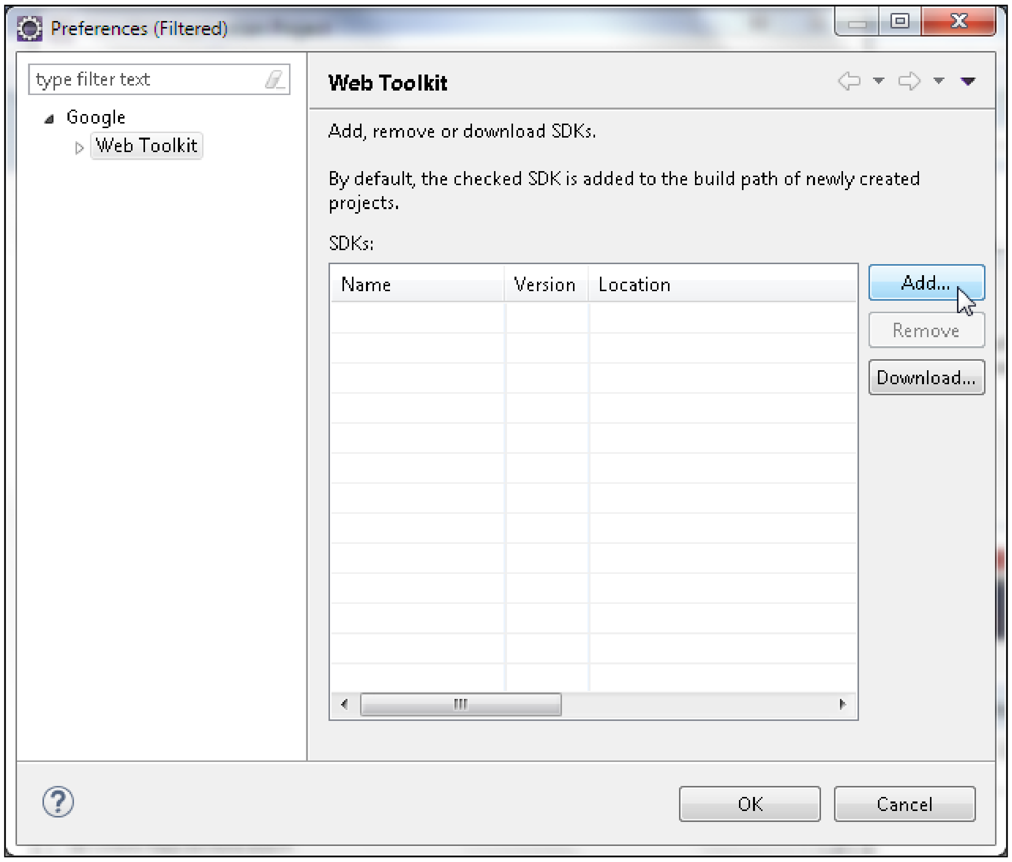
Identifiers for Google App Engine

● Leave App Id field blank

○ Use App Id

Your app will be deployed at:
- http://yourappid.appspot.com for regular applications
- http://yourappid.yourdomain.com for domain applications

A filtered version of **Preferences** gets listed with only **Google | Web Toolkit**. Click on **Add** to add a new SDK, as shown in the following screenshot:



Preferences (Filtered)

type filter text

Google

Web Toolkit

Web Toolkit

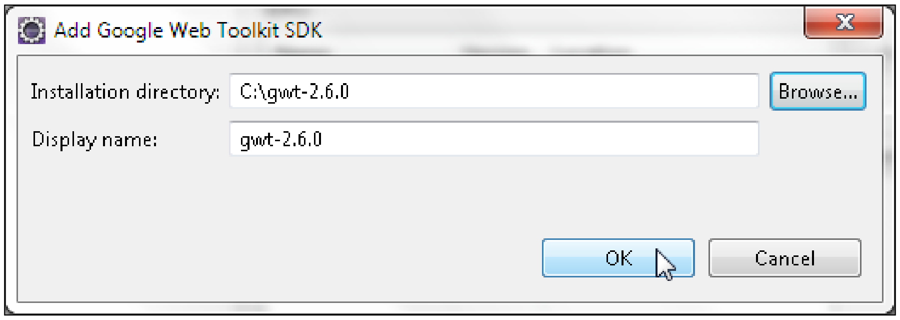
Add, remove or download SDKs.

By default, the checked SDK is added to the build path of newly created projects.

SDKs:

Name	Version	Location

In **Add Google Web Toolkit SDK**, select **Installation directory** for GWT 2.6 (`C:\gwt-2.6`) with **Display name** as `gwt-2.6.0`, as shown in the following screenshot. Then, click on **OK**.

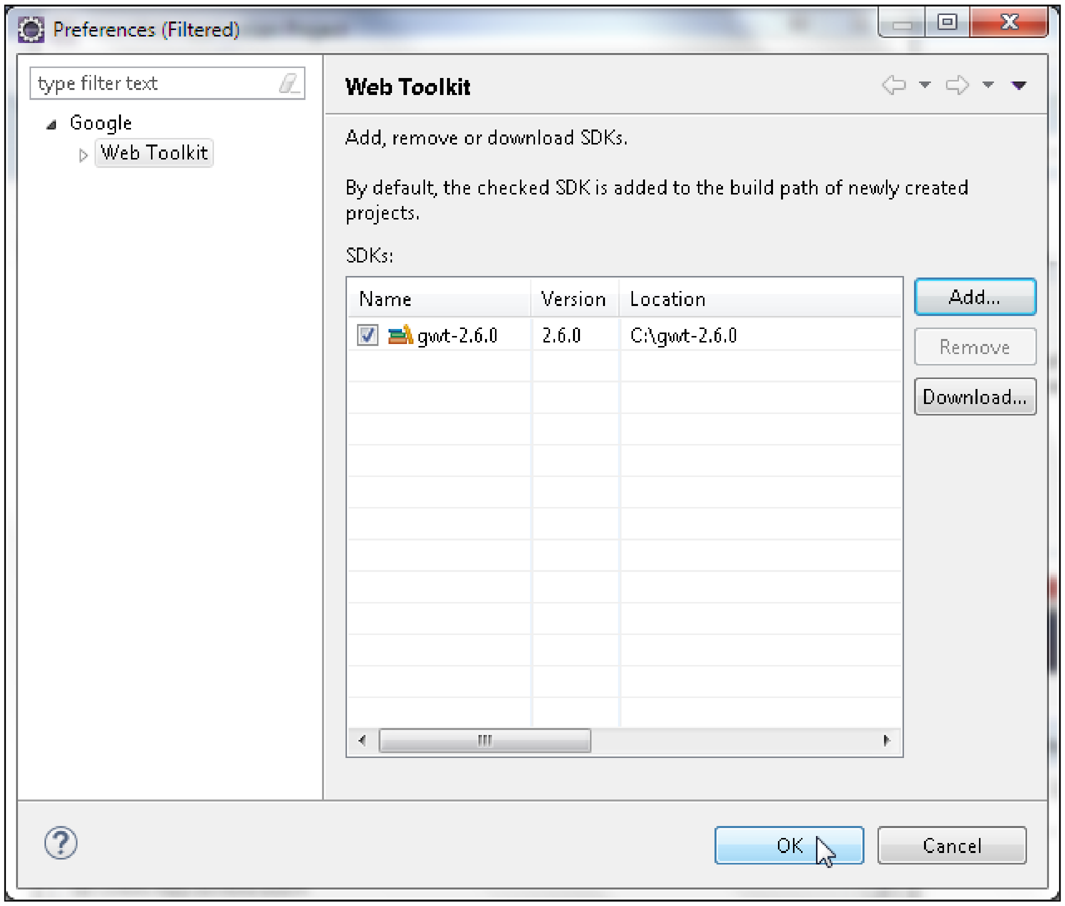


Add Google Web Toolkit SDK

Installation directory: C:\gwt-2.6.0

Display name: gwt-2.6.0

A GWT SDK gets added to **Preferences**, as shown in the following screenshot:



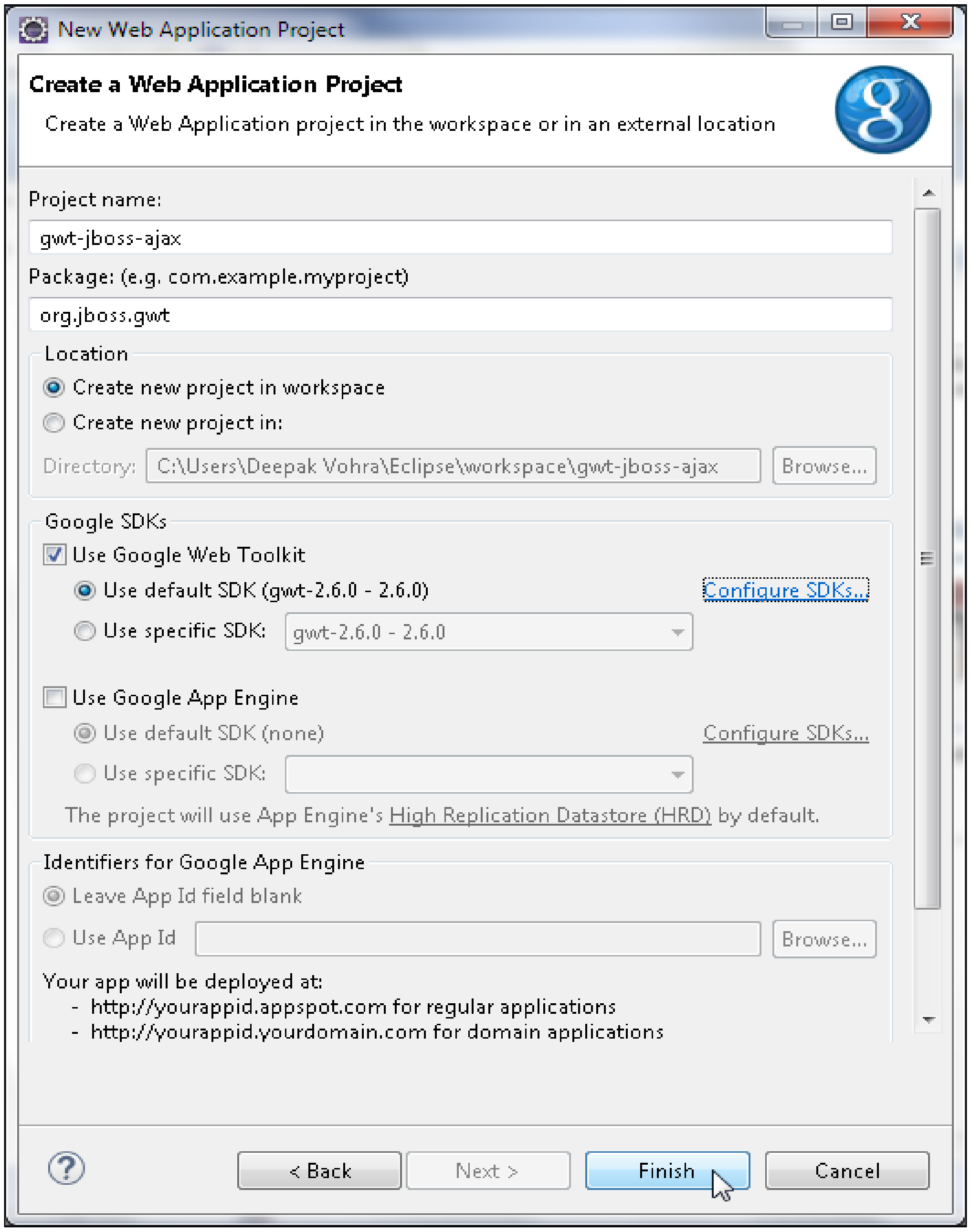
Add, remove or download SDKs.

By default, the checked SDK is added to the build path of newly created projects.

SDKs:

Name	Version	Location
✓ gwt-2.6.0	2.6.0	C:\gwt-2.6.0

In the **New Web Application Project** wizard, click on the **Create a Web Application Project** window. Click on **Finish**.



Create a Web Application Project

Create a Web Application project in the workspace or in an external location

Project name:

gwt-jboss-ajax

Package: (e.g. com.example.myproject)

org.jboss.gwt

Location

[✓ Create new project in workspace]

Create new project in:

Directory: C:\Users\Deepak Vohra\Eclipse\workspace\gwt-jboss-ajax

Google SDKs

[✓ Use Google Web Toolkit]

[✓ Use default SDK (gwt-2.6.0 - 2.6.0)]

Use specific SDK: gwt-2.6.0 - 2.6.0

Use Google App Engine

Use default SDK (none)

Use specific SDK:

The project will use App Engine's High Replication Datastore (HRD) by default.

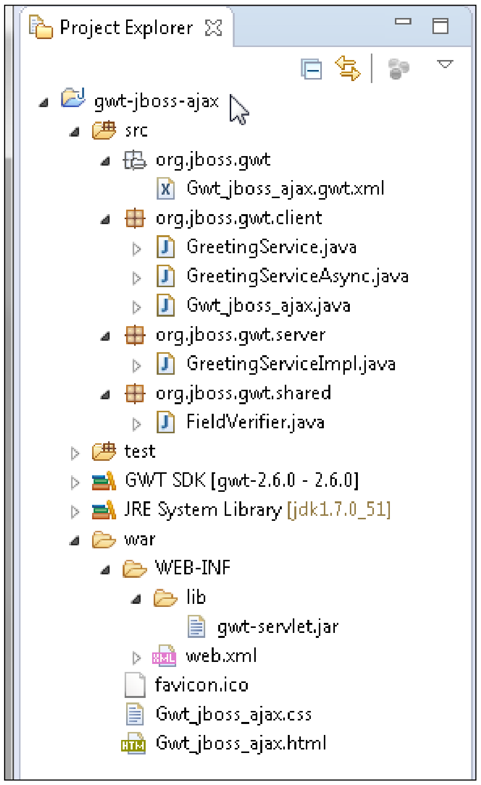
Identifiers for Google App Engine

Leave App Id field blank

Use App Id

Your app will be deployed at:
- http://yourappid.appspot.com for regular applications
- http://yourappid.yourdomain.com for domain applications

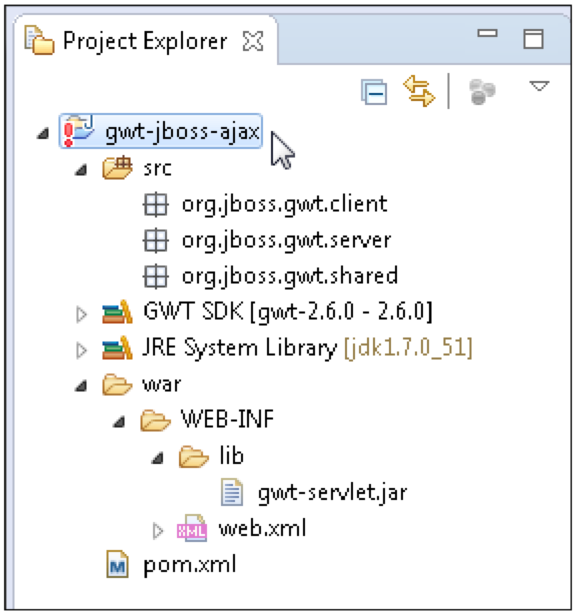
A Google Web Application project, which is essentially a GWT project, gets created with the directory structure shown in the following screenshot:



A typical GWT web project consists of the following files, modules, and pages:

- Client Java source files to be compiled into JavaScript. GWT widgets from the GWT SDK Java API can be added to the client Java source files.
- Server Java source files, which can implement some service logic.
- GWT modules, which specify the configuration for the GWT project. GWT libraries are called modules.
- HTML host pages to run the GWT modules.

We will create a GWT web application to create a catalog entry with an input form. The input widgets for the input form will be specified in a client Java source file. A GWT module will specify the configuration. An HTML host page will run the module in a browser. As GWT widgets support Ajax, we will validate a new catalog entry dynamically with Ajax. Before we create a new GWT project, delete the files created in the default Google Web Application project, except the `gwt-jboss-ajax` project root folder, the subdirectories in the `src` folder, and the `war` directory, including the `web.xml` deployment descriptor in the `//gwt-jboss-ajax/war/WEB-INF` folder. Add a `pom.xml` file (select **XML** | **XML File** in the **New** wizard) for a Maven project to the root folder `/gwt-jboss-ajax`. The directory structure of the `gwt-jboss-ajax` project is shown in the following screenshot:


```
Project Explorer
gwt-jboss-ajax
  src
    org.jboss.gwt.client
    org.jboss.gwt.server
    org.jboss.gwt.shared
  GWT SDK [gwt-2.6.0 - 2.6.0]
  JRE System Library [jdk1.7.0_51]
  war
    WEB-INF
      lib
        gwt-servlet.jar
      web.xml
  pom.xml
```

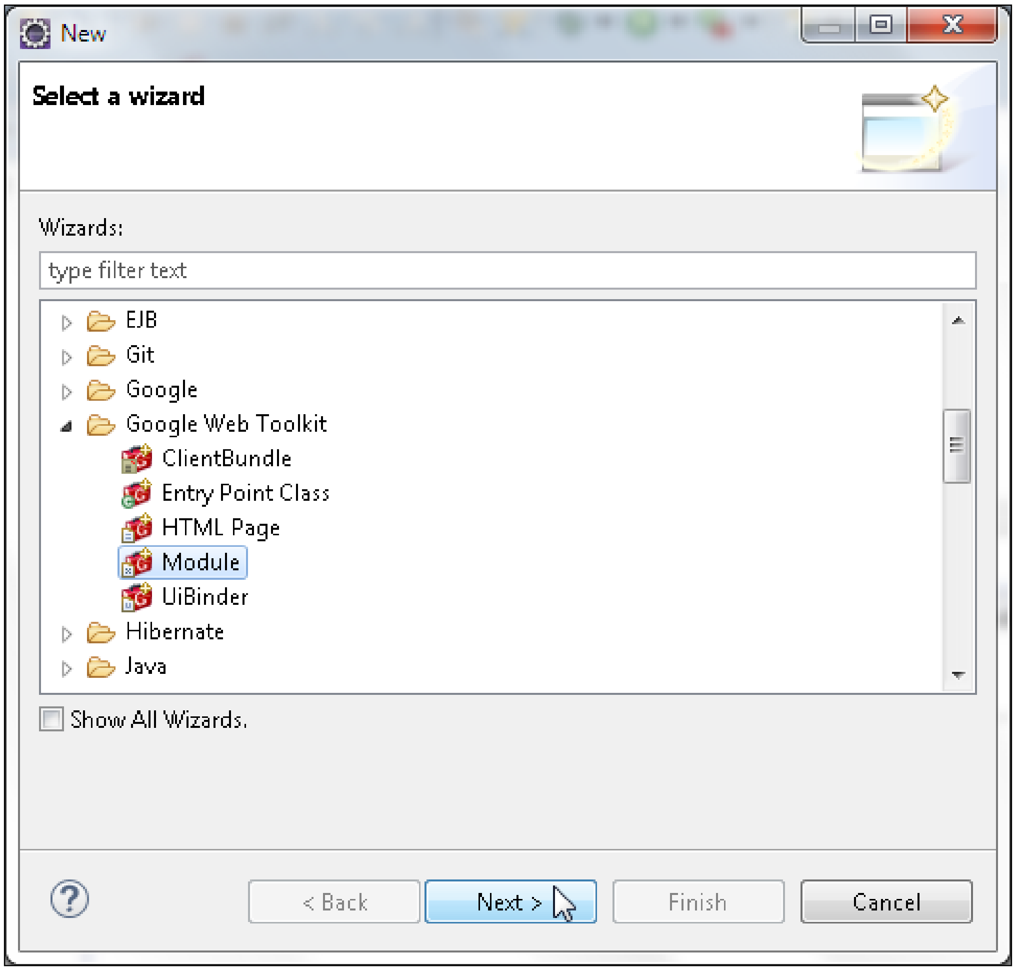
Creating a GWT module

The configuration for a GWT project is specified in a GWT module. A GWT module is an XML file with the `.gwt.xml` extension and has the provision to specify the following GWT configuration:

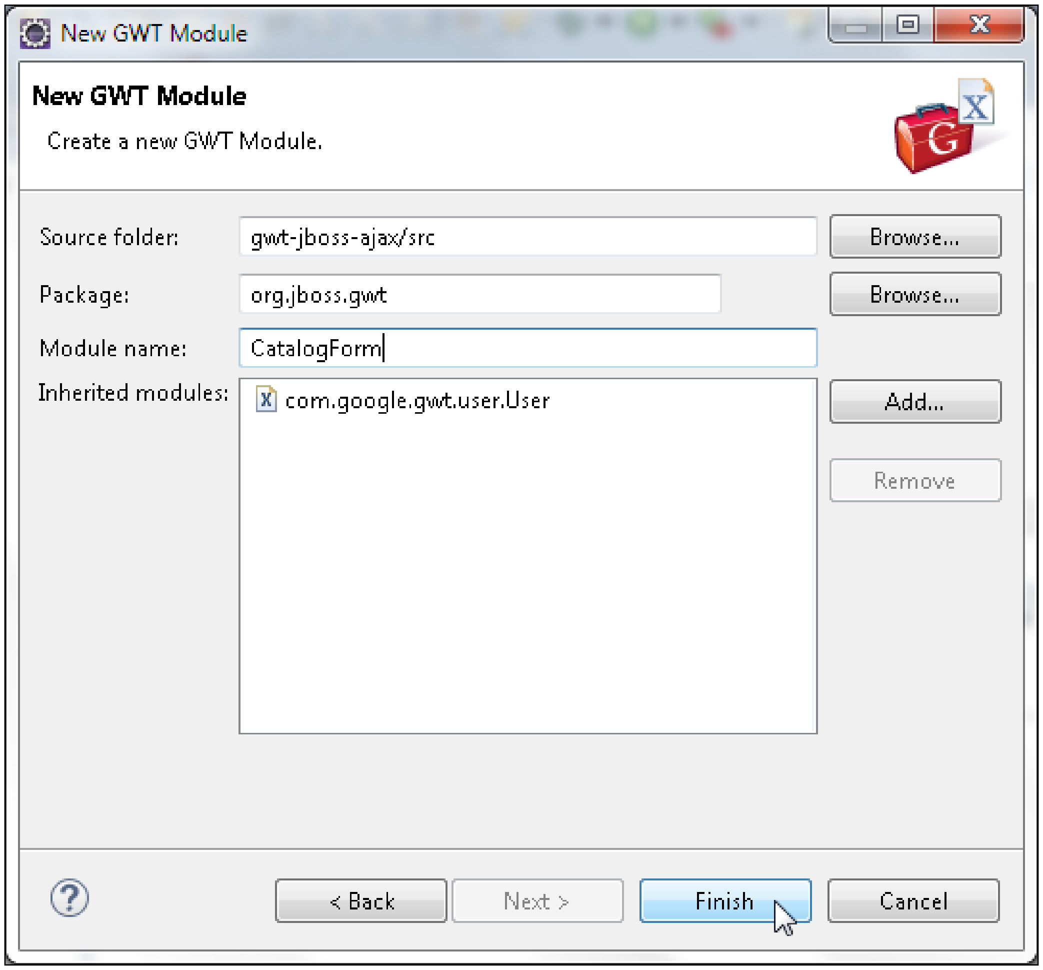
- **Entry-point application class name**: An entry-point class name is an entry point to the module and must be of the type `com.google.gwt.core.client.EntryPoint`. When a module is loaded, all entry-point classes get instantiated and their `onModuleLoad` method gets called.

- **The source path for the GWT project**: The client Java source files of the GWT project that are to be compiled into JavaScript by the GWT compiler must be in the source path. The server Java source files must also be in the source path. The default source path is client.
- The public path for public resources such as CSS and images
- Inherited modules

In this section, we will create a GWT module. Select **File** | **New** | **Other**. In **New**, select **Google Web Toolkit** | **Module**, as shown in the following screenshot. After this, click on **Next**.



In **New GWT Module**, select **Source folder** as `gwt-jboss-ajax/src` and specify **Package** as `org.jboss.gwt` and **Module name** as `CatalogForm`, as shown in the following screenshot. In **Inherited modules**, add the `com.google.gwt.user.User` module, which provides the core GWT functionality, such as the **EntryPoint** class and the GWT widgets and panels. Now, click on **Finish**.

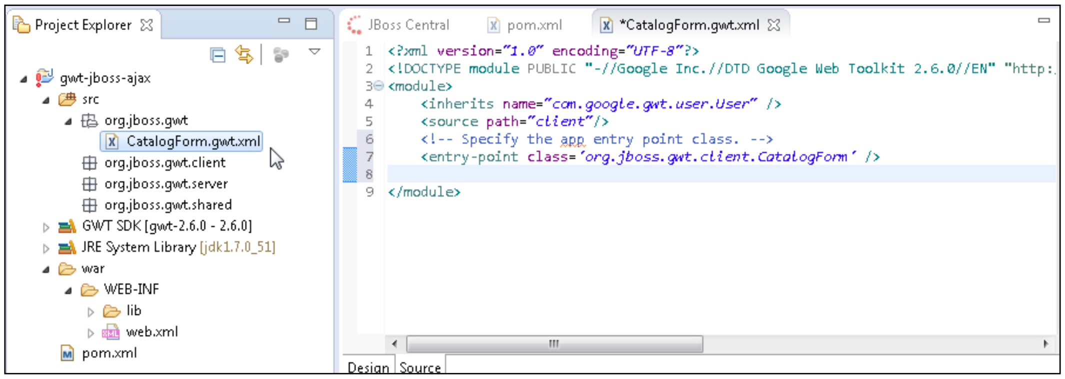


Source folder: gwt-jboss-ajax/src
Package: org.jboss.gwt
Module name: CatalogForm
Inherited modules: com.google.gwt.user.User

A `CatalogForm.gwt.xml` file gets created in the `gwt-jboss-ajax\src\org\jboss\gwt` directory, which is the root package for the GWT project. The inherited `com.google.gwt.user.User` module is configured with the `<inherits/>` element. The source path is configured with the `<source/>` element's path attribute and is set to client, which is relative to the directory containing the module. We will create a client Java source file, the entry-point class, in the client subdirectory in a later section of this chapter. Specify the entry-point class in the `<entry-point/>` element's `class` attribute as `org.jboss.gwt.client.CatalogForm`. The `CatalogForm.gwt.xml` module is listed in the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE module PUBLIC "-//Google Inc.//DTD Google Web Toolkit
2.6.0//EN" "http://google-web-toolkit.googlecode.com/svn/tags/2.6.0/
distro-source/core/src/gwt-module.dtd">
<module>
  <inherits name="com.google.gwt.user.User" />
  <source path="client"/>
  <!-- Specify the app entry point class. -->
  <entry-point class='org.jboss.gwt.client.CatalogForm' />
</module>
```

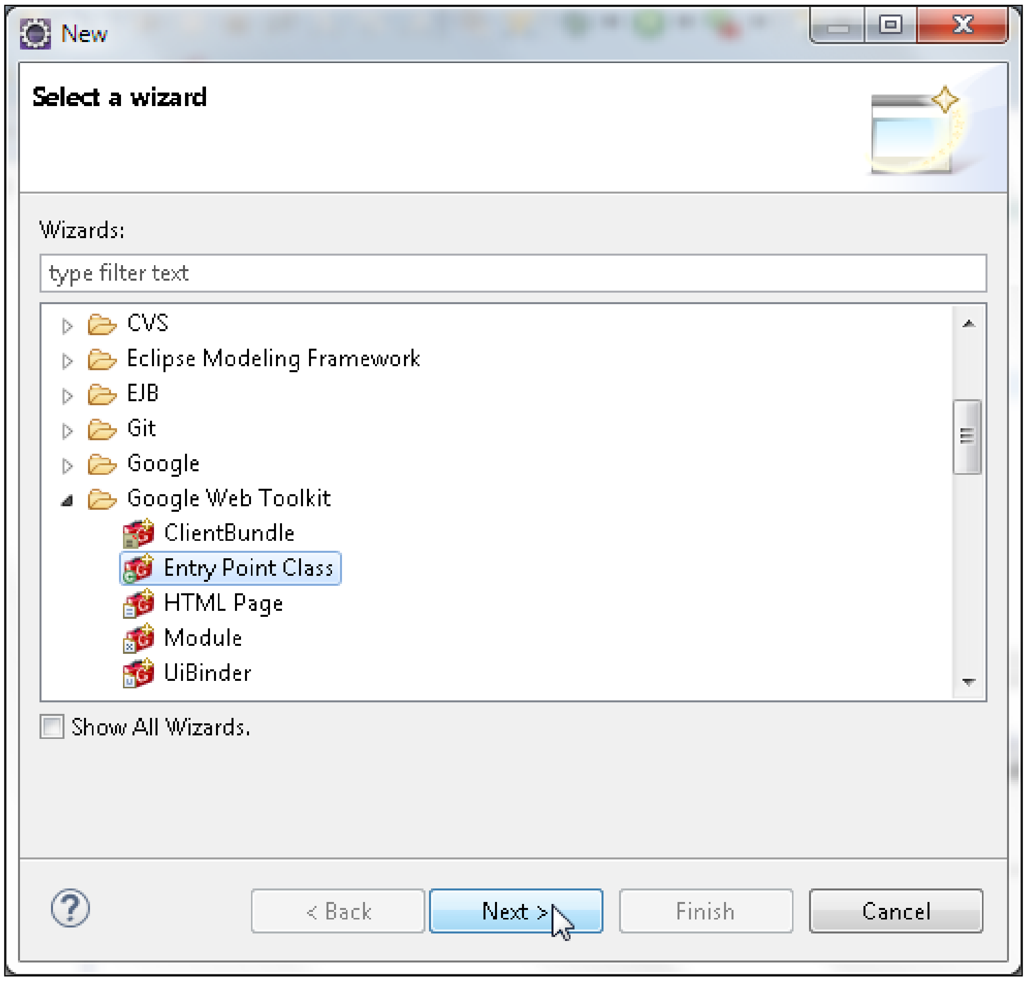
In the following screenshot, the module is shown in **Project Explorer** in the `gwt-jboss-ajax` project:



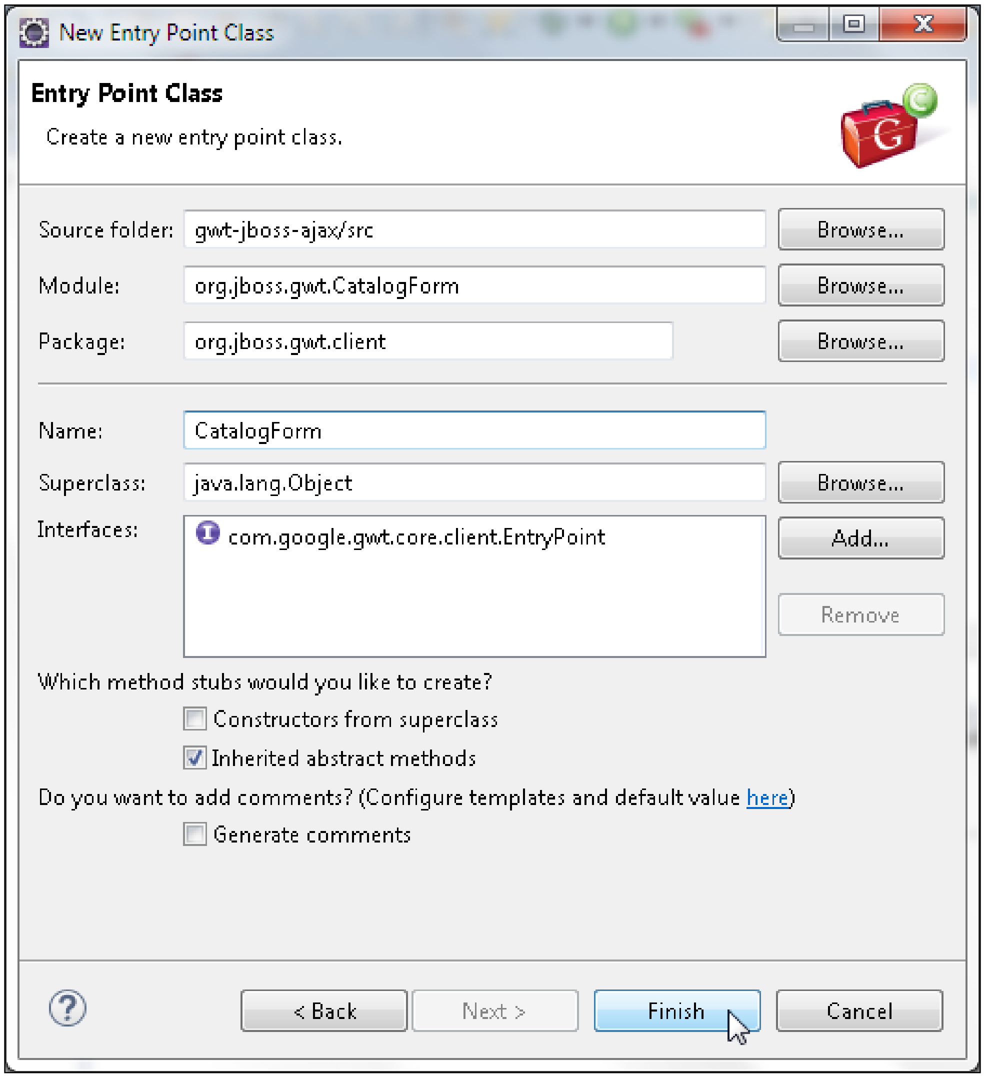
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE module PUBLIC "-//Google Inc.//DTD Google Web Toolkit 2.6.0//EN" "http:
<module>
    <inherits name="com.google.gwt.user.User" />
    <source path="client"/>
    <!-- Specify the app entry point class. -->
    <entry-point class='org.jboss.gwt.client.CatalogForm' />

</module>
```

Creating an entry-point class

When a module gets compiled into JavaScript, the module can be used from an HTML host page. An entry-point class is the entry point to a module and gets instantiated when a module is loaded on a web page or an HTML host page. The entry-point class must implement the `com.google.gwt.core.client.EntryPoint` interface. In this section, we will create the entry-point class `org.jboss.gwt.client.CatalogForm` to create an input form for a catalog entry. We will use GWT widgets and panels from the `com.google.gwt.user.client.ui` package, and GWT events and event handlers from the `com.google.gwt.event.dom.client` package. Select **File** | **New** | **Other** and in **New**, select **Google Web Toolkit** | **Entry Point Class**, as shown in the following screenshot. Then, click on **Next**.



In the **Entry Point Class** wizard, select **Source folder** as `gwt-jboss-ajax | src`, and specify **Module** as `org.jboss.gwt.client.CatalogForm`, **Package** as `org.jboss.gwt.client`, **Name** as `CatalogForm`, and **Interfaces** as `com.google.gwt.core.client.EntryPoint`, as shown in the following screenshot. Now, click on **Finish**.


New Entry Point Class

Entry Point Class

Create a new entry point class.

Source folder: gwt-jboss-ajax/src

Module: org.jboss.gwt.CatalogForm

Package: org.jboss.gwt.client

Name: CatalogForm

Superclass: java.lang.Object

Interfaces: com.google.gwt.core.client.EntryPoint

Which method stubs would you like to create?

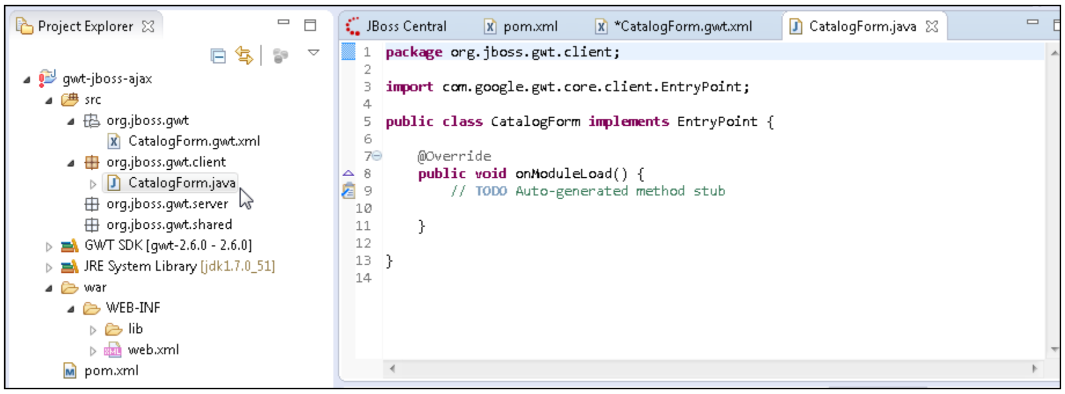
☐ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

☐ Generate comments

The `org.jboss.gwt.client.CatalogForm` entry-point class gets created. The entry-point class is shown in the `gwt-jboss-ajax` web application in **Project Explorer** in the following screenshot:



```
package org.jboss.gwt.client;

import com.google.gwt.core.client.EntryPoint;

public class CatalogForm implements EntryPoint {

    @Override
    public void onModuleLoad() {
        // TODO Auto-generated method stub

    }

}
```

The entry-point class consists of the following GWT widgets, panels, events, and event handlers:

GWT Component	Description	Function in the example GWT project
`com.google.gwt.user.client.ui.RootPanel`	This is a panel to which all widgets are added. A default root panel gets created.	This is used to add the GWT widgets, which include textBox, label, and button.
`com.google.gwt.user.client.ui.TextBox`	This creates a single-line text box.	This is used to add input fields for a catalog entry.
`com.google.gwt.user.client.ui.Label`	This is a text label in a `<div/>` element.	This is used to add labels for the input fields for a catalog entry.
`com.google.gwt.user.client.ui.Button`	This is a button widget.	This is used to generate a click event to submit a catalog entry.
`com.google.gwt.event.dom.client.KeyUpEvent`	This is a key-up event.	This is used to send an Ajax request.
`com.google.gwt.event.dom.client.KeyUpHandler`	This is an event handler for a key-up event.	This is used to validate a Catalog ID value.

GWT Component	Description	Function in the example GWT project
`com.google.gwt.event.dom.client.ClickEvent`	This is a click event.	This is used to submit the catalog entry form.
`com.google.gwt.event.dom.client.ClickHandler`	This is an event handler for a click event.	This is used to create a new catalog entry.

In the entry-point class, import the required widget, panel, event, and event handler classes. When a module is loaded, the `onModuleLoad` method, which is the entry-point method of the entry-point classes, gets called. In the `onModuleLoad()` method of the `CatalogForm` entry-point class, create six `TextBox` widgets for input text fields for a catalog entry. Add the corresponding `Label` widgets for labels to the input fields. A `Label` widget and a `TextBox` widget, for example, are created as follows:

```
final Label label1 = new Label("Catalog ID");
final TextBox textBox1 = new TextBox();
```

Add `Label` to display a validation message. Add a `Button` widget for a `Submit` button. The following line of code accomplishes this:

```
final Button button = new Button("Submit");
```

The catalog is stored in `HashMap<String, ArrayList<String>>`, in which each catalog entry is an `ArrayList<String>` array. The catalog ID for a catalog entry is also the `HashMap` key.

Create a catalog and add two catalog entries. For example, a catalog entry is created and added to the `HashMap` as follows:

```
ArrayList<String> arrayList = new ArrayList<String>();
arrayList.add(0, "catalog1");
arrayList.add(1, "Oracle Magazine");
arrayList.add(2, "Oracle Publishing");
arrayList.add(3, "May-June 2006");
arrayList.add(4, "Tuning Your View Objects");
arrayList.add(5, "Steve Muench");

HashMap<String, ArrayList<String>> catalogHashMap = new
HashMap<String, ArrayList<String>>();
catalogHashMap.put("catalog1", arrayList);
```

Add the widgets to a root panel, which may be the default root panel or a widget-specific root panel. The default root panel is not created directly but accessed using the `get()` method. The root panel for a widget is accessed using the `get(String divId)` method, in which `divId` is the `<div/>` ID for a widget in the HTML host page. Add widgets to widget-specific root panels. For example, a label is added to a label-specific root panel as follows. Use the following line of code for this purpose:

```
RootPanel.get("label1").add(label1);
```

We will create an HTML host page in a later section to specify the ordering of the widgets using `<div/>` elements with associated IDs. The `TextBox` widgets support Ajax. We will validate a Catalog ID using the business logic that the Catalog ID value must not already be in `HashMap` for the catalog, and the Catalog ID must not be an empty string.

To test the validity of a Catalog ID dynamically, add `KeyUpHandler` to `TextBox` for the Catalog ID using the `addKeyUpHandler` method. Create a `KeyUpHandler` object using an inner class. `KeyUpHandler` must implement the `onKeyUp(KeyUpEvent event)` method. The following code encapsulates the discussion in this paragraph:

```
textBox1.addKeyUpHandler(new KeyUpHandler() {
  public void onKeyUp(KeyUpEvent event) {}
}
```

In the `onKeyUp` method, retrieve the Catalog ID value using the `getText()` method for `TextBox`:

```
String catalogId = textBox1.getText();
```

If `HashMap` contains the Catalog ID value, fetch `ArrayList` for the corresponding catalog entry. Set the `Label` widget to display a validation message, as `Catalog Id is not Valid`. Set the `TextBox` values to the catalog entry parameters and disable the `Submit` button. All this is accomplished by the following code:

```
if (catalogHashMap.containsKey(catalogId)) {
  ArrayList<String> arraylist = (ArrayList<String>) catalogHashMap.
get(catalogId);
  label7.setText("Catalog Id is not Valid");
  textBox2.setText((String) arraylist.get(1));
...
  button.setEnabled(false);
}
```

If `HashMap` does not contain the Catalog ID value and is not an empty `String`, set `Label` for the validation message to `Catalog Id is Valid`, set all the input fields to an empty `String`, and enable the `Submit` button so that a new catalog entry can be created.

To create a new catalog entry, add a `ClickHandler` event handler to the `Button` widget using the `addClickHandler()` method. Create a new `ClickHandler` as an inner class and implement the `onClick(ClickEvent event)` method. In the `onClick(ClickEvent event)` method, retrieve the values specified in the `TextBox` widgets using the `getText()` method and create `ArrayList<String>`. Add `ArrayList<String>` to the `HashMap` catalog with the `put` method.

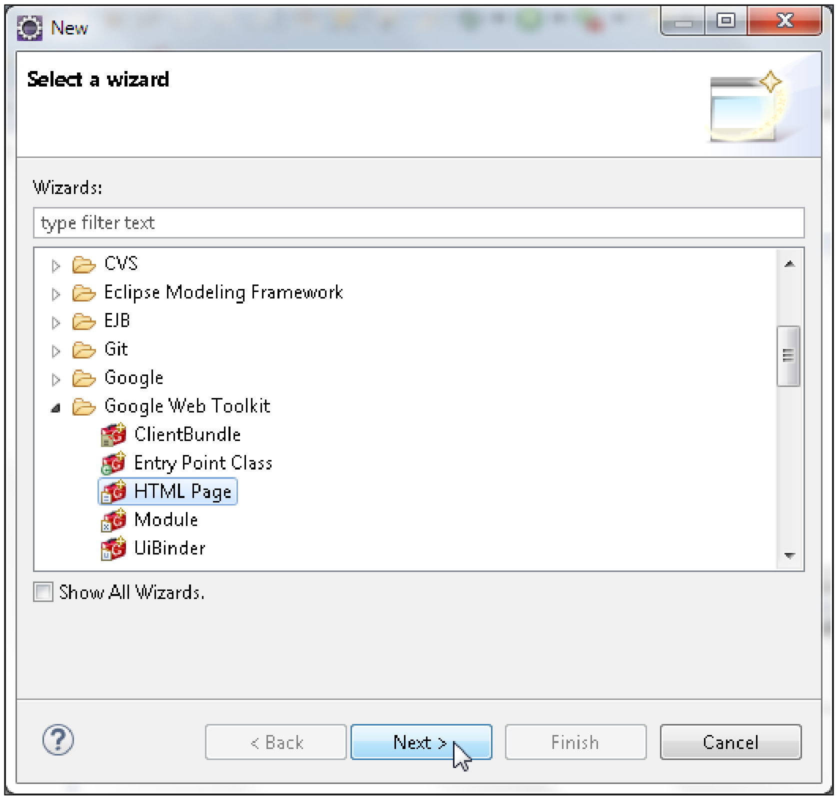
The entry-point class `org.jboss.gwt.client.CatalogForm` is listed in the following code:

```
package org.jboss.gwt.client;

import com.google.gwt.core.client.EntryPoint;

import com.google.gwt.user.client.ui.Button;
import com.google.gwt.user.client.ui.TextBox;
import com.google.gwt.event.dom.client.ClickEvent;
import com.google.gwt.event.dom.client.ClickHandler;
import com.google.gwt.event.dom.client.KeyUpEvent;
import com.google.gwt.event.dom.client.KeyUpHandler;
import com.google.gwt.user.client.ui.Label;
import com.google.gwt.user.client.ui.RootPanel;

import java.util.*;

import java.lang.Exception;

/**
* Entry point classes define <code>onModuleLoad()</code>.
*/
public class CatalogForm implements EntryPoint {

  /**
  * This is the entry point method.
  */

  ArrayList<String> arrayList;
  HashMap<String, ArrayList<String>> catalogHashMap;

  public void onModuleLoad() {

    final Button button = new Button("Submit");
    final Label label1 = new Label("Catalog ID");
    final Label label2 = new Label("Journal");
    final Label label3 = new Label("Publisher");
    final Label label4 = new Label("Edition");
    final Label label5 = new Label("Title");
    final Label label6 = new Label("Author");
    final Label label7 = new Label();

    final TextBox textBox1 = new TextBox();
    final TextBox textBox2 = new TextBox();
    final TextBox textBox3 = new TextBox();
    final TextBox textBox4 = new TextBox();
    final TextBox textBox5 = new TextBox();
    final TextBox textBox6 = new TextBox();

    arrayList = new ArrayList<String>();

    arrayList.add(0, "catalog1");
    arrayList.add(1, "Oracle Magazine");
    arrayList.add(2, "Oracle Publishing");
    arrayList.add(3, "May-June 2006");
    arrayList.add(4, "Tuning Your View Objects");
    arrayList.add(5, "Steve Muench");

    catalogHashMap = new HashMap<String, ArrayList<String>>();
    catalogHashMap.put("catalog1", arrayList);

    arrayList = new ArrayList<String>();

    arrayList.add(0, "catalog2");
    arrayList.add(1, "Oracle Magazine");
    arrayList.add(2, "Oracle Publishing");
    arrayList.add(3, "July-August 2006");
    arrayList.add(4, "Evolving Grid Management");
    arrayList.add(5, "David Baum");

    catalogHashMap.put("catalog2", arrayList);

    textBox1.addKeyUpHandler(new KeyUpHandler() {
      public void onKeyUp(KeyUpEvent event) {

        try {

          String catalogId = textBox1.getText();
          if (catalogHashMap.containsKey(catalogId)) {
            ArrayList<String> arraylist = (ArrayList<String>)
catalogHashMap
            .get(catalogId);
            label7.setText("Catalog Id is not Valid");
            textBox2.setText((String) arraylist.get(1));
            textBox3.setText((String) arraylist.get(2));
            textBox4.setText((String) arraylist.get(3));
```

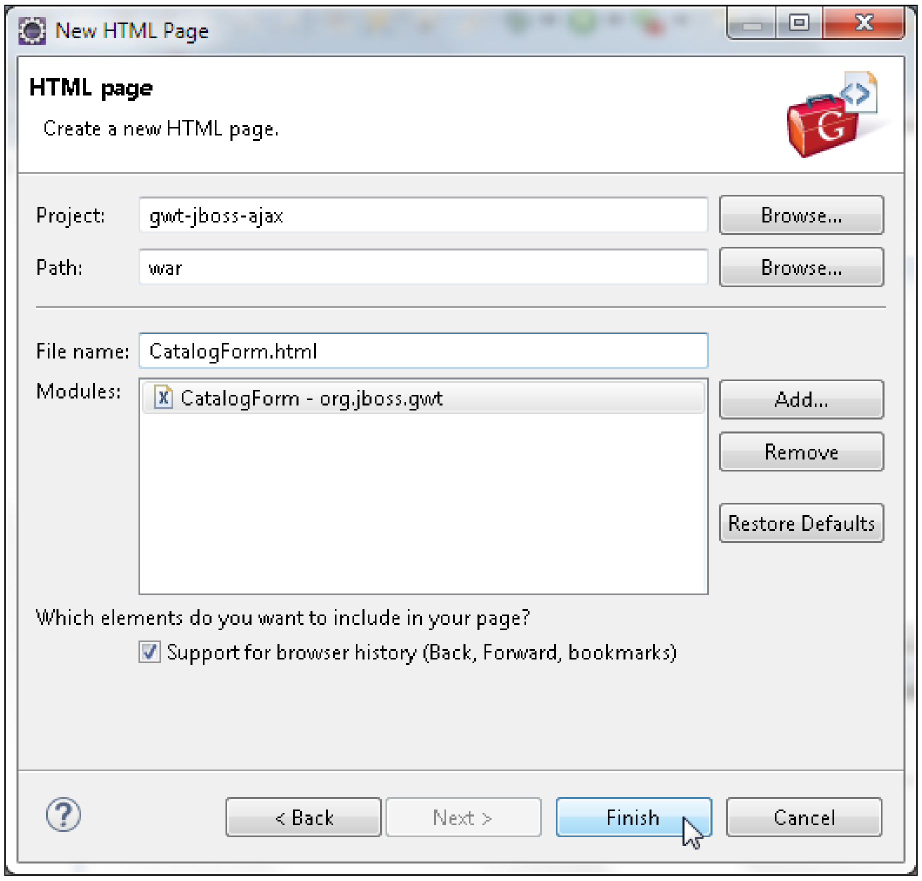
```
                textBox5.setText((String) arraylist.get(4));
                textBox6.setText((String) arraylist.get(5));
                button.setEnabled(false);

              }

              else {
                if (catalogId != "") {
                  label7.setText("Catalog Id is Valid");
                  textBox2.setText("");
                  textBox3.setText("");
                  textBox4.setText("");
                  textBox5.setText("");
                  textBox6.setText("");
                  button.setEnabled(true);
                }

              }

            } catch (Exception e) {
            }

          }
        });

        button.addClickHandler(new ClickHandler() {
          public void onClick(ClickEvent event) {

            String catalogId = textBox1.getText();
            arrayList = new ArrayList<String>();

            arrayList.add(0, catalogId);
            arrayList.add(1, textBox2.getText());
            arrayList.add(2, textBox3.getText());
            arrayList.add(3, textBox4.getText());
            arrayList.add(4, textBox5.getText());
            arrayList.add(5, textBox6.getText());
            catalogHashMap.put(catalogId, arrayList);

          }
        });

        RootPanel.get("label1").add(label1);
        RootPanel.get("label2").add(label2);
        RootPanel.get("label3").add(label3);
        RootPanel.get("label4").add(label4);
        RootPanel.get("label5").add(label5);
        RootPanel.get("label6").add(label6);
        RootPanel.get("textBox1").add(textBox1);
        RootPanel.get("textBox2").add(textBox2);
        RootPanel.get("textBox3").add(textBox3);
        RootPanel.get("textBox4").add(textBox4);
        RootPanel.get("textBox5").add(textBox5);
        RootPanel.get("textBox6").add(textBox6);
        RootPanel.get("button").add(button);
        RootPanel.get("label7").add(label7);
      }
    }
```

Creating an HTML host page

We created a GWT module and an entry-point class, which is associated with the module. To load the module and run the entry-point class, we need an HTML host page. The GWT compiler compiles a module into a JavaScript file `nocache.js`, also called the selection script, when we install the GWT project with Maven in a later section. The JavaScript file for the module is required to be included in an HTML host page in the `<script/>` tag. First, create an HTML page. Select **File** | **New** | **Other**, and in **New**, select **Google Web Toolkit** | **HTML Page** and click on **Next**, as shown in the following screenshot:



Select the `gwt-jboss-ajax` project and **Path** as `war` and specify **File Name** as `CatalogForm.html`. Select the `CatalogForm` module and click on **Finish**, as shown in the following screenshot:



Project: gwt-jboss-ajax
Path: war
File name: CatalogForm.html
Modules: CatalogForm - org.jboss.gwt

Which elements do you want to include in your page?
[✓] Support for browser history (Back, Forward, bookmarks)

A `CatalogForm.html` HTML page gets created in the `war` directory. The default `<script/>` tag generated has `src=".nocache.js"`, which is not the JavaScript generated from the `CatalogForm` module. To accomplish this, use the following code:

```
<!doctype html>
<html>
  <head>
    <meta http-equiv="content-type" content="text/html;
charset=UTF-8">
    <title>CatalogForm</title>
    <script type="text/javascript" language="javascript" src=".
nocache.js"></script>
  </head>
  <body>
    <iframe src="javascript:''" id="__gwt_historyFrame" tabIndex='-1'
style="position:absolute;width:0;height:0;border:0"></iframe>
  </body>
</html>
```

The following are the steps to create an HTML host page:

1. Set the `script` tag as follows:

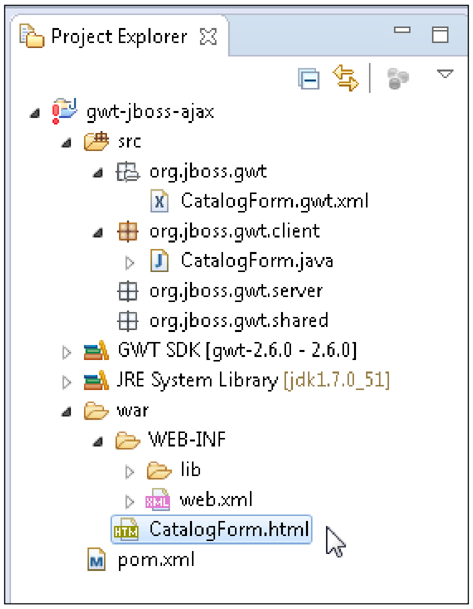
   ```
   <script type="text/javascript" language="javascript" src="org.
   jboss.gwt.CatalogForm/org.jboss.gwt.CatalogForm.nocache.js">
   ```

2. Add a `<table/>` element below the `<iframe/>` element for a catalog entry. The `<table/>` element has `<div/>` elements for the GWT widgets created in the entry-point class. The `<div/>` elements have associated IDs, which are used in the entry-point class `CatalogForm.java` to add the widgets to a widget-specific root panel. Add the `Label` widgets and the corresponding `TextBox` widgets in the same row. The `Label` widget with the `<div/>` ID `label7` is the widget to display a validation message. The `CatalogForm.html` HTML host page is listed in the following code:

   ```
   <!doctype html>
   <html>
   <head>
       <meta name="generator" content="HTML Tidy for Linux/x86 (vers
   25 March 2009), see www.w3.org" />
       <meta http-equiv="content-type" content="text/html;
   charset=us-ascii" />
       <title>
         CatalogForm
       </title>
       <script type="text/javascript" language="javascript" src="org.
   jboss.gwt.CatalogForm/org.jboss.gwt.CatalogForm.nocache.js">
   </script>
     </head>
     <body>
       <iframe src="javascript:''" id="__gwt_historyFrame"
   tabindex='-1' style="position: absolute; width: 0; height: 0;
   border: 0"></iframe>
       <h1>
         Catalog Form
       </h1>
       <table align="left">
         <tr>
           <td id="label1"></td>
           <td id="textBox1"></td>
         </tr>
         <tr>
           <td id="label2"></td>
   ```

```
            <td id="textBox2"></td>
          </tr>
          <tr>
            <td id="label3"></td>
            <td id="textBox3"></td>
          </tr>
          <tr>
            <td id="label4"></td>
            <td id="textBox4"></td>
          </tr>
          <tr>
            <td id="label5"></td>
            <td id="textBox5"></td>
          </tr>
          <tr>
            <td id="label6"></td>
            <td id="textBox6"></td>
          </tr>
          <tr>
            <td id="button"></td>
            <td id="label7"></td>
          </tr>
        </table>
      </body>
    </html>
```

3. The HTML host page is shown in the `gwt-jboss-ajax` GWT web project in **Project Explorer**, as shown in the following screenshot:



Deploying the GWT project with Maven

In this section, we will compile, package, and deploy the GWT web project to WildFly 8.1 with the Maven build tool. The `pom.xml` file is similar to the one used for the project generated on the command line and imported into Eclipse. The **Errai** (a GWT-based framework) dependencies are also included. The packaging for `gwt-jboss-ajax artifactId` is `war`. The GWT version should be `2.6.0`. Refer to the following code to accomplish this:

```
<properties>
  <version.com.google.gwt>2.6.0</version.com.google.gwt>
</properties>
```

Specify the output directory for the Maven WAR plugin configuration as the WildFly 8.1 `deployments` directory, as follows:

```
<outputDirectory>C:\wildfly-8.1.0.Final\standalone\deployments</
outputDirectory>
```

The GWT user library for GWT widgets and panels and support for Ajax is a required dependency with `scope` as `compile`. Refer to the following code for this:

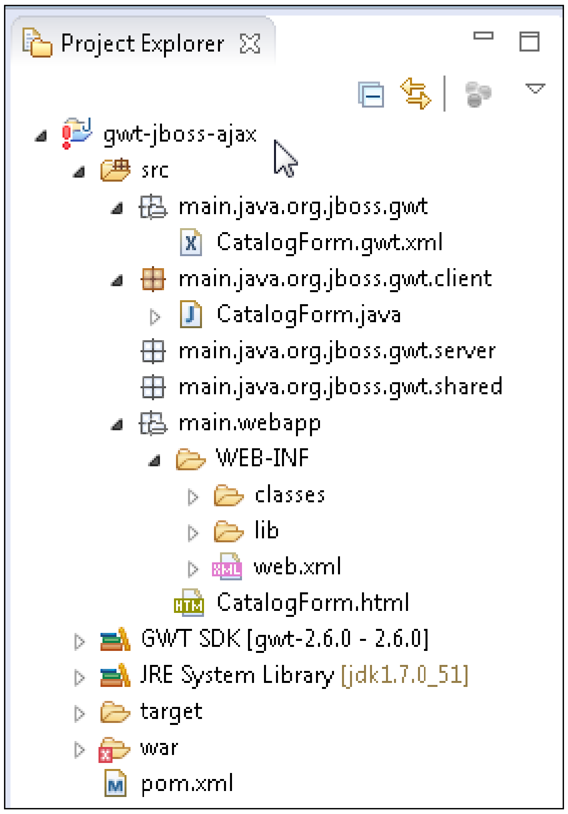
```
<dependency>
  <groupId>com.google.gwt</groupId>
  <artifactId>gwt-user</artifactId>
  <scope>compile</scope>
</dependency>
```

The GWT development dependency is required as it supports the Java-to-JavaScript compiler with `scope` as `provided`. Refer to the following code for this dependency:

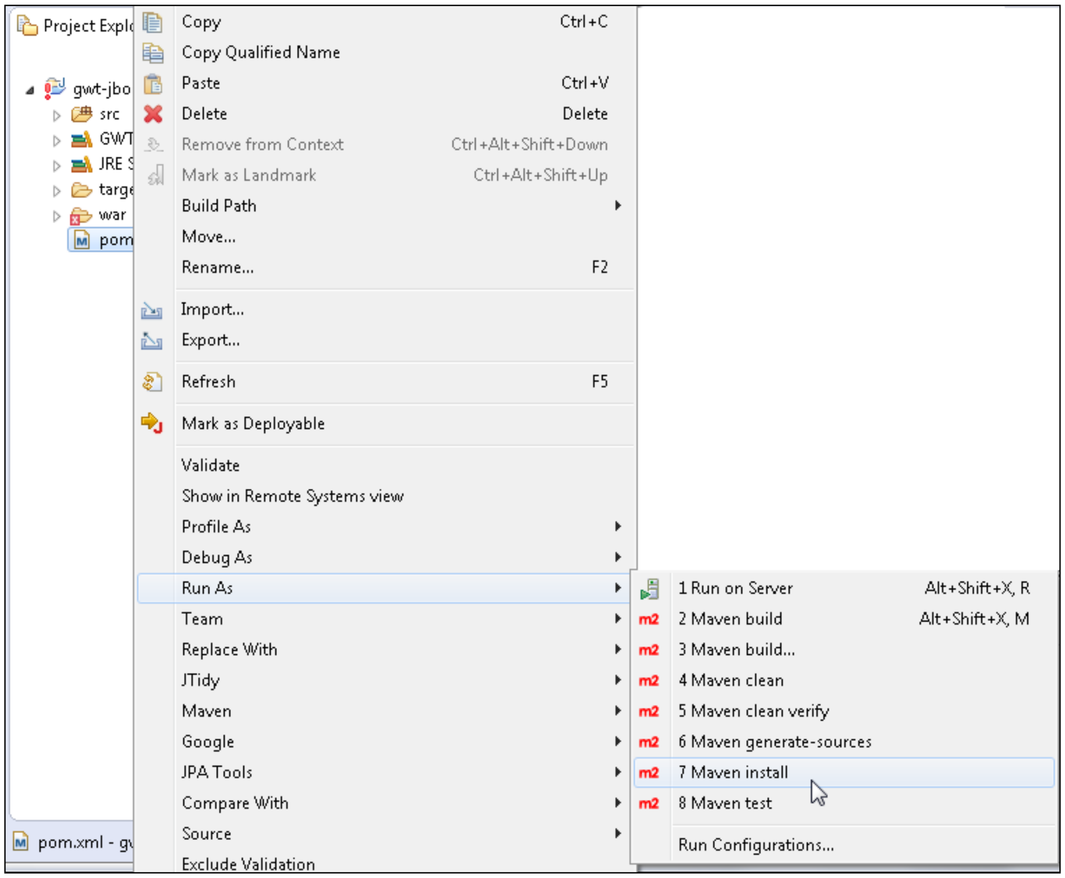
```
<dependency>
  <groupId>com.google.gwt</groupId>
  <artifactId>gwt-dev</artifactId>
  <scope>provided</scope>
</dependency>
```

GWT validation requires the Hibernate Validator and the Validation API. In addition to the Maven compiler plugin and the Maven WAR plugin, the GWT plugin is required to compile the client-side Java to JavaScript.

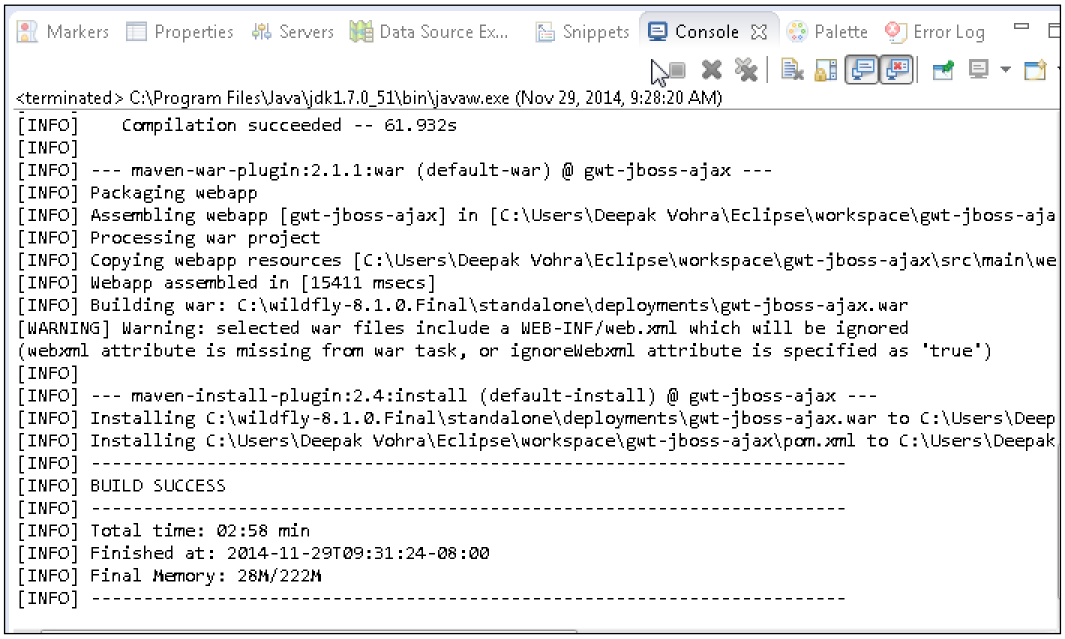
Not all the dependencies included in `pom.xml` are used in the example application of this chapter. Before we can run `pom.xml`, we need to modify the directory structure of the GWT project to the standard directory layout for a Maven project with the `src/main/java` directory for the application Java sources and the `src/main/webapp` directory for the web application sources, as shown in the following screenshot. Create the `src/main/java` and `src/main/webapp` directories and copy the Java source code to the `src/main/java` directory and the web application source code to the `src/main/webapp` directory.



Next, run the Maven build tool. Right-click on `pom.xml` and select **Run As** | **Maven install**, as shown in the following screenshot:

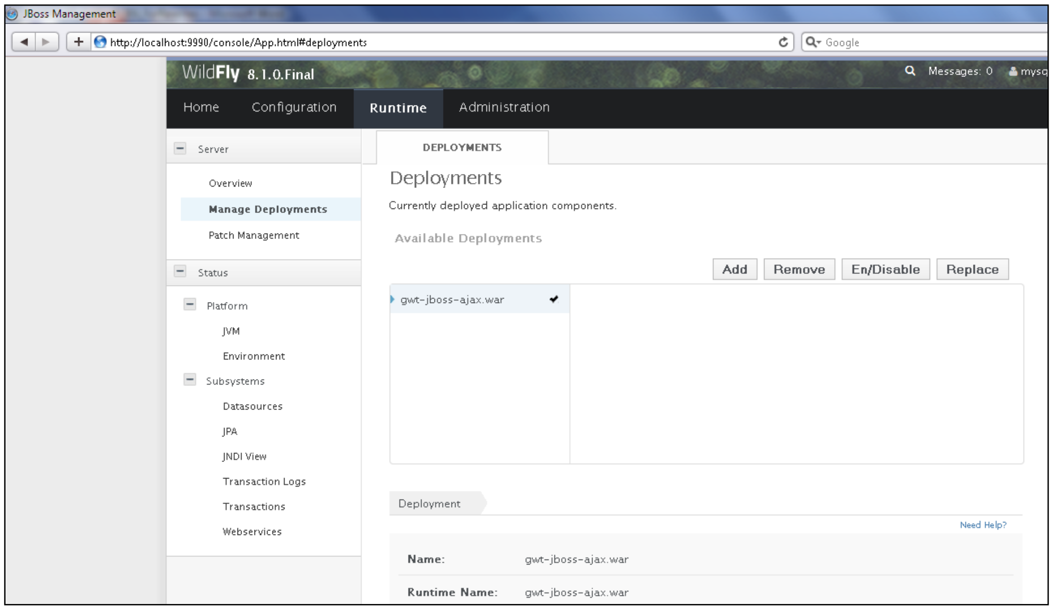


The GWT project gets compiled and packaged to `gwt-jboss-ajax.war`, as shown in the following screenshot. The compiled JavaScript file `org.jboss.gwt.CatalogForm.nocache.js` gets added to the `org.jboss.gwt.CatalogForm` directory, which is included in the relative path in the `script` tag in the `CatalogForm.html` hosted page:



```
<terminated> C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Nov 29, 2014, 9:28:20 AM)
[INFO]    Compilation succeeded -- 61.932s
[INFO]
[INFO] --- maven-war-plugin:2.1.1:war (default-war) @ gwt-jboss-ajax ---
[INFO] Packaging webapp
[INFO] Assembling webapp [gwt-jboss-ajax] in [C:\Users\Deepak Vohra\Eclipse\workspace\gwt-jboss-aja
[INFO] Processing war project
[INFO] Copying webapp resources [C:\Users\Deepak Vohra\Eclipse\workspace\gwt-jboss-ajax\src\main\we
[INFO] Webapp assembled in [15411 msecs]
[INFO] Building war: C:\wildfly-8.1.0.Final\standalone\deployments\gwt-jboss-ajax.war
[WARNING] Warning: selected war files include a WEB-INF/web.xml which will be ignored
(webxml attribute is missing from war task, or ignoreWebxml attribute is specified as 'true')
[INFO]
[INFO] --- maven-install-plugin:2.4:install (default-install) @ gwt-jboss-ajax ---
[INFO] Installing C:\wildfly-8.1.0.Final\standalone\deployments\gwt-jboss-ajax.war to C:\Users\Deep
[INFO] Installing C:\Users\Deepak Vohra\Eclipse\workspace\gwt-jboss-ajax\pom.xml to C:\Users\Deepak
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 02:58 min
[INFO] Finished at: 2014-11-29T09:31:24-08:00
[INFO] Final Memory: 28M/222M
[INFO] ------------------------------------------------------------------------
```

Start the WildFly 8.1 server. The `gwt-jboss-ajax.war` gets deployed to the server. The `gwt-jboss-ajax.war` is shown deployed in the **Administration Console** of WildFly in the following screenshot:



Deployments

Currently deployed application components.

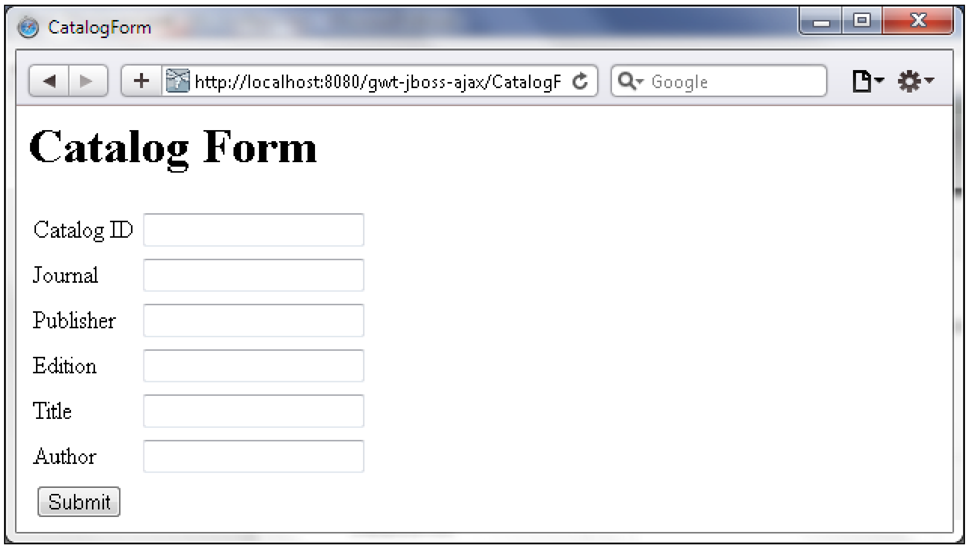
Available Deployments

gwt-jboss-ajax.war ✓

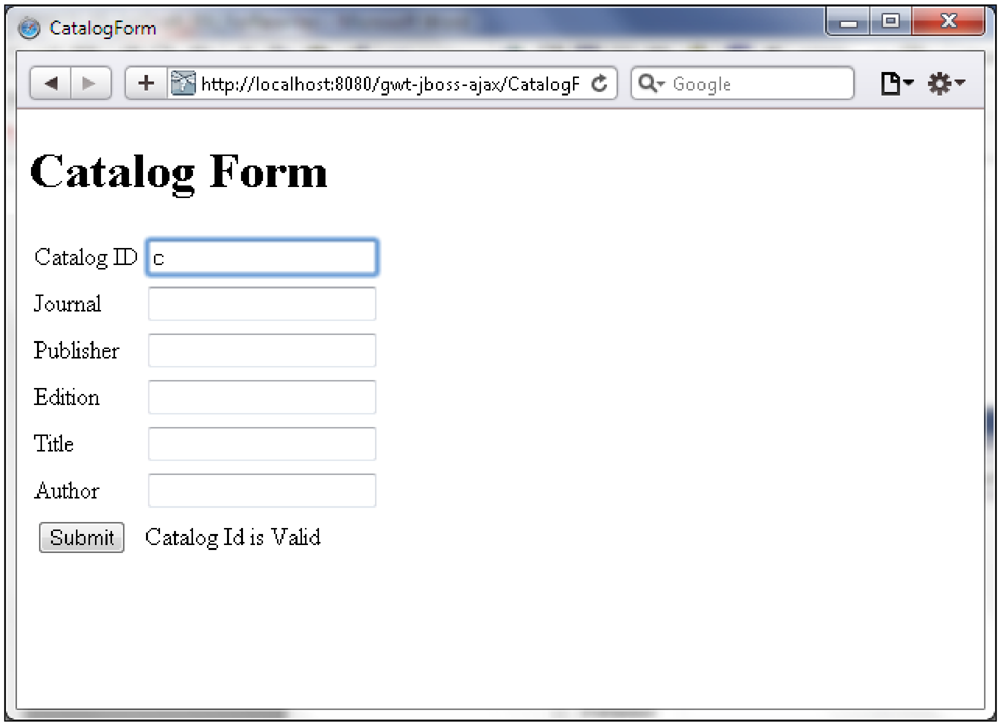
Name:	gwt-jboss-ajax.war
Runtime Name:	gwt-jboss-ajax.war

Running the GWT project

In this section, we will run the GWT web project. Run the GWT HTML hosted page `CatalgForm.html` in a browser with the URL `http://localhost:8080/gwt-jboss-ajax/CatalogForm.html`. The GWT widget-generated input form gets displayed, as shown in the following screenshot:



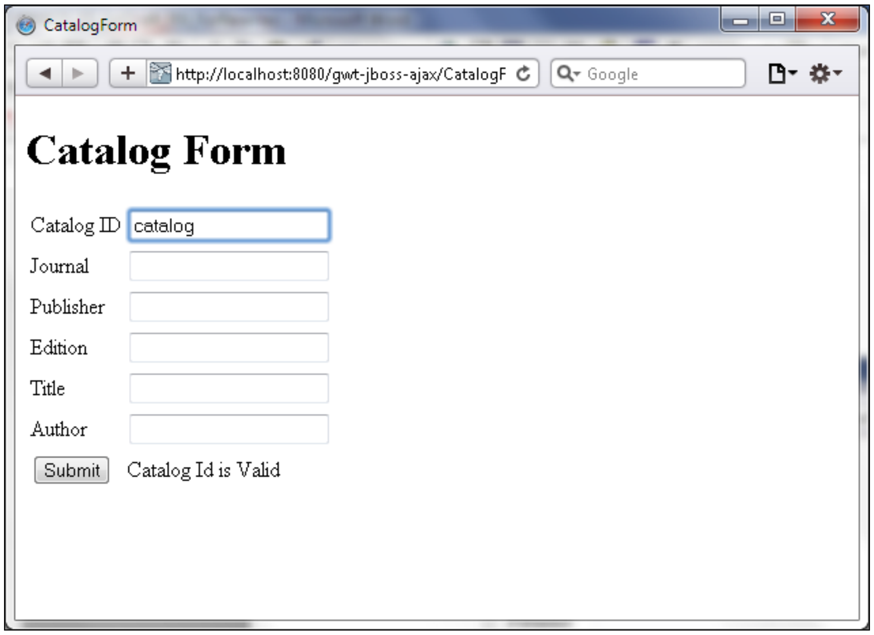
Start to specify a **Catalog ID** value. As the `TextBox` widget supports Ajax, a validation message gets displayed as shown in the following screenshot to indicate whether the **Catalog ID** value is valid according to the logic that the value is not an empty string and is not already in `HashMap` for the catalog.



Catalog Form

Catalog ID	c
Journal	
Publisher	
Edition	
Title	
Author	
Submit	Catalog Id is Valid

An Ajax request is sent with each key-up event as the Catalog ID `TextBox` widget is registered with the `com.google.gwt.event.dom.client.KeyUpHandler` event handler. A value of `catalog` for **Catalog ID** is still valid, as shown in the following screenshot:

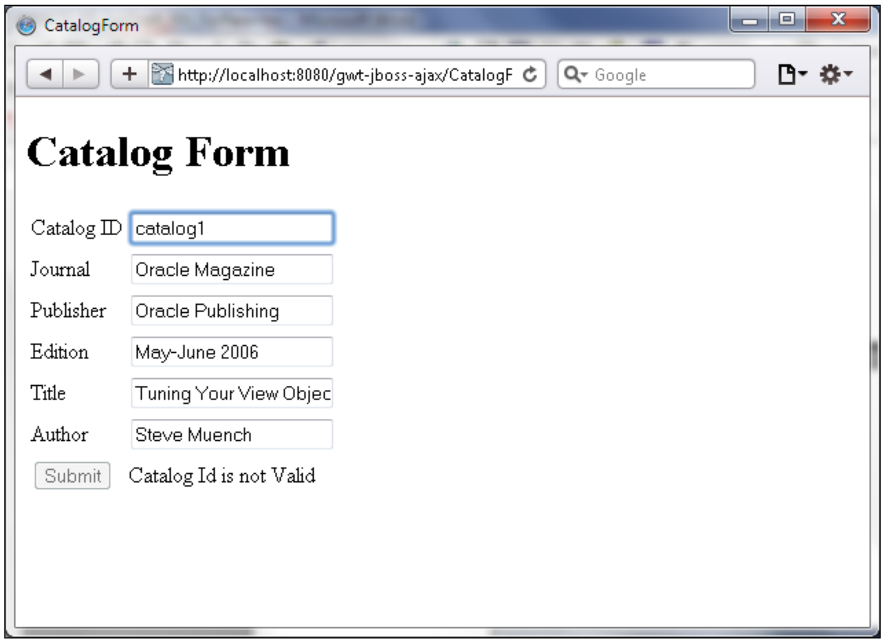


Catalog Form

Catalog ID	catalog
Journal	
Publisher	
Edition	
Title	
Author	

Catalog Id is Valid

Specify a value that is already in `HashMap`, such as **catalog1**. A validation message, **Catalog ID is not Valid**, gets displayed, as shown in the following screenshot. The input fields get filled and the **Submit** button gets disabled:

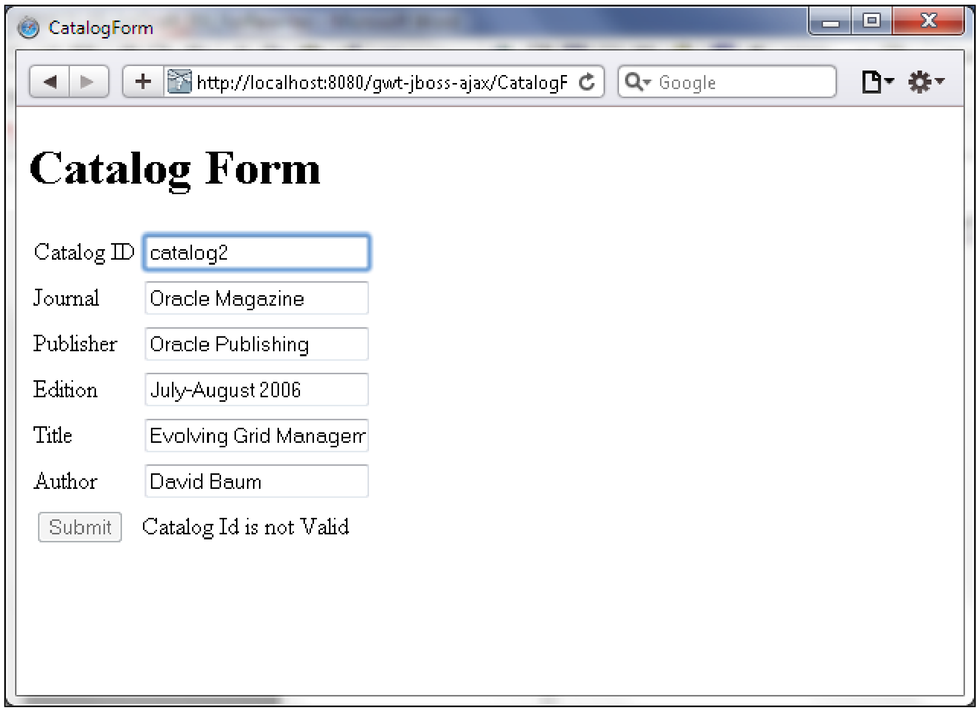


Catalog Form

Catalog ID	catalog1
Journal	Oracle Magazine
Publisher	Oracle Publishing
Edition	May-June 2006
Title	Tuning Your View Objec
Author	Steve Muench

Catalog Id is not Valid

The **Catalog ID** value `catalog2` is also not a valid value, as shown in the following screenshot:

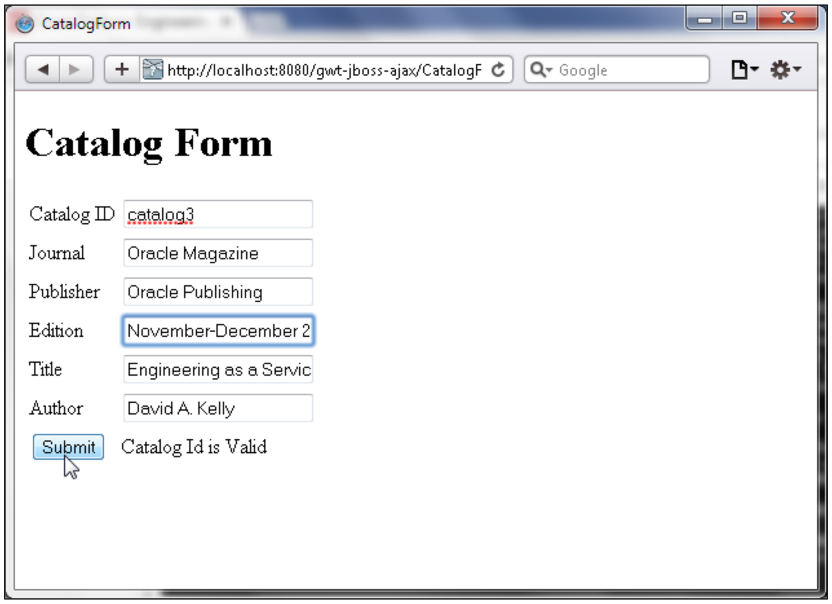


Catalog Form

Catalog ID	catalog2
Journal	Oracle Magazine
Publisher	Oracle Publishing
Edition	July-August 2006
Title	Evolving Grid Managem
Author	David Baum

Submit Catalog Id is not Valid

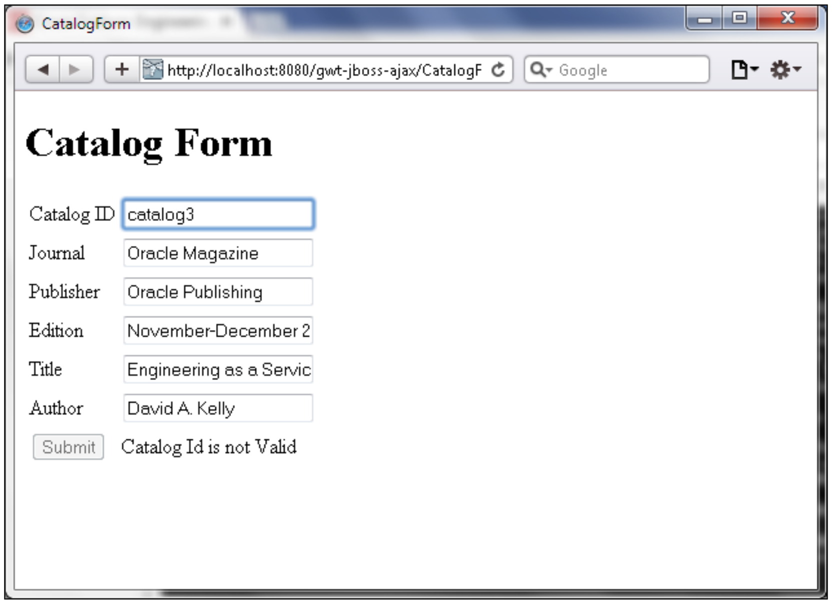
The **Catalog ID** value `catalog3` is a valid value to create a new catalog entry. Specify input field values and click on the **Submit** button to create a new catalog entry, as shown in the following screenshot:



Catalog Form

Catalog ID	catalog3
Journal	Oracle Magazine
Publisher	Oracle Publishing
Edition	November-December 2
Title	Engineering as a Servic
Author	David A. Kelly
Submit	Catalog Id is Valid

A new catalog entry gets added to `HashMap`. A catalog entry with **Catalog ID** `catalog3` gets created. After the user clicks on **Submit**, the user does not receive any notification that the data has been stored. To verify that the data has been stored, the user must clear **Catalog ID** first and re-add **Catalog ID**. If `catalog3` is respecified as **Catalog ID**, `catalog3` is considered as invalid, as shown in the following screenshot:



Catalog Form

Catalog ID	catalog3
Journal	Oracle Magazine
Publisher	Oracle Publishing
Edition	November-December 2
Title	Engineering as a Servic
Author	David A. Kelly
Submit	Catalog Id is not Valid

Summary

GWT compiles client Java into optimized JavaScript, which can be run in a browser. In this chapter, we tested a starter GWT project on WildFly 8.1. We also developed a new GWT web project. We added a module to the GWT project to configure an entry-point class and add the GWT User library. We created an entry-point class, a class that implements the `com.google.gwt.core.client.EntryPoint` interface. We created an HTML hosted page to specify the compiled module as a JavaScript file. We ran the hosted page in a browser to validate and create a new catalog entry.

In the next chapter, we will discuss creating a JAX-WS 2.2 Web Service in Eclipse, building and deploying the application using Maven to WildFly 8.1, and running the web service in WildFly 8.1.

6
Developing a JAX-WS 2.2 Web Service

Java API for XML-based Web Services, known as JAX-WS (`http://jcp.org/aboutJava/communityprocess/mrel/jsr224/index2.html`), is a W3C standards-based technology for communicating between services and clients using XML over the HTTP protocol. Some of the W3C standards that JAX-WS Web Services support are HTTP (`http://www.w3.org/Protocols/`), SOAP (`http://www.w3.org/TR/soap/`), and **Web Service Description Language** (**WSDL**) (`http://www.w3.org/TR/wsdl`). JAX-WS is a platform-independent standard; JAX-WS Web Services may communicate with non-Java clients, for example a .NET client, and a JAX-WS client may communicate with non-Java Web Services, a .NET Web Service for example. JAX-WS makes use of XML in the following artifacts:

- The WSDL is an XML document that describes network services as a set of endpoints operating on messages; an endpoint being a URL location representing a web service
- A client and a web service communicate using SOAP messages, which are in the XML format

A JAX-WS web service consists of the following components:

- A non-final, non-abstract Java class that is annotated with `javax.jws.WebService` annotation. The web service endpoints, the Java class consists of business methods. A web service client creates a proxy of the web service to invoke its methods (operations). The web service optionally returns a response. The web service request and response are SOAP messages over HTTP.
- An optional Java interface that defines the methods implemented in the web service implementation class.

This chapter has the following sections:

- Setting up the environment
- Creating a Java EE web project
- Creating a web descriptor
- Creating a JAX-WS Web Service
- Creating a web service client
- Deploying the JAX-WS application with Maven
- Running the JAX-WS application

Setting up the environment

We need to download and install the following software:

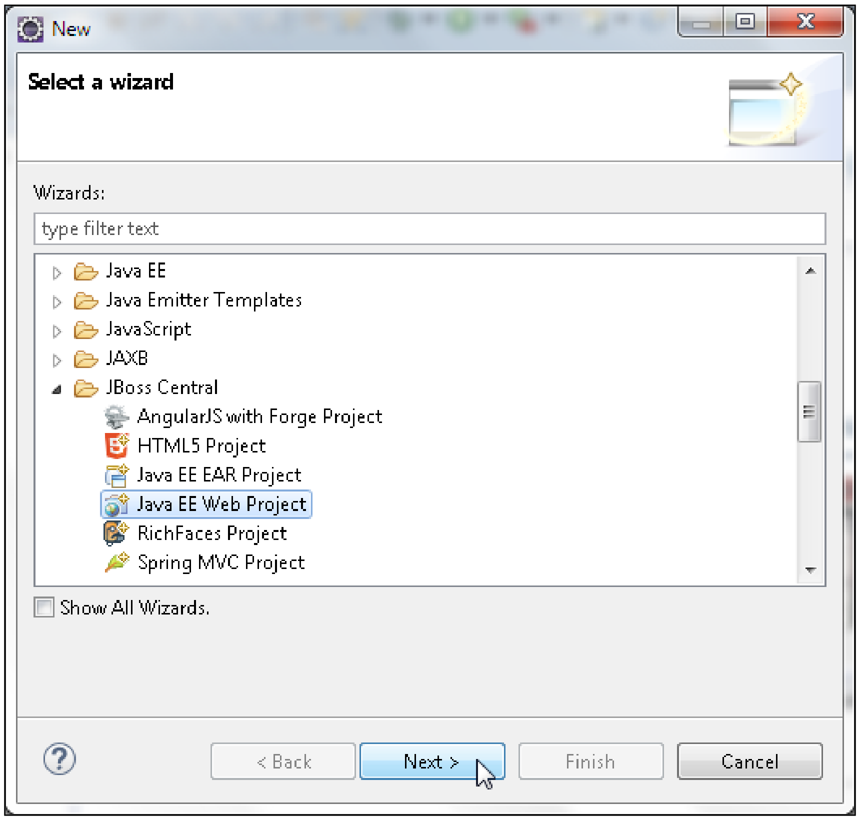
- **WildFly 8.1.0.Final**: Download `wildfly-8.1.0.Final.zip` from `http://wildfly.org/downloads/`.
- **MySQL 5.6 Database-Community Edition**: Download this edition from `http://dev.mysql.com/downloads/mysql/`. When installing MySQL, install **Connector/J** too.
- **Eclipse IDE for Java EE Developers**: Download Eclipse Luna from `https://www.eclipse.org/downloads/packages/release/Luna/SR1`.
- **JBoss Tools (Luna) 4.2.0.Final (or the latest version)**: Install this as a plugin to Eclipse from the Eclipse Marketplace (`http://tools.jboss.org/downloads/installation.html`).
- **Apache Maven**: Download version 3.05 or higher from `http://maven.apache.org/download.cgi`.
- **Java 7**: Download Java 7 from `http://www.oracle.com/technetwork/java/javase/downloads/index.html?ssSourceSiteId=ocomcn`.

Set the environment variables, `JAVA_HOME`, `JBOSS_HOME`, and `MAVEN_HOME`. Add `%JAVA_HOME%/bin`, `%MAVEN_HOME%/bin`, and `%JBOSS_HOME%/bin` to the `PATH` environment variable.

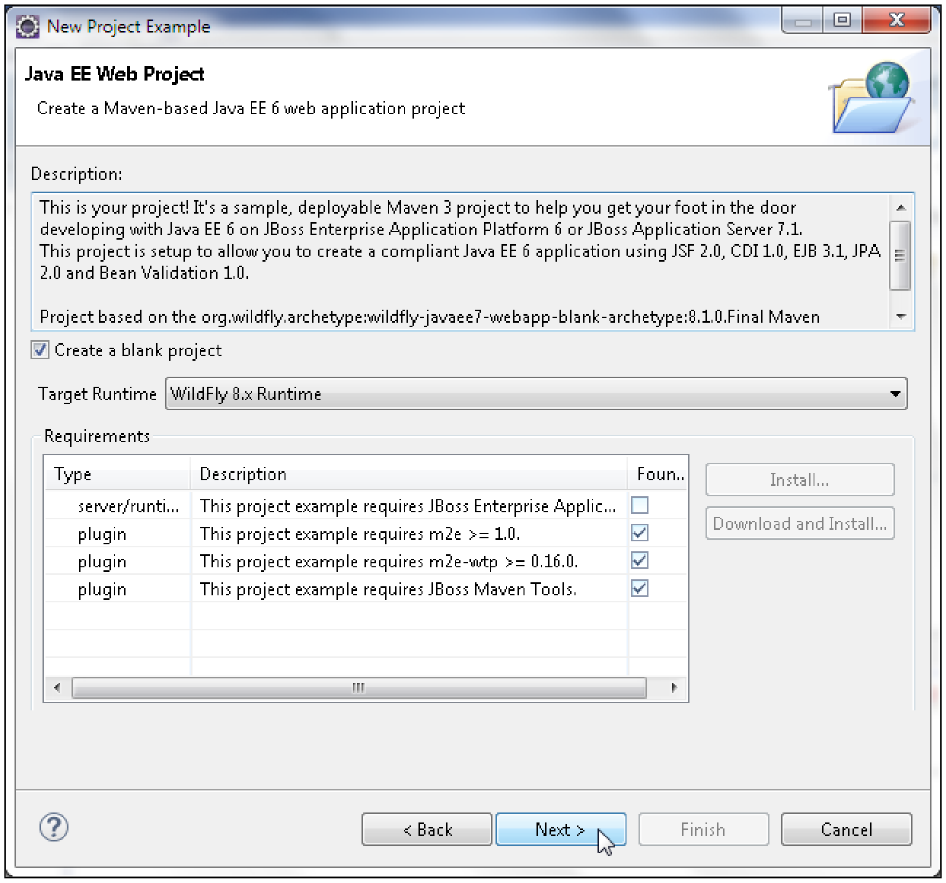
Create a WildFly 8.1.0 runtime as discussed in *Chapter 1, Getting Started with EJB 3.x*.

Creating a Java EE web project

In this section, we will create a Java EE Web Project for a JAX-WS Web Service. Select **File** | **New**. In **New**, select **JBoss Central** | **Java EE Web Project**, as shown in the following screenshot. Now click on **Next**.



A test is run for the requirements, which include **m2e** and **m2eclipse-wtp** plugins, and the **JBoss Maven Tools** plugin. Select the checkbox and create a blank project; select **WildFly 8.x Runtime** for **Target Runtime**, as shown in the following screenshot. Now click on **Next**.



Java EE Web Project

Create a Maven-based Java EE 6 web application project

Description:

This is your project! It's a sample, deployable Maven 3 project to help you get your foot in the door developing with Java EE 6 on JBoss Enterprise Application Platform 6 or JBoss Application Server 7.1.
This project is setup to allow you to create a compliant Java EE 6 application using JSF 2.0, CDI 1.0, EJB 3.1, JPA 2.0 and Bean Validation 1.0.

Project based on the org.wildfly.archetype:wildfly-javaee7-webapp-blank-archetype:8.1.0.Final Maven

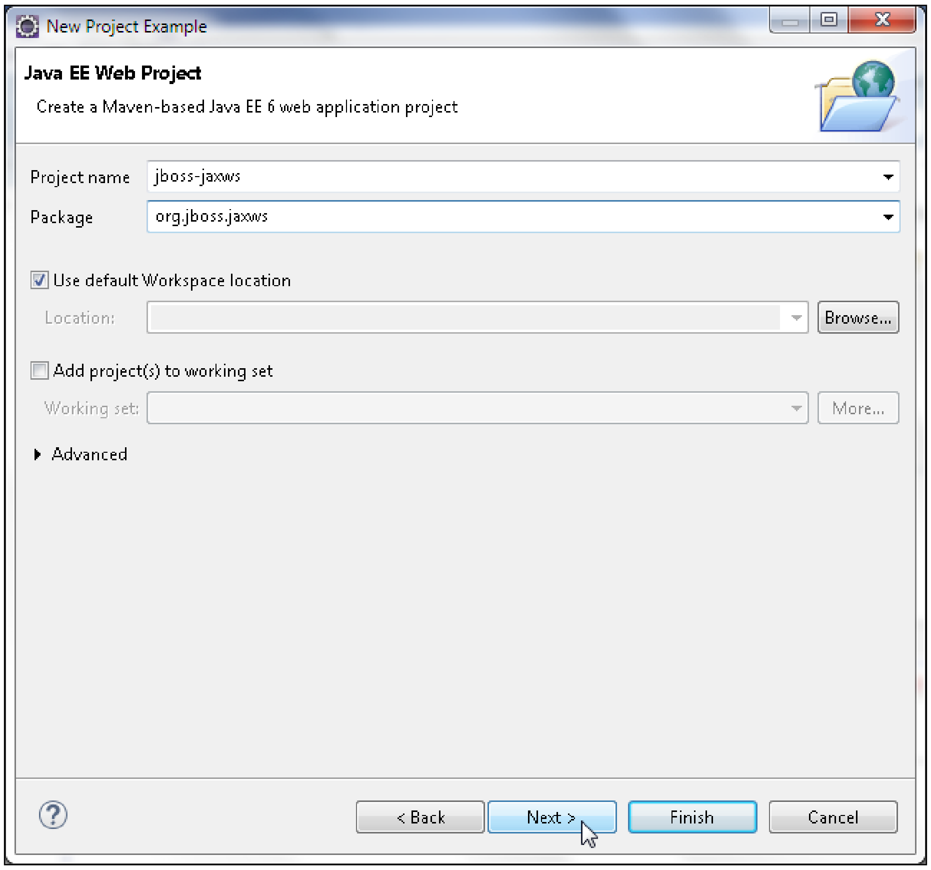
[✓] Create a blank project

Target Runtime: WildFly 8.x Runtime

Requirements

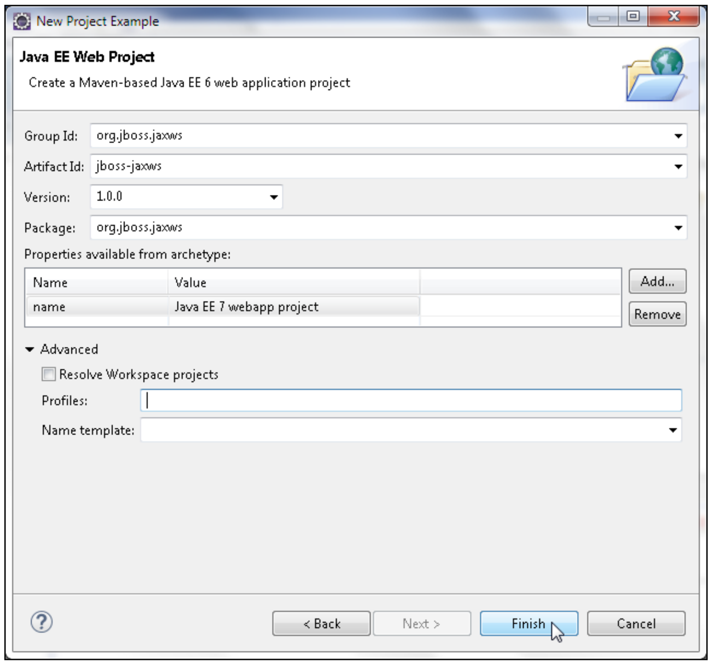
Type	Description	Foun..
server/runti...	This project example requires JBoss Enterprise Applic...	☐
plugin	This project example requires m2e >= 1.0.	✓
plugin	This project example requires m2e-wtp >= 0.16.0.	✓
plugin	This project example requires JBoss Maven Tools.	✓

Specify **Project name** (`jboss-jaxws`) and **Package** (`org.jboss.jaxws`), as shown in the following screenshot. Now click on **Next**.



Project name: jboss-jaxws
Package: org.jboss.jaxws
[✓] Use default Workspace location
[ ] Add project(s) to working set

Specify **Group Id** as `org.jboss.jaxws`, **Artifact Id** (`jboss-jaxws`), **Version** (`1.0.0`), and **Package** (`org.jboss.jaxws`), as shown in the following screenshot. Then click on **Finish**.



Java EE Web Project

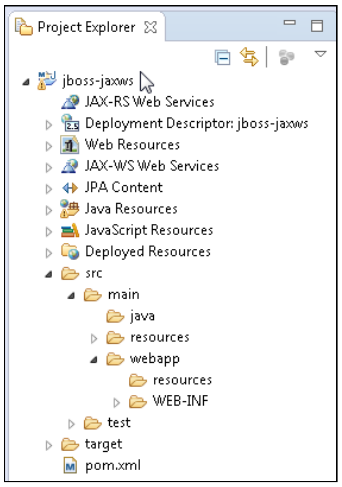
Create a Maven-based Java EE 6 web application project

Group Id: org.jboss.jaxws
Artifact Id: jboss-jaxws
Version: 1.0.0
Package: org.jboss.jaxws

Properties available from archetype:

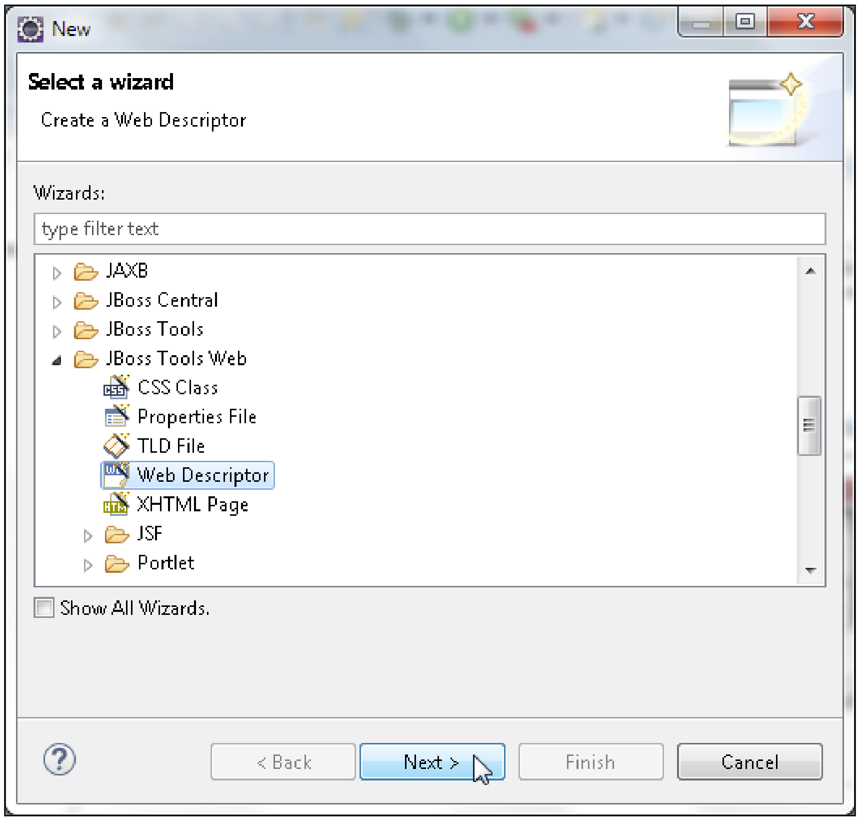
Name	Value
name	Java EE 7 webapp project

A Maven-based Java EE Web Project, `jboss-jaxws`, is generated, as shown in **Project Explorer** in the following screenshot. Delete the `jboss-jaxws/src/main/resources/META-INF/persistence.xml` configuration file:

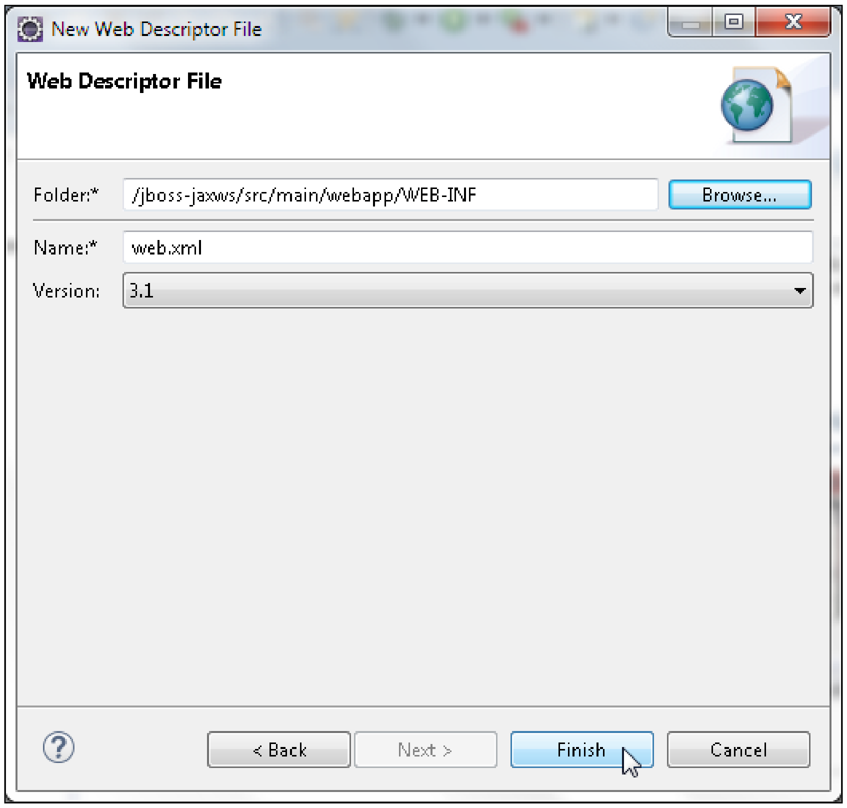


Creating a web descriptor

A JAX-WS Web Service requires a Web Descriptor. In this section, we will create a web descriptor. Select **File** | **New** | **Other**. In **New**, select **JBoss Tools Web** | **Web Descriptor** and click on **Next**, as shown in the following screenshot:



Click on **Browse** for the **Folder** field to select the `webapp/WEB-INF` folder. Specify **Name** as `web.xml`, select **Version** as **3.1**, and click on **Finish**, as shown in the following screenshot:



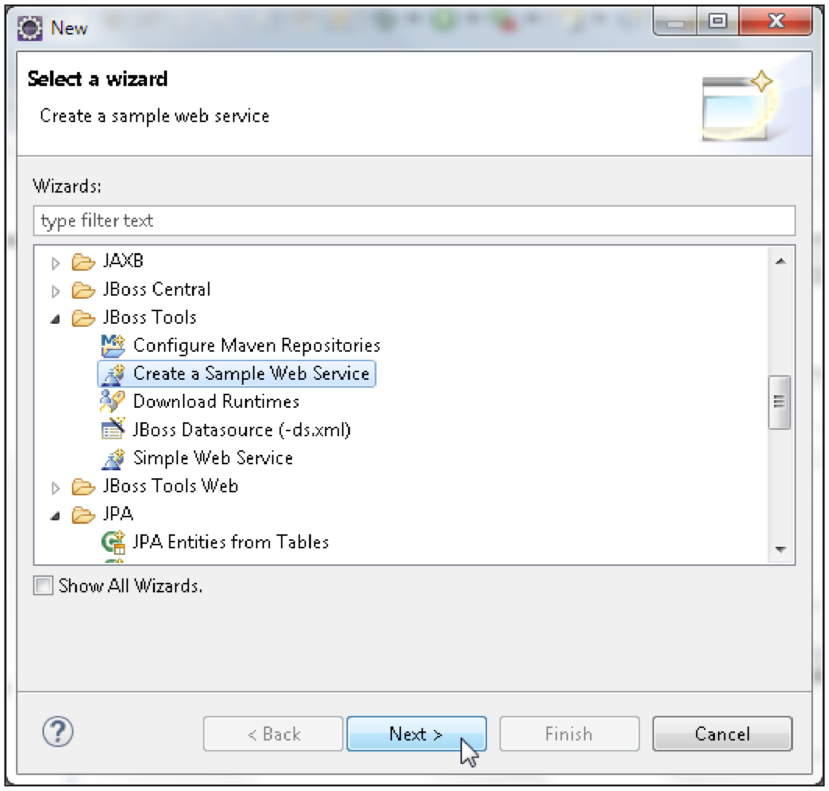
Folder:* /jboss-jaxws/src/main/webapp/WEB-INF

Name:* web.xml

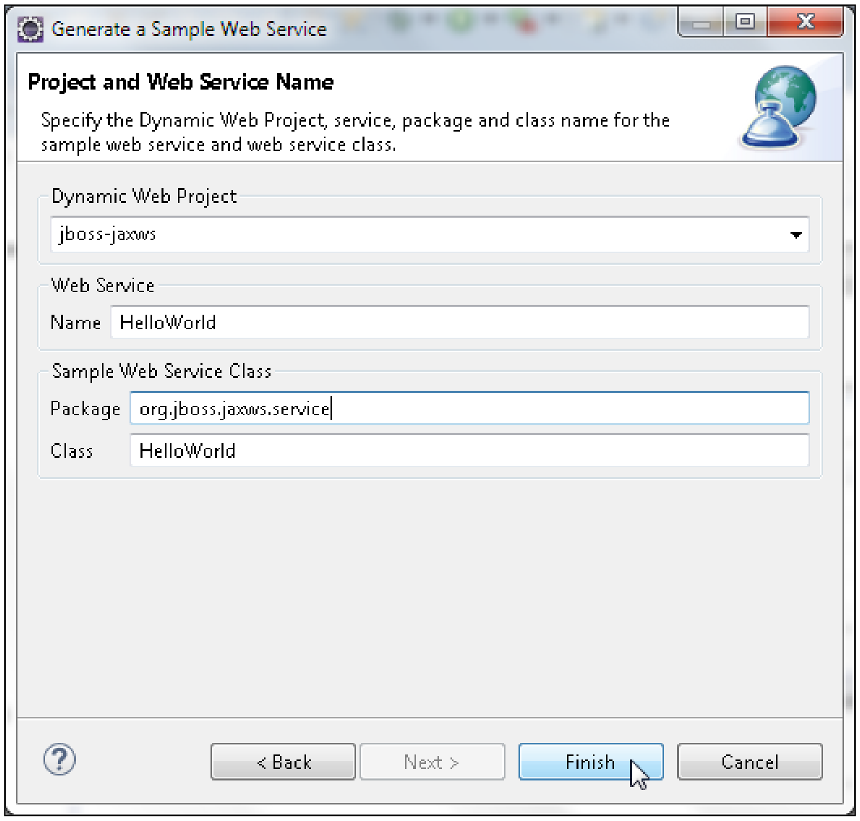
Version: 3.1

Creating a JAX-WS web service

In this section, we will create a JAX-WS Web Service in the Java EE Web Project. Select **File** | **New** | **Other**. In **New**, select **JBoss Tools** | **Create a Sample Web Service**, as shown in the following screenshot. Now click on **Next**.



In **Generate a Sample Web Service**, specify **Project and Web Service Name**. Select the `jboss-jaxws` web project and specify **Name** of **Web Service** as `HelloWorld`. For **Sample Web Service Class**, specify **Package** as `org.jboss.jaxws.service` and **Class** as `HelloWorld`, as shown in the following screenshot. Now click on **Finish**.



Project and Web Service Name

Specify the Dynamic Web Project, service, package and class name for the sample web service and web service class.

Dynamic Web Project: jboss-jaxws

Web Service

Name: HelloWorld

Sample Web Service Class

Package: org.jboss.jaxws.service

Class: HelloWorld

The `HelloWorld` class gets created. Annotate the class with the `@WebService` annotation, which indicates that the class implements a web service. Add the following elements in the `@WebService` annotation:

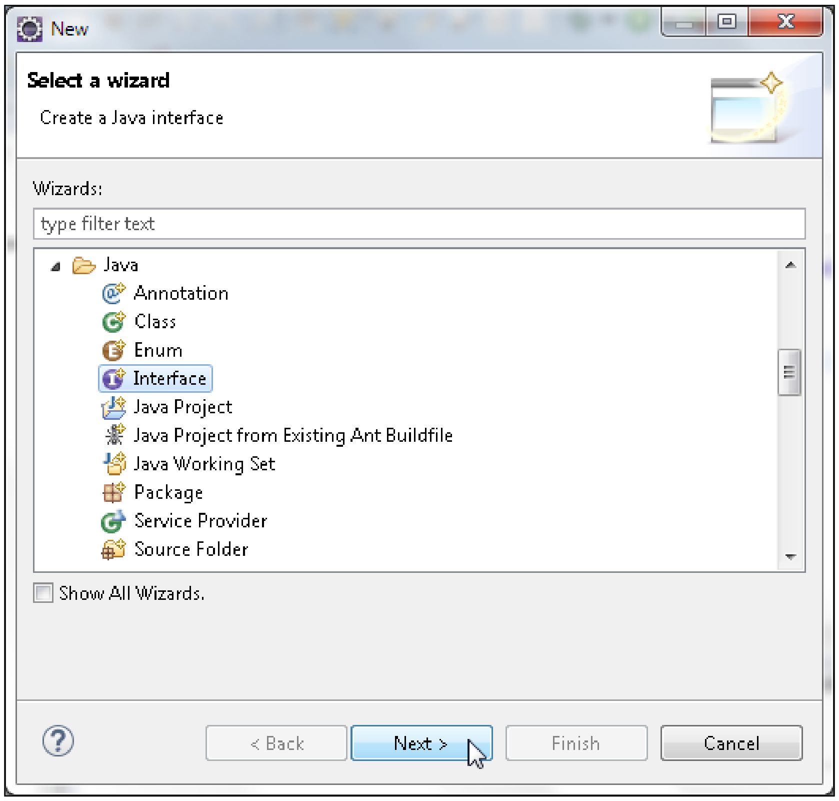
Element	Description	Value
`portName`	This is the port name of the web service. This is also `wsdl:portname` in the web service WSDL.	`HelloWorldPort`

Element	Description	Value
`serviceName`	This is the service name of the web service. This is also the `ws:servicename` in the WSDL.	`HelloWorldService`
`targetNamespace`	This is the target namespace for the `wsdl:service` in the WSDL.	`http://org.jboss.jaxws.service/`
`endpointInterface`	This is the name of the service endpoint interface defining the web service methods.	`org.jboss.jaxws.service.HelloWS`

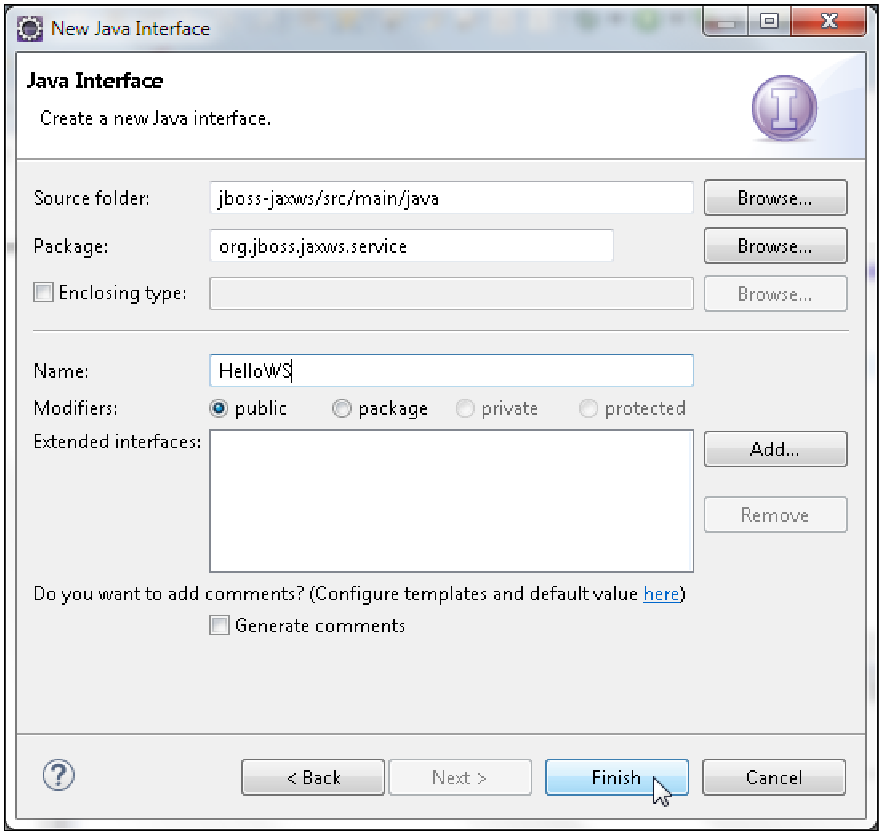
The `HelloWorld` implementation class implements the `HelloWS` interface. Add a business method `sayHello(String)` that takes a `String` name as argument and returns a `String` message. The business method is annotated with the `@WebMethod` annotation. A business method must not be `static` or `final`. The business methods are exposed to a client as web service operations. The web service endpoint Java class is listed here:

```
package org.jboss.jaxws.service;

import javax.jws.WebMethod;
import javax.jws.WebService;

@WebService(portName = "HelloWorldPort",     serviceName =
"HelloWorldService", targetNamespace =
"http://org.jboss.jaxws.service/", endpointInterface =
"org.jboss.jaxws.service.HelloWS")
public class HelloWorld implements HelloWS{

  @WebMethod ()
  public String sayHello(String name) {

      return "Hello "+name +" Welcome to Web Services!";

  }
}
```

As the endpoint implements an endpoint interface, create a Java interface. Select **File | New | Other**. In **New**, select **Interface** and click on **Next**, as shown in the following screenshot:



In **New Java Interface**, select **Source folder** as `src/main/java`, specify **Package** as `org.jboss.jaxws.service`, and specify **Name** as `HelloWS`, as shown here. Then click on **Finish**.


New Java Interface

Java Interface

Create a new Java interface.

Source folder: jboss-jaxws/src/main/java

Package: org.jboss.jaxws.service

Enclosing type:

Name: HelloWS

Modifiers: [✓ public] package private protected

Extended interfaces:

Do you want to add comments? (Configure templates and default value here)

Generate comments

The `org.jboss.jaxws.HelloWS` interface gets added to the `jobss-jaxws` project. Annotate the `HelloWS` interface with `@WebService`, which indicates that the Java interface is a web service interface. The `name` element of `@WebService` specifies the web service name, which is used as the name of `wsdl:portType`. The `targetNamespace` specifies the target namespace for the web service. The endpoint interface defines a `sayHello` method annotated with `@WebMethod`, with its `operationName` element set to `hello`. The endpoint interface is listed here:

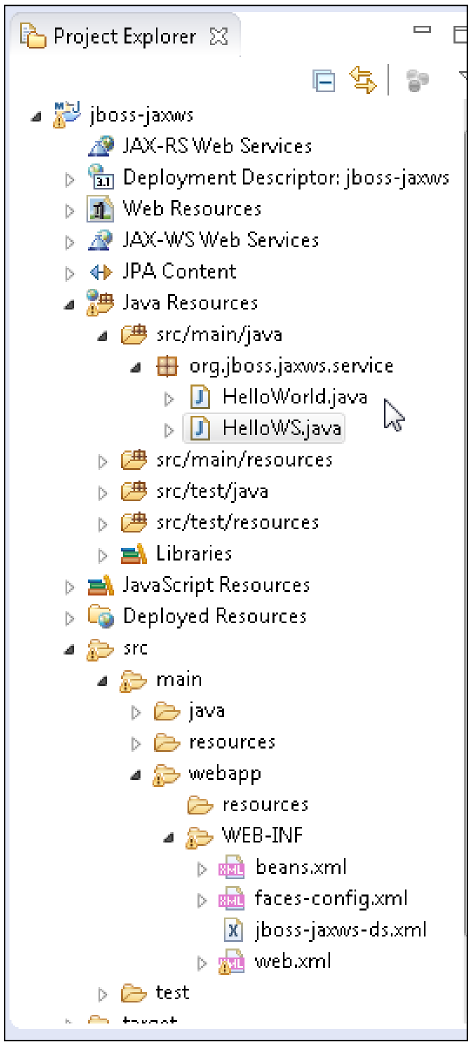
```
package org.jboss.jaxws.service;

import javax.jws.WebMethod;
import javax.jws.WebService;

@WebService(name = "HelloWS",   targetNamespace =
"http://org.jboss.jaxws.service/")
```

```
public interface HelloWS {
  @WebMethod(operationName = "hello")
  public String sayHello(String name);

}
```

A JAX-WS Web Service can be published using the JSR-109 programming model for implementing web services in Java. WildFly 8.1 supports the JSR-109 deployment model in which a web service can be configured as a servlet class in `web.xml`. We will use the JSR-109 deployment model; we need to configure the web service as a servlet in `web.xml`. Specify the endpoint class `org.jboss.jaxws.service.HelloWorld` as a servlet in `web.xml` with the corresponding servlet mapping URL as `/HelloWorld`. The `web.xml` file is listed here:

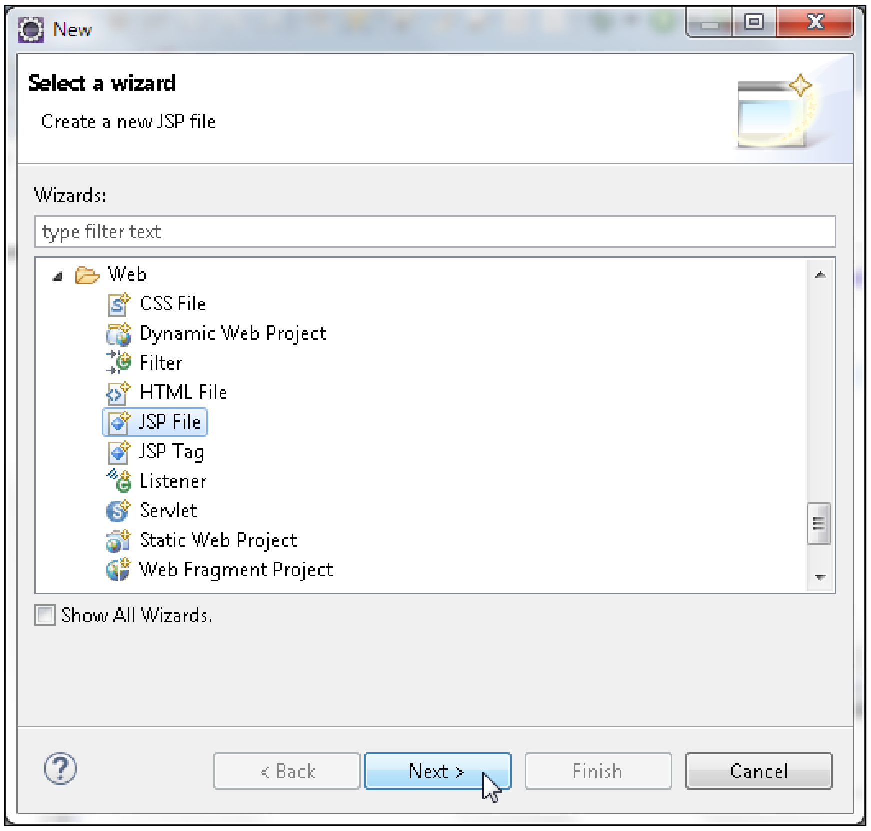
```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd" version="3.1">
  <servlet>
    <display-name>HelloWorld</display-name>
    <servlet-name>HelloWorld</servlet-name>
    <servlet-class>org.jboss.jaxws.service.HelloWorld</servlet-
    class>
  </servlet>
  <servlet-mapping>
    <servlet-name>HelloWorld</servlet-name>
    <url-pattern>/HelloWorld</url-pattern>
  </servlet-mapping>
</web-app>
```

The `HelloWorld` web service class and the service endpoint interface `HelloWS` are shown in Eclipse in the following screenshot:

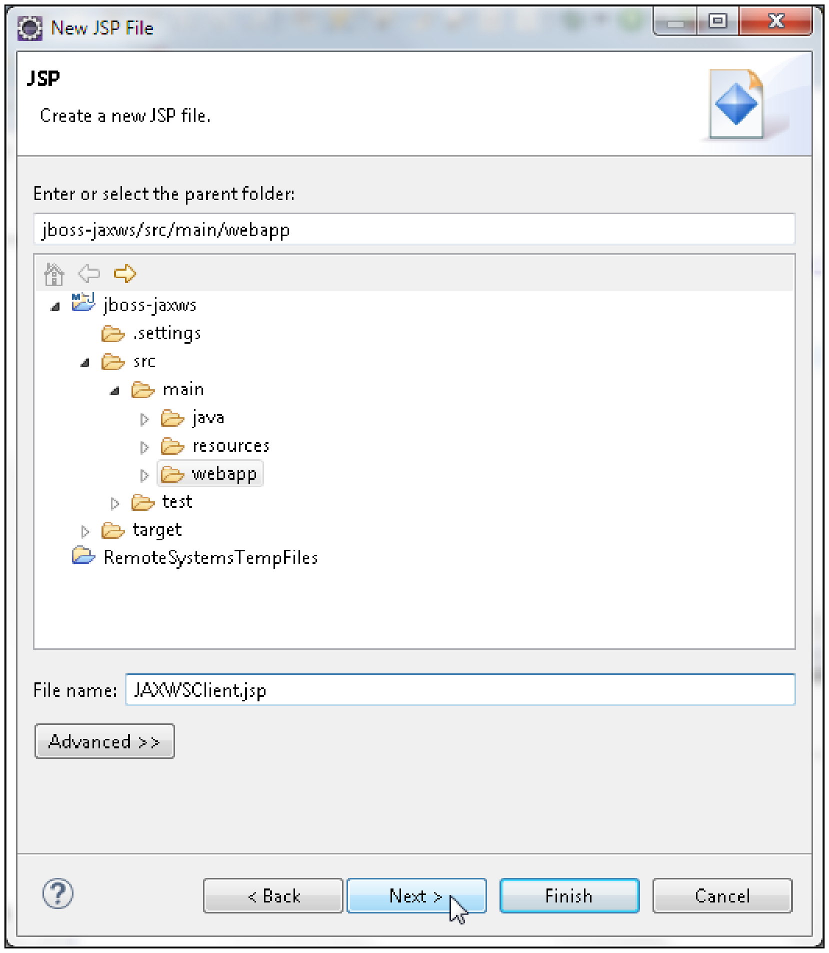


Creating a web service client

In this section, we will create a JSP Web Service client for the `HelloWorld` web service. To create a JSP, select **File** | **New** | **Other**. In **New**, select **Web** | **JSP File** and click on **Next**, as shown in the following screenshot:



In the **New JSP File** wizard, select the `webapp` folder, specify **File name** as `JAXWSClient.jsp`, and click on **Next**, as shown in the following screenshot:


New JSP File

JSP

Create a new JSP file.

Enter or select the parent folder:

jboss-jaxws/src/main/webapp

```
jboss-jaxws
    .settings
    src
        main
            java
            resources
            webapp
        test
    target
RemoteSystemsTempFiles
```

File name: JAXWSClient.jsp

Select the **New JSP file (html)** template and click on **Finish**. The `JAXWSClient.jsp` file gets added to the `webapp` folder. In the client JSP, we will invoke the web service with a name and display the web service response in the browser. First, create a URL object for the WSDL. The URL for a web service is constructed from the context root + endpoint:

```
URL wsdlLocation = new URL("http://localhost:8080/jboss-jaxws/
HelloWorld?WSDL");
```

The `jboss-jaxws` in the URL is the context root and the `/HelloWorld` is the servlet mapping URL for the web service endpoint as specified in the `web.xml`. Next, create a `QName` object for the service name. A `QName` represents a qualified name. Specify arguments to the `QName` constructor as the target namespace `http://org.jboss.jaxws.service/` and the service name `HelloWorldService`:

```
QName serviceName = new
QName("http://org.jboss.jaxws.service/","HelloWorldService");
```

The client view of a web service is provided by a `javax.xml.ws.Service` object. Create `javax.xml.ws.Service` from the WSDL location URL and `QName` for the service name:

```
Service service = Service.create(wsdlLocation, serviceName);
```

Get a proxy to the web service using the `getPort(Class endpointInterface)` method. Specify the endpoint interface class as `org.jboss.jaxws.service.HelloWS.class`:

```
HelloWS port =
service.getPort(org.jboss.jaxws.service.HelloWS.class);
```

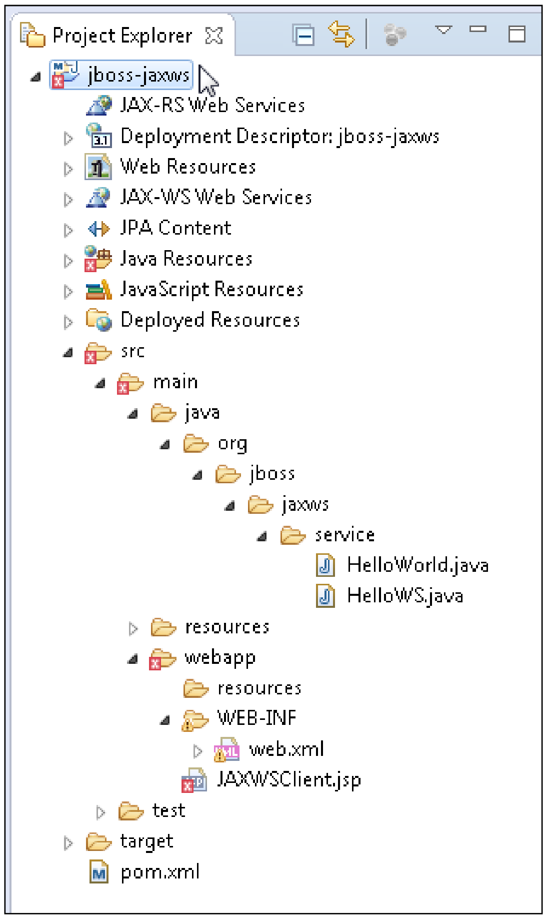
Invoke the `sayHello` method of the web service proxy with a name as an argument. The `sayHello` method returns a `String`. Output the response from the web service:

```
String result = port.sayHello("John Smith");
out.println(result);
```

The `JAXWSClient.jsp` is listed here:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-
1" pageEncoding="ISO-8859-1"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<%@ page
  import="org.jboss.jaxws.service.*,javax.xml.ws.WebServiceRef,
  java.net.URL,javax.xml.namespace.QName,javax.xml.ws.Service"%>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/xml;
    charset=windows-1252" />
    <title>JAXWS Client</title>
  </head>
  <body>
    <%URL wsdlLocation = new URL (
    "http://localhost:8080/jboss-jaxws/HelloWorld?WSDL");
    QName serviceName = new
    QName("http://org.jboss.jaxws.service/", "HelloWorldService");
    Service service = Service.create(wsdlLocation, serviceName);

    HelloWS port =
service.getPort(org.jboss.jaxws.service.HelloWS.class);
    String result = port.sayHello("John Smith");
    out.println(result);
%>
  </body>
</html>
```

The directory structure of the `jboss-jaxws` application is shown in Java EE web project in the following screenshot:



```
Project Explorer
jboss-jaxws
    JAX-RS Web Services
    Deployment Descriptor: jboss-jaxws
    Web Resources
    JAX-WS Web Services
    JPA Content
    Java Resources
    JavaScript Resources
    Deployed Resources
    src
        main
            java
                org
                    jboss
                        jaxws
                            service
                                HelloWorld.java
                                HelloWS.java
            resources
            webapp
                resources
                WEB-INF
                    web.xml
                JAXWSClient.jsp
        test
    target
    pom.xml
```

Deploying the JAX-WS application with Maven

In this section, we will compile, package, and deploy the `jboss-jaxws` application to WildFly 8.1. Specify the JAX-WS-related dependencies discussed in the following table in the `pom.xml` file:

Dependency	Description
Group Id: `com.sun.xml.ws` **Artifact Id**: `jaxws-rt`	Open source reference implementation of the JSR 224: Java API for XML Web Services
Group Id: `javax.xml.ws` **Artifact Id**: `jaxws-api`	JAX-WS (JSR 224) API
Group Id: `com.sun.xml.bind` **Artifact Id**: `jaxb-impl`	JAXB (JSR 222) Reference implementation
Group Id: `com.sun.xml.bind` **Artifact Id**: `jaxb-xjc`	JAXB (JSR 222) Reference implementation-schema compiler

We will use the JAX-WS Maven Plugin (`http://jax-ws-commons.java.net/jaxws-maven-plugin/`), which is the Maven adapter for the JAX-WS's toolset. The plugin provides `wsgen` and `wsimport` goals to generate the required portable artifacts for a web service. First, run the `wsgen` goal to generate the JAX-WS web service portable artifacts, including the WSDL, from an endpoint implementation class. Subsequently, run the `wsimport` goal to generate the web service portable artifacts, used by web service clients, from a WSDL. For the `wsgen` goal, specify the service endpoint interface in the `<sei/>` element as `org.jboss.jaxws.service.HelloWorld`, and specify the service name in the `<serviceName/>` element as `HelloWorldService`. Set `<genwsdl/>` for the `wsgen` goal to `true`. Specify the Maven Compiler Plugin and the Maven WAR Plugin, with the output directory set to the deployments directory. The `pom.xml` code is listed here:

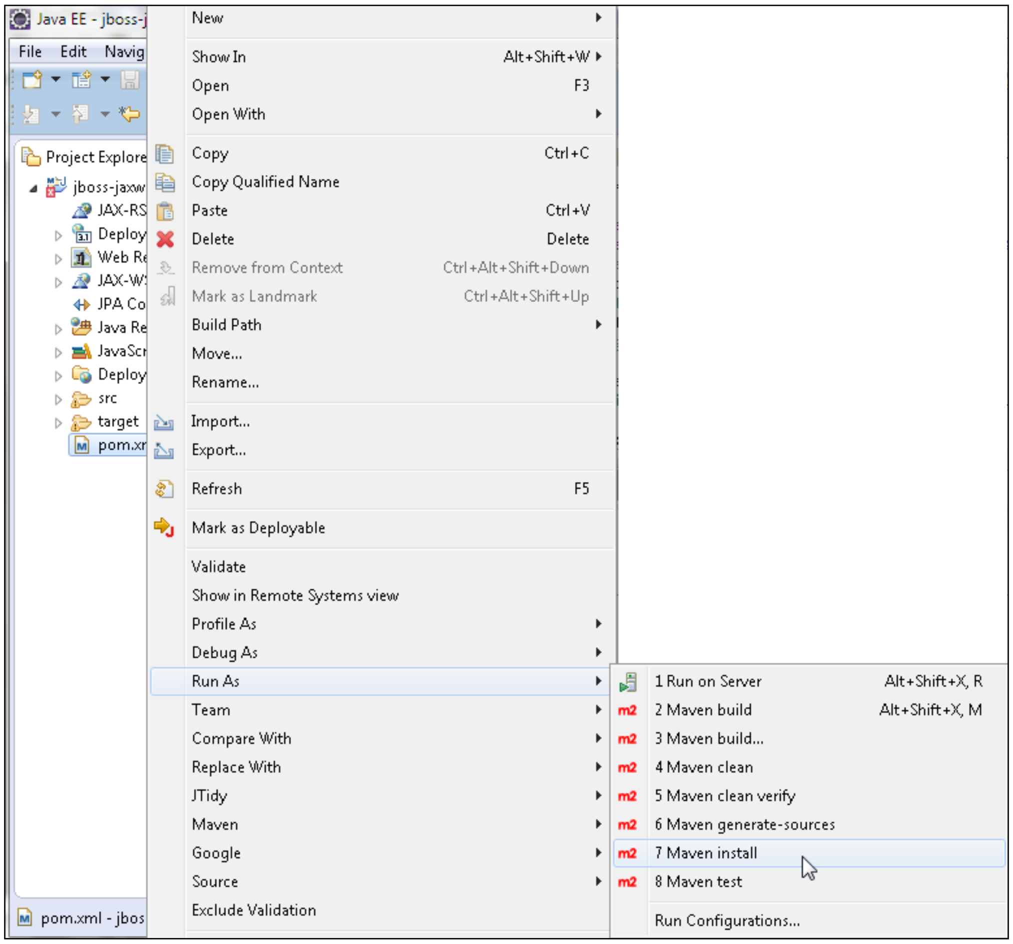
```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.jboss.jaxws</groupId>
  <artifactId>jboss-jaxws</artifactId>
  <version>1.0.0</version>
  <packaging>war</packaging>
  <name>Java EE 7 webapp project</name>
```

```
<description>A starter Java EE 7 webapp project for use on JBoss
WildFly / WildFly, generated from the jboss-javaee6-webapp
archetype</description>
<url>http://wildfly.org</url>
<properties>
  <project.build.sourceEncoding>UTF-
  8</project.build.sourceEncoding>
  <!-- JBoss dependency versions -->
  <version.wildfly.maven.plugin>1.0.2.Final
  </version.wildfly.maven.plugin>
  <!-- Define the version of the JBoss BOMs we want to import to
  specify
    tested stacks. -->
  <version.jboss.bom>8.1.0.Final</version.jboss.bom>
  <!-- other plugin versions -->
  <version.compiler.plugin>3.1</version.compiler.plugin>
  <version.war.plugin>2.1.1</version.war.plugin>
  <!-- maven-compiler-plugin -->
  <maven.compiler.target>1.7</maven.compiler.target>
  <maven.compiler.source>1.7</maven.compiler.source>
</properties>
<repositories>
  <repository>
    <id>JBoss Repository</id>
    <url>https://repository.jboss.org/nexus/content
    /groups/public/</url>
  </repository>
</repositories>
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.wildfly.bom</groupId>
      <artifactId>jboss-javaee-7.0-with-tools</artifactId>
      <version>${version.jboss.bom}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
    <dependency>
      <groupId>org.wildfly.bom</groupId>
      <artifactId>jboss-javaee-7.0-with-hibernate</artifactId>
      <version>${version.jboss.bom}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
```

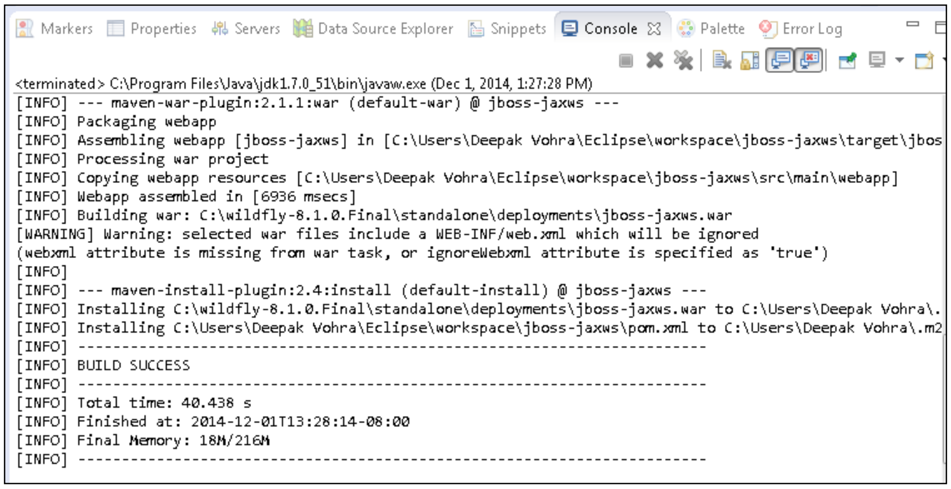
```
    </dependencyManagement>
    <dependencies>
      <!-- First declare the APIs we depend on and need for
      compilation. All of them are provided by JBoss WildFly -->
      <!-- Import the CDI API, we use provided scope as the API is
      included in
        JBoss WildFly -->
      <dependency>
        <groupId>javax.enterprise</groupId>
        <artifactId>cdi-api</artifactId>
        <scope>provided</scope>
      </dependency>
      <dependency>
        <groupId>com.sun.xml.ws</groupId>
        <artifactId>jaxws-rt</artifactId>
        <version>2.2.8</version>
        <scope>provided</scope>
      </dependency>
      <dependency>
        <groupId>javax.xml.ws</groupId>
        <artifactId>jaxws-api</artifactId>
        <version>2.2.8</version>
        <scope>provided</scope>
      </dependency>
      <dependency>
        <groupId>com.sun.xml.bind</groupId>
        <artifactId>jaxb-impl</artifactId>
        <version>2.2.7</version>
        <scope>provided</scope>
      </dependency>
      <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>javax.servlet-api</artifactId>
        <version>3.1.0</version>
      </dependency>
      <!-- Import the Common Annotations API (JSR-250), we use
      provided scope
        as the API is included in JBoss WildFly -->
      <dependency>
        <groupId>org.jboss.spec.javax.annotation</groupId>
        <artifactId>jboss-annotations-api_1.2_spec</artifactId>
        <scope>provided</scope>
      </dependency>
      <dependency>
```

```
        <groupId>javax.xml.bind</groupId>
        <artifactId>jaxb-api</artifactId>
        <version>2.2.7</version>
        <scope>provided</scope>
      </dependency>
    </dependencies>
    <build>
      <!-- Maven will append the version to the finalName (which is
      the name
        given to the generated war, and hence the context root) -->
      <finalName>${project.artifactId}</finalName>
      <pluginManagement>
        <plugins>
          <plugin>
            <groupId>org.eclipse.m2e</groupId>
            <artifactId>lifecycle-mapping</artifactId>
            <version>1.0.0</version>
            <configuration>
              <lifecycleMappingMetadata>
                <pluginExecutions>
                  <pluginExecution>
                    <pluginExecutionFilter>
                      <groupId>org.jvnet.jax-ws-commons</groupId>
                      <artifactId>jaxws-maven-plugin</artifactId>
                      <versionRange>[2.2,)</versionRange>
                      <goals>
                        <goal>wsimport</goal>
                      </goals>
                    </pluginExecutionFilter>
                    <action>
                      <ignore />
                    </action>
                  </pluginExecution>
                </pluginExecutions>
              </lifecycleMappingMetadata>
            </configuration>
          </plugin>
          <!-- Compiler plugin enforces Java 1.6 compatibility and
          activates annotation processors -->
          <plugin>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>${version.compiler.plugin}</version>
            <configuration>
              <source>${maven.compiler.source}</source>
            <target>${maven.compiler.target}</target>
          </configuration>
        </plugin>
        <plugin>
          <artifactId>maven-war-plugin</artifactId>
          <version>${version.war.plugin}</version>
          <configuration>
            <outputDirectory>C:\wildfly-
            8.1.0.Final\standalone\deployments</outputDirectory>
            <!-- Java EE 7 doesn't require web.xml, Maven needs to
            catch up! -->
            <failOnMissingWebXml>false</failOnMissingWebXml>
          </configuration>
        </plugin>
        <!-- The WildFly plugin deploys your war to a local
        WildFly container -->
        <!-- To use, run: mvn package wildfly:deploy -->
        <plugin>
          <groupId>org.wildfly.plugins</groupId>
          <artifactId>wildfly-maven-plugin</artifactId>
          <version>${version.wildfly.maven.plugin}</version>
        </plugin>
        <plugin>
          <groupId>org.jvnet.jax-ws-commons</groupId>
          <artifactId>jaxws-maven-plugin</artifactId>
          <version>2.2</version>
          <executions>
            <execution>
              <id>HelloWorldService</id>
              <phase>compile</phase>
              <goals>
                <goal>wsgen</goal>
              </goals>
              <configuration>
                <sei>org.jboss.jaxws.service.HelloWorld</sei>
                <genwsdl>true</genwsdl>
                <servicename>HelloWorldService</servicename>
                <keep>true</keep>
              </configuration>
            </execution>
          </executions>
        </plugin>
        <plugin>
          <groupId>org.jvnet.jax-ws-commons</groupId>
          <artifactId>jaxws-maven-plugin</artifactId>
          <version>2.3</version>
          <executions>
            <execution>
              <id>HelloWorldService</id>
              <goals>
                <goal>wsimport</goal>
              </goals>
            </execution>
          </executions>
          <configuration>
            <packagename>org.jboss.jaxws.service</packagename>
            <target>2.0</target>
            <keep>true</keep>
          </configuration>
          <dependencies>
            <dependency>
              <groupId>com.sun.xml.ws</groupId>
              <artifactId>jaxws-tools</artifactId>
              <version>2.2.8</version>
              <exclusions>
                <exclusion>
                  <groupId>org.jvnet.staxex</groupId>
                  <artifactId>stax-ex</artifactId>
                </exclusion>
              </exclusions>
            </dependency>
            <dependency>
              <groupId>org.jvnet.staxex</groupId>
              <artifactId>stax-ex</artifactId>
              <version>1.7.6</version>
              <exclusions>
                <exclusion>
                  <groupId>javax.xml.stream</groupId>
                  <artifactId>stax-api</artifactId>
                </exclusion>
              </exclusions>
            </dependency>
          </dependencies>
        </plugin>
      </plugins>
    </pluginManagement>
  </build>
</project>
```

Right-click on `pom.xml` in the **Project Explorer** and select **Run As | Maven install**, as shown in the following screenshot:

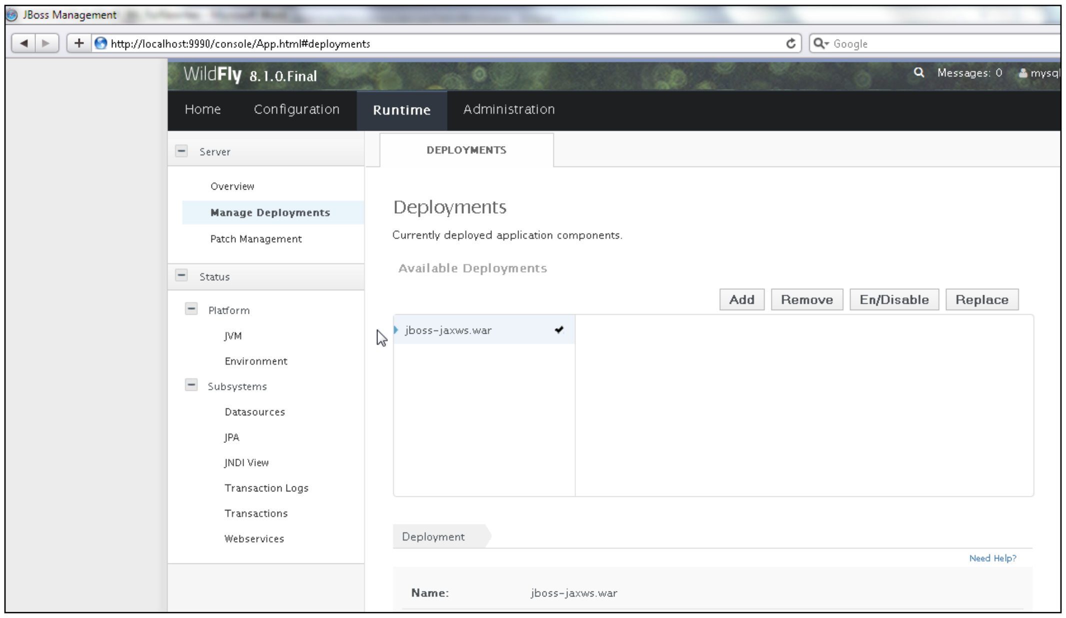


The `jboss-jaxws` project gets compiled and packaged into a `jboss-jaxws.war` archive. A `BUILD SUCCESS` message indicates that the Maven build completed without any error, as shown here:



```
<terminated> C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Dec 1, 2014, 1:27:28 PM)
[INFO] --- maven-war-plugin:2.1.1:war (default-war) @ jboss-jaxws ---
[INFO] Packaging webapp
[INFO] Assembling webapp [jboss-jaxws] in [C:\Users\Deepak Vohra\Eclipse\workspace\jboss-jaxws\target\jbos
[INFO] Processing war project
[INFO] Copying webapp resources [C:\Users\Deepak Vohra\Eclipse\workspace\jboss-jaxws\src\main\webapp]
[INFO] Webapp assembled in [6936 msecs]
[INFO] Building war: C:\wildfly-8.1.0.Final\standalone\deployments\jboss-jaxws.war
[WARNING] Warning: selected war files include a WEB-INF/web.xml which will be ignored
(webxml attribute is missing from war task, or ignoreWebxml attribute is specified as 'true')
[INFO]
[INFO] --- maven-install-plugin:2.4:install (default-install) @ jboss-jaxws ---
[INFO] Installing C:\wildfly-8.1.0.Final\standalone\deployments\jboss-jaxws.war to C:\Users\Deepak Vohra\.
[INFO] Installing C:\Users\Deepak Vohra\Eclipse\workspace\jboss-jaxws\pom.xml to C:\Users\Deepak Vohra\.m2
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 40.438 s
[INFO] Finished at: 2014-12-01T13:28:14-08:00
[INFO] Final Memory: 18M/216M
[INFO] ------------------------------------------------------------------------
```

Start WildFly 8.1 if it is not already started. The `jboss-jaxws.war` archive gets deployed to WildFly 8.1, as shown in the **Administration** console:



Deployments

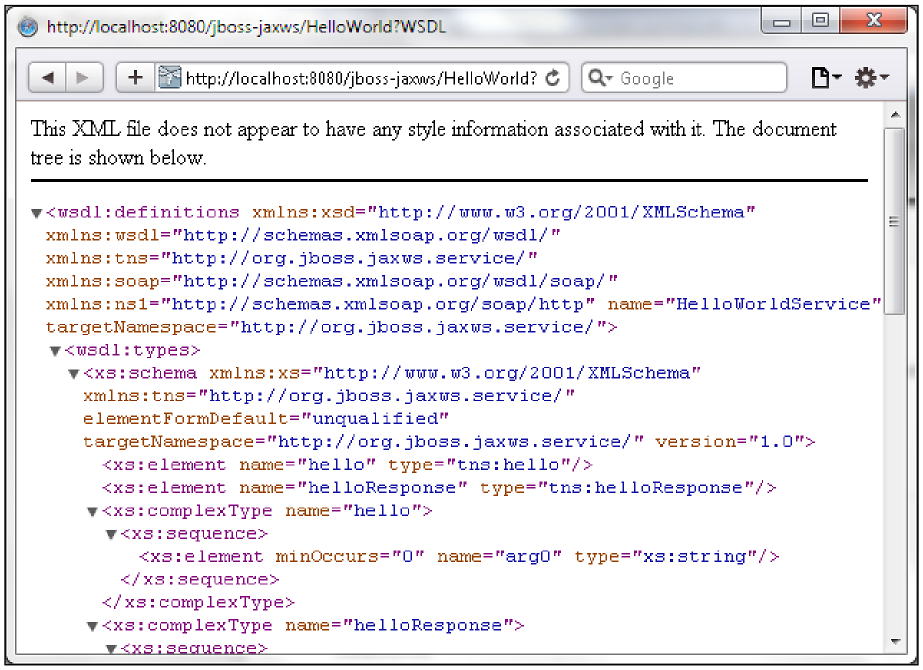
Currently deployed application components.

Available Deployments

jboss-jaxws.war ✓

Name: jboss-jaxws.war

The WSDL for the web service can be invoked in a browser with the URL `http://localhost:8080/jboss-jaxws/HelloWorld?WSDL`, as shown in the following screenshot:



http://localhost:8080/jboss-jaxws/HelloWorld?WSDL

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
▼<wsdl:definitions xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
 xmlns:tns="http://org.jboss.jaxws.service/"
 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:ns1="http://schemas.xmlsoap.org/soap/http" name="HelloWorldService"
 targetNamespace="http://org.jboss.jaxws.service/">
  ▼<wsdl:types>
    ▼<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
     xmlns:tns="http://org.jboss.jaxws.service/"
     elementFormDefault="unqualified"
     targetNamespace="http://org.jboss.jaxws.service/" version="1.0">
       <xs:element name="hello" type="tns:hello"/>
       <xs:element name="helloResponse" type="tns:helloResponse"/>
     ▼<xs:complexType name="hello">
       ▼<xs:sequence>
          <xs:element minOccurs="0" name="arg0" type="xs:string"/>
        </xs:sequence>
       </xs:complexType>
     ▼<xs:complexType name="helloResponse">
       ▼<xs:sequence>
```

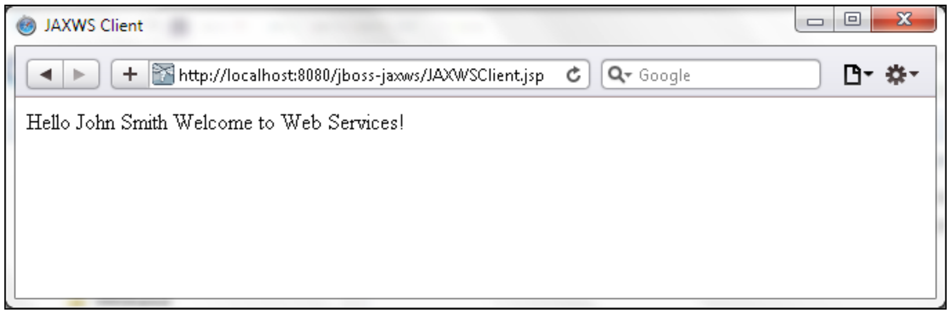
The WSDL file's content is shown as follows:

```
<wsdl:definitions xmlns:xsd="http://www.w3.org/2001/XMLSchema"xmln
s:wsdl="http://schemas.xmlsoap.org/wsdl/"xmlns:tns="http://org.jbo
ss.jaxws.service/"xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap
/"xmlns:ns1="http://schemas.xmlsoap.org/soap/http" name="HelloWorl
dService"targetNamespace="http://org.jboss.jaxws.service/">
<wsdl:types>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"xmlns:tns="h
ttp://org.jboss.jaxws.service/"elementFormDefault="unqualified"tar
getNamespace="http://org.jboss.jaxws.service/" version="1.0">
<xs:element name="hello" type="tns:hello"/>
<xs:element name="helloResponse" type="tns:helloResponse"/>
<xs:complexType name="hello">
<xs:sequence>
<xs:element minOccurs="0" name="arg0" type="xs:string"/>
</xs:sequence>
</xs:complexType>
<xs:complexType name="helloResponse">
```

```
<xs:sequence>
<xs:element minOccurs="0" name="return" type="xs:string"/>
</xs:sequence>
</xs:complexType>
</xs:schema>
</wsdl:types>
<wsdl:message name="hello">
<wsdl:part element="tns:hello" name="parameters"></wsdl:part>
</wsdl:message>
<wsdl:message name="helloResponse">
<wsdl:part element="tns:helloResponse" name="parameters">
</wsdl:part>
</wsdl:message>
<wsdl:portType name="HelloWS">
<wsdl:operation name="hello">
<wsdl:input message="tns:hello" name="hello"></wsdl:input>
<wsdl:output message="tns:helloResponse"name="helloResponse">
</wsdl:output>
</wsdl:operation>
</wsdl:portType>
<wsdl:binding name="HelloWorldServiceSoapBinding" type="tns:HelloW
S">
<soap:binding style="document"transport="http://schemas.xmlsoap.or
g/soap/http"/>
<wsdl:operation name="hello">
<soap:operation soapAction="" style="document"/>
<wsdl:input name="hello">
<soap:body use="literal"/>
</wsdl:input>
<wsdl:output name="helloResponse">
<soap:body use="literal"/>
</wsdl:output>
</wsdl:operation>
</wsdl:binding>
<wsdl:service name="HelloWorldService">
<wsdl:port binding="tns:HelloWorldServiceSoapBinding"
name="HelloWorldPort">
<soap:address location="http://localhost:8080/jboss-
jaxws/HelloWorld"/>
</wsdl:port>
</wsdl:service>
</wsdl:definitions>
```

Running the JAX-WS application

In this section, we will invoke the web service using a JSP client. Invoke the `JAXWSClient.jsp` file in a browser with the URL `http://localhost:8080/jboss-jaxws/JAXWSClient.jsp`. The web service response gets displayed in the browser, as shown in the following screenshot:



Hello John Smith Welcome to Web Services!

Summary

In this chapter, we developed a JAX-WS Web Service in Eclipse IDE with JBoss Tools sample web service. We created an endpoint implementation class and an endpoint interface. We developed a JSP client for the web service. The web service is deployed using the JSR-109 deployment model, which involves configuring the endpoint class as a servlet in `web.xml`. We compiled and packaged the web service project with Maven build tools and deployed the web application to WildFly 8.1. Subsequently, we ran the web service client JSP in a browser.

In the next chapter, we will discuss RESTful web services with WildFly 8.1.

7 Developing a JAX-RS 1.1 Web Service

In this chapter, we will discuss **Representational State Transfer** (**REST**) web services, specifically those based on Java API for RESTful web services (JAX-RS 1.x), which is defined in JSR 311 (`http://jcp.org/en/jsr/detail?id=311`). The new version of JAX-RS (2.0) has become available and is discussed in *Chapter 9, Using JAX-RS 2.0 in Java EE 7 with RESTEasy*. The previous version is discussed first, as it forms the basis for the new version and also because a lot of developers are still using the previous version. Migration to a new version of existing applications does not happen as and when a new version becomes available. REST is a protocol independent, loosely coupled, software architecture style for distributed systems. Protocol independent implies that REST supports any protocol that supports transfer of representational state, but we will discuss REST over HTTP only. In comparison to SOAP, REST is less strongly typed, does not require XML parsing, and does not require message headers. RESTful web services are based on the REST principle and are simple, lightweight, and fast. A RESTful web service exposes a set of resources, which are simply sources of information, identified by URIs in HTTP. A resource could be a database record or a **Plain Old Java Object** (**POJO**) for example. What are exchanged are not resources themselves, but representations of resources, which are typically documents in HTML, XML, JSON, and plain text, but they could be an image or some other format. Based on the representations of resources and included metadata, a client makes changes to the resources.

RESTful web services follow these RESTful principles:

- Every resource has a unique base URI.
- For invoking web service operations, the HTTP protocol methods such as `GET`, `PUT`, `POST`, and `DELETE` are used.

- A client sends a request to a service, and the service returns a representation of a resource requested to the client.
- Client sessions are not stored on the server, which makes it easier to scale the service with less data to replicate in a clustered environment.

In this chapter, we will use the Jersey JAX-RS (JSR 311) **Reference Implementation** (**RI**) to create RESTful web services in Java. This chapter has the following sections:

- Setting up the environment
- Creating a Java EE web project
- Creating a sample RESTful web service
- Deploying the RESTful web service
- Running the RESTful web service
- Creating a Java client
- Running the Java client
- Creating a JSP client
- Running the JSP client

Setting up the environment

We need to download and install the following software:

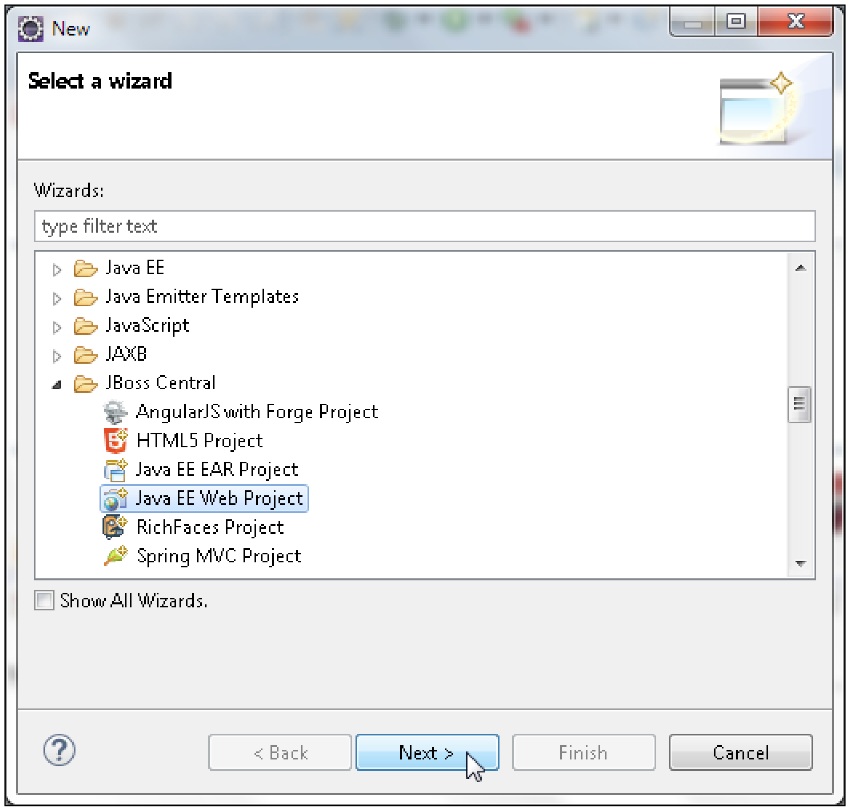
- **WildFly 8.1.0.Final**: Download `wildfly-8.1.0.Final.zip` from `http://wildfly.org/downloads/`.
- **MySQL 5.6 Database-Community Edition**: Download this edition from `http://dev.mysql.com/downloads/mysql/`. When installing MySQL, install **Connector/J** too.
- **Eclipse IDE for Java EE Developers**: Download Eclipse Luna from `https://www.eclipse.org/downloads/packages/release/Luna/SR1`.
- **JBoss Tools (Luna) 4.2.0.Final (or the latest version)**: Install this as a plugin to Eclipse from the Eclipse Marketplace (`http://tools.jboss.org/downloads/installation.html`).
- **Apache Maven**: Download version 3.05 or higher from `http://maven.apache.org/download.cgi`.
- **Java 7**: Download Java 7 from `http://www.oracle.com/technetwork/java/javase/downloads/index.html?ssSourceSiteId=ocomcn`.

Set the environment variables `JAVA_HOME`, `JBOSS_HOME`, and `MAVEN_HOME`. Add `%JAVA_HOME%/bin`, `%MAVEN_HOME%/bin`, and `%JBOSS_HOME%/bin` to the `PATH` environment variable.

Create a WildFly 8.1.0 runtime as discussed in *Chapter 1*, *Getting Started with EJB 3.x*.

Creating a Java EE web project

In this section, we will create a **Java EE Web Project**. Select **File** | **New** | **Other**. In **New**, select **JBoss Central** | **Java EE Web Project**, as shown here. Now, click on **Next**.



New

Select a wizard

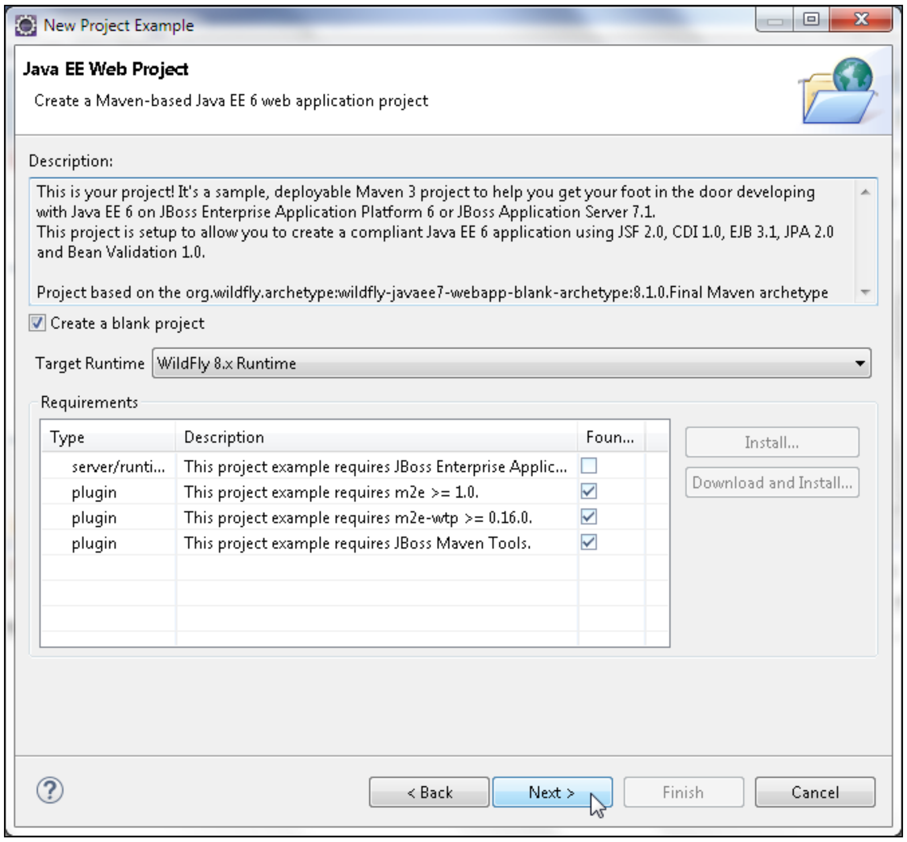
Wizards:

type filter text

- Java EE
- Java Emitter Templates
- JavaScript
- JAXB
- JBoss Central
  - AngularJS with Forge Project
  - HTML5 Project
  - Java EE EAR Project
  - Java EE Web Project
  - RichFaces Project
  - Spring MVC Project

Show All Wizards.

A list for requirements, which includes the **m2e** and **m2eclipse-wtp** plugins, and the **JBoss Maven Tools** plugin is shown as follows. Check the **Create a blank project** checkbox and select **WildFly 8.x Runtime** as **Target Runtime**. Then, click on **Next**.



Description:

This is your project! It's a sample, deployable Maven 3 project to help you get your foot in the door developing with Java EE 6 on JBoss Enterprise Application Platform 6 or JBoss Application Server 7.1.
This project is setup to allow you to create a compliant Java EE 6 application using JSF 2.0, CDI 1.0, EJB 3.1, JPA 2.0 and Bean Validation 1.0.

Project based on the org.wildfly.archetype:wildfly-javaee7-webapp-blank-archetype:8.1.0.Final Maven archetype

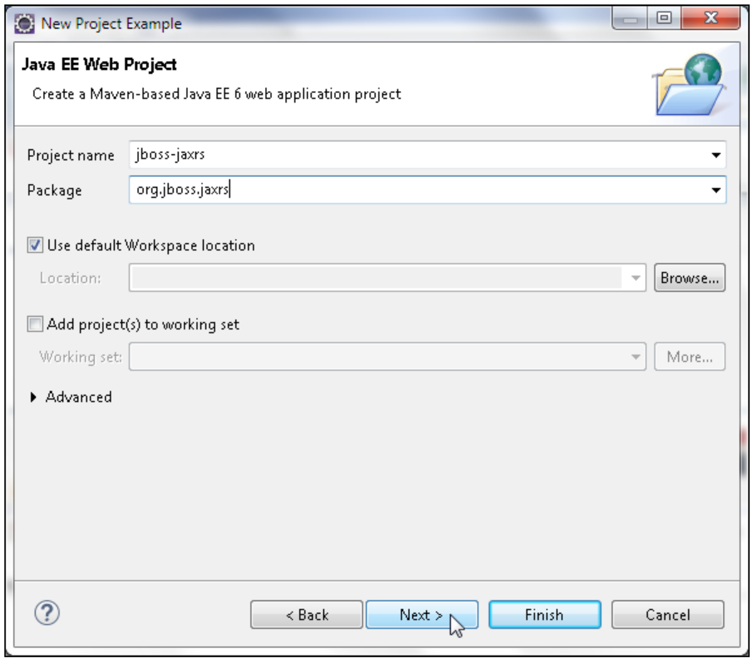
☑ Create a blank project

Target Runtime: WildFly 8.x Runtime

Requirements

Type	Description	Foun...
server/runti...	This project example requires JBoss Enterprise Applic...	☐
plugin	This project example requires m2e >= 1.0.	☑
plugin	This project example requires m2e-wtp >= 0.16.0.	☑
plugin	This project example requires JBoss Maven Tools.	☑

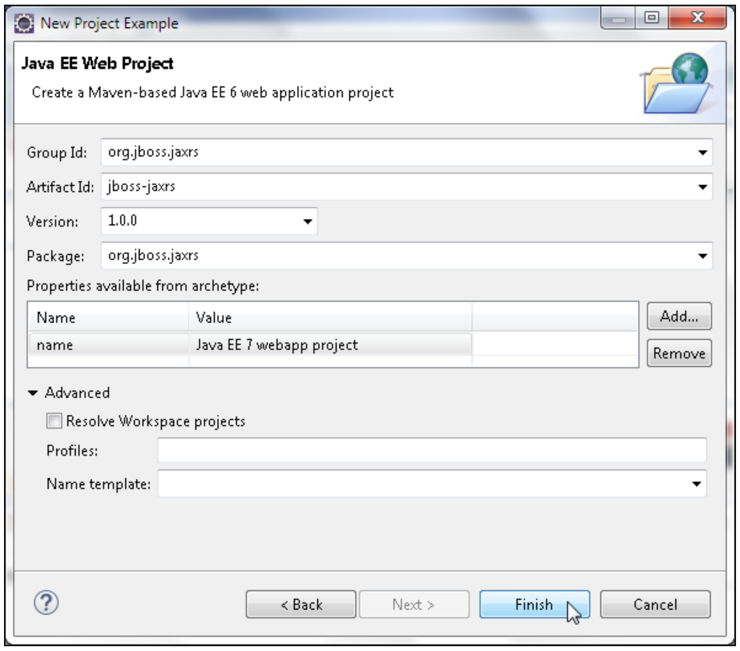
Specify **Project name** (`jboss-jaxrs`) and **Package** (`org.jboss.jaxrs`), and click on **Next**, as shown here:


New Project Example
Java EE Web Project
Create a Maven-based Java EE 6 web application project
Use default Workspace location
Location:
Browse...
Add project(s) to working set
Working set:
More...
Advanced
< Back
Next >
Finish
Cancel

Project name: jboss-jaxrs

Package: org.jboss.jaxrs

Specify **Group Id** (`org.jboss.jaxrs`), **Artifact Id** (`jboss-jaxrs`), **Version** (`1.0.0`), and **Package** (`org.jboss.jaxrs`) and click on **Finish**, as shown here:



Group Id: org.jboss.jaxrs

Artifact Id: jboss-jaxrs

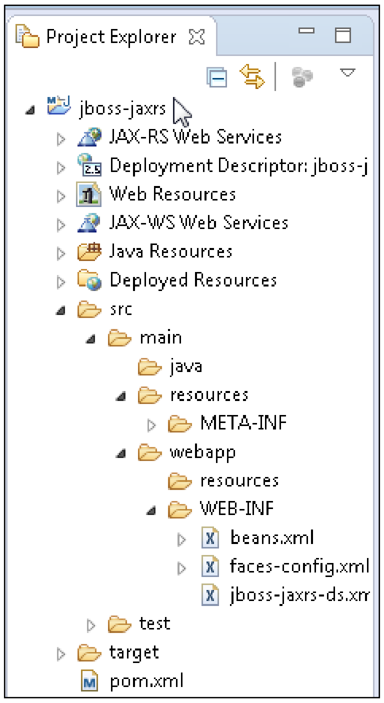
Version: 1.0.0

Package: org.jboss.jaxrs

Properties available from archetype:

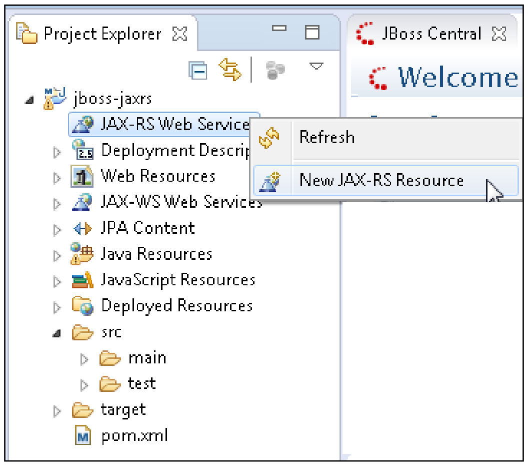
Name	Value
name	Java EE 7 webapp project

The `jboss-jaxrs` Java EE web project is shown in **Project Explorer**:

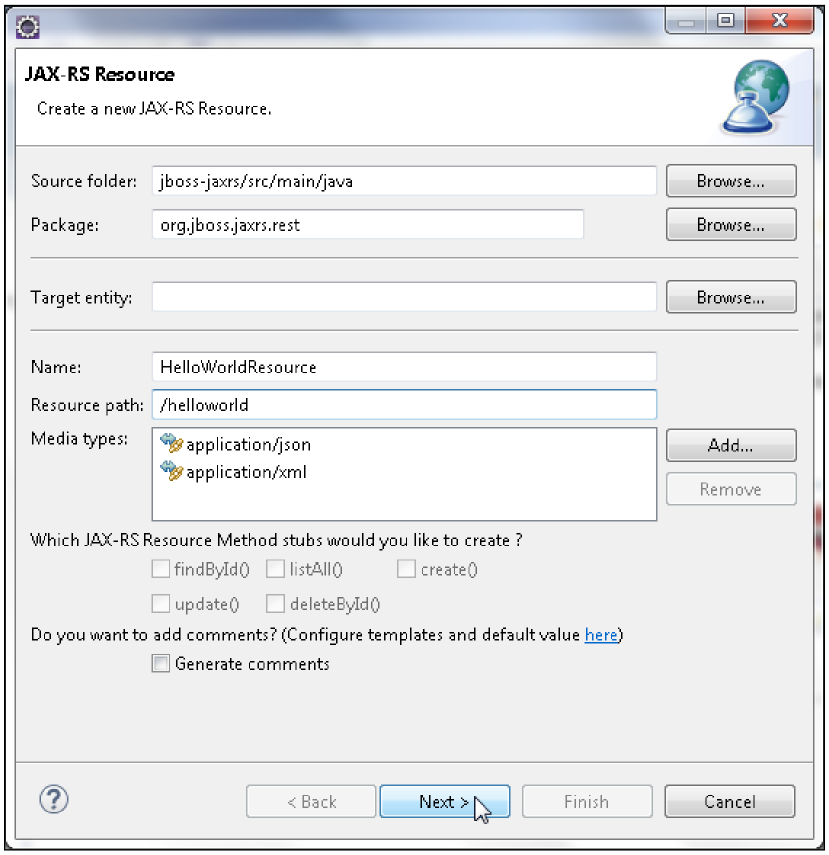


Creating a sample RESTful web service

In this section, we will create a RESTful web service. Right-click on the **JAX-RS Web Services** node in **Project Explorer** and select **New JAX-RS Resource**, as shown here:



Select the `jboss-jaxrs` project and specify **Package** as `org.jboss.jaxrs.rest`, **Name** as `HelloWorldResource`, **Resource path** as `/helloworld`, and click on **Next**, as shown here:



JAX-RS Resource

Create a new JAX-RS Resource.

Source folder: jboss-jaxrs/src/main/java

Package: org.jboss.jaxrs.rest

Target entity:

Name: HelloWorldResource

Resource path: /helloworld

Media types: application/json, application/xml

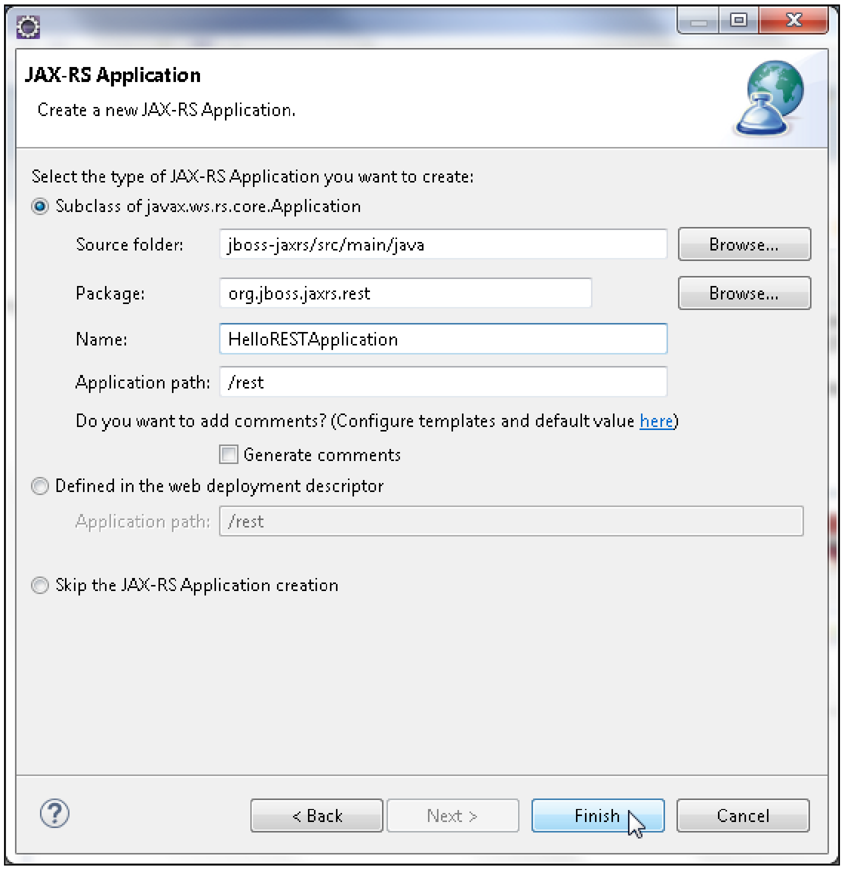
Which JAX-RS Resource Method stubs would you like to create ?

☐ findById() ☐ listAll() ☐ create() ☐ update() ☐ deleteById()

Do you want to add comments? (Configure templates and default value here)

☐ Generate comments

In the **JAX-RS Application** window, select the **Source folder** as `jboss-jaxrs/src/main/java` and specify **Package** as `org.jboss.jaxrs.rest`, **Name** as `HelloRESTApplication`, **Application path** as `/rest`, and click on **Finish**, as shown here:


JAX-RS Application

Create a new JAX-RS Application.

Select the type of JAX-RS Application you want to create:

(●) Subclass of javax.ws.rs.core.Application

Source folder: jboss-jaxrs/src/main/java

Package: org.jboss.jaxrs.rest

Name: HelloRESTApplication

Application path: /rest

Do you want to add comments? (Configure templates and default value here)

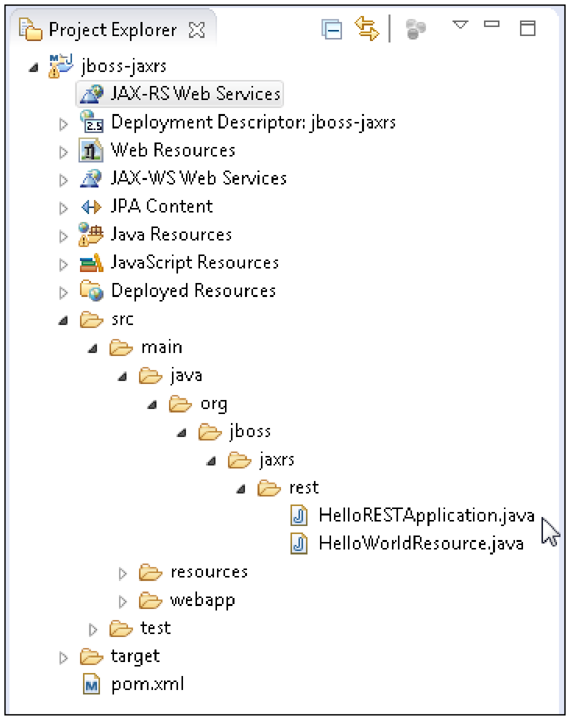
[ ] Generate comments

( ) Defined in the web deployment descriptor

Application path: /rest

( ) Skip the JAX-RS Application creation

The `org.jboss.jaxrs.rest.HelloWorldResource` class gets generated, as shown in the **Project Explorer**.



A JAX-RS RESTful web service resource is defined using a root resource class. A web service's URI is constructed from path designators, which are specified using the `@PATH` annotation. A resource is defined using a root resource class. The root resource classes are POJOs annotated with `@PATH` with one or more class methods annotated with a resource method designator (`@GET`, `@PUT`, `@POST`, `@DELETE`) or with the `@PATH` annotation or with both a resource method designator and `@PATH`. The root class methods that are annotated with resource method designators are called resource methods. The resource methods that are annotated with `@PATH` are called subresource methods. The class methods that are annotated only with `@PATH` are called subresource locators.

The resources respond to HTTP methods such as `GET`, `POST`, `PUT`, and `DELETE` (`http://www.w3.org/Protocols/HTTP/Methods.html`). For example, a client may get a resource representation with `GET`, upload a modified copy of the resource using `PUT`, or delete the resource using `DELETE`. The resource methods (and the subresource methods and subresource locators) may return a resource representation in various formats such as HTML, plain text, XML, PDF, JPEG, and JSON.

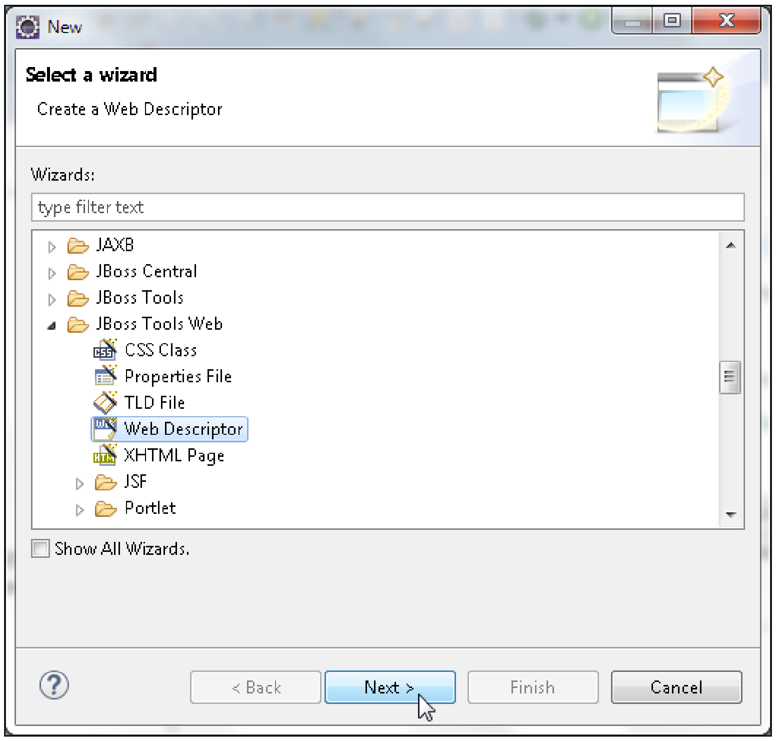
We will create a root resource class with some resource methods using the `@GET` request method designator. Annotate the Java class `org.jboss.jaxrs.rest.HelloWorldResource` with the `@PATH` annotation. Specify the URI path on which the Java class will be hosted as `/helloworld`:

```
@Path("/helloworld")
public class HelloWorldResource {
…
}
```

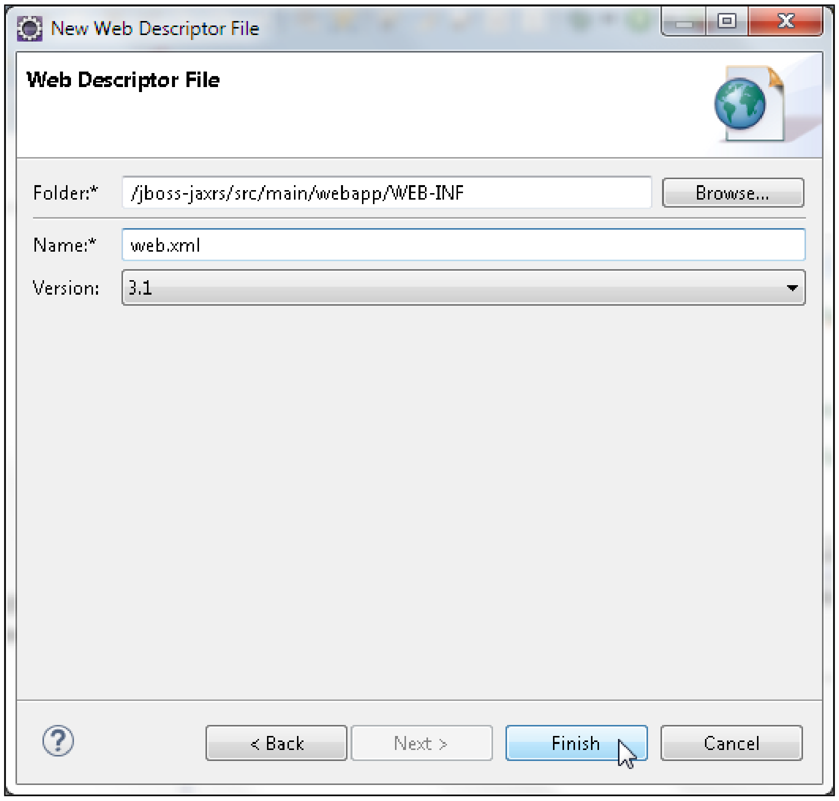
Next, add resource methods to produce three different MIME types. Add the resource methods `getClichedMessage()`, `getXMLMessage()`, and `getHTMLMessage()` (method names are arbitrary) annotated with the `@GET` annotation, which indicates that the resource methods will process the HTTP `GET` requests from a client. Each of the resource method produces a different MIME type resource representation. Each of the resource methods returns `String`, but the MIME type returned and specified in the `@Produces` annotation is different for the different resource methods. We will output a `"Hello JAX-RS"` message in three different MIME types: `text/plain`, `text/xml`, and `text/html`. The `String` value returned in each of the resource methods is in the format corresponding to the designated MIME type. Add two versions of the `getXMLMessage()` method one for a JSP client, as discussed in a later section. Add a resource method annotated with `@Produces("application/xml")` to demonstrate the requirement for the produced MIME type to match an acceptable MIME type in a client. The root resource class is listed as follows; some sections have been commented out for testing the resource methods separately:

```
package org.jboss.jaxrs.rest;

import javax.ws.rs.GET;
import javax.ws.rs.Produces;
import javax.ws.rs.Path;
import javax.ws.rs.core.MediaType;

// The Java class will be hosted at the URI path "/helloworld"
@Path("/helloworld")
public class HelloWorldResource {

  // The Java method will process HTTP GET requests
  //
  // The Java method will produce content identified by the MIME
  Media
  // type "text/plain"
  @GET
  @Produces("text/plain")
  public String getClichedMessage() {
    // Return some cliched textual content
    return "Hello JAX-RS";
  }
  /**
   * @GET
   * @Produces("application/xml") public String getXMLMessage() {
  return
   *                              "<?xml version=\"1.0\"?>" +
   *                              "<hello> Hello JAX-RS" +
  "</hello>"; }
  */

//for java  client
@GET
  @Produces("text/xml")
  public String getXMLMessage() {
    return "<?xml version=\"1.0\"?>" + "<hello> Hello JAX-RS" +
    "</hello>";
  }

//for jsp client
  /**@GET
  @Produces("text/xml")
  public String getXMLMessage () {
    return "&lt;?xml version=\"1.0\"?&gt;" + "&lt;hello&gt;Hello
    JAX-RS" + "&lt;/hello&gt;";
  }*/

  @GET
  @Produces("text/html")
  public String getHTMLMessage() {
    return "<html> " + "<title>" + "Hello JAX-RS" + "</title>"
        + "<body><h1>" + "Hello JAX-RS" + "</body></h1> " +
        "</html> ";
  }

}
```

We also need to create a web deployment descriptor `web.xml`. Select **File** | **New** | **Other**, and in **New**, select **JBoss Tools Web** | **Web Descriptor** and click on **Next**, as shown here:



In **New Web Descriptor File**, select the `/jboss-jaxrs/src/main/webapp/WEB-INF` folder and specify **Name** as `web.xml` and select **Version** as **3.1**, and click on **Finish**, as shown here:



Folder:* /jboss-jaxrs/src/main/webapp/WEB-INF

Name:* web.xml

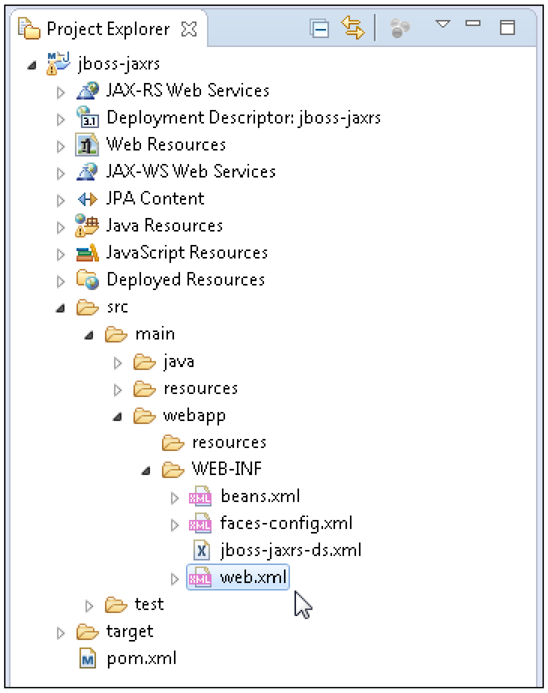
Version: 3.1

In the `web.xml` file, specify a servlet for the servlet class `com.sun.jersey.spi.container.servlet.ServletContainer`, which is a servlet for deploying root resource classes. Specify the servlet mapping URL pattern as `/jaxrs/*`:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd">
 <display-name>EclipseJAX-RS</display-name>
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.htm</welcome-file>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>
  <servlet>
    <servlet-name>JAX-RS Servlet</servlet-name>
```

```
        <servlet-class>com.sun.jersey.spi.container.servlet.
    ServletContainer
    </servlet-class>
        <load-on-startup>1</load-on-startup>
      </servlet>
      <servlet-mapping>
        <servlet-name>JAX-RS Servlet</servlet-name>
        <url-pattern>/jaxrs/*</url-pattern>
      </servlet-mapping>
    </web-app>
```

The `web.xml` file is shown in **Project Explorer** as follows:



Deploying the RESTful web service

In this section, we will compile, package, and deploy the JAX-RS application `jboss-jaxrs` with Maven. A `pom.xml` file gets created when a Java EE Web Project is created. In the `pom.xml` file, the `jboss-jaxrs` Artifact ID is specified with packaging as `war`. The JAX-RS API is provided by WildFly 8.1:

```
    <dependency>
      <groupId>org.jboss.spec.javax.ws.rs</groupId>
      <artifactId>jboss-jaxrs-api_1.1_spec</artifactId>
      <version>1.0.1.Final</version>
      <scope>provided</scope>
    </dependency>
```

As we are using the Jersey JAX-RS RI, include the dependencies for Jersey, which are available in the Group ID `com.sun.jersey`:

```
  <dependency>
     <groupId>com.sun.jersey</groupId>
    <artifactId>jersey-server</artifactId>
    <version>1.18</version>
  </dependency>
  <dependency>
    <groupId>com.sun.jersey</groupId>
    <artifactId>jersey-core</artifactId>
    <version>1.18</version>
  </dependency>

  <dependency>
    <groupId>com.sun.jersey</groupId>
    <artifactId>jersey-servlet</artifactId>
    <version>1.18</version>
  </dependency>
  <dependency>
    <groupId>com.sun.jersey</groupId>
    <artifactId>jersey-client</artifactId>
    <version>1.18</version>
  </dependency>
```

Add the Maven Compiler plugin and the Maven WAR plugin and specify the output directory as the `deployments` directory of WildFly 8.1 installation. The `pom.xml` file is listed as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.jboss.jaxrs</groupId>
  <artifactId>jboss-jaxrs</artifactId>
  <version>1.0.0</version>
  <packaging>war</packaging>
  <name>WildFly JAX-RS</name>
```

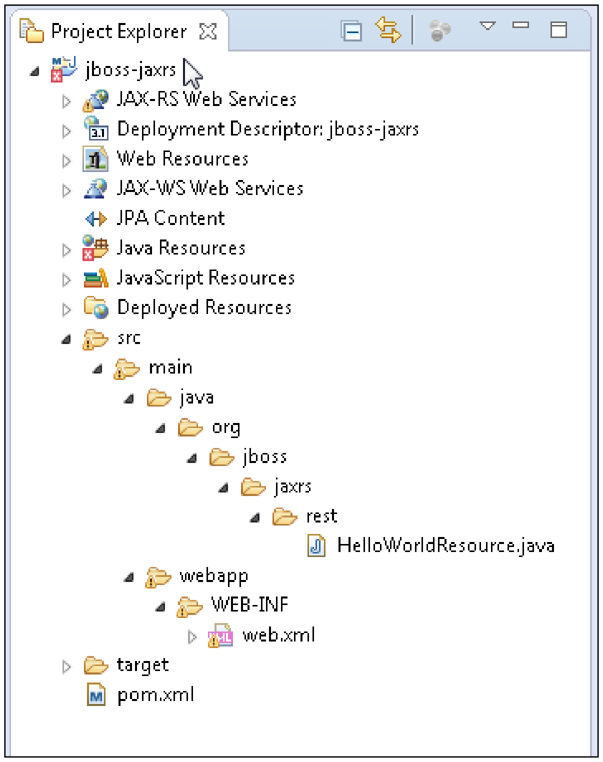
```
<description>A starter Java EE 7 webapp project for use on JBoss
WildFly / WildFly, generated from the jboss-javaee6-webapp
archetype</description>
<url>http://wildfly.org</url>
<properties>
  <!-- Explicitly declaring the source encoding eliminates the
  following
    message: -->
  <!-- [WARNING] Using platform encoding (UTF-8 actually) to
  copy filtered
    resources, i.e. build is platform dependent! -->
  <project.build.sourceEncoding>UTF-
  8</project.build.sourceEncoding>
  <!-- JBoss dependency versions -->
  <version.wildfly.maven.plugin>1.0.2.Final
  </version.wildfly.maven.plugin>
  <!-- Define the version of the JBoss BOMs we want to import to
  specify
    tested stacks. -->
  <version.jboss.bom>8.1.0.Final</version.jboss.bom>
  <version.arquillian.container>8.1.0.Final
  </version.arquillian.container>
  <!-- other plugin versions -->
  <version.compiler.plugin>3.1</version.compiler.plugin>
  <version.war.plugin>2.1.1</version.war.plugin>
  <!-- maven-compiler-plugin -->
  <maven.compiler.target>1.7</maven.compiler.target>
  <maven.compiler.source>1.7</maven.compiler.source>
</properties>
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.wildfly.bom</groupId>
      <artifactId>jboss-javaee-7.0-with-tools</artifactId>
      <version>${version.jboss.bom}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
    <dependency>
      <groupId>org.jboss.spec</groupId>
      <artifactId>jboss-javaee-web-7.0</artifactId>
      <version>1.0.0.Final</version>
      <type>pom</type>
```

```
            <scope>import</scope>
          </dependency>
        </dependencies>
      </dependencyManagement>
      <dependencies>
        <!-- First declare the APIs we depend on and need for
        compilation. All of them are provided by JBoss WildFly -->
        <!-- Import the CDI API, we use provided scope as the API is
        included in
          JBoss WildFly -->
        <dependency>
          <groupId>javax.enterprise</groupId>
          <artifactId>cdi-api</artifactId>
          <scope>provided</scope>
        </dependency>
        <!-- Import the Common Annotations API (JSR-250), we use
        provided scope as the API is included in JBoss WildFly -->
        <dependency>
          <groupId>org.jboss.spec.javax.annotation</groupId>
          <artifactId>jboss-annotations-api_1.2_spec</artifactId>
          <scope>provided</scope>
        </dependency>
        <!-- Import the JAX-RS API, we use provided scope as the API
        is included in JBoss WildFly -->
        <dependency>
          <groupId>javax.servlet</groupId>
          <artifactId>javax.servlet-api</artifactId>
          <version>3.1.0</version>
        </dependency>
        <dependency>
          <groupId>org.jboss.spec.javax.ws.rs</groupId>
          <artifactId>jboss-jaxrs-api_1.1_spec</artifactId>
          <version>1.0.1.Final</version>
          <scope>provided</scope>
        </dependency>
        <dependency>
          <groupId>com.sun.jersey</groupId>
          <artifactId>jersey-server</artifactId>
          <version>1.18</version>
        </dependency>
        <dependency>
          <groupId>com.sun.jersey</groupId>
          <artifactId>jersey-core</artifactId>
      <version>1.18</version>
    </dependency>
    <dependency>
      <groupId>com.sun.jersey</groupId>
      <artifactId>jersey-servlet</artifactId>
      <version>1.18</version>
    </dependency>
    <dependency>
      <groupId>com.sun.jersey</groupId>
      <artifactId>jersey-client</artifactId>
      <version>1.18</version>
    </dependency>
  </dependencies>
  <build>
    <!-- Maven will append the version to the finalName (which is
    the name given to the generated war, and hence the context
    root) -->
    <finalName>${project.artifactId}</finalName>
    <pluginManagement>
      <plugins>
        <!-- Compiler plugin enforces Java 1.6 compatibility and
        activates annotation processors -->
        <plugin>
          <groupId>org.eclipse.m2e</groupId>
          <artifactId>lifecycle-mapping</artifactId>
          <version>1.0.0</version>
          <configuration>
            <lifecycleMappingMetadata>
              <pluginExecutions>
                <pluginExecution>
                  <pluginExecutionFilter>
                    <groupId>org.jvnet.jax-ws-commons</groupId>
                    <artifactId>jaxws-maven-plugin</artifactId>
                    <versionRange>[2.2,)</versionRange>
                    <goals>
                      <goal>wsimport</goal>
                    </goals>
                  </pluginExecutionFilter>
                  <action>
                    <ignore />
                  </action>
                </pluginExecution>
        </pluginExecutions>
      </lifecycleMappingMetadata>
    </configuration>
  </plugin>
  <plugin>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>${version.compiler.plugin}</version>
    <configuration>
      <source>${maven.compiler.source}</source>
      <target>${maven.compiler.target}</target>
    </configuration>
  </plugin>
  <plugin>
    <artifactId>maven-war-plugin</artifactId>
    <version>${version.war.plugin}</version>
    <configuration>
      <outputDirectory>C:\wildfly-
      8.1.0.Final\standalone\deployments</outputDirectory>
      <!-- Java EE 7 doesn't require web.xml, Maven needs to
      catch up! -->
      <failOnMissingWebXml>false</failOnMissingWebXml>
    </configuration>
  </plugin>
  <!-- The WildFly plugin deploys your war to a local
  WildFly container -->
  <!-- To use, run: mvn package wildfly:deploy -->
  <plugin>
    <groupId>org.wildfly.plugins</groupId>
    <artifactId>wildfly-maven-plugin</artifactId>
    <version>${version.wildfly.maven.plugin}</version>
  </plugin>
  <plugin>
    <groupId>org.jvnet.jax-ws-commons</groupId>
    <artifactId>jaxws-maven-plugin</artifactId>
    <version>2.2</version>
    <executions>
      <execution>
        <id>HelloWorldService</id>
        <phase>compile</phase>
        <goals>
          <goal>wsgen</goal>
        </goals>
```

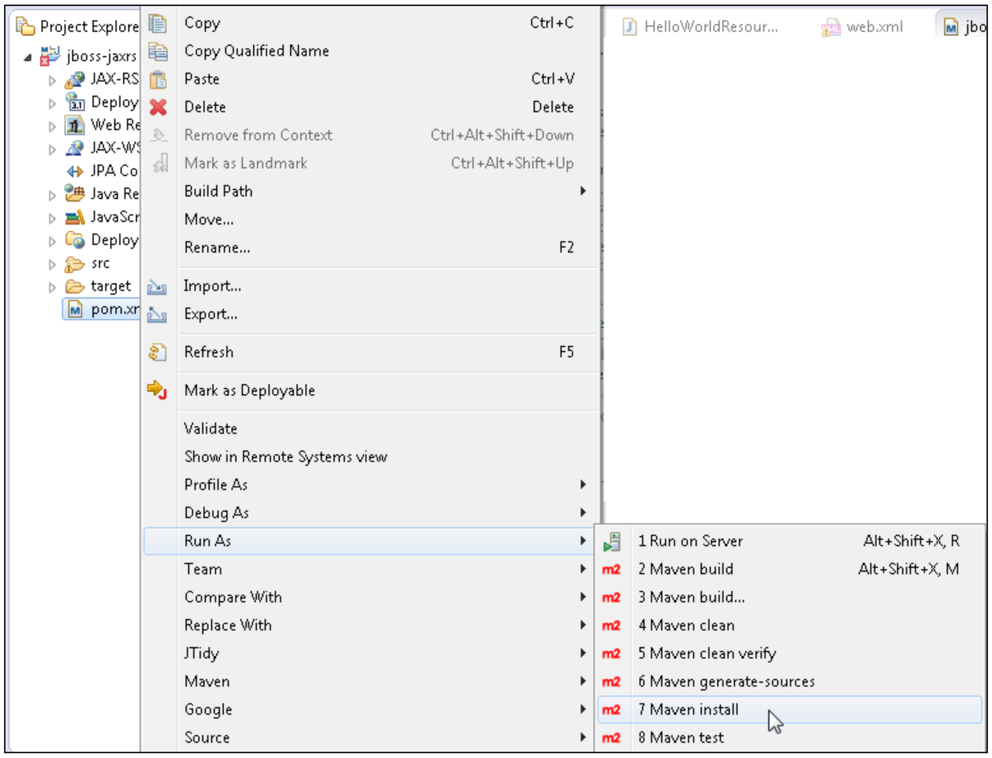
```
            <configuration>
              <sei>org.jboss.jaxws.service.HelloWorld</sei>
              <genwsdl>true</genwsdl>
              <servicename>HelloWorldService</servicename>
              <keep>true</keep>
            </configuration>
          </execution>
        </executions>
      </plugin>
      <plugin>
        <groupId>org.jvnet.jax-ws-commons</groupId>
        <artifactId>jaxws-maven-plugin</artifactId>
        <version>2.3</version>
        <executions>
          <execution>
            <id>HelloWorldService</id>
            <goals>
              <goal>wsimport</goal>
            </goals>
          </execution>
        </executions>
        <configuration>
          <packagename>org.jboss.jaxws.service</packagename>
          <target>2.0</target>
          <keep>true</keep>
        </configuration>
        <dependencies>
          <dependency>
            <groupId>com.sun.xml.ws</groupId>
            <artifactId>jaxws-tools</artifactId>
            <version>2.2.8</version>
            <exclusions>
              <exclusion>
                <groupId>org.jvnet.staxex</groupId>
                <artifactId>stax-ex</artifactId>
              </exclusion>
            </exclusions>
          </dependency>
          <dependency>
            <groupId>org.jvnet.staxex</groupId>
            <artifactId>stax-ex</artifactId>
            <version>1.7.6</version>
```

```
                  <exclusions>
                    <exclusion>
                      <groupId>javax.xml.stream</groupId>
                      <artifactId>stax-api</artifactId>
                    </exclusion>
                  </exclusions>
                </dependency>
              </dependencies>
            </plugin>
          </plugins>
        </pluginManagement>
      </build>
    </project>
```

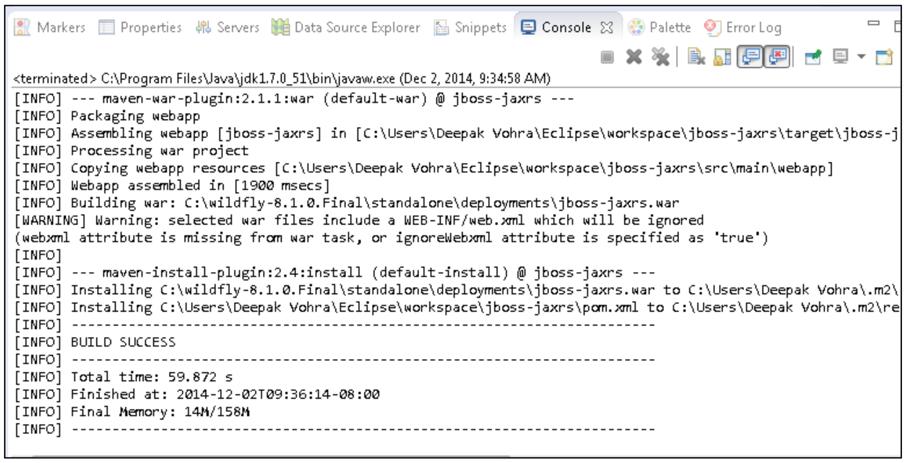
Before we install the `jboss-jaxrs` application, delete `org.jboss.jaxrs.rest.HelloRESTApplication.java`, which gets created automatically. Also, delete the resources and test directories. The directory structure of the `jboss-jaxrs` application is shown here:



Right-click on `pom.xml` and select **Run As** | **Maven install**, as shown here:

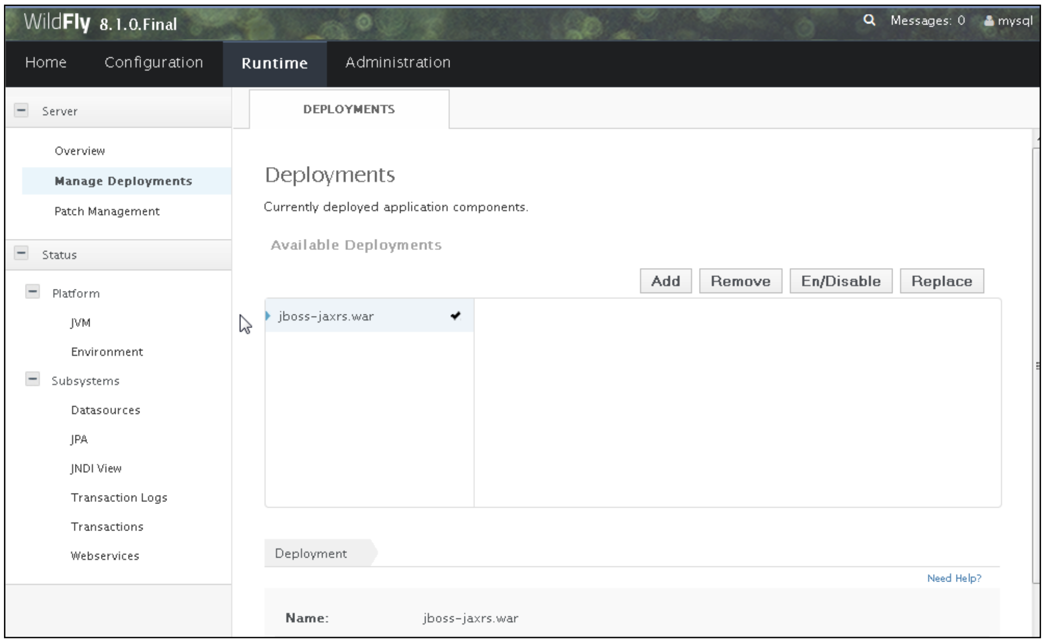


The `jboss-jaxrs` application gets compiled, packaged into `jboss-jaxrs.war` and outputs to the `deployments` directory. **Maven install** outputs the message `BUILD SUCCESS`, as shown here:



```
<terminated> C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Dec 2, 2014, 9:34:58 AM)
[INFO] --- maven-war-plugin:2.1.1:war (default-war) @ jboss-jaxrs ---
[INFO] Packaging webapp
[INFO] Assembling webapp [jboss-jaxrs] in [C:\Users\Deepak Vohra\Eclipse\workspace\jboss-jaxrs\target\jboss-j
[INFO] Processing war project
[INFO] Copying webapp resources [C:\Users\Deepak Vohra\Eclipse\workspace\jboss-jaxrs\src\main\webapp]
[INFO] Webapp assembled in [1900 msecs]
[INFO] Building war: C:\wildfly-8.1.0.Final\standalone\deployments\jboss-jaxrs.war
[WARNING] Warning: selected war files include a WEB-INF/web.xml which will be ignored
(webxml attribute is missing from war task, or ignoreWebxml attribute is specified as 'true')
[INFO]
[INFO] --- maven-install-plugin:2.4:install (default-install) @ jboss-jaxrs ---
[INFO] Installing C:\wildfly-8.1.0.Final\standalone\deployments\jboss-jaxrs.war to C:\Users\Deepak Vohra\.m2\
[INFO] Installing C:\Users\Deepak Vohra\Eclipse\workspace\jboss-jaxrs\pom.xml to C:\Users\Deepak Vohra\.m2\re
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 59.872 s
[INFO] Finished at: 2014-12-02T09:36:14-08:00
[INFO] Final Memory: 14M/158M
[INFO] ------------------------------------------------------------------------
```

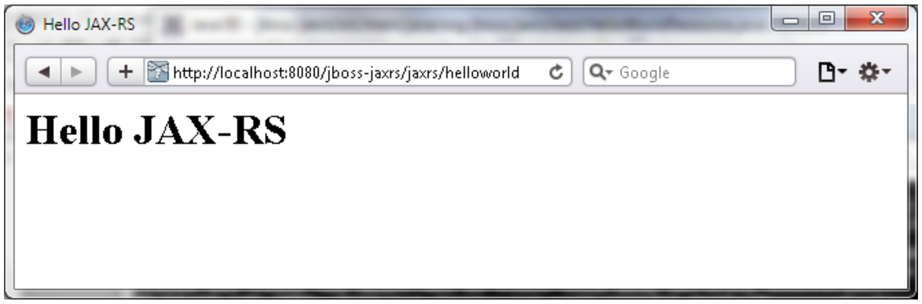
Start the WildFly 8.1 server if not already started. By default, the root resource classes are scanned for in the `WEB-INF/lib` and `WEB-INF/classes` directories. The `org.jboss.jaxrs.rest.HelloWorldResource` root resource class gets found. The `jboss-jaxrs.war` application gets deployed to the server. Now, log in to the WildFly 8.1 **Administration** console and click on **Manage Deployments**. The `jboss-jaxrs.war` file should be listed as deployed, which is shown as follows:



Running the RESTful web service

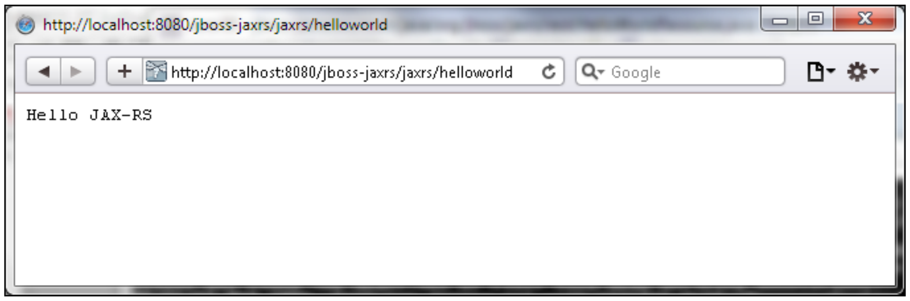
In this section, we will run the root resource class. Invoke the URL `http://localhost:8080/jboss-jaxrs/jaxrs/helloworld`. The `jboss-jaxrs` is included in the URL as it is the context of the web application. The `jaxrs` is included to invoke the `servlet com.sun.jersey.spi.container.servlet.ServletContainer`. The `/helloworld` is the resource URI for the RESTful web service. We did not annotate any resource method or class method with `@PATH`; therefore, we won't invoke a specific method. The first resource method in the root resource class gets invoked. Make different resource methods as the first in the root resource class to test different MIME type outputs. If HTML output is required, make the `getHTMLMessage` method the first in the root resource class.

The HTML output in the browser is shown as follows:



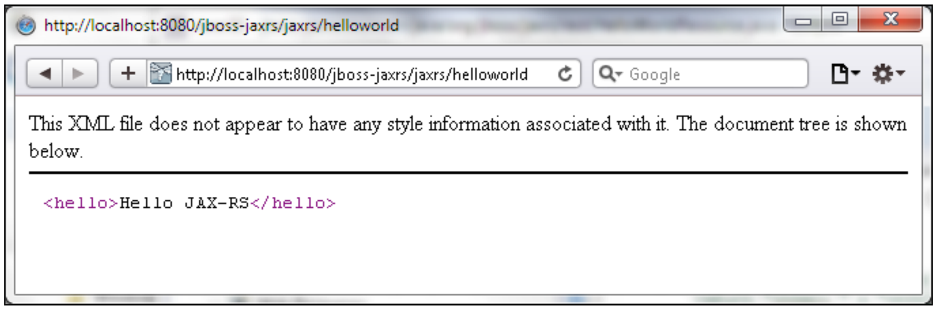
Hello JAX-RS

If the `getClichedMessage()` resource method is made sequentially the first in the resource class, the `text/plain` representation gets displayed in the browser when the URL `http://localhost:8080/jboss-jaxrs/jaxrs/helloworld` is invoked as shown here:


http://localhost:8080/jboss-jaxrs/jaxrs/helloworld
http://localhost:8080/jboss-jaxrs/jaxrs/helloworld
Google

```
Hello JAX-RS
```

If the `getXMLMessage()` resource method is made sequentially the first in the resource class, the XML representation gets displayed in the browser as shown here:


http://localhost:8080/jboss-jaxrs/jaxrs/helloworld
http://localhost:8080/jboss-jaxrs/jaxrs/helloworld
Google

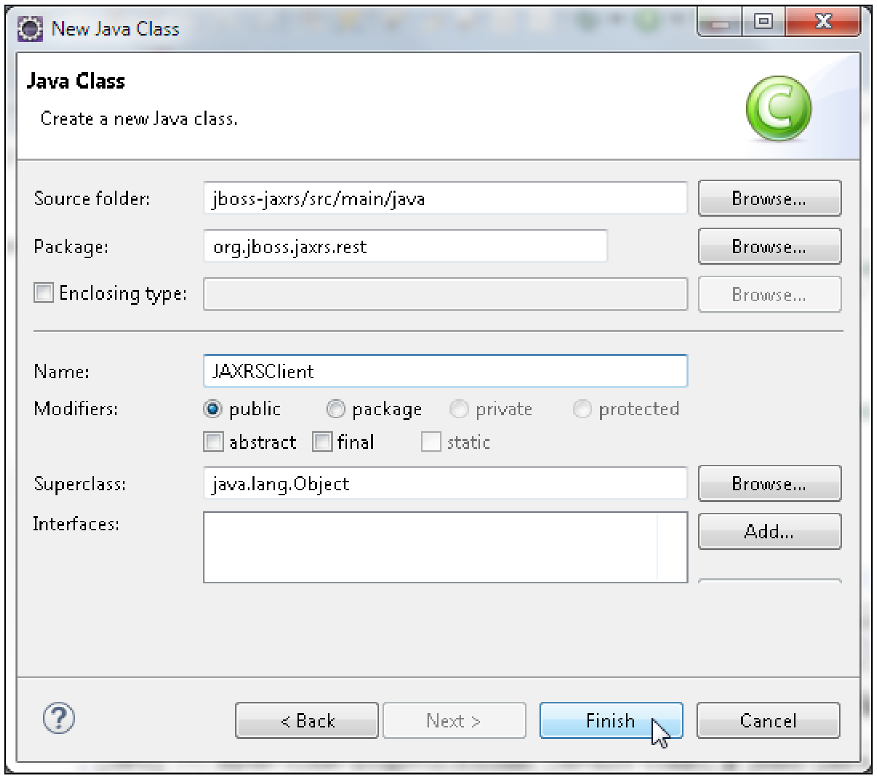
This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
<hello>Hello JAX-RS</hello>
```

When a different method is invoked in the resource class after making a modification, the `jboss-jaxrs` application would need to be reinstalled. To reinstall, first the Maven project must be cleaned for which you need to right-click on `pom.xml` and select **Run As** | **Maven clean.** A `BUILD SUCCESS` message indicates that the files generated from the previous installation have been removed.

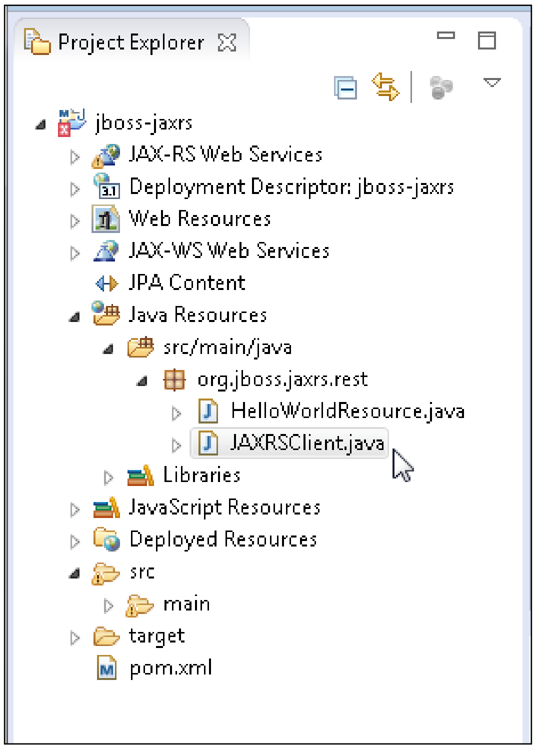
Creating a Java client

In this section, we will use the Jersey client API to create a JAX-RS Web Service client. Create a Java class for a Java client. Select **File** | **New** | **Other**. In **New**, select **Class** and click on **Next**. In the **New Java Class** wizard, select **Source folder** as `src/main/java`, specify **Package** as `org.jboss.jaxrs.rest`, and **Name** as `JAXRSClient`, which is shown as follows. Then click on **Finish**.



Source folder: jboss-jaxrs/src/main/java
Package: org.jboss.jaxrs.rest
Enclosing type: [ ]
Name: JAXRSClient
Modifiers: [✓ public] package private protected
abstract final static
Superclass: java.lang.Object
Interfaces:

The `org.jboss.jaxrs.rest.JAXRSClient` class gets created, as shown here:



Jersey provides a client API to invoke a JAX-RS Web Service. In the Java client, invoke the resource methods using the Jersey client API. Create a resource instance using the `com.sun.jersey.api.client.Client` class. First, we need to create a `ClientConfig` object, which represents the client configuration such as property names and features using the no-arguments constructor for `DefaultClientConfig`. Create a `Client` object from the `ClientConfig` object using the static method `create(ClientConfig)`. Create a `WebResource` object from the `Client` object using the `resource(URI)` method of the `Client` object with the `URI` argument being the base URI for the JAX-RS Web Service, which may be obtained with the `getBaseURI()` method. A `WebResource` object is an encapsulation of a web resource and is used to send requests to and receive responses from the web resource:

```
ClientConfig config = new DefaultClientConfig();
Client client = Client.create(config);
WebResource service = client.resource(getBaseURI());
```

Add a method `getBaseURI()` to return the base URI. The base URI is obtained with the `UriBuilder` class using the static method `fromUri` for the URL `http://localhost:8080/jboss-jaxrs` and subsequently the `build` method to build a `URI` object:

```
private static URI getBaseURI() {
    return UriBuilder.fromUri("http://localhost:8080/jboss-jaxrs")
  .build();
}
```

Get the web service response by invoking the `get()` method on a `WebResource` object with `String.class` as argument, as the return type of the resource methods is `String`. First, add paths to the base URI of the `WebResource` object using the `path()` method, which returns a new `WebResource` object with the path added. Add `jaxrs` to the URI path to invoke the `com.sun.jersey.spi.container.servlet.ServletContainer` servlet and add `helloworld` to the URI path to invoke the `HelloWorldResource` resource. Add acceptable media types to the `WebResource` object using the `accept()` method. We will test `HelloWorldResource` using three different media types: `TEXT_PLAIN`, `TEXT_XML`, and `TEXT_HTML`. For example, a `TEXT_PLAIN` response output is as follows:

```
System.out.println(service.path("jaxrs").path("helloworld").accept
(MediaType.TEXT_PLAIN).get(String.class));
```

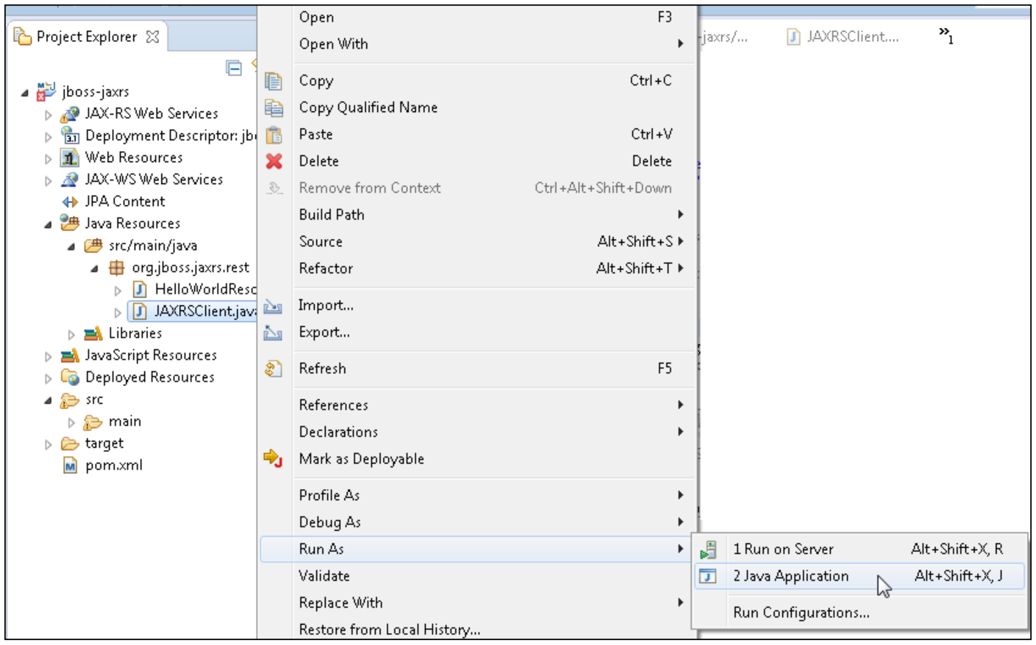
Similarly, output the response as XML and HTML using the `MediaType.TEXT_XML` and `MediaType.TEXT_HTML` media types. The `JAXRSClient.java` class is listed as follows:

```
package org.jboss.jaxrs.rest;

import java.net.URI;
import javax.ws.rs.core.*;
import com.sun.jersey.api.client.*;
import com.sun.jersey.api.client.config.*;

public class JAXRSClient {
  public static void main(String[] args) {
    ClientConfig config = new DefaultClientConfig();
    Client client = Client.create(config);
    WebResource service = client.resource(getBaseURI());
  System.out.println(service.path("jaxrs").path("helloworld").accept
(MediaType.TEXT_PLAIN).get(String.class));
  System.out.println(service.path("jaxrs").path("helloworld").accept
(MediaType.TEXT_XML).get(String.class));
```

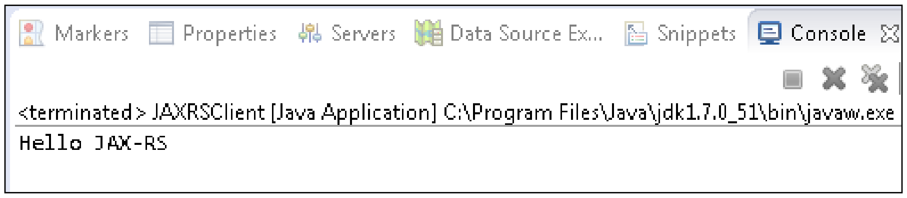
```
      System.out.println(service.path("jaxrs").path("helloworld").accept
    (MediaType.TEXT_HTML).get(String.class));
      }
      private static URI getBaseURI() {
        return UriBuilder.fromUri("http://localhost:8080/jboss-
        jaxrs").build();
      }
    }
```

Running the Java client

In this section, we will run the Java client. First, run the client to test outputting just the media type `TEXT_PLAIN`. The acceptable media type in the client class must match a MIME media type produced in the resource class. In the `JAXRSClient` class, uncomment only the `System.out` statement that accepts the `TEXT_PLAIN` media type. Uncomment all the resource methods in the `HelloWorldResource` class. Then right-click on `JAXRSClient.java` and select **Run As | Java Application**, as shown here:



The resource response gets displayed in the `TEXT_PLAIN` media type, as shown here:



```
<terminated> JAXRSClient [Java Application] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe
Hello JAX-RS
```

An acceptable media type must match a media type produced by the resource class. If a produced MIME media type in the root resource class is not found for an acceptable media type, a `com.sun.jersey.api.client.UniformInterfaceException` exception is generated. For example, set an acceptable media type to `MediaType.TEXT_XML` by uncommenting the following `System.out` statement in the `JAXRSClient` class:

```
System.out.println(service.path("jaxrs").path("helloworld").accept
(MediaType.TEXT_XML).get(String.class));
```

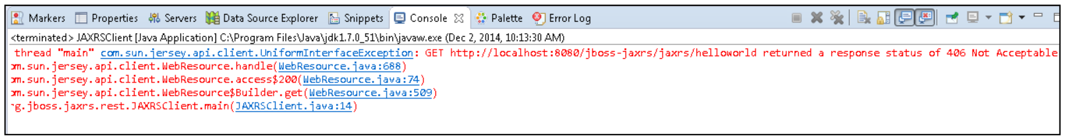
The `System.out` statements for the other acceptable media types may be kept uncommented or commented out as each resource method is invoked independent of the other resource methods. In `HelloWorldResource`, comment out the following resource method:

```
@GET
  @Produces("text/xml")
  public String getXMLMessage() {
    return "<?xml version=\"1.0\"?>" + "<hello> Hello JAX-RS" +
    "</hello>";
  }
```

And uncomment the resource method that produces the MIME media type `application/xml`:

```
@GET
  @Produces("application/xml")
  public String getXMLMessage() {
    return "<?xml version=\"1.0\"?>" + "<hello> Hello JAX-RS" +
    "</hello>";
  }
```

Redeploy the Maven application after making any modification to the application. To redeploy right-click on `pom.xml` in **Project Explorer** and select **Run As** | **Maven clean**. Subsequently, right-click on `pom.xml` and select **Run As** | **Maven install**. Run `JAXRSClient.java` as a Java application. The `UniformInterfaceException` exception gets generated. A response status of `406` (`http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html`) is outputted, as shown here:

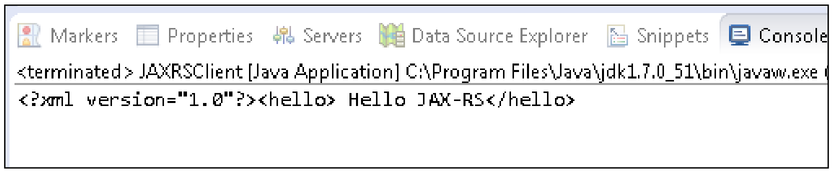


```
<terminated> JAXRSClient [Java Application] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Dec 2, 2014, 10:13:30 AM)
 thread "main" com.sun.jersey.api.client.UniformInterfaceException: GET http://localhost:8080/jboss-jaxrs/jaxrs/helloworld returned a response status of 406 Not Acceptable
m.sun.jersey.api.client.WebResource.handle(WebResource.java:688)
m.sun.jersey.api.client.WebResource.access$200(WebResource.java:74)
m.sun.jersey.api.client.WebResource$Builder.get(WebResource.java:509)
g.jboss.jaxrs.rest.JAXRSClient.main(JAXRSClient.java:14)
```

The error can be removed by uncommenting the resource method that produces the MIME media type `text/xml` and commenting out the method that produces the MIME type `application/xml`:

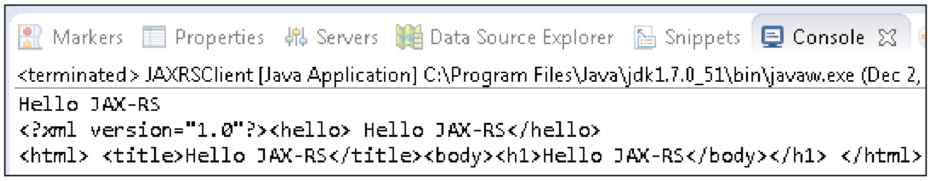
```
@GET
  @Produces("text/xml")
  public String getXMLMessage() {
    return "<?xml version=\"1.0\"?>" + "<hello> Hello JAX-RS" +
    "</hello>";
  }
```

If the `JAXRSClient.java` application is run by making the acceptable media type as `MediaType.TEXT_XML` and by uncommenting the resource method that produces the `text/xml` media type, the following output gets generated:



```
<terminated> JAXRSClient [Java Application] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe
<?xml version="1.0"?><hello> Hello JAX-RS</hello>
```

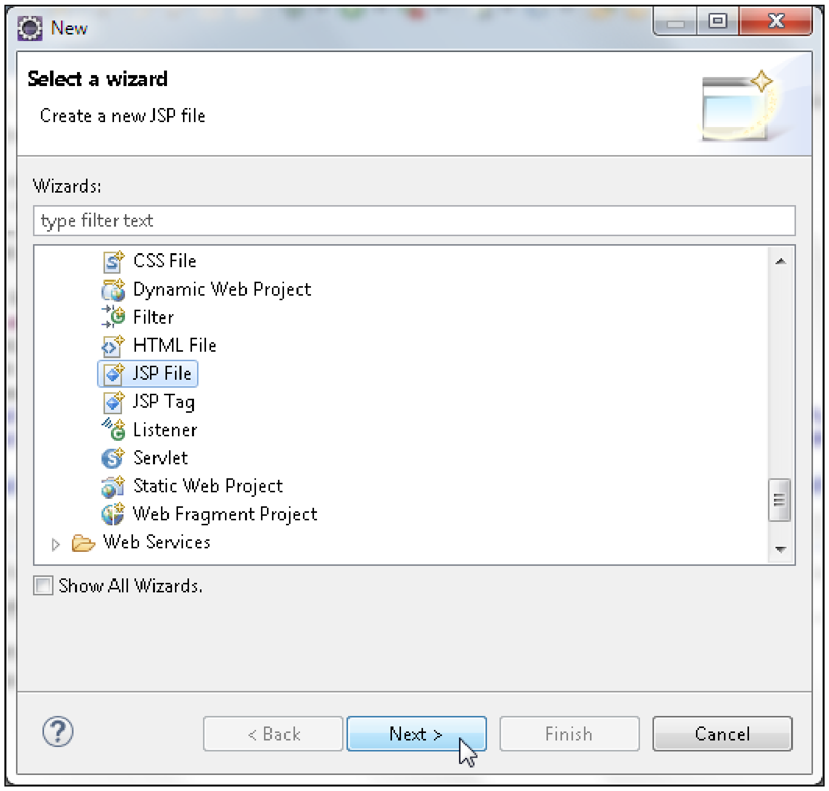
When we did not use a client for the root resource class, we were able to invoke only one resource method at a time. The resource method that produces the MIME media type that matches the most acceptable media type in the client gets invoked. Multiple MIME media types may be specified in the same `@Produces` annotation instance. If more than one media types are equally acceptable, the one specified sequentially first gets selected. Next, we will invoke all the resource methods from the client, uncomment all the resource methods in the `HelloWorldResource` class, and uncomment the `System.out` statements for all the acceptable media types in the `JAXRSClient` class. Run the `JAXRSClient` class as a Java application. Response from all the resource methods in three different media types gets an output, as shown here:



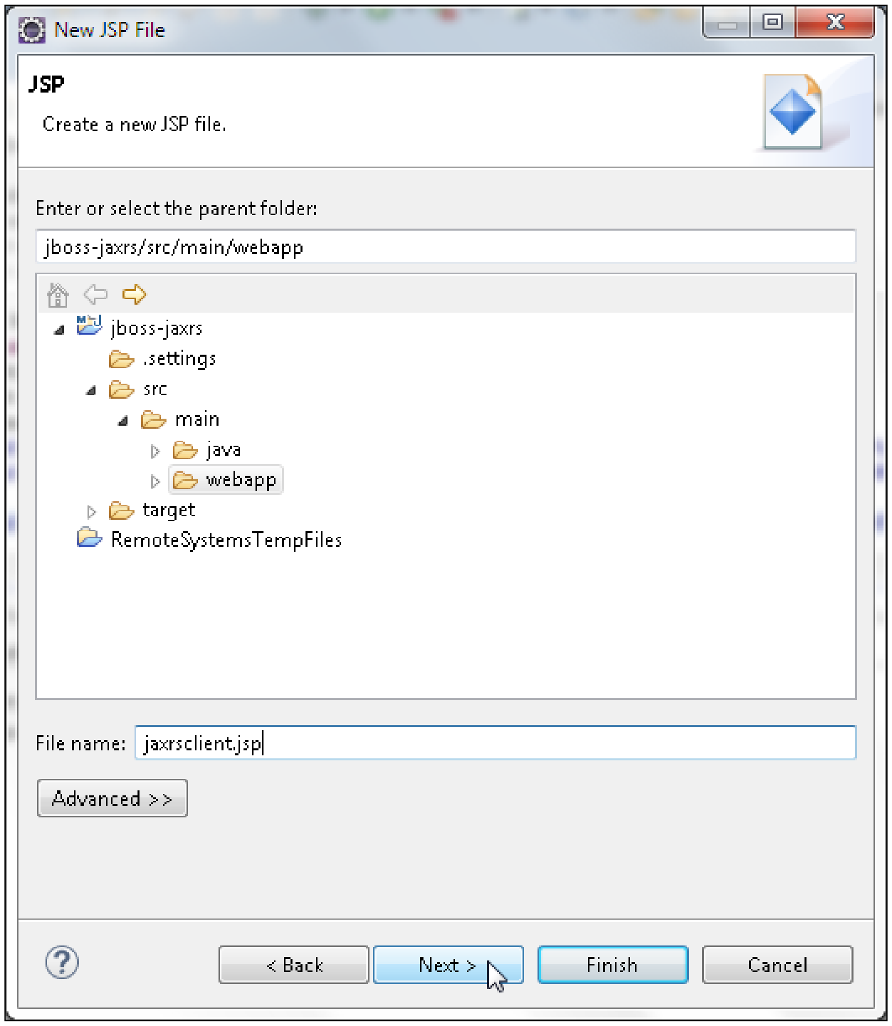
```
<terminated> JAXRSClient [Java Application] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Dec 2,
Hello JAX-RS
<?xml version="1.0"?><hello> Hello JAX-RS</hello>
<html> <title>Hello JAX-RS</title><body><h1>Hello JAX-RS</body></h1> </html>
```

Creating a JSP client

In the previous section, we used a java client for the root resource class. To package a client with the web application, a browser-based client such as a JSP client will be required. In this section, we will create a JSP client for the root resource class. Select **File** | **New** | **Other**, and in **New**, select **Web** | **JSP File** and click on **Next**, as shown here:



In **New JSP File** wizard, select the `webapp` folder and specify **File name** as `jaxrsclient.jsp`, as shown here. Now click on **Next**.


New JSP File

JSP

Create a new JSP file.

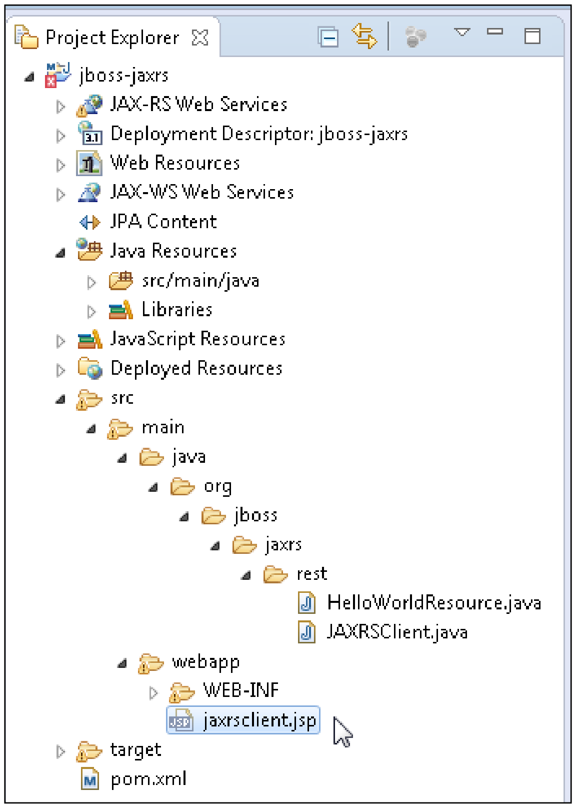
Enter or select the parent folder:

jboss-jaxrs/src/main/webapp

```
jboss-jaxrs
    .settings
    src
        main
            java
            webapp
    target
RemoteSystemsTempFiles
```

File name: jaxrsclient.jsp

Select the **New JSP file (html)** template, click on **Finish**. The `jaxrsclient.jsp` file gets added to the `webapp` folder, as shown here:



In the `jaxrsclient.jsp` JSP client, the root resource class resource methods are invoked as in the Java client. The `jaxrsclient.jsp` file is listed here:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-
1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<%@ page import="java.net.URI,javax.ws.rs.core.*,com.sun.jersey.api.
client.
*,com.sun.jersey.api.client.config.*"%>
<html>
<head>
<meta http-equiv="Content-Type"
    content="text/xml; charset=windows-1252" />
<title>JAX-RS Client</title>
</head>
<body>
    <%
```

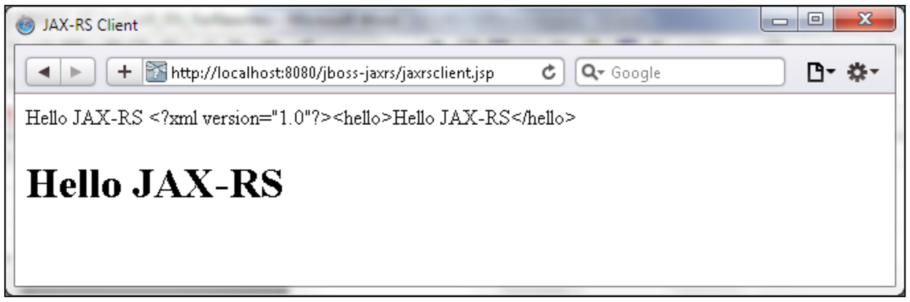
```
            ClientConfig clientconfig = new DefaultClientConfig();
            Client client = Client.create(clientconfig);
            WebResource service =
            client.resource(UriBuilder.fromUri
            ("http://localhost:8080/jboss-jaxrs").build());
        out.println(service.path("jaxrs").path("helloworld").
        accept(MediaType.TEXT_PLAIN).get(String.class));
        out.println(service.path("jaxrs").path("helloworld").
        accept(MediaType.TEXT_XML).get(String.class));
        out.println(service.path("jaxrs").path("helloworld").
        accept(MediaType.TEXT_HTML).get(String.class));
        %>
    </body>
    </html>
```

Running the JSP client

In this section, we will run the `jaxrsclient.jsp` file. Before we run the JSP client, we need to make a slight modification in the root resource class to output an XML string. Modify the `getXMLMessage()` method as follows:

```
@GET
  @Produces("text/xml")
  public String getXMLMessage() {
    return "&lt;?xml version=\"1.0\"?&gt;" + "&lt;hello&gt;Hello
    JAX-RS" + "&lt;/hello&gt;";
  }
```

Start WildFly 8.1.1 if not already started. With all the resource methods uncommented in the root resource class and all the `out.println` statements for the different acceptable media types uncommented in the JSP client, clean and redeploy/reinstall the Maven project. Invoke the URL `http://localhost:8080/jboss-jaxrs/jaxrsclient.jsp` in a browser. All the resource methods get invoked and three different media types get an output, as shown here:



```
Hello JAX-RS <?xml version="1.0"?><hello>Hello JAX-RS</hello>
```

Hello JAX-RS

Summary

In this chapter, we developed a JAX-RS RESTful web service using the Jersey JAX-RS RI. We created a Java EE Web Project for the RESTful web service. First, we created a root resource class with three resource methods to produce three different media types. We compiled, packaged, and deployed the `jboss-jaxrs` application to WildFly 8.1. We tested the root resource class to output the different media types. Subsequently, we used a Java client to invoke the root resource class methods. We also used a JSP client to test the RESTful web service.

In the next chapter, we will discuss Spring MVC with WildFly 8.1.

8
Using Spring MVC 4.1

The Spring framework is based on the **Inversion of Control** (**IoC**) principle, in which the objects define their own dependencies. **Dependency Injection** is a form of IoC. The container injects the dependencies when the bean is created. The dependency injection is implemented using the `BeanFactory` and `ApplicationContext` interfaces. The `BeanFactory` interface is for managing beans and `ApplicationContext` (`WebApplicationContext` is used for a web application) is for configuring an application. `BeanFactory` applies the IoC pattern to separate configuration and dependency specifications. The `ApplicationContext` interface provides extended (extends) features to `BeanFactory`.

The Spring MVC framework is based on the **Model-View-Controller** design pattern and is implemented using `DispatcherServlet`. `DispatcherServlet` fields requests and delegates them to a request handler using web request URI mapping provided by the `@RequestMapping` annotation. The request handler or Controller returns `Model` and `View` using the `ModelAndView` holder. In this chapter, we will create a Spring MVC application in Eclipse, compile and package the application using the Maven build tool, and deploy the application to WildFly 8.1. Subsequently, we will run the Spring application in a browser. In this chapter, we will cover the following topics:

- Setting up the environment
- Creating a Spring MVC project
- Creating a MySQL data source
- Creating a JPA configuration file
- Creating the Model
- Creating the Data Access Object design pattern
- Creating a web descriptor
- Creating the request handler
- Creating the View

- Creating a Spring MVC context
- Deploying the Spring project with Maven
- Creating deployment structure and infrastructure deployment descriptors
- Installing the Spring MVC Maven project
- Running the Spring application

Setting up the environment

We need to install the following software:

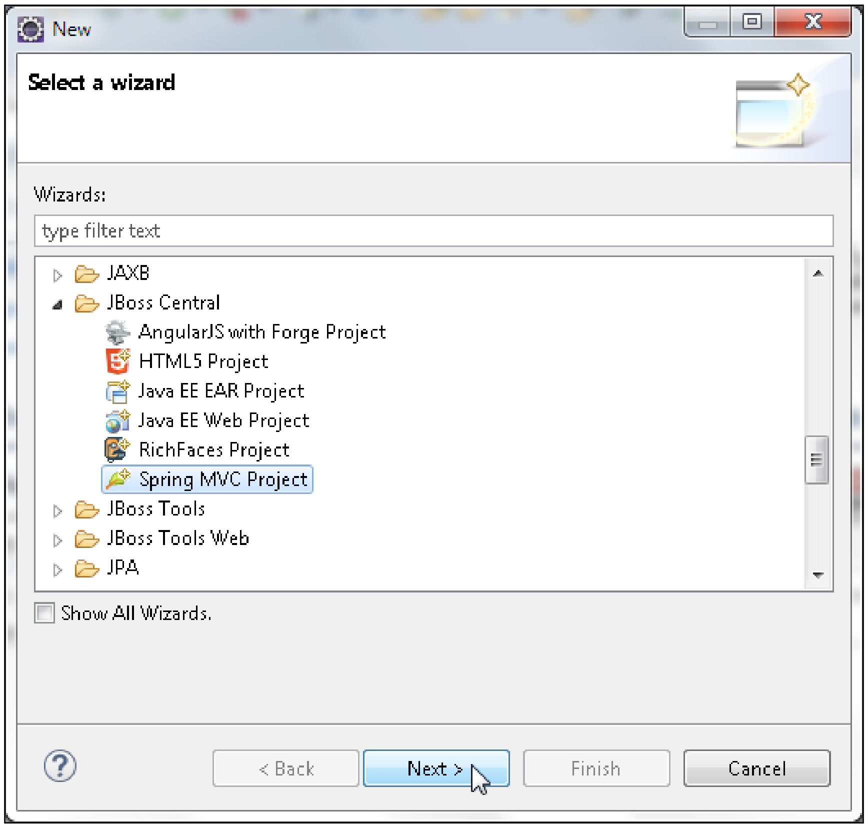
- **WildFly 8.1.0.Final**: Download `wildfly-8.1.0.Final.zip` from `http://wildfly.org/downloads/`.
- **MySQL 5.6 Database-Community Edition**: Download this edition from `http://dev.mysql.com/downloads/mysql/`. When installing MySQL, also install **Connector/J**.
- **Eclipse IDE for Java EE Developers**: Download Eclipse Luna from `https://www.eclipse.org/downloads/packages/release/Luna/SR1`.
- **JBoss Tools (Luna) 4.2.0.Final**: Install this as a plugin to Eclipse from Eclipse Marketplace (`http://tools.jboss.org/downloads/installation.html`).
- **Apache Maven**: Download version 3.05 or higher from `http://maven.apache.org/download.cgi`.
- **Java 7**: Download Java 7 from `http://www.oracle.com/technetwork/java/javase/downloads/index.html?ssSourceSiteId=ocomcn`.

Set the `JAVA_HOME`, `JBOSS_HOME`, `MAVEN_HOME`, and `MYSQL_HOME` environment variables. Add `%JAVA_HOME%/bin`, `%MAVEN_HOME%/bin`, `%JBOSS_HOME%/bin`, and `%MYSQL_HOME/bin` to the `PATH` environment variable.

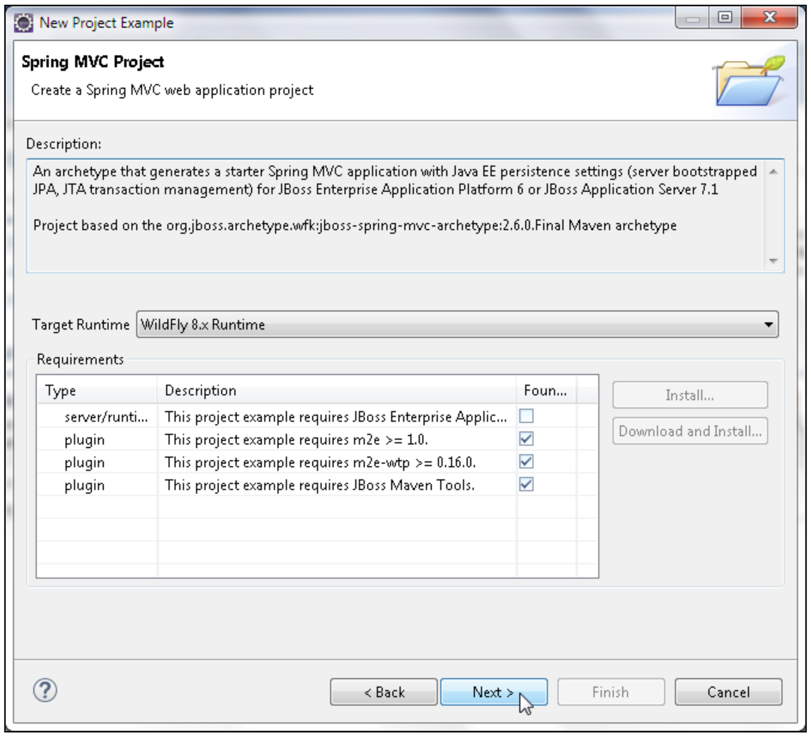
Create a WildFly 8.1.0 runtime as discussed in *Chapter 1, Getting Started with EJB 3.x*. Create a MySQL data source with the JNDI name `java:jboss/datasources/MySQLDS` as explained in *Chapter 1, Getting Started with EJB 3.x*.

Creating a Spring MVC project

First, create a Spring MVC project. Select **File** | **New** | **Other**. In **New**, select **JBoss Central** | **Spring MVC Project**, as shown in the following screenshot. Then, click on **Next**.



In the **Spring MVC Project** wizard, select **Target Runtime** as `WildFly8.x Runtime`, as shown in the following screenshot. Now, click on **Next**.



Description:

An archetype that generates a starter Spring MVC application with Java EE persistence settings (server bootstrapped JPA, JTA transaction management) for JBoss Enterprise Application Platform 6 or JBoss Application Server 7.1

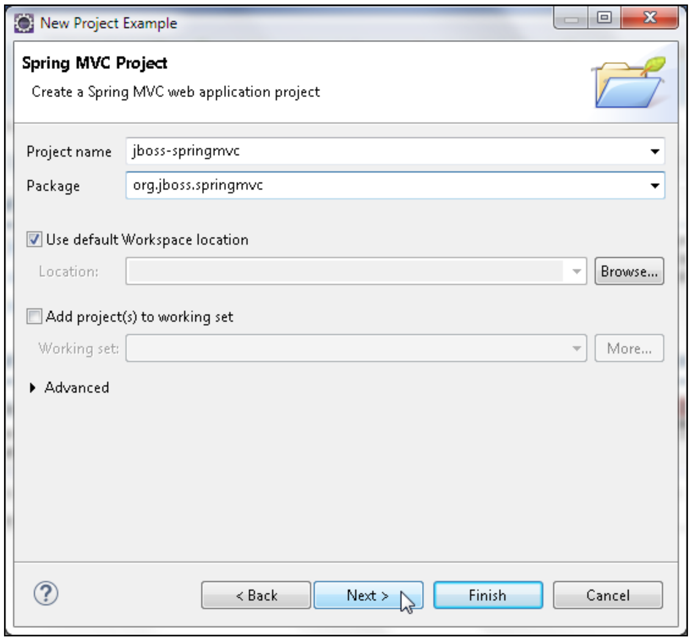
Project based on the org.jboss.archetype.wfk:jboss-spring-mvc-archetype:2.6.0.Final Maven archetype

Target Runtime: WildFly 8.x Runtime

Requirements

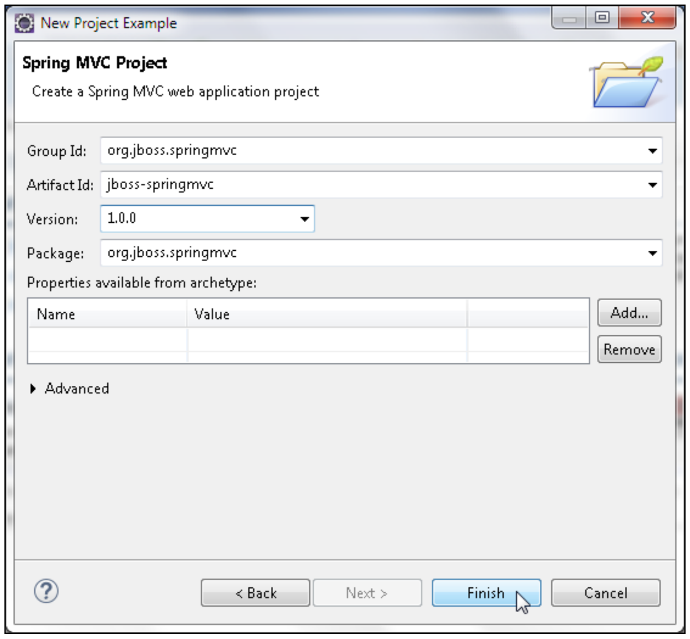
Type	Description	Foun...
server/runti...	This project example requires JBoss Enterprise Applic...	☐
plugin	This project example requires m2e >= 1.0.	✓
plugin	This project example requires m2e-wtp >= 0.16.0.	✓
plugin	This project example requires JBoss Maven Tools.	✓

Specify **Project name** (`jboss-springmvc`) and **Package** (`org.jboss.springmvc`), as shown in the following screenshot. After this, click on **Next**.



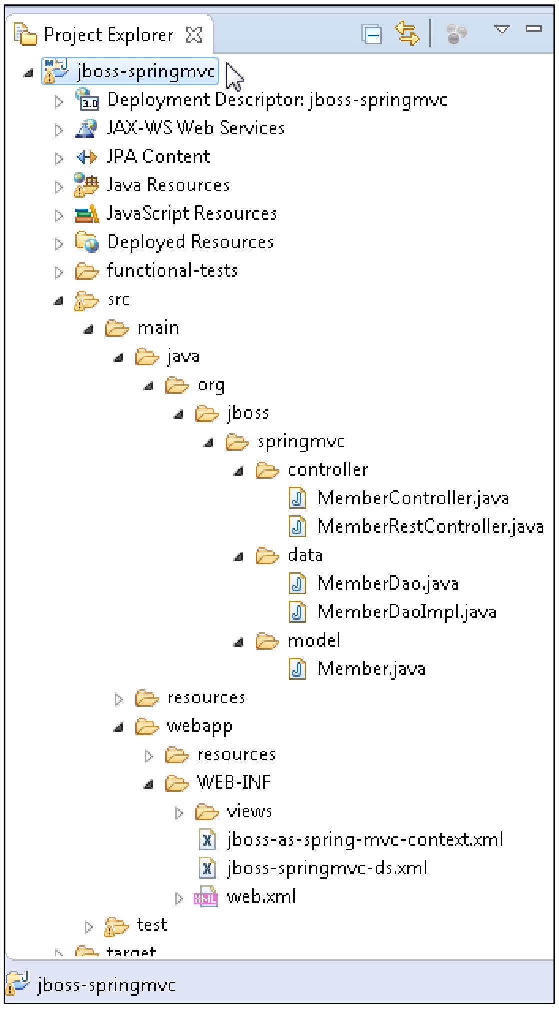
Project name: jboss-springmvc
Package: org.jboss.springmvc
[✓] Use default Workspace location
[ ] Add project(s) to working set

Specify **Group Id** (`org.jboss.springmvc`), **Artifact Id** (`springmvc`), **Version** (`1.0.0`), and **Package** (`org.jboss.springmvc`), as shown in the following screenshot, and click on **Finish**.



Group Id: org.jboss.springmvc
Artifact Id: jboss-springmvc
Version: 1.0.0
Package: org.jboss.springmvc

The `jboss-springmvc` web project gets created, as shown in the following screenshot:



```
Project Explorer
◢ jboss-springmvc
    ▷ Deployment Descriptor: jboss-springmvc
    ▷ JAX-WS Web Services
    ▷ JPA Content
    ▷ Java Resources
    ▷ JavaScript Resources
    ▷ Deployed Resources
    ▷ functional-tests
    ◢ src
        ◢ main
            ◢ java
                ◢ org
                    ◢ jboss
                        ◢ springmvc
                            ◢ controller
                                  MemberController.java
                                  MemberRestController.java
                            ◢ data
                                  MemberDao.java
                                  MemberDaoImpl.java
                            ◢ model
                                  Member.java
            ▷ resources
            ◢ webapp
                ▷ resources
                ◢ WEB-INF
                    ▷ views
                      jboss-as-spring-mvc-context.xml
                      jboss-springmvc-ds.xml
                    ▷ web.xml
        ▷ test
    ▷ target
jboss-springmvc
```

In the subsequent section, we will develop the Spring MVC application and discuss the Spring MVC application artifacts in detail; the application shown in the **Project Explorer** tab is the default application and will be replaced with the application we will develop.

Creating a JPA configuration file

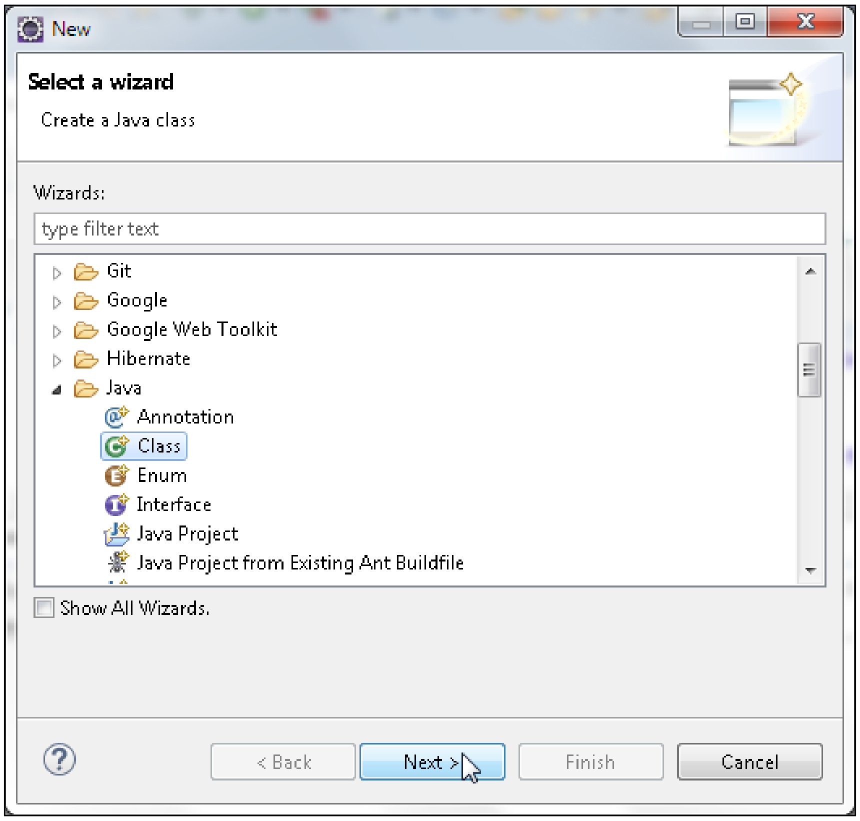
We will use an EJB 3.0 entity bean for object/relational mapping in the Spring MVC application. The `springmvc\src\main\resources\META-INF\persistence.xml` configuration file in the `jboss-springmvc` project was created when we created the Spring MVC project. The `persistence.xml` file specifies a persistence provider to be used for object/relational mapping of entities to the database. Specify a persistence unit using the `persistence-unit` element. Set `transaction-type` to `JTA` (default). Specify the persistence provider as the Hibernate persistence provider, `org.hibernate.ejb.HibernatePersistence`. Set the `jta-data-source` element value to the `java:jboss/datasources/MySQLDS` data source that we created earlier. The DDL generation strategy is set to `create-drop` using the `hibernate.hbm2ddl.auto` property. With the `create-drop` strategy, the required tables are created and dropped. The `hibernate.show_sql` property is set to `false`, implying that all SQL statements be output, which is an alternative method to debugging. The `hibernate.dialect` file is set to `org.hibernate.dialect.MySQLDialect` for the MySQL database. The `persistence.xml` configuration file is listed in the following code:

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="2.0"
xsi:schemaLocation="         http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
  <persistence-unit name="primary" transaction-type="JTA">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <jta-data-source>java:jboss/datasources/MySQLDS</jta-data-source>
    <class>org.jboss.springmvc.model.Catalog</class>
    <exclude-unlisted-classes />
    <properties>
      <property name="jboss.entity.manager.factory.jndi.name"
value="java:jboss/mysql/persistence" />
      <!-- Properties for Hibernate -->
      <property name="hibernate.hbm2ddl.auto" value="create-drop" />
      <property name="hibernate.show_sql" value="false" />
      <property name="hibernate.dialect" value="org.hibernate.dialect.
MySQLDialect" />
    </properties>
  </persistence-unit>
</persistence>
```

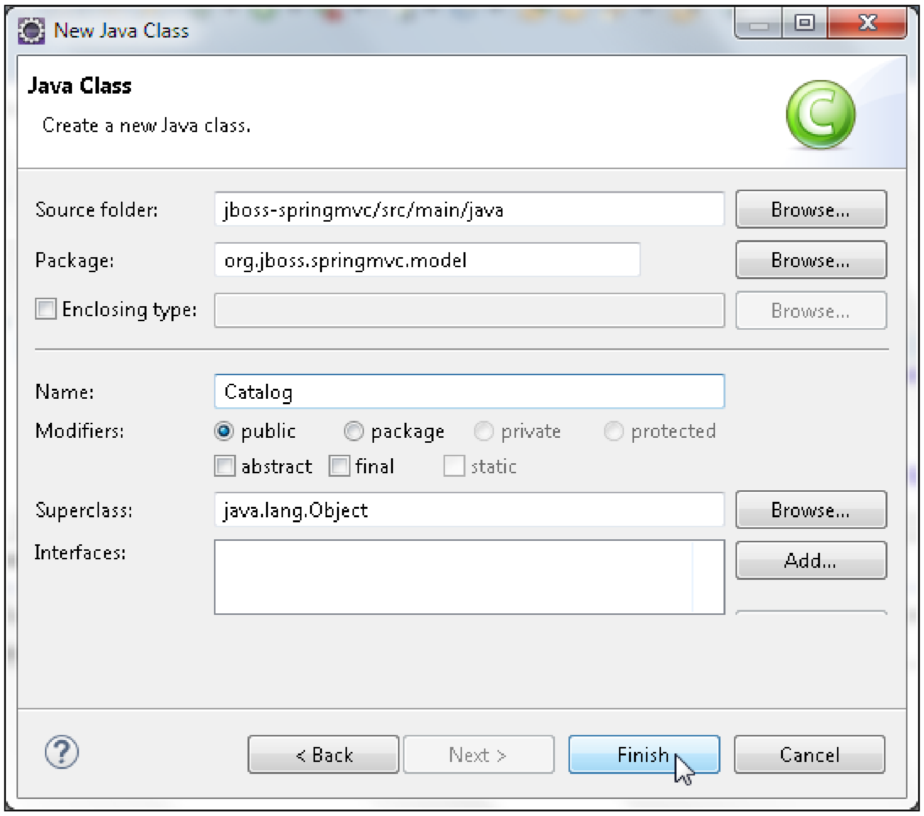
The JPA specification does not require a persistence provider to create tables even if the `hibernate.hbm2ddl.auto` property is set to `create-drop` or `create`. The Hibernate persistence provider supports creating tables.

Creating the Model

In this section, we will create an EJB 3.0 entity bean for a domain model. Create the `org.jboss.springmvc.model.Catalog` entity bean class, which is just a **plain old Java object** (**POJO**). To create a Java class, select **File** | **New** | **Other**, and in **New**, select **Java** | **Class**, as shown in the following screenshot:



In the **New Java Class** wizard, select **Source folder** as `jboss-springmvc/src/main/java`, specify **Package** as `org.jboss.springmvc.model`, and specify the class **Name** as `Catalog`, as shown in the following screenshot:



New Java Class

Java Class

Create a new Java class.

Source folder: jboss-springmvc/src/main/java

Package: org.jboss.springmvc.model

Enclosing type:

Name: Catalog

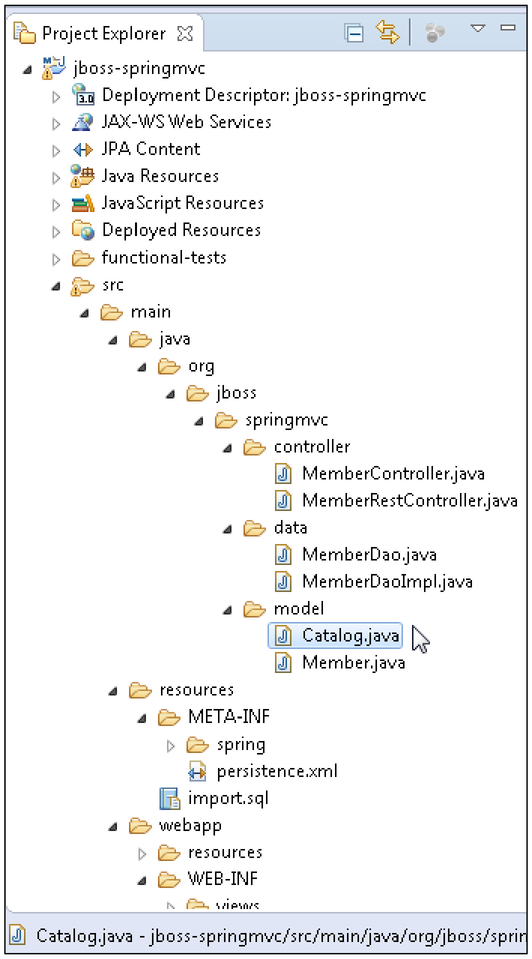
Modifiers: [✓ public] package private protected

abstract final static

Superclass: java.lang.Object

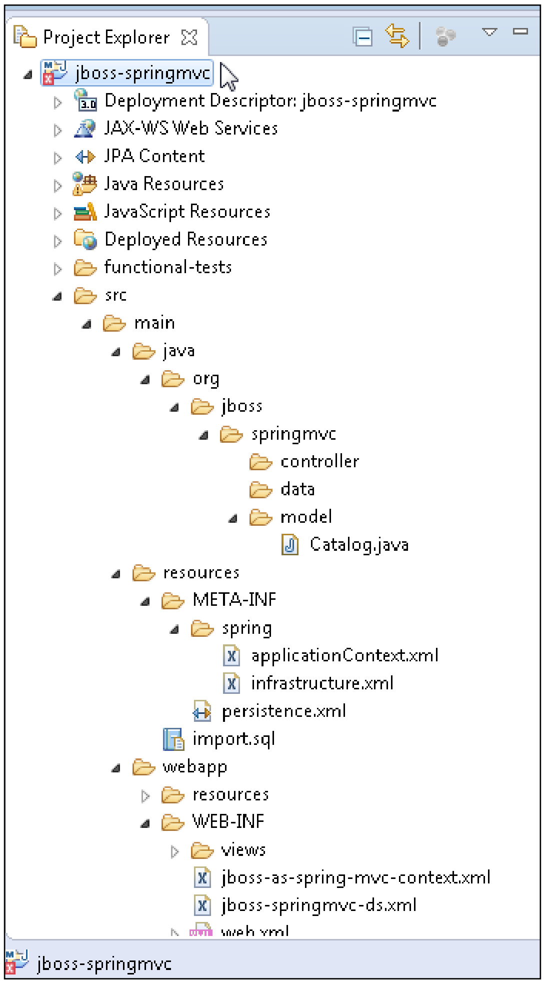
Interfaces:

The `org.jboss.springmvc.model.Catalog.java` class gets added to the project, as shown in **Project Explorer** in the following screenshot:



```
Project Explorer
jboss-springmvc
  Deployment Descriptor: jboss-springmvc
  JAX-WS Web Services
  JPA Content
  Java Resources
  JavaScript Resources
  Deployed Resources
  functional-tests
  src
    main
      java
        org
          jboss
            springmvc
              controller
                MemberController.java
                MemberRestController.java
              data
                MemberDao.java
                MemberDaoImpl.java
              model
                Catalog.java
                Member.java
      resources
        META-INF
          spring
          persistence.xml
        import.sql
      webapp
        resources
        WEB-INF
          views
Catalog.java - jboss-springmvc/src/main/java/org/jboss/sprin
```

Remove the `Member.java` class and the other `Member*` classes from the project to get the directory structure, as shown in the following screenshot:



Annotate the `Catalog` entity class with the `@Entity` annotation and the `@Table` annotation. In the `@Table` annotation, specify the table name as `CATALOG` and `uniqueConstraints` using the `@UniqueConstraint` annotation for the `ID` column. Specify the no-argument constructor, which is required in an entity class. The entity class implements the `Serializable` interface to serialize a cache-enabled entity bean to a cache when persisted to a database. To associate a version number with a serializable class by serialization runtime, specify a `serialVersionUID` variable. Declare variables for the bean properties: `id`, `journal`, `publisher`, `edition`, `title`, and `author`. Add getter/setter methods for the bean properties. The `@Id` annotation specifies the identifier property. The `@Column` annotation specifies the column name associated with the property. The `nullable` element is set to `false` as the primary key cannot be `null`.

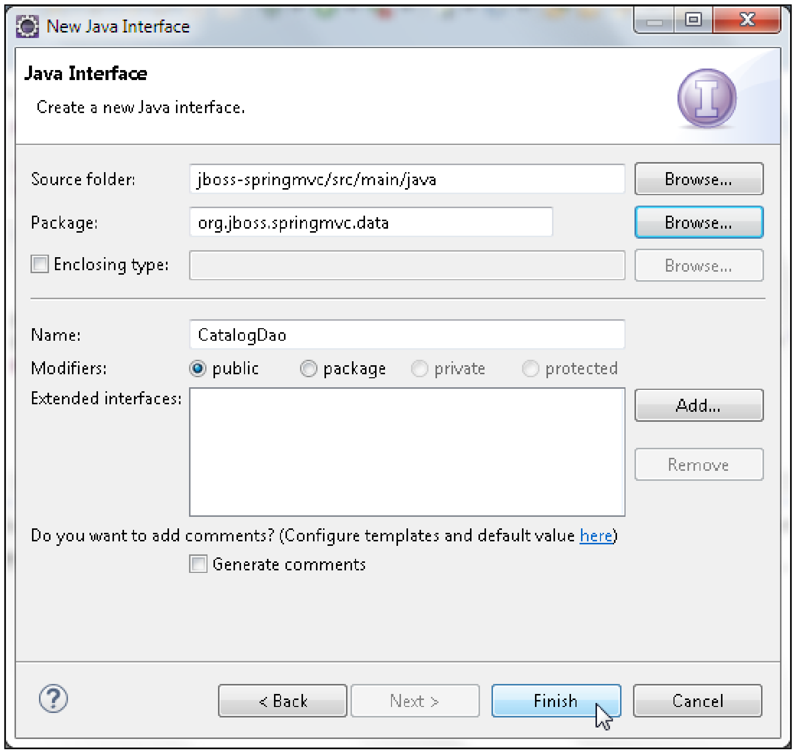
If we were using the Oracle database, we would have specified the primary key generator to be of the type sequence using the `@SequenceGenerator` annotation. The generation strategy is specified with the `@GeneratedValue` annotation. For the Oracle database, the generation strategy would be `strategy=GenerationType.SEQUENCE`, but because MySQL database supports auto-increment of primary key column values by generating a sequence, we have set the generation strategy to `GenerationType.AUTO`. The `Catalog.java` entity class is listed in the following code snippet:

```
package org.jboss.springmvc.model;

import java.io.Serializable;
import javax.persistence.*;

/**
* The persistent class for the CATALOG database table.
*
*/
@Entity
@Table(name = "CATALOG", uniqueConstraints = @
UniqueConstraint(columnNames = "ID"))
public class Catalog implements Serializable {
  private static final long serialVersionUID = 1L;
  private int id;
  private String journal;
  private String publisher;
  private String edition;
  private String title;
  private String author;
  public Catalog() {
  }
  @Id
  @Column(name = "ID", nullable = false)
  @GeneratedValue(strategy = GenerationType.AUTO)
  public int getId() {
    return this.id;
  }
```

```
    public void setId(int id) {
      this.id = id;
    }

    public String getJournal() {
      return this.journal;
    }

    public void setJournal(String journal) {
      this.journal = journal;
    }
    public String getPublisher() {
      return this.publisher;
    }
    public void setPublisher(String publisher) {
      this.publisher = publisher;
    }
    public String getEdition() {
      return this.edition;
    }
    public void setEdition(String edition) {
      this.edition = edition;
    }
    public String getTitle() {
      return this.title;
    }
    public void setTitle(String title) {
      this.title = title;
    }
    public String getAuthor() {
      return this.author;
    }
    public void setAuthor(String author) {
      this.author = author;
    }
  }
```

Creating the Data Access Object design pattern

The advantages of the Data Access Object design pattern, such as reusable software and decoupling of the business logic and database persistence logic, are well established. We shall use the DAO design pattern for the data access layer. The DAO design pattern will provide standard operations for persisting a `Catalog` entity instance and getting `Catalog` entity instances. Create an `org.jboss.springmvc.data.CatalogDao` Java interface, as shown in the following screenshot:



Source folder: jboss-springmvc/src/main/java
Package: org.jboss.springmvc.data
Enclosing type:
Name: CatalogDao
Modifiers: [✓ public] package private protected
Extended interfaces:
Do you want to add comments? (Configure templates and default value here)
Generate comments

Add method definitions for the `persist(Catalog)` methods to persist of the `Catalog` object and `getAllCatalogs()` to get `List` of `Catalog` objects. The `CatalogDao` interface is listed in the following code:

```
package org.jboss.springmvc.data;
import java.util.List;
import javax.persistence.TypedQuery;
import org.jboss.springmvc.model.Catalog;

public interface CatalogDao
{
  public void persist(Catalog catalog);
  public List<Catalog> getAllCatalogs();
}
```

Create an `org.jboss.springmvc.data.CatalogDaoImpl` implementation class to implement the `org.jboss.springmvc.data.CatalogDao` interface. Annotate the class with `@Component`, which makes the class a "component" that is auto-detected using class-path scanning when we use annotation-based configuration. In the `resources/META-INF/spring/applicationContext.xml` application context configuration file, auto-detection of the `org.jboss.springmvc.data` package gets configured with the following `<context:component/>` element within the `<beans/>` element:

```
<context:component-scan base-package="org.jboss.springmvc.data" />
```

Spring MVC autodetects the `CatalogDaoImpl` class and injects a `CatalogDaoImpl` object, `catalogDaoImpl`, into any `@Autowired` field, method parameter, or constructor parameter of the type `CatalogDaoImpl`. Annotation-based transactions are configured using the following declaration in `applicationContext.xml`:

```
<tx:annotation-driven />
```

The `\\jboss-springmvc\src\main\resources\META-INF\spring\applicationContext.xml` file is listed in the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:context="http://www.springframework.org/schema/context"
xmlns:tx="http://www.springframework.org/schema/tx" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.
springframework.org/schema/beans http://www.springframework.org/
schema/beans/spring-beans-3.2.xsd                            http://
www.springframework.org/schema/context http://www.springframework.
org/schema/context/spring-context-3.2.xsd                 http://www.
springframework.org/schema/tx http://www.springframework.org/schema/
tx/spring-tx-3.2.xsd">
  <context:component-scan base-package="org.jboss.springmvc.model" />
  <context:component-scan base-package="org.jboss.springmvc.data" />
  <tx:annotation-driven />
</beans>
```

Annotate the `persist(Catalog)` method with `@Transactional`. Inject `EntityManager` into the DAO implementation class using the `@PersistenceContext` annotation. The `persist()` method of `EntityManager` is used to persist a `Catalog` instance. To retrieve all `Catalog` instances, create a `TypedQuery` object using the `createQuery` method of the `EntityManager` object. Obtain a `List<Catalog>` object as a result using the `getResultList()` method of the `TypedQuery` object. The `org.jboss.springmvc.repo.CatalogDaoImpl` DAO implementation class is listed in the following code:

```
package org.jboss.springmvc.data;
import org.jboss.springmvc.model.*;
import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.TypedQuery;
import org.springframework.stereotype.Component;
import org.springframework.transaction.annotation.Transactional;

@Component
public class CatalogDaoImpl implements CatalogDao
{
  // Injected database connection:
  @PersistenceContext private EntityManager em;
  // Stores a new catalog:
  @Transactional

  public void persist(Catalog catalog) {
    em.persist(catalog);
  }


  // Retrieves all the catalogs:
  public List<Catalog> getAllCatalogs() {
    TypedQuery<Catalog> query = em.createQuery(
    "SELECT c FROM Catalog c ORDER BY c.id", Catalog.class);
    return query.getResultList();

  }
}
```

Creating a web descriptor

We need to add the application context configuration file to the `jboss-springmvc` web application class-path. Add a `<listener/>` tag for the `org.springframework.web.context.ContextLoaderListener` class, which starts up Spring's `WebApplicationContext` root. Specify the `config` location for the root context with the `contextConfigLocation` context parameter. The `org.springframework.web.servlet.DispatcherServlet` servlet dispatches requests to the handler. Specify the `init` parameter, `contextConfigLocation`, to configure another application context for the Spring MVC application in `/WEB-INF/jboss-as-spring-mvc-context.xml`, which is discussed in a later section. Specify a servlet mapping the URL pattern for the `DispatcherServlet` servlet. The `web.xml` file is listed in the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="..."
version="3.1">
  <display-name>Java EE 7 Starter Application</display-name>
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:/META-INF/spring/applicationContext.xml,
                classpath:/META-INF/spring/infrastructure.xml</param-
value>
  </context-param>
  <listener>
    <listener-class>org.springframework.web.context.
ContextLoaderListener</listener-class>
  </listener>
  <servlet>
    <servlet-name>jboss-spring</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</
servlet-class>
    <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>/WEB-INF/jboss-as-spring-mvc-context.xml</param-
value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>jboss-spring</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>
</web-app>
```

Creating a request handler

A typical request handler in Spring MVC is based on the `@Controller` and `@RequestMapping` annotations. Create a Java class, `org.jboss.springmvc.controller.CatalogController`, annotated with the `@Controller` annotation, which makes the class auto-detectable for injection by Spring. In the `\springmvc\src\main\webapp\WEB-INF\jboss-as-spring-mvc-context.xml` context file, which is listed and discussed later in the chapter, the component scanning for the `org.jboss.springmvc.mvc` base package is configured, as follows:

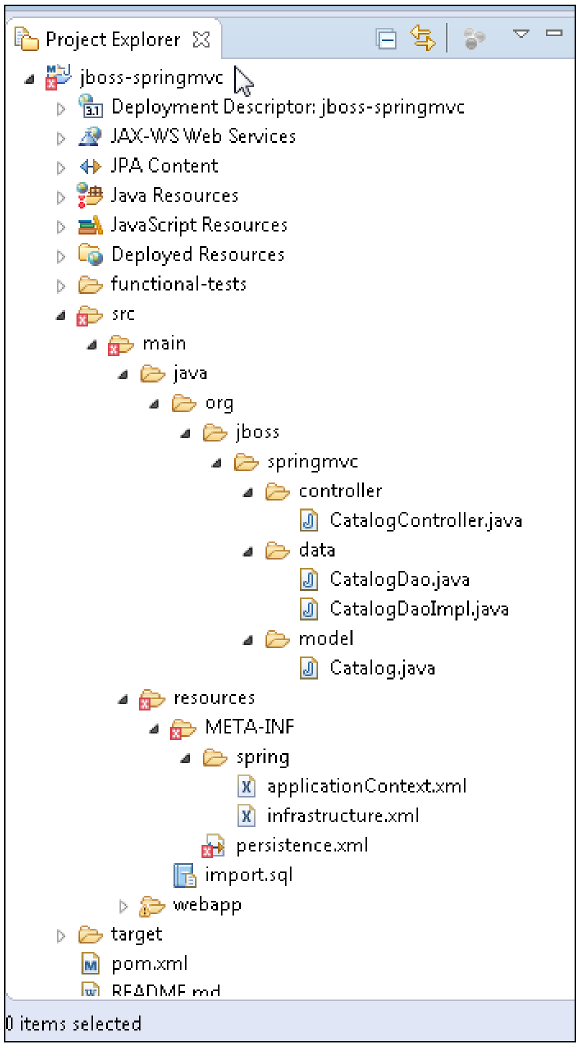
```
<context:component-scan base-package="org.jboss.springmvc.controller"
/>
```

`BeanFactory` injects a `catalogController` bean when the `CatalogController` class is required to handle a request. The handler class creates a model `Map` with data and selects a view name to be rendered. The `createAndDisplayCatalog` method takes the servlet request sent by `DispatcherServlet` and returns an `org.springframework.web.servlet.ModelAndView` object, which is a holder for `Model` and `View` in the Spring MVC framework. Map the servlet request onto the `createAndDisplayCatalog` handler method using the `@RequestMapping` annotation. Specify the path mapping URI as `/catalog`.

Autowire the `org.jboss.springmvc.data.CatalogDao` DAO interface to Spring's dependency injection mechanism using the `@AutoWire` annotation. A `catalogDaoImpl` bean is injected on startup as, by default, the application context pre-instantiates all singleton (the default scope in Spring) beans at startup. The `createAndDisplayCatalog` method gets the request parameter values and creates a `Catalog` object. The `Catalog` object is stored using the autowired DAO object's persist method. Having created a model `Map`, the `createAndDisplayCatalog` method returns a `org.springframework.web.servlet.ModelAndView` object with the `View` catalog and the `catalogDao` model. The `org.springframework.web.servlet.ModelAndView` object is returned to `DispatcherServlet` for the servlet to render the response. The view ID is resolved using the view resolver specified in the `jboss-as-spring-mvc-context.xml` context file. The `Map` model is made available to the view template. The controller class is listed in the following code:

```
package org.jboss.springmvc.controller;

import org.jboss.springmvc.data.*;
import org.jboss.springmvc.model.*;
import javax.servlet.http.HttpServletRequest;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;

@Controller
public class CatalogController {
  @Autowired
  private CatalogDao catalogDao;

  @RequestMapping(value = "/catalog")
  public ModelAndView createAndDisplayCatalog(HttpServletRequest
request) {
    // Handle a new catalog (if any):
    String journal = request.getParameter("journal");
    String publisher = request.getParameter("publisher");
    String edition = request.getParameter("edition");
    String title = request.getParameter("title");
    String author = request.getParameter("author");
    if (journal != null && publisher != null && edition != null
    && title != null && author != null) {
      Catalog catalog = new Catalog();
      catalog.setJournal(journal);
      catalog.setPublisher(publisher);
      catalog.setEdition(edition);
      catalog.setTitle(title);
      catalog.setAuthor(author);
      catalogDao.persist(catalog);
    }
    // Prepare the result view (catalog):
    return new ModelAndView("catalog", "catalogDao", catalogDao);
  }
}
```

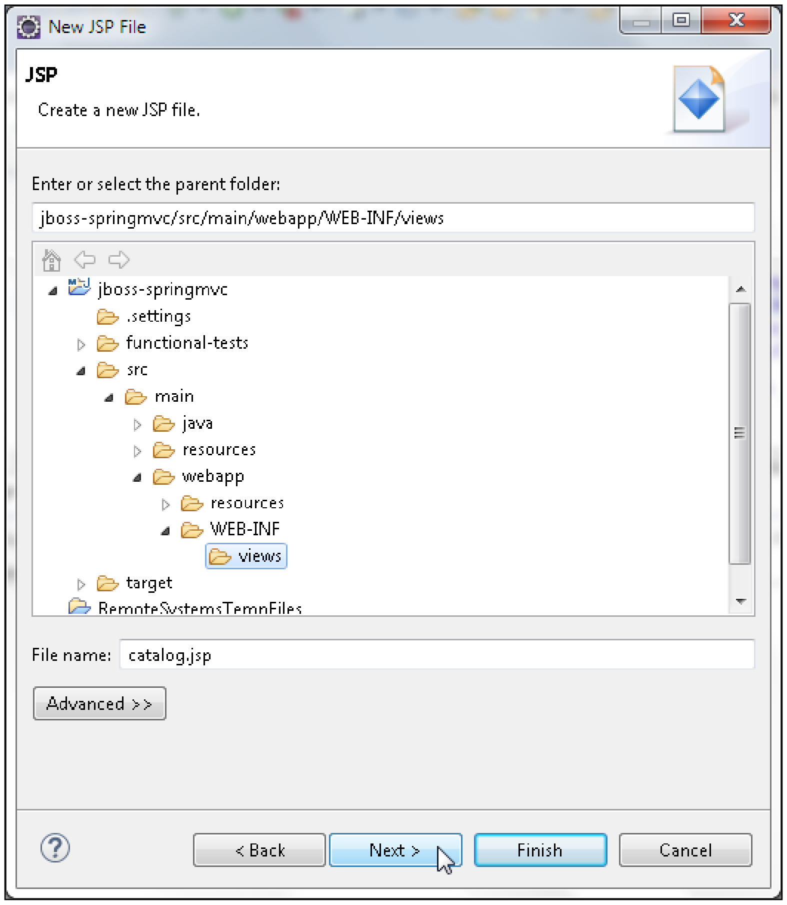
The model, DAO, and controller classes are shown in **Project Explorer** in the following screenshot:



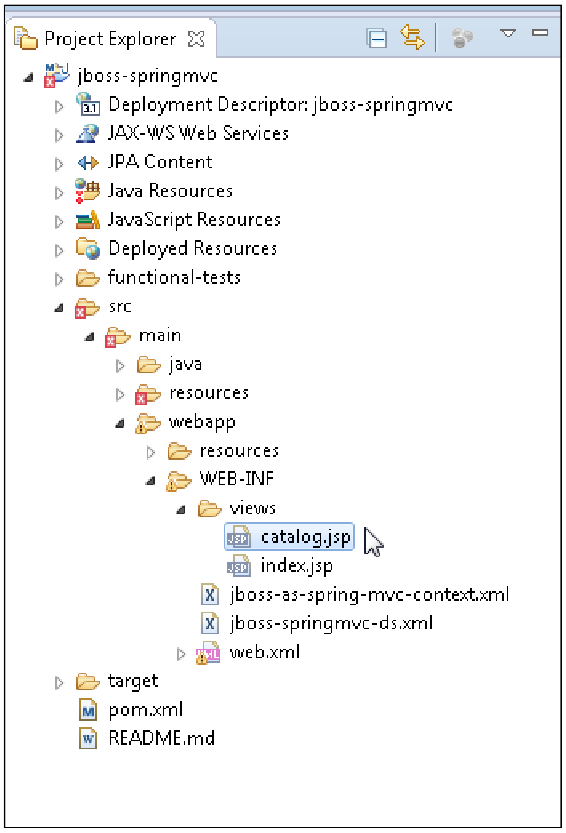
```
Project Explorer
jboss-springmvc
  Deployment Descriptor: jboss-springmvc
  JAX-WS Web Services
  JPA Content
  Java Resources
  JavaScript Resources
  Deployed Resources
  functional-tests
  src
    main
      java
        org
          jboss
            springmvc
              controller
                CatalogController.java
              data
                CatalogDao.java
                CatalogDaoImpl.java
              model
                Catalog.java
      resources
        META-INF
          spring
            applicationContext.xml
            infrastructure.xml
          persistence.xml
        import.sql
      webapp
  target
  pom.xml
  README.md
0 items selected
```

Creating the View

The view resolver, configured in the `WEB-INF/jboss-as-spring-mvc-context.xml` web application context, resolves `View` from the `ModelAndView` object, and the response is rendered with a `catalog.jsp` view template. Create `catalog.jsp` in the `WEB-INF/views` directory. To create a JSP, select **File** | **New** | **Other**, and in **New**, select **Web** | **JSP File** and click on **Next**. In the **New JSP File** wizard, select the `WEB-INF/views` folder, specify **File name** as `catalog.jsp`, and click on **Next**, as shown in the following screenshot. Subsequently click on **Finish**.



The `catalog.jsp` class and the other classes added, which include the controller class, the DAO classes, and the model class, are shown in **Project Explorer** in the following screenshot:



The request initiates from the `catalog.jsp` view template, and the response is rendered with `catalog.jsp`. To send a request to `DispatcherServlet`, the `<form/>` element has the `action` attribute set to `catalog`, which gets mapped to the `/catalog` path URI specified using the `@RequestMapping` annotation in the `CatalogController` request handler class. The `createAndDisplayCatalog` handler method gets invoked when the form is posted. In `catalog.jsp`, instantiate a bean from the `org.jboss.springmvc.data.CatalogDao` interface with the scope request. Add `<form/>` with `<input/>` elements for input fields to create a new `Catalog` object. To render the response from a `org.jboss.springmvc.data.CatalogDao` bean, invoke the `getAllCatalogs()` method to obtain `List` of `Catalog` objects. Iterate over `List` to output `Catalog` instance properties.

The `catalog.jsp` is listed in the following code:

```
<%@page contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"%>
<%@page import="org.jboss.springmvc.model.Catalog,org.jboss.springmvc.
data.CatalogDao,org.jboss.springmvc.controller.CatalogController"%>
<jsp:useBean id="catalogDao" type="org.jboss.springmvc.data.
CatalogDao" scope="request" />
<!--scope should be request for the bean to be found  -->
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>Spring MVC-JPA Catalog</title>
<head>
    <meta name="generator" content="HTML Tidy for Linux/x86 (vers 25
March 2009), see www.w3.org" />
    <title>
      Spring MVC-JPA Catalog
    </title>
  </head><!-- action should be same as the mapping in the controller
-->
  <body>
    <h2>
      Catalog form
    </h2>
    <form method="post" action="catalog">
      Journal: <input type="text" name="journal" />Publisher:
<input type="text" name="publisher" />Edition: <input type="text"
name="edition" />Title: <input type="text" name="title" />Author:
<input type="text" name="author" />
      <table>
        <tr>
          <td></td>
        </tr>
        <tr>
          <td></td>
        </tr>
        <tr>
          <td></td>
        </tr>
        <tr>
          <td></td>
        </tr>
```

```
          <tr>
            <td></td>
          </tr>
        </table>
        <p>
          <input type="submit" value="Add" />
        </p>
      </form>
      <table>
        <%
          for (Catalog catalog : catalogDao.getAllCatalogs()) {
        %>
        <tr>
          <td>
            <%=catalog.getId()%>
          </td>
          <td>
            <%=catalog.getJournal()%>
          </td>
          <td>
            <%=catalog.getPublisher()%>
          </td>
          <td>
            <%=catalog.getEdition()%>
          </td>
          <td>
            <%=catalog.getTitle()%>
          </td>
          <td>
            <%=catalog.getAuthor()%>
          </td>
        </tr><%
          }
        %>
      </table>
    </body>
  </html>
```

Creating a Spring MVC context

The Spring MVC web application context is configured in `springmvc\src\main\webapp\WEB-INF\ jboss-as-spring-mvc-context.xml`. In addition to adding the `org.jboss.springmvc.controller` package for auto-detection, support for processing requests using the controller methods annotated with `@RequestMapping` is configured with the `<mvc:annotation-driven />` declaration. Add support for the annotation-based transaction manager; the `CatalogDao` interface's `persist` method is annotated with `@Transactional`. Add a view resolver bean for the `org.springframework.web.servlet.view.InternalResourceViewResolver` class. It is a best practice to add the view templates, such as `catalog.jsp`, within the `WEB-INF` directory so that only the controller has access to them. Add a bean property prefix with the `/WEB-INF/views` value to resolve the view templates. The `jboss-as-spring-mvc-context.xml` file is listed in the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/
beans" xmlns:context="http://www.springframework.org/schema/
context" xmlns:mvc="http://www.springframework.org/schema/mvc"
xmlns:tx="http://www.springframework.org/schema/tx" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.
springframework.org/schema/beans http://www.springframework.org/
schema/beans/spring-beans-3.2.xsd                            http://
www.springframework.org/schema/context http://www.
springframework.org/schema/context/spring-context-3.2.xsd
http://www.springframework.org/schema/mvc http://www.springframework.
org/schema/mvc/spring-mvc-3.2.xsd                             http://
www.springframework.org/schema/tx http://www.springframework.org/
schema/tx/spring-tx.xsd">
  <context:component-scan base-package="org.jboss.springmvc.
controller" />
  <mvc:annotation-driven />
  <!-- Add JPA support -->
  <bean id="emf" class="org.springframework.orm.jpa.
LocalContainerEntityManagerFactoryBean">
    <property name="loadTimeWeaver">
      <bean class="org.springframework.instrument.classloading.
InstrumentationLoadTimeWeaver" />
    </property>
  </bean>
  <!-- Add Transaction support -->
  <bean id="txManager" class="org.springframework.orm.jpa.
JpaTransactionManager">
    <property name="entityManagerFactory" ref="emf" />
  </bean>
  <!-- Use @Transaction annotations for managing transactions -->
  <tx:annotation-driven transaction-manager="txManager" />
  <bean id="viewResolver" class="org.springframework.web.servlet.view.
InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/views/" />
    <property name="suffix" value=".jsp" />
  </bean>
  <mvc:resources mapping="/static/**" location="/" />
  <mvc:default-servlet-handler />
</beans>
```

The Spring MVC provides two levels of application contexts, the root application context for application-level services and a servlet application context for the servlet-level beans and services. The `/META-INF/spring/applicationContext.xml` file is configured in `web.xml` is the root-level application context, and `/WEB-INF/jboss-as-spring-mvc-context.xml` is the context for `DispatcherServlet`. The servlet context configuration overrides the root context for beans with the same name.

Deploying the Spring project with Maven

Next, compile, package, and deploy the Spring MVC application using the Maven build tool. The Hibernate JPA API, the Hibernate Validator API, and the Hibernate Entity manager API are provided by WildFly 8.1. Add a dependency on the MySQL JDBC connector, as follows:

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>5.1.22</version>
</dependency>
```

The Spring dependencies are available in the `org.springframework` group ID. The Spring version is configured as Spring 4.1.2 by default in `pom.xml`, as follows:

```
<version.spring>4.1.2.RELEASE</version.spring>
```

The Spring Web MVC dependency is also configured as version 4.1.2. All the required Spring framework dependencies—Spring AOP, Spring Beans, Spring Context, Spring Core, Spring JDBC, Spring ORM, Spring Tx, Log4j, AOP alliance, Commons logging, Cglib, Spring Expression Language, and Spring Web—are configured in `pom.xml`. In `<build/>`, configure the Maven compiler plugin and the Maven WAR plugin. Set the output directory for the Maven War plugin as the `C:\wildfly-8.1.0.Final\standalone\deployments` directory of the WildFly 8.1 standalone installation.

The `pom.xml` file is listed in the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.
apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.jboss.springmvc</groupId>
  <artifactId>jboss-springmvc</artifactId>
  <packaging>war</packaging>
  <version>1.0.0</version>
  <name>Getting Started with Spring on JBoss</name>
  <properties>
    <!-- Spring version -->
    <version.spring>4.1.2.RELEASE</version.spring>
    <!-- Spring Third Party dependencies -->
    <version.aopalliance>1.0</version.aopalliance>
    <!-- Third Party dependencies -->
    <version.standard.taglibs>1.1.2</version.standard.taglibs>
    <version.commons.logging>1.1.1</version.commons.logging>
    <!-- JBoss AS plugin for deployment -->
    <version.jboss.as.maven.plugin>7.6.Final</version.jboss.as.maven.
plugin>
  </properties>
  <repositories>
    <repository>
      <id>springsource-milestones</id>
      <name>SpringSource Milestones Proxy</name>
      <url>https://oss.sonatype.org/content/repositories/springsource-
milestones</url>
    </repository>
    <repository>
      <id>Maven</id>
      <name>Maven</name>
      <url>http://repo1.maven.org/maven2</url>
    </repository>
  </repositories>
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.jboss.spec</groupId>
        <artifactId>jboss-javaee-web-6.0</artifactId>
        <version>3.0.0.Final</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
      <!-- Spring dependencies -->
      <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-aop</artifactId>
        <version>${version.spring}</version>
      </dependency>
      <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-beans</artifactId>
        <version>${version.spring}</version>
      </dependency>
      <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>${version.spring}</version>
      </dependency>
      <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context-support</artifactId>
        <version>${version.spring}</version>
      </dependency>
      <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>${version.spring}</version>
      </dependency>
      <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-expression</artifactId>
        <version>${version.spring}</version>
      </dependency>
      <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-jdbc</artifactId>
        <version>${version.spring}</version>
      </dependency>
      <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-orm</artifactId>
```

```
          <version>${version.spring}</version>
        </dependency>
        <dependency>
          <groupId>org.springframework</groupId>
          <artifactId>spring-test</artifactId>
          <version>${version.spring}</version>
          <scope>test</scope>
        </dependency>
        <dependency>
          <groupId>org.springframework</groupId>
          <artifactId>spring-tx</artifactId>
          <version>${version.spring}</version>
        </dependency>
        <dependency>
          <groupId>org.springframework</groupId>
          <artifactId>spring-web</artifactId>
          <version>${version.spring}</version>
        </dependency>
        <dependency>
          <groupId>org.springframework</groupId>
          <artifactId>spring-webmvc</artifactId>
          <version>${version.spring}</version>
        </dependency>
        <!-- Third Party dependencies -->
        <dependency>
          <groupId>aopalliance</groupId>
          <artifactId>aopalliance</artifactId>
          <version>${version.aopalliance}</version>
        </dependency>
        <dependency>
          <groupId>taglibs</groupId>
          <artifactId>standard</artifactId>
          <version>${version.standard.taglibs}</version>
        </dependency>
        <dependency>
          <groupId>commons-logging</groupId>
          <artifactId>commons-logging</artifactId>
          <version>${version.commons.logging}</version>
        </dependency>
      </dependencies>
    </dependencyManagement>
    <dependencies>
    <!-- Import the JPA API using the provided scope It is included in
JBoss AS 7 / EAP 6 -->
    <dependency>
      <groupId>org.hibernate.javax.persistence</groupId>
      <artifactId>hibernate-jpa-2.0-api</artifactId>
      <scope>provided</scope>
    </dependency>
    <!-- JSR-303 (Bean Validation) Implementation -->
    <!-- Provides portable constraints such as @Email -->
    <!-- Hibernate Validator is shipped in JBoss AS 7 -->
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-validator</artifactId>
      <version>4.1.0.Final</version>
      <exclusions>
        <exclusion>
          <groupId>org.slf4j</groupId>
          <artifactId>slf4j-api</artifactId>
        </exclusion>
      </exclusions>
    </dependency>
    <!-- Annotation processor that raising compilation errors whenever
constraint annotations are incorrectly used. -->
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-validator-annotation-processor</
artifactId>
      <version>4.1.0.Final</version>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-entitymanager</artifactId>
      <version>4.1.0.Final</version>
    </dependency>
    <!-- Import Spring dependencies, these are either from community
or versions      certified in WFK2 -->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-aop</artifactId>
      <version>${version.spring}</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
    <artifactId>spring-beans</artifactId>
    <version>${version.spring}</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>${version.spring}</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context-support</artifactId>
    <version>${version.spring}</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
    <version>${version.spring}</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-expression</artifactId>
    <version>${version.spring}</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>${version.spring}</version>
    <!-- <scope>provided</scope> -->
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-orm</artifactId>
    <version>${version.spring}</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-tx</artifactId>
    <version>${version.spring}</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-web</artifactId>
```

```
      <version>${version.spring}</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-webmvc</artifactId>
      <version>${version.spring}</version>
    </dependency>
    <!-- Other community dependencies -->
    <dependency>
      <groupId>aopalliance</groupId>
      <artifactId>aopalliance</artifactId>
      <version>${version.aopalliance}</version>
    </dependency>
    <dependency>
      <groupId>taglibs</groupId>
      <artifactId>standard</artifactId>
      <version>${version.standard.taglibs}</version>
    </dependency>
    <dependency>
      <groupId>commons-logging</groupId>
      <artifactId>commons-logging</artifactId>
      <version>${version.commons.logging}</version>
    </dependency>
    <!-- Add cglib for the CatalogDaoTest -->
    <dependency>
      <groupId>cglib</groupId>
      <artifactId>cglib-nodep</artifactId>
      <version>2.2</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>5.1.22</version>
    </dependency>
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>servlet-api</artifactId>
      <version>2.5</version>
    </dependency>
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>javax.servlet-api</artifactId>
      <version>3.0.1</version>
    </dependency>
    <!-- Add JSON dependency, specified in jboss-deployment-structure.
xml -->
    <dependency>
      <groupId>org.codehaus.jackson</groupId>
      <artifactId>jackson-mapper-asl</artifactId>
      <version>1.9.3</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>org.codehaus.jackson</groupId>
      <artifactId>jackson-core-asl</artifactId>
      <version>1.9.3</version>
      <!-- <scope>provided</scope> -->
    </dependency>
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-simple</artifactId>
      <version>1.6.4</version>
      <!-- <scope>provided</scope> -->
    </dependency>
  </dependencies>
  <build>
    <finalName>jboss-springmvc</finalName>
    <plugins>
      <!-- Force Java 6 -->
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>2.4</version>
        <configuration>
          <source>1.7</source>
          <target>1.7</target>
        </configuration>
      </plugin>
      <!-- Deployent on AS from console -->
      <plugin>
        <groupId>org.jboss.as.plugins</groupId>
        <artifactId>jboss-as-maven-plugin</artifactId>
        <version>${version.jboss.as.maven.plugin}</version>
      </plugin>
      <plugin>
        <artifactId>maven-war-plugin</artifactId>
        <version>2.2</version>
        <configuration>
          <outputDirectory>C:\wildfly-8.1.0.Final\standalone\
deployments</outputDirectory>
          <failOnMissingWebXml>false</failOnMissingWebXml>
          <warName>${project.artifactId}</warName>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

Some of the dependencies listed do not get used in the example application but could be used in some other Spring MVC application. Before we run the `jboss-springmvc` application, remove any test files and classes from the project. Also, remove `import.sql` from the `resources` directory.

Creating deployment structure and infrastructure deployment descriptors

We have not yet discussed two files: `webapp/WEB-INF/jboss-deployment-structure.xml` and `resources/META-INF/spring/infrastructure.xml`. The `jboss-deployment-structure.xml` file is a JBoss-specific deployment descriptor used to configure fine-grained class-loading. If the `jboss-deployment-structure.xml` file is not included, the default class-loading is used, which might not be what is required for an application. The `jboss-deployment-structure.xml` file can be used for the following purposes:

- Preventing the inclusion of automatic dependencies
- Adding additional dependencies
- Defining additional modules
- Adding additional resource roots to a module
- Modifying an `EAR` deployments isolated class loading behavior

We have used `jboss-deployment-structure.xml` to include dependencies on the `org.codehaus.jackson.jackson-core-asl`, `org.codehaus.jackson.jackson-mapper-asl`, `org.slf4j`, and `mysql` modules.

The `jboss-deployment-structure.xml` file is listed as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss-deployment-structure xmlns="urn:jboss:deployment-
structure:1.0">
  <deployment>
    <!-- <exclusions>
      <module name="org.hibernate" />
    </exclusions> -->
    <dependencies>
      <module name="mysql" />
      <module name="org.codehaus.jackson.jackson-core-asl" />
      <module name="org.codehaus.jackson.jackson-mapper-asl" />
      <module name="org.slf4j" />
    </dependencies>
  </deployment>
</jboss-deployment-structure>
```

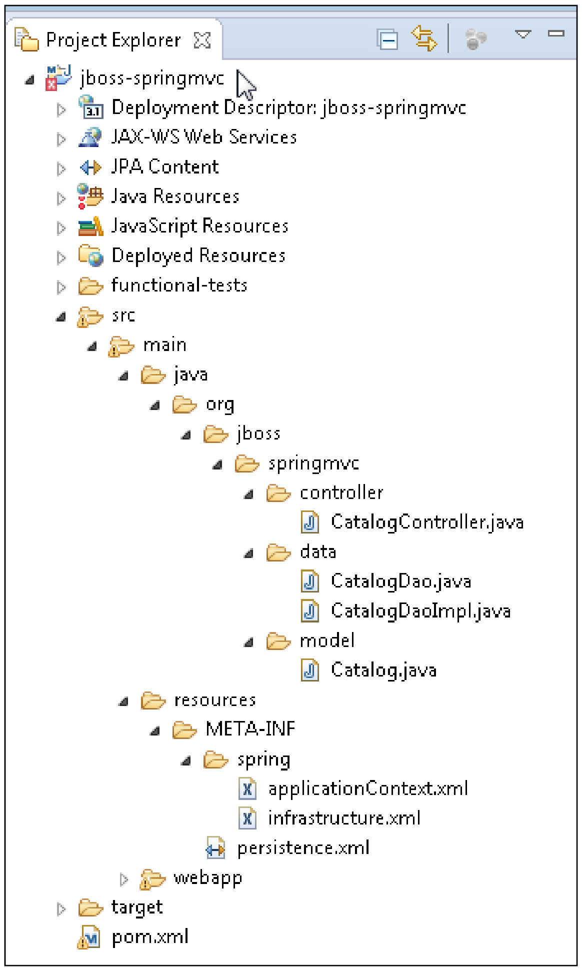
The `infrastructure.xml` file is used for the following purposes:

- Registering the entity manager factory in JNDI
- Registering the data source in JNDI
- Enabling JTA transaction management

The `infrastructure.xml` file is listed as follows:

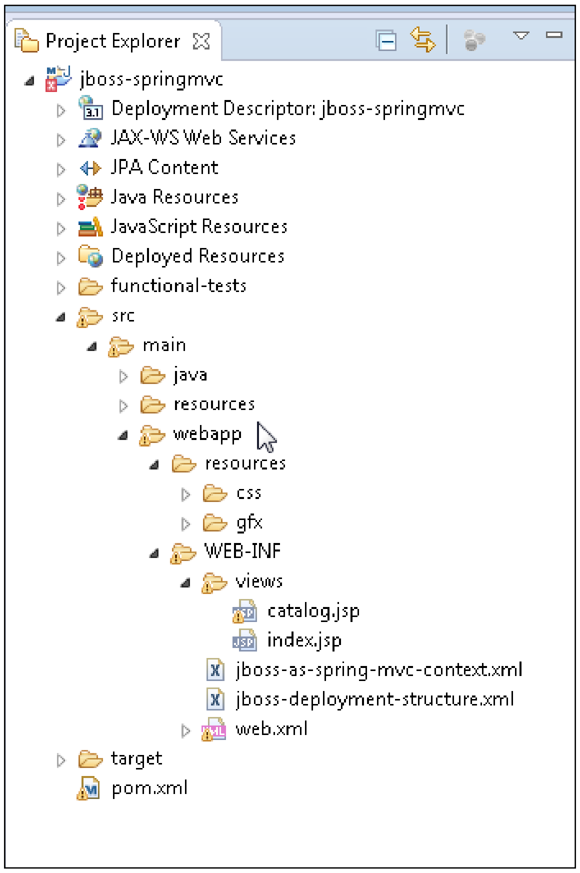
```
<?xml version="1.0" encoding="UTF-8"?>
<jboss-deployment-structure xmlns="urn:jboss:deployment-
structure:1.0">
  <deployment>
    <!-- <exclusions>
      <module name="org.hibernate" />
    </exclusions> -->
    <dependencies>
      <module name="mysql" />
      <module name="org.codehaus.jackson.jackson-core-asl" />
      <module name="org.codehaus.jackson.jackson-mapper-asl" />
      <module name="org.slf4j" />
    </dependencies>
  </deployment>
</jboss-deployment-structure>
```

The directory structure of the Spring Maven application's `jboss-springmvc\src\main\java` and `jboss-springmvc\src\main\resources` directories is shown in **Project Explorer** in the following screenshot:



```
jboss-springmvc
  Deployment Descriptor: jboss-springmvc
  JAX-WS Web Services
  JPA Content
  Java Resources
  JavaScript Resources
  Deployed Resources
  functional-tests
  src
    main
      java
        org
          jboss
            springmvc
              controller
                CatalogController.java
              data
                CatalogDao.java
                CatalogDaoImpl.java
              model
                Catalog.java
      resources
        META-INF
          spring
            applicationContext.xml
            infrastructure.xml
          persistence.xml
      webapp
  target
  pom.xml
```

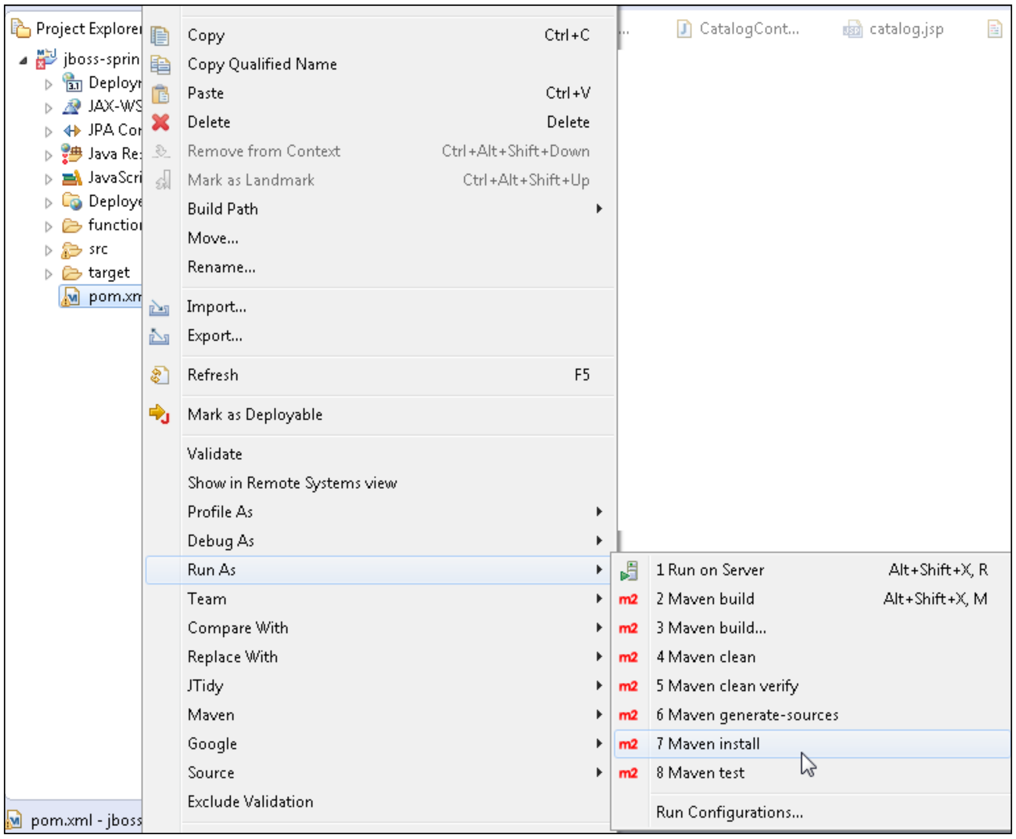
The directory structure of the Spring Maven application's `jboss-springmvc\src\main\webapp` directory is also shown in **Project Explorer** in the following screenshot:



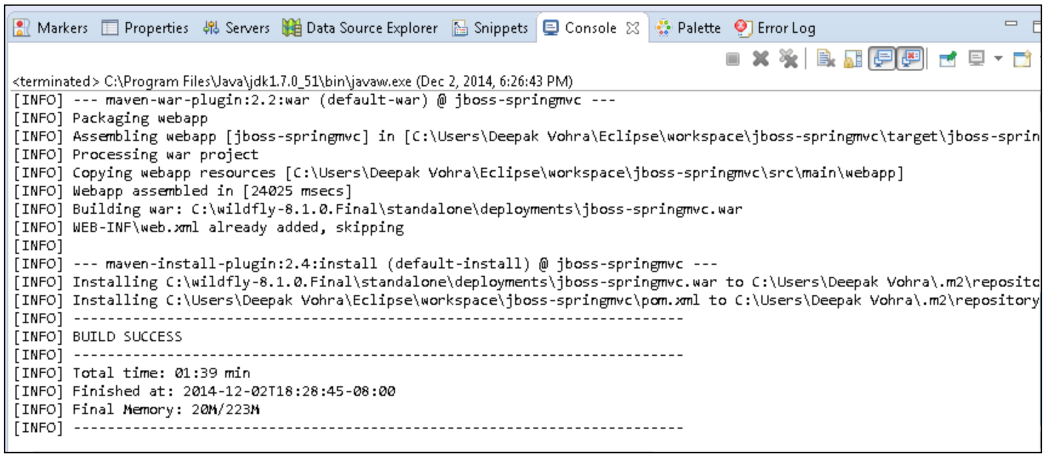
```
jboss-springmvc
  Deployment Descriptor: jboss-springmvc
  JAX-WS Web Services
  JPA Content
  Java Resources
  JavaScript Resources
  Deployed Resources
  functional-tests
  src
    main
      java
      resources
      webapp
        resources
          css
          gfx
        WEB-INF
          views
            catalog.jsp
            index.jsp
          jboss-as-spring-mvc-context.xml
          jboss-deployment-structure.xml
          web.xml
  target
  pom.xml
```

Installing the Spring MVC Maven project

To run the Maven build tool, right-click on `pom.xml` and select **Run As** | **Maven install**, as shown in the following screenshot:

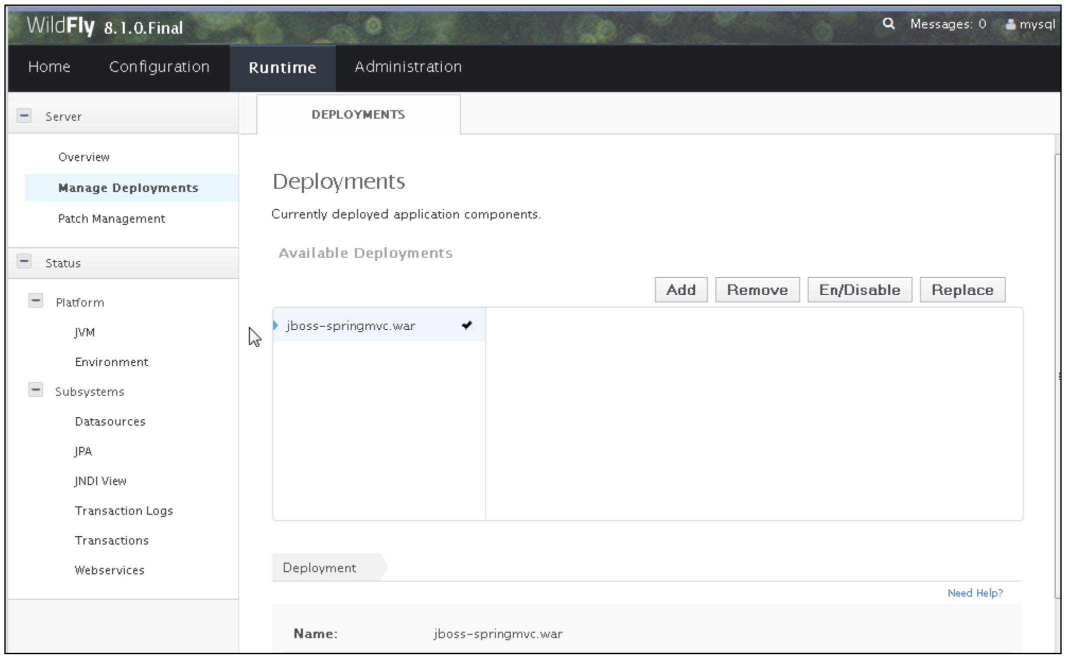


The `jboss-springmvc` application gets compiled and packaged to the `jboss-springmvc.war` archive, which gets output to the `deployments` directory. Maven outputs a **BUILD SUCCESS** message, as shown in the following screenshot:



```
<terminated> C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Dec 2, 2014, 6:26:43 PM)
[INFO] --- maven-war-plugin:2.2:war (default-war) @ jboss-springmvc ---
[INFO] Packaging webapp
[INFO] Assembling webapp [jboss-springmvc] in [C:\Users\Deepak Vohra\Eclipse\workspace\jboss-springmvc\target\jboss-sprin
[INFO] Processing war project
[INFO] Copying webapp resources [C:\Users\Deepak Vohra\Eclipse\workspace\jboss-springmvc\src\main\webapp]
[INFO] Webapp assembled in [24025 msecs]
[INFO] Building war: C:\wildfly-8.1.0.Final\standalone\deployments\jboss-springmvc.war
[INFO] WEB-INF\web.xml already added, skipping
[INFO]
[INFO] --- maven-install-plugin:2.4:install (default-install) @ jboss-springmvc ---
[INFO] Installing C:\wildfly-8.1.0.Final\standalone\deployments\jboss-springmvc.war to C:\Users\Deepak Vohra\.m2\reposito
[INFO] Installing C:\Users\Deepak Vohra\Eclipse\workspace\jboss-springmvc\pom.xml to C:\Users\Deepak Vohra\.m2\repository
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 01:39 min
[INFO] Finished at: 2014-12-02T18:28:45-08:00
[INFO] Final Memory: 20M/223M
[INFO] ------------------------------------------------------------------------
```

Start the WildFly 8.1 server if it is not already started. The `jboss-springmvc.war` archive gets deployed to WildFly 8.1. Now log in to the WildFly **Administration Console** and click on **Manage Deployments**. The `jboss-springmvc.war` file, as deployed, is shown in the following screenshot:



Deployments

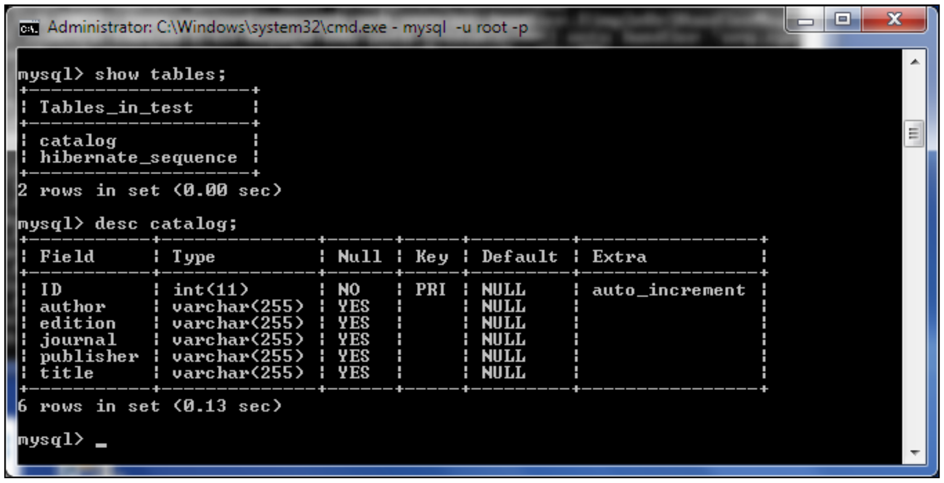
Currently deployed application components.

Available Deployments

jboss-springmvc.war ✓

Name: jboss-springmvc.war

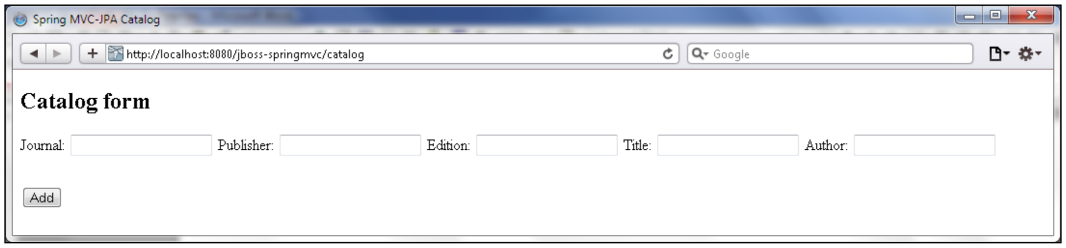
The **catalog** schema gets exported to the MySQL database. A `hibernate_sequence` to auto-increment the `ID` field also gets created. The `catalog` schema description can be output with the `desc catalog` command in the MySQL command-line client, as shown in the following screenshot:


Administrator: C:\Windows\system32\cmd.exe - mysql -u root -p

```
mysql> show tables;
+--------------------+
| Tables_in_test     |
+--------------------+
| catalog            |
| hibernate_sequence |
+--------------------+
2 rows in set (0.00 sec)

mysql> desc catalog;
+-----------+--------------+------+-----+---------+----------------+
| Field     | Type         | Null | Key | Default | Extra          |
+-----------+--------------+------+-----+---------+----------------+
| ID        | int(11)      | NO   | PRI | NULL    | auto_increment |
| author    | varchar(255) | YES  |     | NULL    |                |
| edition   | varchar(255) | YES  |     | NULL    |                |
| journal   | varchar(255) | YES  |     | NULL    |                |
| publisher | varchar(255) | YES  |     | NULL    |                |
| title     | varchar(255) | YES  |     | NULL    |                |
+-----------+--------------+------+-----+---------+----------------+
6 rows in set (0.13 sec)

mysql> _
```

Running the Spring MVC application

Run the Spring MVC application in a browser using the URL `http://localhost:8080/jboss-springmvc/catalog`. The input view template `catalog.jsp` gets displayed, as shown in the browser in the following screenshot:

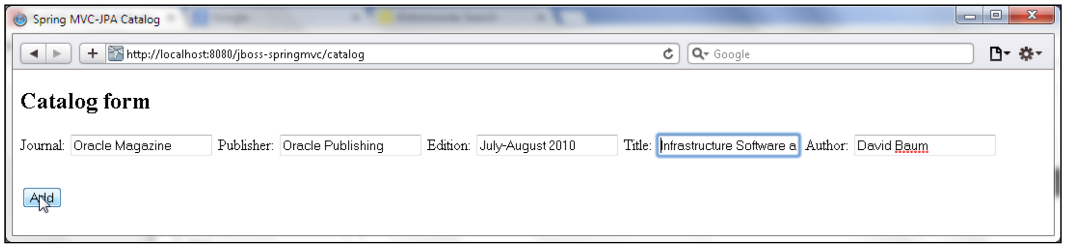


Catalog form

Journal: Publisher: Edition: Title: Author:

Add

Specify the values for a catalog entry in the input form and click on **Add**, as shown in the following screenshot:

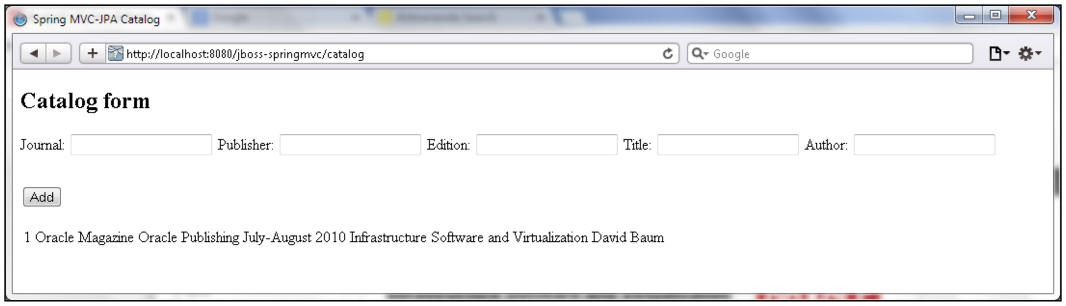


Catalog form

Journal:	Oracle Magazine	Publisher:	Oracle Publishing	Edition:	July-August 2010	Title:	Infrastructure Software a	Author:	David Baum

Add

A catalog entry gets created and persisted to the MySQL database table `catalog` and gets displayed in a table, as shown in the following screenshot:



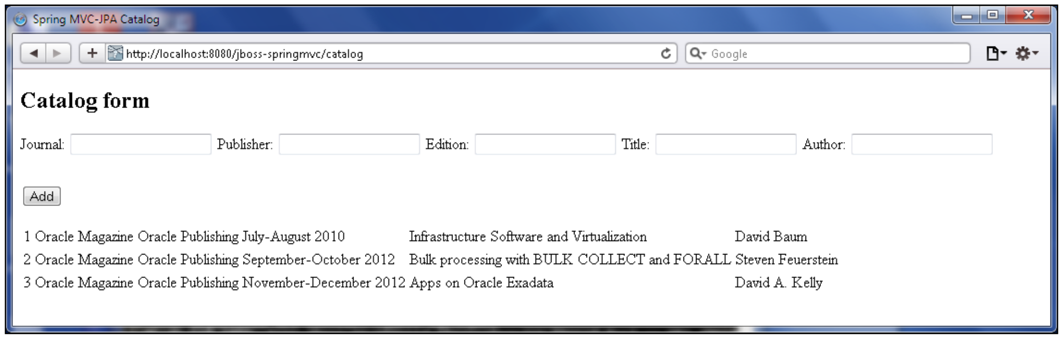
Catalog form

Journal: Publisher: Edition: Title: Author:

Add

1 Oracle Magazine Oracle Publishing July-August 2010 Infrastructure Software and Virtualization David Baum

Similarly, create other catalog entries, as shown in the table in the following screenshot:



Catalog form

Journal: Publisher: Edition: Title: Author:

Add

1	Oracle Magazine	Oracle Publishing	July-August 2010	Infrastructure Software and Virtualization	David Baum
2	Oracle Magazine	Oracle Publishing	September-October 2012	Bulk processing with BULK COLLECT and FORALL	Steven Feuerstein
3	Oracle Magazine	Oracle Publishing	November-December 2012	Apps on Oracle Exadata	David A. Kelly

Summary

In this chapter, we developed a Spring 4.1 MVC application in Eclipse to create a catalog. We compiled and packaged the application with Maven and deployed the application to WildFly 8.1. Subsequently, we tested the Spring MVC application to create a catalog.

In the next chapter, we will discuss a new feature in Java EE 7 and JAX-RS 2.0.

9
Using JAX-RS 2.0 in Java EE 7 with RESTEasy

JSR 311 (`http://jcp.org/en/jsr/detail?id=311`) specifies the Java API for RESTful Web services (JAX-RS) for developing **REST** (**Representational State Transfer**) Web services with Java. REST is a protocol independent, loosely coupled, software architecture style for distributed systems. A RESTful Web service exposes a set of resources, which are simply sources of information, identified by **URI**s (**Uniform Resource Identifiers**) in HTTP. RESTful Web services follow these RESTful principles:

- Every resource has a unique base URI
- For invoking Web service operations, the HTTP protocol methods such as `GET`, `PUT`, `POST`, and `DELETE` are used
- A client sends a request to a service, and the service returns a representation of a resource requested to the client
- Client sessions are not stored on the server, which makes it easier to scale the service with less data to replicate in a clustered environment

JSR 339 (`https://www.jcp.org/en/jsr/detail?id=339`) develops the JAX-RS 2.0 version. JAX-RS 2.0 provides several new features, such as a Client API, support for **validation**, **filters** and **interceptors**, and **asynchronous** processing. We will discuss the salient new features in JAX RS 2.0 using the RESTEasy (`http://resteasy.jboss.org/`) implementation. This chapter has the following sections:

- Setting up the environment
- The Client API
- Filters and interceptors
- Asynchronous processing

- Cancelling a request
- Session bean EJB resource
- Making an asynchronous call from the client

Setting up the environment

We need to install the following software:

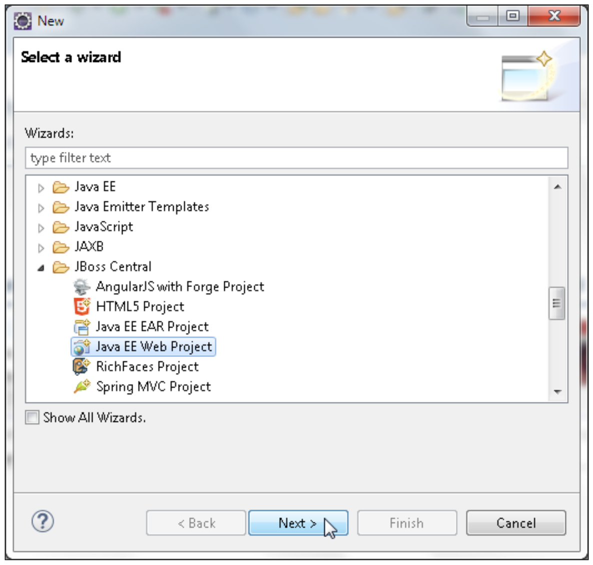
- **WildFly 8.1.0.Final**: Download `wildfly-8.1.0.Final.zip` from `http://wildfly.org/downloads/`.
- **Eclipse IDE for Java EE Developers**: Download Eclipse Luna from `https://www.eclipse.org/downloads/packages/release/Luna/SR1`.
- **JBoss Tools (Luna) 4.2.0.Final (or the latest version)**: Install this as a plugin to Eclipse from Eclipse Marketplace (`http://tools.jboss.org/downloads/installation.html`).
- **Apache Maven**: Download version 3.05 or higher from `http://maven.apache.org/download.cgi`.
- **Java 7**: Download Java 7 from `http://www.oracle.com/technetwork/java/javase/downloads/index.html?ssSourceSiteId=ocomcn`.)

Set the environment variables `JAVA_HOME`, `JBOSS_HOME`, and `MAVEN_HOME`. Add `%JAVA_HOME%/bin`, `%MAVEN_HOME%/bin`, and `%JBOSS_HOME%/bin` to the `PATH` environment variable.

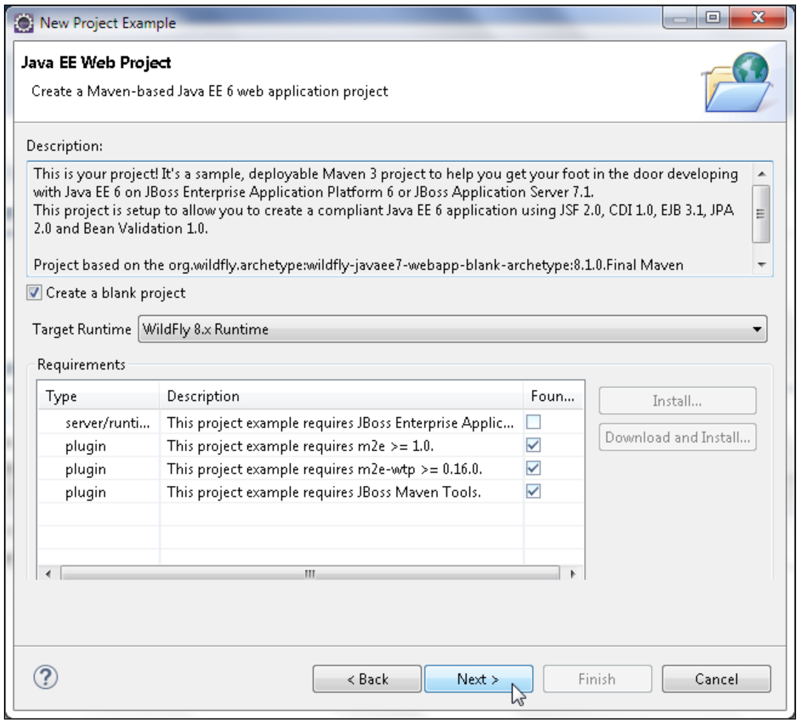
Create a WildFly 8.1.0 runtime as discussed in *Chapter 1*, *Getting Started with EJB 3.x*. Create a MySQL data source with the JNDI name `java:jboss/datasources/MySQLDS` as explained in *Chapter 1*, *Getting Started with EJB 3.x*.

Creating a Java EE web project

First, we need to create a Java EE web project for which you need to select **File** | **New** | **Other**. In **New**, select **Web** | **Java EE Web Project**, as shown in the following screenshot:



In the **Java EE Web Project** wizard, select **Create a blank project** and select `WildFly 8.x Runtime` as **Target Runtime**, which is shown as follows. A test gets run to find whether the required plugins are installed. Then, click on **Next**.



This is your project! It's a sample, deployable Maven 3 project to help you get your foot in the door developing with Java EE 6 on JBoss Enterprise Application Platform 6 or JBoss Application Server 7.1.
This project is setup to allow you to create a compliant Java EE 6 application using JSF 2.0, CDI 1.0, EJB 3.1, JPA 2.0 and Bean Validation 1.0.

Project based on the org.wildfly.archetype:wildfly-javaee7-webapp-blank-archetype:8.1.0.Final Maven

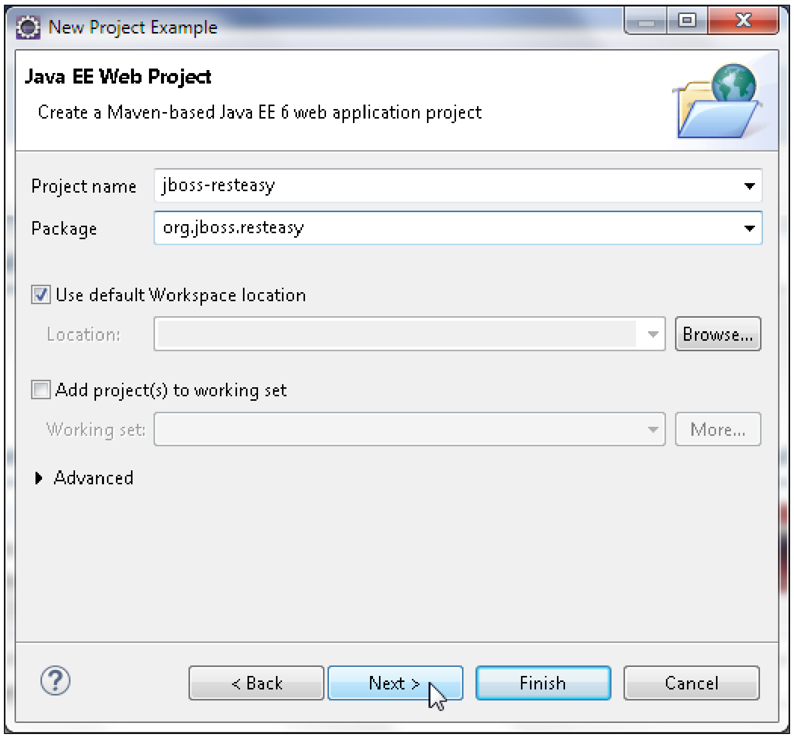
☑ Create a blank project

Target Runtime: WildFly 8.x Runtime

Requirements

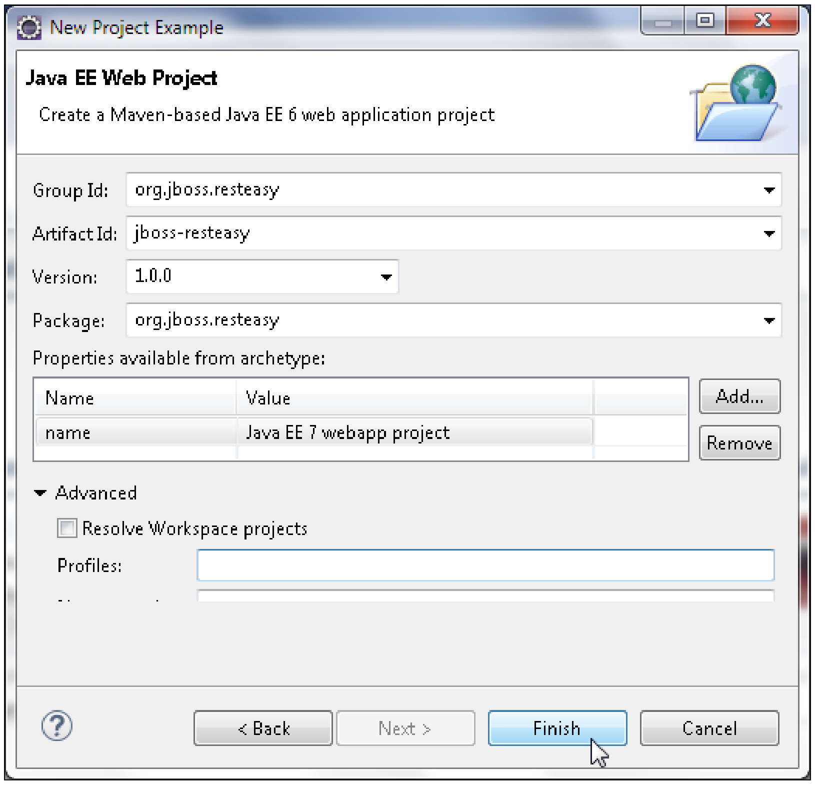
Type	Description	Foun...
server/runti...	This project example requires JBoss Enterprise Applic...	☐
plugin	This project example requires m2e >= 1.0.	☑
plugin	This project example requires m2e-wtp >= 0.16.0.	☑
plugin	This project example requires JBoss Maven Tools.	☑

Specify **Project name** (`jboss-resteasy`) and **Package** (`org.jboss.resteasy`), and click on **Next** as follows:



Project name: jboss-resteasy
Package: org.jboss.resteasy
[✓] Use default Workspace location

Specify **Group Id** (`org.jboss.resteasy`), **Artifact Id** (`jboss-resteasy`), **Version** (`1.0.0`), and **Package** (`org.jboss.resteasy`), and click on **Next**, as shown here:


New Project Example

Java EE Web Project

Create a Maven-based Java EE 6 web application project

Group Id: org.jboss.resteasy

Artifact Id: jboss-resteasy

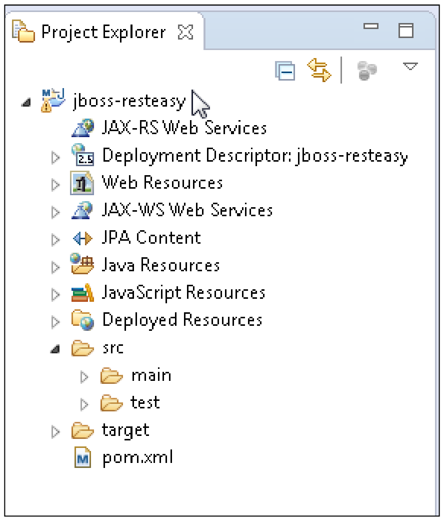
Version: 1.0.0

Package: org.jboss.resteasy

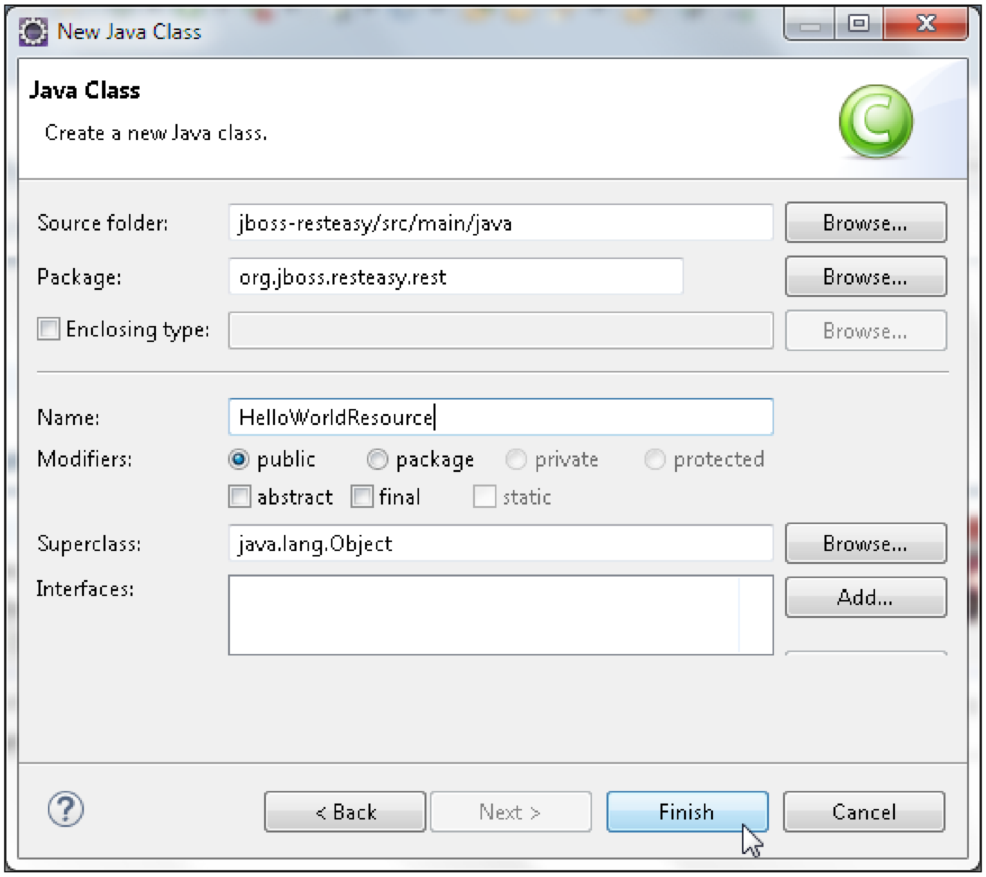
Properties available from archetype:

Name	Value
name	Java EE 7 webapp project

The `jboss-resteasy` Maven project gets created and gets added to **Project Explorer**, as shown here:



Next, add a JAX-RS resource class (`HelloWorldResource`), which is just a Java class. Select **File** | **New** | **Other**, and in **New**, select **Java** | **Class** and click on **Next**. Select **Source folder** (`jboss-resteasy/src/main/java`) and specify **Package** (`org.jboss.resteasy.rest`) and the class **Name** (`HelloWorldResource`), and click on **Finish**, as shown here:



Source folder: jboss-resteasy/src/main/java

Package: org.jboss.resteasy.rest

Enclosing type:

Name: HelloWorldResource

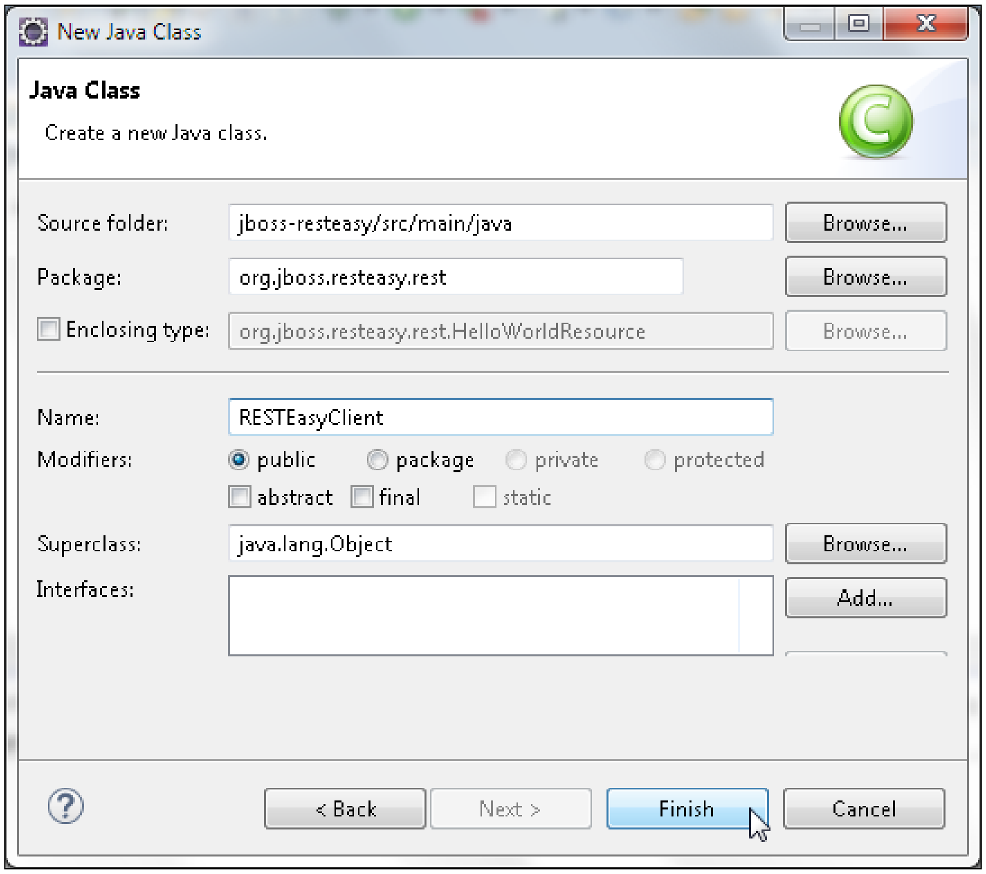
Modifiers: [✓ public] package private protected

abstract final static

Superclass: java.lang.Object

Interfaces:

Similarly, add a Java client class (`RESTEasyClient`) as follows:



Java Class

Create a new Java class.

Source folder: jboss-resteasy/src/main/java

Package: org.jboss.resteasy.rest

[ ] Enclosing type: org.jboss.resteasy.rest.HelloWorldResource

Name: RESTEasyClient

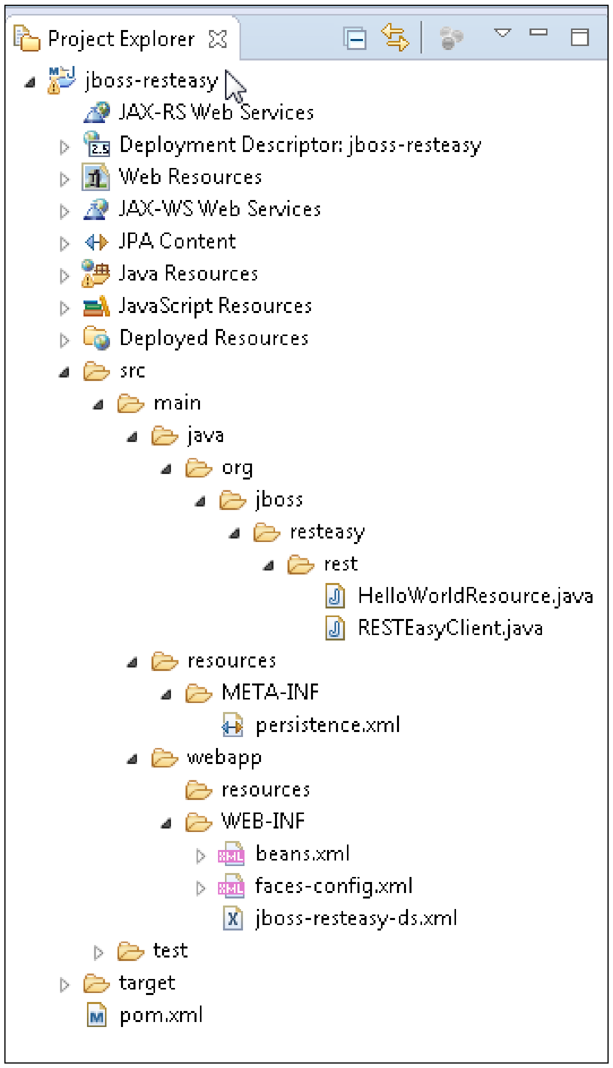
Modifiers: [● public] ○ package ○ private ○ protected

[ ] abstract [ ] final [ ] static

Superclass: java.lang.Object

Interfaces:

The directory structure of the `jboss-resteasy` application is shown in the following screenshot:



```
Project Explorer
jboss-resteasy
    JAX-RS Web Services
    Deployment Descriptor: jboss-resteasy
    Web Resources
    JAX-WS Web Services
    JPA Content
    Java Resources
    JavaScript Resources
    Deployed Resources
    src
        main
            java
                org
                    jboss
                        resteasy
                            rest
                                HelloWorldResource.java
                                RESTEasyClient.java
            resources
                META-INF
                    persistence.xml
            webapp
                resources
                WEB-INF
                    beans.xml
                    faces-config.xml
                    jboss-resteasy-ds.xml
        test
    target
    pom.xml
```

Add the JAX-RS- and RESTEasy-related dependencies to `pom.xml`, as follows:

```
<dependencies>

  <!-- Import the JAX-RS API, we use provided scope as the API is
included in JBoss WildFly -->
  <dependency>
    <groupId>org.jboss.resteasy</groupId>
```

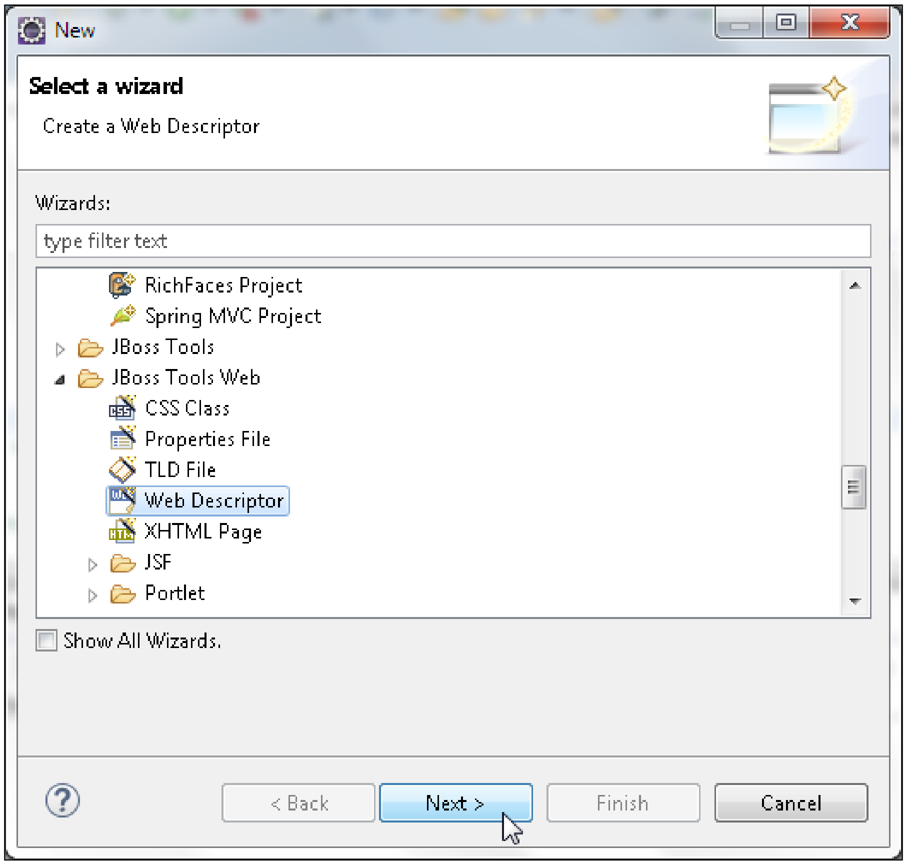
```
        <artifactId>jaxrs-api</artifactId>
        <version>3.0.10.Final</version>
        <scope>provided</scope>
      </dependency>

      <dependency>
        <groupId>org.jboss.resteasy</groupId>
        <artifactId>resteasy-jackson-provider</artifactId>
        <version>3.0.10.Final</version>
        <scope>provided</scope>
      </dependency>
      <dependency>
        <groupId>org.jboss.resteasy</groupId>
        <artifactId>resteasy-jaxrs</artifactId>
        <version>3.0.10.Final</version>
        <scope>provided</scope>
      </dependency>

      <dependency>
        <groupId>org.apache.httpcomponents</groupId>
        <artifactId>httpclient</artifactId>
        <version>4.3.6</version>
        <scope>provided</scope>
      </dependency>

      <dependency>
        <groupId>org.apache.httpcomponents</groupId>
        <artifactId>httpcore</artifactId>
        <version>4.3.3</version>
        <scope>provided</scope>
      </dependency>

      <dependency>
        <groupId>org.jboss.resteasy</groupId>
        <artifactId>resteasy-client</artifactId>
        <version>3.0.10.Final</version>
        <scope>provided</scope>
      </dependency>
```

To the build, add the Maven compiler plugin and the Maven WAR plugin. In the Maven WAR plugin configuration, specify the output directory to which the built application is to be deployed: the `C:\wildfly-8.1.0.Final\standalone\deployments` directory:

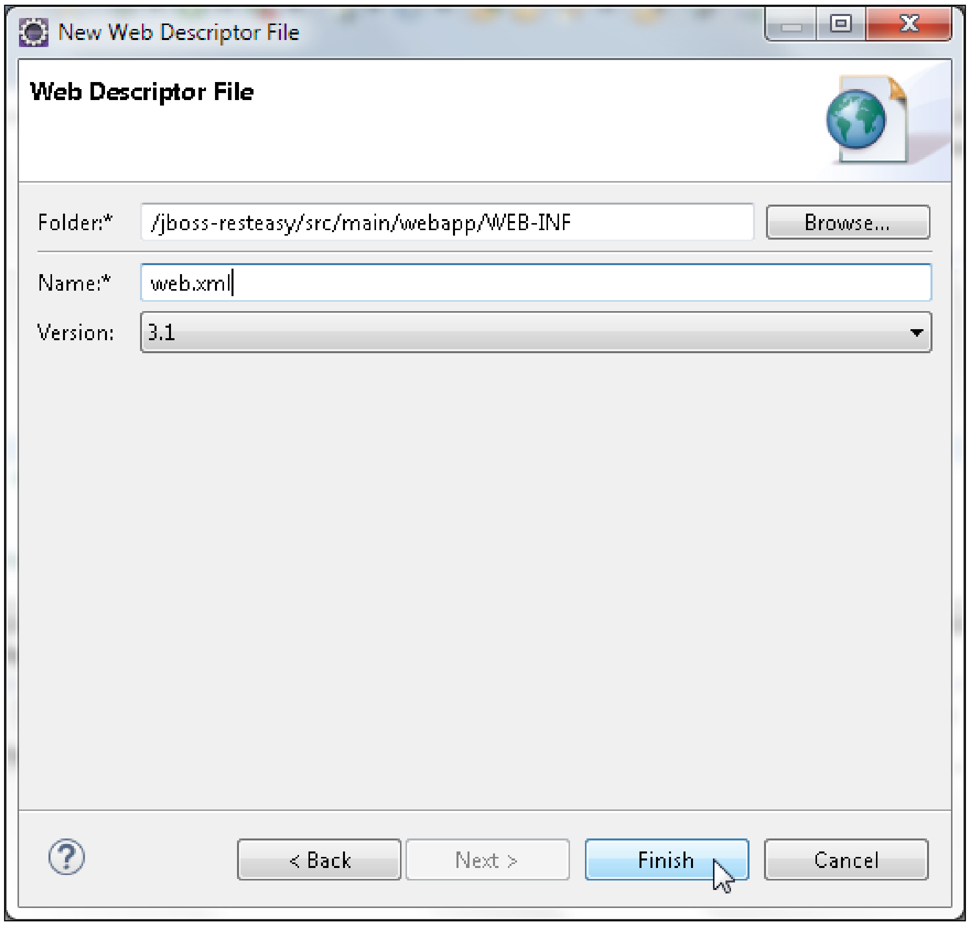
```
<build>
<!-- Maven will append the version to the finalName (which is the name
given to the generated war, and hence the context root) -->
```

```
    <finalName>${project.artifactId}</finalName>
    <plugins>
      <plugin>
        <artifactId>maven-war-plugin</artifactId>
        <version>${version.war.plugin}</version>
        <configuration>
          <outputDirectory>C:\wildfly-8.1.0.Final\standalone\
  deployments</outputDirectory>
          <failOnMissingWebXml>false</failOnMissingWebXml>
        </configuration>
      </plugin>
    </plugins>
  </build>
```

The complete `pom.xml` is available in the code download for this chapter. Next, create the web deployment descriptor `web.xml`. Select **File** | **New** | **Other**. In **New**, select **JBoss Tools Web** | **Web Descriptor** and click on **Next**, as shown here:



In **New Web Descriptor File**, select **Folder** as the `WEB-INF` directory, specify **Name** as `web.xml`, and select **Version** as `3.1`, as shown here:

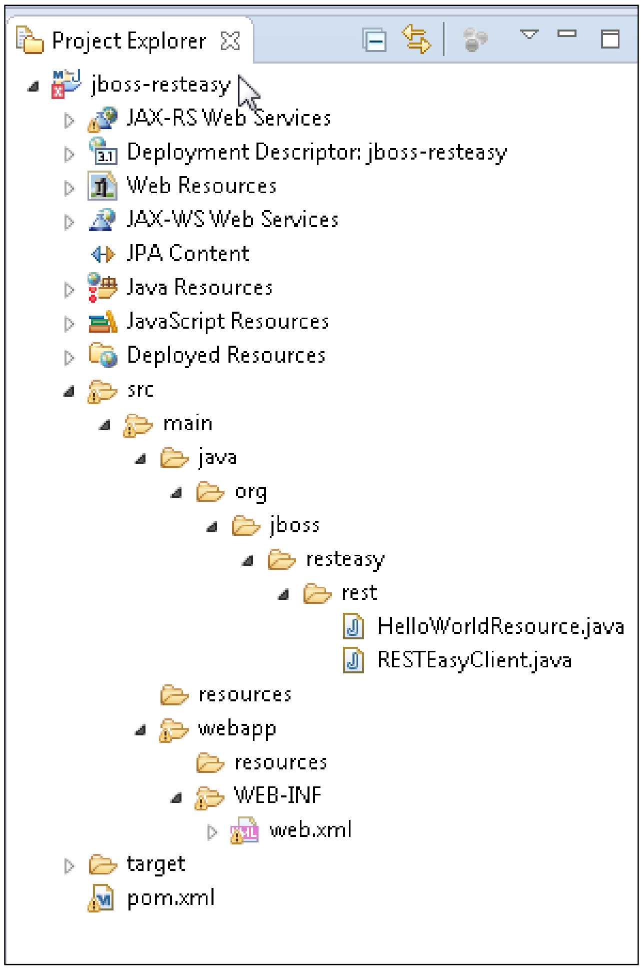


Folder:* /jboss-resteasy/src/main/webapp/WEB-INF

Name:* web.xml

Version: 3.1

The `web.xml` deployment descriptor gets added to `WEB-INF`, which is shown as follows:



Add the RESTEasy dispatcher servlet to the `web.xml` file including its URL mapping. Add the context parameter required for RESTEasy to scan for JAX-RS classes. Also, add the context parameter for the RESTEasy servlet mapping the prefix. The `web.xml` file is listed as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="3.1"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.
jcp.org/xml/ns/javaee/web-app_3_1.xsd">
  <context-param>
```

```
        <param-name>resteasy.servlet.mapping.prefix</param-name>
        <param-value>/rest</param-value>
    </context-param>
    <context-param>
        <param-name>resteasy.scan</param-name>
        <param-value>true</param-value>
    </context-param>
    <listener>
        <listener-class>org.jboss.resteasy.plugins.server.servlet.
ResteasyBootstrap</listener-class>
    </listener>
    <servlet>
        <servlet-name>Resteasy</servlet-name>
        <servlet-class>org.jboss.resteasy.plugins.server.servlet.
HttpServletDispatcher</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>Resteasy</servlet-name>
        <url-pattern>/rest/*</url-pattern>
    </servlet-mapping>
</web-app>
```

The Client API

The Client API is a high-level API used to access web resources and integrate with JAX-RS providers and is included in the `javax.ws.rs.client` package.

> Previously, different implementations provided the Client API, but in JAX-RS 2.0, the Client API is provided as a core API.

Creating a client instance

A `Client` instance is required to build and run client requests to access or consume web resources. In the RESTEasy client class `RESTEasyClient.java`, create a `Client` instance from `ClientBuilder` using the `newClient()` method as follows:

```
Client client = ClientBuilder.newClient();
```

Providers, filters, and features can be configured with the `Client` object using the `register()` method. For example, the `org.jboss.resteasy.plugins.providers.JaxrsFormProvider.class` provider class is registered as follows:

```
client.register(org.jboss.resteasy.plugins.providers.
JaxrsFormProvider.class);
```

Accessing a resource

The `Client` API is used to access a web resource as follows:

1. Create a `WebTarget` object from the resource URI using the overloaded `target()` method of the `Client` object. The path appended to the URI is to enable the REST service to handle multiple inputs:

   ```
   WebTarget target = client.target("http://localhost:8080/jboss-
   resteasy/rest/helloworld");
   ```

2. Add one or more path elements to the `WebTarget` object if required using the `path()` method, which returns a `WebTarget` object:

   ```
   WebTarget target = target.path("text");
   ```

3. Create a request from the `WebTarget` object using the overloaded `request()` method, in which you need to define the accepted response media types. Invoke the HTTP `GET` method for the request using the overloaded `get()` method to obtain an invocation response as a `Response` object:

   ```
   Response response=target.request("text/plain").get();
   ```

4. Obtain the message entity input stream as a `String` object:

   ```
   String value = response.readEntity(String.class);
   ```

5. The fluent API can be used to build and submit the client request and obtain a response by linking the method invocations:

   ```
   String response = client.target("http://localhost:8080/jboss-
   resteasy/rest/helloworld").path("text").request("text/plain").
   get(String.class);
   ```

The `RESTEasyClient.java` class is listed as follows:

```
package org.jboss.resteasy.rest;

import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;

import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.*;

import org.jboss.resteasy.client.jaxrs.ResteasyClient;

public class RESTEasyClient {

  public static void main(String[] args) {

    Client client =    ClientBuilder.newClient();
```

```
    String response = client.target("http://localhost:8080/jboss-
resteasy/rest/helloworld").path("text").request("text/plain").
get(String.class);


    System.out.println(response);
  }

}
```

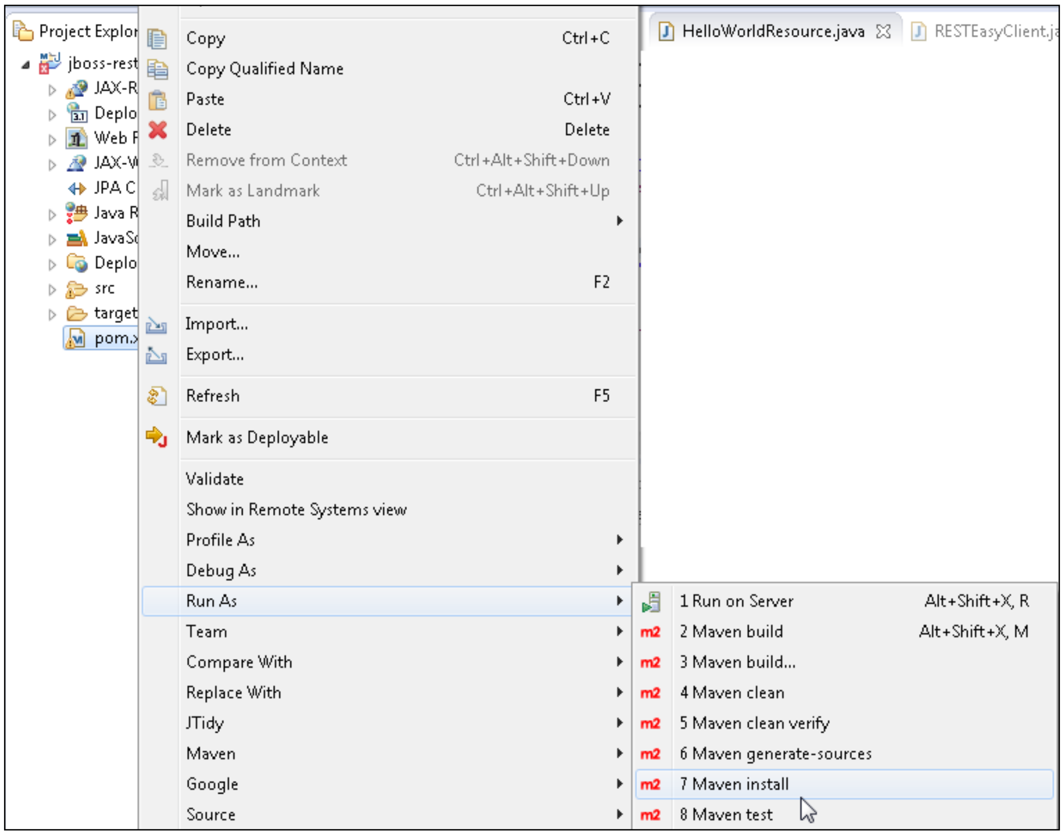
Create a resource class hosted at the `/helloworld` URI path to test the `Client` API. Add a resource method at the relative URI path `/text` to return a `Hello` message from a name. The resource class `HelloWorldResource` is listed as follows:

```
package org.jboss.resteasy.rest;

import javax.ws.rs.GET;
import javax.ws.rs.Produces;
import javax.ws.rs.Path;


@Path("/helloworld")
public class HelloWorldResource {

  @GET
  @Produces("text/plain")
  @Path("/text")
  public String getClichedMessage() {

    return "Hello John Smith";
  }

}
```

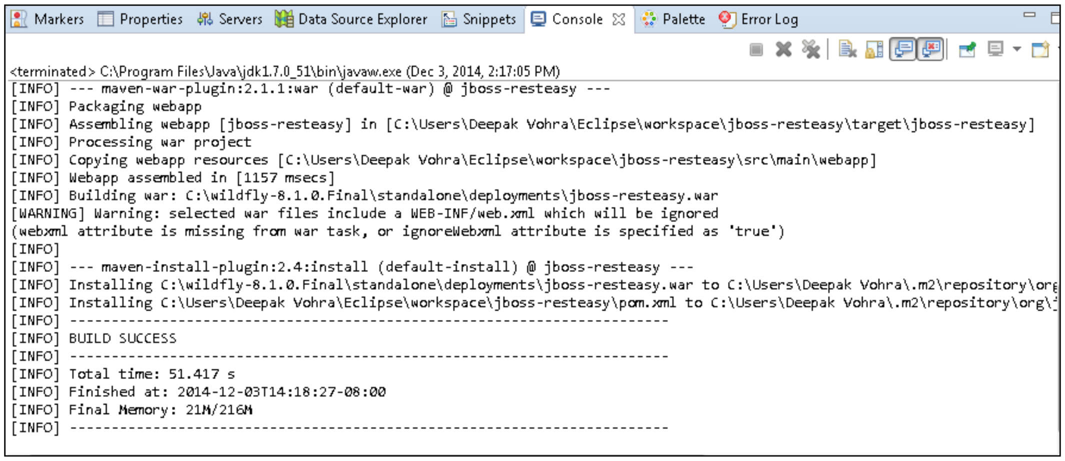
To test the resource class and the client, compile, package, and deploy the `jboss-resteasy` application to WildFly. We will add the output directory as the WildFly `deployments` directory to the configuration for the Maven WAR plugin, as shown here:

```
<plugin>
  <artifactId>maven-war-plugin</artifactId>
  <version>${version.war.plugin}</version>
  <configuration>
    <outputDirectory>C:\wildfly-8.1.0.Final\standalone\deployments</
outputDirectory>
    <failOnMissingWebXml>false</failOnMissingWebXml>
  </configuration>
</plugin>
```

Right-click on `pom.xml` and select **Run As** | **Maven install**, which is shown as follows:

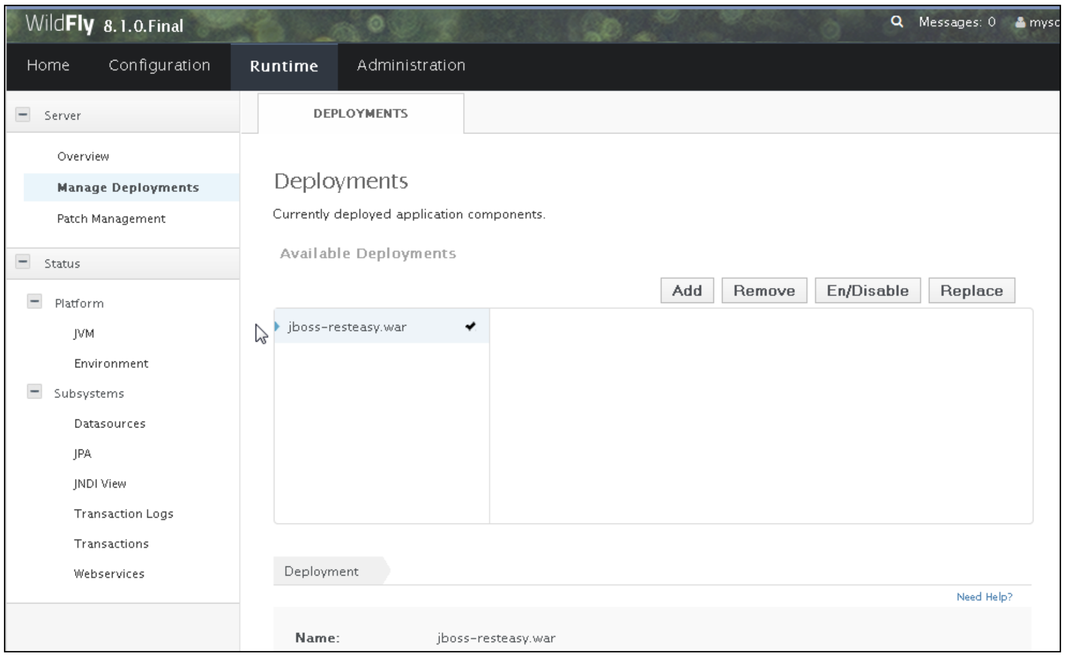


The `jboss-resteasy` application gets compiled, built, and outputted to the WildFly 8.1 `deployments` directory is indicated by the **BUILD SUCCESS** message in the **Console**, which is shown as follows:



```
<terminated> C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Dec 3, 2014, 2:17:05 PM)
[INFO] --- maven-war-plugin:2.1.1:war (default-war) @ jboss-resteasy ---
[INFO] Packaging webapp
[INFO] Assembling webapp [jboss-resteasy] in [C:\Users\Deepak Vohra\Eclipse\workspace\jboss-resteasy\target\jboss-resteasy]
[INFO] Processing war project
[INFO] Copying webapp resources [C:\Users\Deepak Vohra\Eclipse\workspace\jboss-resteasy\src\main\webapp]
[INFO] Webapp assembled in [1157 msecs]
[INFO] Building war: C:\wildfly-8.1.0.Final\standalone\deployments\jboss-resteasy.war
[WARNING] Warning: selected war files include a WEB-INF/web.xml which will be ignored
(webxml attribute is missing from war task, or ignoreWebxml attribute is specified as 'true')
[INFO]
[INFO] --- maven-install-plugin:2.4:install (default-install) @ jboss-resteasy ---
[INFO] Installing C:\wildfly-8.1.0.Final\standalone\deployments\jboss-resteasy.war to C:\Users\Deepak Vohra\.m2\repository\org
[INFO] Installing C:\Users\Deepak Vohra\Eclipse\workspace\jboss-resteasy\pom.xml to C:\Users\Deepak Vohra\.m2\repository\org\
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 51.417 s
[INFO] Finished at: 2014-12-03T14:18:27-08:00
[INFO] Final Memory: 21M/216M
[INFO] ------------------------------------------------------------------------
```

Now, log in to the WildFly 8.1 **Administration Console** and click on **Manage Deployments**. The `jboss-resteasy.war` application should be listed as deployed, which is shown in the following screenshot:



Deployments

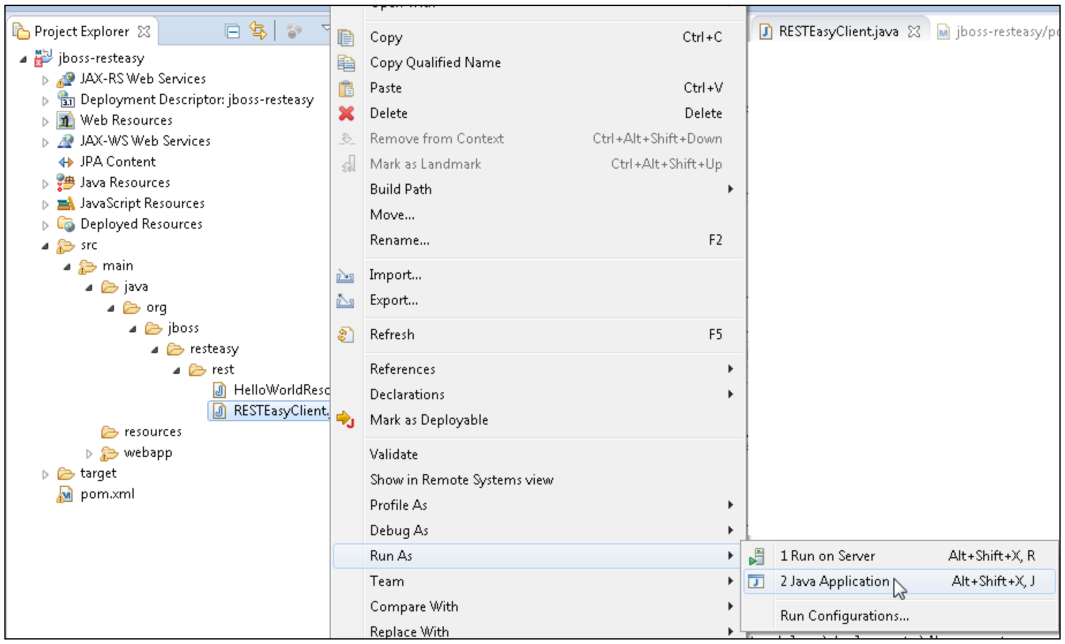
Currently deployed application components.

Available Deployments

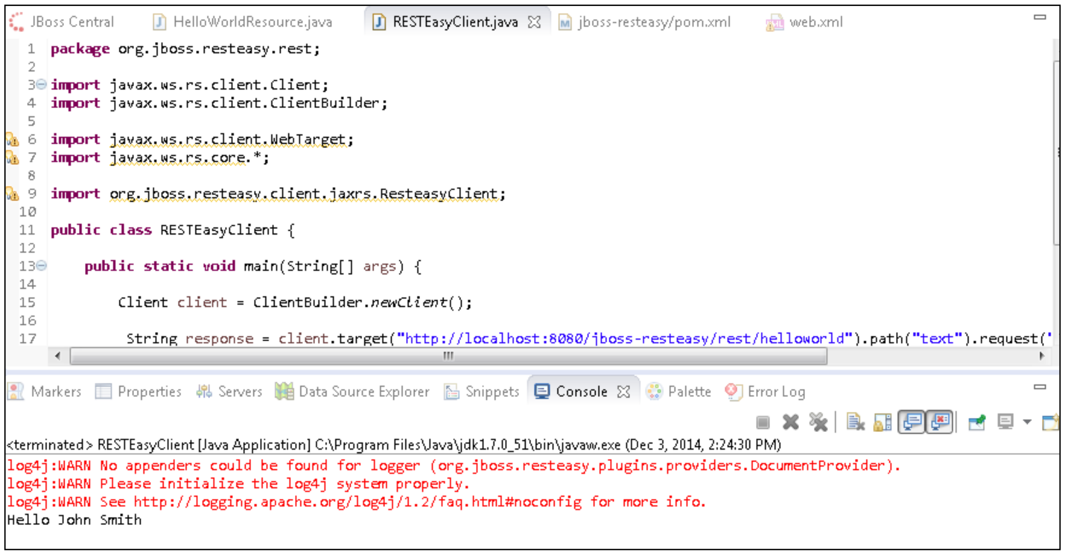
jboss-resteasy.war ✔

Name: jboss-resteasy.war

To test the client, right-click on the `RESTEasyClient.java` class in **Project Explorer** and select **Run As** | **Java Application**, as shown here:



The client application runs to invoke the resource class and produces the output, as shown in the following **Console**:



```
package org.jboss.resteasy.rest;

import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;

import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.*;

import org.jboss.resteasy.client.jaxrs.ResteasyClient;

public class RESTEasyClient {

    public static void main(String[] args) {

        Client client = ClientBuilder.newClient();

         String response = client.target("http://localhost:8080/jboss-resteasy/rest/helloworld").path("text").request('
```

```
<terminated> RESTEasyClient [Java Application] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Dec 3, 2014, 2:24:30 PM)
log4j:WARN No appenders could be found for logger (org.jboss.resteasy.plugins.providers.DocumentProvider).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
Hello John Smith
```

Setting a query parameter

To invoke a resource method with parameters, the `@QueryParam` annotation can be used to bind request parameters to resource method parameters. In a variation of the resource class used in the previous subsection, add a `String` parameter to the resource method. Annotate the parameter declaration with `@QueryParam` and set its default value as `DefaultValue`:

```
@GET
@Produces("text/plain")
@Path("/text")
public String getClichedMessage(@QueryParam("name") @
DefaultValue("John Smith") String name) {
  return "Hello " +name;
}
```

In the client class, the query parameter can be sent in the request using the `queryParam()` method as follows:

```
String response = client.target("http://localhost:8080/jboss-resteasy/
rest/helloworld").path("text").queryParam("name", "John Smith").
request("text/plain").get(String.class);
```

Alternatively, the query parameter can be included in the request URI, which is shown as follows:

```
String response = client.target("http://localhost:8080/jboss-resteasy/
rest/helloworld/text?name=John Smith").request("text/plain").
get(String.class);
```

Setting a template parameter

The resource URI can also be built using template parameters. In a variation of `HelloWorldResource`, specify a template parameter `{name}` in the `@Path` annotating the resource method. Bind the template parameter to the resource method parameter using the `@PathParam` annotation:

```
@GET
@Produces("text/plain")
@Path("/text/{name}")
public String getClichedMessage(@PathParam("name") String name) {
  return "Hello " +name;
}
```

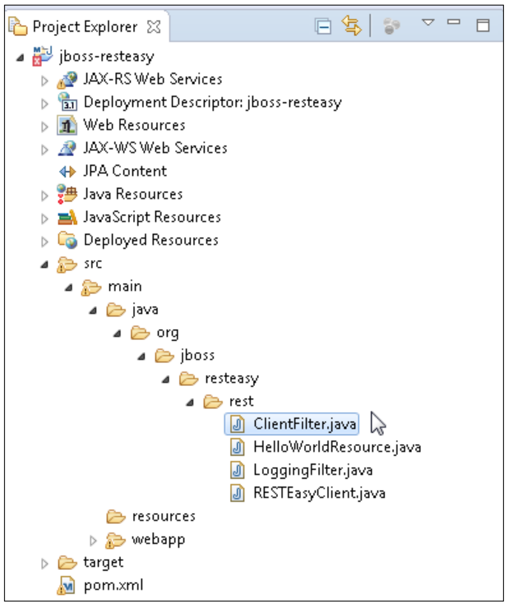
In the `RESTEasyClient` class, include the value for the `{name}` template parameter in the resource URI as follows:

```
String response = client.target("http://localhost:8080/jboss-resteasy/
rest/helloworld/text/John Smith").request("text/plain").get(String.
class);
```

Redeploy the `jboss-resteasy` application and rerun the client to produce the same output.

Filters and interceptors

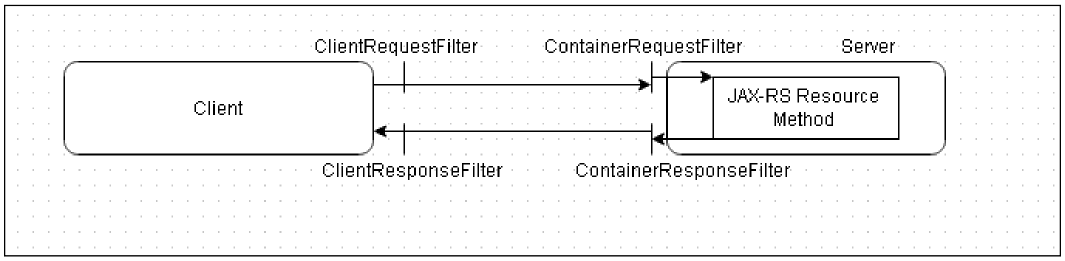
Filters provide extended functionality such as logging and authentication. Interceptors provide extended functions such as entity compression. In this section, we will discuss the support for filters at specific extension points in JAX-RS 2.0 implementation. The two types of filters are provided in JAX-RS 2.0: **client filters** and **container filters**. The client filters are on the client side and the container filters are on the container side. Interfaces corresponding to the client filters are included in the Client API and are `javax.ws.rs.client.ClientRequestFilter` and `javax.ws.rs.client.ClientResponseFilter`. Interfaces for the container filters, which are included in the Server API, are `javax.ws.rs.container.ContainerRequestFilter` and `javax.ws.rs.container.ContainerResponseFilter`. To be discovered by the JAX-RS runtime, filters implementing the interfaces must be annotated with the `@Provider` annotation. Create Java classes `LoggingFilter` (for the container filter example), and `ClientFilter` (for the client filter example), as shown in **Project Explorer**.



Before we discuss the client and container filters, we need to discuss the junctions at which the filters intercept communication between the client and the server:

1. The `ClientRequestFilter` intercepts communication before the client HTTP request is sent over to the server.
2. The `ContainerRequestFilter` intercepts after the client is sent over to the server but before the JAX-RS resource method is invoked.
3. The `ContainerResponseFilter` intercepts after the JAX-RS resource method is invoked but before the response is sent back to the client.
4. The `ClientResponseFilter` is invoked after the server HTTP response is sent over to the client but before the response is unmarshalled.

The junctions of request/response interception are illustrated in the following diagram:



Creating a client filter

First, we will discuss the client filters with an example. The `ClientRequestFilter` is run in the invocation pipeline before the HTTP request is delivered to the network. The `ClientRequestFilter` should be annotated by `@Provider`, which is the marker that is discovered by the JAX-RS runtime during the scanning phase. The `ClientResponseFilter` is run after the response is received from the server and before the control is returned to the application. Make the `ClientFilter` class implement the `ClientRequestFilter` and `ClientResponseFilter` interfaces. Add implementation for the `filter(ClientRequestContext arg0)` and `filter(ClientRequestContext arg0, ClientResponseContext arg1)` methods. In the `ClientRequestFilter` implementation method `filter(ClientRequestContext arg0)`, output some headers using the `getHeaderString(String)` method of `ClientRequestContext`. For example, the `Accept-Charset` and `Accept-Encoding` headers give out the following output:

```
System.out.println("Accept-Charset: " + arg0.getHeaderString("Accept-
Charset"));
System.out.println("Accept-Encoding: " + arg0.getHeaderString("Accept-
Encoding"));
```

Set a new resource URI using the `setUri(URI)` method as follows:

```
arg0.setUri(new URI("http://localhost:8080/jboss-resteasy/rest/
helloworld/text/Smith,John"));
```

The `ClientFilter` class is listed as follows:

```
package org.jboss.resteasy.rest;

import java.io.IOException;
import java.net.URI;
import java.net.URISyntaxException;

import javax.ws.rs.client.ClientRequestContext;
import javax.ws.rs.client.ClientRequestFilter;
import javax.ws.rs.client.ClientResponseContext;
import javax.ws.rs.client.ClientResponseFilter;

import javax.ws.rs.core.Response;
import javax.ws.rs.ext.Provider;

@Provider
public class ClientFilter implements ClientRequestFilter,
ClientResponseFilter {

  @Override
  public void filter(ClientRequestContext arg0, ClientResponseContext
arg1)
  throws IOException {
  }
  @Override
  public void filter(ClientRequestContext arg0) throws IOException {

    System.out.println("Entity Class: " + arg0.getEntityClass());
    System.out.println("Accept: " + arg0.getHeaderString("Accept"));
    System.out.println("Accept-Charset: "
    + arg0.getHeaderString("Accept-Charset"));
    System.out.println("Accept-Encoding: "
    + arg0.getHeaderString("Accept-Encoding"));
    System.out.println("Accept-Language: "
    + arg0.getHeaderString("Accept-Language"));
    System.out.println("Accept-Ranges: "
    + arg0.getHeaderString("Accept-Ranges"));
    System.out.println("Allow: " + arg0.getHeaderString("Allow"));
    System.out.println("Authorization: "
    + arg0.getHeaderString("Authorization"));
    System.out.println("Cache-Control: "
    + arg0.getHeaderString("Cache-Control"));
    System.out.println("Content-Encoding: "
    + arg0.getHeaderString("Content-Encoding"));
    System.out.println("Content-Location: "
    + arg0.getHeaderString("Content-Location"));
    System.out.println("Accept-Encoding: "
    + arg0.getHeaderString("Accept-Encoding"));
    System.out.println("Content-Type: "
    + arg0.getHeaderString("Content-Type"));
    System.out.println("Host: " + arg0.getHeaderString("Host"));
    System.out.println("Pragma: " + arg0.getHeaderString("Pragma"));
    System.out.println("Server: " + arg0.getHeaderString("Server"));
    System.out.println("User-Agent: " + arg0.getHeaderString("User-
Agent"));

    try {
      arg0.setUri(new URI(
      "http://localhost:8080/jboss-resteasy/rest/helloworld/text/
Smith,John"));
    } catch (URISyntaxException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
    }
    //arg0.abortWith(Response.notAcceptable(null).build());
  }
}
```

For a client filter issue that could occur, refer to the section *Fixing a Common Issue* at the end of this chapter. In the client class, `RESTEasyClient` registers the client filter with the client:

```
client.register(ClientFilter.class);
```

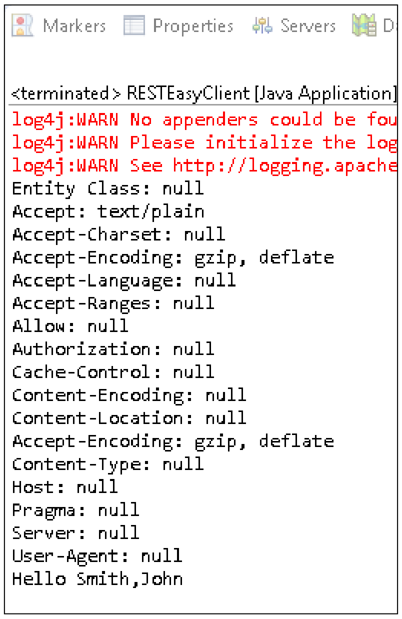
We will use the following root resource class `HelloWorldResource` to test the client filter:

```
package org.jboss.resteasy.rest;
import javax.ws.rs.GET;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.Path;

@Path("/helloworld")
public class HelloWorldResource {

  @GET
  @Produces("text/plain")
  @Path("/text/{name}")
  public String getClichedMessage(@PathParam("name") String name) {
    return "Hello " +name;
  }
}
```

The client class `RESTEasyClient` to test the client filter with is listed as follows:

```
package org.jboss.resteasy.rest;
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.*;

public class RESTEasyClient {

  public static void main(String[] args) {
    Client client = ClientBuilder.newClient();
    client.register(ClientFilter.class);
    String response = client.target("http://localhost:8080/jboss-
resteasy/rest/helloworld/text/John Smith").request("text/plain").
get(String.class);
    System.out.println("Text response "+ response);
  }
}
```

Redeploy the `jboss-resteasy` application. To redeploy, right-click on `pom.xml` in **Project Explorer** and select **Run As** | **Maven clean**, and subsequently, right-click on `pom.xml` and select **Run As** | **Maven install**. Run the `RESTEasyClient.java` class to generate the following output in the **Console** screen shown as follows:



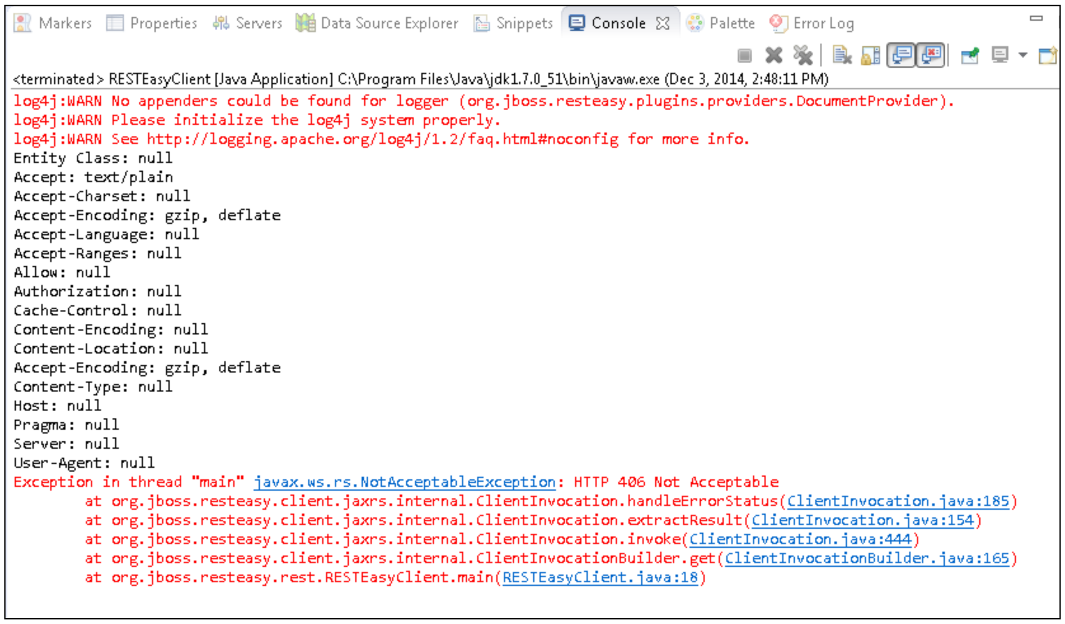
```
<terminated> RESTEasyClient [Java Application]
log4j:WARN No appenders could be fou
log4j:WARN Please initialize the log
log4j:WARN See http://logging.apache
Entity Class: null
Accept: text/plain
Accept-Charset: null
Accept-Encoding: gzip, deflate
Accept-Language: null
Accept-Ranges: null
Allow: null
Authorization: null
Cache-Control: null
Content-Encoding: null
Content-Location: null
Accept-Encoding: gzip, deflate
Content-Type: null
Host: null
Pragma: null
Server: null
User-Agent: null
Hello Smith,John
```

As we modified the resource URI in the client filter, the response message is not for John Smith as specified in the client class, but for **Smith, John**.

The filter chain processing may be aborted and response is returned to the client with the `abortWith(Response response)` method. The client response filters get applied before the client gets the response. As an example, break the filter chain and return a `notAcceptable(null)` response:

```
arg0.abortWith(Response.notAcceptable(null).build());
```

Keep `RESTEasyClient` and `HelloWorldResource` the same and redeploy the `jboss-restaesy` application. Rerun the `RESTEasyClient` class to generate the following output, which includes a `NotAcceptableException` shown as follows:



```
<terminated> RESTEasyClient [Java Application] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Dec 3, 2014, 2:48:11 PM)
log4j:WARN No appenders could be found for logger (org.jboss.resteasy.plugins.providers.DocumentProvider).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
Entity Class: null
Accept: text/plain
Accept-Charset: null
Accept-Encoding: gzip, deflate
Accept-Language: null
Accept-Ranges: null
Allow: null
Authorization: null
Cache-Control: null
Content-Encoding: null
Content-Location: null
Accept-Encoding: gzip, deflate
Content-Type: null
Host: null
Pragma: null
Server: null
User-Agent: null
Exception in thread "main" javax.ws.rs.NotAcceptableException: HTTP 406 Not Acceptable
	at org.jboss.resteasy.client.jaxrs.internal.ClientInvocation.handleErrorStatus(ClientInvocation.java:185)
	at org.jboss.resteasy.client.jaxrs.internal.ClientInvocation.extractResult(ClientInvocation.java:154)
	at org.jboss.resteasy.client.jaxrs.internal.ClientInvocation.invoke(ClientInvocation.java:444)
	at org.jboss.resteasy.client.jaxrs.internal.ClientInvocationBuilder.get(ClientInvocationBuilder.java:165)
	at org.jboss.resteasy.rest.RESTEasyClient.main(RESTEasyClient.java:18)
```

Creating a container filter

Container filters are Server API filters. A `ContainerRequestFilter` filter is run after receiving a request from the client. A `ContainerResponseFilter` is run in the response pipeline before the HTTP response is delivered to the client. Next, we will create a container filter for logging/outputting some information about the request. Extension points before and after the match are provided in the `ContainerRequestFilter` interface. The pre-match filter is run before the request has been matched with a resource method, and the post-match filter is applied after the resource method matching; the default is post-match. We will use pre-match with the `@PreMatching` annotation. Annotate the filter class with `@Provider` for the filter to be discovered by the JAX-RS runtime during the scanning phase.

Make the example container filter `LoggingFilter`, implement the `ContainerRequestFilter`, `ContainerResponseFilter` interfaces, and provide implementation for the `filter(ContainerRequestContext requestContext)` and `filter(ContainerRequestContext requestContext, ContainerResponseContext responseContext)` methods. In the `ContainerRequestFilter` implementation method `filter(ContainerRequestContext requestContext)`, output the request method with the `getMethod()` method, request URI with the `getUriInfo().getAbsolutePath()`, media type with `getMediaType()`, and acceptable media types with `getAcceptableMediaTypes()`. Include a no-argument constructor in the `LoggingFilter` so that the filter may be instantiated. Register the `LoggingFilter` with the client configuration using the `register` method. Comment out the registration of the `ClientFilter` as we will apply only the `LoggingFilter` in the `RESTEasyClient` class:

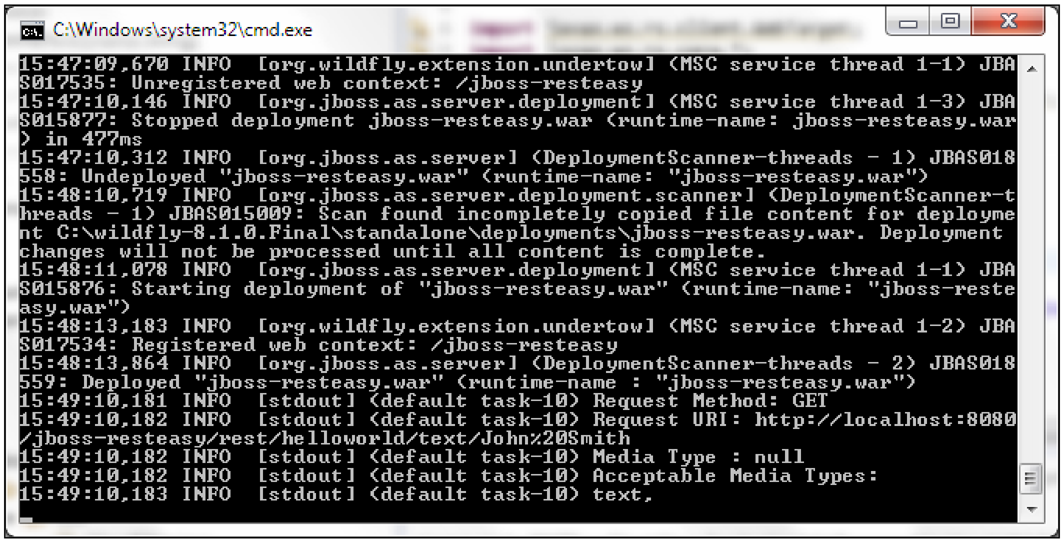
```
Client client = ClientBuilder.newClient();
//client.register(ClientFilter.class);
client.register(LoggingFilter.class);
```

By default, container filters are bound to all the resources the client request is sent to, but a resource-specific container filter can be applied using the `@NameBinding` annotation:

The `LoggingFilter.java` class is listed as follows:

```
package org.jboss.resteasy.rest;

import java.io.IOException;
import java.util.Iterator;
import java.util.List;
import javax.ws.rs.container.ContainerRequestContext;
import javax.ws.rs.container.ContainerRequestFilter;
import javax.ws.rs.container.ContainerResponseContext;
import javax.ws.rs.container.ContainerResponseFilter;
import javax.ws.rs.container.PreMatching;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.ext.Provider;

@Provider
@PreMatching
public class LoggingFilter implements ContainerRequestFilter,
ContainerResponseFilter {
  public LoggingFilter() {
  }

  @Override
  public void filter(ContainerRequestContext requestContext)
  throws IOException {
    System.out.println("Request Method: " + requestContext.
getMethod());
    System.out.println("Request URI: "+ requestContext.getUriInfo().
getAbsolutePath());
    System.out.println("Media Type : "+ requestContext.
getMediaType());
    List<MediaType> mediaTypes = requestContext.
getAcceptableMediaTypes();
    Iterator<MediaType> iter = mediaTypes.iterator();
    System.out.println("Acceptable Media Types: ");
    while (iter.hasNext()) {
      MediaType mediaType = iter.next();
      System.out.println(mediaType.getType() + ", ");

    }
  }
  @Override
  public void filter(ContainerRequestContext requestContext,
  ContainerResponseContext responseContext) throws IOException {

  }
}
```

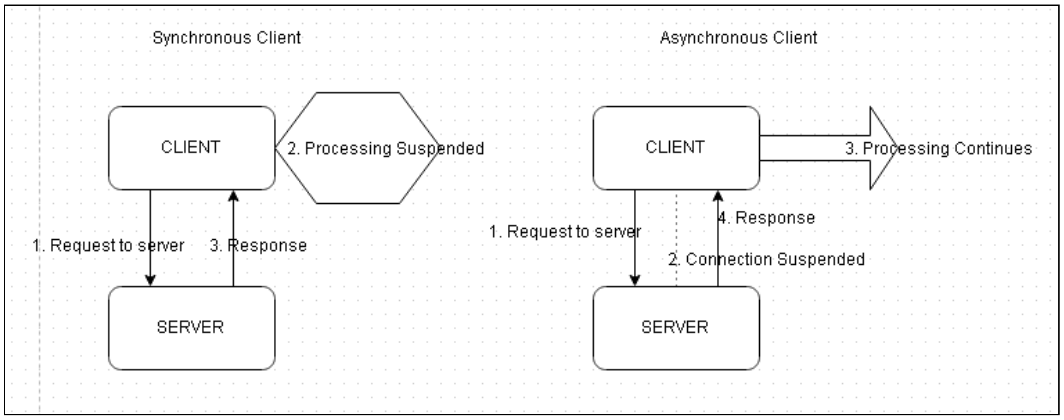
Keep `RESTEasyClient` and `HelloWorldResource` the same as the `ClientFilter` example and redeploy the `jboss-restaesy` application. Run the `RESTEasyClient` application to generate the output shown from the container filter, which is shown here:



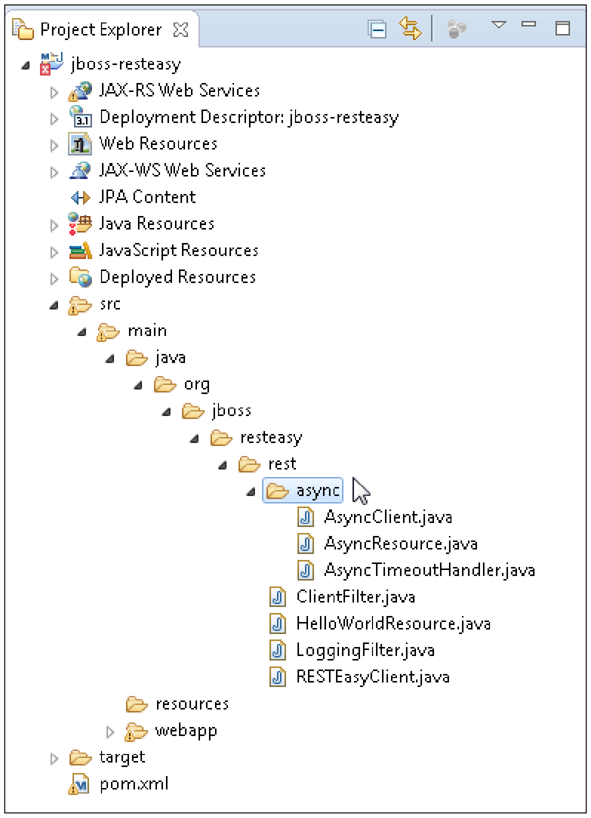
```
15:47:09,670 INFO  [org.wildfly.extension.undertow] (MSC service thread 1-1) JBA
S017535: Unregistered web context: /jboss-resteasy
15:47:10,146 INFO  [org.jboss.as.server.deployment] (MSC service thread 1-3) JBA
S015877: Stopped deployment jboss-resteasy.war (runtime-name: jboss-resteasy.war
) in 477ms
15:47:10,312 INFO  [org.jboss.as.server] (DeploymentScanner-threads - 1) JBAS018
558: Undeployed "jboss-resteasy.war" (runtime-name: "jboss-resteasy.war")
15:48:10,719 INFO  [org.jboss.as.server.deployment.scanner] (DeploymentScanner-t
hreads - 1) JBAS015009: Scan found incompletely copied file content for deployme
nt C:\wildfly-8.1.0.Final\standalone\deployments\jboss-resteasy.war. Deployment
changes will not be processed until all content is complete.
15:48:11,078 INFO  [org.jboss.as.server.deployment] (MSC service thread 1-1) JBA
S015876: Starting deployment of "jboss-resteasy.war" (runtime-name: "jboss-reste
asy.war")
15:48:13,183 INFO  [org.wildfly.extension.undertow] (MSC service thread 1-2) JBA
S017534: Registered web context: /jboss-resteasy
15:48:13,864 INFO  [org.jboss.as.server] (DeploymentScanner-threads - 2) JBAS018
559: Deployed "jboss-resteasy.war" (runtime-name : "jboss-resteasy.war")
15:49:10,181 INFO  [stdout] (default task-10) Request Method: GET
15:49:10,182 INFO  [stdout] (default task-10) Request URI: http://localhost:8080
/jboss-resteasy/rest/helloworld/text/John%20Smith
15:49:10,182 INFO  [stdout] (default task-10) Media Type : null
15:49:10,182 INFO  [stdout] (default task-10) Acceptable Media Types:
15:49:10,183 INFO  [stdout] (default task-10) text,
```

Asynchronous processing

JAX-RS 2.0 has added support for asynchronous processing in both the client API and the server API. By default, when a client sends a request to the server, it suspends all other processing till the response is received. With asynchronous processing, the client suspends connection with the server and continues to process while a server response is being generated and sent back to the client. When the response is delivered to the client, the client re-establishes a connection with the server and accepts the response. The client-server model in synchronous and asynchronous request/response is illustrated in the following diagram:



Similarly, by default a server thread blocks all other incoming client requests while waiting for an external process to complete one client request. With asynchronous processing, the server suspends connection with the client so that it may accept other client requests. When a response is available for a client request, the server re-establishes a connection with the client and sends the request. In this section, we will discuss asynchronous processing with an example. Create Java classes `AsyncResource` (for a root resource class), `AsyncClient`(for a client), and `AsyncTimeoutHandler` (for a timeout handler). The directory structure of the `async` classes is shown in **Project Explorer** as follows:



```
Project Explorer
◢ jboss-resteasy
  ▷ JAX-RS Web Services
  ▷ Deployment Descriptor: jboss-resteasy
  ▷ Web Resources
  ▷ JAX-WS Web Services
    JPA Content
  ▷ Java Resources
  ▷ JavaScript Resources
  ▷ Deployed Resources
  ◢ src
    ◢ main
      ◢ java
        ◢ org
          ◢ jboss
            ◢ resteasy
              ◢ rest
                ◢ async
                    AsyncClient.java
                    AsyncResource.java
                    AsyncTimeoutHandler.java
                  ClientFilter.java
                  HelloWorldResource.java
                  LoggingFilter.java
                  RESTEasyClient.java
        resources
      ▷ webapp
  ▷ target
    pom.xml
```

The server API has added the `javax.ws.rs.container.AsyncResponse` interface to represent an asynchronous response for server-side processing of an asynchronous response. The `javax.ws.rs.container.Suspended` interface is provided to inject a suspended `AsyncResponse` instance into a resource method parameter. The `AsyncResponse` instance is bound to an active client request and can be used to provide a response asynchronously when a response is available. When a response is to be sent to the client, the `AsyncResponse` instance resumes the suspended request.

Suspended response

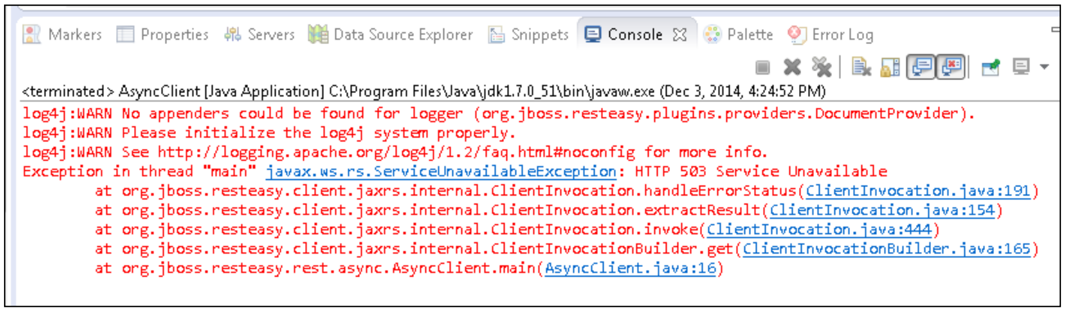
A resource or subresource method that injects a suspended `AsyncResponse` using the `@Suspended` annotation must declare the return type as void. If the injected `AsyncResponse` instance does not cancel or resume a suspended asynchronous response, the response is suspended indefinitely. In the `AsyncResource` root resource class, add a resource method (called timeout for example), which has a suspended `AsyncResponse` instance injected into a resource method parameter using the `@Suspended` annotation, as shown in the following listing. A template parameter `{timeout}` is included in the path URI for the resource method:

```
package org.jboss.resteasy.rest.async;

import javax.ws.rs.GET;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.Path;
import javax.ws.rs.container.AsyncResponse;
import javax.ws.rs.container.Suspended;


@Path("/helloworld")
public class AsyncResource {

  @GET
  @Path("/timeout/{timeout}")
  Produces("text/plain")
  public void timeout(@PathParam("timeout") String timeoutStr,@
Suspended AsyncResponse ar) {}
}
```

In the `AsyncClient`, class includes a value of `60` for the `{timeout}` template parameter in the request URI, as shown in the following listing:

```
package org.jboss.resteasy.rest.async;

import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.*;

public class AsyncClient {

  public static void main(String[] args) {

    Client client =     ClientBuilder.newClient();

    WebTarget target = client.target("http://localhost:8080/jboss-
resteasy/rest/helloworld/timeout/60");

    String response = target.request("text/plain").get(String.class);
    System.out.println("Text response: " + response);

  }

}
```

Run the `pom.xml` file to deploy the `jboss-resteasy` application. When the `AsyncClient` application is run, the server does not return a response as the asynchronous response is suspended with the following exception being returned:



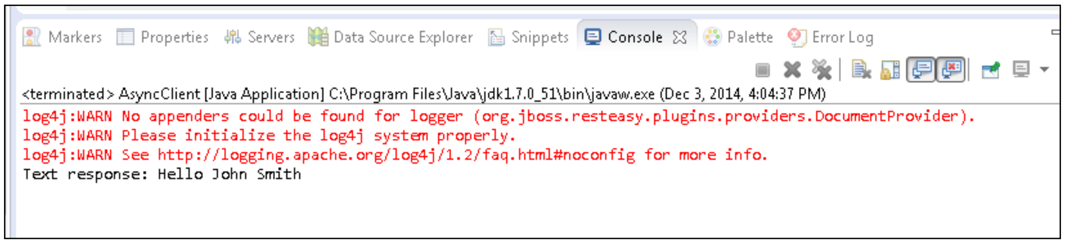
```
<terminated> AsyncClient [Java Application] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Dec 3, 2014, 4:24:52 PM)
log4j:WARN No appenders could be found for logger (org.jboss.resteasy.plugins.providers.DocumentProvider).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
Exception in thread "main" javax.ws.rs.ServiceUnavailableException: HTTP 503 Service Unavailable
	at org.jboss.resteasy.client.jaxrs.internal.ClientInvocation.handleErrorStatus(ClientInvocation.java:191)
	at org.jboss.resteasy.client.jaxrs.internal.ClientInvocation.extractResult(ClientInvocation.java:154)
	at org.jboss.resteasy.client.jaxrs.internal.ClientInvocation.invoke(ClientInvocation.java:444)
	at org.jboss.resteasy.client.jaxrs.internal.ClientInvocationBuilder.get(ClientInvocationBuilder.java:165)
	at org.jboss.resteasy.rest.async.AsyncClient.main(AsyncClient.java:16)
```

Resuming request processing

The suspended `AsyncResponse` may choose to resume the request processing, usually when a response is available, using the `resume(Object)` method. Build the response using the `ResponseBuilder` object, which may be obtained for the `Response static` method `ok(Object)`. Set the media type for the response using the `ResponseBuilder` method `type(MediaType)` and create a `Response` object using the `build()` method. Resume the suspended request processing using the `resume(Object)` method to send the `Response` object. The `AsyncResource` root resource class is listed as follows:

```
package org.jboss.resteasy.rest.async;

import javax.ws.rs.GET;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.Path;
import javax.ws.rs.container.AsyncResponse;
import javax.ws.rs.container.Suspended;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;

@Path("/helloworld")
public class AsyncResource {

  @GET
  @Path("/timeout/{timeout}")
  @Produces("text/plain")
  public void timeout(@PathParam("timeout") String timeoutStr,
  @Suspended AsyncResponse ar) {

    try {
      Response hello = Response.ok("Hello John Smith").type(MediaType.
TEXT_PLAIN).build();
      ar.resume(hello);

    } catch (Exception e) {
      System.out.println(e.getMessage());
    }
  }
}
```

The client class is the same as listed in the *Suspended response* section of this chapter. To compile and package the `jboss-resteasy` application, right-click on `pom.xml` and select **Run As** | **Maven Install**. Start the WildFly 8.1 server to deploy the application, and after the application has deployed, run the client class `AsyncClient`. Right-click on `AsyncClient.java` in **Package Explorer** and select **Run As** | **Java Application**. The client runs to produce the output, which is shown as follows:



```
<terminated> AsyncClient [Java Application] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Dec 3, 2014, 4:04:37 PM)
log4j:WARN No appenders could be found for logger (org.jboss.resteasy.plugins.providers.DocumentProvider).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
Text response: Hello John Smith
```

The `Response` object to be sent may be a `String` literal. If a `String` literal is used in the `resume(Object)` as shown here, a **Hello after a timeout** message gets generated:

```
ar.resume("Hello after a timeout");
```

Resuming a request with a suspend timeout handler

The `AsyncResponse` instance may choose to update the suspended set data to set a new suspend time-out. A new suspend time-out is set as follows using the `setTimeout(long time, TimeUnit unit)` method:

```
ar.setTimeout(timeout, TimeUnit.SECONDS);
```

The `ar` variable is the `AsyncResponse` object. The new suspended timeout value overrides the previous timeout value. At the first invocation of `setTimeout`, the suspend timeout has gone from being suspended indefinitely to being suspended for the specified timeout value. The `javax.ws.rs.container.TimeoutHandler` interface is used to provide custom resolution of timeout events. The default resolution of a timeout event is for the JAX-RS 2.0 runtime to generate a `Service unavailable` exception. Set a suspend timeout handler using the `setTimeoutHandler(TimeoutHandler handler)` method:

```
ar.setTimeoutHandler(new AsyncTimeoutHandler("Timeouted after " +
timeout + " seconds"));
```

The `AsyncResource` class to set a suspend timeout handler is listed as follows:

```
package org.jboss.resteasy.rest.async;

import java.util.concurrent.TimeUnit;

import javax.ws.rs.GET;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.Path;
import javax.ws.rs.container.AsyncResponse;
import javax.ws.rs.container.Suspended;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;

@Path("/helloworld")
public class AsyncResource {

  @GET
  @Path("/timeout/{timeout}")
  @Produces("text/plain")
  public void timeout(@PathParam("timeout") String timeoutStr,
  @Suspended AsyncResponse ar) {

    try {
      long timeout = Long.parseLong(timeoutStr);
      System.out.println("timeout - enter with timeout=" + timeoutStr
      + "s");
      ar.setTimeoutHandler(new AsyncTimeoutHandler("Timeouted after "
      + timeout + " seconds"));
      ar.setTimeout(timeout, TimeUnit.SECONDS);
    } catch (Exception e) {
      System.out.println(e.getMessage());
    }
  }
}
```

Make the `AsyncTimeoutHandler` timeout handler class implement the `TimeoutHandler` interface. In the `AsyncTimeoutHandler`, implement the `handleTimeout(AsyncResponse asyncResponse)` method in which the suspended timeout can be handled with one of the following methods:

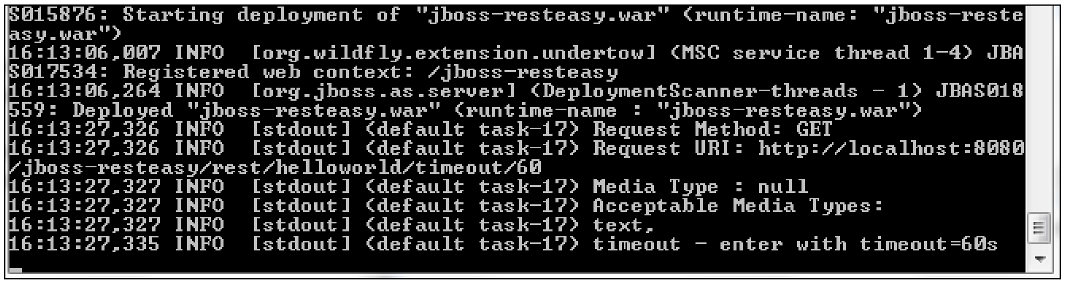
- The asynchronous response can be resumed using the `resume(Object)` method

- The response can be resumed using the `resume(Throwable)` method to throw an exception
- The response can be cancelled using the `cancel()` method
- The suspend timeout can be extended using another invocation of the `setTimeout(long time, TimeUnit unit)` method

In the `AsyncTimeoutHandler` class, resume the asynchronous response using the `resume(Object)` method to return a response to the client. The `AsyncTimeoutHandler` class is listed as follows:

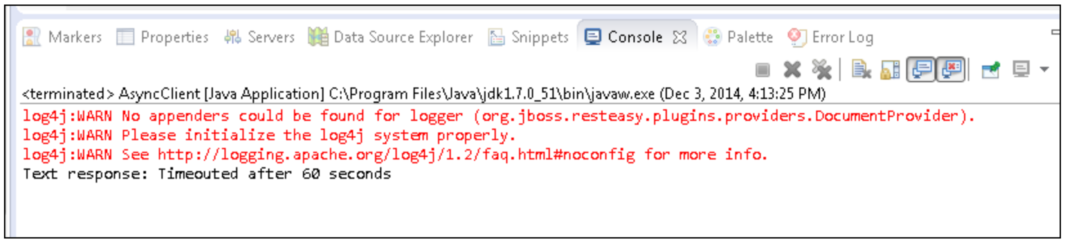
```
package org.jboss.resteasy.rest.async;
import java.util.concurrent.TimeUnit;
import javax.ws.rs.container.AsyncResponse;
import javax.ws.rs.container.TimeoutHandler;
public class AsyncTimeoutHandler implements TimeoutHandler {
  private String _message;
  boolean keepSuspended = false;
  //boolean cancel = true;
  boolean cancel = false;
  int retryPeriod = 10;
  AsyncTimeoutHandler(String message) {
    _message = message;
  }
  @Override
  public void handleTimeout(AsyncResponse ar) {
    System.out.println("handleTimeout - enter");
    if (keepSuspended) {
      ar.setTimeout(10, TimeUnit.SECONDS);
    } else if (cancel) {
      ar.cancel(retryPeriod);
    } else {
      ar.resume(_message);
    }
    /*Response hello = Response.ok("Hello after a timeout").
type(MediaType.TEXT_PLAIN).build();
    ar.resume(hello);*/
  }
}
```

Redeploy the application with Maven and rerun the `AsyncClient` class. The new suspended timeout gets applied and the response gets suspended for 60 seconds as indicated by the message `timeout- enter with timeout=60s`, which is shown as follows:



```
S015876: Starting deployment of "jboss-resteasy.war" (runtime-name: "jboss-reste
asy.war")
16:13:06,007 INFO  [org.wildfly.extension.undertow] (MSC service thread 1-4) JBA
S017534: Registered web context: /jboss-resteasy
16:13:06,264 INFO  [org.jboss.as.server] (DeploymentScanner-threads - 1) JBAS018
559: Deployed "jboss-resteasy.war" (runtime-name : "jboss-resteasy.war")
16:13:27,326 INFO  [stdout] (default task-17) Request Method: GET
16:13:27,326 INFO  [stdout] (default task-17) Request URI: http://localhost:8080
/jboss-resteasy/rest/helloworld/timeout/60
16:13:27,327 INFO  [stdout] (default task-17) Media Type : null
16:13:27,327 INFO  [stdout] (default task-17) Acceptable Media Types:
16:13:27,327 INFO  [stdout] (default task-17) text,
16:13:27,335 INFO  [stdout] (default task-17) timeout - enter with timeout=60s
```

When the request processing is resumed, the following response, as shown in the following screenshot, is sent to the client and output from the client class `AsyncClient.java`:



```
<terminated> AsyncClient [Java Application] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Dec 3, 2014, 4:13:25 PM)
log4j:WARN No appenders could be found for logger (org.jboss.resteasy.plugins.providers.DocumentProvider).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
Text response: Timeouted after 60 seconds
```

In the previous example, we resumed the request in the timeout handler. A request can be resumed in a resource method in which the new suspended timeout and the timeout handler are set before the new suspend timeout has run as shown in the resource method in the following listing:

```
@GET
@Path("/timeout/{timeout}")
@Produces("text/plain")
public void timeout(@PathParam("timeout") String timeoutStr, @
Suspended AsyncResponse ar) {
  try {
    long timeout = Long.parseLong(timeoutStr);
    System.out.println("timeout - enter with timeout=" + timeoutStr +
"s");
    ar.setTimeoutHandler(new AsyncTimeoutHandler("Timeouted after " +
timeout + " seconds"));
    ar.setTimeout(timeout, TimeUnit.SECONDS);
```

```
        Response hello = Response.ok("Hello before the suspend timeout of
    60 seconds has run").type(MediaType.TEXT_PLAIN).build();
        ar.resume(hello);
      } catch (Exception e) {
        System.out.println(e.getMessage());
      }
    }
```

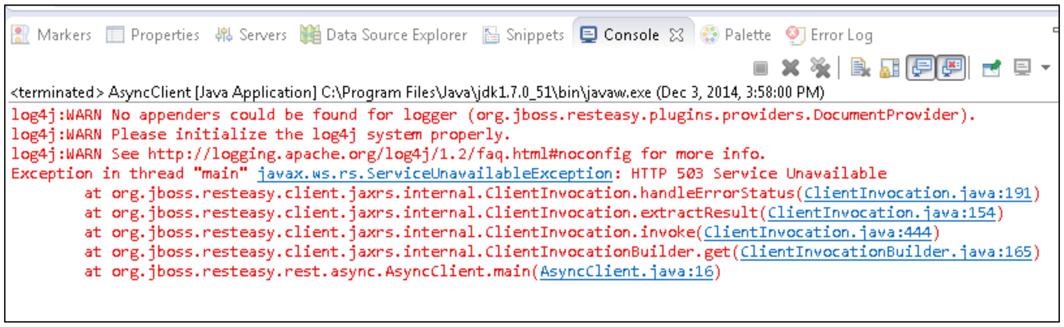
For a new suspend timeout to be applied, the request must be resumed in the timeout handler. If a new suspended timeout and a timeout handler are set and the suspended timeout handler does not take any action, the default resolution is for the request processing to be resumed with a `ServiceUnAvailableException` exception. To resume the request to send a response, the request has to be resumed explicitly using the `resume(Object)` method.

Cancelling a request

The `AsyncResponse` instance can cancel the response in the suspended timeout handler or the resource method using the overloaded `cancel()` method:

```
boolean cancel = true;
int retryPeriod = 10;
if (cancel) {
  System.out.println("Cancel the suspeneded request processing");
  ar.cancel(retryPeriod);
}
```

The client gets the following exception when a response is cancelled, which is shown as follows:



```
<terminated> AsyncClient [Java Application] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Dec 3, 2014, 3:58:00 PM)
log4j:WARN No appenders could be found for logger (org.jboss.resteasy.plugins.providers.DocumentProvider).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
Exception in thread "main" javax.ws.rs.ServiceUnavailableException: HTTP 503 Service Unavailable
	at org.jboss.resteasy.client.jaxrs.internal.ClientInvocation.handleErrorStatus(ClientInvocation.java:191)
	at org.jboss.resteasy.client.jaxrs.internal.ClientInvocation.extractResult(ClientInvocation.java:154)
	at org.jboss.resteasy.client.jaxrs.internal.ClientInvocation.invoke(ClientInvocation.java:444)
	at org.jboss.resteasy.client.jaxrs.internal.ClientInvocationBuilder.get(ClientInvocationBuilder.java:165)
	at org.jboss.resteasy.rest.async.AsyncClient.main(AsyncClient.java:16)
```

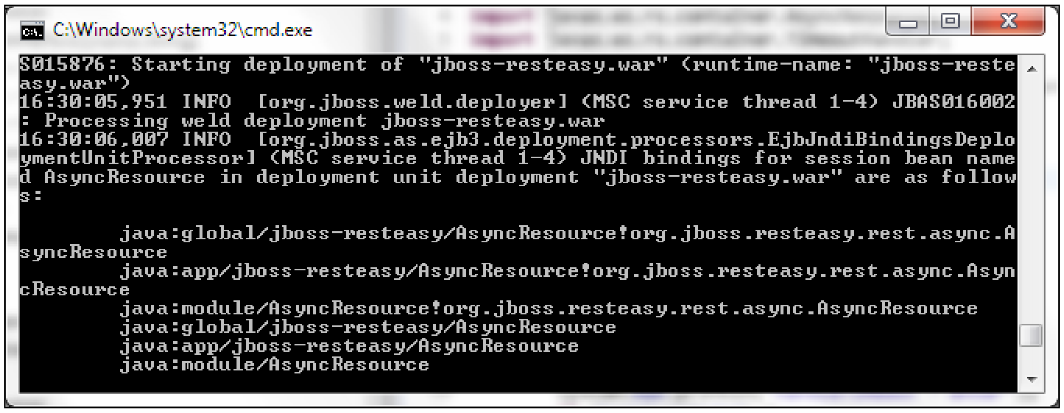
Session bean EJB resource

JAX-RS 2.0 supports stateless and singleton session beans as root resource classes. In this section, we will run the `AsyncResource` root resource class as a stateless session bean. We added an EJB-related dependency to `pom.xml`.

Annotate the `AsyncResource` with the `Stateless` annotation. The `@Path` annotation must also be applied to the class:

```
@Path("/helloworld")
@Stateless
public class AsyncResource {}
```

When the application is run, the root resource class gets added to the JNDI just as any other session bean would. The JNDI binding for `AsycnResource` is shown as follows:



C:\Windows\system32\cmd.exe

```
S015876: Starting deployment of "jboss-resteasy.war" (runtime-name: "jboss-reste
asy.war")
16:30:05,951 INFO  [org.jboss.weld.deployer] (MSC service thread 1-4) JBAS016002
: Processing weld deployment jboss-resteasy.war
16:30:06,007 INFO  [org.jboss.as.ejb3.deployment.processors.EjbJndiBindingsDeplo
ymentUnitProcessor] (MSC service thread 1-4) JNDI bindings for session bean name
d AsyncResource in deployment unit deployment "jboss-resteasy.war" are as follow
s:

        java:global/jboss-resteasy/AsyncResource!org.jboss.resteasy.rest.async.A
syncResource
        java:app/jboss-resteasy/AsyncResource!org.jboss.resteasy.rest.async.Asyn
cResource
        java:module/AsyncResource!org.jboss.resteasy.rest.async.AsyncResource
        java:global/jboss-resteasy/AsyncResource
        java:app/jboss-resteasy/AsyncResource
        java:module/AsyncResource
```

Making an asynchronous call from the client

We have as yet discussed only the asynchronous support in the Server API. The asynchronous request processing has also been made available in the client API. A client request can be sent asynchronously using the `async()` method.

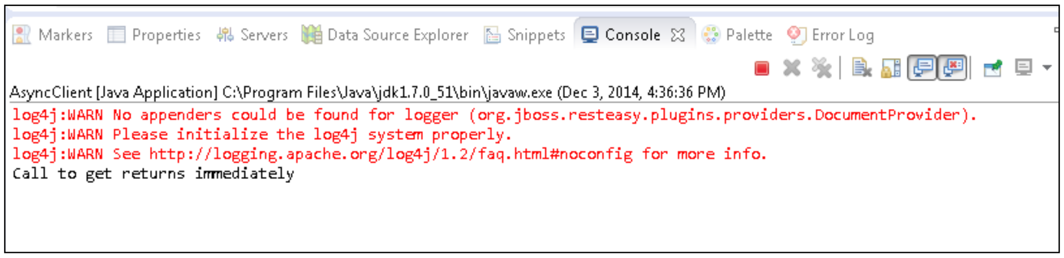
To invoke a resource asynchronously implies that the call returns immediately. Optionally, a callback can be registered using the `InvocationCallback` interface. Implement the `completed(RESPONSE response)` and `failed(Throwable throwable)` methods. The `completed(RESPONSE response)` method is called when the invocation completes successfully and the `failed(Throwable throwable)` method is called when the invocation fails. A client request is made asynchronously in the `AsyncClient.java` client as follows:

```
WebTarget target = client.target("http://localhost:8080/jboss-
resteasy/rest/helloworld/timeout/60");
AsyncInvoker asyncInvoker = target.request("text/plain").async();
asyncInvoker.get(new InvocationCallback<String>() {
  @Override
  public void completed(String response) {
    System.out.println("Invocation completed and response available");
  }
  @Override
  public void failed(Throwable arg0) {}
});
System.out.println("Call to get returns immediately");
```

Set a suspended timeout and timeout handler in the resource method and resume the request in the timeout handler as follows:

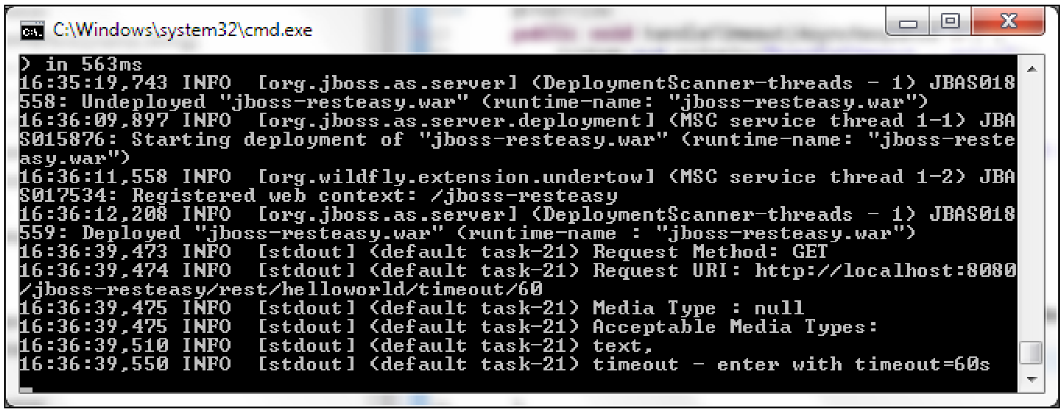
```
Response hello = Response.ok("Hello after a timeout").type(MediaType.
TEXT_PLAIN).build();
ar.resume(hello);
```

When the application is run, the call returns immediately with the message `Call to get returns immediately` (the message can get output and the processing continue without the message being noticed).



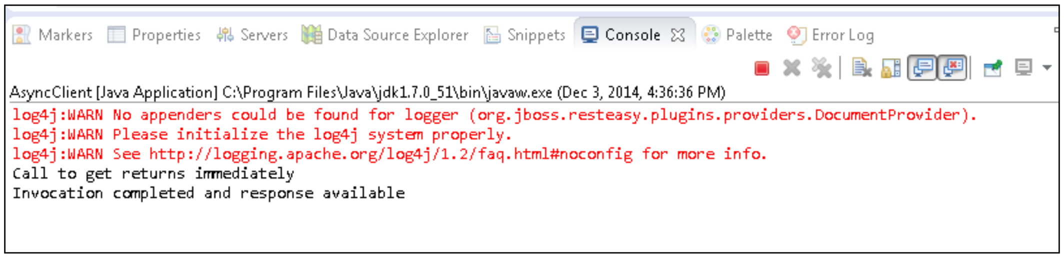
```
AsyncClient [Java Application] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Dec 3, 2014, 4:36:36 PM)
log4j:WARN No appenders could be found for logger (org.jboss.resteasy.plugins.providers.DocumentProvider).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
Call to get returns immediately
```

And the suspend timeout starts with the message. This is shown in the following screenshot:



```
C:\Windows\system32\cmd.exe
> in 563ms
16:35:19,743 INFO  [org.jboss.as.server] (DeploymentScanner-threads - 1) JBAS018
558: Undeployed "jboss-resteasy.war" (runtime-name: "jboss-resteasy.war")
16:36:09,897 INFO  [org.jboss.as.server.deployment] (MSC service thread 1-1) JBA
S015876: Starting deployment of "jboss-resteasy.war" (runtime-name: "jboss-reste
asy.war")
16:36:11,558 INFO  [org.wildfly.extension.undertow] (MSC service thread 1-2) JBA
S017534: Registered web context: /jboss-resteasy
16:36:12,208 INFO  [org.jboss.as.server] (DeploymentScanner-threads - 1) JBAS018
559: Deployed "jboss-resteasy.war" (runtime-name : "jboss-resteasy.war")
16:36:39,473 INFO  [stdout] (default task-21) Request Method: GET
16:36:39,474 INFO  [stdout] (default task-21) Request URI: http://localhost:8080
/jboss-resteasy/rest/helloworld/timeout/60
16:36:39,475 INFO  [stdout] (default task-21) Media Type : null
16:36:39,475 INFO  [stdout] (default task-21) Acceptable Media Types:
16:36:39,510 INFO  [stdout] (default task-21) text,
16:36:39,550 INFO  [stdout] (default task-21) timeout - enter with timeout=60s
```

After the suspended timeout has run, the response is returned as shown in the following screenshot. The message output when the call returns immediately is also shown here:



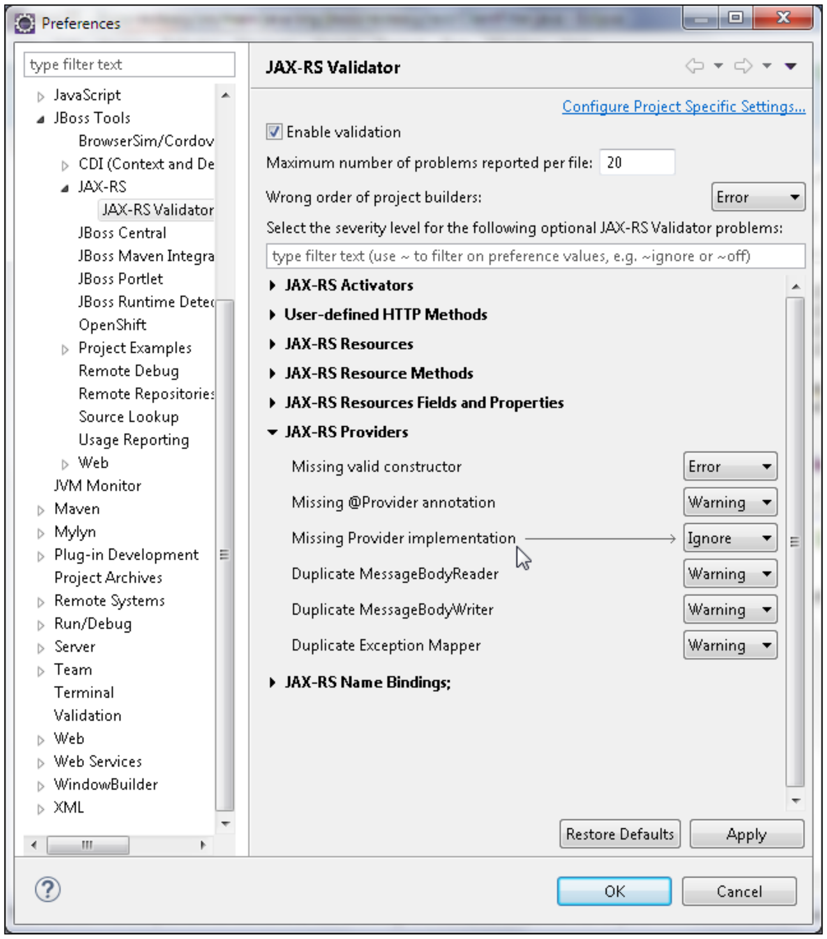
```
AsyncClient [Java Application] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Dec 3, 2014, 4:36:36 PM)
log4j:WARN No appenders could be found for logger (org.jboss.resteasy.plugins.providers.DocumentProvider).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
Call to get returns immediately
Invocation completed and response available
```

Fixing a common issue

A RESTEasy application may generate the following error:

```
The Provider must implement at least one of the following interfaces:
javax.ws.rs.ext.MessageBodyReader, javax.ws.rs.ext.MessageBodyWriter,
javax.ws.rs.ext.ExceptionMapper or javax.ws.rs.ext.ContextResolver."
```

To fix the error, select **Windows** | **Preferences**. In **Preferences,** select **JBoss Tools** | **JAX-RS** | **JAX-RS Validator**. Select **JAX-RS Providers** and set **Missing Provider implementation** to **Ignore**, and click on **Apply** and **OK**, as shown in the following screenshot:



☑ Enable validation

Maximum number of problems reported per file: 20

Wrong order of project builders: Error

Select the severity level for the following optional JAX-RS Validator problems:

type filter text (use ~ to filter on preference values, e.g. ~ignore or ~off)

- JAX-RS Activators
- User-defined HTTP Methods
- JAX-RS Resources
- JAX-RS Resource Methods
- JAX-RS Resources Fields and Properties
- JAX-RS Providers

Missing valid constructor	Error
Missing @Provider annotation	Warning
Missing Provider implementation	Ignore
Duplicate MessageBodyReader	Warning
Duplicate MessageBodyWriter	Warning
Duplicate Exception Mapper	Warning

- JAX-RS Name Bindings;

Summary

In this chapter, we discussed the salient new features in JAX RS 2.0 with an example using the RESTEasy implementation. We discussed the new client API, the filters and interceptors, asynchronous processing, cancelling a request, and using EJB as a REST Web service resource.

In the next chapter, we will discuss another new feature introduced in Java EE 7, support for JSON processing.

10
Processing JSON with Java EE 7

JavaScript Object Notation (**JSON**) is a lightweight data-interchange format that is commonly used in web services to send and receive data. Currently, Java web applications use different implementation libraries to consume/produce JSON. JSR-353 introduces the Java API for JSON processing for generating, parsing, transforming, and querying JSON. With standardized JSON API implementation, libraries are not required, which makes the applications more portable. The objective of JSR-353 is to provide JSON APIs to produce/consume streaming JSON and to build a Java object model for JSON. Support for Java API for JSON has been added to Java EE 7. In this chapter, we will discuss support for generating and parsing JSON. This chapter has the following sections:

- Setting the environment
- Creating a Maven project
- Creating JSON
- Parsing JSON
- Processing JSON in a RESTful web service

Setting up the environment

We need to download and install the following software:

- **WildFly 8.1.0.Final**: Download `wildfly-8.1.0.Final.zip` from `http://wildfly.org/downloads/`.
- **MySQL 5.6 Database-Community Edition**: Download this edition from `http://dev.mysql.com/downloads/mysql/`. When installing MySQL, also install **Connector/J**.

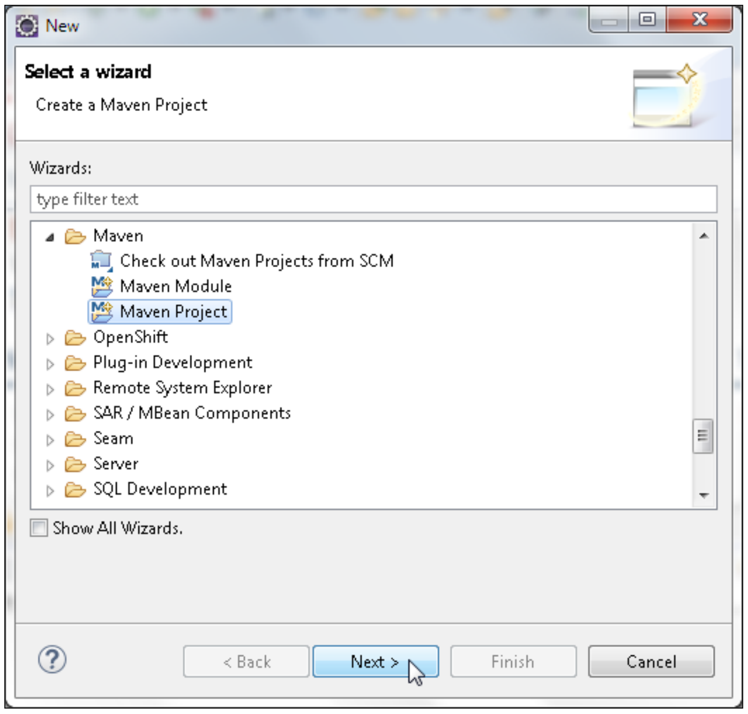
- **Eclipse IDE for Java EE Developers**: Download Eclipse Luna from `https://www.eclipse.org/downloads/packages/release/Luna/SR1`.
- **JBoss Tools (Luna) 4.2.0.Final (or the latest version)**: Install this as a plugin to Eclipse from Eclipse Marketplace (`http://tools.jboss.org/downloads/installation.html`).
- **Apache Maven**: Download version 3.05 or higher from `http://maven.apache.org/download.cgi`.
- **Java 7**: Download Java 7 from `http://www.oracle.com/technetwork/java/javase/downloads/index.html?ssSourceSiteId=ocomcn`.

Set the `JAVA_HOME`, `JBOSS_HOME`, and `MAVEN_HOME` environment variables Add `%JAVA_HOME%/bin`, `%MAVEN_HOME%/bin`, and `%JBOSS_HOME%/bin` to the `PATH` environment variable.

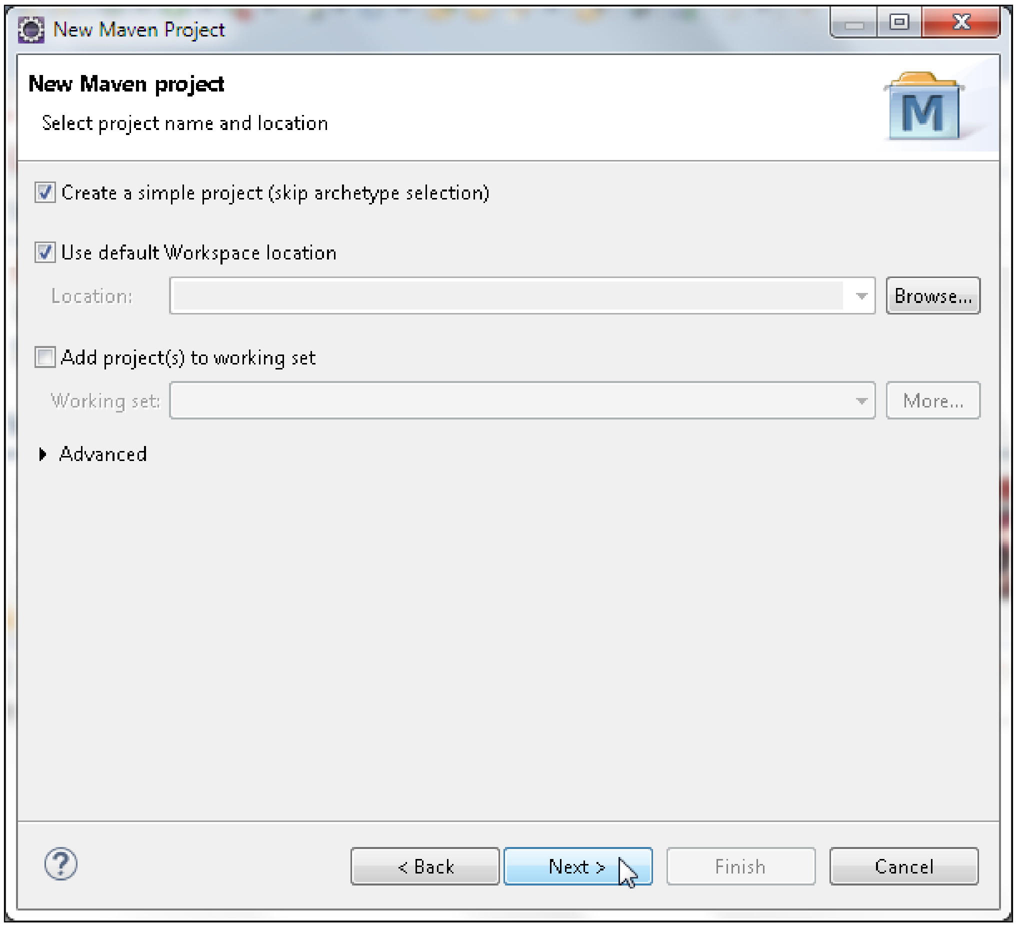
Create a WildFly 8.1.0 runtime as discussed in *Chapter 1, Getting Started with EJB 3.x*.

Creating a Maven project

Create a Maven project in the Eclipse IDE for JSON processing. Select **File** | **New** | **Other**, and in the **New** gallery, select **Maven** | **Maven Project**, as shown here:



In the **New Maven Project** wizard, select **Create a simple project** and click on **Next**, as shown in the following screenshot:



New Maven Project

New Maven project

Select project name and location

☑ Create a simple project (skip archetype selection)

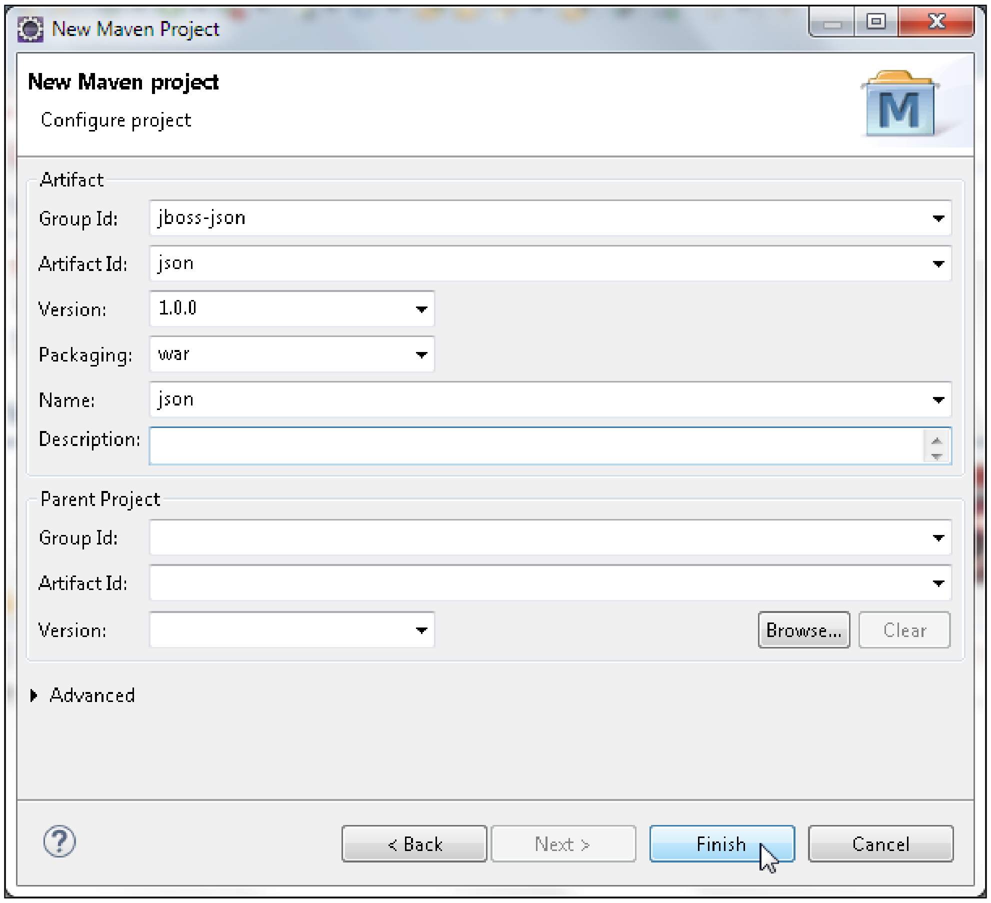
☑ Use default Workspace location

Location:

☐ Add project(s) to working set

Working set:

Specify **Group Id** (`jboss-json`), **Artifact Id** (`json`), **Version** (`1.0`), **Packaging** (`war`), and **Name** (`json`) for **New Maven Project** and click on **Finish**, as shown here:



New Maven project

Configure project

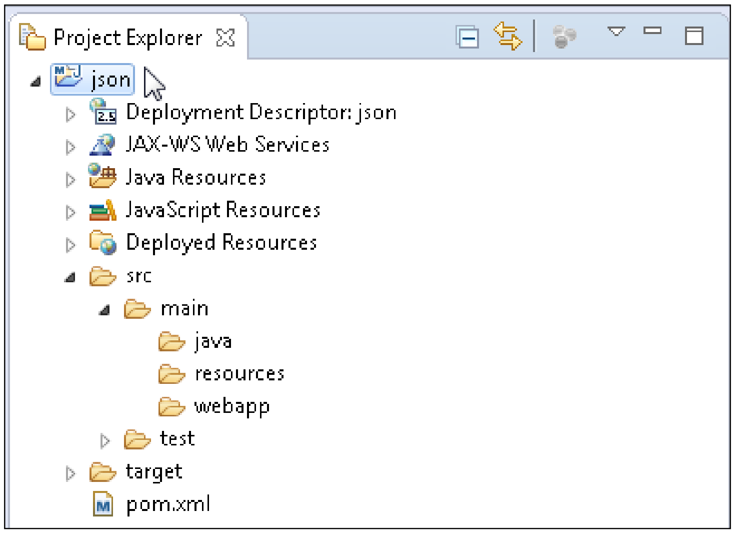
Artifact

Group Id:	jboss-json
Artifact Id:	json
Version:	1.0.0
Packaging:	war
Name:	json
Description:	

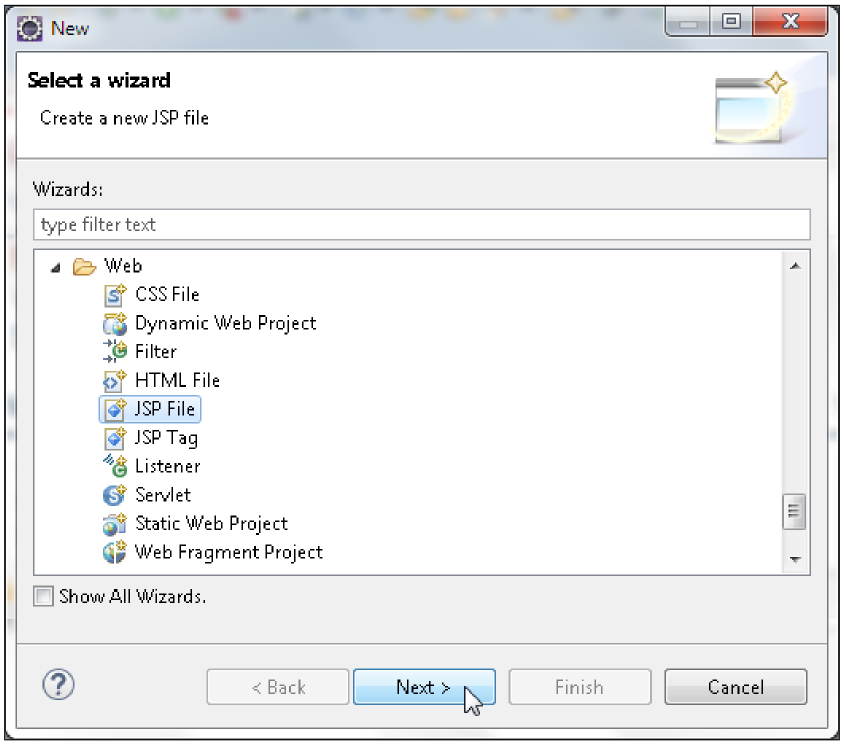
Parent Project

Group Id:	
Artifact Id:	
Version:	

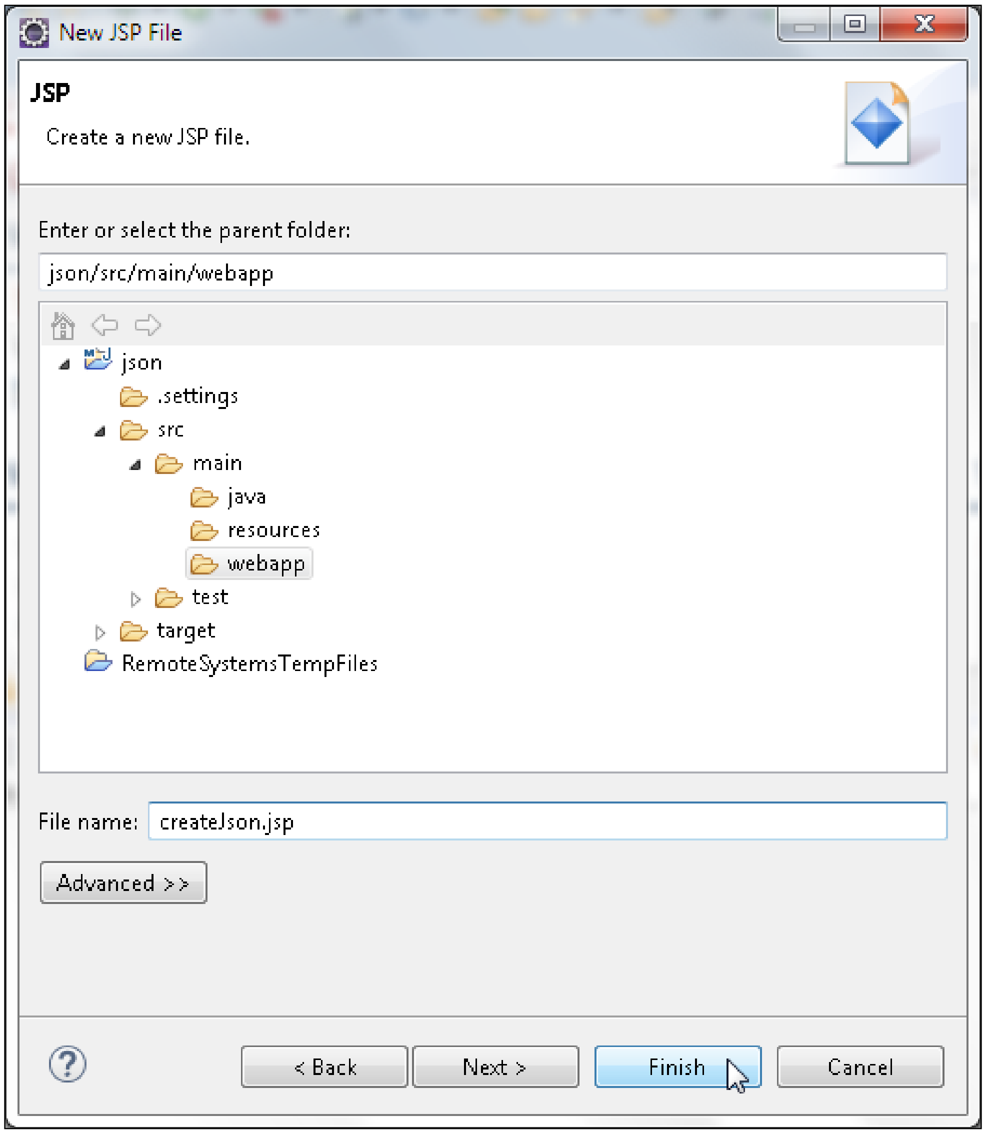
A new Maven project gets created as shown here:



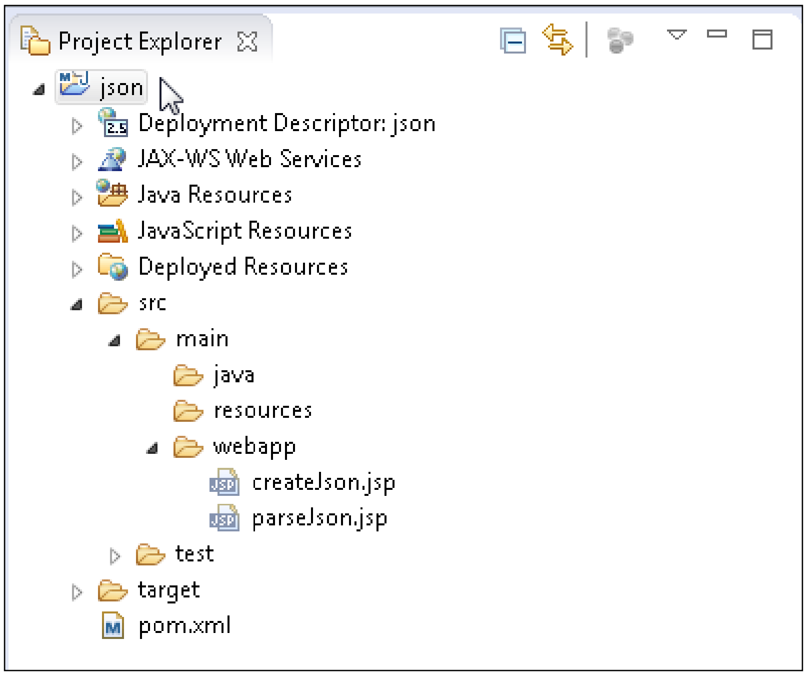
Next, add JSPs to process JSON. Add `createJson.jsp` to create JSON and add `parseJson.jsp` to parse JSON. Select **File** | **New** | **Other**, and in the **New** gallery, select **Web** | **JSP File** and click on **Next**, as shown here:



Select the `webapp` folder, specify **File name** (`createJson.jsp`), and click on **Finish**, as shown here:



The `createJson.jsp` file gets added to the Maven project. Similarly, create `parseJson.jsp`. The directory structure of the Maven web application in the Eclipse IDE is shown in the following screenshot:



Add the `jboss-json-api_1.0_spec`, `webapp-javaee7` and `javax.ws.rs-api` dependencies to `pom.xml`:

```
<dependencies>

  <dependency>
    <groupId>org.jboss.spec.javax.json</groupId>
    <artifactId>jboss-json-api_1.0_spec</artifactId>
    <version>1.0.0.Final</version>
  </dependency>

  <dependency>
    <groupId>org.codehaus.mojo.archetypes</groupId>
    <artifactId>webapp-javaee7</artifactId>
    <version>1.1</version>
  </dependency>
  <dependency>
    <groupId>javax.ws.rs</groupId>
    <artifactId>javax.ws.rs-api</artifactId>
```

```
        <version>2.0</version>
      </dependency>

    </dependencies>
```

The complete `pom.xml` file is available in the code download for this chapter.

Creating JSON

In this section, we will create a Java object model for JSON and output the JSON to a file. We will use the Java API for JSON processing to create the following JSON structure in `createJson.jsp`:

```
{
"journal":"Oracle Magazine",
"edition":
{"date":"March April 2013","cover":"Public Cloud. Private Cloud"},
"catalog":
[
{"title":"PaaS Fits the Enterprise","author":"David Baum"},
{"title":"On Becoming Others, Limits, and Restoration","author":"Tom
Kyte"}
]
}
```

A JSON Java object model is represented by the `JsonObject` class. The `JsonObjectBuilder` interface can be used to initialize a JSON object model and create a JSON object. First, create a `JsonObjectBuilder` object using `createObjectBuilder()`, the `Json` class static method, as follows:

```
JsonObjectBuilder builder = Json.createObjectBuilder();
```

`JsonObjectBuilder` provides the overloaded `add()` method to add name/value pairs of different data types to the JSON object model. Add a name/value pair for `"journal"`, as follows:

```
builder=builder.add("journal", "Oracle Magazine");
```

To create a hierarchy of JSON Java object model structures, invoke `Json.createObjectBuilder()` for each of the substructures. Add the `"edition"` JSON object model, as follows:

```
builder=builder.add("edition",
Json.createObjectBuilder().add("date","March April
2013").add("cover", "Public Cloud. Private Cloud"));
```

A JSON array can be added using the static method `createArrayBuilder()`, which returns a JSON array builder `JsonArrayBuilder` object, from the `Json` class. To build the JSON array, invoke `Json.createObjectBuilder()` for each JSON object model substructure, as follows:

```
builder=builder.add("catalog",
Json.createArrayBuilder().add(Json.createObjectBuilder().
add("title", "PaaS Fits the Enterprise").add("author","David
Baum")).add(Json.createObjectBuilder().add("title","On Becoming
Others, Limits, and Restoration").add("author","Tom Kyte")));
```

Create a `JsonObject` object from the `JsonObjectBuilder` object using the `build()` method:

```
JsonObject value = builder.build();
```

The `JsonWriter` class is used to output a JSON object or array. Create a `JsonWriter` object to output to the `jsonOutput.txt` file, as follows:

```
JsonWriter jsonWriter= Json.createWriter(new FileOutputStream(new
File("C:/json/jsonOutput.txt")));
```

Output the `JsonObject` object using the `writeObject` method, as follows:

```
jsonWriter.writeObject(value);
```

An alternative method to create `JsonObjectBuilder` is to use the `JsonBuilderFactory` interface, which is suitable if multiple instances of the `JsonObjectBuilder` object are required. A `JsonBuilderFactory` object is created using `createBuilderFactory()`, the `Json` class static method. Subsequently, invoke the `createObjectBuilder()` method of the `JsonBuilderFactory` object to create a `JsonObjectBuilder` object:

```
JsonBuilderFactory factory = Json.createBuilderFactory(null);
JsonObjectBuilder builder = factory.createObjectBuilder();
```

An alternative to creating a `JsonWriter` object is the `JsonWriterFactory` interface, which is suitable if multiple `JsonWriter` objects are required. Create a `JsonWriterFactory` object using the static method `createWriterFactory()`, and subsequently, create a `JsonWriter` object using the factory method `createWriter()`:

```
JsonWriterFactory jsonWriterFactory =
Json.createWriterFactory(null);
JsonWriter jsonWriter= jsonWriterFactory.createWriter(new
FileOutputStream(new File("C:/json/jsonOutput.txt")));
```

The `JsonGenerator` interface is provided to generate JSON in a streaming way. A `JsonGenerator` object can be created using the `Json` class static method `createGenerator()` or from a `JsonGeneratorFactory` factory, which can be created using the `Json` class static method `createGeneratorFactory()`. The two modes of creating a `JsonGenerator` object are:

```
JsonGenerator generator = Json.createGenerator(new
FileOutputStream(new File("C:/json/jsonOutput.txt")));

JsonGeneratorFactory factory = Json.createGeneratorFactory(null);
JsonGenerator generator = factory.createGenerator(new
FileOutputStream(new File("C:/json/jsonOutput.txt")));
```

`JsonGenerator` provides several methods to generate the `JsonObject` and `JsonArray` name/value pairs. The `writeStartObject()` method starts a `JsonObject` object model. The `writeStartArray()` method starts a `JsonArray` object model. The corresponding method to end a `JsonObject` or `JsonArray` object model is `writeEnd()`. The `generator` methods can be invoked in a sequence to generate a complete JSON object model:

```
generator
.writeStartObject()
.write("journal", "Oracle Magazine")
.writeStartObject("edition")
.write("date", "March April 2013")
.write("cover", "Public Cloud. Private Cloud")
         .writeEnd()
         .writeStartArray("catalog")
             .writeStartObject()
                 .write("title", "PaaS Fits the Enterprise")
                 .write("author", "David Baum")
             .writeEnd()
             .writeStartObject()
                 .write("title", "On Becoming Others, Limits, and
                 Restoration")
                 .write("author", "Tom Kyte")
             .writeEnd()
         .writeEnd()
     .writeEnd();
```

The `createJson.jsp` fileis listed as follows:

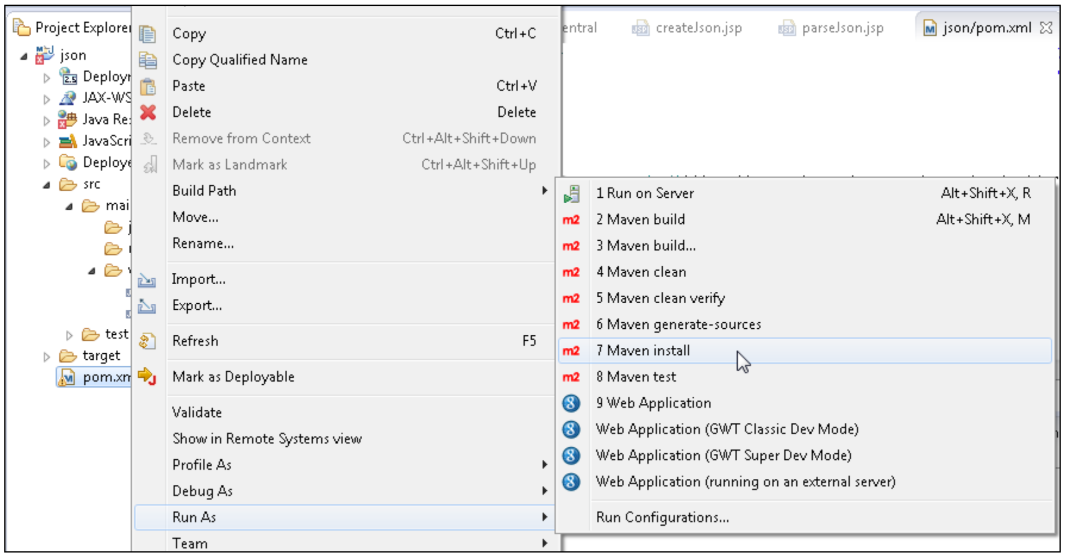
```
<%@ page language="java" contentType="text/html; charset=ISO-8859-
1"
         pageEncoding="ISO-8859-1"%>
```

```
<%@ page import="javax.json.*, java.io.*,javax.json.stream.*"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
        <title>JSON Array</title>
    </head>
    <body>
        <%
                JsonObjectBuilder builder =
                Json.createObjectBuilder();
                builder=builder.add("journal", "Oracle Magazine");

                builder=builder.add("edition",
                Json.createObjectBuilder().add("date","March April
                2013").add("cover", "Public Cloud. Private
                Cloud"));
                builder=builder.add("catalog",
                Json.createArrayBuilder()
                .add(Json.createObjectBuilder()
                .add("title","PaaS Fits the Enterprise")
                .add("author","David Baum"))
                .add(Json.createObjectBuilder()
                .add("title","On Becoming Others, Limits, and
                Restoration").add("author","Tom Kyte")));

                JsonObject value = builder.build();
                JsonWriter jsonWriter= Json.createWriter(new
                FileOutputStream(new
                File("C:/json/jsonOutput.txt")));
                jsonWriter.writeObject(value);
                jsonWriter.close();
                out.println("JSON Array output to
                jsonOutput.txt");

     /**    JsonBuilderFactory factory =
            Json.createBuilderFactory(null);
         JsonObjectBuilder builder =
         factory.createObjectBuilder();
                builder=builder.add("journal", "Oracle Magazine");

                builder=builder.add("edition",
                factory.createObjectBuilder().add("date","March
                April 2013").add("cover", "Public Cloud. Private
                Cloud"));
```

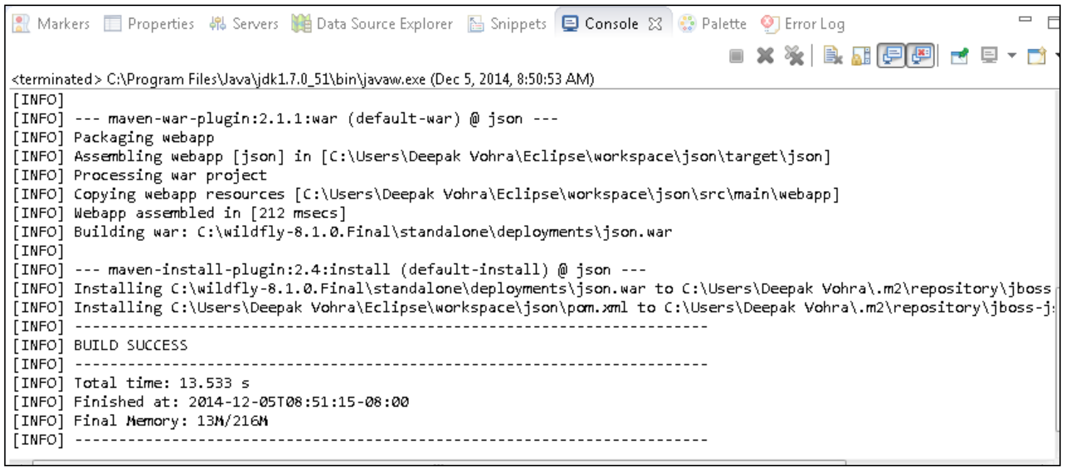
```
                  builder=builder.add("catalog",
                  factory.createArrayBuilder()
                  .add(factory.createObjectBuilder()
                  .add("title","PaaS Fits the
                  Enterprise").add("author","David Baum"))
                  .add(factory.createObjectBuilder()
                  .add("title","On Becoming Others, Limits, and
                  Restoration").add("author","Tom Kyte")));

                  JsonObject value = builder.build();
                  JsonWriterFactory jsonWriterFactory =
                  Json.createWriterFactory(null);
                  JsonWriter jsonWriter=
                  jsonWriterFactory.createWriter(new
                  FileOutputStream(new
                  File("C:/json/jsonOutput.txt")));
                  jsonWriter.writeObject(value);
                  jsonWriter.close();
                  out.println("JSON Array output to
                  jsonOutput.txt");
   */

       /**    JsonGeneratorFactory factory =
              Json.createGeneratorFactory(null);
           JsonGenerator generator = factory.createGenerator(new
           FileOutputStream(new File("C:/json/jsonOutput.txt")));

          //  JsonGenerator generator = Json.createGenerator(new
          FileOutputStream(new File("C:/json/jsonOutput.txt")));
            generator
        .writeStartObject()
            .write("journal", "Oracle Magazine")
            .writeStartObject("edition")
                .write("date", "March April 2013")
                .write("cover", "Public Cloud. Private Cloud")
            .writeEnd()
            .writeStartArray("catalog")
                .writeStartObject()
                    .write("title", "PaaS Fits the Enterprise")
                    .write("author", "David Baum")
                .writeEnd()
                .writeStartObject()
                    .write("title", "On Becoming Others, Limits, and
                    Restoration")
                    .write("author", "Tom Kyte")
                .writeEnd()
            .writeEnd()
        .writeEnd();
    generator.close();
     out.println("JSON Array output to jsonOutput.txt");*/
```

```
            %>
        </body>
    </html>
```

To run `createJson.jsp`, first build and install the Maven project. Right-click on the project node in **Package Explorer** and select **Run As** | **Maven install**, as shown in the following screenshot. All the JSPs in the chapter can be built together; for retesting, the Maven project would need to be rebuilt.

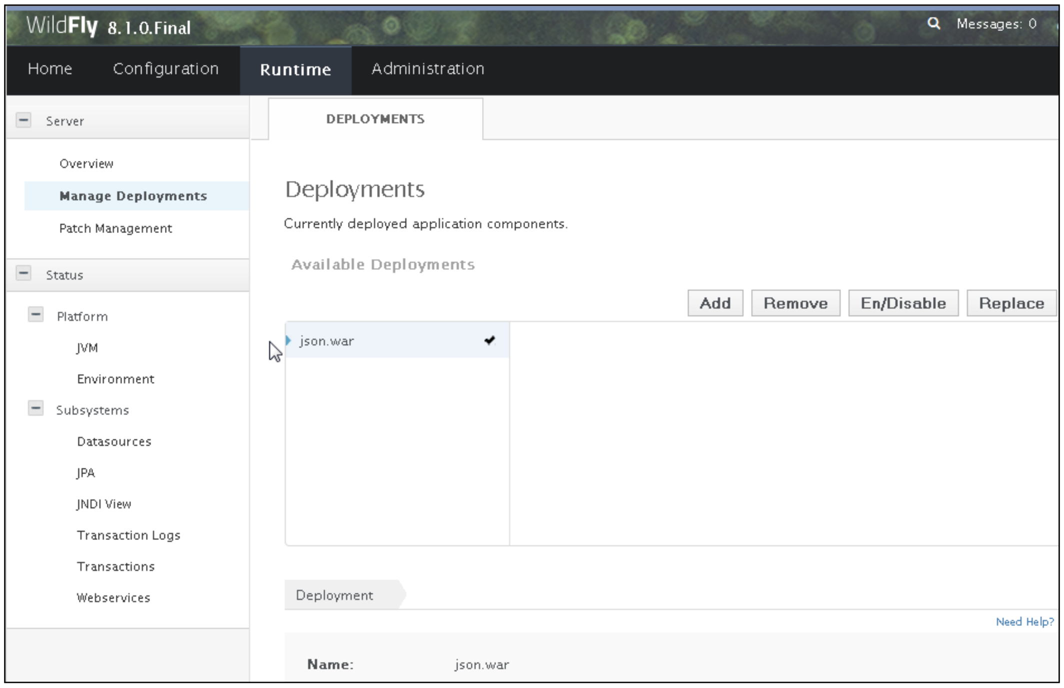


The JSON web application gets deployed to WildFly 8.1. The Maven build outputs the message `BUILD SUCCESS`, as shown here:



```
<terminated> C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Dec 5, 2014, 8:50:53 AM)
[INFO]
[INFO] --- maven-war-plugin:2.1.1:war (default-war) @ json ---
[INFO] Packaging webapp
[INFO] Assembling webapp [json] in [C:\Users\Deepak Vohra\Eclipse\workspace\json\target\json]
[INFO] Processing war project
[INFO] Copying webapp resources [C:\Users\Deepak Vohra\Eclipse\workspace\json\src\main\webapp]
[INFO] Webapp assembled in [212 msecs]
[INFO] Building war: C:\wildfly-8.1.0.Final\standalone\deployments\json.war
[INFO]
[INFO] --- maven-install-plugin:2.4:install (default-install) @ json ---
[INFO] Installing C:\wildfly-8.1.0.Final\standalone\deployments\json.war to C:\Users\Deepak Vohra\.m2\repository\jboss
[INFO] Installing C:\Users\Deepak Vohra\Eclipse\workspace\json\pom.xml to C:\Users\Deepak Vohra\.m2\repository\jboss-j
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 13.533 s
[INFO] Finished at: 2014-12-05T08:51:15-08:00
[INFO] Final Memory: 13M/216M
[INFO] ------------------------------------------------------------------------
```

Start WildFly **Administration Console** with the URL `http://localhost:8080`. Now, click on **Administration Console.** Specify the login **Name** and **Password** to log in to **Administration Console.** Click on **Manage Deployments**. The `json.war` file is listed and deployed, as shown here:

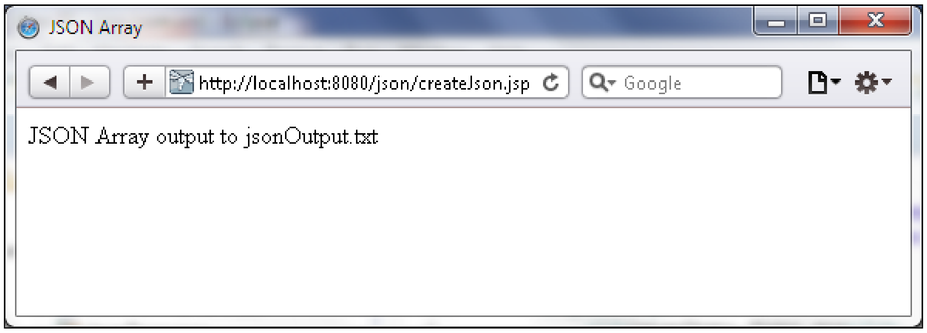


Currently deployed application components.

json.war ✔

Name: json.war

Run `createJson.jsp` on the WildFly server with the URL `http://localhost:8080/json-1.0/createJson.jsp`, as shown in the following screenshot. The `createJson.jsp` yes runs on the server to output the JSON Java object model to `C:/json/jsonOutput.txt file`.



JSON Array output to jsonOutput.txt

The JSON output is shown here:

```
{"journal":"Oracle Magazine","edition":{"date":"March April
2013","cover":"Public Cloud. Private
Cloud"},"catalog":[{"title":"PaaS Fits the
Enterprise","author":"David Baum"},{"title":"On Becoming Others,
Limits, and Restoration","author":"Tom Kyte"}]}
```

Parsing JSON

In this section, we will parse the `jsonOutput.txt` file generated in the previous section. The `JsonParser` interface is provided to parse JSON in a streaming way. Create `JsonParser` using the static method `createParser()` from the `Json` class, as follows:

```
JsonParser parser=Json.createParser(new FileInputStream(new
File("C:/json/jsonOutput.txt")));
```

An alternative to creating a `JsonParser` object is using a `JsonParserFactory` factory, which can be created using the static method `createParserFactory()`, as shown in the following lines of code:

```
JsonParserFactory factory = Json.createParserFactory(null);
JsonParser parser = factory.createParser(new FileInputStream(new
File("C:/json/jsonOutput.txt")));
```

A `JsonParser` object can be used to parse a JSON string using `StringReader`. First, create `StringReader` for JSON. Subsequently, create a `JsonParser` object using the `Json` class static method `createParser(Reader)`, as follows:

```
StringReader reader = new StringReader("{\"journal\":\"Oracle
Magazine\",\"edition\":{\"date\":\"March April
2013\",\"cover\":\"Public Cloud. Private
Cloud\"},\"catalog\":[{\"title\":\"PaaS Fits the
Enterprise\",\"author\":\"David Baum\"},{\"title\":\"On Becoming
Others, Limits, and Restoration\",\"author\":\"Tom Kyte\"}]}");
JsonParser parser = Json.createParser(reader);
```

`JsonParser` parses JSON using the pull parsing model, in which the client pulls (calls) the next parsing event using the `next()` method. The following parsing events are generated:

Parsing Event	Description
`START_OBJECT`	Start of a JSON object
`END_OBJECT`	End of a JSON object
`START_ARRAY`	Start of a JSON array
`END_ARRAY`	End of a JON array

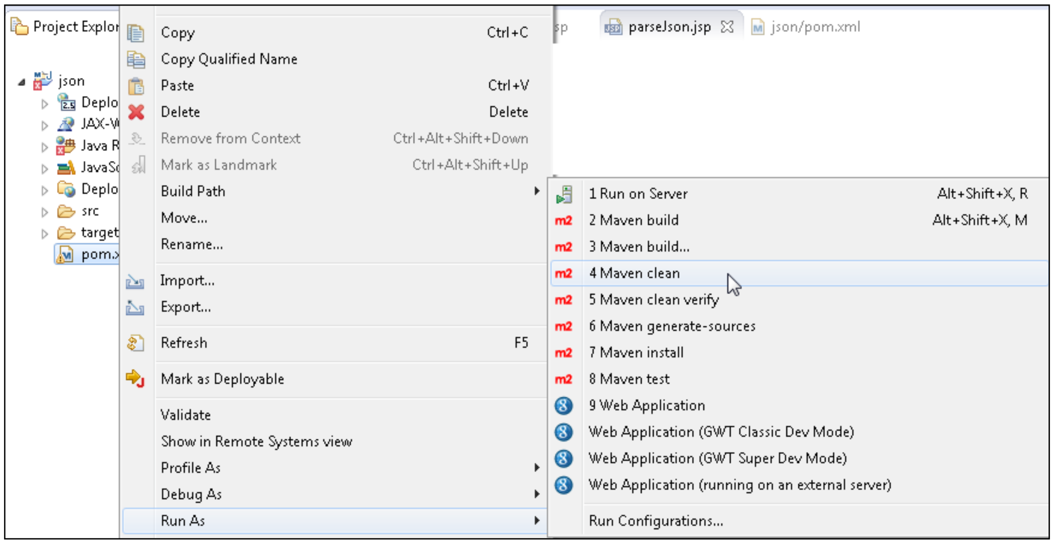
Parsing Event	**Description**
`KEY_NAME`	Name in a key/value pair of a JSON object
`VALUE_STRING`	String value in a JSON object or array
`VALUE_NUMBER`	Number value in a JSON object or array
`VALUE_TRUE`	Boolean value `true` in a JSON object or array
`VALUE_FALSE`	Boolean value `false`
`VALUE_NULL`	Null value in a JSON object or array

Use the `hasNext()` method to find if a next parsing event is available, and use the `next()` method to obtain the next event. Use a `switch` statement to output the event name and the name/value pairs:

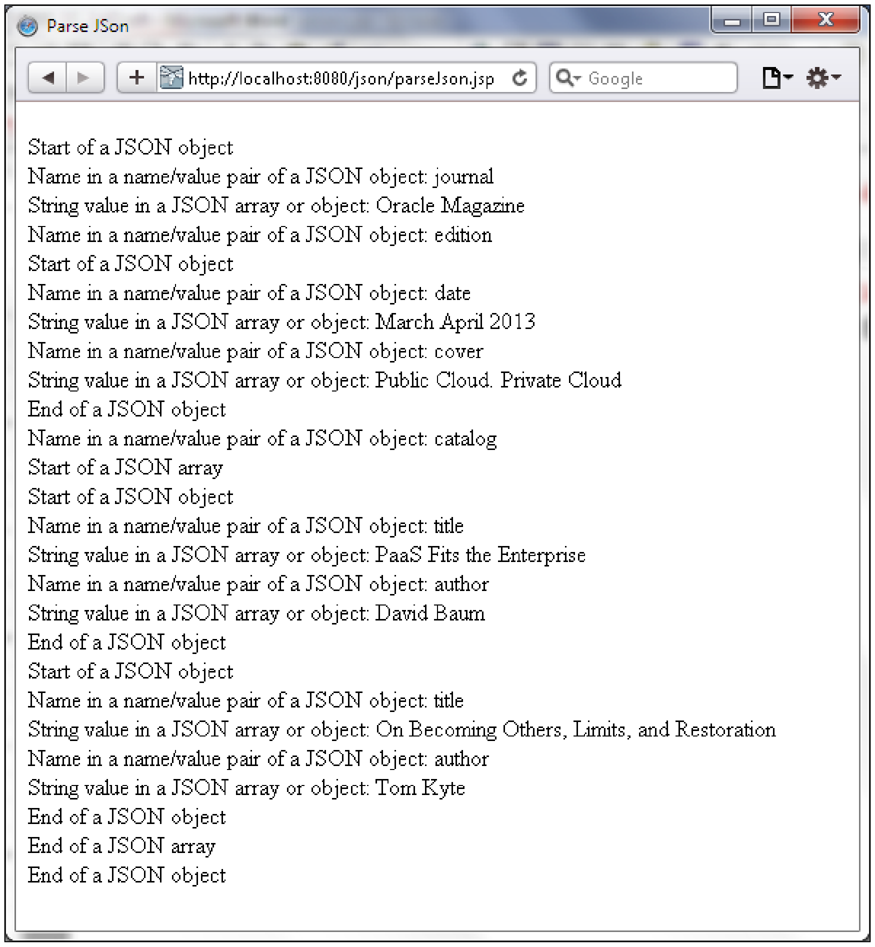
```
while(parser.hasNext()){
        JsonParser.Event parsingState= parser.next();
        switch(parsingState){
}
}
```

The `parseJson.jsp` is available in the code download for this chapter.

Some sections in the code listing have been commented out to test the different approaches to parsing. To deploy and run `parseJson.jsp`, we need to rebuild the Maven project. First, clean the Maven project. Right-click on the `json` project node in Eclipse and select **Run As** | **Maven clean**, as shown here:



The `BUILD SUCCESS` message indicates that the files generated from the previous build have been deleted. Invoke `parseJson.jsp` in a browser with the URL `http://localhost:8080/json-1.0/parseJson.jsp` to output the event names and key/value pairs in the JSON structure, as shown here:



```
Start of a JSON object
Name in a name/value pair of a JSON object: journal
String value in a JSON array or object: Oracle Magazine
Name in a name/value pair of a JSON object: edition
Start of a JSON object
Name in a name/value pair of a JSON object: date
String value in a JSON array or object: March April 2013
Name in a name/value pair of a JSON object: cover
String value in a JSON array or object: Public Cloud. Private Cloud
End of a JSON object
Name in a name/value pair of a JSON object: catalog
Start of a JSON array
Start of a JSON object
Name in a name/value pair of a JSON object: title
String value in a JSON array or object: PaaS Fits the Enterprise
Name in a name/value pair of a JSON object: author
String value in a JSON array or object: David Baum
End of a JSON object
Start of a JSON object
Name in a name/value pair of a JSON object: title
String value in a JSON array or object: On Becoming Others, Limits, and Restoration
Name in a name/value pair of a JSON object: author
String value in a JSON array or object: Tom Kyte
End of a JSON object
End of a JSON array
End of a JSON object
```

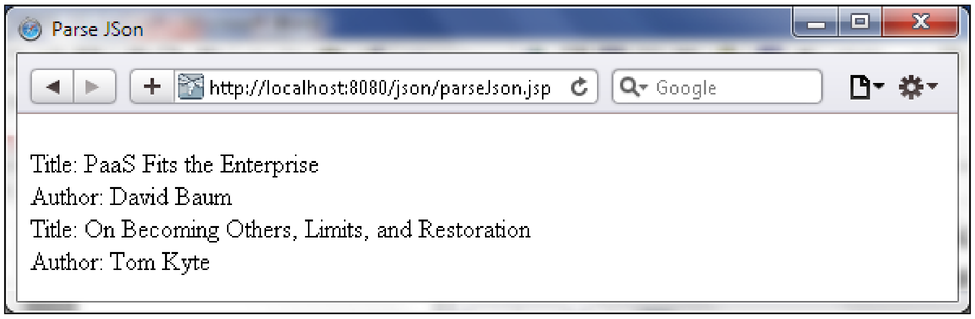
An alternative interface for parsing a JSON structure is the `JsonReader` interface, which reads a JSON object or an array from an input source. `JsonReader` can be created from `InputStream` or `Reader`. A `JsonReader` interface can be created using the `Json` class static method `createReader()` or using the `JsonReaderFactory` factory, which is created with the `Json` class static method `createReaderFactory()`. Here's how we accomplish this:

```
JsonReaderFactory factory = Json.createReaderFactory(null);
JsonReader jsonReader = factory.createReader(new
StringReader("[{\"title\":\"PaaS Fits the
Enterprise\",\"author\":\"David Baum\"},{\"title\":\"On Becoming
Others, Limits, and Restoration\",\"author\":\"Tom Kyte\"}]"));
```

Alternatively, a `JsonReader` interface can be created using the `Json` class static method `createReader()`. A JSON array (`JsonArray` object) is obtained from a `JsonReader` interface using the `readArray()` method, and a JSON object (`JsonObject` object) is obtained using the `readObject()` method. Subsequently, the `get()` methods of `JsonObject` and `JsonArray` can be used to obtain key/value pairs. For example, a JSON array is read from `StringReader`, and the name/value pairs in the array output as follows:

```
JsonReader jsonReader = Json.createReader(new
StringReader("[{\"title\":\"PaaS Fits the
Enterprise\",\"author\":\"David Baum\"},{\"title\":\"On Becoming
Others, Limits, and Restoration\",\"author\":\"Tom Kyte\"}]"));
 JsonArray array = jsonReader.readArray();
jsonReader.close();
JsonObject catalog = array.getJsonObject(0);
 out.println("Title: "+catalog.getString("title"));
   out.println("Author: "+catalog.getString("author"));
   JsonObject catalog2 = array.getJsonObject(1);
 out.println("Title: "+catalog2.getString("title"));
 out.println("Author: "+catalog2.getString("author"));
```

Now, rerun `parseJson.jsp` to output the name/value pairs in a JSON array, as shown here:



```
Title: PaaS Fits the Enterprise
Author: David Baum
Title: On Becoming Others, Limits, and Restoration
Author: Tom Kyte
```

Processing JSON in a RESTful web service

The JSON format is commonly used in RESTful web services to exchange messages. In this section, we will discuss how the Java API or JSON processing is used in a RESTful web service. First, add the RESTful web service dependency to `pom.xml`:

```
<dependency>
  <groupId>javax.ws.rs</groupId>
  <artifactId>javax.ws.rs-api</artifactId>
  <version>2.0</version>
</dependency>
```

Create a sample REST web service to test the JSON API. Create a resource class `JsonResource` annotated with `@PATH` to identify the URI path. Here's how we accomplish this:

```
package org.json;

import java.io.StringReader;
import javax.ws.rs.GET;
import javax.ws.rs.Produces;
import javax.ws.rs.Path;

import javax.json.*;
import javax.ws.rs.Consumes;
import javax.ws.rs.POST;
import javax.ws.rs.QueryParam;
import javax.ws.rs.core.MediaType;

@Path("jaxrs")
public class JsonResource {
}
```

To package the resource class, create a class that extends the `javax.ws.rs.core.Application` class. Annotate the `Application` subclass with the `@ApplicationPath` annotation to define the base URI pattern for the resource. Override the `getClasses()` method to return the list of RESTful web service resources. Add the `org.json.JsonResource` class to `HashSet` to return from the `getClasses()` method. The `JsonResourceApplication` class extends `Application` subclass, as listed here:

```
package org.json;

import java.util.HashSet;
import java.util.Set;
import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;

@ApplicationPath("resources")
public class JsonResourceApplication extends Application {

    @Override
    public Set<Class<?>> getClasses() {
        Set<Class<?>> set = new HashSet<Class<?>>();
        set.add(org.json.JsonResource.class);

        return set;
    }

}
```

JsonArray as a return type in the resource method for a GET request

Java EE 7 provides support for the `JsonObject`/`JsonArray`/`JsonStructure` API in a RESTful web service. `JsonObject`/`JsonArray`/`JsonStructure` can be used as return type and parameter type in a resource method. In this section, we use `JsonArray` as the return type of a resource method. Add a resource method (`getJsonMessage()`) in the `JsonResource` class that produces `application/json` and accepts `GET` requests. In the resource method, create `JsonReader` from `StringReader`, and create `JsonArray` from `JsonReader`, which was also discussed in the previous section. Return `JsonArray` from the resource method listed here:

```
    @GET
    @Produces({MediaType.APPLICATION_JSON})
    public JsonArray getJsonMessage() {
        JsonReader jsonReader = Json.createReader(new
        StringReader("[{\"title\":\"PaaS Fits the
        Enterprise\",\"author\":\"David Baum\"},{\"title\":\"On
        Becoming Others, Limits, and
        Restoration\",\"author\":\"Tom Kyte\"}]"));

        JsonArray array = jsonReader.readArray();
        jsonReader.close();
        return array;

    }
```

Create the `jaxrsGetJsonReturnType.jsp` JSP. In this JSP, use the new `Client` API introduced in JAX-RS 2.0 to create `Client` and invoke the RESTful web service. Create a `Client` object from the `ClientBuilder` static method `newClient()`. A `Client` object is used to send requests and receive responses from a RESTful web service. In the fluent `Client` API method, invocations can be linked. Invoke the `target(String)` method to build a new web resource target.

Build the resource path using the `path()` method. Add `"resources"` to the path for the `Application` subclass. Add `"jaxrs"` to the path to access the resource class `JsonResource`. As the `getJsonMessage()` method is not annotated with `@PATH`, we won't add a path component for the resource method. Build the request with the `request()` method, and specify the acceptable response type as `application/json` (or `MediaType.APPLICATION_JSON`). Invoke the `get()` method for the request and specify the Java type of the response entity as `JsonArray.class`:

```
Client client = ClientBuilder.newClient();
JsonArray array = client.target("http://localhost:8080/json-
1.0").path("resources").path("jaxrs").request(MediaType.APPLICATION_
JSON).get(JsonArray.class);
```

Obtain `JsonObject` in `JsonArray`, which is returned from the REST web service using the `getJsonObject(int index)` method. For example, the name/value pairs in the first `JsonObject` in `JsonArray` are output as follows:

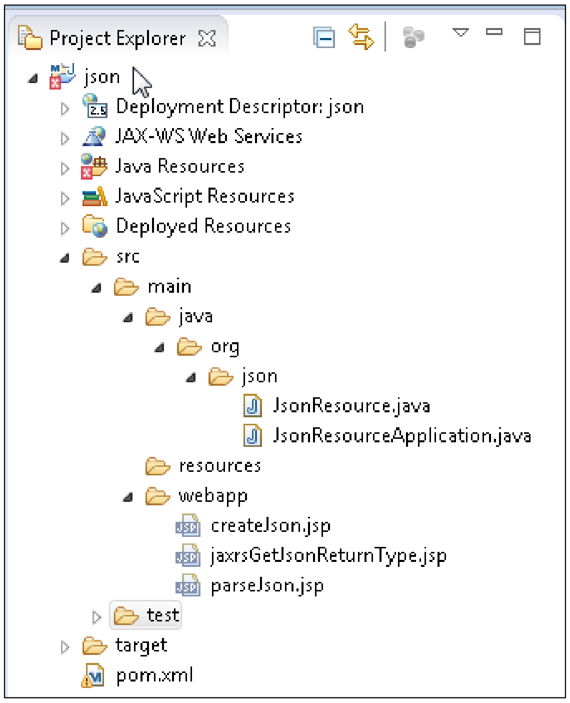
```
JsonObject catalog = array.getJsonObject(0);
out.println("Title: "+catalog.getString("title"));
out.println("Author: "+catalog.getString("author"));
```

The `jaxrsGetJsonReturnType.jsp` file is listed here:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-
1"
         pageEncoding="ISO-8859-1"%>
<%@ page import="javax.ws.rs.client.ClientBuilder,javax.ws.rs.client.
Client
,javax.json.*,javax.json.stream.*,javax.ws.rs.core.*"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html;
        charset=ISO-8859-1">
        <title>JSON Array</title>
    </head>
    <body>
        <%
```

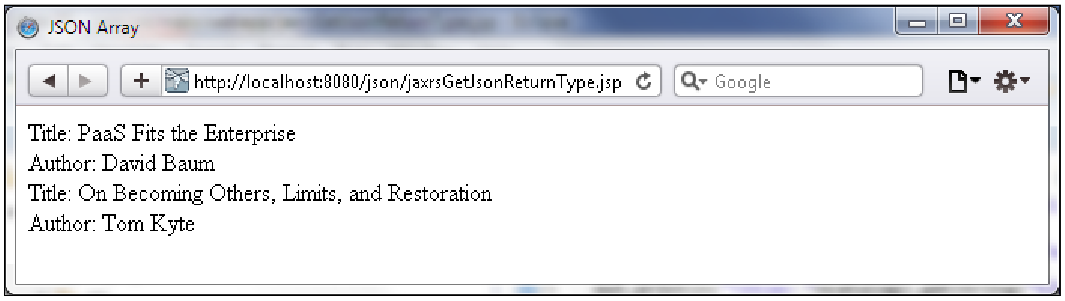
```
 Client client = ClientBuilder.newClient();
JsonArray array =
client.target("http://localhost:8080/json").
path("resources").path ("jaxrs").request(MediaType.APPLICATION_JSON).
get(JsonArray.class);

 JsonObject catalog = array.getJsonObject(0);

  out.println("Title: "+catalog.getString("title"));
   out.println("<br/>");
   out.println("Author: "+catalog.getString("author"));
   out.println("<br/>");

   JsonObject catalog2 = array.getJsonObject(1);

  out.println("Title: "+catalog2.getString("title"));
   out.println("<br/>");
   out.println("Author: "+catalog2.getString("author"));
   out.println("<br/>");

        %>
    </body>
</html>
```

The directory structure of the `json` application is shown here:



```
json
  Deployment Descriptor: json
  JAX-WS Web Services
  Java Resources
  JavaScript Resources
  Deployed Resources
  src
    main
      java
        org
          json
            JsonResource.java
            JsonResourceApplication.java
      resources
      webapp
        createJson.jsp
        jaxrsGetJsonReturnType.jsp
        parseJson.jsp
    test
  target
  pom.xml
```

Before deploying the application, run **Maven clean** to delete the deployment classes from the previous Maven build. Remove the `json.war` file from the `deployments` directory of the WildFly server. Run **Maven install** as explained earlier to deploy the JSON application to WildFly. Start WildFly if it is not already started. Run `jaxrsGetJsonReturnType.jsp` with the `http://localhost:8080/json/jaxrsGetJsonReturnType.jsp` URL in a browser to invoke the RESTful web service, and output the name/value pairs in `JsonArray`, which is returned from the web service, as shown here:



```
Title: PaaS Fits the Enterprise
Author: David Baum
Title: On Becoming Others, Limits, and Restoration
Author: Tom Kyte
```

Encoding a JsonObject object for a GET request

`JsonObject/JsonArray` can be used as a resource method parameter type, but it has a limitation; a `JsonObject/JsonArray` type cannot not be included in a `GET` request because the `GET` request URI must not contain special characters (such as `{`,`}`, `[`, and `]`) that are used in JSON. A JSON object or array must be UTF-8 encoded to `String` for the request URI and, therefore, the parameter type of a resource method cannot be `JsonObject/JsonArray` without using a provider to convert from string to `JsonObject/JsonArray` (an approach we discuss in a later section). In this section, we encode `JsonObject` as `String` and send a request to a resource method with a `@QueryParam` annotated parameter of the type `String`.

In the resource class `JsonResource`, add a resource method that has an annotated parameter, `@QueryParam`, of the type `String`. In the resource method, create `JsonObject` from `String` by first creating `JsonReader` using `StringReader` and subsequently obtaining `JsonObject` with the `readObject()` method. Obtain the `"catalog"` JSON array from `JsonObject` using the `getJsonArray()` method and return the `JsonArray` object. Add the following resource method to the `JsonResource` class:

```
@Path("jsonp")
@GET
@Produces({MediaType.APPLICATION_JSON})
```

```
        public JsonArray getJsonArray(@QueryParam("jsonObject") String
        jsonObjectStr) {
            JsonReader jsonReader = Json.createReader(new
            StringReader(jsonObjectStr));
            JsonObject jsonObject = jsonReader.readObject();
            jsonReader.close();
            JsonArray jsonArray = jsonObject.getJsonArray("catalog");
            return jsonArray;


        }
```

Add the `jaxrsGetJsonStringMethodParam.jsp` JSP. In this JSP, build a `JsonObject` object, as discussed in an earlier section *Creating JSON*. Encode `JsonObject` to a UTF-8 `String` using the `URLEncoder.encode(String,String)` static method, as follows:

```
String jsonObjectStr =  URLEncoder.encode(jsonObject.toString(),
"UTF-8");
```

Build the client request using the fluent `Client` API. Create a `Client` object from `ClientBuilder` using the static method `newClient()`. Build the web resource target using `http://localhost:8080/json` as the base URI. Add URI paths for the `Application` subclass (`path ("resources")`), resource class (`path ("jaxrs")`), and resource method (`path ("jsonp")`). Set the `jsonObject` query parameter value to the encoded `jsonObjectStr` string using the `queryParam()` method. Build the request using the request method and invoke the request to get a response of the `JsonArray.class` Java type using the `get` method:

```
Client client = ClientBuilder.newClient();
JsonArray array =
client.target("http://localhost:8080/json").path("resources").path
("jaxrs").path("jsonp").queryParam("jsonObject",jsonObjectStr).req
uest(MediaType.APPLICATION_JSON).get(JsonArray.class);
```

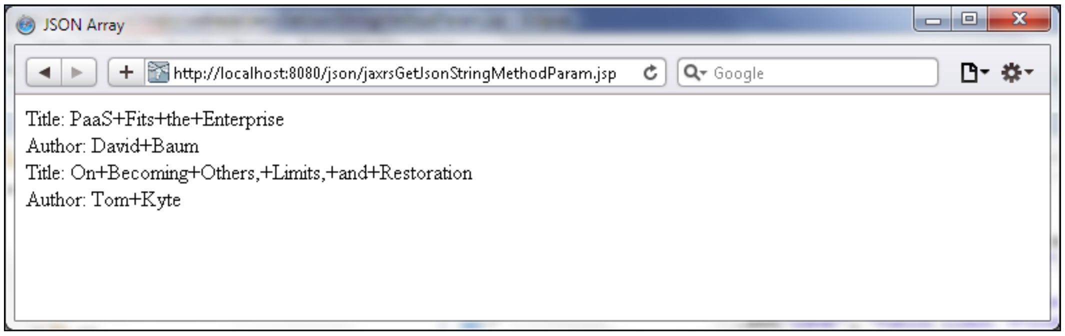
Output the name/value pairs in the JSON array, as discussed in the previous section. The `jaxrsGetJsonStringMethodParam.jsp` is listed here:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
  pageEncoding="ISO-8859-1"%>
<%@ page
  import="java.net.URLEncoder,java.io.StringReader,
  javax.ws.rs.client.ClientBuilder,
  javax.ws.rs.client.Client,javax.json.*,
  javax.json.stream.*,javax.ws.rs.core.*"%>
```

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html;
    charset=ISO-8859-1">
    <title>JSON Array</title>
  </head>
  <body>
    <%
      JsonObjectBuilder builder = Json.createObjectBuilder();
      builder = builder.add("journal", "Oracle Magazine");
      builder = builder.add("edition",
      Json.createObjectBuilder().add("date", "March April
      2013")
      .add("cover", "Public Cloud. Private Cloud"));
      builder = builder
      .add("catalog",
      Json.createArrayBuilder()
      .add(Json
      .createObjectBuilder()
      .add("title",
      "PaaS Fits the Enterprise")
      .add("author", "David Baum"))
      .add(Json
      .createObjectBuilder()
      .add("title",
      "On Becoming Others, Limits, and Restoration")
      .add("author", "Tom Kyte")));

      JsonObject jsonObject = builder.build();
      String jsonObjectStr =
      URLEncoder.encode(jsonObject.toString(),
      "UTF-8");
      Client client = ClientBuilder.newClient();
      JsonArray array =
      client.target("http://localhost:8080/json")
      .path("resources").path("jaxrs").path("jsonp")
      .queryParam("jsonObject", jsonObjectStr)
      .request(MediaType.APPLICATION_JSON).get(JsonArray.class);
      JsonObject catalog = array.getJsonObject(0);

      out.println("Title: " + catalog.getString("title"));
      out.println("<br/>");
```

```
            out.println("Author: " + catalog.getString("author"));
            out.println("<br/>");

            JsonObject catalog2 = array.getJsonObject(1);

            out.println("Title: " + catalog2.getString("title"));
            out.println("<br/>");
            out.println("Author: " + catalog2.getString("author"));
            out.println("<br/>");
            %>
        </body>
    </html>
```

Run `jaxrsGetJsonStringMethodParam.jsp` with the URL `http://localhost:8080/json/jaxrsGetJsonStringMethodParam.jsp` in a browser to output JSON array name/value pairs, as shown in the next screenshot. Because of the UTF-8 encoding of `JsonObject` to `String`, the whitespace is replaced with a `+` in the `String` values output:


JSON Array
http://localhost:8080/json/jaxrsGetJsonStringMethodParam.jsp
Google

```
Title: PaaS+Fits+the+Enterprise
Author: David+Baum
Title: On+Becoming+Others,+Limits,+and+Restoration
Author: Tom+Kyte
```

JsonObject as a parameter type in the resource method for a POST request

We couldn't send the JSON object in a `GET` request without encoding it into `String`, because a `GET` request includes the key/value pairs in the request URI. A JSON object in the request URI would generate an `org.apache.jasper.JasperException: javax.ws.rs.core.UriBuilderException: java.net.URISyntaxException` exception.

A JSON object can be sent in a `POST` request as the key/value pairs are sent in the request itself and not the request URI. In this section, we will send a `POST` request to a resource method that has a `JsonObject` type parameter. In the resource method, obtain the `"catalog"` JsonArray from `JsonObject` and return the `JsonArray` object. Add a resource method annotated with `@POST` and with a `JsonObject` parameter, as follows:

```
@Path("post")
@POST
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public JsonArray post(final JsonObject jsonObject) {
     JsonArray jsonArray = jsonObject.getJsonArray("catalog");

        return jsonArray;
}
```

Create the `jaxrsPostJsonMethodParam.jsp` JSP, which creates a `JsonObject` object as in the previous subsections on *Processing JSON in a RESTful web service*. Create a `Client` object using the fluent `Client` API. Build the web resource path by including the path URI for the post resource method. Build a `POST` request invocation using the `buildPost(Entity<?> entity)` method. Create an `Entity<JsonObject>` object using the static method `json(T entity)` in `javax.ws.rs.client.Entity<T>`. Invoke the request using the `invoke(Class<T> responseType)` method with the response type `JsonArray.class`:

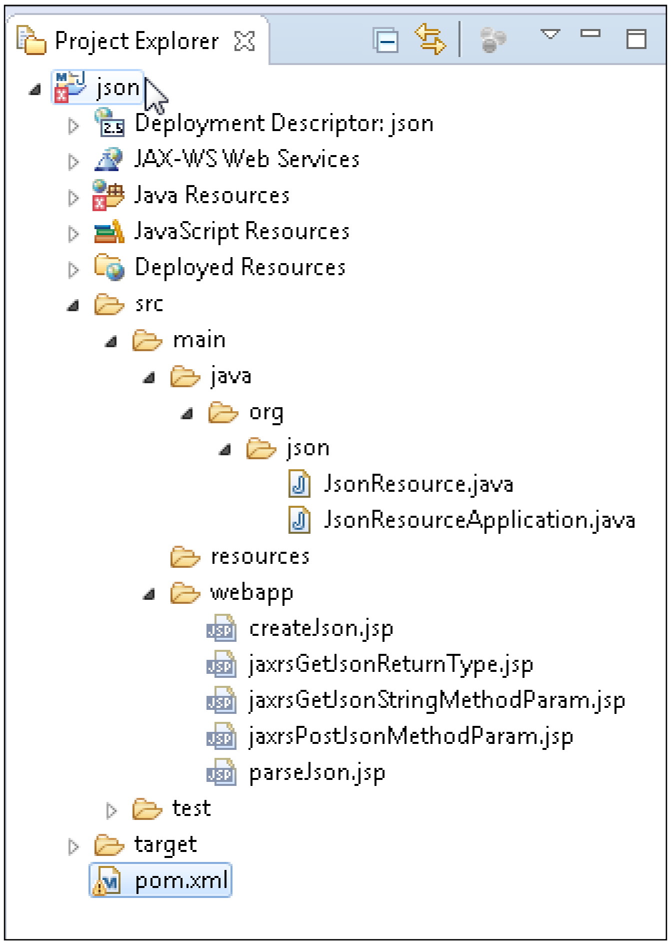
```
JsonObject jsonObject = builder.build();
Client client = ClientBuilder.newClient();
JsonArray array =
client.target("http://localhost:8080/json").
path("resources").path("jaxrs").path("post").
request(MediaType.APPLICATION_JSON).
buildPost(Entity.json(jsonObject)).invoke(JsonArray.class);
```

Output the name/value pairs in the JSON array. The `jaxrsPostJsonMethodParam.jsp` file is listed here:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-
1"
         pageEncoding="ISO-8859-1"%>
<%@ page import="javax.ws.rs.client.*,java.io.StringReader,
javax.json.*,javax.json.stream.*,
javax.ws.rs.core.*"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
```

```
            <meta http-equiv="Content-Type" content="text/html;
            charset=ISO-8859-1">
            <title>JSON Array</title>
        </head>
        <body>
            <%
              JsonObjectBuilder builder = Json.createObjectBuilder();
                    builder=builder.add("journal", "Oracle Magazine");
                    builder=builder.add("edition",
                    Json.createObjectBuilder().add("date","March April
                    2013").add("cover", "Public Cloud. Private
                    Cloud"));
                    builder=builder.add("catalog",
                    Json.createArrayBuilder().
                    add(Json.createObjectBuilder().
                    add("title","PaaS Fits the
                    Enterprise").add("author","David
                    Baum")).add(Json.createObjectBuilder().
                    add("title","On Becoming Others, Limits, and
                    Restoration").add("author","Tom Kyte")));

                    JsonObject jsonObject = builder.build();
                    Client client = ClientBuilder.newClient();
    JsonArray array =
    client.target("http://localhost:8080/json").path("resources").path
    ("jaxrs").path("post").request(MediaType.APPLICATION_JSON).buildPo
    st(Entity.json(jsonObject)).invoke(JsonArray.class);
      JsonObject catalog=array.getJsonObject(0);
      out.println("Title: "+catalog.getString("title"));
      out.println("<br/>");
      out.println("Author: "+catalog.getString("author"));
      out.println("<br/>");
      JsonObject catalog2 = array.getJsonObject(1);
      out.println("Title: "+catalog2.getString("title"));
      out.println("<br/>");
      out.println("Author: "+catalog2.getString("author"));
      out.println("<br/>");

            %>
        </body>
    </html>
```

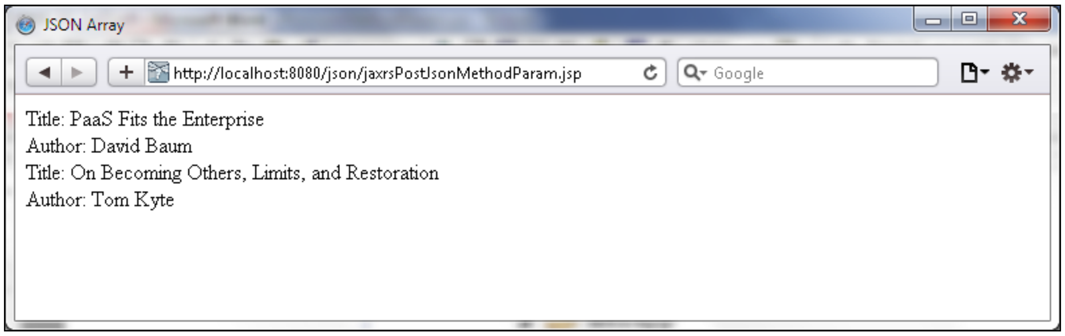
The directory structure of the `json` application is shown in **Package Explorer** in the following screenshot:



Project Explorer

```
json
  Deployment Descriptor: json
  JAX-WS Web Services
  Java Resources
  JavaScript Resources
  Deployed Resources
  src
    main
      java
        org
          json
            JsonResource.java
            JsonResourceApplication.java
      resources
      webapp
        createJson.jsp
        jaxrsGetJsonReturnType.jsp
        jaxrsGetJsonStringMethodParam.jsp
        jaxrsPostJsonMethodParam.jsp
        parseJson.jsp
    test
  target
  pom.xml
```

Run **Maven clean** to delete previously generated deployment files, and run **Maven install** to redeploy the `json` application. Then, run `jaxrsPostJsonMethodParam.jsp` in the `http://localhost:8080/json/jaxrsPostJsonMethodParam.jsp` URL to output name/value pairs in the JSON array, as shown in next screenshot. As we did not encode the JSON object sent with the request, the `JSON array` string values have whitespace instead of `+`.



```
Title: PaaS Fits the Enterprise
Author: David Baum
Title: On Becoming Others, Limits, and Restoration
Author: Tom Kyte
```

Summary

In this chapter, we discussed the Java API for JSON processing in Java EE 7 to create a JSON object, create a JSON array, and parse a JSON object/array. Processing JSON in a RESTful web service was also discussed in this chapter. This chapter concludes the book We explored the commonly used Java EE technologies as used with WildFly and Maven in Eclipse. We also discussed the salient new technologies in Java EE 7, support for JAX-RS 2.0, and support for processing JSON.

Index

Symbol

@WebService annotation
  about 218
  elements 218

A

Ajax
  about 123
  environment, setting 123, 124
Ajax application
  deploying, with Maven 147-153
  running 154-158
Apache Maven
  URL 82
artifacts, Hibernate 71
asynchronous call, JAX-RS 2.0
  making, from client 355, 356
Asynchronous JavaScript and XML. *See* **Ajax**
asynchronous processing, JAX-RS 2.0
  about 345-347
  request, cancelling 354
  request processing, resuming 349, 350
  request, resuming with suspend timeout handler 350-353
  suspended response 347, 348
AsyncResource root resource class 349
AsyncTimeoutHandler class 352

B

BasicTemplate.xhtml Facelet template 102
BOM (Bill of Materials) 34

C

Catalog.java entity class
  URL 18
CDI (Context and Dependency Injection) API 114
Client API, JAX-RS 2.0
  about 329
  client instance, creating 329
  query parameter, setting 335
  resource, accessing 330-334
  template parameter, setting 335
client filter
  about 336
  creating 337-341
ClientFilter class 338
Common Annotations API 114
container filter
  about 336, 342
  creating 342-344
CRUD
  JSPs, creating for 56-58

D

Data Access Object design pattern, Spring MVC application
  creating 288-290
data source
  configuring, with MySQL database 10-13
DDL script 62
Dependency Injection 275
deployment structure descriptor, Spring MVC application
  creating 308-311
Document Object Model (DOM) 123

E

EAR module
deploying 30-33
Eclipse
GWT project, creating 167-171
Eclipse IDE for Java EE Developers
URL 82
EJB 3.x
environment, setting up 2, 3
objective 1
EJB (Enterprise JavaBeans) API 114
elements, @WebService annotation
endpointInterface 219
portName 218
serviceName 219
targetNamespace 219
entities
creating 14-18
entry-point class, GWT
creating 187-195
event handlers 190
events 190
panels 190
widgets 189
Errai 199

F

Facelets
about 81, 88
URL 88
Facelets application
configuration file 88
Facelets composition pages 88
Facelets header and footer pages 88
Facelets template page 88
managed bean 88
running 120
Facelets template
about 97
creating 97-102
filters, JAX-RS 2.0
about 336
client filter 336
ClientRequestFilter 337
ClientResponseFilter 337
container filter 336
ContainerRequestFilter 337
ContainerResponseFilter 337

G

Google Web Toolkit (GWT) 161
GWT module
creating 183-186
entry-point class, creating 187-192
entry-point class application name 183
source path, for GWT project 184
GWT project
creating 177-183
creating, in Eclipse 167-171
deploying, with Maven 199-202
environment, setting up 162-165
HTML host page, creating 195-198
running 203-207
starter project, deploying to
WildFly 8.1 172-175
starter project, running on
WildFly 8.1 165, 166, 176

H

header and footer JSF pages
creating 103-105
HelloWorldResource class 331
Hibernate
about 37
artifacts 71
URL , for properties 19
Hibernate Annotations API 26
Hibernate configuration file
advantages 52
creating 51-54
Hibernate Validator API 114
Hibernate web application
environment, setting 38
Hibernate XML Mapping file
creating 42-47
properties file, creating 48-51
HQL commands
URL 65
HTML host page, GWT
creating 195-198

HTTP methods
URL 247

I

infrastructure deployment descriptor, Spring MVC application
creating 308-311
input and output Facelets composition pages
creating 106-109
interceptors, JAX-RS 2.0 336
Inversion of Control (IoC) principle 275

J

Java 7
URL 82
JavaBean class
creating 58
Java Database Connectivity (JDBC) 37
Java EE 7 Maven project
configuring 34, 35
Java EE design patterns
URL 20
Java EE project, EJB 3.x
creating 6-9
Java EE web project
creating, for Ajax 125-128
creating, for Hibernate 39-42
creating, for JAX-RS RESTful Web Service 241-244
creating, for JAX-WS Web Service 211-214
creating, for JSF 2.x Facelets 84-88
deploying, with Maven 114-119
user interface, creating for Ajax 129-138
Java Persistence API (JPA) 1, 114
JavaScript Object Notation. *See* **JSON**
JavaServer Faces (JSF) 2.x 81
JAX-RS 2.0
about 317
asynchronous call, making from client 355, 356
asynchronous processing 345-347
Client API 329
common issue, resolving 357
environment, setting 318
filters 336
interceptors 336
Java EE web project, creating 318-328
session bean EJB resource 355
JAX-RS API 114
JAX-RS RESTful Web Service
creating 244-252
deploying 252-261
environment, setting 240, 241
Java client, creating 263-265
Java client, running 266-268
JSP client, creating 269-271
JSP client, running 272
running 261-263
JAX-WS
about 209
application, deploying with Maven 228-236
application, running 238
JAX-WS Maven Plugin
about 228
URL 228
JAX-WS Web Service
about 209
components 209
creating 217-222
environment, setting up 210
web descriptor, creating 215, 216
jboss-ejb3-ear subproject
configuring 28, 29
jboss-ejb3-ejb subproject
configuring 25-27
jboss-ejb3-web subproject
configuring 27, 28
JBoss Tools (Luna) 4.2.0.Final
URL 82
JNDI contexts 24
JNDI (Java Naming and Directory Interface) 24
JPA configuration file, Spring MVC application
creating 281
JPA persistence configuration file
creating 19, 20
JSF 2.2 81
JSF 2.x Facelets
environment, setting 82, 83
JSF Managed Bean Facility 89

JSON
about 359
creating 366-372
environment, setting 359, 360
parsing 373-376
JSON, processing in RESTful Web Service
about 377
JsonArray, as return type in resource method for GET request 378-381
JsonObject, as parameter type in resource method for POST request 384-388
JsonObject, encoding for GET request 381-384
JSP client
creating 22-25
running 34
JSPs
creating, for CRUD 56, 57
JSP Web Service client
creating 224-227
JSR 311
URL 317
JSR 339
URL 317

L

LoggingFilter.java class 343

M

managed bean
about 89
creating 89-96
Maven
Ajax application, deploying with 147-153
GWT project, deploying with 199-202
Java EE web project, deploying with 114
JAX-WS application, deploying with 228-236
Spring MVC application, deploying with 300-308
Maven project
creating, for JSON 360-365
installing 70-73
Model, Spring MVC application
creating 282-285
Model-View-Controller design pattern 275
MySQL 5.6 Database-Community Edition
URL 82
MySQL database
data source, configuring with 10-13
URL 10

P

Plain Old Java Object (POJO) 1, 239

R

Representational State Transfer (REST) 317
request handler, Spring MVC application
creating 292-294
RESTEasy
URL 317
RESTEasyClient.java class 330
RESTful principles 239, 240

S

schema
exporting 61, 62
schemaExport.jsp
running 73-75
servlet, Ajax application
creating 139-147
session bean EJB resource 355
session bean facade
creating 20, 21
Spring MVC application
context, creating 299, 300
creating 277-280
Data Access Object design pattern, creating 288-290
deploying, with Maven 300-308
deployment structure descriptor, creating 308-311
environment, setting 276
infrastructure deployment descriptor, creating 308-311
JPA configuration file, creating 281

Model, creating 282-286
request handler, creating 292
running 314, 315
View, creating 295, 296
web descriptor, creating 291
Spring MVC Maven project
installing 312-314
starter project, GWT
deploying, to WildFly 8.1 172-175
running, on WildFly 8.1 165, 166, 176

T

table
data, creating 63, 64
data, retrieving 65
row, creating 76
row, deleting 68-78
row, retrieving 76
row, updating 67
updating 77, 78

U

URIs (Uniform Resource Identifiers) 317

V

View Declaration Language (VDL) 81
View, Spring MVC application
creating 295, 296

W

W3C standards
URL 209
web descriptor
about 110
creating 111-113
creating, for JAX-WS Web Service 215, 216
creating, for Spring MVC application 291
Web Service Description Language (WSDL)
URL 209
WildFly 8.1.0.Final
URL 82
WildFly runtime
creating 3-5