Creating JEE Database Applications

In the previous chapter, we learned how to use source control management software from Eclipse. Specifically, we learned how to use SVN and Git from Eclipse. In this chapter, we will get back to discussing JEE application development. Most web applications today require access to the database. In this chapter, we will learn two ways to access databases from JEE web applications: using JDBC APIs, and using JPA APIs.

JDBC4 has been part of JDK since version 1.1. It provides uniform APIs to access different relational databases. Between JDBC APIs and the database sits the JDBC driver for that database (either provided by the vendor of the database or a third-party vendor). JDBC translates common API calls to database-specific calls. The results returned from the database are also converted into objects of common data access classes. Although JDBC APIs require you to write a lot more code to access the database, it is still popular in JEE web applications because of its simplicity, flexibility of using database-specific SQL statements, and low learning curve.

JPA is the result of **Java Specification Request** 220 (which stands for **JSR**). One of the problems of using JDBC APIs directly is converting object representation of data to relation data. Object representation is in your JEE application, which needs to be mapped to tables and columns in the relational database. The process is reversed when handling data returned from the relational database. If there is a way to automatically map object-oriented representation of data in web applications to relational data, it would save a lot of developer time. This is also called **object-relational mapping** (**ORM**). Hibernate (`http://hibernate.org/`) is a very popular framework for ORM in Java applications.

Many of the concepts of such popular third-party ORM frameworks were incorporated in JPA. Just as JDBC provides uniform APIs for accessing relational databases, JPA provides uniform APIs for accessing ORM libraries. Third-party ORM frameworks provide implementations of JPA on top of their own framework. The JPA implementation may use the JDBC APIs underneath.

We will explore many features of JDBC and JPA in this chapter as we build applications using these frameworks. In fact, we will build the same application, once using JDBC and then using JPA.

The application that we are going to build is for student-course management. The goal is to take an example that can show how to model relationships between tables and use them in JEE applications. We will use a MySQL database and Tomcat web application container. Although this chapter is about database programming in JEE, we will revisit some of the things we learned about JSTL and JSF in `Chapter 2`, *Creating a Simple JEE Web Application*. We will use them to create user interfaces for our database web application. Make sure that you have configured Tomcat in Eclipse as described in `Chapter 2`, *Creating a Simple JEE Web Application*.

We will cover the following topics:

- Core JDBC concepts
- Using JDBC to access the database
- Using JDBC connection pool
- Core JPA concepts
- Using JPA to map entities (classes) to tables in the database
- Configuring relationships between JPA entities

Let's first create a database and tables for this application.

Creating database schema

There are many ways of creating database tables and relationships in MySQL:

- You can use **data description language** (**DDL**) statements directly at MySQL Command Prompt from the Terminal
- You can use MySQL Workbench and create tables directly
- You can create an entity-relationship diagram in MySQL Workbench, export it to create a DDL script, and then run this script to create tables and relationships

We will use the third option. If you just want to get the script to create tables and want to skip creating the ER diagram, then jump to the *Script to create tables and relationships* section of this chapter.

If you have not already installed MySQL and MySQL Workbench, then refer to `Chapter 1`, *Introducing JEE and Eclipse*, for instructions:

1. Open MySQL Workbench. Select the File | New Model menu. A blank model will be created with the option to create ER diagrams:

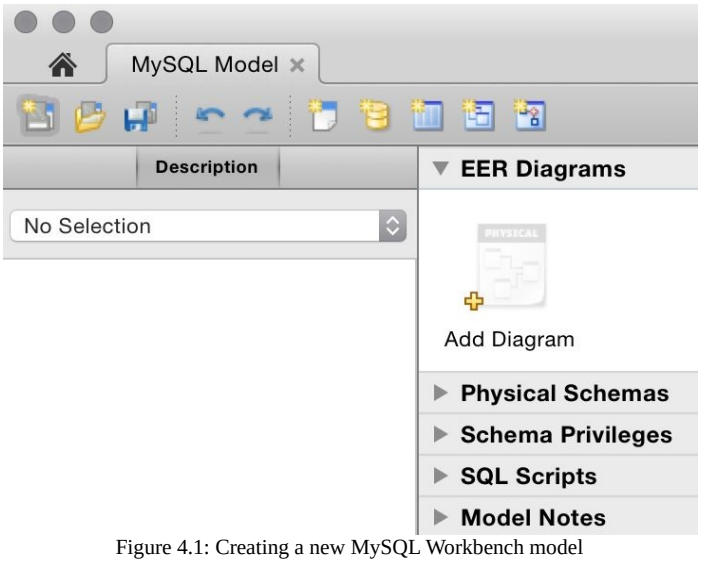


Figure 4.1: Creating a new MySQL Workbench model

2. Double-click the Add Diagram icon; a blank ER diagram will be opened:

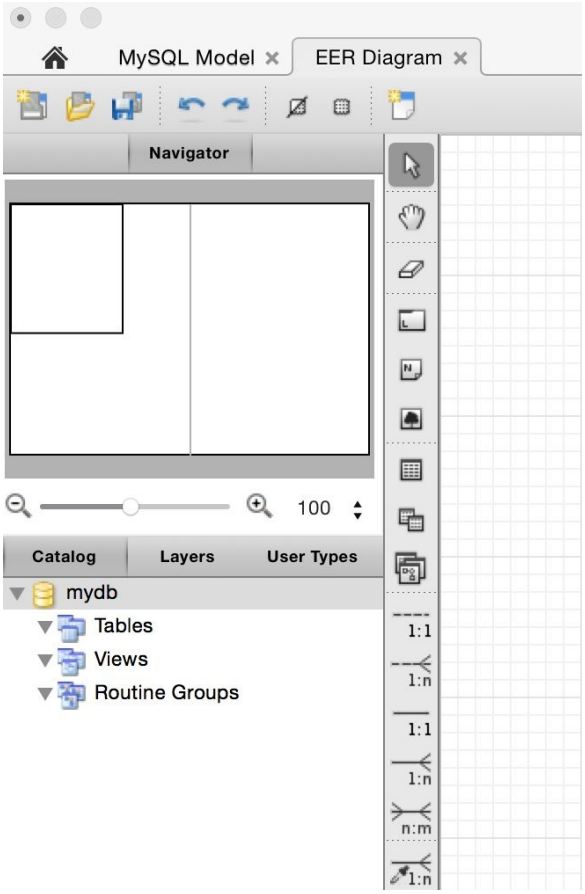


Figure 4.2: Creating a new ER diagram

3. By default, the new schema is named `mydb`. Double-click on it to open properties of the schema. Rename the schema`course_management`:

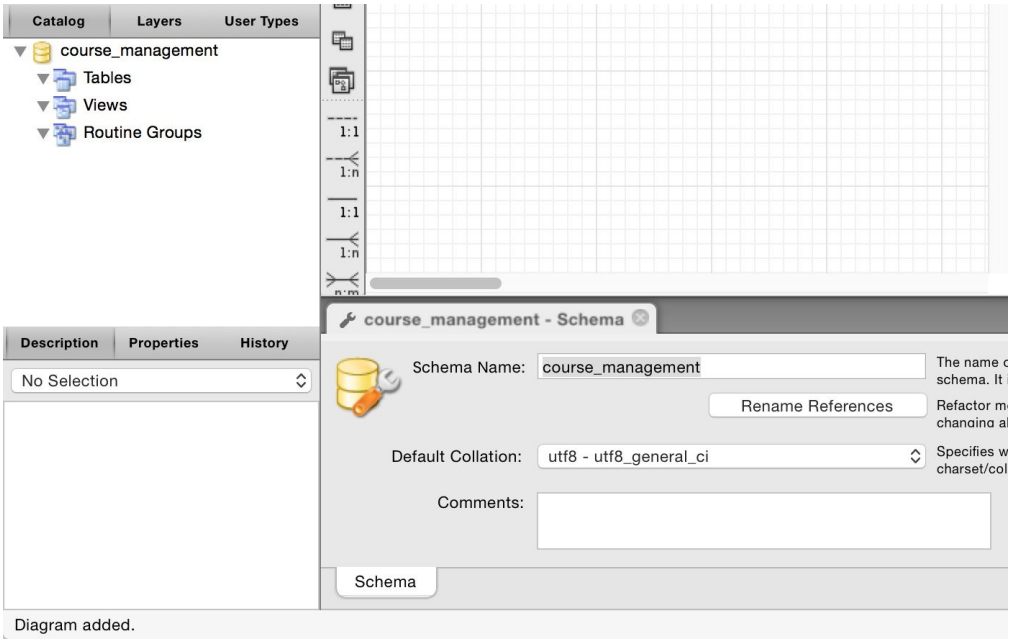


Figure 4.3: Renaming the schema

4. Hover over the toolbar buttons on the left-hand side of the page, and you will see tool tips about their functions. Click on the button for a new table and then click on the blank page. This will insert a new table with the name `table1`. Double-click the table icon to open the Properties page of the table. In the Properties page, change the name of the table to `Course`:

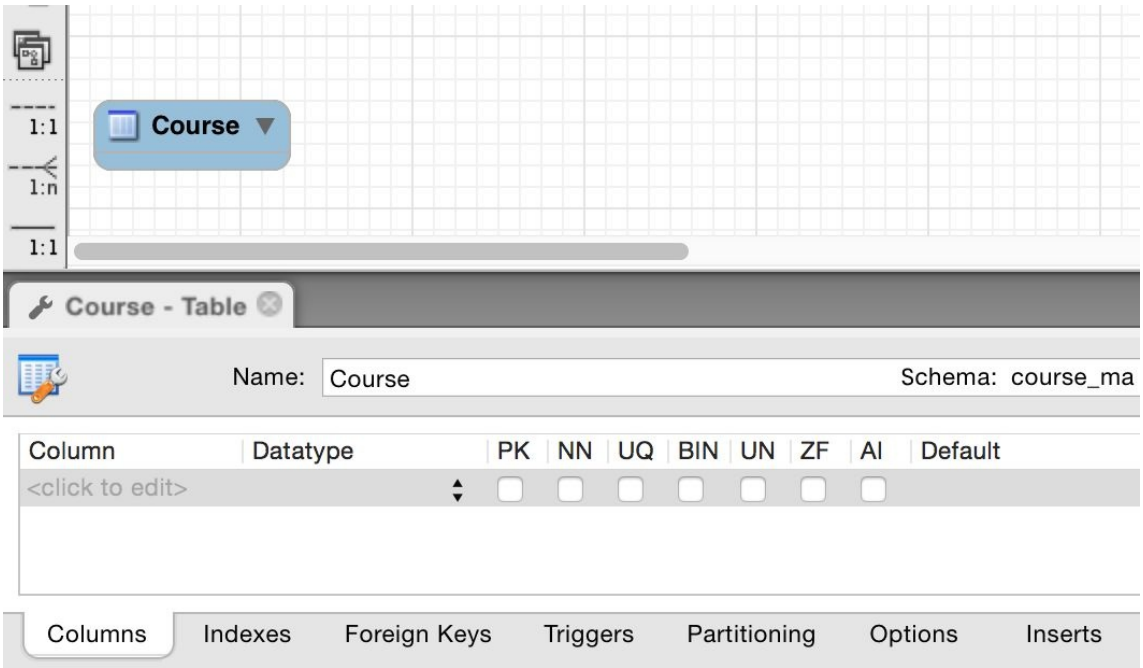


Figure 4.4: Creating a table in ER diagram

5. We will now create columns of the table. Double-click on the first column and name it id. Check the PK (**primary key**), NN (**not null**), and AI (**auto increment**) checkboxes. Add other columns as shown in the following screenshot:

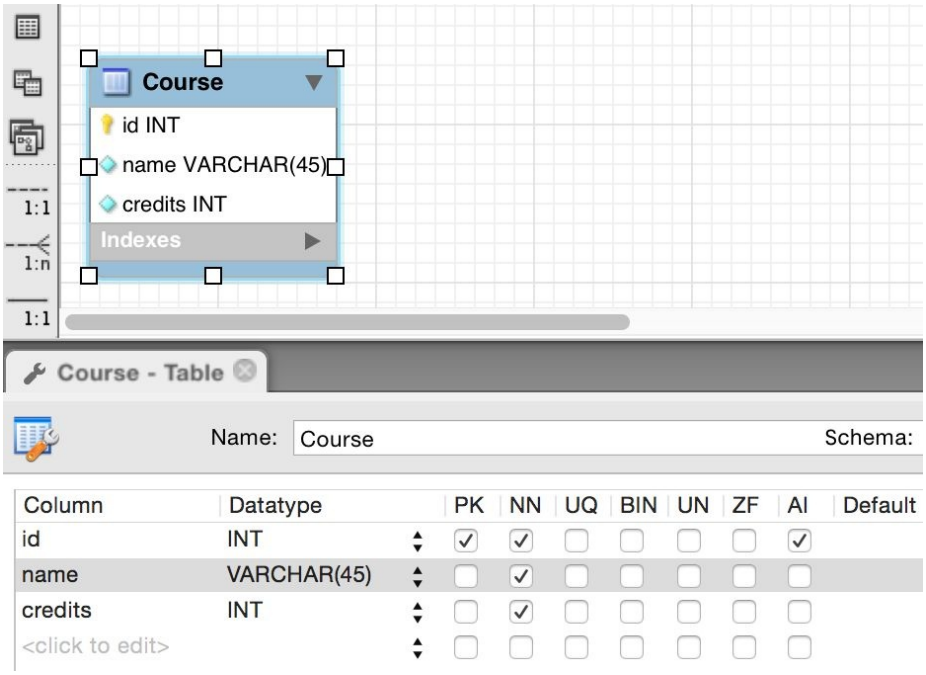


Figure 4.5: Creating columns in a table in the ER diagram

6. Create other tables, namely `Student` and `Teacher`, as shown in the following screenshot:

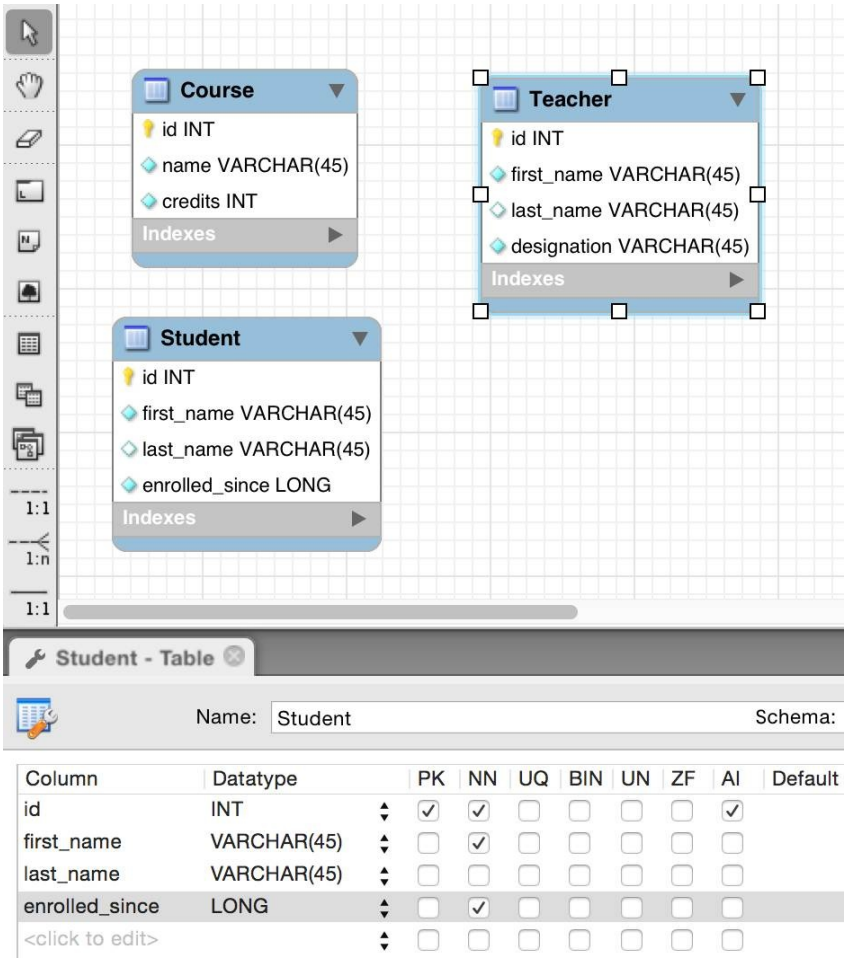


Figure 4.6: Creating additional tables

Note that if you want to edit column properties of any table, then double-click the table in the ER diagram. Just selecting a table by a single click would not change the table selection in the Properties page. All columns in all tables are required (not null), except the `last_name` column in `Student` and `Teacher` tables.

We will now create relationships between the tables. One course can have many students, and students can take many courses. So, there is a many-to-many relationship between `Course` and `Student`.

We will assume that one course is taught by only one teacher. However, a teacher can teach more than one course. Therefore, there is a many-to-one relationship between `Course` and `Teacher`.

Let's now model these relationships in the ER diagram:

1. First, we will create a non-identifying relationship between `Course` and `Teacher`.
2. Click on the non-identifying one-to-many button in the toolbar (dotted lines and 1:n).
3. Then, click on the `Course` table first and then on the `Teacher` table. It will create a relationship as shown in *Figure 4.7*. Note that a foreign key `Teacher_id` is created in the `Course` table. We don't want to make a `Teacher_id` field required in `Course`. A course can exist without a teacher in our application. Therefore, double-click on the link joining `Course` and `Teacher` tables.
4. Then, click on the Foreign Key tab.

5. On the Referenced Table side, uncheck the Mandatory checkbox:

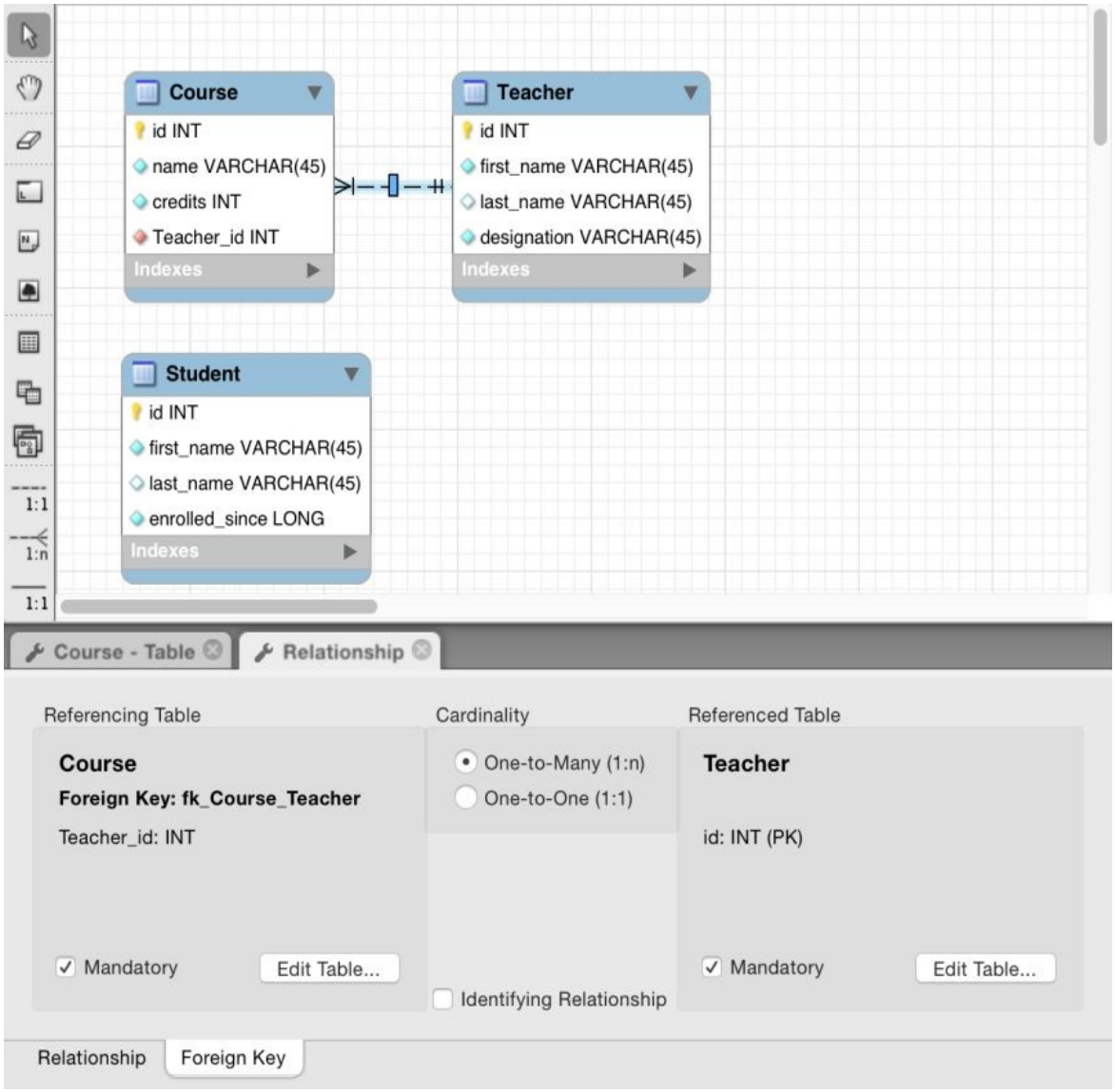


Figure 4.7: Creating a one-to-many relationship between tables

Creation of a many-to-many relationship requires a link table to be created. To create a many-to-many relationship between `Course` and `Student`, click on the icon for many-to-many (n:m) and then click on the `Course` table and `Student` table. This will create a third table (link table) called `Course_has_Student`. We will rename this table `Course_Student`. The final diagram is as shown in the following screenshot:

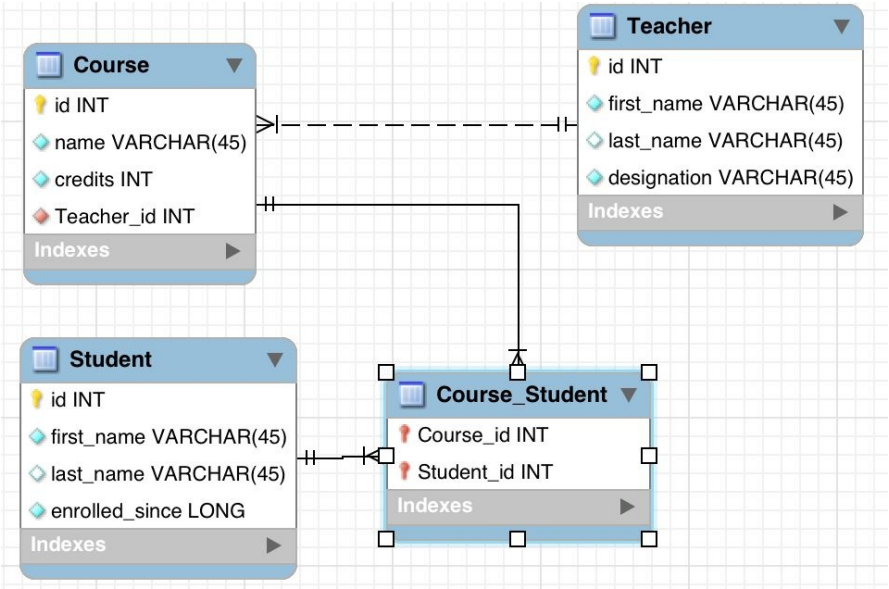


Figure 4.8: ER diagram for the course management example

Follow these steps to create DDL scripts from the ER diagram:

1. Select the File | Export | Forward Engineer SQL Create Script... menu.
2. On the SQL Export Options page, select checkboxes for two options:
    - Generate DROP Statements Before Each CREATE Statement
    - Generate DROP SCHEMA
3. Specify the Output SQL Script File path if you want to save the script.
4. On the last page of the Export wizard, you will see the script generated by MySQL Workbench. Copy this script by clicking the Copy to Clipboard button.

Script to create tables and relationships

The following is the DDL script to create tables and relationships for the course management example: -- MySQL Script generated by MySQL Workbench -- Sun Mar 8 18:17:07 2015

-- Model: New Model Version: 1.0

-- MySQL Workbench Forward Engineering

SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0; SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0; SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='TRADITIONAL,ALLOW_INVALID_DATES';

-- ---

-- Schema course_management

-- ---

DROP SCHEMA IF EXISTS `course_management` ;

-- ---

-- Schema course_management

-- ---

CREATE SCHEMA IF NOT EXISTS `course_management` DEFAULT CHARACTER SET utf8 COLLATE utf8_general_ci ; USE `course_management` ;

```
-- -----------------------------------------------------
-- Table `course_management`.`Teacher`
-- -----------------------------------------------------
DROP TABLE IF EXISTS `course_management`.`Teacher` ;
CREATE TABLE IF NOT EXISTS `course_management`.`Teacher` (
`id` INT NOT NULL AUTO_INCREMENT,
`first_name` VARCHAR(45) NOT NULL, `last_name` VARCHAR(45) NULL,
`designation` VARCHAR(45) NOT NULL, PRIMARY KEY (`id`))
ENGINE = InnoDB;


-- -----------------------------------------------------
-- Table `course_management`.`Course`
-- -----------------------------------------------------
DROP TABLE IF EXISTS `course_management`.`Course` ;
CREATE TABLE IF NOT EXISTS `course_management`.`Course` (
`id` INT NOT NULL AUTO_INCREMENT,
`name` VARCHAR(45) NOT NULL,
`credits` INT NOT NULL,
`Teacher_id` INT NULL,
```

```
PRIMARY KEY (`id`),

INDEX `fk_Course_Teacher_idx` (`Teacher_id` ASC), CONSTRAINT
`fk_Course_Teacher`

FOREIGN KEY (`Teacher_id`)

REFERENCES `course_management`.`Teacher` (`id`) ON DELETE NO
ACTION

ON UPDATE NO ACTION)

ENGINE = InnoDB;



-- -----------------------------------------------------

-- Table `course_management`.`Student`

-- -----------------------------------------------------

DROP TABLE IF EXISTS `course_management`.`Student` ;

CREATE TABLE IF NOT EXISTS `course_management`.`Student` (

`id` INT NOT NULL AUTO_INCREMENT,

`first_name` VARCHAR(45) NOT NULL, `last_name` VARCHAR(45) NULL,

`enrolled_since` MEDIUMTEXT NOT NULL, PRIMARY KEY (`id`))

ENGINE = InnoDB;



-- -----------------------------------------------------
```

```
-- Table `course_management`.`Course_Student`

-- -----------------------------------------------------

DROP TABLE IF EXISTS `course_management`.`Course_Student` ;

CREATE TABLE IF NOT EXISTS `course_management`.`Course_Student` (

`Course_id` INT NOT NULL,

`Student_id` INT NOT NULL,

PRIMARY KEY (`Course_id`, `Student_id`), INDEX
`fk_Course_has_Student_Student1_idx` (`Student_id` ASC), INDEX
`fk_Course_has_Student_Course1_idx` (`Course_id` ASC), CONSTRAINT
`fk_Course_has_Student_Course1`

FOREIGN KEY (`Course_id`)

REFERENCES `course_management`.`Course` (`id`) ON DELETE NO
ACTION

ON UPDATE NO ACTION,

CONSTRAINT `fk_Course_has_Student_Student1`

FOREIGN KEY (`Student_id`)

REFERENCES `course_management`.`Student` (`id`) ON DELETE NO
ACTION

ON UPDATE NO ACTION)

ENGINE = InnoDB;



SET SQL_MODE=@OLD_SQL_MODE;
```

SET FOREIGN_KEY_CHECKS=@OLD_FOREIGN_KEY_CHECKS; SET UNIQUE_CHECKS=@OLD_UNIQUE_CHECKS;

Creating tables in MySQL

Let's now create tables and relationships in the MySQL database by using the script created in the previous section.

Make sure that MySQL is running and there is an open connection to the server from MySQL Workbench (see `Chapter 1`, *Introducing JEE and Eclipse*, for more details):

1. Create a new query tab (the first button in the toolbar) and paste the preceding script.
2. Execute the query.
3. At the end of the execution, refresh schemas in the left-hand pane. You should see the course_management schema and the tables created in it.

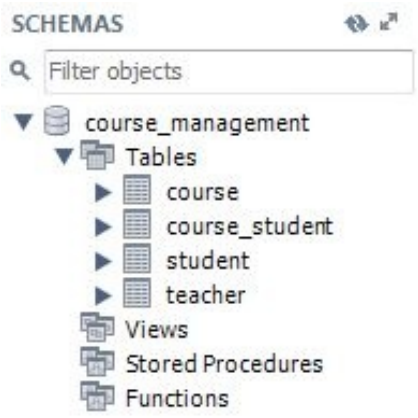


Figure 4.9: MySQL schema for the course management example

Creating a database application using JDBC

In this section, we will use JDBC to create a simple course management web application. We will use the MySQL schema created in the previous section. Furthermore, we will create the web application using Tomcat; we have already seen how to create one in `Chapter 2`, *Creating a Simple JEE Web Application*. We have also learned how to use JSTL and JSF in the same chapter. In this section, we will use JSTL and JDBC to create the course management application, and in the next section, we will use JSF and JPA to create the same application. We will use Maven (as described in `Chapter 2`, *Creating a Simple JEE Web Application*) for project management, and of course, our IDE is going to be Eclipse JEE.

Creating a project and setting up Maven dependencies

We will perform the following steps to create the Maven project for our application:

1. Create a Maven web project as described in `Chapter 2`, *Creating a Simple JEE Web Application*.
2. Name the project `CourseManagementJDBC`.
3. Add dependencies for servlet and JSP, but do not add a dependency for JSF.

4. To add the dependency for JSTL, open `pom.xml` and go to the Dependencies tab. Click on the Add... button. Type `javax.servlet` in the search box and select jstl:

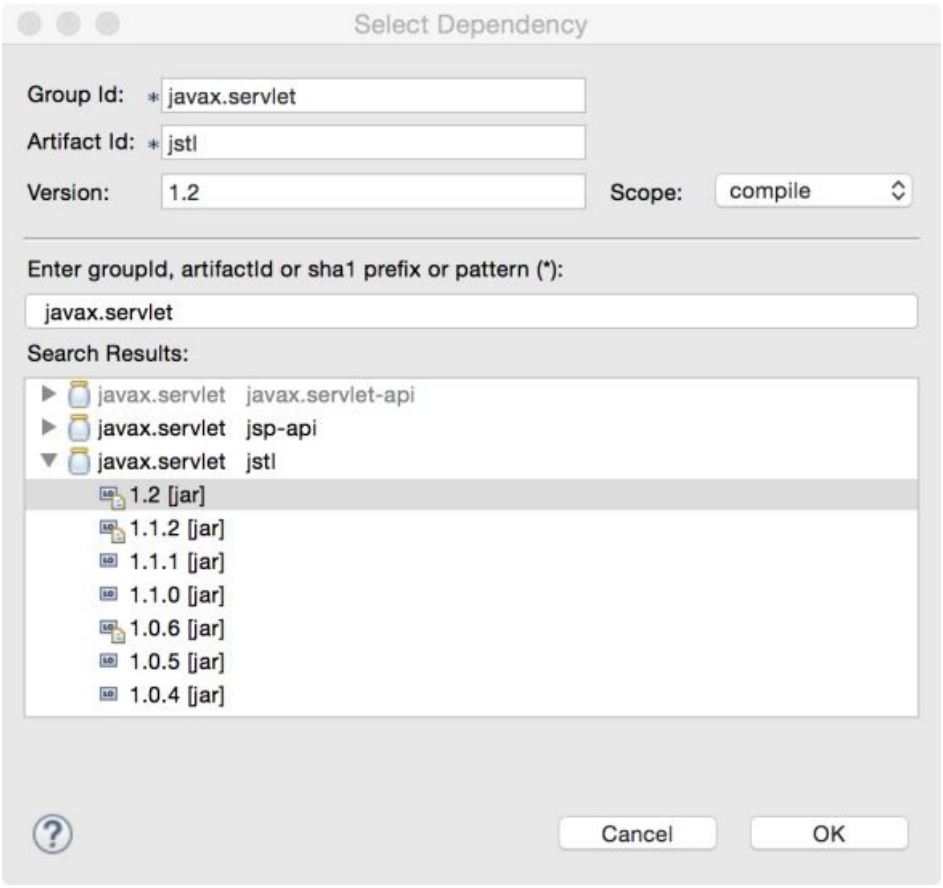


Figure 4.10: Adding a dependency for jstl

5. Add the dependency for the MySQL JDBC driver too:

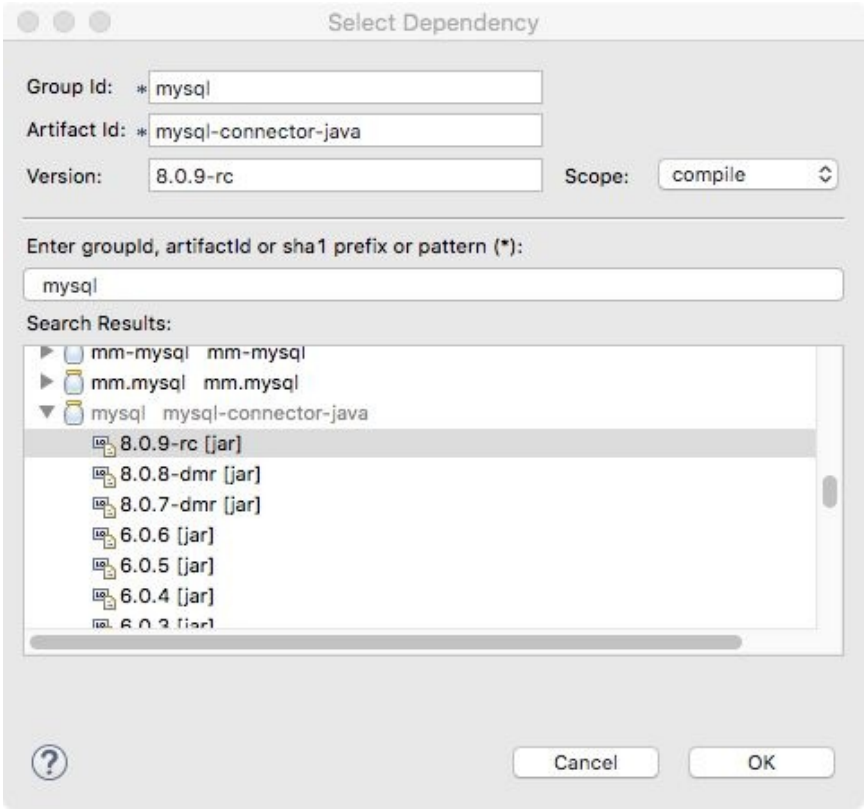


Figure 4.11: Adding dependency for the MySQL JDBC driver

Here is the `pom.xml` file after adding dependencies:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>packt.book.jee.eclipse</groupId>
  <artifactId>CourseManagementJDBC</artifactId>
  <version>1</version>
  <packaging>war</packaging>
  <dependencies>
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>javax.servlet-api</artifactId>
      <version>3.1.0</version>
<scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>jstl</artifactId>
      <version>1.2</version>
    </dependency>
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>8.0.9-rc</version>
    </dependency>
```

```
    <dependency>
      <groupId>javax.servlet.jsp</groupId>
      <artifactId>jsp-api</artifactId>
      <version>2.2</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>
</project>
```

Note that the dependencies for servlet and JSP are marked as provided, which means that they will be provided by the web container (Tomcat) and will not be packaged with the application.

The description of how to configure Tomcat and add a project to it is skipped here. Refer to `Chapter 2`, *Creating a Simple JEE Web Application*, for these details. This section will also not repeat information on how to run JSP pages and about JSTL that were covered in `Chapter 2`, *Creating a Simple JEE Web Application.*

Creating JavaBeans for data storage

We will first create JavaBean classes for `Student`, `Course`, and `Teacher`. Since both student and teacher are people, we will create a new class called `Person` and have `Student` and `Teacher` classes extend it. Create these JavaBeans in the `packt.book.jee.eclipse.ch4.beans` package as follows.

The code for the `Course` bean will be as follows: package packt.book.jee.eclipse.ch4.bean;

public class Course {

private int id;

private String name;

private int credits;

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getName() {

return name;

}

```
public void setName(String name) {

this.name = name;

}

public int getCredits() {

return credits;

}

public void setCredits(int credits) {

this.credits = credits; }

}
```

The code for the `Person` bean will be as follows: package packt.book.jee.eclipse.ch4.bean;

```
public class Person {

private int id;

private String firstName; private String lastName;

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getFirstName() {
```

return firstName;

}

public void setFirstName(String firstName) {

this.firstName = firstName; }

public String getLastName() {

return lastName;

}

public void setLastName(String lastName) {

this.lastName = lastName; }

}

The code for the `Student` bean will be as follows: package packt.book.jee.eclipse.ch4.bean;

public class Student extends Person {

private long enrolledsince;

public long getEnrolledsince() {

return enrolledsince; }

public void setEnrolledsince(long enrolledsince) {

this.enrolledsince = enrolledsince; }

}

The `Teacher` bean will be as follows: package packt.book.jee.eclipse.ch4.bean;

```
public class Teacher extends Person {
private String designation;
public String getDesignation() {
return designation;
}

public void setDesignation(String designation) {
this.designation = designation; }
}
```

Creating JSP to add a course

Let's now create a JSP page to add new courses. Right-click on the project in Package Explorer and select the New | Other... option. Type `jsp` in the filter box and select JSP File. Name the file `addCourse.jsp`. Eclipse will create the file in the `src/main/webapp` folder of the project.

Type the following code in `addCourse.jsp`: <%@ page language="java" contentType="text/html; charset=UTF-8"

pageEncoding="UTF-8"%>

<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd"> <html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=UTF-8"> <title>Add Course</title> </head>

<body>

<c:set var="errMsg" value="${null}"/> <c:set var="displayForm" value="${true}"/> <c:if test="${\"POST\".equalsIgnoreCase(pageContext.request.method) && pageContext.request.getParameter(\"submit\") != null}"> <jsp:useBean id="courseBean" class="packt.book.jee.eclipse.ch4.bean.Course"> <c:catch var="beanStorageException"> <jsp:setProperty name="courseBean" property="*" /> </c:catch>

</jsp:useBean>

<c:choose>

<c:when test="${!courseBean.isValidCourse() || beanStorageException != null}"> <c:set var="errMsg" value="Invalid course details. Please try again"/> </c:when>

<c:otherwise>

<c:redirect url="listCourse.jsp"/> </c:otherwise>

</c:choose>

</c:if>

<h2>Add Course:</h2>

<c:if test="${errMsg != null}"> <span style="color: red;"> <c:out value="${errMsg}"></c:out>

</c:if>

<form method="post">

Name: <input type="text" name="name">
 Credits : <input type="text" name="credits">
 <button type="submit" name="submit">Add</button> </form>

</body>

</html>

Most of the code should be familiar, if you have read `Chapter 2`, *Creating a Simple JEE Web Application* (see the *Using JSTL* section). We have a form to add courses. At the top of the file, we check whether the `post` request is made; if so, store content of the form in `courseBean` (make sure that names of the `form` field are the same as the members defined in the bean). The new tag that we have used here is `<c:catch>`. It is like a *try-catch* block in Java. Any exception thrown from within the body of `<c:catch>` is assigned to the variable name declared in the `var`

attribute. Here, we are not doing anything with `beanStorageException`; we are suppressing the exception. When an exception is thrown, the `credits` field of the `Course` bean will remain set to zero and it will be caught in the `courseBean.isValidCourse` method. If the course data is valid, then we redirect the request to the `listCourse.jsp` page using the JSTL `<c:redirect>` tag.

We need to add the `isValidCourse` method in the `Course` bean. Therefore, open the class in the editor and add the following method: public boolean isValidCourse() {

return name != null && credits != 0; }

We also need to create `listCourse.jsp`. For now, just create a simple JSP with no JSTL/Java code and with only one header in the `body` tag: <h2>Courses:</h2>

Right-click on `addCourse.jsp` in Package Explorer and select Run As | Run on Server. If you have configured Tomcat properly and added your project in Tomcat (as described in `Chapter 2`, *Creating a Simple JEE Web Application*), then you should see the JSP page running in the internal Eclipse browser. Test the page with both valid and invalid data (a wrong credit value; for example, a non-numeric value). If the data entered is valid, then you would be redirected to `listCourse.jsp`, or else the same page would be displayed with the error message.

Before we start writing JDBC code, let's learn some fundamental concepts of JDBC.

JDBC concepts

Before performing any operations in JDBC, we need to establish a connection to the database. Here are some of the important classes/interfaces in JDBC for executing SQL statements:

JDBC class/interface	Description
`java.sql.Connection`	Represents the connection between the application and the backend database. Must for performing any action on the database.
`java.sql.DriverManager`	Manages JDBC drivers used in the application. Call the `DriverManager.getConnection` static method to obtain the connection.
`java.sql.Statement`	Used for executing static SQL statements.
`java.sql.PreparedStatement`	Used for preparing parameterized SQL statements. SQL statements are pre-compiled and can be executed repeatedly with different parameters.
`Java.sqlCallableStatement`	Used for executing a stored procedure.
`java.sql.ResultSet`	Represents a row in the database table in the result returned after execution of an SQL query by `Statement` or `PreparedStatement`.

You can find all the interfaces for JDBC at `http://docs.oracle.com/javase/8/docs/api/java/sql/package-frame.html`.

Many of these are interfaces, and implementations of these interfaces are provided by the JDBC drivers.

Creating database connections

Make sure that the JDBC driver for the database you want to connect to is downloaded and is in the classpath. In our project, we have already ensured this by adding a dependency in Maven. Maven downloads the driver and adds it to the class path of our web application.

It is always a good practice to make sure that the JDBC driver class is available when the application is running. If it is not, we can set a suitable error message and not perform any JDBC operations. The name of the MySQL JDBC driver class is `com.mysql.cj.jdbc.Driver`: try { Class.forName("com.mysql.cj.jdbc.Driver"); } catch (ClassNotFoundException e) { //log excetion //either throw application specific exception or return return; }

Then, get the connection by calling the `DriverManager.getConnection` method:

```
try {
  Connection con =
 DriverManager.getConnection("jdbc:mysql://localhost:3306/schema_name?" +
      "user=your_user_name&password=your_password");
//perform DB operations and then close the connection
  con.close();
}
catch (SQLException e) {
  //handle exception
}
```

The argument to `DriverManager.getConnection` is called a connection URL. It is specific to the JDBC driver. So, check the documentation of the JDBC driver to understand how to create a connection URL. The URL format in the preceding code snippet is for MySQL. See https://dev.mysql.com/doc/connector-j/8.0/en/connector-j-reference-configuration-properties.html.

The connection URL contains the following details: hostname of the MySQL database server, port on which it is running (default is 3306), and the schema name (database name that you want to connect to). You can pass username and password to connect to the database as URL parameters.

Creating a connection is an expensive operation. Also, database servers allow a certain maximum number of connections to it, so connections should be created

sparingly. It is advisable to cache database connections and reuse. However, make sure that you close the connection when you no longer need it, for example, in the `final` blocks of your code. Later, we will see how to create a pool of connections so that we create a limited number of connections, take them out of the pool when required, perform the required operations, and return them to the pool so that they can be reused.

Executing SQL statements

Use `Statement` for executing static SQL (having no parameters) and `PreparedStatement` for executing parameterized statements.

To avoid the risk of SQL injection, refer to `https://www.owasp.org/index.php/SQL_injection`.

To execute any `Statement`, you first need to create the statement using the `Connection` object. You can then perform any SQL operation, such as `create`, `update`, `delete`, and `select`. The `Select` statement (query) returns a `ResultSet` object. Iterate over the `ResultSet` object to get individual rows.

For example, the following code gets all rows from the `Course` table: Statement stmt = null; ResultSet rs = null;

try {

stmt = con.createStatement(); rs = stmt.executeQuery("select * from Course");

List<Course> courses = new ArrayList<Course>(); //Depending on the database that you connect to, you may have to //call rs.first() before calling rs.next(). In the case of a MySQL

//database, it is not necessary to call rs.first() while (rs.next()) {

Course course = new Course(); course.setId(rs.getInt("id")); course.setName(rs.getString("name")); course.setCredits(rs.getInt("credits")); courses.add(course); }

}

catch (SQLException e) {

//handle exception

```
e.printStackTrace();

}

finally {

try {

if (rs != null)

rs.close();

if (stmt != null)

stmt.close();

}

catch (SQLException e) {

//handle exception

}

}
```

Things to note:

- Call `Connection.createStatement ()` to create an instance of `Statement`.
- `Statement.executeQuery` returns `ResultSet`. If the SQL statement is not a query, for example `create`, `update`, and `delete` statements, then call `Statement.execute` (which returns `true` if the statement is executed successfully; or else, `false`) or call `Statement.executeUpdate` (which returns the number of rows affected or zero if none is affected).
- Pass the SQL statement to the `Statement.executeQuery` function. This can be any valid SQL string understood by the database.
- Iterate over `ResultSet` by calling the `next` method, until it returns `false`.
- Call different variations of `get` methods (depending on the data type of the column) to obtain values of columns in the current row that the `ResultSet` is pointing to. You can either pass positional index of the column in SQL that

you passed to `executeQuery` or column names as used in the database table or alias specified in the SQL statement. For example, we would use the following code if we had specified column names in the SQL:

```
rs = stmt.executeQuery("select id, name, credits as courseCredit from Course");
```

Then, we could retrieve column values as follows: course.setId(rs.getInt(1)); course.setName(rs.getString(2)); course.setCredits(rs.getInt("courseCredit"));

- Make sure you close `ResultSet` and `Statement`.

Instead of getting all courses, if you want to get a specific course, you would want to use `PreparedStatement`: PreparedStatement stmt = null; int courseId = 10;

ResultSet rs = null;

try {

stmt = con.prepareStatement("select * from Course where id = ?"); stmt.setInt(1, courseId); rs = stmt.executeQuery();

Course course = null;

if (rs.next()) {

course = new Course(); course.setId(rs.getInt("id")); course.setName(rs.getString("name")); course.setCredits(rs.getInt("credits")); }

}

catch (SQLException e) {

//handle exception

e.printStackTrace();

}

finally {

```
try {
if (rs != null)
rs.close();
if (stmt != null
stmt.close();
}
catch (SQLException e) {
//handle exception
}
}
```

In this example, we are trying to get the course with ID `10`. We first get an instance of `PreparedStatement` by calling `Connection.prepareStatement`. Note that you need to pass an SQL statement as an argument to this function. Parameters in the query are replaced by the `?` placeholder. We then set the value of the parameter by calling `stmt.setInt`. The first argument is the position of the parameter (it starts from `1`) and the second argument is the value. There are many variations of the `set` method for different data types.

Handling transactions

If you want to perform multiple changes to the database as a single unit, that is, either all changes should be done or none, then you need to start a transaction in JDBC. You start a transaction by calling `Connection. setAutoCommit(false)`. Once all operations are executed successfully, commit the changes to the database by calling `Connection.commit`. If for any reason you want to abort the transaction, call `Connection.rollback()`. Changes are not done in the database until you call `Connection.commit`.

Here is an example of inserting a bunch of courses into the database. Although in a real application, it may not make sense to abort a transaction when one of the courses is not inserted, here we assume that either all courses must be inserted into the database or none:

```
PreparedStatement stmt = con.prepareStatement("insert into Course (id, name, credits)
values (?,?,?)");




con.setAutoCommit(false);

try {

  for (Course course : courses) {

    stmt.setInt(1, course.getId());

    stmt.setString(2, course.getName());

    stmt.setInt(3, course.getCredits());

    stmt.execute();
```

```
  }

  //commit the transaction now

  con.commit();

}

catch (SQLException e) {

  //rollback commit

  con.rollback();

}
```

There is more to learn about transactions than explained here. Refer to Oracle's JDBC tutorial at `http://docs.oracle.com/javase/tutorial/jdbc/basics/transactions.html`.

Using a JDBC database connection pool

As mentioned before, a JDBC database connection is an expensive operation and connection objects should be reused. Connection pools are used for this purpose. Most web containers provide their own implementation of a connection pool along with ways to configure it using JNDI. Tomcat also lets you configure a connection pool using JNDI. The advantage of configuring a connection pool using JNDI is that the database configuration parameters, such as hostname and port, remain outside the source code and can be easily modified. See `http://tomcat.apache.org/tomcat-8.0-doc/jdbc-pool.html`.

However, a Tomcat connection pool can also be used without JNDI, as described in the preceding link. In this example, we will use a connection pool without JNDI. The advantage is that you can use the connection pool implementation provided by a third party; your application then becomes easily portable to other web containers. With JNDI, you can also port your application, as long as you create the JNDI context and resources in the web container that you are switching to.

We will add the dependency of the Tomcat connection pool library to Maven's `pom.xml`. Open the `pom.xml` file and add the following dependencies (see `Chapter 2`, *Creating a Simple JEE Web Application*, to know how to add dependencies to Maven): <dependency> <groupId>org.apache.tomcat</groupId> <artifactId>tomcat-jdbc</artifactId> <version>9.0.6</version> </dependency>

Note that you can use any other implementation of the JDBC connection pool. One such connection pool library is HikariCP (`https://github.com/brettwooldridge/HikariCP`).

We also want to move the database properties out of the code. Therefore, create a file called `db.properties` in `src/main/resources`. Maven puts all files in this folder in the classpath of the application. Add the following properties in `db.properties`:

db_host=localhost db_port=3306

db_name=course_management

db_user_name=your_user_name

db_password=your_password

db_driver_class_name=com.mysql.cj.jdbc.Driver

We will create a singleton class to create JDBC connections using the Tomcat connection pool. Create a `packt.book.jee.eclipse.ch4.db.connection` package and create a `DatabaseConnectionFactory` class in it: package packt.book.jee.eclipse.ch4.db.connection;

// skipping imports to save space here

/**

* Singleton Factory class to create JDBC database connections *

*/

public class DatabaseConnectionFactory {

//singleton instance

private static DatabaseConnectionFactory conFactory = new DatabaseConnectionFactory();

private DataSource dataSource = null;

//Make the construction private

private DatabaseConnectionFactory() {}

/**

* Must be called before any other method in this class.

```
* Initializes the data source and saves it in an instance
variable *

* @throws IOException

*/

public synchronized void init() throws IOException {

//Check if init was already called if (dataSource != null)

return;



//load db.properties file first

InputStream inStream =
this.getClass().getClassLoader().getResourceAsStream("db.properties");
Properties dbProperties = new Properties(); dbProperties.load(inStream);

inStream.close();



//create Tomcat specific pool properties PoolProperties p = new PoolProperties();
p.setUrl("jdbc:mysql://" + dbProperties.getProperty("db_host") +

":" + dbProperties.getProperty("db_port") + "/" +

dbProperties.getProperty("db_name"));

p.setDriverClassName(dbProperties.getProperty("db_driver_class_name"));
p.setUsername(dbProperties.getProperty("db_user_name"));
p.setPassword(dbProperties.getProperty("db_password")); p.setMaxActive(10);



dataSource = new DataSource();

dataSource.setPoolProperties(p);
```

}

//Provides access to singleton instance public static DatabaseConnectionFactory getConnectionFactory() {

return conFactory;

}

//returns database connection object public Connection getConnection () throws SQLException {

if (dataSource == null)

throw new SQLException("Error initializing datasource"); return dataSource.getConnection(); }

}

We must call the `init` method of `DatabaseConnectionFactory` before getting connections from it. We will create a servlet and load it on startup. Then, we will call `DatabaseConnectionFactory.init` from the `init` method of the servlet.

Create `package packt.book.jee.eclipse.ch4.servlet` and then create an `InitServlet` class in it: package packt.book.jee.eclipse.ch4.servlet; import java.io.IOException;

import javax.servlet.ServletConfig;

import javax.servlet.ServletException; import javax.servlet.annotation.WebServlet; import javax.servlet.http.HttpServlet;

import packt.book.jee.eclipse.ch4.db.connection.DatabaseConnectionFactory;

@WebServlet(value="/initServlet", loadOnStartup=1) public class InitServlet extends HttpServlet {

private static final long serialVersionUID = 1L;

```
public InitServlet() {

super();

}


public void init(ServletConfig config) throws ServletException {

try {

DatabaseConnectionFactory.getConnectionFactory().init(); }

catch (IOException e) {

config.getServletContext().log(e.getLocalizedMessage(),e); }

}

}
```

Note that we have used the `@WebServlet` annotation to mark this class as a servlet and the `loadOnStartup` attribute is set to `1`, to tell the web container to load this servlet on startup.

Now we can call the following statement to get a `Connection` object from anywhere in the application: Connection con = DatabaseConnectionFactory.getConnectionFactory().getConnection();

If there are no more connections available in the pool, then the `getConnection` method throws an exception (in particular, in the case of the `Tomcat` datasource, it throws `PoolExhaustedException`). When you close the connection that was obtained from the connection pool, the connection is returned to the pool for reuse.

Saving courses in database tables using JDBC

Now that we have figured out how to use the JDBC connection pool and get a connection from it, let's write the code to save a course to the database.

We will create **Course Data Access Object** (**CourseDAO**), which will have functions required to directly interact with the database. We are thus separating the code to access the database from the UI and business code.

Create `package packt.book.jee.eclipse.ch4.dao`. Create a class called `CourseDAO` in it: package packt.book.jee.eclipse.ch4.dao;

import java.sql.Connection;

import java.sql.PreparedStatement; import java.sql.ResultSet;

import java.sql.SQLException;

import java.sql.Statement;

import packt.book.jee.eclipse.ch4.bean.Course; import packt.book.jee.eclipse.ch4.db.connection.DatabaseConnectionFactory;

public class CourseDAO {

public static void addCourse (Course course) throws SQLException
{

//get connection from connection pool Connection con = DatabaseConnectionFactory.getConnectionFactory().getConnection(); try {

```
final String sql = "insert into Course (name, credits)
values (?,?)"; //create the prepared statement with an option to get auto-
generated keys PreparedStatement stmt = con.prepareStatement(sql,
Statement.RETURN_GENERATED_KEYS); //set parameters

stmt.setString(1, course.getName()); stmt.setInt(2, course.getCredits());

stmt.execute();



//Get auto-generated keys

ResultSet rs = stmt.getGeneratedKeys();

if (rs.next())

course.setId(rs.getInt(1));

rs.close();

stmt.close();

}

finally {

con.close();

}

}

}
```

We have already seen how to insert a record using JDBC. The only new thing in the preceding code is to get the autogenerated ID. Recall that the `id` column in the `Course` table is autogenerated. This is the reason that we did not specify it in the insert SQL: String sql = "insert into Course (name, credits) values (?,?)";

When we prepare a statement, we are telling the driver to get the autogenerated

ID. After the row is inserted into the table, we get the autogenerated ID by calling the following: ResultSet rs = stmt.getGeneratedKeys();

We have already created `addCourse.jsp`. Somehow `addCourse.jsp` needs to send the form data to `CourseDAO` in order to save the data in the database. `addCourse.jsp` already has access to the `Course` bean and saves the form data in it. So, it makes sense for the `Course` bean to interface between `addCourse.jsp` and `CourseDAO`. Let's modify the `Course` bean to add an instance of `CourseDAO` as a member variable and then create a function to add a course (instance of `CourseDAO`) to the database: public class Course {

....

private CourseDAO courseDAO = new CourseDAO();

...

public void addCourse() throws SQLException {

courseDAO.addCourse(this);

}

}

We will then modify `addCourse.jsp` to call the `addCourse` method of the `Course` bean. We will have to add this code after the form is submitted and the data is validated: <c:catch var="addCourseException"> ${courseBean.addCourse()}

</c:catch>

<c:choose>

<c:when test="${addCourseException != null}"> <c:set var="errMsg" value="${addCourseException.message}"/> </c:when>

<c:otherwise>

<c:redirect url="listCourse.jsp"/> </c:otherwise>

</c:choose>

One thing to note in the preceding code is the following statement: ${courseBean.addCourse()}

You can insert **Expression Language** (**EL**) in JSP as discussed previously. This method does not return anything (it is a void method). Therefore, we didn't use the `<c:set>` tag. Furthermore, note that the call is made within the `<c:catch>` tag. If any `SQLException` is thrown from the method, then it will be assigned to the `addCourseException` variable. We then check whether `addCourseException` is set in the `<c:when>` tag. If the value is not null, then it means that the exception was thrown. We set the error message, which is later displayed on the same page. If no error is thrown, then the request is redirected to `listCourse.jsp`. Here is the complete code of `addCourse.jsp`: <%@ page language="java" contentType="text/html; charset=UTF-8"

pageEncoding="UTF-8"%>

<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd"> <html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=UTF-8"> <title>Insert title here</title> </head>

<body>

<c:set var="errMsg" value="${null}"/> <c:set var="displayForm" value="${true}"/> <c:if test="${"POST".equalsIgnoreCase(pageContext.request.method) && pageContext.request.getParameter("submit") != null}"> <jsp:useBean id="courseBean" class="packt.book.jee.eclipse.ch4.bean.Course"> <c:catch

```
var="beanStorageException"> <jsp:setProperty name="courseBean"
property="*" /> </c:catch>

</jsp:useBean>

<c:choose>

<c:when test="${!courseBean.isValidCourse() ||
beanStorageException != null}"> <c:set var="errMsg" value="Invalid course
details. Please
try again"/> </c:when>

<c:otherwise>

<c:catch var="addCourseException"> ${courseBean.addCourse()}

</c:catch>

<c:choose>

<c:when test="${addCourseException != null}"> <c:set var="errMsg"
value="${addCourseException.message}"/> </c:when>

<c:otherwise>

<c:redirect url="listCourse.jsp"/> </c:otherwise>

</c:choose>

</c:otherwise>

</c:choose>

</c:if>



<h2>Add Course:</h2> <c:if test="${errMsg != null}"> <span style="color:
red;"> <c:out value="${errMsg}"></c:out> </span>

</c:if>
```

<form method="post">

Name: <input type="text" name="name">
 Credits : <input type="text" name="credits">
 <button type="submit" name="submit">Add</button> </form>

</body>

</html>

Run the page, either in Eclipse or outside (see Chapter 2, *Creating a Simple JEE Web Application*, to know how to run JSP in Eclipse and view it in Eclipse's internal browser) and add a couple of courses.

Getting courses from database tables using JDBC

We will now modify `listCourses.jsp` to display the courses that we have added using `addCourse.jsp`. However, we first need to add a method in `CourseDAO` to get all courses from the database.

Note that the `Course` table has a one-to-many relationship with `Teacher`. It stores the teacher ID in it. Further, the teacher ID is not a required field, so a course can exist in the `Course` table with null `teacher_id`. To get all the details of a course, we need to get the teacher for the course too. However, we cannot create a simple join in an SQL query to get the details of a course and of the teacher for each course, because a teacher may not have been set for the course. In such cases, we use the *left outer join*, which returns all records from the table on the left-hand side of the join, but only matching records from the table on the right-hand side of the join. Here is the SQL statement to get all courses and teachers for each course: select course.id as courseId, course.name as courseName, course.credits as credits, Teacher.id as teacherId, Teacher.first_name as firstName,Teacher.last_name as lastName, Teacher.designation designation

from Course left outer join Teacher on course.Teacher_id = Teacher.id

order by course.name

We will use the preceding query in `CourseDAO` to get all courses. Open the `CourseDAO` class and add the following method: public List<Course> getCourses () throws SQLException {

//get connection from connection pool Connection con = DatabaseConnectionFactory.getConnectionFactory().getConnection();

List<Course> courses = new ArrayList<Course>(); Statement stmt = null;

```
ResultSet rs = null;

try {

stmt = con.createStatement();



//create SQL statement using left outer join StringBuilder sb = new
StringBuilder("select course.id as
courseId, course.name as courseName,") .append("course.credits as credits,
Teacher.id as teacherId,
Teacher.first_name as firstName, ") .append("Teacher.last_name as lastName,
Teacher.designation
designation ") .append("from Course left outer join Teacher on ")
.append("course.Teacher_id = Teacher.id ") .append("order by course.name");

//execute the query

rs = stmt.executeQuery(sb.toString());

//iterate over result set and create Course objects //add them to course list

while (rs.next()) {

Course course = new Course(); course.setId(rs.getInt("courseId"));
course.setName(rs.getString("courseName"));
course.setCredits(rs.getInt("credits")); courses.add(course);



int teacherId = rs.getInt("teacherId"); //check whether teacher id was null in the
table if (rs.wasNull()) //no teacher set for this course.

continue;

Teacher teacher = new Teacher(); teacher.setId(teacherId);

teacher.setFirstName(rs.getString("firstName"));
teacher.setLastName(rs.getString("lastName"));
```

teacher.setDesignation(rs.getString("designation")); course.setTeacher(teacher);

}

return courses;

}

finally {

try {if (rs != null) rs.close();} catch (SQLException e) {}

try {if (stmt != null) stmt.close();} catch (SQLException e) {}

try {con.close();} catch (SQLException e) {}

}

}

We have used `Statement` to execute the query because it is a static query. We have used `StringBuilder` to build the SQL statement because it is a relatively large query (compared to those that we have written so far) and we would like to avoid concatenation of string objects, because Strings are immutable. After executing the query, we iterate over the resultset and create a `Course` object and add it to the list of courses, which is returned at the end.

One interesting thing here is the use of `ResultSet.wasNull`. We want to check whether the `teacher_id` field in the `Course` table for that particular row was null. Therefore, immediately after calling `rs.getInt("teacherId")`, we check whether the value fetched by `ResultSet` was null by calling `rs.wasNull`. If `teacher_id` was null, then the teacher was not set for that course, so we continue the loop, skipping the code to create a `Teacher` object.

In the final block, we catch an exception when closing `ResultSet`, `Statement`, and `Connection` and ignore it.

Let's now add a method in the `Course` bean to fetch courses by calling the

`getCourses` method of `CourseDAO`. Open the `Course` bean and add the following method: public List<Course> getCourses() throws SQLException {

return courseDAO.getCourses();

}

We are now ready to modify `listCourse.jsp` to display courses. Open the JSP and replace the existing code with the following: <%@ page language="java" contentType="text/html; charset=UTF-8"

pageEncoding="UTF-8"%>

<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd"> <html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=UTF-8"> <title>Courses</title> </head>

<body>

<c:catch var="err">

<jsp:useBean id="courseBean" class="packt.book.jee.eclipse.ch4.bean.Course"/> <c:set var="courses" value="${courseBean.getCourses()}"/> </c:catch>

<c:choose>

<c:when test="${err != null}"> <c:set var="errMsg" value="${err.message}"/> </c:when>

<c:otherwise>

</c:otherwise>

</c:choose>

<h2>Courses:</h2>

<c:if test="${errMsg != null}"> <span style="color: red;"> <c:out value="${errMsg}"></c:out>

</c:if>

<table>

<tr>

<th>Id</th>

<th>Name</th>

<th>Credits</th> <th>Teacher</th> </tr>

<c:forEach items="${courses}" var="course"> <tr>

<td>${course.id}</td> <td>${course.name}</td> <td>${course.credits}</td> <c:choose>

<c:when test="${course.teacher != null}"> <td>${course.teacher.firstName} </td> </c:when>

<c:otherwise>

<td></td>

</c:otherwise>

</c:choose>

</tr>

</c:forEach>

</table>

</body>

</html>

Most of the code should be easy to understand because we have used similar code in previous examples. At the beginning of the script, we create a `Course` bean and get all the courses and assign the course list to a variable called `courses`: <c:catch var="err"> <jsp:useBean id="courseBean" class="packt.book.jee.eclipse.ch4.bean.Course"/> <c:set var="courses" value="${courseBean.getCourses()}"/> </c:catch>

To display courses, we create a HTML table and set its headers. A new thing in the preceding code is the use of the `<c:forEach>` JSTL tag to iterate over the list. The `forEach` tag takes the following two attributes:

- List of objects
- Variable name of a single item when iterating over the list

In the preceding case, the list of objects is provided by the `courses` variable that we set at the beginning of the script and we identify a single item in the list with the variable name `course`. We then display the course details and teacher for the course, if any.

Writing code to add `Teacher` and `Student` and list them is left to readers as an exercise. The code would be very similar to that for `course`, but with different table and class names.

Completing add course functionality

We still haven't completed the functionality for adding a new course; we need to provide an option to assign a teacher to a course when adding a new course. Assuming that you have implemented `TeacherDAO` and created `addTeacher` and `getTeachers` methods in the `Teacher` bean, we can now complete the add course functionality.

First, modify `addCourse` in `CourseADO` to save the teacher ID for each course, if it is not zero. The SQL statement to insert course changes is as follows:

```
String sql = "insert into Course (name, credits, Teacher_id) values (?,?,?)";
```

We have added the `Teacher_id` column and the corresponding parameter holder `?`. We will set `Teacher_id` to null if it is zero; or else the actual value:

```
if (course.getTeacherId() == 0)
  stmt.setNull(3, Types.INTEGER);
else
  stmt.setInt(3,course.getTeacherId());
```

We will then modify the `Course` bean to save the teacher ID that will be passed along with the `POST` request from the HTML form:

```
public class Course {

  private int teacherId;
  public int getTeacherId() {
    return teacherId;
}
  public void setTeacherId(int teacherId) {
    this.teacherId = teacherId;
  }
}
```

Next, we will modify `addCourse.jsp` to display the drop-down list of teachers when adding a new course. We first need to get the list of teachers. Therefore, we will create a `Teacher` bean and call the `getTeachers` method on it. We will do this just before the Add Course header:

```
<jsp:useBean id="teacherBean" class="packt.book.jee.eclipse.ch4.bean.Teacher"/>
<c:catch var="teacherBeanErr">
<c:set var="teachers" value="${teacherBean.getTeachers()}"/>
</c:catch>
```

```
<c:if test="${teacherBeanErr != null}">
  <c:set var="errMsg" value="${err.message}"/>
</c:if>
```

Finally, we will display the HTML drop-down list in the form and populate it with teacher names:

```
Teacher :
<select name="teacherId">
<c:forEach items="${teachers}" var="teacher">
<option value="${teacher.id}">${teacher.firstName}
</option>
</c:forEach>
</select>
```

Download the accompanying code for this chapter to see the complete source code of `CourseDAO` and `addCourse.jsp`.

With this, we conclude our discussion on using JDBC to create a web application that uses a database. With the examples that you have seen so far, you should be in a good position to complete the remaining application by adding functionality to modify and delete records in the database. The `update` and `delete` SQL statements can be executed by `Statement` or `PreparedStatement`, just as `insert` statements are executed using these two classes.

Using Eclipse Data Source Explorer

It is sometimes useful if you can see data in database tables from your IDE and can modify it. This is possible in Eclipse JEE using Data Source Explorer. This view is displayed in a tab at the lower pane, just below editors, in the Java EE perspective. If you do not see the view, or have closed the view, you can reopen it by selecting the Window | Show View | Other menu. Type `data source` in the filter textbox and you should see the view name under the Data Management group. Open the view:

Figure 4.12: Data Source Explorer Right-click on the Database Connections node and select New. From the list, select MySQL:

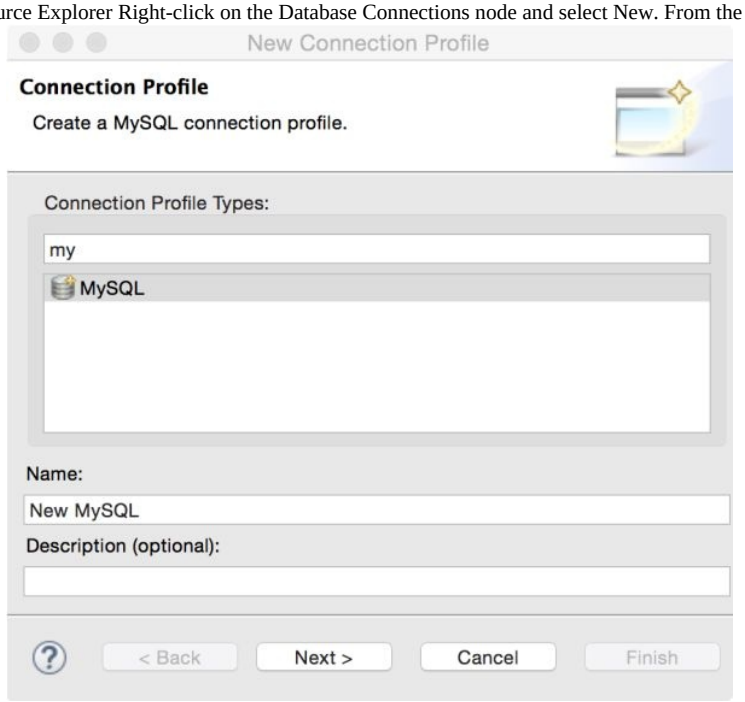


Figure 4.13: Select the MySQL Connection Profile Click Next. If the drivers list is empty, you haven't configured the driver yet. Click on the icon next to the drop-down list for drivers to open the configuration page:

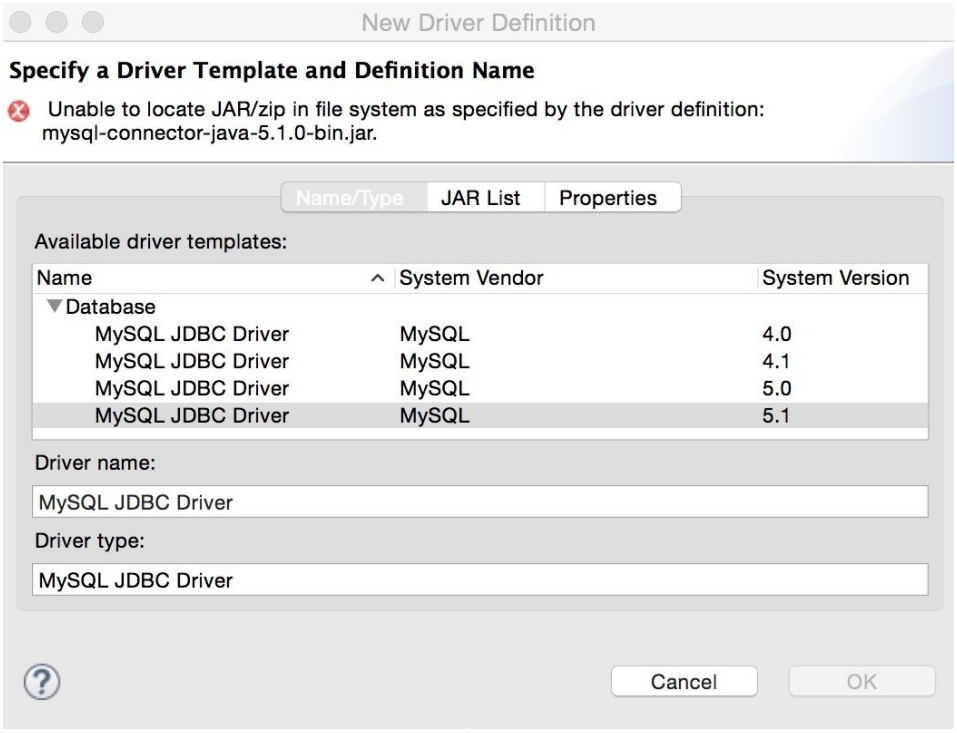


Figure 4.14: Selecting Database Driver in JDBC New Driver Definition page Select the appropriate MySQL version and click on the JAR List tab:

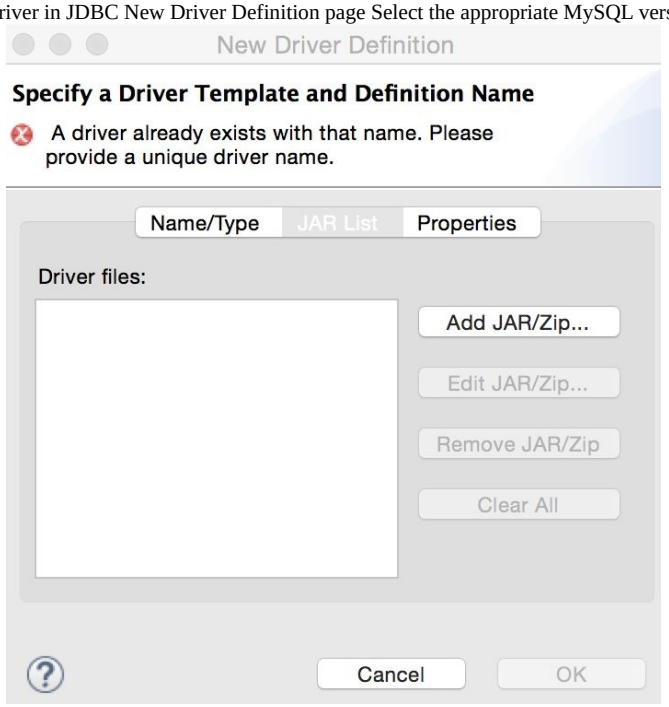


Figure 4.15: Adding Driver Files in JDBC New Driver Definition page Remove any files from the Driver files list. Click on the Add JAR/Zip... button. This opens the File Open dialog. Select the JAR file for the MySQL driver version that you have selected. Since Maven has already downloaded the JAR file for you, you can select it from the local Maven repository. On OS X and Linux, the path is `~/.m2/repository/mysql/mysql-connector-java/<version_num>/mysql_connector_java_version_num/mysql-connector-java-version_num.jar` (`version_num` is a placeholder for the actual version number in the path). On Windows, you can find the Maven repository at `C:\Users\{your-username}\.m2` and then, the relative path for the MySQL driver is the same as that in OS X.

If you have trouble finding the JAR in the local Maven repository, you can download the JAR file (for the MySQL JDBC driver) from `http://dev.mysql.com/downloads/connector/j/`.

Once you specify the correct driver JAR file, you need to set the following properties:

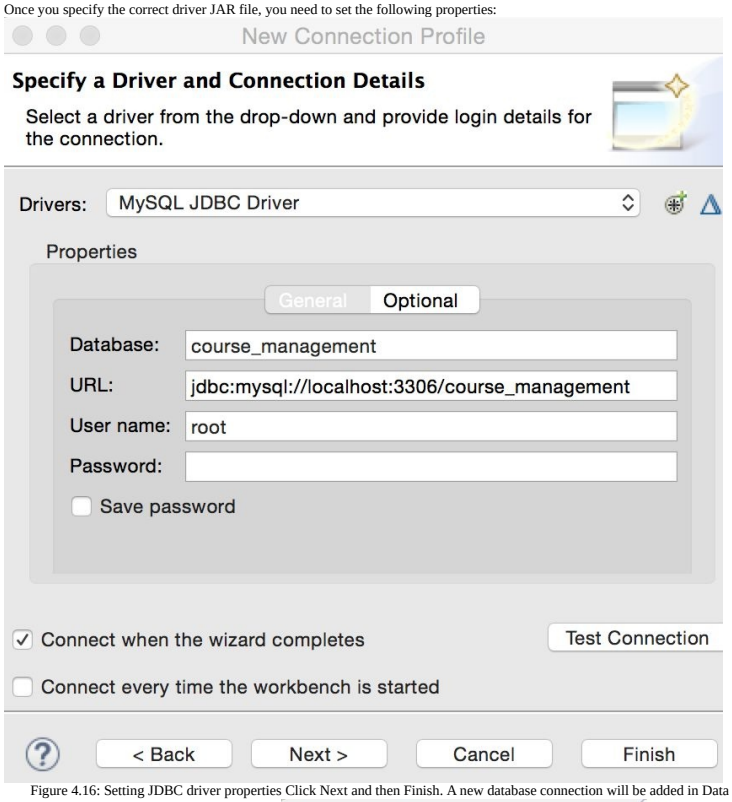


Figure 4.16: Setting JDBC driver properties Click Next and then Finish. A new database connection will be added in Data Source Explorer. You can now browse the database schema and tables:

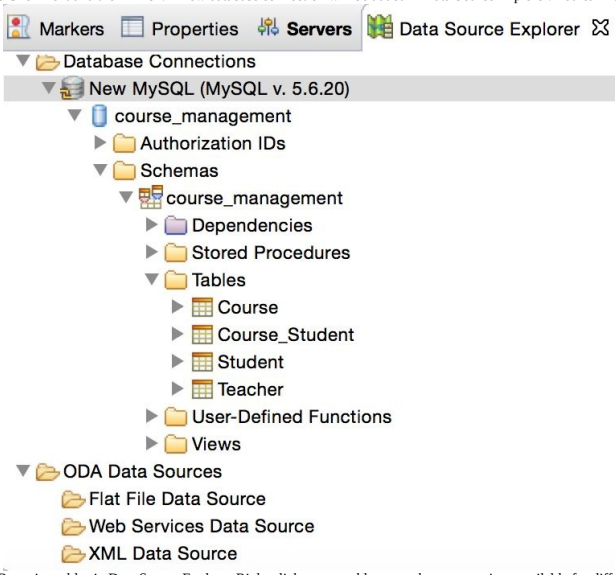


Figure 4.17: Browsing tables in Data Source Explorer Right-click on any table to see the menu options available for different actions:

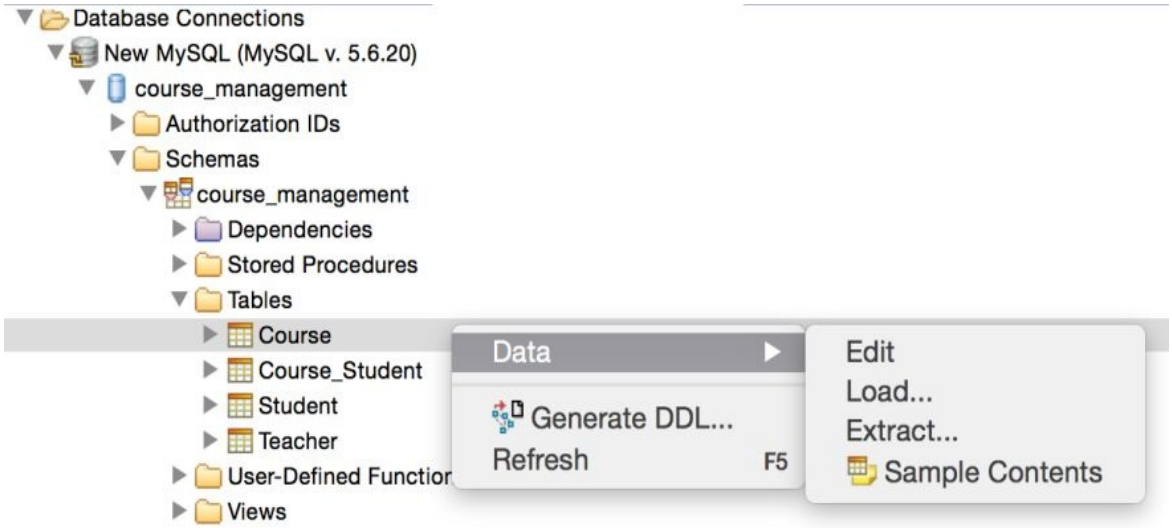


Figure 4.18: Table menu options in Data Source Explorer Select the Edit menu to open a page in the editor where you can see the existing records in the table. You can also modify or add new data in the same page. Select the Load option to load data from an external file into the table. Select the Extract option to export data from the table.

Creating database applications using JPA

In the previous section, we learned how to create the *Course Management* application using JDBC and JSTL. In this section, we will build the same application using JPA and JSF. We have learned how to create a web application using JSF in `Chapter 2`, *Creating a Simple JEE Web Application*. We will use much of that knowledge in this section.

As mentioned at the beginning of this chapter, JPA is an ORM framework, which is now part of the JEE specification. At the time of writing, it is in version 2.2. We will learn a lot about JPA as we develop our application.

Create the Maven project called `CourseManagementJPA` with group ID `packt.book.jee_eclipse` and artifact ID `CourseManagementJPA`. Eclipse JEE has great tools for creating applications using JPA, but you need to convert your project to a JPA project. We will see how to do this later in this section.

Creating user interfaces for adding courses using JSF

Before we write any data access code using JPA, let's first create the user interface using JSF. As we have learned in `Chapter 2`, *Creating a Simple JEE Web Application*, we need to add Maven dependencies for JSF. Add the following dependencies in `pom.xml`: <dependencies> <dependency>

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

<dependency>

<groupId>com.sun.faces</groupId> <artifactId>jsf-api</artifactId> <version>2.2.16</version> </dependency>

<dependency>

<groupId>com.sun.faces</groupId> <artifactId>jsf-impl</artifactId> <version>2.2.16</version> </dependency>

</dependencies>

When you run the application later, if Tomcat throws an exception for not finding `javax.faces.webapp.FacesServlet` *then you may have to download* `jsf-api-2.2.16.jar` *(*`http://central.maven.org/maven2/com/sun/faces/jsf-impl/2.2.16/jsf-impl-2.2.16.jar`*), and* `jsf-impl-2.2.16.jar` *(*`http://central.maven.org/maven2/com/sun/faces/jsf-impl/2.2.16/jsf-impl-2.2.16.jar`*), and copy them to the* `<tomcat-install-folder>/lib` *folder. Set scopes for these two libraries as provided:* `<scope>provided</scope>` *in* `pom.xml`*. Then clean the project (Run As | Maven Clean) and install it again (Run As | Maven Install).*

We need to add `web.xml`, add a declaration for the JSF servlet in it, and add the servlet mapping. Eclipse provides you a very easy way to add `web.xml` (which should be in the `WEB-INF` folder). Right-click on the project and select the Java EE Tools | Generate Deployment Descriptor Stub menu. This creates the `WEB-INF` folder under `src/main/webapp` and creates `web.xml` in the `WEB-INF` folder with the default

content. Now, add the following servlet and mapping: <servlet> <servlet-name>JSFServlet</servlet-name> <servlet-class>javax.faces.webapp.FacesServlet</servlet-class> <load-on-startup>1</load-on-startup> </servlet>

<servlet-mapping>

<servlet-name>JSFServlet</servlet-name> <url-pattern>*.xhtml</url-pattern> </servlet-mapping>

Let's now create JavaBeans for `Course`, `Teacher`, `Student`, and `Person`, just as we created them in the previous example for JDBC. Create a `packt.book.jee.eclipse.ch4.jpa.bean` package and create the following JavaBeans.

Here is the source code of the `Course` bean (in `Course.java`): package packt.book.jee.eclipse.ch4.jpa.bean;

import java.io.Serializable;

import javax.faces.bean.ManagedBean; import javax.faces.bean.RequestScoped;

@ManagedBean (name="course")

@RequestScoped

public class Course implements Serializable {

private static final long serialVersionUID = 1L;

private int id;

private String name;

private int credits;

private Teacher teacher;

public int getId() {

```
return id;
}
public void setId(int id) {
this.id = id;
}
public String getName() {
return name;
}
public void setName(String name) {
this.name = name;
}
public int getCredits() {
return credits;
}
public void setCredits(int credits) {
this.credits = credits;
}
public boolean isValidCourse() {
return name != null && credits != 0; }
public Teacher getTeacher() {
return teacher;
```

}

public void setTeacher(Teacher teacher) {

this.teacher = teacher;

}

}

Here is the source code of the `Person` bean (in `Person.java`): package packt.book.jee.eclipse.ch4.jpa.bean;

import java.io.Serializable;

public class Person implements Serializable{

private static final long serialVersionUID = 1L;

private int id;

private String firstName;

private String lastName;

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

```
public String getFirstName() {

return firstName;

}

public void setFirstName(String firstName) {

this.firstName = firstName;

}

public String getLastName() {

return lastName;

}

public void setLastName(String lastName) {

this.lastName = lastName;

}

}
```

Here is the source code of the `Student` bean (in `Student.java`): package packt.book.jee.eclipse.ch4.jpa.bean;

import javax.faces.bean.ManagedBean; import javax.faces.bean.RequestScoped; import java.util.Date;

```
@ManagedBean (name="student")

@RequestScoped

public class Student extends Person {
```

private static final long serialVersionUID = 1L;

private Date enrolledsince;

public Date getEnrolledsince() {

return enrolledsince;

}

public void setEnrolledsince(Date enrolledsince) {

this.enrolledsince = enrolledsince; }

}

And, finally, here is the source code of the `Teacher` bean (in `Teacher.java`): package packt.book.jee.eclipse.ch4.jpa.bean;

import javax.faces.bean.ManagedBean; import javax.faces.bean.RequestScoped;

@ManagedBean (name="teacher")

@RequestScoped

public class Teacher extends Person {

private static final long serialVersionUID = 1L;

private String designation;

public String getDesignation() {

return designation;

}

public void setDesignation(String designation) {

this.designation = designation; }

public boolean isValidTeacher() {

return getFirstName() != null; }

}

All are JSF managed beans in `RequestScope`*. Refer to the JSF discussion in* `Chapter 2`*, Creating a Simple JEE Web Application, for more about managed beans and scopes.*

These beans are now ready to use in JSF pages. Create a JSF page and name it `addCourse.xhtml` and add the following content: <html xmlns="http://www.w3.org/1999/xhtml"

xmlns:f="http://java.sun.com/jsf/core"

xmlns:h="http://java.sun.com/jsf/html">

<h2>Add Course:</h2> <h:form>

<h:outputLabel value="Name:" for="name"/> <h:inputText value="#{course.name}" id="name"/>
 <h:outputLabel value="Credits:" for="credits"/> <h:inputText value="#{course.credits}" id="credits"/>

<h:commandButton value="Add" action="#{courseServiceBean.addCourse} "/> </h:form>

</html>

The page uses JSF tags and managed beans to get and set values. Notice the value of the `action` attribute of the `h:commandButton` tag—it is the

`courseServiceBean.addCourse` method, which will be called when the Add button is clicked. In the application that we created using JDBC, we wrote code to interact with DAOs in the JavaBeans. For example, the `Course` bean had the `addCourse` method. However, in the JPA project we will handle it differently. We will create service bean classes (they are also managed beans, just like `Course`) to interact with the data access objects and have the `Course` bean contain only the values set by the user.

Create a package named `packt.book.jee.eclipse.ch4.jpa.service_bean`. Create the class named `CourseServiceBean` in this package with the following code: package packt.book.jee.eclipse.ch4.jpa.service_bean;

import javax.faces.bean.ManagedBean; import javax.faces.bean.ManagedProperty; import javax.faces.bean.RequestScoped;

import packt.book.jee.eclipse.ch4.jpa.bean.Course;

@ManagedBean(name="courseServiceBean") @RequestScoped

public class CourseServiceBean {

@ManagedProperty(value="#{course}") private Course course;

private String errMsg= null;

public Course getCourse() {

return course;

}

public void setCourse(Course course) {

this.course = course;

```
}

public String getErrMsg() {
return errMsg;
}

public void setErrMsg(String errMsg) {
this.errMsg = errMsg;
}

public String addCourse() {
return "listCourse";
}
}
```

`CourseServiceBean` is a managed bean and it contains the `errMsg` field (to store any error message during the processing of requests), the `addCourse` method, and the `course` field (which is annotated with `@ManagedProperty`).

The `ManagedProperty` annotation tells the JSF implementation to inject another bean (specified as the `value` attribute) in the current bean. Here, we expect `CourseServiceBean` to have access to the `course` bean at runtime, without instantiating it. This is part of the **dependency injection** (**DI**) framework supported by Java EE. We will learn more about the DI framework in Java EE in later chapters. The `addCourse` function doesn't do much at this point, it just returns the `"listCourse"` string. If you want to execute `addCourse.xhtml` at this point, create a `listCourse.xml` file with some placeholder content and test `addCourse.xhtml`. We will add more

content to `listCourse.xml` later in this section.

JPA concepts

JPA is an ORM framework in JEE. It provides a set of APIs that the JPA implementation providers are expected to implement. There are many JPA providers, such as **EclipseLink** (https://eclipse.org/eclipselink/), **Hibernate JPA** (http://hibernate.org/orm/), and **OpenJPA** (http://openjpa.apache.org/). Before we start writing the persistence code using JPA, it is important to understand basic concepts of JPA.

Entity

Entity represents a single object instance that is typically related to one table. Any **Plain Old Java Object** (**POJO**) can be converted to an entity by annotating the class with `@Entity`. Members of the class are mapped to columns of a table in the database. Entity classes are simple Java classes, so they can extend or include other Java classes or even another JPA entity. We will see an example of this in our application. You can also specify validation rules for members of the Entity class; for example, you can mark a member as not null using the `@NotNull` annotation. These annotations are provided by Java EE Bean Validation APIs. See https://javaee.github.io/tutorial/bean-validation002.html#GIRCZ for a list of validation annotations.

EntityManager

`EntityManager` provides the persistence context in which the entities exist. The persistence context also allows you to manage transactions. Using `EntityManager` APIs, you can perform query and write operations on entities. The entity manager can be web-container-managed (in which case an instance of `EntityManager` is injected by the container), or application-managed. In this chapter, we are going to look at application-managed entity managers. We will visit container-managed entity managers in `Chapter 7`, *Creating JEE Applications with EJB*, when we learn about EJBs. The persistence unit of the entity manager defines the database connectivity information and groups entities that become part of the persistence unit. It is defined in the configuration file called `persistence.xml` and is expected to be in `META-INF` in the class path.

`EntityManager` has its own persistence context, which is a cache of entities. Updates to entities are first done in the cache and then pushed to the database when a transaction is committed or when the data is explicitly pushed to the database.

When an application is managing `EntityManager`, it is advisable to have only one instance of `EntityManager` for a persistence unit.

EntityManagerFactory

`EntityManagerFactory` creates `EntityManager`. `EntityManagerFactory` itself is obtained by calling a static `Persistence.createEntityManagerFactory` method. An argument to this function is a `persistence-unit` name that you have specified in `persistence.xml`.

Creating a JPA application

The following are the typical steps in creating a JPA application:

1. Create a database schema (tables and relationships). Optionally, you can create tables and relationships from JPA entities. We will see an example of this. However, it should be mentioned here that although creating tables from JPA entities is fine for development, it is not recommended in the production environment; doing so may result in a non-optimized database model.
2. Create `persistence.xml` and specify the database configurations.
3. Create entities and relationships.
4. Get an instance of `EntityManagerFactory` by calling `Persistence.createEntityManagerFactory`.
5. Create an instance of `EntityManager` from `EntityManagerFactory`.
6. Start a transaction on `EntityManager` if you are performing `insert` or `update` operations on the entity.
7. Perform operations on the entity.
8. Commit the transaction.

Here is an example snippet:

```
EntityManagerFactory factory =
 Persistance.Persistence.createEntityManagerFactory("course_management")


EntityManager entityManager = factory.createEntityManager();


EntityTransaction txn = entityManager.getTransaction();


txn.begin();


entityManager. persist(course);


txn.commit();
```

You can find a description of JPA annotations at http://www.eclipse.org/eclipselink/documentation/2.7/jpa/extensions/annotations_ref.htm.
JPA tools in Eclipse EE make adding many of the annotations very easy, as we will see in this section.

Creating a new MySQL schema

For this example, we will create a separate MySQL schema (we won't use the same schema that we created for the JDBC application, although it is possible to do so). Open MySQL Workbench and connect to your MySQL database (see `Chapter 1`, *Introducing JEE and Eclipse*, if you do not know how to connect to the MySQL database from MySQL Workbench).

Right-click in the Schema window and select Create Schema...:

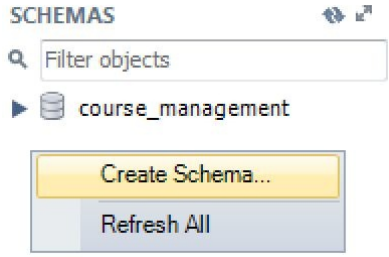


Figure 4.19: Creating a new MySQL schema

Name the new schema `course_management_jpa` and click Apply. We are going to use this schema for the JPA application.

Setting up a Maven dependency for JPA

In this example, we will use the EclipseLink (https://eclipse.org/eclipselink/) JPA implementation. We will use the MySQL JDBC driver and Bean Validation framework for validating members of entities. Finally, we will use Java annotations provided by JSR0250. So, let's add Maven dependencies for all these: <dependency>

<groupId>org.eclipse.persistence</groupId> <artifactId>eclipselink</artifactId> <version>2.5.2</version> </dependency>

<dependency>

<groupId>mysql</groupId> <artifactId>mysql-connector-java</artifactId> <version>5.1.34</version> </dependency>

<dependency>

<groupId>javax.validation</groupId> <artifactId>validation-api</artifactId> <version>1.1.0.Final</version> </dependency>

<dependency>

<groupId>javax.annotation</groupId> <artifactId>jsr250-api</artifactId> <version>1.0</version> </dependency>

Converting a project into a JPA project

Many JPA tools become active in Eclipse JEE only if the project is a JPA project. Although we have created a Maven project, it is easy to add an Eclipse JPA facet to it:

1. Right-click on the project and select Configure | Convert to JPA Project:

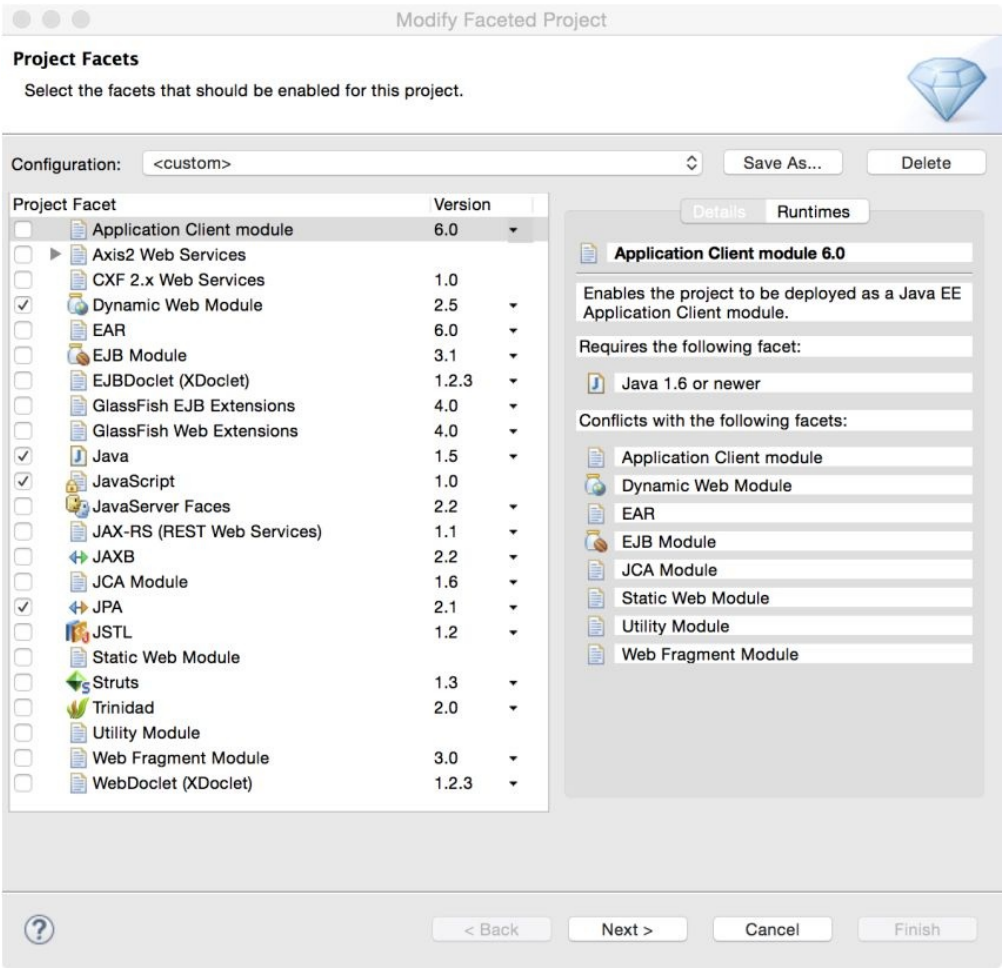


Figure 4.20: Addding a JPA facet to a project

2. Make sure JPA is selected.

3. On the next page, select EclipseLink 2.5.x as the platform.
4. For the JPA implementation type, select Disable Library Configuration.

5. The drop-down list for Connection lists any connections you might have configured in Data Source Explorer. For now, do not select any connection. At the bottom of the page, select the Discover annotated classes automatically option:

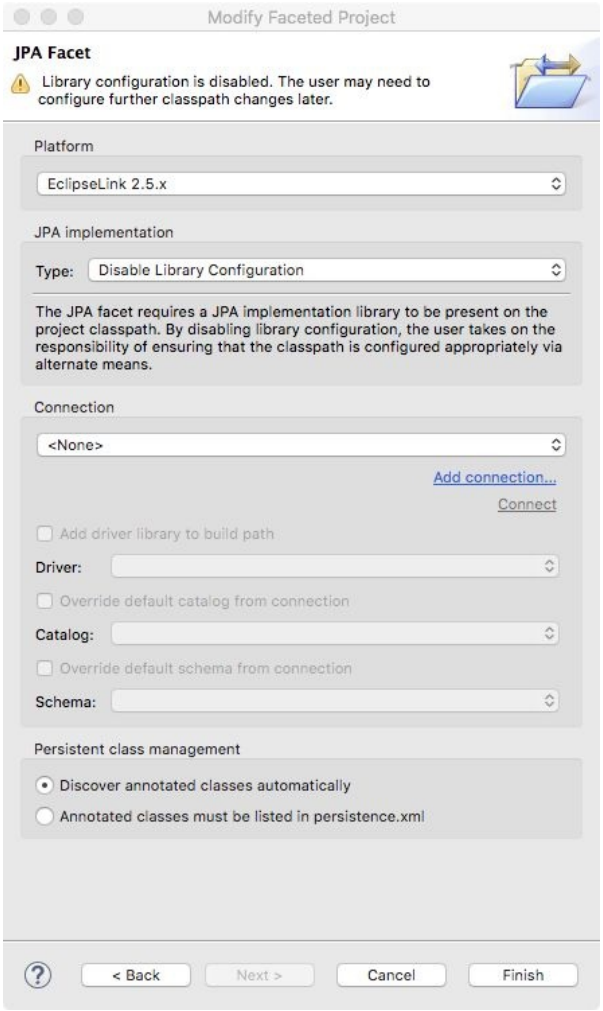


Figure 4.21: Configuring a JPA facet

6. Click Finish.
7. Notice that the JPA Content group is created under the project and `persistence.xml` is created in it. Open `persistence.xml` in the editor.
8. Click on the Connection tab and change Transaction type to Resource Local. We have selected Resource Local because, in this chapter, we are

going to manage `EntityManager`. If you want the JEE container to manage `EntityManager`, then you should set Transaction type to JTA. We will see an example of the JTA transaction type in `Chapter 7`, *Creating JEE Application with EJB*.

9. Enter EclipseLink connection pool attributes as shown in the following screenshot and save the file:

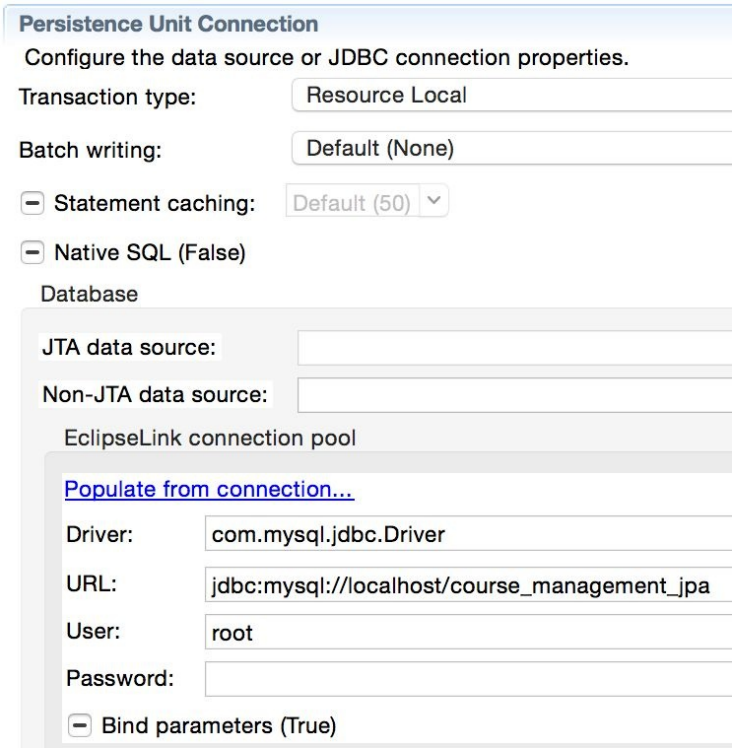


Figure 4.22: Setting up Persistence Unit Connection

10. Next, click on the Schema Generation tab. Here, we will set the options to generate database tables and relationships from entities. Select the options as shown in the following screenshot:

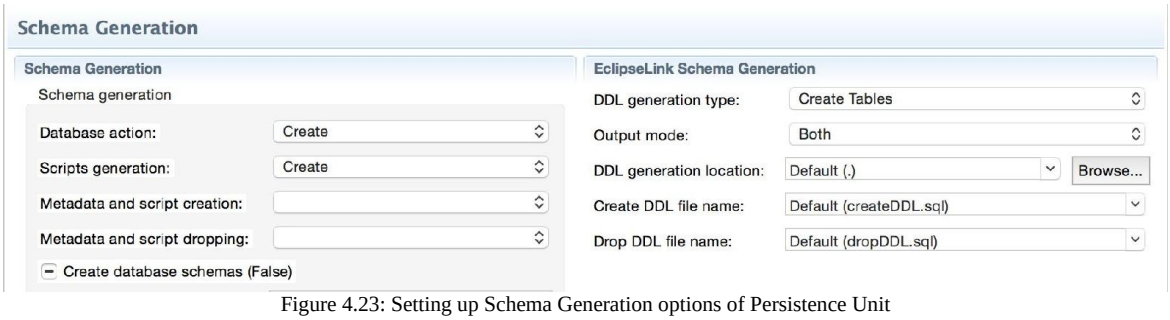


Figure 4.23: Setting up Schema Generation options of Persistence Unit

Here is the content of the `persistence.xml` file after setting the preceding options:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
  <persistence-unit name="CourseManagementJPA" transaction-type="RESOURCE_LOCAL">
    <properties>
      <property name="javax.persistence.jdbc.driver"
       value="com.mysql.jdbc.Driver"/>      <property name="javax.persistence.jdbc.url"
       value="jdbc:mysql://localhost/course_management_jpa"/>
      <property name="javax.persistence.jdbc.user" value="root"/>
      <property name="javax.persistence.schema-
       generation.database.action" value="create"/>      <property
name="javax.persistence.schema-
       generation.scripts.action" value="create"/>
<property name="eclipselink.ddl-generation" value="create- tables"/>
      <property name="eclipselink.ddl-generation.output-mode" value="both"/>
    </properties>
  </persistence-unit>
</persistence>
```

Creating entities

We have already created JavaBeans for `Course`, `Person`, `Student`, and `Teacher`. We will now convert them to JPA entities using the `@Entity` annotation. Open `Course.java` and add the following annotations:

```
@ManagedBean (name="course")
@RequestScoped
@Entity
public class Course implements Serializable
```

The same bean can act as a managed bean for JSF and an entity for JPA. Note that if the name of the class is different from the table name in the database, you will need to specify a `name` attribute of the `@Entity` annotation. For example, if our `Course` table were called `SchoolCourse`, then the entity declaration would be as follows:

```
@Entity(name="SchoolCourse")
```

To specify the primary key of the `Entity`, use the `@Id` annotation. In the `Course` table, `id` is the primary key and is autogenerated. To indicate autogeneration of the value, use the `@GeneratedValue` annotation. Use the `@column` annotation to indicate that the member variable corresponds to a column in the table. So, the annotations for `id` are as follows:

```
@Id
@GeneratedValue(strategy=GenerationType.IDENTITY)
@Column(name="id")
private int id;
```

You can specify validations for a column using Bean Validation framework annotations, as mentioned earlier. For example, the course name should not be null:

```
@NotNull
@Column(name="name")
private String name;
```

Furthermore, the minimum value of credits should be `1`:

```
@Min(1)
@Column(name="credits")
private int credits;
```

In the preceding examples, the `@Column` annotation is not required to specify the name of the column if the field name is the same as the column name.

If you are using JPA entities to create tables and want to exactly specify the type of columns, then you can use the `columnDefinition` attribute of the `@Column` annotation; for example, to specify a column of type `varchar` with length `20`, you could use `@Column(columnDefinition="VARCHAR(20)")`.

Refer to `https://javaee.github.io/javaee-spec/javadocs/javax/persistence/Column.html` to see all the attributes of the `@Column` annotation.

We will add more annotations to `Course Entity` as needed later. For now, let's turn our attention to the `Person` class. This class is the parent class of the `Student` and `Teacher` classes. However, in the database, there is no `Person` table and all the fields of `Person` and `Student` are in the `Student` table; and the same for the `Teacher` table. So, how do we model this in JPA? Well, JPA supports inheritance of entities and provides control over how they should be mapped to database tables. Open the `Person` class and add the following annotations:

```
@Entity
@Inheritance(strategy=TABLE_PER_CLASS)
public abstract class Person implements Serializable { ...
```

We are not only identifying the `Person` class as `Entity`, but we are also indicating that it is used for inheritance (using `@Inheritance`). The inheritance strategy decides how tables are mapped to classes. There are three possible strategies:

- `SINGLE_TABLE`: In this case, fields of parent and child classes would be mapped to the table of the parent class. If we use this strategy, then the fields of `Person`, `Student`, and `Teacher` will be mapped to the table for the `Person` entity.
- `TABLE_PER_CLASS`: In this case, each concrete class (non-abstract class) is mapped to a table in the database. All fields of the parent class are also mapped to the table for the child class. For example, all fields of `Person` and `Student` will be mapped to columns in the `Student` table. Since `Person` is marked as abstract, no table will be mapped by the `Person` class. It exists only to provide inheritance support in the application.
- `JOINED`: In this case, the parent and its children are mapped to separate tables. For example, `Person` will be mapped to the `Person` table, and `Student` and `Teacher` will be mapped to the corresponding tables in the database.

As per the schema that we created for the JDBC application, we have `Student` and `Teacher` tables with all the required columns and there is no `Person` table. Therefore, we have selected the `TABLE_PER_CLASS` strategy here.

See more information about entity inheritance in JPA at `https://javaee.github.io/tutorial/persistence-intro003.html#BNBQN`.

The fields `id`, `firstName`, and `lastName` in the `Person` table are shared by `Student` and `Teacher`. Therefore, we need to mark them as columns in the tables and set the primary key. So, add the following annotations to the fields in the `Person` class:

```
@Id
  @GeneratedValue(strategy=GenerationType.IDENTITY)
  @Column(name="id")
  private int id;

  @Column(name = "first_name")
  @NotNull
  private String firstName;

  @Column(name = "last_name")
  private String lastName;
```

Here, column names in the table do not match class fields. Therefore, we have to specify the name attribute in `@Column` annotations.

Let's now mark the `Student` class as `Entity`:

```
@Entity
@ManagedBean (name="student")
@RequestScoped
public class Student extends Person implements Serializable
```

The `Student` class has a `Date` field called `enrolledSince`, which is of the `java.util.Date` type. However, JDBC and JPA use the `java.sql.Date` type. If you want JPA to automatically convert `java.sql.Date` to `java.util.Date`, then you need to mark the field with the `@Temporal` annotation:

```
@Temporal(DATE)
@Column(name="enrolled_since")
private Date enrolledSince;
```

Open the `Teacher` class and add the `@Entity` annotation to it:

```
@Entity
@ManagedBean (name="teacher")
@RequestScoped
public class Teacher extends Person implements Serializable
```

Then, map the `designation` field in the class:

```
@NotNull
@Column(name="designation")
private String designation;
```

We have now added annotations for all tables and their fields that do not participate in table relationships. We will now model the relationships between tables in our classes.

Configuring entity relationships

First, we will model the relationship between `Course` and `Teacher`. There is a one-to-many relationship between them: one teacher may teach a number of courses. Open `Course.java` in the editor. Open the JPA perspective in Eclipse JEE (Window | Open Perspective | JPA menu).

Configuring many-to-one relationships

With `Course.java` open in the editor, click on the JPA Details tab in the lower window (just below the editor window). In `Course.java`, click on the `teacher` member variable. The JPA Details tab shows the details of this attribute:

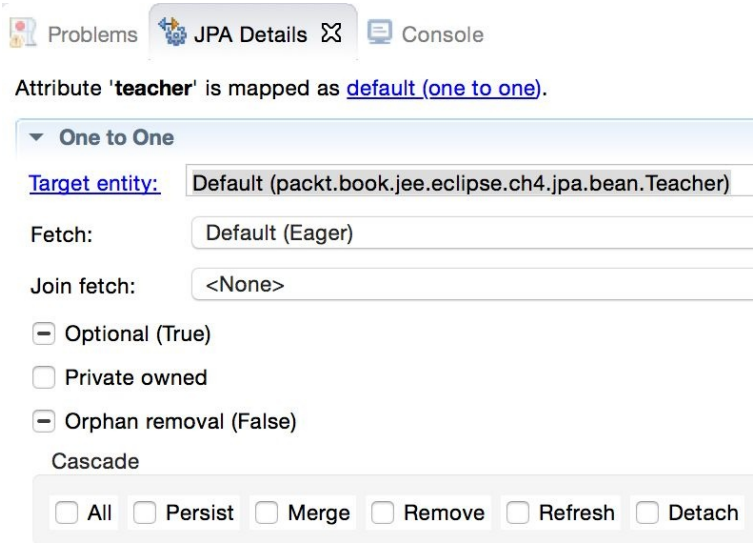


Figure 4.24: JPA details of an entity attribute Target entity is auto-selected (as `Teacher`) because we have marked `Teacher` as an entity and the type of the `teacher` field is `Teacher`.

However, Eclipse has assumed a one-to-one relationship between `Course` and `Teacher`, which is not correct. There is a many-to-one relationship between `Course` and `Teacher`. To change this, click on the (one_to_one) hyperlink at the top of the JPA Details view and select the Many To One in Mapping Type Selection dialog box.

Select only Merge and Refresh cascade options; otherwise, duplicate entries will be added in the `Teacher` table for every `Teacher` that you selected for a `Course`.

See `https://javaee.github.io/tutorial/persistence-intro002.html#BNBQH` for more details on entity relationships and cascade options.

When you select Merge and Refresh cascade options, the `cascade` attribute added to the annotation is added to the `teacher` field in the `Course` entity: @ManyToOne(cascade = { MERGE, REFRESH }) private Teacher teacher;

Scroll down the JPA Details page to see Joining Strategy. This determines how columns in `Course` and `Teacher` tables are joined:

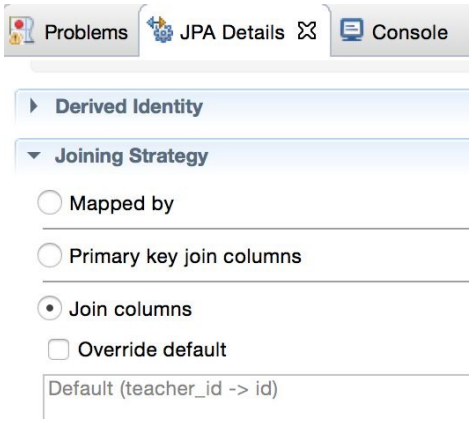


Figure 4.25: Editing Joining Strategy in an entity relationship Note that the default joining strategy is that the `teacher_id` column in the `Course` table maps to the `id` column in the `Teacher` table. Eclipse has just guessed `teacher_id` (the appended `id` to the `teacher` field in the `Course` entity), but if we had a different join column in the `Course` table, for example, `teacherId`, then we would need to override the default join columns. Click on the Override default checkbox and then on the Edit button on the right-hand side of the textbox:

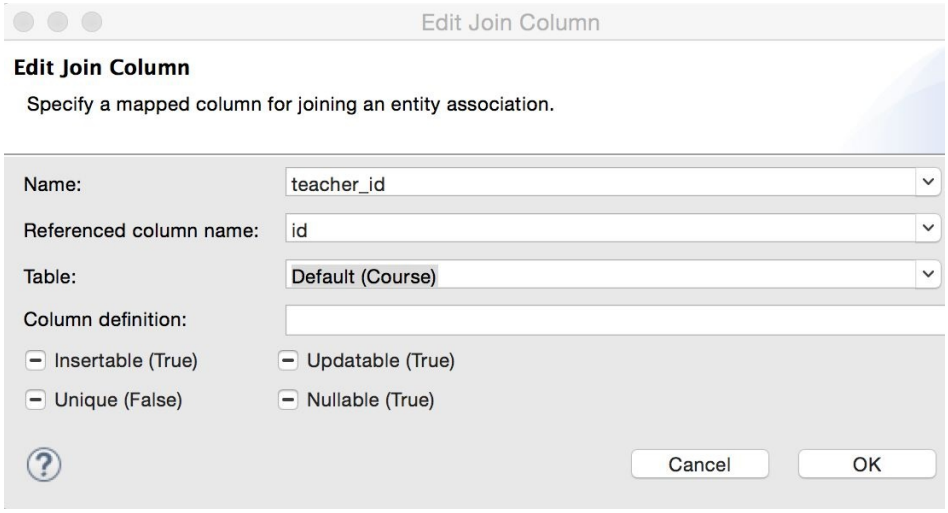


Figure 4.26: Editing Join Column In our case, the default options match the table columns, so we will keep them unchanged. When you select the Override default checkbox, the `@JoinColumn` annotation is added to the `teacher` field in the `Course` entity: @JoinColumn(name = "teacher_id", referencedColumnName = "id") @ManyToOne(cascade = { MERGE, REFRESH }) private Teacher teacher;

All the required annotations for the `teacher` field are now added.

Configuring many-to-many relationships

We will now configure `Course` and `Student` entities for a many-to-many relationship (a course can have many students, and one student can take many courses).

Many-to-many relations could be unidirectional or bidirectional. For example, you may only want to track students enrolled in the courses (so the `Course` entity will have a list of students) and not students taking the courses (the `Student` entity does not keep a list of courses). This is an unidirectional relationship where only the `Course` entity knows about the students, but the `Student` entity does not know about the courses).

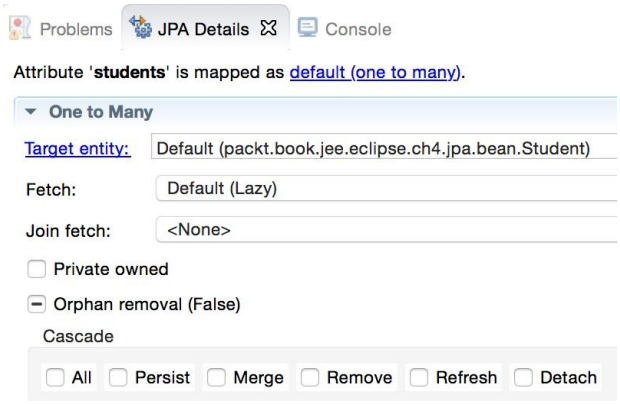
In a bidirectional relationship, each entity knows about the other one. Therefore, the `Course` entity will keep a list of students and the `Student` entity will keep a list of courses. We will configure the bidirectional relationship in this example.

A many-to-many relationship also has one owning side and the other inverse side. You can mark either entity in the relationship as the owning entity. From the configuration point of view, the inverse side is marked by the `mappedBy` attribute to the `@ManyToMany` annotation.

In our application, we will make `Student` as the owning side of the relationship and `Course` as the inverse side. A many-to-many relationship in the database needs a join table, which is configured in the owning entity using the `@JoinTable` annotation.

We will first configure a many-to-many relationship in the `Course` entity. Add a member variable in `Course` to hold a list of `Student` entities and add the getter and the setter for it: private List<Student> students; public List<Student> getStudents() { return students; } public void setStudents(List<Student> students) { this.students = students; }

Then, click on the `students` field (added previously) and notice the settings in the



JPA Details view:

Figure 4.27: Default JPA details for the students field in Course Entity Because the `students` field is a list of `Student` entities, Eclipse has assumed a one-to-many relationship (see the link at the top of the JPA Details view). We need to change this. Click on the one_to_many link and select Many To Many.

Check the Merge and Refresh cascade options. Since we are putting a `Course` entity on the inverse side of the relationship, select Mapped By as Joining Strategy. Enter `courses` in the Attributes text field. The compiler will show an error for this because we don't have a `courses` field in the `Student` entity yet. We will fix this shortly. The JPA settings for the `students` field should be as shown in the following screenshot:

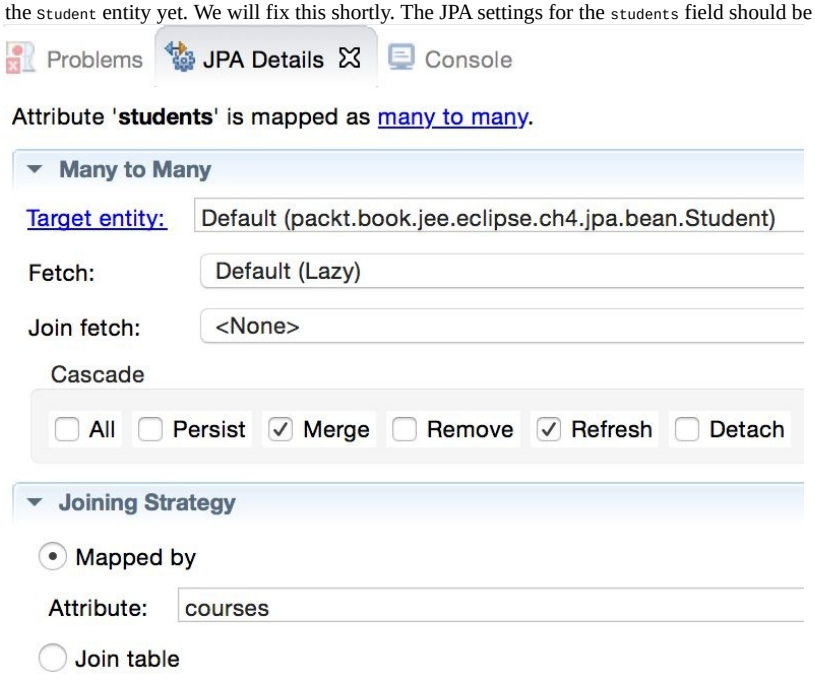
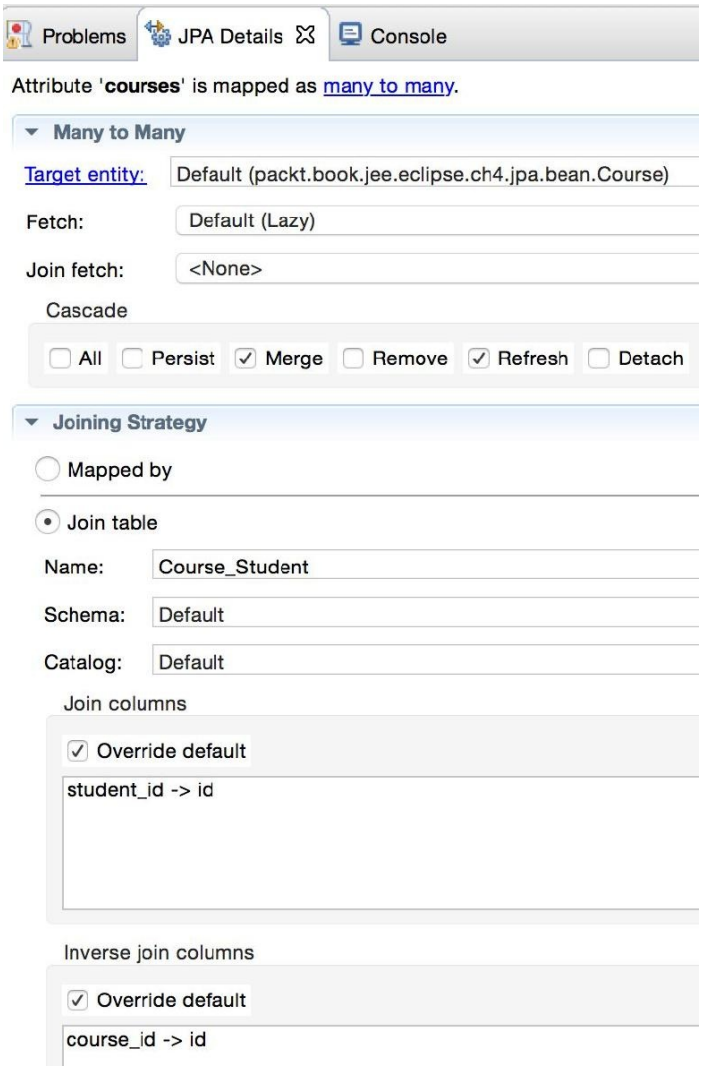


Figure 4.28: Modified JPA settings for the students field in Course Entity Annotations for the `students` field in the `Course` entity should be as follows: @ManyToMany(cascade = { MERGE, REFRESH }, mappedBy = "courses") private List<Student> students;

Open `Student.java` in the editor. Add the `courses` field and the getter and the setter for it. Click on the `courses` field in the file and change the relationship from one-to-many to many-to-many in JPA Details view (as described previously for the `students` field in the `Course` entity). Select the Merge and Refresh cascade options. In the Joining Strategy section, make sure that the Join table option is selected. Eclipse creates the default join table by concatenating the owning table and the inverse table, separated by an underscore (in this case `Student_Course`). Change this to Course_Student to make it consistent with the schema that we created for the JDBC application.

In the Join columns section, select the Override default checkbox. Eclipse has named the join columns `students_id->id`, but in the Course_Student table we created in the JDBC application, we had a column named `student_id`. So, click the Edit button and change the name to student_id.

Similarly, change Inverse join columns from `courses_id->id` to `course_id->id`. After these changes, the JPA Details for the `courses` field



should be as shown in the following screenshot:

Figure 4.29: JPA Details for the courses field in Student entity The previous settings create the following annotations for the `courses` field: @ManyToMany(cascade = { MERGE, REFRESH }) @JoinTable(name = "Course_Student", joinColumns = @JoinColumn(name = "student_id", referencedColumnName = "id"), inverseJoinColumns = @JoinColumn(name = "course_id", referencedColumnName = "id")) List<Course> courses;

We have set all the entity relationships required for our application. Download the accompanying code for this chapter to see the complete source code for `Course`, `Student`, and `Teacher` entities.

We need to add the entities we created previously in `persistence.xml`. Open the file and make sure that the General tab is open. In the Managed Classes session, click the Add button. Type the name of the entity you want to add (for example, `Student`) and select the class from the list. Add all the four entities we have created:

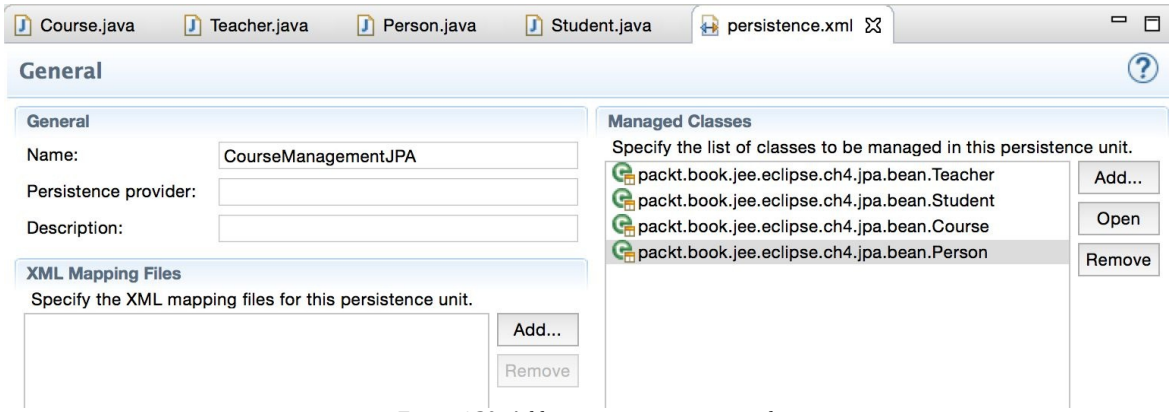


Figure 4.30: Add entities in persistence.xml

Creating database tables from entities

Follow these steps to create database tables from entities and relationships that we have modeled:

1. Right-click on the project and select JPA Tool | Generate Tables from Entities:

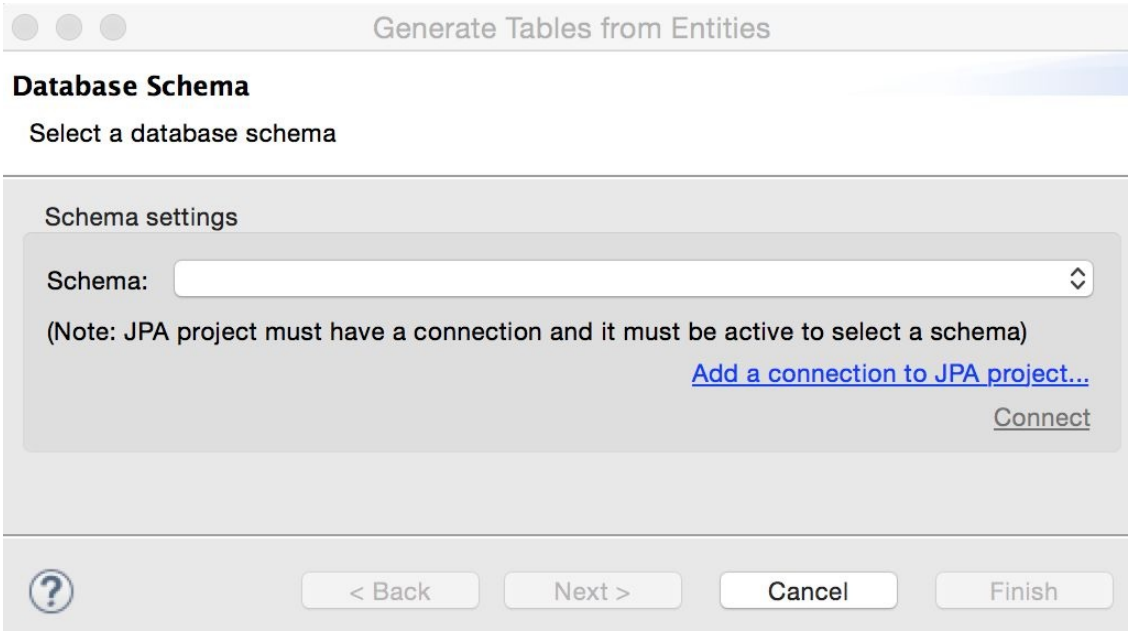


Figure 4.31: JPA Details for the courses field in Student entity

2. Because we haven't configured any schema for our JPA project, the Schema drop-down will be empty. Click the Add a connection to JPA project link:

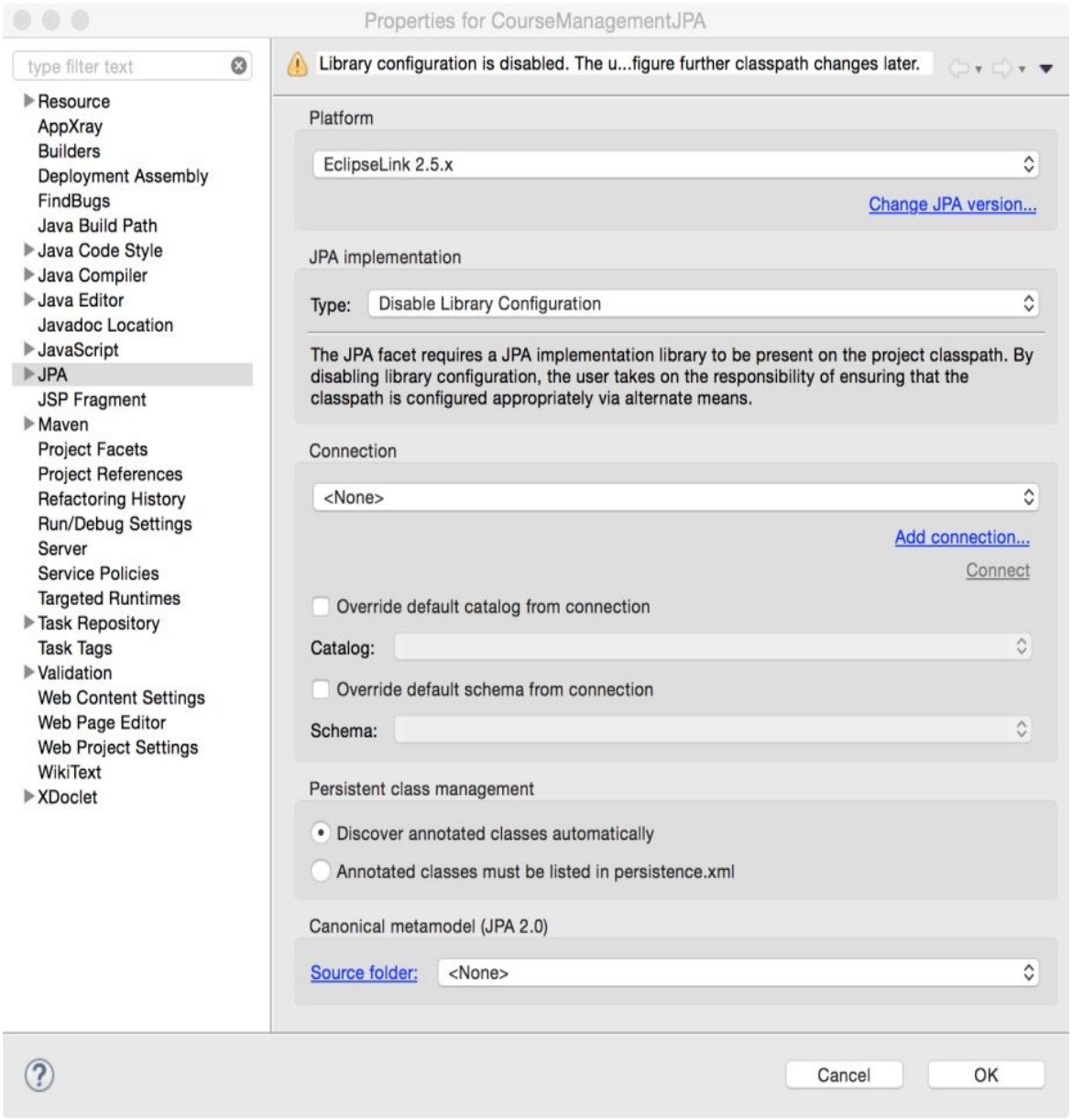


Figure 4.32: JPA project properties

3. Click the Add connection link and create a connection to the `course_management_jpa` schema we created earlier. We have already seen how to create a connection to the MySQL schema in the *Using Eclipse Data Source Explorer* section of this chapter.
4. Select `course_management_jpa` in the drop-down list shown in *Figure 4.31* and click Next:

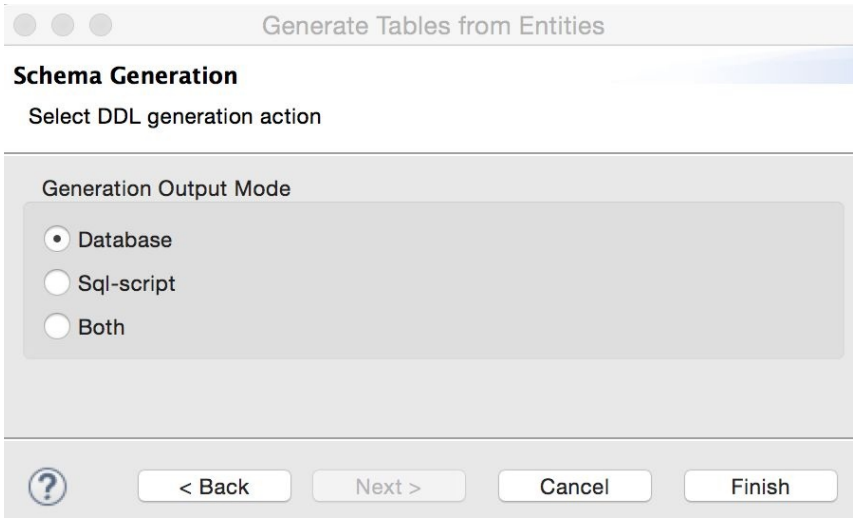


Figure 4.33: Schema Generation from entities

5. Click Finish.

Eclipse generates DDL scripts for creating tables and relationships and executes these scripts in the selected schema. Once the script is run successfully, open the Data Source Explorer view (see the *Using Eclipse Data Source Explorer* section of this chapter) and browse tables in course_management_jpa connection. Make sure that tables and fields are created according to the entities we have created:

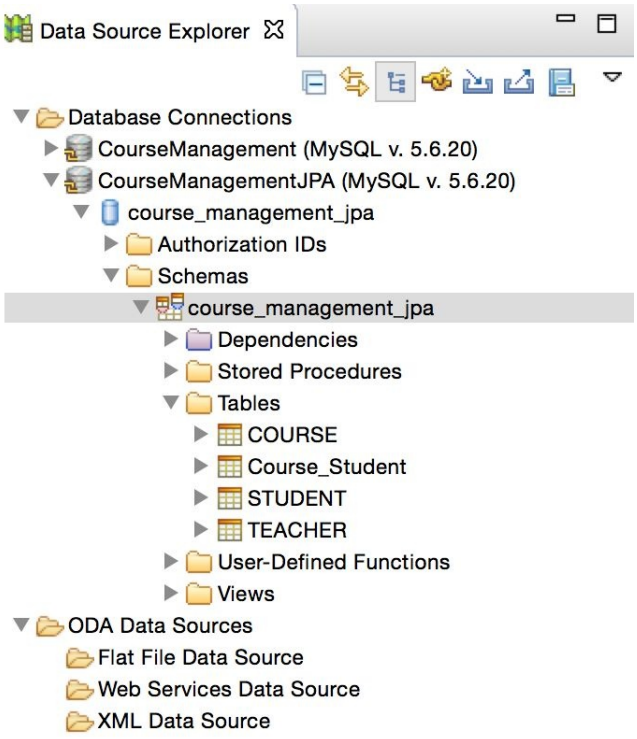


Figure 4.34: Tables created from JPA entities

This feature of Eclipse and JPA makes it very easy to update the database as you modify your entities.

Using JPA APIs to manage data

We will now create classes that use JPA APIs to manage data for our course management application. We will create service classes for Course, Teacher, and Student entities and add methods that directly access the database through JPA APIs.

As mentioned in the *JPA concepts* section, it is a good practice to cache an instance of `EntityManagerFactory` in our application. Furthermore, managed beans of JSF act as a link between the UI and the backend code, and as a conduit to transfer data between the UI and the data access objects. Therefore, they must have an instance of the data access objects (which use JPA to access data from the database). To cache an instance of `EntityManagerFactory`, we will create another managed bean, whose only job is to make the `EntityManagerFactory` instance available to other managed beans.

Create an `EntityManagerFactoryBean` class in the `packt.book.jee.eclipse.ch4.jpa.service_bean` package. This package contains all the managed beans. `EntityManagerFactoryBean` creates an instance of `EntityManagerFactory` in the constructor and provides a getter method: package packt.book.jee.eclipse.ch4.jpa.service_bean;

import javax.faces.bean.ApplicationScoped; import javax.faces.bean.ManagedBean;

import javax.persistence.EntityManagerFactory; import javax.persistence.Persistence;

//Load this bean eagerly, i.e., before any request is made @ManagedBean(name="emFactoryBean", eager=true) @ApplicationScoped

public class EntityManagerFactoryBean {

private EntityManagerFactory entityManagerFactory;

public EntityManagerFactoryBean() {

entityManagerFactory =
Persistence.createEntityManagerFactory("CourseManagementJPA"); }

public EntityManagerFactory getEntityManagerFactory() {

return entityManagerFactory;

}

}

Note the argument passed in the following: entityManagerFactory =
Persistence.createEntityManagerFactory("CourseManagementJPA");

It is the name of the persistence unit in `persistence.xml`.

Now let's create service classes that actually use the JPA APIs to access database tables.

Create a package called `packt.book.jee.eclipse.ch4.jpa.service`. Create the class named `CourseService`. Every service class will need access to `EntityManagerFactory`. So, create a private member variable as follows: private EntityManagerFactory factory;

The constructor takes an instance of `EntityManagerFactoryBean` and gets the reference of `EntityManagerFactory` from it: public CourseService(EntityManagerFactoryBean factoryBean) {

this.factory = factoryBean.getEntityManagerFactory(); }

Let's now add a function to get all `courses` from the database: public List<Course> getCourses() {

EntityManager em = factory.createEntityManager(); CriteriaBuilder cb = em.getCriteriaBuilder(); CriteriaQuery<Course> cq = cb.createQuery(Course.class); TypedQuery<Course> tq = em.createQuery(cq); List<Course> courses = tq.getResultList(); em.close();

return courses;

}

Note how `CriteriaBuilder`, `CriteriaQuery`, and `TypesQuery` are used to get all the courses. It is a type-safe way to execute the query.

See `https://javaee.github.io/tutorial/persistence-criteria.html#GJITV` for detailed discussion on how to use the JPA criteria APIs.

We could have done the same thing using **Java Persistence Query Language (JQL)**—`http://www.oracle.com/technetwork/articles/vasiliev-jpql-087123.html`—but it is not type-safe. However, here is an example of using JQL to write the `getCourses` function: public List<Course> getCourses() {

EntityManager em = factory.createEntityManager(); List<Course> courses = em.createQuery("select crs from Course crs").getResultList(); em.close();

return courses;

}

Add a method to insert the course into the database: public void addCourse (Course course) {

EntityManager em = factory.createEntityManager(); EntityTransaction txn = em.getTransaction(); txn.begin();

em.persist(course);

txn.commit();

}

The code is quite simple. We get the entity manager and then start a transaction, because it is an `update` operation. Then, we call the `persist` method on `EntityManager` by passing an instance of `Course` to save. Then, we commit the transaction. The methods to update and delete are also simple. Here is the entire source code of `CourseService`: package packt.book.jee.eclipse.ch4.jpa.service;

// imports skipped

import packt.book.jee.eclipse.ch4.jpa.bean.Course; import packt.book.jee.eclipse.ch4.jpa.service_bean.EntityManagerFactoryBean;

public class CourseService {

private EntityManagerFactory factory;

public CourseService(EntityManagerFactoryBean factoryBean) {

factory = factoryBean.getEntityManagerFactory(); }

public List<Course> getCourses() {

EntityManager em = factory.createEntityManager(); CriteriaBuilder cb = em.getCriteriaBuilder(); CriteriaQuery<Course> cq = cb.createQuery(Course.class); TypedQuery<Course> tq = em.createQuery(cq); List<Course> courses = tq.getResultList(); em.close();

return courses;

}

public void addCourse (Course course) {

EntityManager em = factory.createEntityManager(); EntityTransaction txn =

em.getTransaction(); txn.begin();

em.persist(course);

txn.commit();

}

public void updateCourse (Course course) {

EntityManager em = factory.createEntityManager(); EntityTransaction txn = em.getTransaction(); txn.begin();

em.merge(course);

txn.commit();

}

public Course getCourse (int id) {

EntityManager em = factory.createEntityManager(); return em.find(Course.class, id);

}

public void deleteCourse (Course course) {

EntityManager em = factory.createEntityManager(); EntityTransaction txn = em.getTransaction(); txn.begin();

Course mergedCourse = em.find(Course.class, course.getId()); em.remove(mergedCourse);

txn.commit();

}

}

Let's now create `StudentService` and `TeacherService` classes with the following methods: public class StudentService {

private EntityManagerFactory factory;

public StudentService (EntityManagerFactoryBean factoryBean) {

factory = factoryBean.getEntityManagerFactory(); }

public void addStudent (Student student) {

EntityManager em = factory.createEntityManager(); EntityTransaction txn = em.getTransaction(); txn.begin();

em.persist(student);

txn.commit();

}

public List<Student> getStudents() {

EntityManager em = factory.createEntityManager(); CriteriaBuilder cb = em.getCriteriaBuilder(); CriteriaQuery<Student> cq = cb.createQuery(Student.class); TypedQuery<Student> tq = em.createQuery(cq); List<Student> students = tq.getResultList(); em.close();

return students;

}

```
}


public class TeacherService {

private EntityManagerFactory factory;

public TeacherService (EntityManagerFactoryBean factoryBean) {

factory = factoryBean.getEntityManagerFactory(); }


public void addTeacher (Teacher teacher) {

EntityManager em = factory.createEntityManager(); EntityTransaction txn =
em.getTransaction(); txn.begin();

em.persist(teacher);

txn.commit();

}


public List<Teacher> getTeacher() {

EntityManager em = factory.createEntityManager(); CriteriaBuilder cb =
em.getCriteriaBuilder(); CriteriaQuery<Teacher> cq =
cb.createQuery(Teacher.class); TypedQuery<Teacher> tq = em.createQuery(cq);
List<Teacher> teachers = tq.getResultList(); em.close();

return teachers;

}


public Teacher getTeacher (int id) {
```

```
EntityManager em = factory.createEntityManager(); return
em.find(Teacher.class, id); }

}
```

Wiring user interface with JPA service classes

Now that we have all data access classes ready, we need to connect the user interface that we have created for adding courses, `addCourse.xhtml`, to pass data and get data from the JPA service classes. As mentioned previously, we are going to do this using managed beans, in this case, `CourseServiceBean`.

`CourseServiceBean` will need to create an instance of `CourseService` and call the `addCourse` method. Open `CourseServiceBean` and create a member variable as follows: private CourseService courseService ;

We also need an instance of the `EntityManagerFactoryBean` managed bean that we created earlier: @ManagedProperty(value="#{emFactoryBean}") private EntityManagerFactoryBean factoryBean;

The `factoryBean` instance is injected by the JSF runtime and is available only after the managed bean is completely constructed. However, for this bean to be injected, we need to provide a setter method. Therefore, add a setter method for `factoryBean`. We can have JSF call a method of our bean after it is fully constructed by annotating the method with `@PostConstruct`. So, let's create a method called `postConstruct`: @PostConstruct

public void init() {

courseService = new CourseService(factoryBean); }

Then, modify the `addCourse` method to call our service method: public String addCourse() {

courseService.addCourse(course);

return "listCourse";

}

Since the `listCourse.xhtml` page will need to get a list of courses, let's also add the `getCourses` method in `CourseServiceBean`: public List<Course> getCourses() {

return courseService.getCourses(); }

Here is `CourseServiceBean` after the preceding changes:
@ManagedBean(name="courseServiceBean") @RequestScoped

public class CourseServiceBean {

private CourseService courseService ;

@ManagedProperty(value="#{emFactoryBean}") private
EntityManagerFactoryBean factoryBean;

@ManagedProperty(value="#{course}") private Course course;

private String errMsg= null;

@PostConstruct

public void init() {

courseService = new CourseService(factoryBean); }

public void setFactoryBean(EntityManagerFactoryBean factoryBean)
{

this.factoryBean = factoryBean;

}

```
public Course getCourse() {
return course;
}

public void setCourse(Course course) {
this.course = course;
}

public String getErrMsg() {
return errMsg;
}

public void setErrMsg(String errMsg) {
this.errMsg = errMsg;
}

public String addCourse() {
courseService.addCourse(course); return "listCourse";
}
```

```
public List<Course> getCourses() {

return courseService.getCourses(); }



}
```

Finally, we will write the code to display a list of courses in `listCourse.xhtml`:

```
<html xmlns="http://www.w3.org/1999/xhtml"

xmlns:f="http://java.sun.com/jsf/core"

xmlns:h="http://java.sun.com/jsf/html"

xmlns:c="http://java.sun.com/jsp/jstl/core">

<h2>Courses:</h2>

<h:form>

<h:messages style="color:red"/> <h:dataTable value="#
{courseServiceBean.courses}"
var="course"> <h:column>

<f:facet name="header">ID</f:facet> <h:outputText value="#{course.id}"/>
</h:column>

<h:column>

<f:facet name="header">Name</f:facet> <h:outputText value="#
{course.name}"/> </h:column>

<h:column>

<f:facet name="header">Credits</f:facet> <h:outputText value="#
{course.credits}"
style="float:right" /> </h:column>

</h:dataTable>
```

```
</h:form>


<h:panelGroup rendered="#{courseServiceBean.courses.size() ==
0}"> <h3>No courses found</h3> </h:panelGroup>


<c:if test="#{courseServiceBean.courses.size() > 0}"> <b>Total number of
courses <h:outputText value="#{courseServiceBean.courses.size()}"/> </b>

</c:if>

<p/>

<h:button value="Add" outcome="addCourse"/> </html>
```

Because of space constraints, we will not discuss how to add functionality to delete/update courses, or to create a course with the `Teacher` field selected. Please download the source code for the examples discussed in this chapter to see completed projects.