

Wildfly

Java® EE Development with WildFly®

Sundar

Alchemy

Disclaimer

WildFly is a registered trademark of Red Hat, Inc.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Table of Contents

Preface	v
Module1: 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
Module2: 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

[i]

Table of Contents

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
Module3: 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
Module4: 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
Module5: 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

[ii]

Table of Contents

Deploying the GWT project with Maven	199
Running the GWT project	203
Summary	208
Module6: 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
Module7: 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
Module8: 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

Table of Contents

Module9: 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
Cancelled a request	354
Session bean EJB resource	355
Making an asynchronous call from the client	355
Fixing a common issue	357
Summary	358
Module10: 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

[v]

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

What this book covers

In *Module1, 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 *Module2, 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 *Module3, 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 *Module4, 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 *Module5, 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 *Module6, 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 *Module7, 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 *Module8, 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 *Module9, 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

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 *Module10, 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 Program

We have used WildFly 8.1.0 in the program. 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 Program 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 Program.

Conventions

In this Program, 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**."

[x]

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 Module 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. We will cover 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

[1]

- 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:

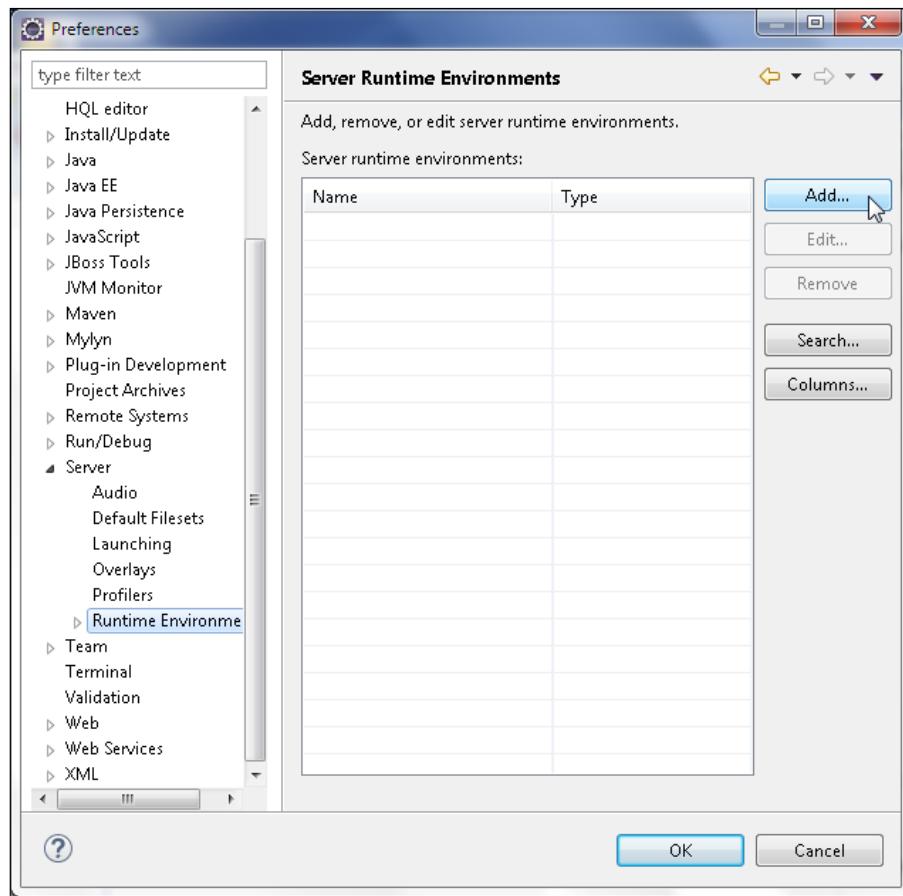
- **WildFly 8.1.0.Final:** Download `wildfly-8.1.0.Final.zip` from <http://wildfly.org/downloads/>.
- **MySQL 5.6 Database-Community Edition:** Download this edition from <http://dev.mysql.com/downloads/mysql/>. When installing MySQL, also install **Connector/J**.
- **Eclipse IDE for Java EE Developers:** 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).

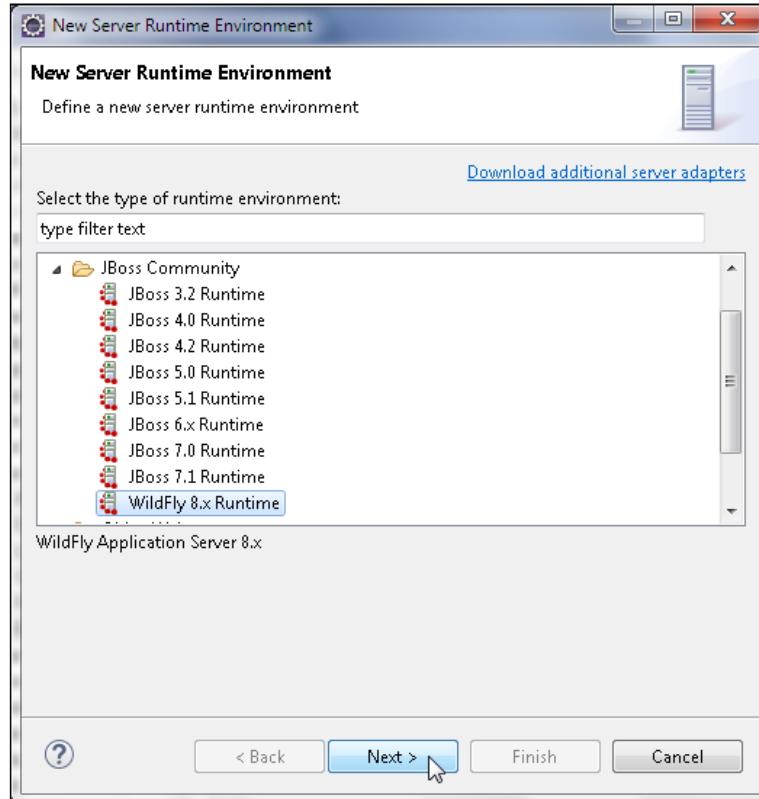
Creating a WildFly runtime

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:

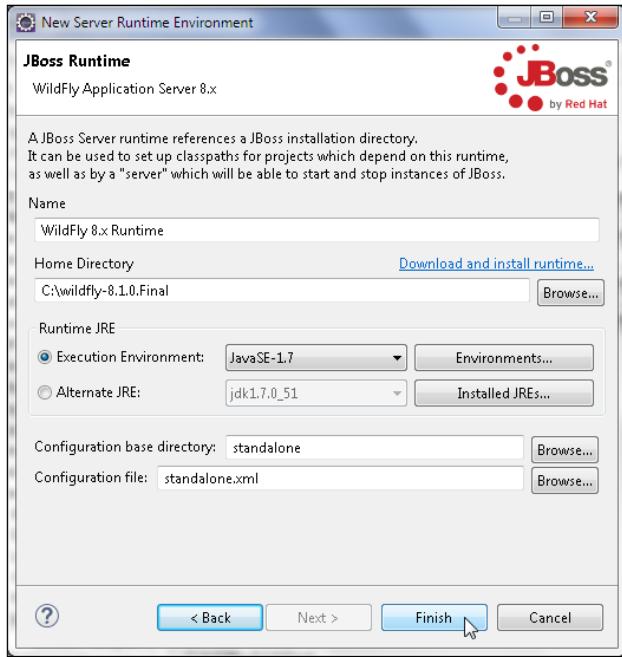


Getting Started with EJB 3.x

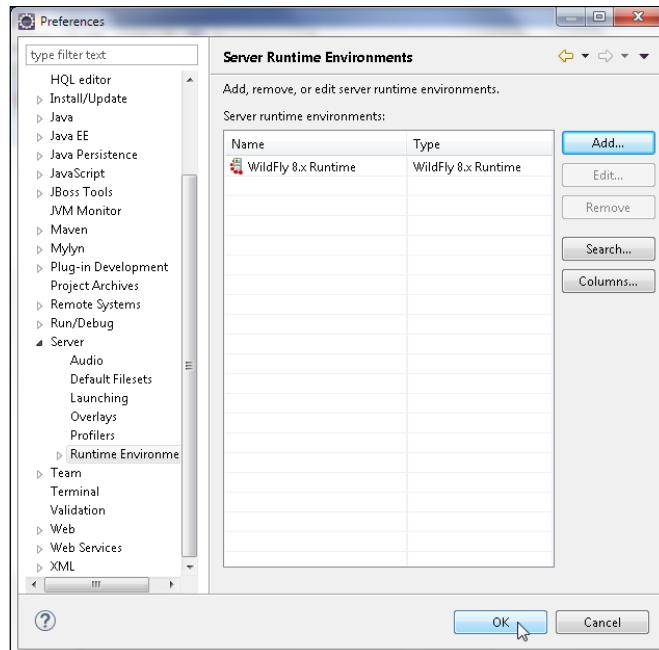
In **New Server Runtime Environment**, select **JBoss Community | WildFly & Runtime**. Click on **Next**:



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 Module and 8.2 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**:



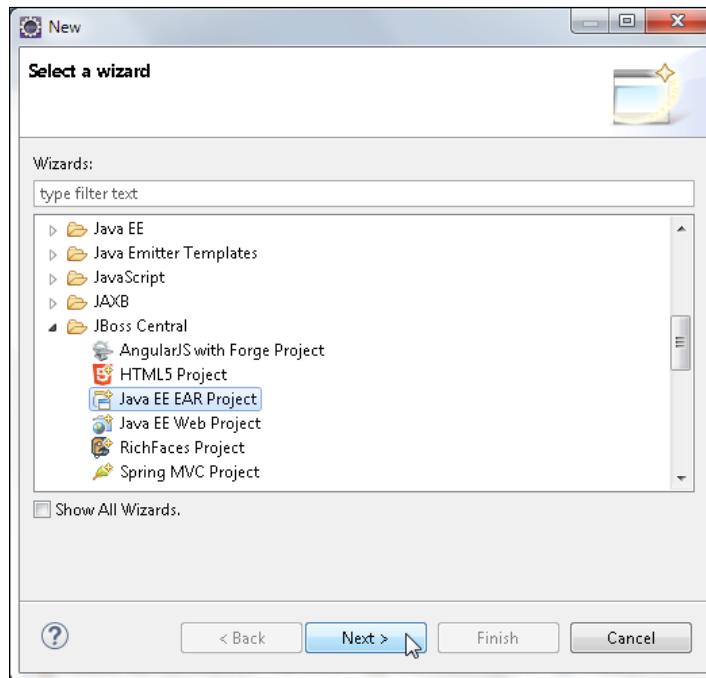
A new server runtime environment for WildFly 8.x Runtime gets created, as shown in the following screenshot. Click on **OK**:



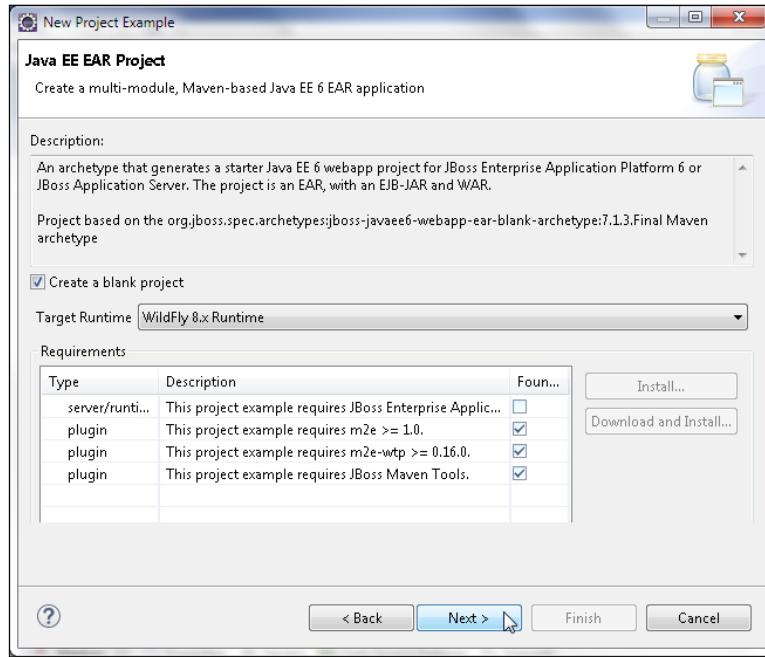
Creating a **Server Runtime Environment** for WildFly 8.x is not a prerequisite for creating a Java EE project in Eclipse. In the next section, we will create a new Java EE project for an EJB 3.x application.

Creating a Java EE project

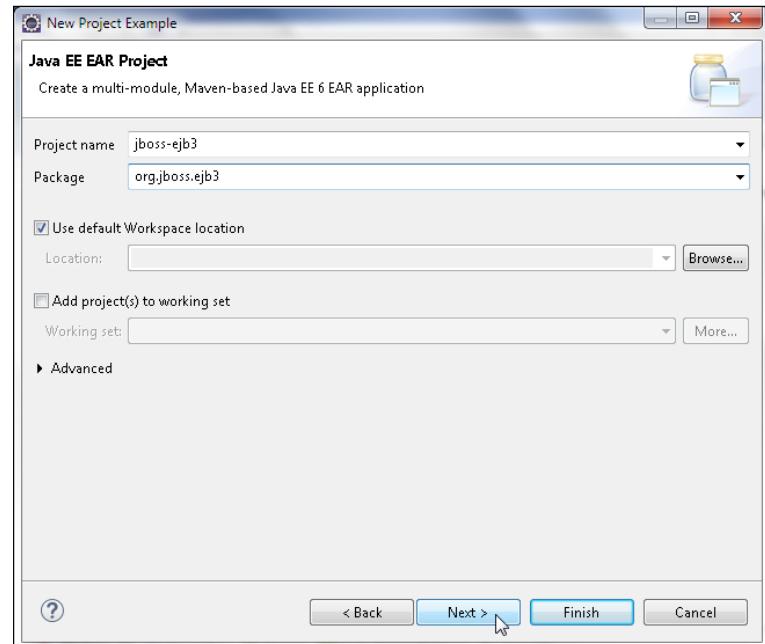
JBoss Tools provides project templates for different types of JBoss projects. In this section, we will create a Java EE project for an EJB 3.x application. Select **File | New | Other** in Eclipse IDE. In the **New** wizard, select the **JBoss Central | Java EE EAR Project** wizard. Click on the **Next**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

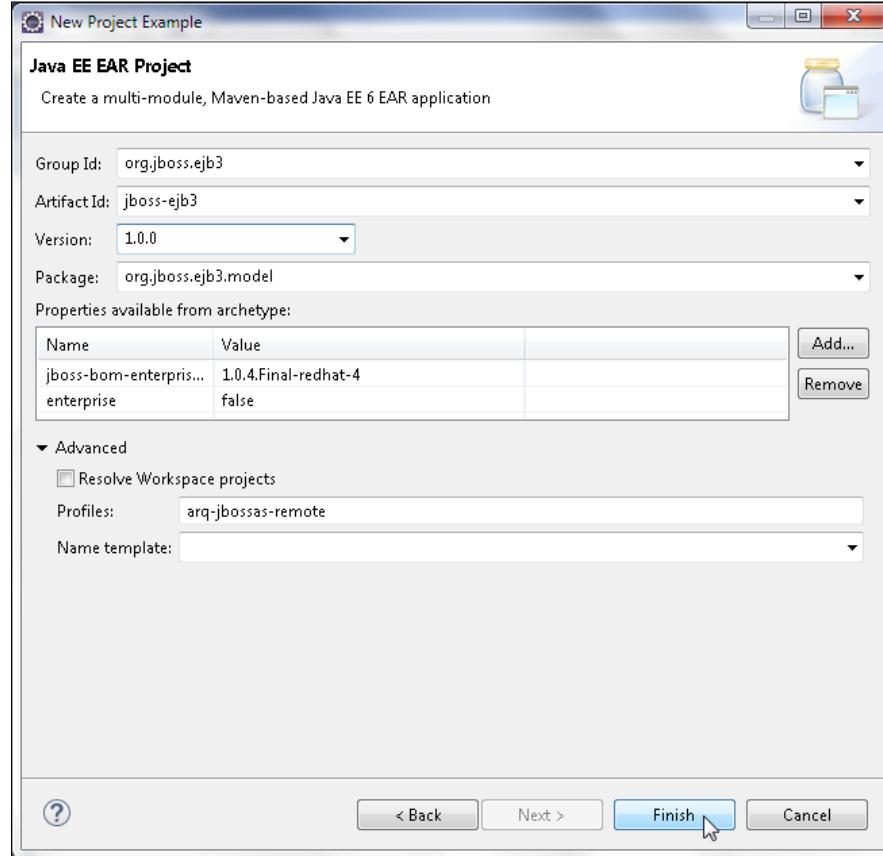


Specify **Project name** as `jboss-ejb3`, **Package** as `org.jboss.ejb3`, and tick the **Use default Workspace location** box. Click on the **Next** button:

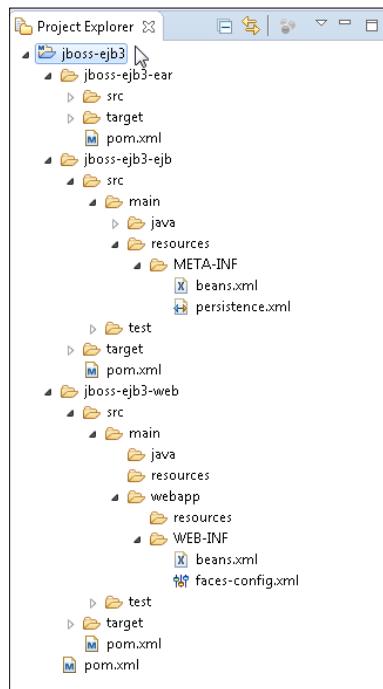


Getting Started with EJB 3.x

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:



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.



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:



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:

The screenshot shows the JBoss Management console interface for WildFly 8.1.0.Final. The left sidebar has sections for Server (Overview, Manage Deployments, Patch Management), Status (Platform, JVM, Environment, Subsystems, Datasources, JPA, JNDI View, Transaction Logs, Transactions, Webservices), and Tools/Settings. The main content area has tabs for DATA SOURCES and XA DATA SOURCES. Under DATA SOURCES, it says 'Data Source Metrics' and 'Datasources runtime utilisation.' Below this is a table with two rows:

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

Below the table is a 'Pool Usage' chart showing '50% Active Count' compared to 'Available' (0 to 20 scale) and '5% Max Used'. At the bottom right of the main content area are 'Tools' and 'Settings' buttons.

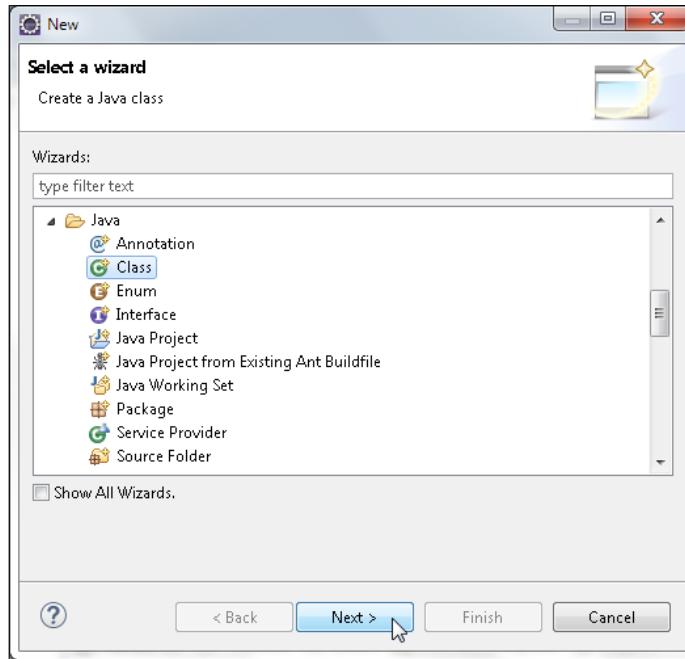
8. If a connection with the MySQL database is established, a **Successfully created JDBC connection** message will get displayed:

The screenshot shows the JBoss Management console interface for WildFly 8.1.0.Final. A modal dialog box titled 'Test Connection' is open over the main content area. The dialog displays the message: 'Successfully created JDBC connection. Successfully connected to database MySQLDS.' There is an 'OK' button at the bottom right of the dialog. The background content is identical to the previous screenshot, showing the Data Source Metrics page.

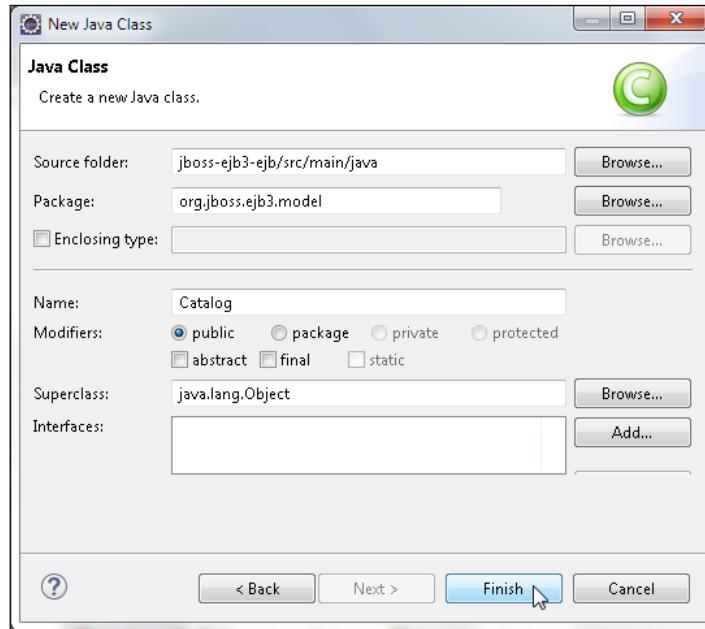
In the next section, we will create entities for the EJB 3.x application.

Creating entities

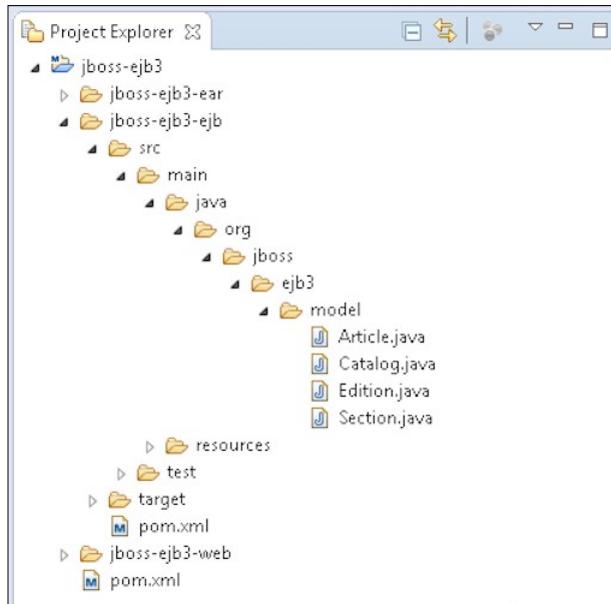
In EJB 3.x, an entity is a **POJO (Plain Old Java Object)** persistent domain object that represents a database table row. As an entity is a Java class, create a Java class in the `jboss-ejb3-ejb` subproject of the `jboss-ejb3` project. Select **File | New**. In the **New** window, select **Java | Class**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



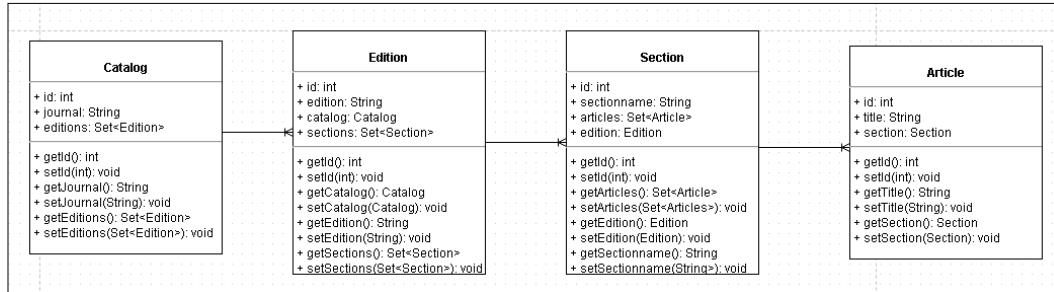
Similarly, add Java classes for the `Edition.java`, `Section.java`, and `Article.java` entities, as shown in the following **Project Explorer**:



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 [sundar github](#).

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 [Sundar github](#)

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" />
    </properties>
  </persistence-unit>
</persistence>
```

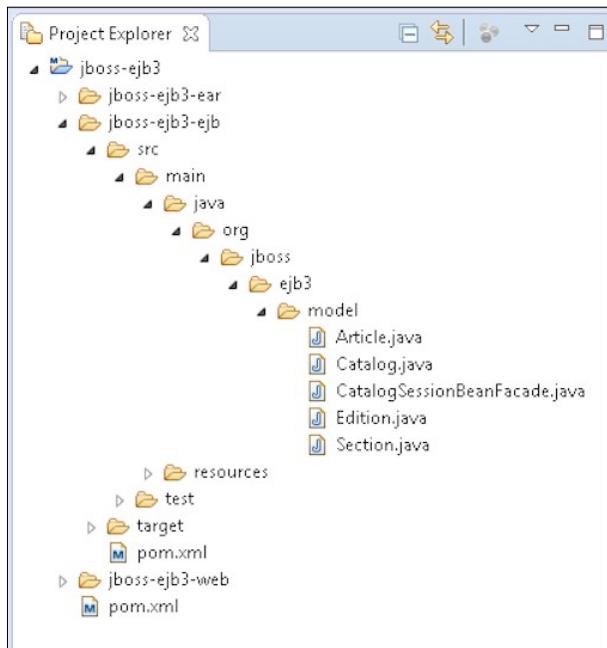


```
<property name="hibernate.dialect" value="org.hibernate.dialect.  
MySQLDialect" />  
</properties>  
</persistence-unit>  
</persistence>
```

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 `getEditions`, `getSections`, `getArticles`, `getCatalogs` 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);
```

[21]

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);
```

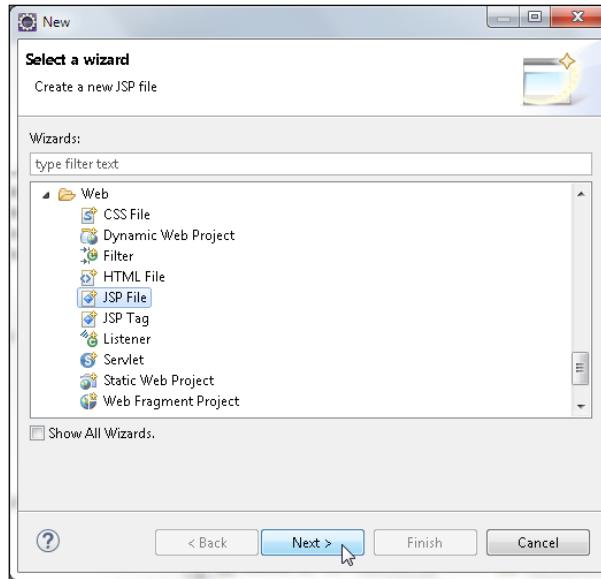
Next, we will delete data with the `deleteSomeData` method, wherein we first create a `Query` object using the named query `findCatalogByJournal`. Specify the journal to delete with the `setParameter` method of the `Query` object. Get the `List` result with the `getResultSet` method of the `Query` object. 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.getResultSet();
    for (Catalog catalog : catalogs) {
        em.remove(catalog);
    }
}
```

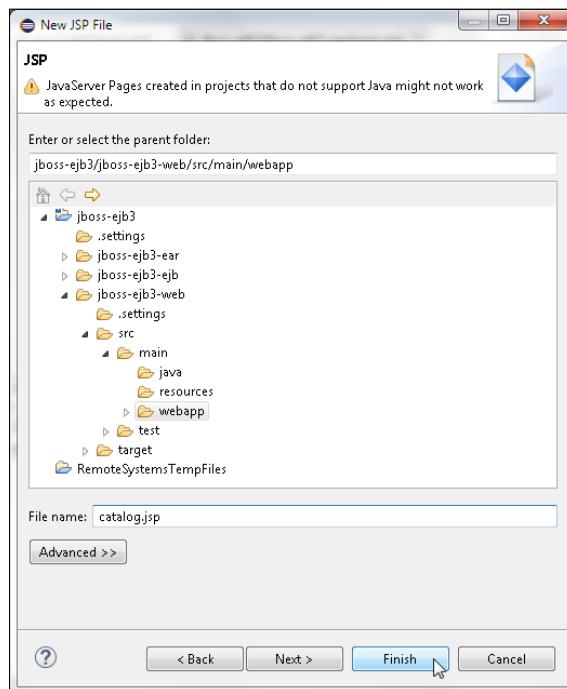
The `CatalogSessionBeanFacade` session bean class is available in the code downloaded for the chapter.

Creating a JSP client

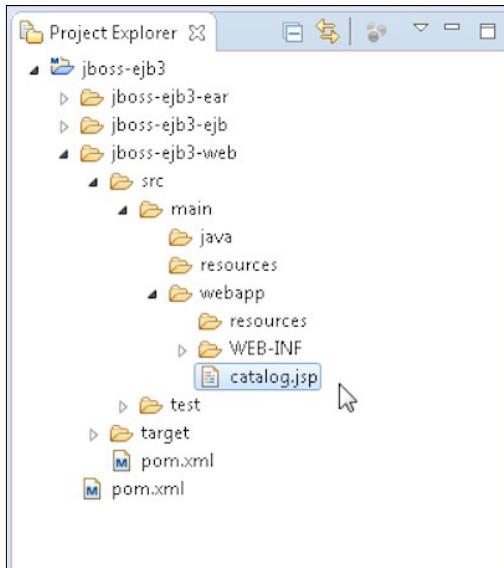
Next, we will create a JSP client to test the EJB entities. We will look up the session bean using a local JNDI name. Subsequently, we will invoke the `testData` method of the session bean to test database persistence using these entities. First create a JSP file. Select **File | New | Other**, and in 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** 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
<code>java:comp</code>	This is the namespace that is scoped to the current component, the EJB.
<code>java:module</code>	This namespace is scoped to the current module.
<code>java:app</code>	This namespace is scoped to the current application.
<code>java:global</code>	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:

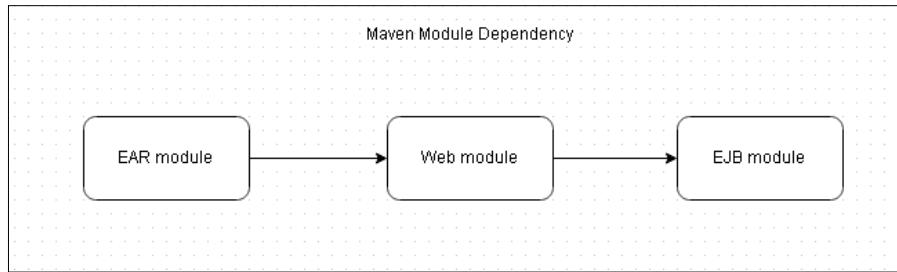
```
bean.createTestData();  
List<Catalog> catalogs = beanRemote.getAllCatalogs();  
out.println("<br/>" + "List of Catalogs" + "<br/>");  
for (Catalog catalog : catalogs) {  
    out.println("Catalog Id:" + catalog.getId() + "<br/>");  
    out.println("Catalog Journal:" + catalog.getJournal() + "<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-ejb3-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 codedownload 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>
```

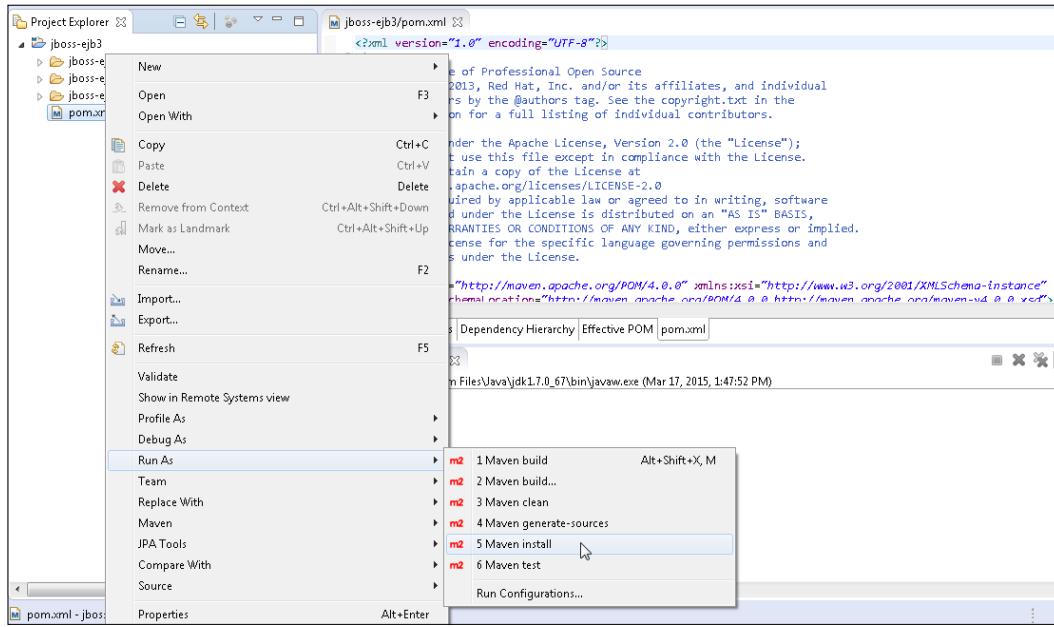
Specify the JBoss AS version as `8.1.0.Final`:

```
<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:

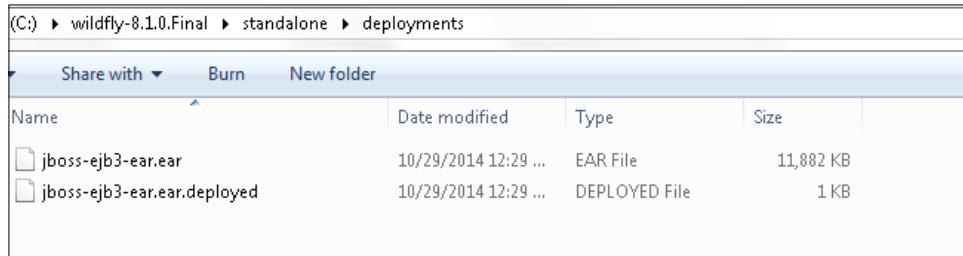


As the output from the pom.xml indicates all the three modules: ejb, web, and ear get built. The ear module gets copied to the deployments directory in WildFly 8.x:

```
<terminated> C:\Program Files\Java\jdk1.7.0_67\bin\javaw.exe (Mar 17, 2015, 2:16:05 PM)
[INFO]
[INFO] --- 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 from org.jboss.jboss-ejb3:jboss-ejb3-ejb:1.0.0 to [jboss-ejb3-web.war]
[INFO] Copying artifact [org.jboss.jboss-ejb3:ejb:1.0.0] to [jboss-ejb3-ejb.jar]
[INFO] Copying 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\jbc
[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: 22N/189M
[INFO]
```

Getting Started with EJB 3.x

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:



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-ejb3-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>

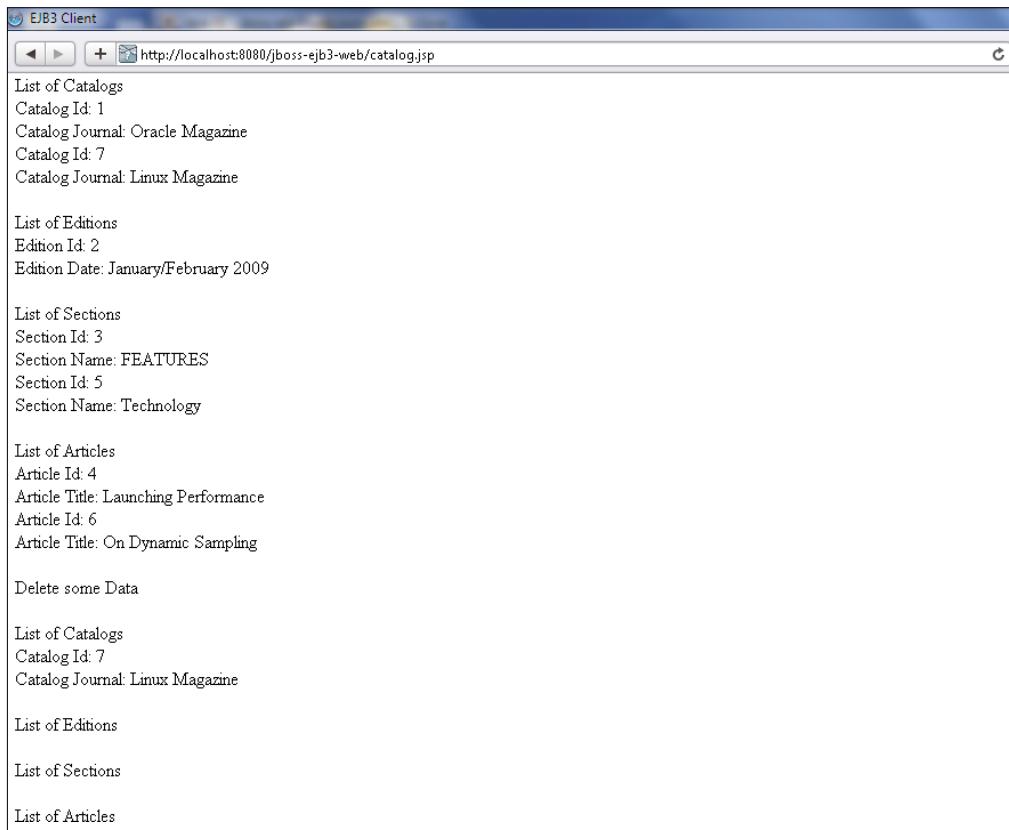
mysql>
```

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:

The screenshot shows the WildFly 8.1.0.Final administration console interface. The top navigation bar includes Home, Configuration, Runtime (which is selected), and Administration. On the left, a sidebar menu lists Server, Overview, Manage Deployments (which is selected), Patch Management, Status, Platform (JVM, Environment, Subsystems), Datasources, JPA, JNDI View, Transaction Logs, Transactions, and Webservices. The main content area is titled 'DEPLOYMENTS' and contains a section for 'Deployments'. It says 'Currently deployed application components.' and shows a list of 'Available Deployments' with a single entry: 'jboss-ejb3-ear.ear'. Below this is a 'Deployment' form with fields for 'Name:' (set to 'jboss-ejb3-ear.ear') and 'Runtime Name:' (also set to 'jboss-ejb3-ear.ear'). Action buttons at the top right of the deployment list include Add, Remove, En/Disable, and Replace.

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:



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 2.2 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.javaee.ejb</groupId>
<artifactId>jboss-ejb-api_3.2_spec</artifactId>
<scope>provided</scope>
</dependency>
<dependency>
<groupId>org.hibernate.javaee.persistence</groupId>
<artifactId>hibernate-jpa-2.1-api</artifactId>
<scope>provided</scope>
</dependency>
<dependency>
<groupId>org.jboss.spec.javaee.servlet</groupId>
<artifactId>jboss-servlet-api_3.1_spec</artifactId>
<scope>provided</scope>
</dependency>
<dependency>
<groupId>org.jboss.spec.javaee.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.

- 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

[37]

- 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 *Module1, 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 *Module1, Getting Started with EJB 3.x*:

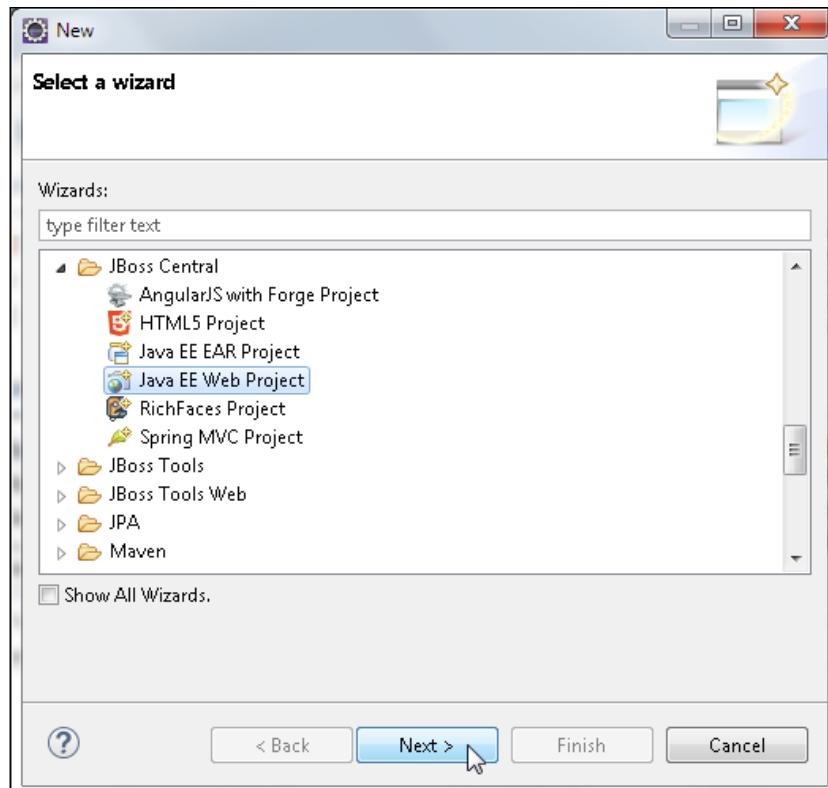
`JAVA_HOME`, `JBOSS_HOME`, `MAVEN_HOME`, and `MYSQL_HOME`. Add `%JAVA_HOME%/bin`,
`%MAVEN_HOME%/bin`, `%JBOSS_HOME%/bin`, and `%MYSQL_HOME%/bin` to the `PATH`
environment variable.

Create a WildFly 8.1.0 runtime as discussed in *Module1, Getting Started with EJB 3.x*.

Creating a Java EE web project

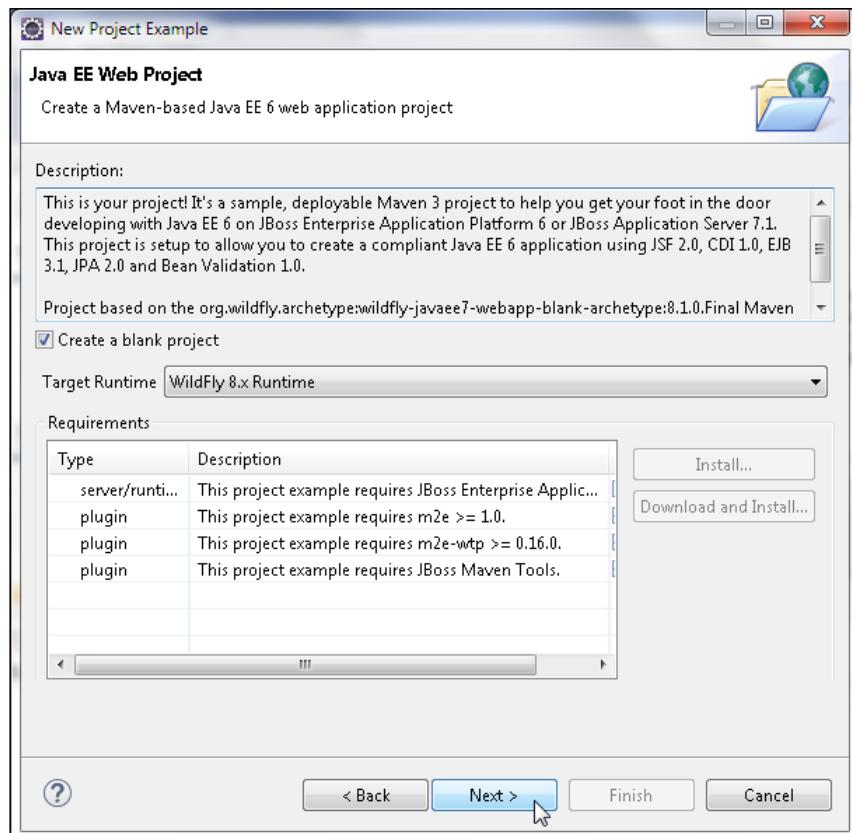
In this section, we will create **Java EE Web Project** in Eclipse IDE. Perform the following steps to accomplish this task:

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:

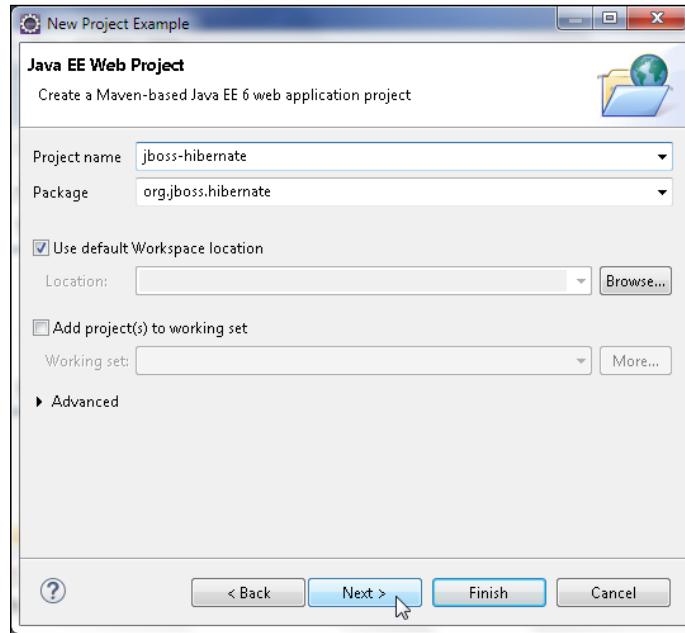


The **Java EE Web Project** wizard gets started. A test is run for the requirements, which includes a JBoss server runtime, the JBoss Tools runtime, and the **m2e** and **m2eclipse-wtp** plugins.

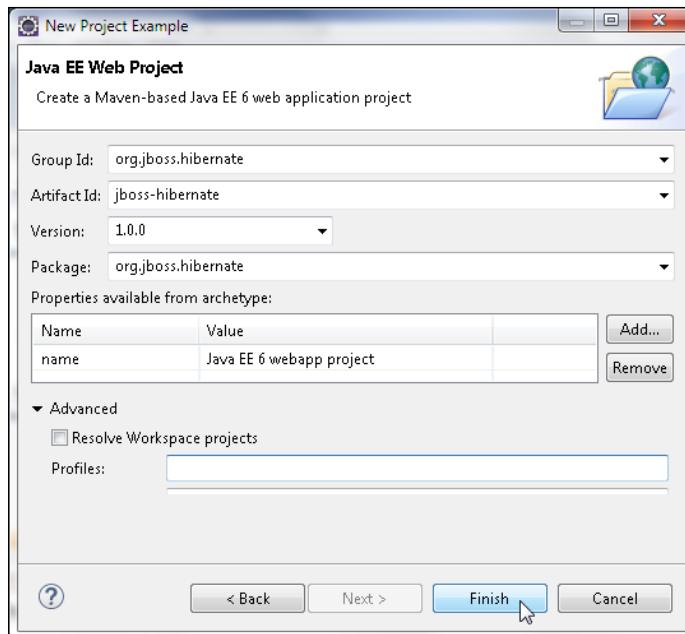
2. Select the **Create a blank project** checkbox and **Target Runtime WildFly 8.x Runtime** and click on **Next**, as shown in the following screenshot. Even though Java EE Web Project indicates the Java EE version as Java EE 6, a Java EE 7 web project is actually created.



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

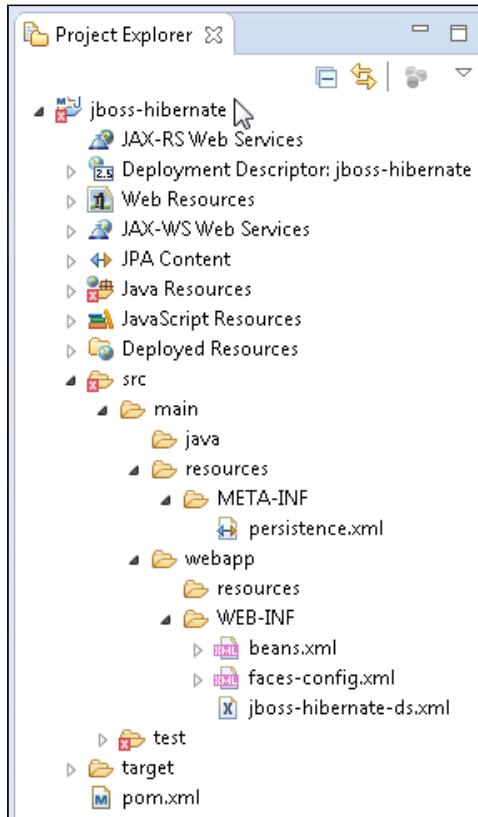


4. Specify **Group Id** (org.jboss.hibernate), **Artifact Id** (jboss-hibernate), **Version** (1.0.0), and **Package** (org.jboss.hibernate), as shown in the following screenshot. Click on the **Finish** button.



[41]

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

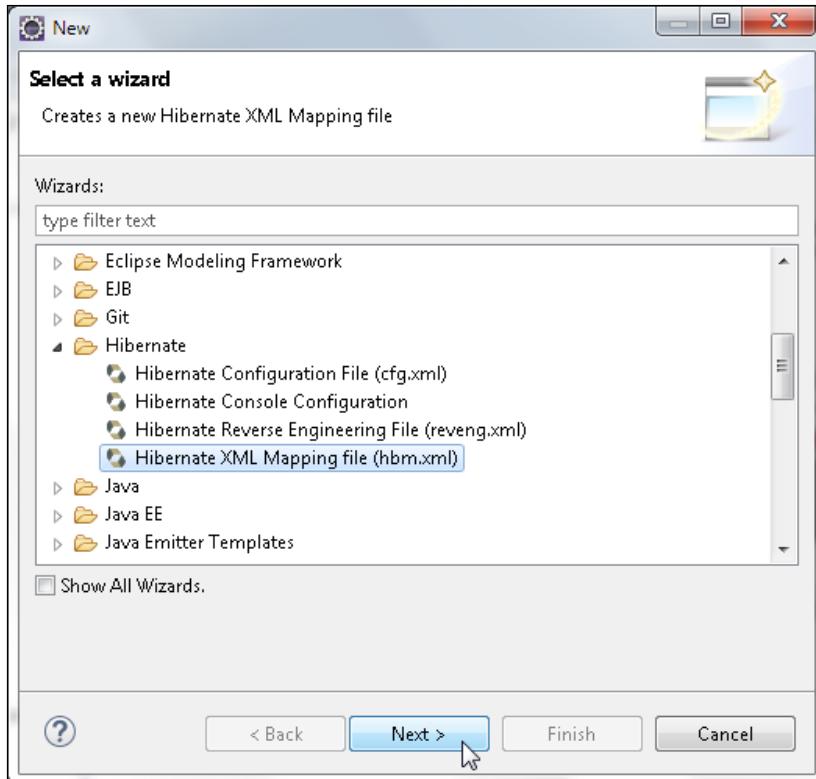


Creating a Hibernate XML Mapping file

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:

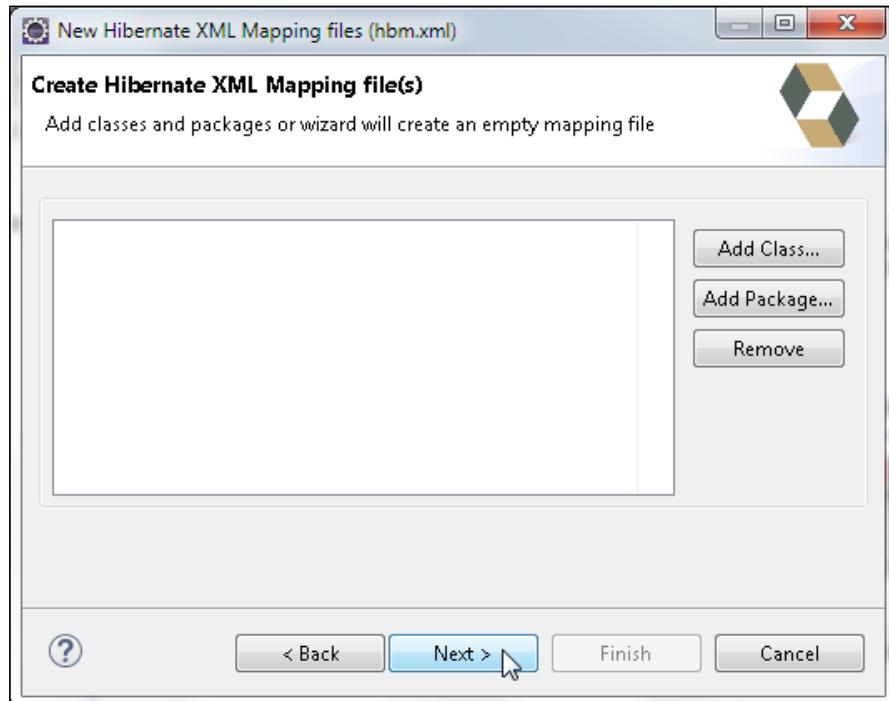
1. To create a Hibernate XML Mapping file, select **File | New | Other**.

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



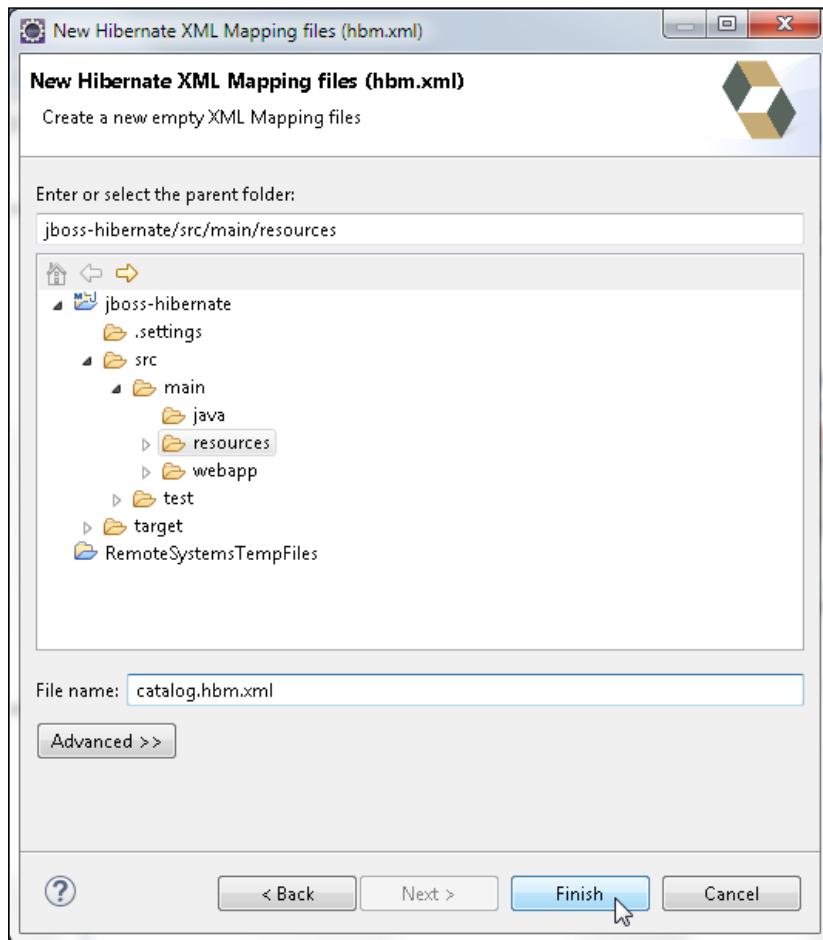
The **New Hibernate XML Mapping files** wizard gets started. As we have not yet defined any persistence classes to map, we will first create an empty XML mapping file.

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



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**.



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.

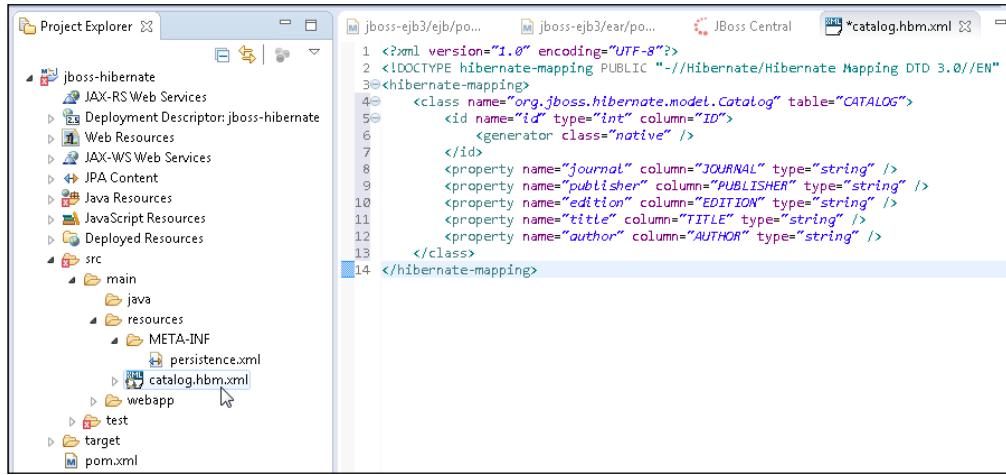
As we are using the MySQL database, which supports identity columns using `AUTO_INCREMENT`, we can use the generation strategy as `identity` or `native`. An identity column is a table column of the `INTEGER` type, with `AUTO_INCREMENT` and `PRIMARY KEY` or `UNIQUE KEY` specified, such as `id INTEGER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

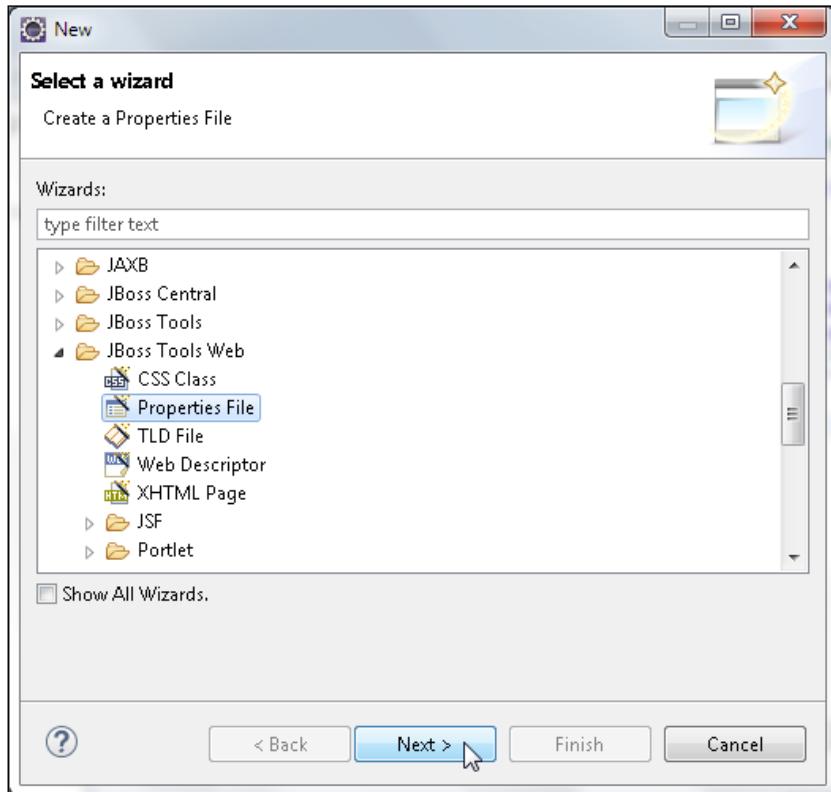
```
<class name="org.jboss.hibernate.model.Catalog" table="CATALOG">
    <id name="id" type="int" column="ID">
        <generator class="native" />
    </id>
    <property name="journal" column="JOURNAL" type="string" />
    <property name="publisher" column="PUBLISHER" type="string" />
    <property 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:



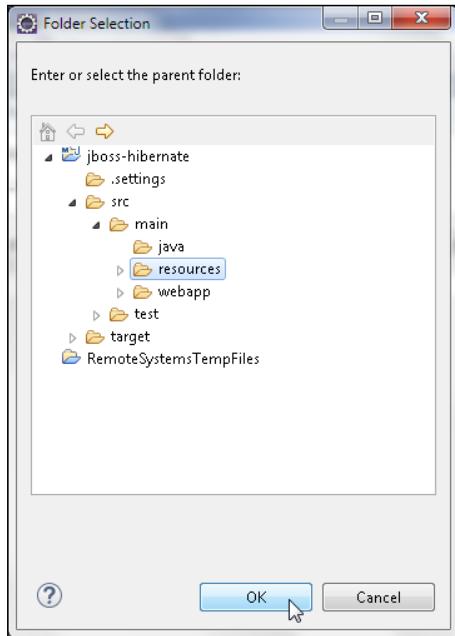
Creating a properties file

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:

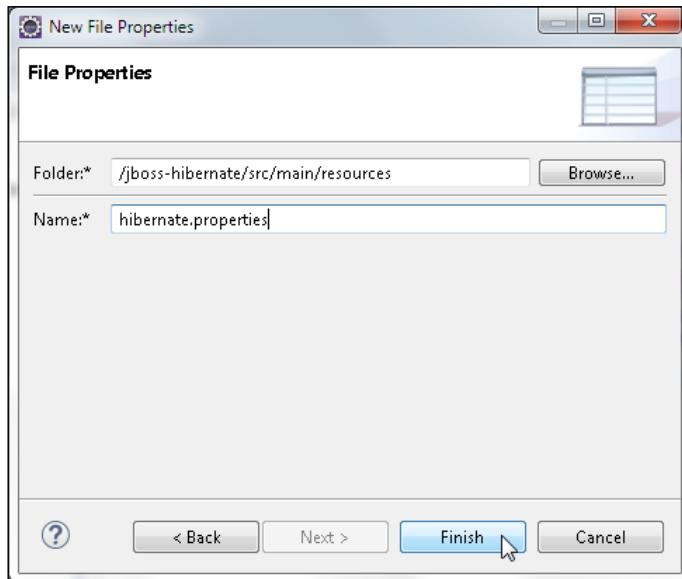


The **New File Properties** wizard gets started.

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:

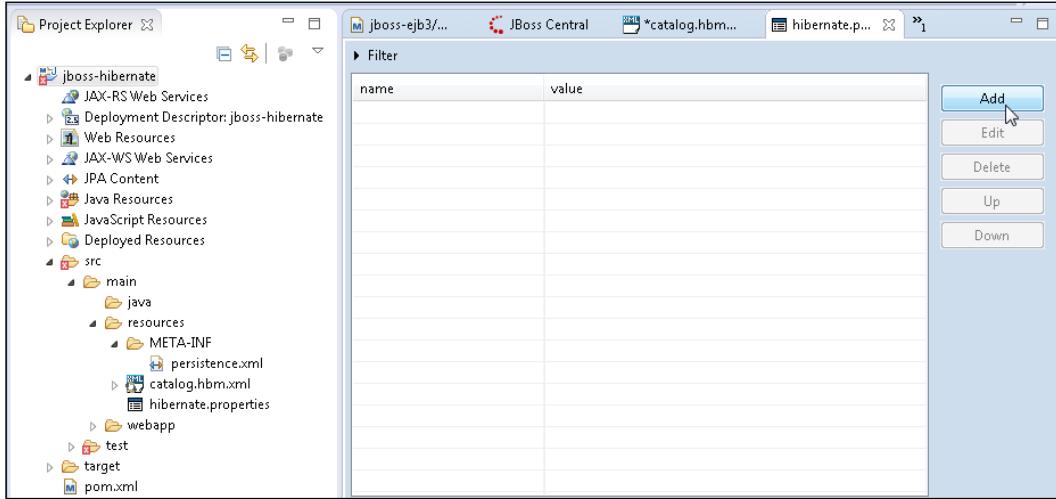


3. Specify **Name*** as `hibernate.properties` and click on **Finish**, as shown in the following screenshot:

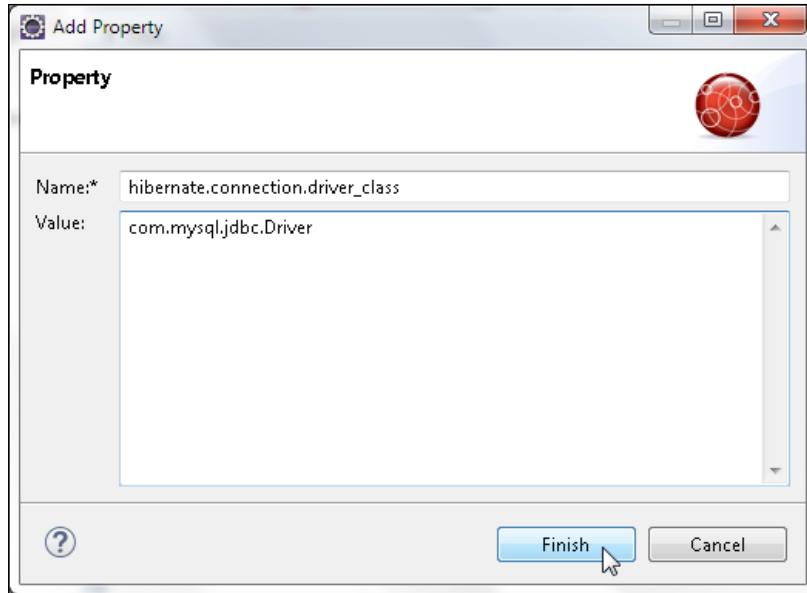


The `hibernate.properties` file gets added to the `resources` folder.

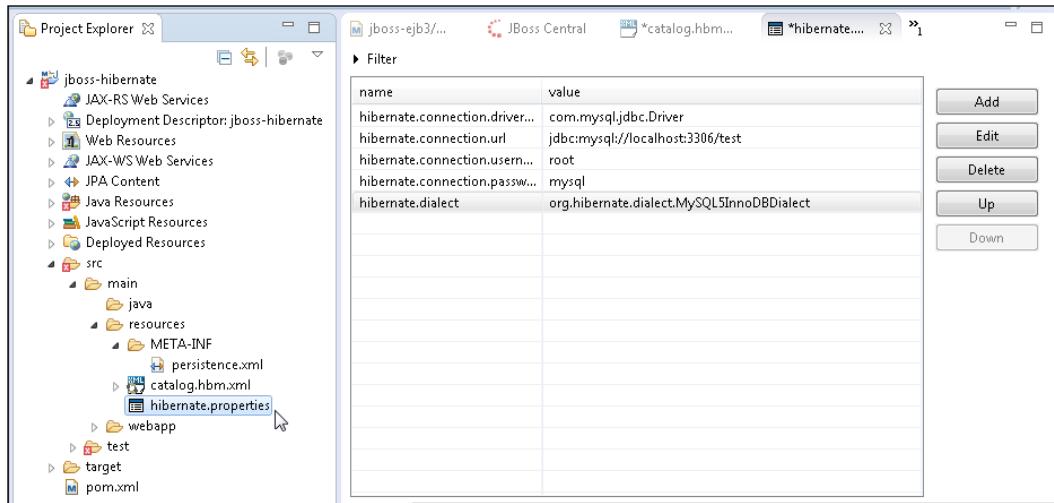
4. In the `hibernate.properties` table, click on the **Add** button to add a property, as shown in the following screenshot:



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:



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:



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.

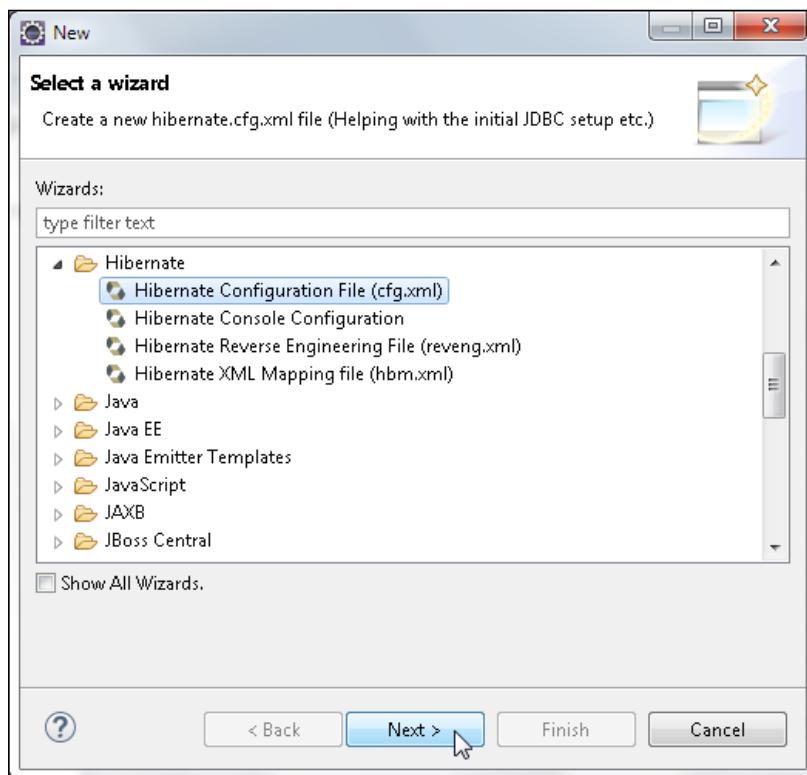
Either the properties file or the configuration file can be used to specify the configuration, or both can be used. If both are provided, the configuration file overrides the properties file for the configuration parameters specified in both.

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.

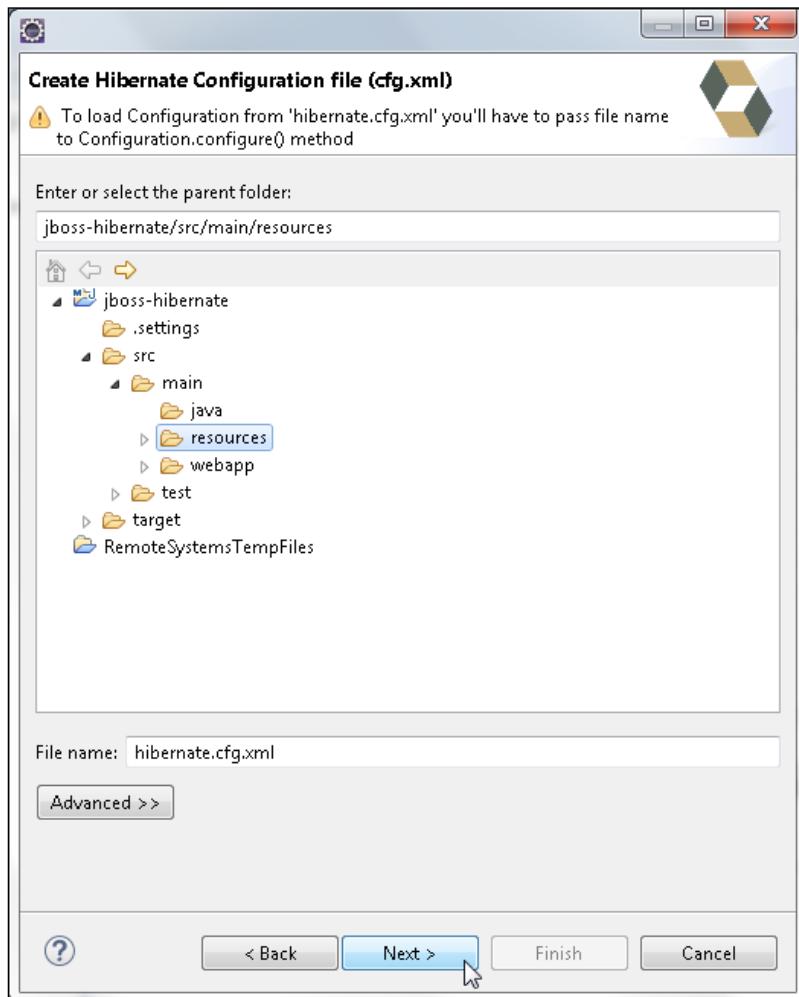
The following are the steps to create a Hibernate configuration file:

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



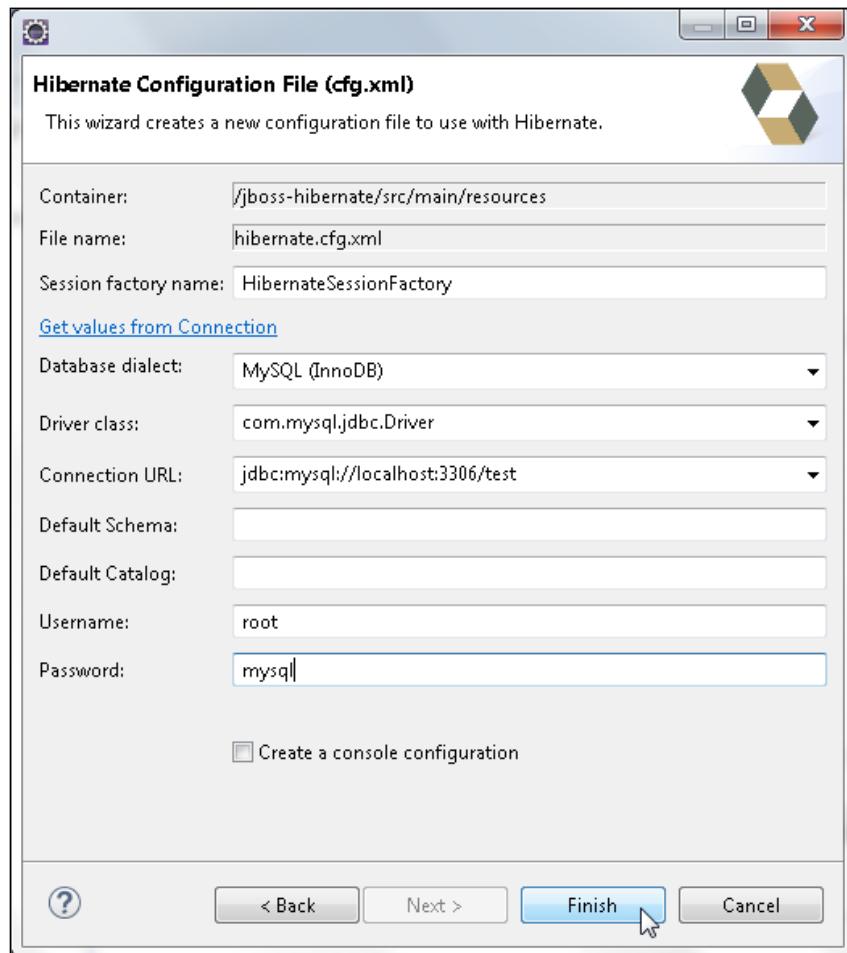
The **Create Hibernate Configuration file** wizard gets started.

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:



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:

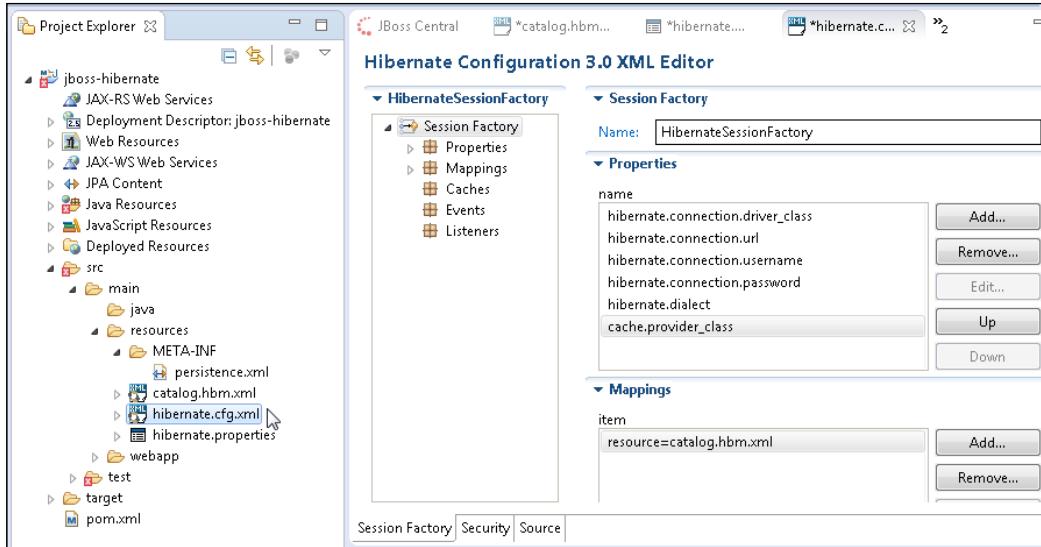


Hibernate provides transaction-level caching of persistence data in a session by default. Hibernate has the provision for a query-level cache, which is turned off by default, to frequently run queries. Hibernate also has the provision for a second-level cache on the `SessionFactory` level or on the cluster 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">mysql116</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>
```

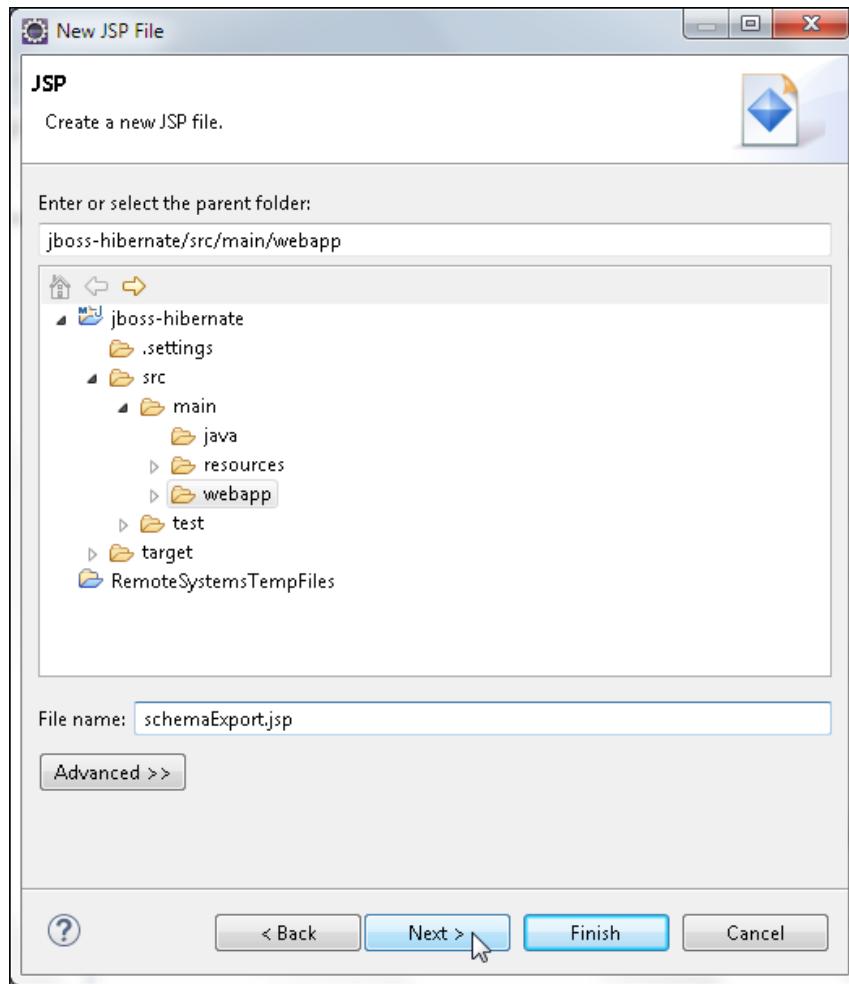
The `hibernate.cfg.xml` file is shown in the jboss-hibernate project folder, as follows:



Creating JSPs for CRUD

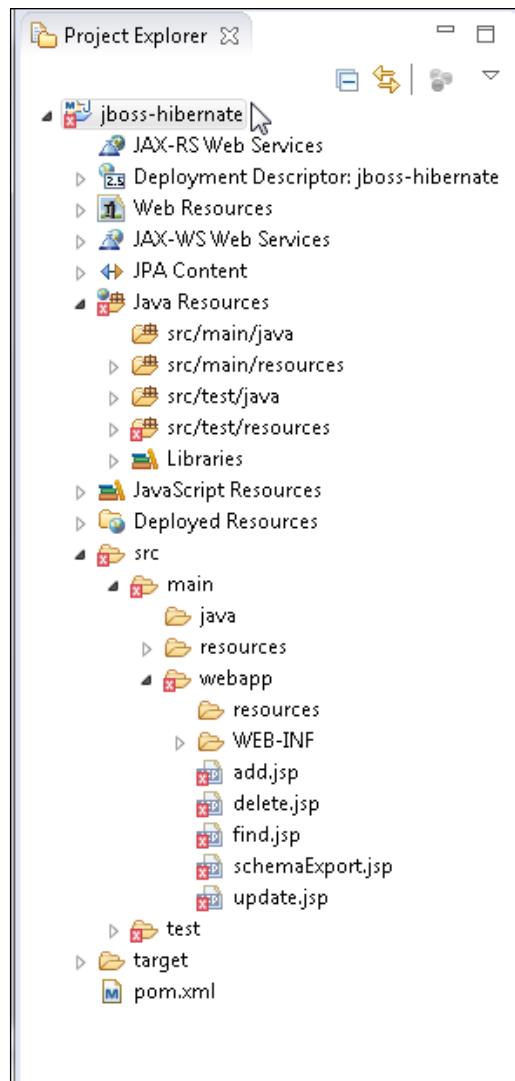
We have created the required configuration files for Hibernate. Next, we will create the JSPs to persist, load, update, and delete POJO domain objects, which are also referred to as **create, read, update, delete (CRUD)**. Perform the following steps to accomplish this:

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).

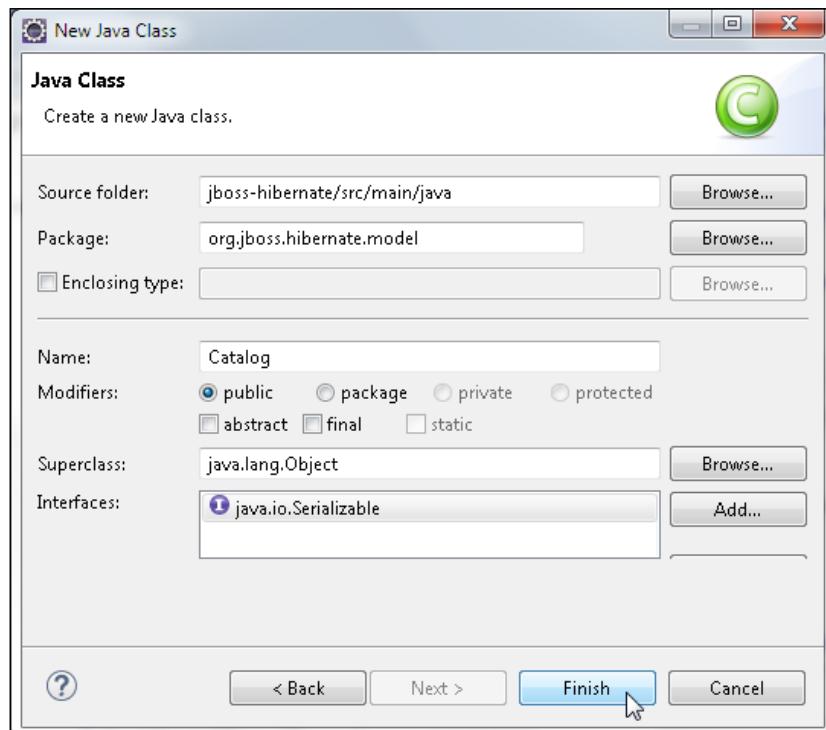
The directory structure of the `jboss-hibernate` project is shown in the following screenshot. The JSP files might indicate an error, which will get fixed as the application is developed and the Maven dependencies are added.



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:



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:

The screenshot shows the Eclipse IDE interface. On the left, the Project Explorer view displays the `jboss-hibernate` project structure. Inside the `src/main/java` folder, there is a package named `org.jboss.hibernate.model` containing a file named `Catalog.java`. The right side of the screen shows the code editor with the content of `Catalog.java`:

```

1 package org.jboss.hibernate.model;
2
3 import java.io.Serializable;
4
5 public class Catalog implements Serializable {
6     /** identifier field */
7     private Integer id;
8     /** nullable persistent field */
9     private String journal;
10    /** nullable persistent field */
11    private String publisher;
12    /** nullable persistent field */
13    private String edition;
14    /** nullable persistent field */
15    private String title;
16    /** nullable persistent field */
17    private String author;
18
19    /** full constructor */
20    public Catalog(String journal, String publisher, String edition,
21                   String title, String author) {
22        this.journal = journal;
23        this.publisher = publisher;
24        this.edition = edition;
25        this.title = title;
26        this.author = author;
27    }

```

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);
```

[61]

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

```
schemaExport.set outputFile("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.set outputFile("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");
        %>
    </body>
</html>
```

```
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, getList 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.

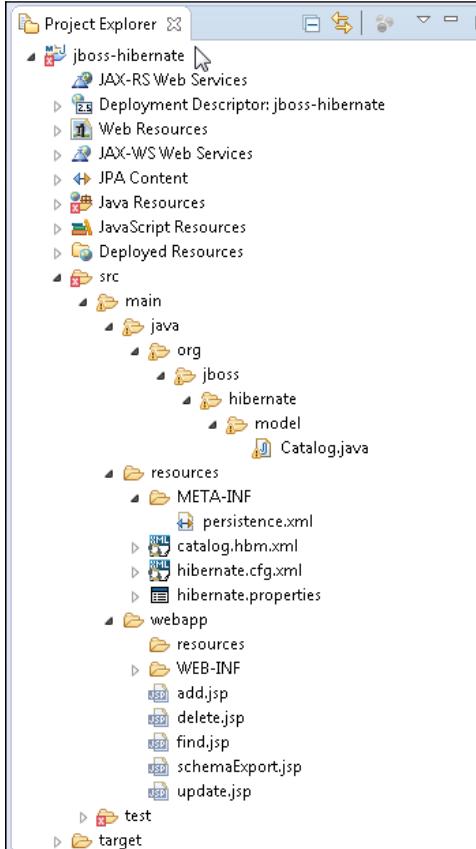
5. Optionally, using the following code, output a message to indicate deletion:

```
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:



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 accomplish this, use the following code:

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

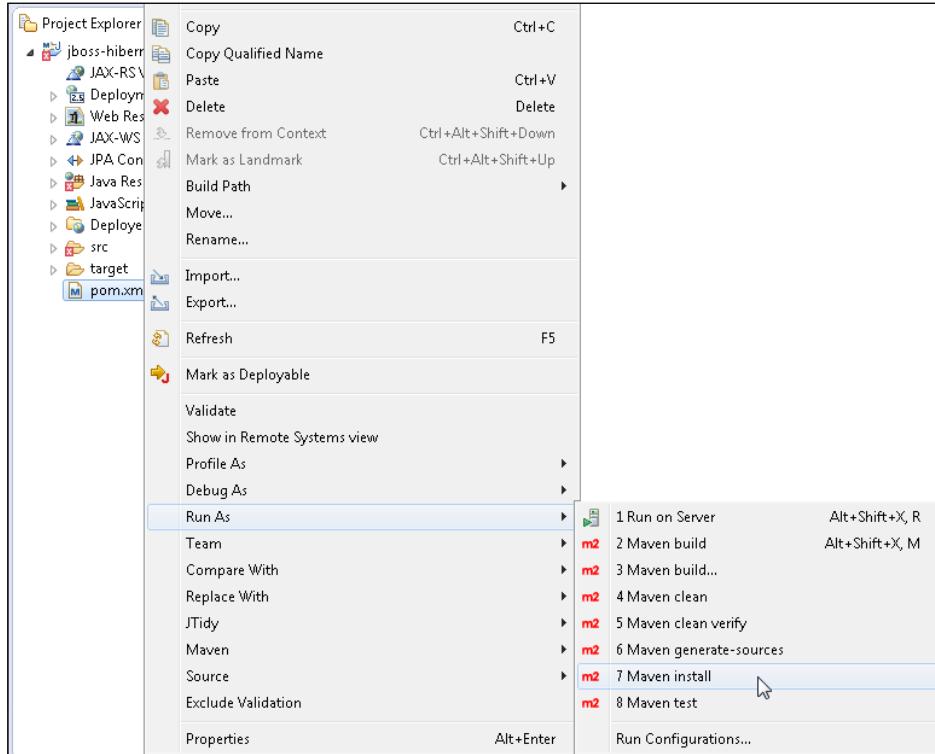
Hibernate provides Hibernate-related artifacts in the group ID `org.hibernate`. Include the Hibernate artifacts listed in the following table.

Artifact	Description
<code>hibernate-core</code>	This is the main artifact for the Hibernate API.
<code>hibernate-annotations</code>	This is for the annotations metadata.
<code>hibernate-commons-annotations</code>	This pertains to the EJB 3 style annotations for Hibernate.
<code>hibernate-ehcache</code>	This provides the cache for second-level cache.
<code>hibernate-c3p0</code>	This is the C3P0 connection-pooling library.
<code>antlr</code>	This is the parser generator.
<code>hibernate-cglib-repack</code>	This is the CGLib code generation library and also signifies ASM dependencies.
<code>hibernate-tools</code>	This provides tools to generate various Hibernate source artifacts, such as mapping files and Java entities.
<code>hibernate-envers</code>	This is used for auditing and versioning of persistent classes.
<code>hibernate-jpamodelgen</code>	This is an annotation processor to generate JPA 2 static meta-model classes. The <code>Analog entity</code> in Module1 - 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.

Developing Object/Relational Mapping with Hibernate 4

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

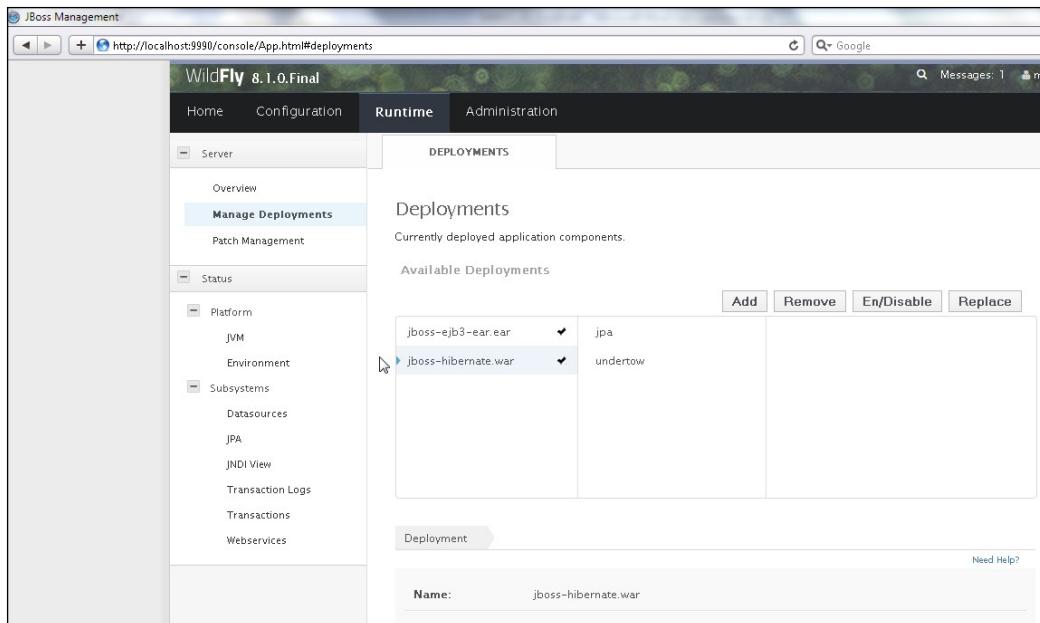


The Maven project is compiled and the jboss-hibernate.war archive gets generated and output to the deployments directory of WildFly 8, as shown in the following screenshot:

```
<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\re
[INFO] Installing C:\Users\Deepak Vohra\Eclipse\workspace\jboss-hibernate\pom.xml to C:\Users\Deepak Vohra\.m2\re
[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:



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.



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.MySQL5InnoDBialect  
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 connection 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 size: 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
Yorshayi2554,INFO [org.hibernate.engine.jdbc.connections.
1at67n24,039vEMMnaqstdout] (default task-8)          AUTHOR
ConnectionProviderImpl (default task-8) HHH000030: Cleaning up
donated connection INFO [stdout] (default task-8)          primary key
[JDBC:mysql://localhost:3306/test]
10:57:24,049 INFO [stdout] (default task-8)          ENGINE=InnoDB
[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 Uohra>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)

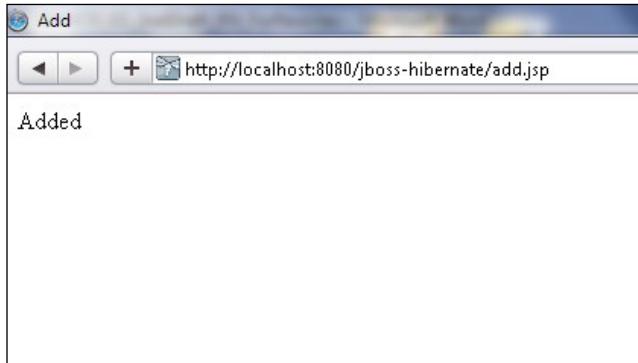
mysql> DESC CATALOG;
+-----+-----+-----+-----+-----+-----+
| Field | Type  | Null | Key  | Default | Extra       |
+-----+-----+-----+-----+-----+-----+
| ID   | int(11) | NO   | PRI  | NULL    | auto_increment |
| JOURNAL | varchar(255) | YES  |       | NULL    |             |
| PUBLISHER | varchar(255) | 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.



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> SELECT * FROM CATALOG;
+----+-----+-----+-----+-----+
| ID | JOURNAL | PUBLISHER | EDITION | TITLE
+----+-----+-----+-----+
| 1  | Oracle Magazine | Oracle Publishing | Jan-Feb 2004 | Understanding Optimization
| 2  | Oracle Magazine | Oracle Publishing | March-April 2005 | Starting with Oracle ADF | Steve Muench |
+----+-----+-----+-----+
2 rows in set <0.00 sec>

mysql>
```

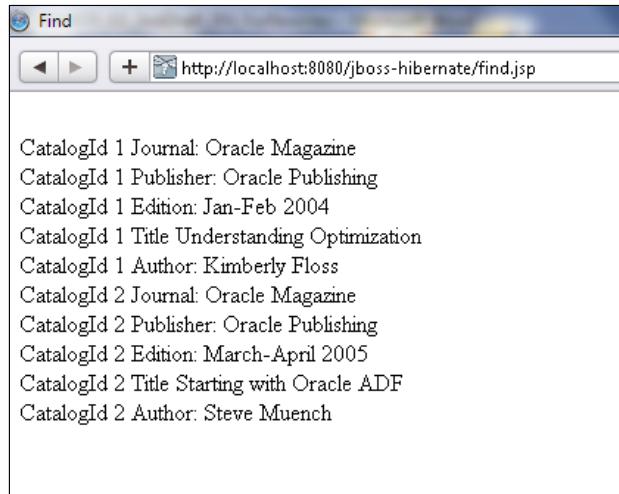
The screenshot shows a terminal window running MySQL. The command 'SELECT * FROM CATALOG;' is run, resulting in the following output:

ID	JOURNAL	PUBLISHER	EDITION	TITLE
1	Oracle Magazine	Oracle Publishing	Jan-Feb 2004	Understanding Optimization
2	Oracle Magazine	Oracle Publishing	March-April 2005	Starting with Oracle ADF Steve Muench

There are two rows returned, both with ID values of 1 and 2.

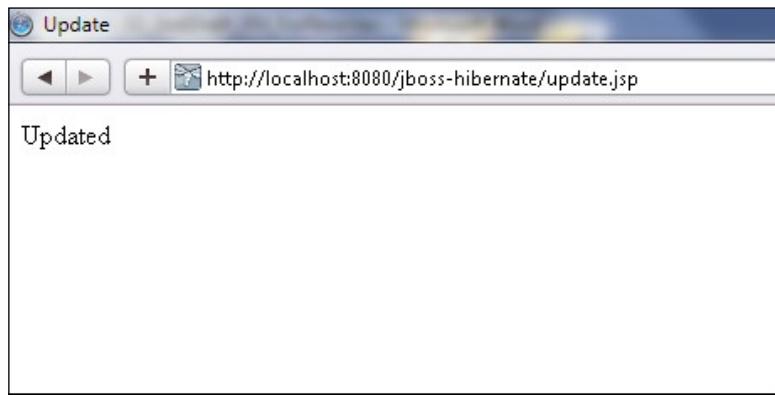
Retrieving table data

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.

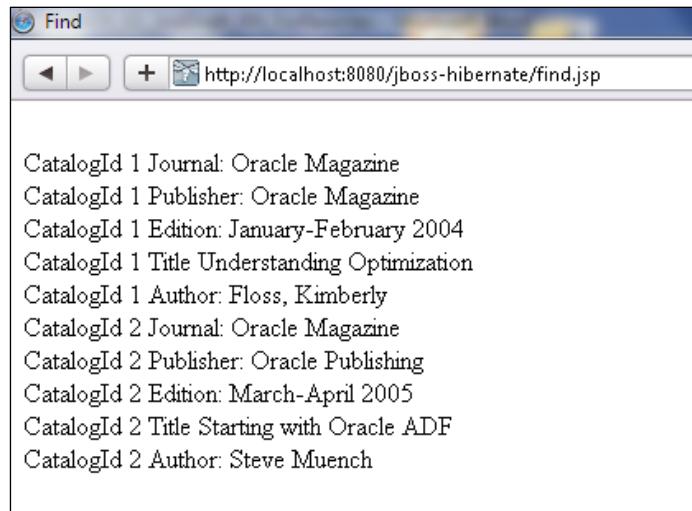


Updating the table

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.



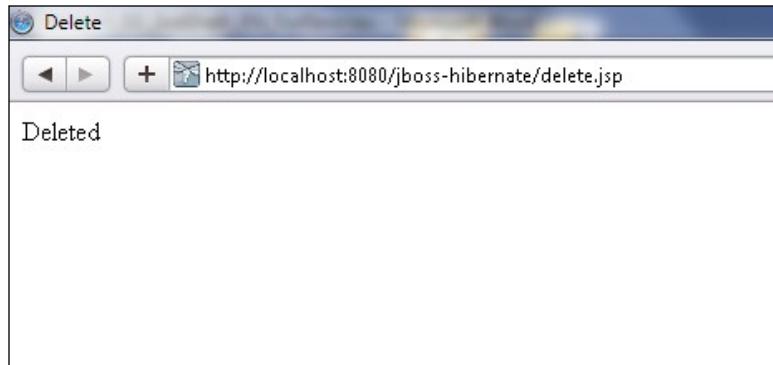
Run `find.jsp` to get and display the updated `CATALOG` table data. Invoke the URL `http://localhost:8080/jboss-hibernate/find.jsp`, as shown in the following screenshot:



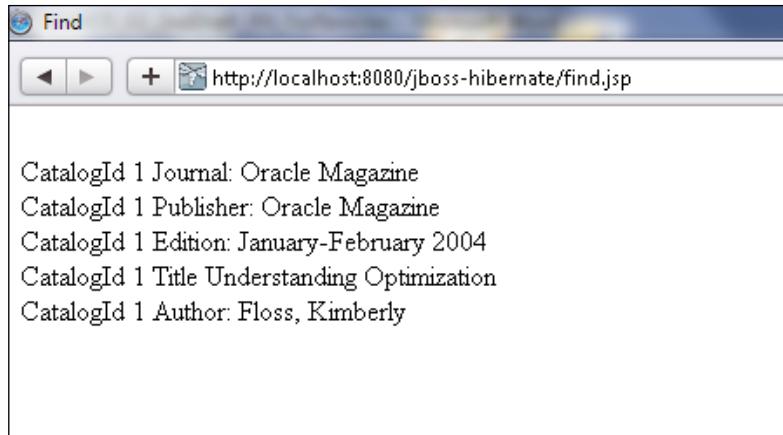
Deleting the table row

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`.



-
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:



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.

