Installing the Tomcat server

Tomcat is a web container. It supports APIs in the presentation layer described earlier. In addition, it supports JDBC and JPA. It is easy to configure and could be a good option if you do not want to use EJBs.

Download the latest version of Tomcat from `http://tomcat.apache.org/`. Unzip the downloaded file in a folder. Set the `JAVA_HOME` environment variable to point to the folder where JDK is installed (the folder path should be the JDK folder, which has `bin` as one of the subfolders). To start the server, run `startup.bat` in Command Prompt on Windows and `startup.sh` in a Terminal window on Mac and Linux. If there are no errors, then you should see the message `Server startup in --ms` or `Tomcat started`.

The default Tomcat installation is configured to use port `8080`. If you want to change the port, open `server.xml` under the `conf` folder and look for a connector declaration such as the following: <Connector port="8080" protocol="HTTP/1.1" connectionTimeout="20000" redirectPort="8443" />

Change the port value to any port number you want, though in this book we will be using the default port `8080`. Before we open the default page of Tomcat, we will add a user for administration of the Tomcat server. Open `tomcat-users.xml` under the `conf` folder using any text editor. At the end of the file, you will see commented example of how to add users. Add the following configuration before the closure of `</tomcat-users>` tag: <role rolename="manager-gui"/> <user username="admin" password="admin" roles="manager-gui"/>

Here, we are adding a user `admin`, with password also as `admin`, to a role called `manager-gui`. This role has access to web pages for managing an application in Tomcat. This and other security roles are defined in `web.xml` of the `manager` application. You can find it at `webapps/manager/WEB-INF/web.xml`.

For more information for managing Tomcat server, refer to `http://tomcat.apache.org/tomcat-8.0-doc/manager-howto.html`.

After making the preceding changes, open a web browser and browse to

`http://localhost:8080` (modify the port number if you have changed the default port). You will see the following default Tomcat page:

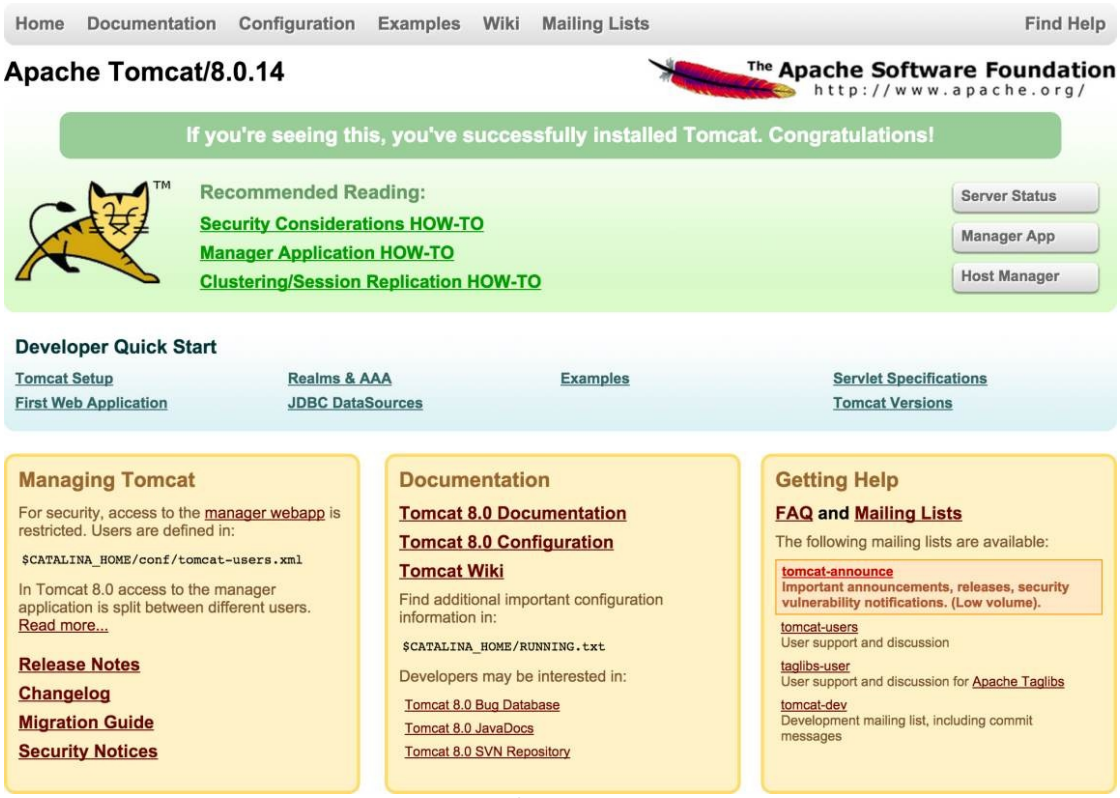

Figure 1.6: The default Tomcat web application

Click on the Manager App button on the right. You will be asked for the username and password. Enter the username and password you configured in `tomcat-users.xml` for `manager-gui`, as described earlier. After you are successfully logged in, you will see the Tomcat Web Application Manager page, as shown in *Figure 1.7*. You can see all the applications deployed in Tomcat in this page. You can also deploy your applications from this page:

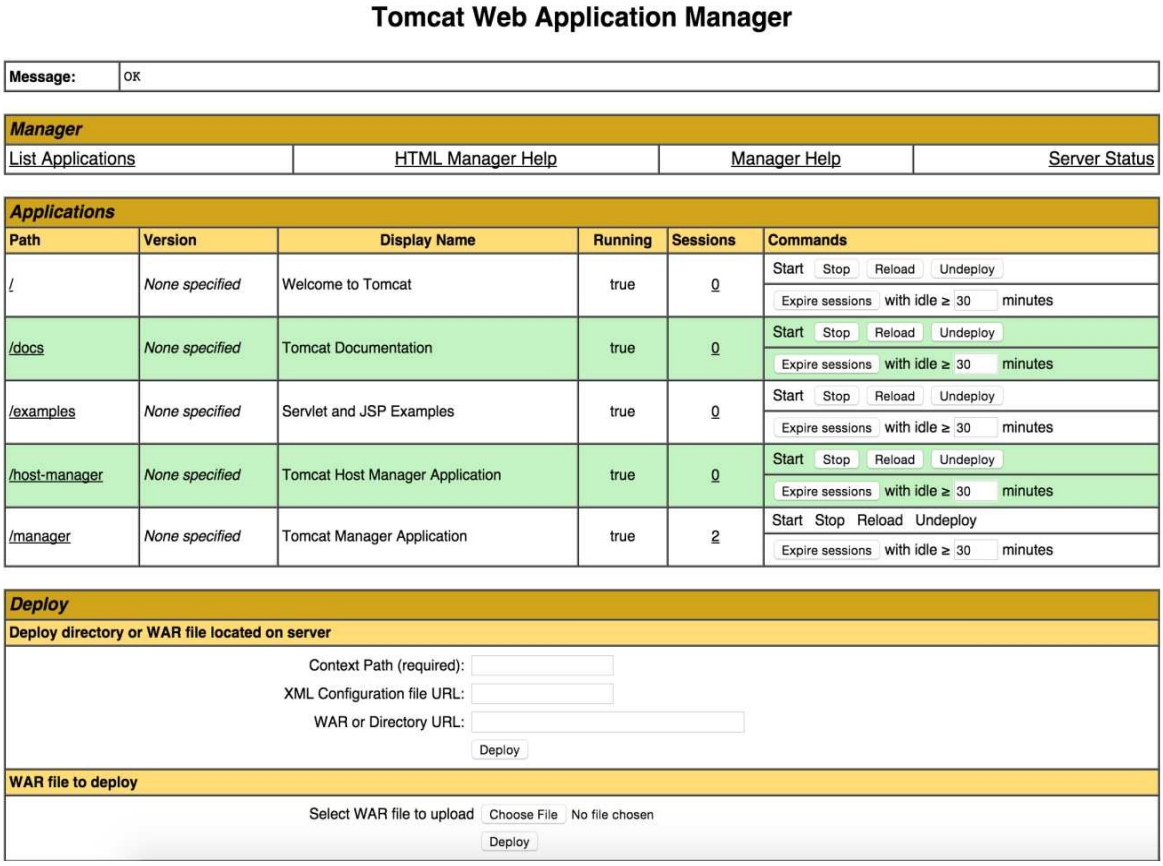

Figure 1.7: Tomcat Web Application Manager

To stop the Tomcat server, press *Ctrl*/*cmd* + *C* or run the shutdown script in the `bin` folder.

Installing the GlassFish server

Download GlassFish from `https://glassfish.java.net/download.html`. GlassFish comes in two flavors: Web Profile and Full Platform. Web Profile is like Tomcat, which does not include EJB support. So download the Full Platform.

Unzip the downloaded file in a folder. The default port of the GlassFish server is `8080`. If you want to change that, open `glassfish/domains/domain1/config/domain.xml` in a text editor (you could open it in Eclipse too, using the File | Open File menu option) and look for `8080`. You should see it in one of the `<network-listener>`. Change the port if you want to (which may be the case if some other application is already using that port).

To start the server, run the `startserv` script (`.bat` or `.sh` depending on the OS you use). Once the server has started, open a web browser and browse to `http://localhost:8080`. You should see a page like the following:

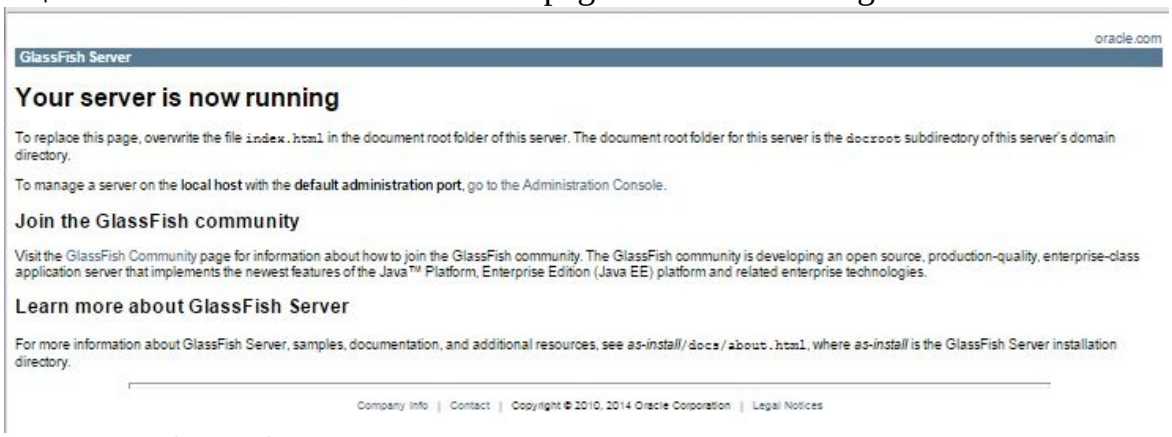

Figure 1.8: The default Glassfish web application This page is located at `glassfish/domains/domain1/docroot/index.html`. Click on the go to the Administration Console link in the page to open the GlassFish administrator (see the following screenshot):

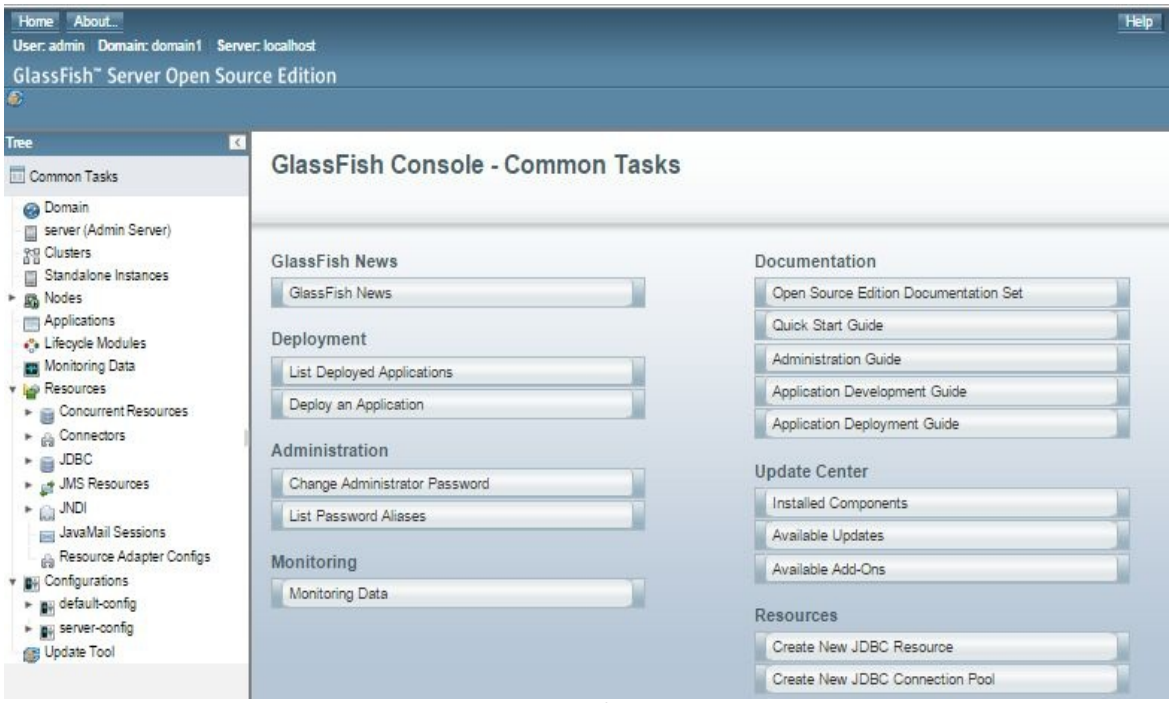

Figure 1.9: The Glassfish administrator

For details on administrating the GlassFish server, refer to `https://javaee.github.io/glassfish/doc/5.0/administration-guide.pdf`.

To stop the GlassFish Server, run the `stopserv` script in the `glassfish/bin` folder.

Installing MySQL

We will be using a MySQL database for many of the examples in this book. The following sections describe how to install and configure MySQL for different platforms.

We would like to install MySQL Workbench too, which is a client application to manage MySQL Server. Download MySQL Workbench from https://dev.mysql.com/downloads/workbench/.

Installing MySQL on Windows

Download MySQL Community Server from `http://dev.mysql.com/downloads/mysql/`. You can either download the web installer or the all-in-one installer. The web installer would download only those components that you have selected. The following instructions show the download options using the web installer.

The web installer first downloads a small application, and it gives you options to select the components that you want to install:

1. Select the Custom option and click on Next:

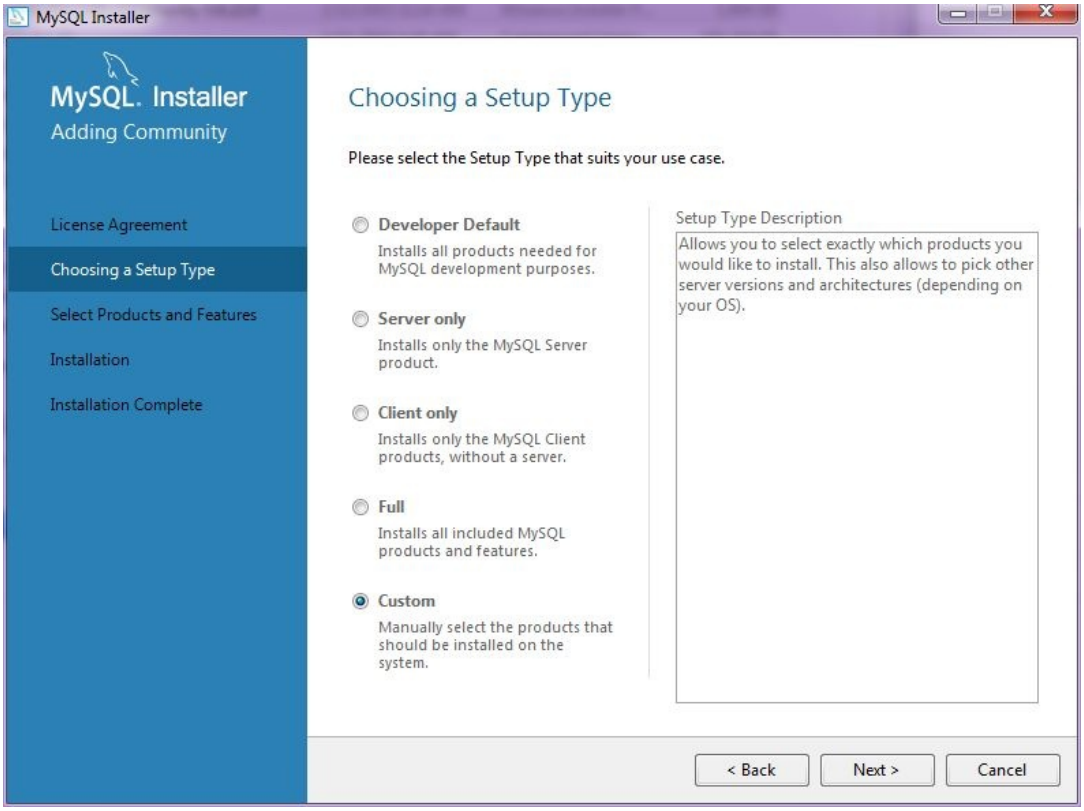

Figure 1.10: MySQL Installer for Windows

2. Select the MySQL Server and MySQL Workbench products and complete the installation. During the installation of the server, you will be asked to set the `root` password and given the option to add more users. It is always a

good idea to add a user other than root for applications to use:

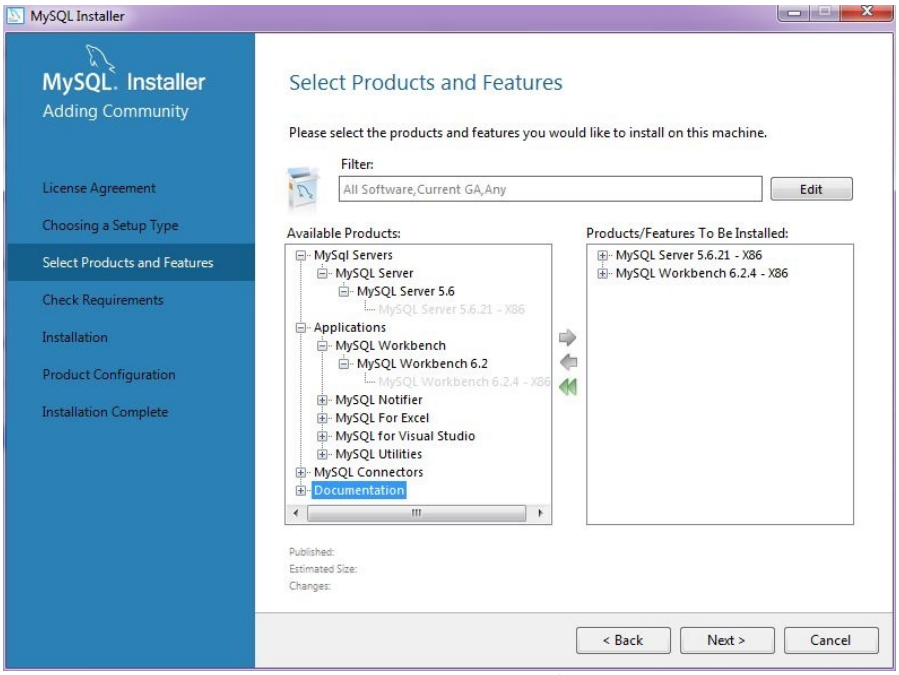

Figure 1.11: Select MySQL products and features to Install

3. Make sure you select All Hosts when adding a user so that you are able to access MySQL database from any remote machine that has network access to the machine where MySQL is installed:

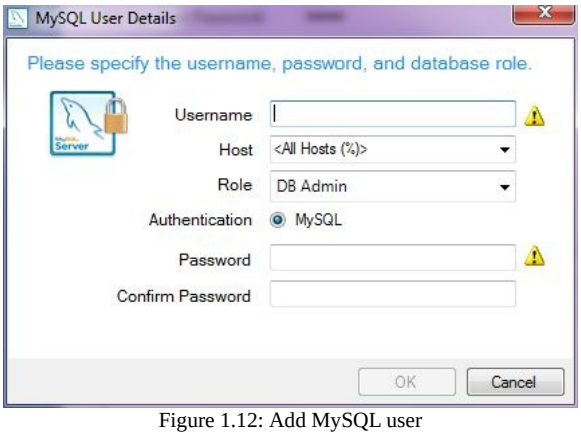

Figure 1.12: Add MySQL user

4. Run MySQL Workbench after installation. You will find that the default connection to the local MySQL instance is already created for you:

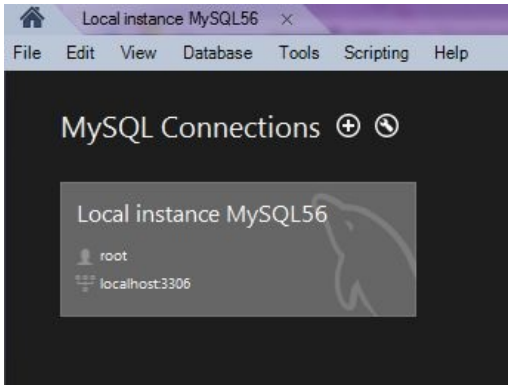

Figure 1.13: MySQL Workbench connections

5. Click on the local connection and you will be asked to enter the `root` password. Enter the `root` password that you typed during the installation of MySQL Server. MySQL Workbench opens and displays the default test schema:

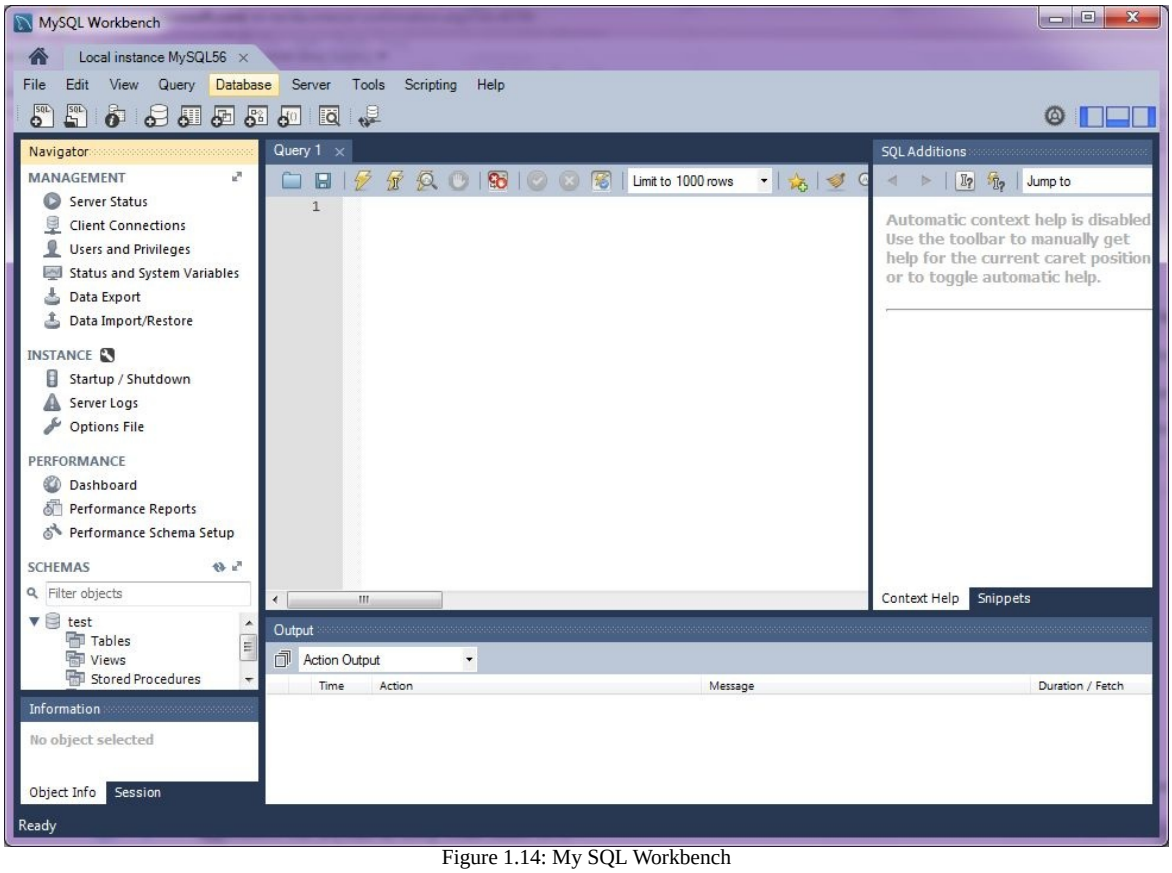

Figure 1.14: My SQL Workbench

Installing MySQL on macOS X

OS X versions before 10.7 had MySQL Server installed by default. If you are using OS X 10.7 or later, then you will need to download and install MySQL Community Server from http://dev.mysql.com/downloads/mysql/.

There are many different ways to install MySQL on OS X. See http://dev.mysql.com/doc/refman/5.7/en/osx-installation.html for installation instructions for OS X. Note that users on OS X should have administrator privileges to install MySQL Server.

Once you install the server, you can start it either from Command Prompt or from the system preferences:

1. To start it from Command Prompt, execute the following command in the Terminal:

```
sudo /usr/local/mysql/support-files/mysql.server start
```

2. To start it from System Preferences, open the preferences and click the MySQL icon:

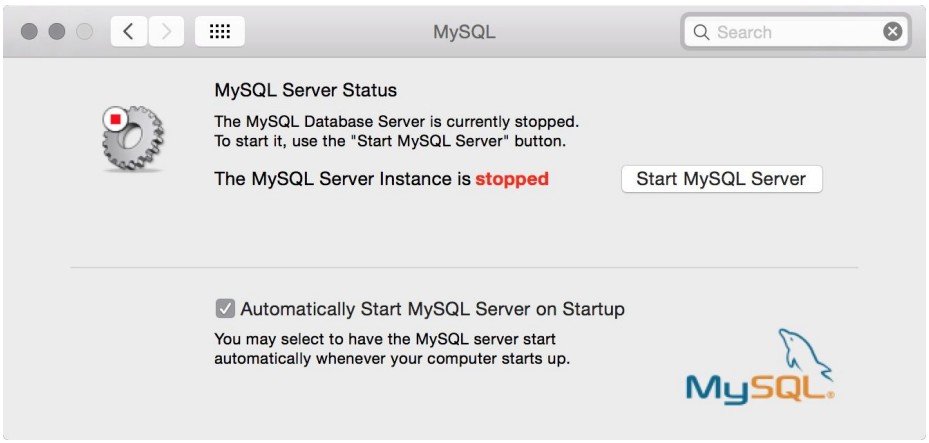

Figure 1.15: MySQL System Preferences - OS X

3. Click the Start MySQL Server button.

Installing MySQL on Linux

There are many different ways to install MySQL on Linux. Refer to https://dev.mysql.com/doc/refman/5.7/en/linux-installation.html for details.

Creating MySQL users

You can create MySQL users either from Command Prompt or by using MySQL Workbench:

1. To execute SQL and other commands from Command Prompt, open the Terminal and type the following command:

```
mysql -u root -p<root_password>
```

2. Once logged in successfully, you will see the `mysql` Command Prompt:

```
mysql>
```

3. To create a user, first select the `mysql` database:

```
mysql>use mysql;
Database changed
mysql>create user 'user1'@'%' identified by 'user1_pass';
mysql>grant all privileges on *.* to 'user1'@'%' with grant option
```

The preceding command will create a user named `'user1'` with password `'user1_pass'` having all privileges, for example to insert, update, and select from the database. And because we have specified the host as `'%'`, this user can access the server from any host.

See `https://dev.mysql.com/doc/refman/5.7/en/adding-users.html` for more details on adding users to MySQL database

If you prefer a **graphical user interface** (**GUI**) to manage the users, then run MySQL Workbench, connect to the local MySQL server (see *Figure 1.13* MySQL Workbench connections), and then click on Users and Privileges under the Management section:

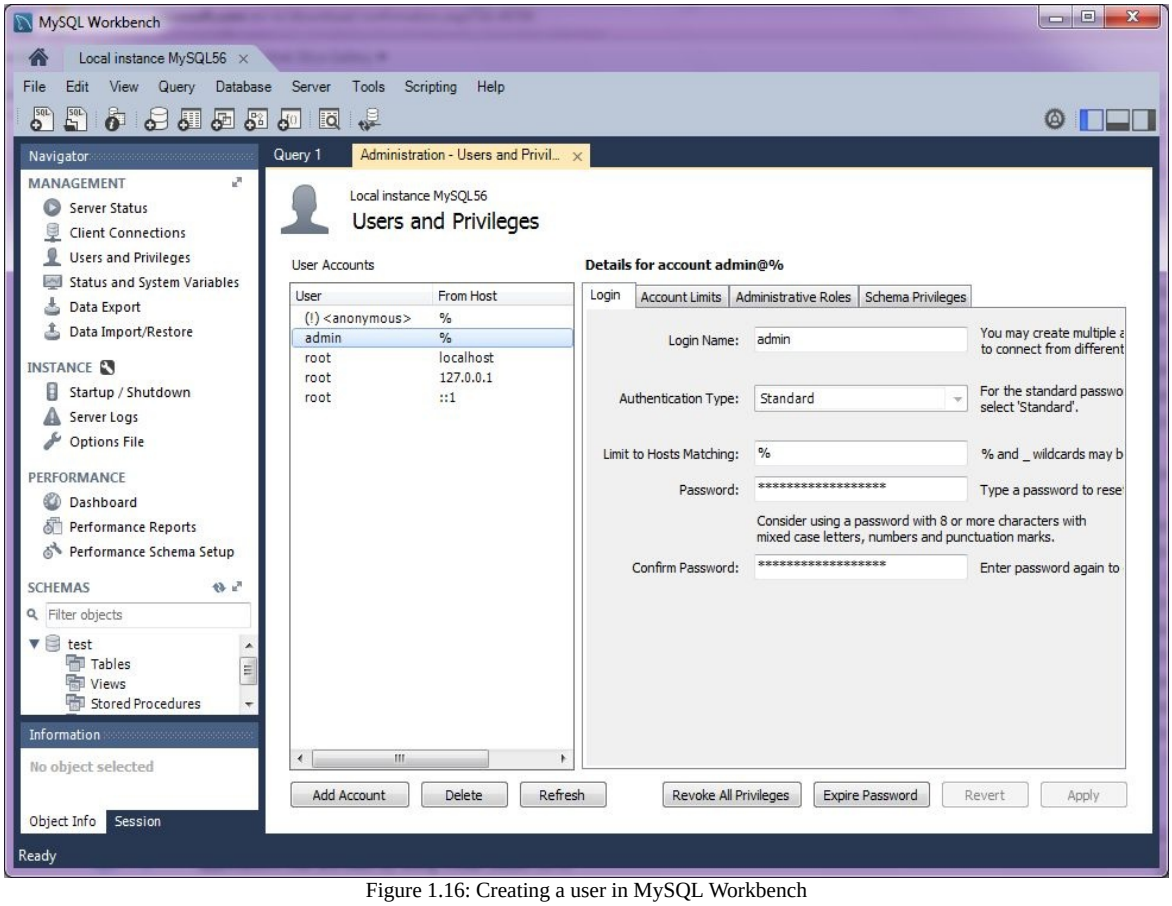

Figure 1.16: Creating a user in MySQL Workbench

Having installed all the preceding products, you should be in a position to start developing JEE applications. We may need some additional software, but we will see how to install and configure it at the appropriate time.

Creating a Simple JEE Web Application

The previous chapter gave you a brief introduction to JEE and Eclipse. We also learned how to install the Eclipse JEE package and also how to install and configure Tomcat. Tomcat is a servlet container and it is easy to use and configure. Therefore, many developers use it to run JEE web applications on local machines.

In this chapter, we will cover the following topics:

- Configuring Tomcat in Eclipse and deploying web applications from within Eclipse
- Using different technologies to create web applications in JEE, for example, JSP, JSTL, JSF, and servlets
- Using the Maven dependency management tool

Configuring Tomcat in Eclipse

We will perform the following steps to configure Tomcat in Eclipse:

1. In the Java EE perspective of Eclipse, you will find the Servers tab at the bottom. Since no server is added yet, you will see a link in the tab as shown in the following screenshot—No servers are available. Click this link to create a new server....

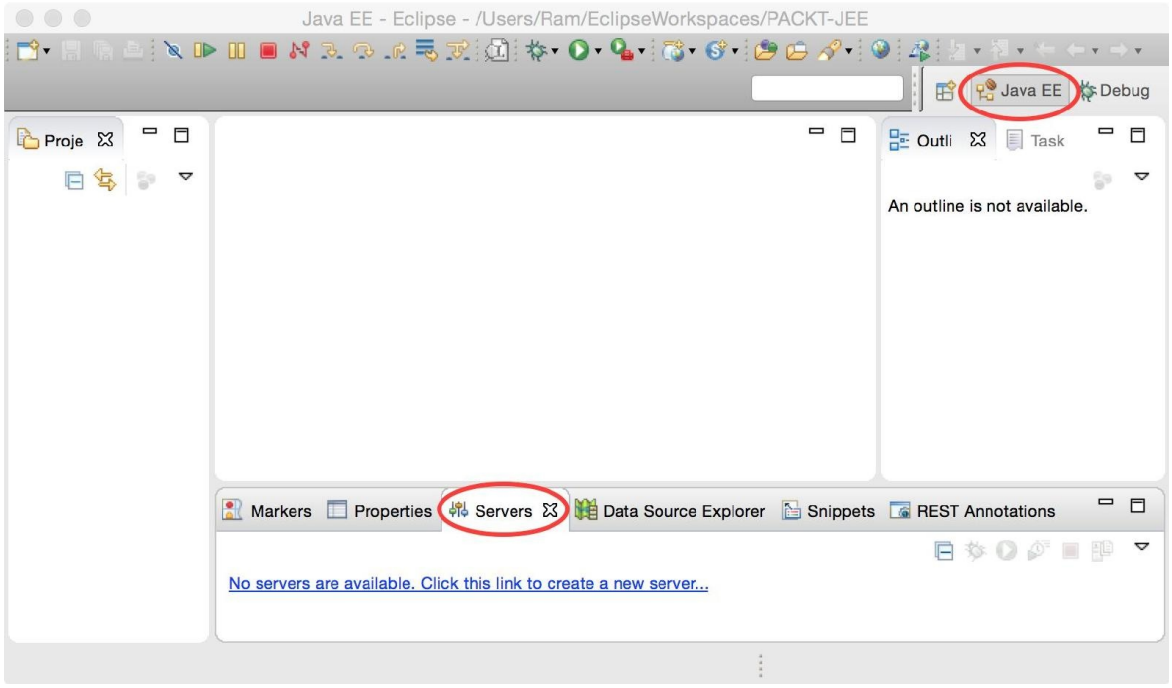

Figure 2.1: The Servers tab in Eclipse JEE

2. Click the link in the Servers tab to add a new server.
3. Expand the `Apache` group and select the Tomcat version that you have already installed. If Eclipse and the Tomcat server are on the same machine, then leave Server's host name as `localhost`. Otherwise, enter hostname or IP address of the Tomcat server. Click Next:

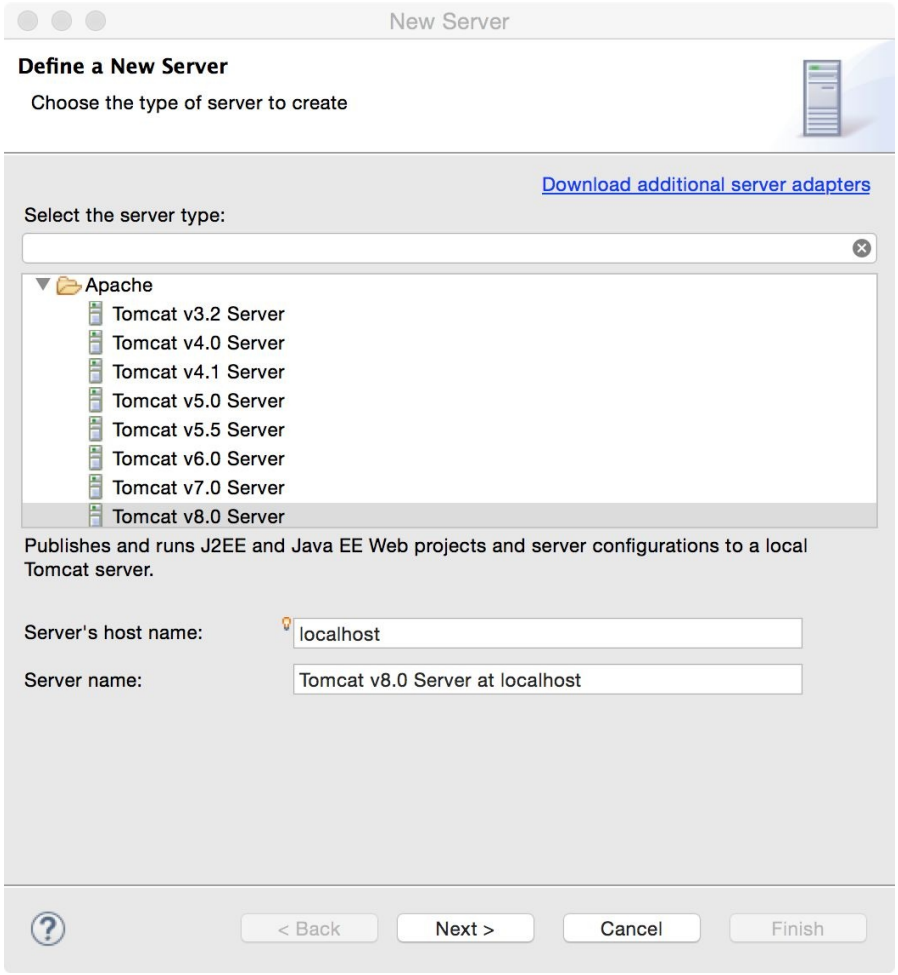

Figure 2.2: Selecting a server in the New Server wizard

4. Click the Browse... button and select the folder where Tomcat is installed.

5. Click Next until you complete the wizard. At the end of it, you will see the Tomcat server added to the Servers view. If Tomcat is not already started, you will see the status as stopped.

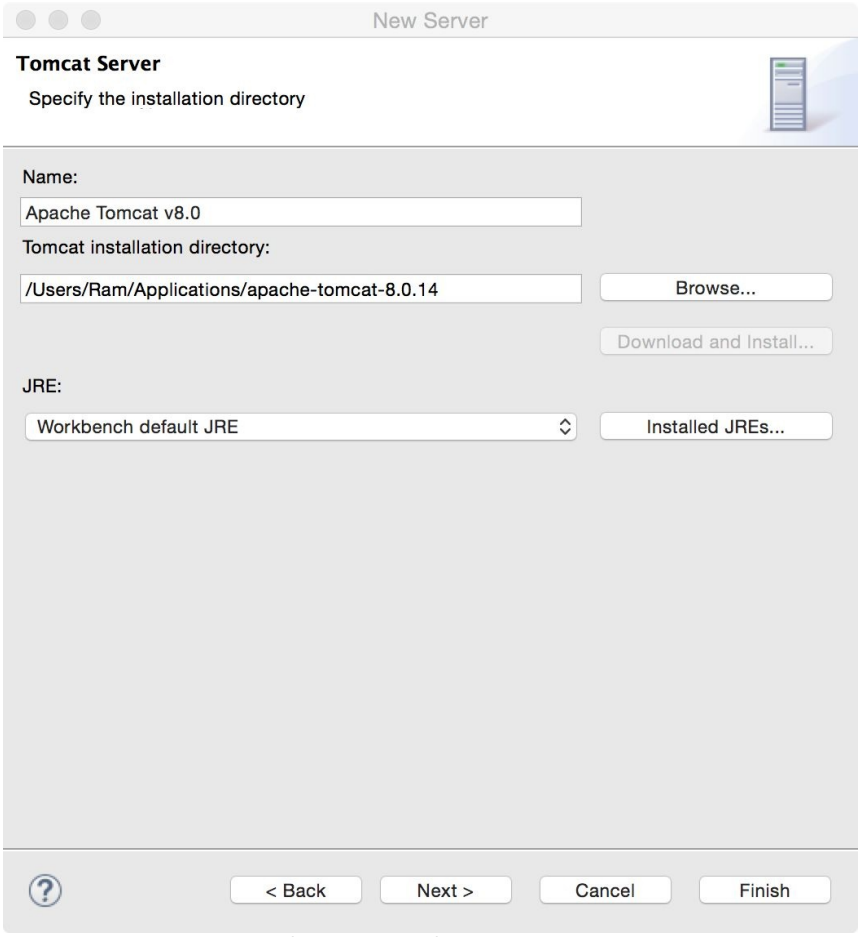

Figure 2.3: Configuring Tomcat folder in the New Server wizard

6. To start the server, right-click on the server and select Start. You can also start the server by clicking the Start button in the toolbar of the Server view.

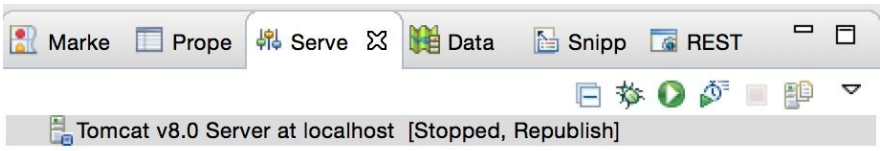

Figure 2.4: The Tomcat server added to the Servers view Once the server is started successfully, you will see the status c `Started`. If you click on the Console tab, you will see console messages that the Tomcat server outputs during start

If you expand the Servers group in the Project Explorer view, you will see the Tomcat server that you just added. Expand th server node to view configuration files. This is an easy way to edit the Tomcat configuration so that you don't have to go lo configuration files in the filesystem.

Double-click `server.xml` to open it in the XML editor. You get the Design view as well as the Source view (two tabs at the b editor). We have learned how to change the default port of Tomcat in the last chapter. You can easily change that in the Ecli by opening `server.xml` and going to the Connector node. If you need to search the text, you can switch to the Source tab (at of the editor).

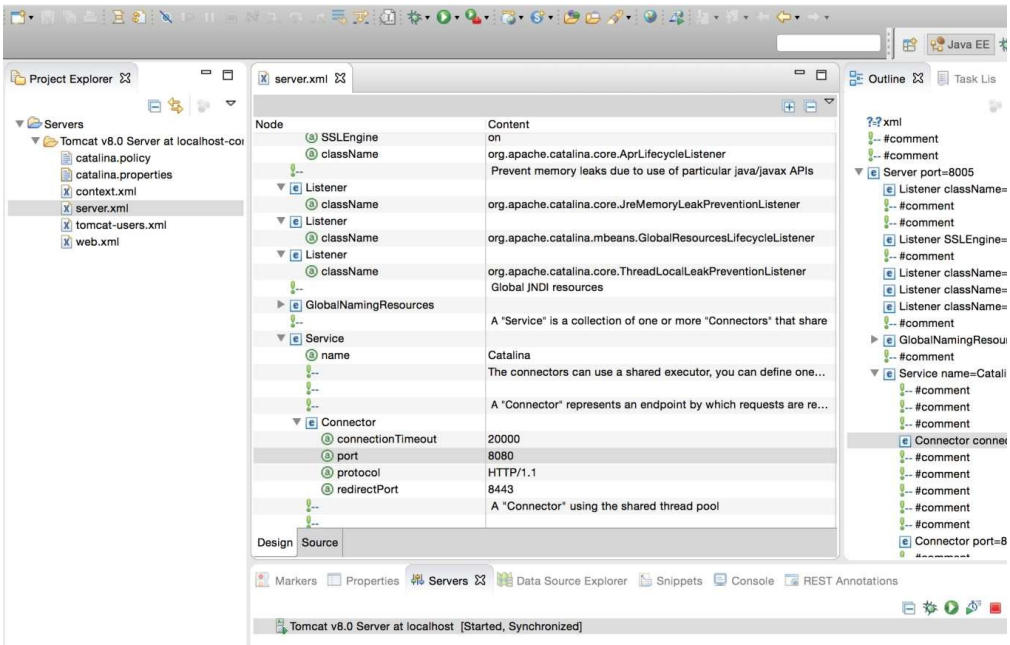

Figure 2.5: Open server.xml You can also easily edit `tomcat-users.xml` to add/edit Tomcat users. Recall that we added a Tomcat user in `Chapter 1`, *Introducing JEE and Eclip* the Tomcat server.

By default, Eclipse does not change anything in the Tomcat installation folder when you add the server in Eclipse. Instead, it creates a folder in the workspace and copies Tomcat config folder. Applications that are deployed in Tomcat are also copied and published from this folder. This works well in development, when you do not want to modify Tomcat settings or an deployed in the server. However, if you want to use the actual Tomcat installation folder, then you need to modify server settings in Eclipse. Double-click the server in the Servers view editor.

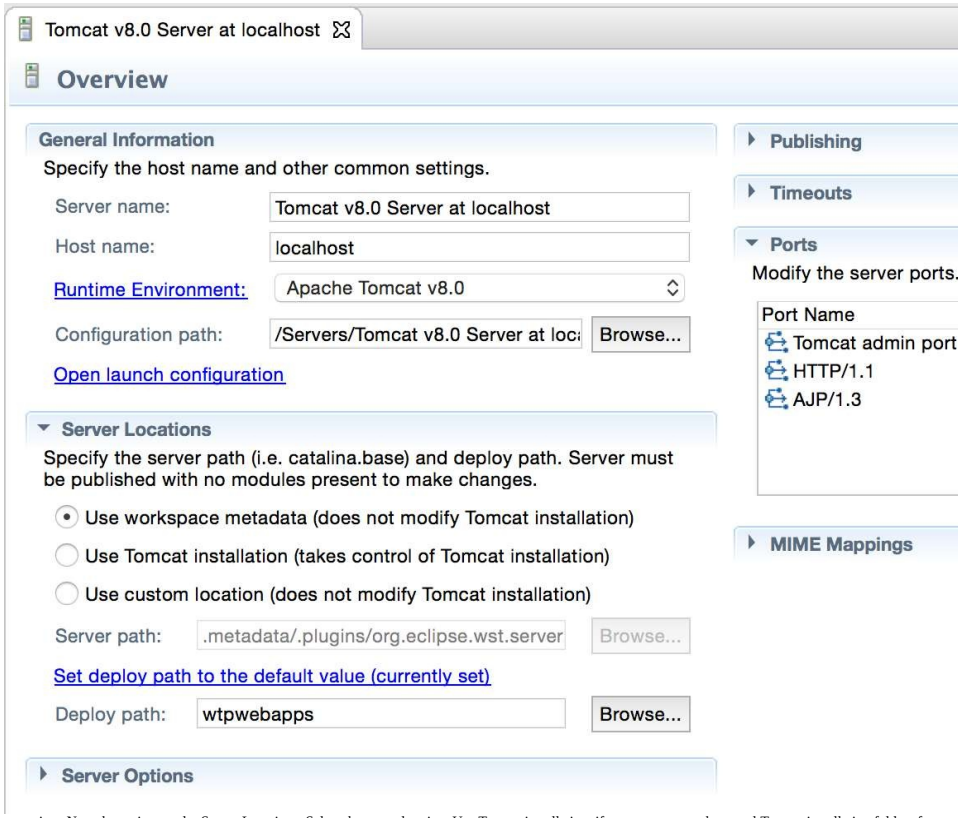

Figure 2.6: Tomcat settings Note the options under Server Locations. Select the second option, Use Tomcat installation, if you want to use the actual Tomcat installation folders for con publishing applications from within Eclipse.

JavaServer Pages

We will start with a project to create a simple JSP. We will create a login JSP that submits data to itself and validates the user.

Creating a dynamic web project

We will perform the following steps to create a dynamic web project:

1. Select the File | New | Other menu. This opens the selection wizard. At the top of the wizard, you will find a textbox with a cross icon on the extreme right side.
2. Type `web` in the textbox. This is the filter box. Many wizards and views in Eclipse have such a filter textbox, which makes finding items very easy.

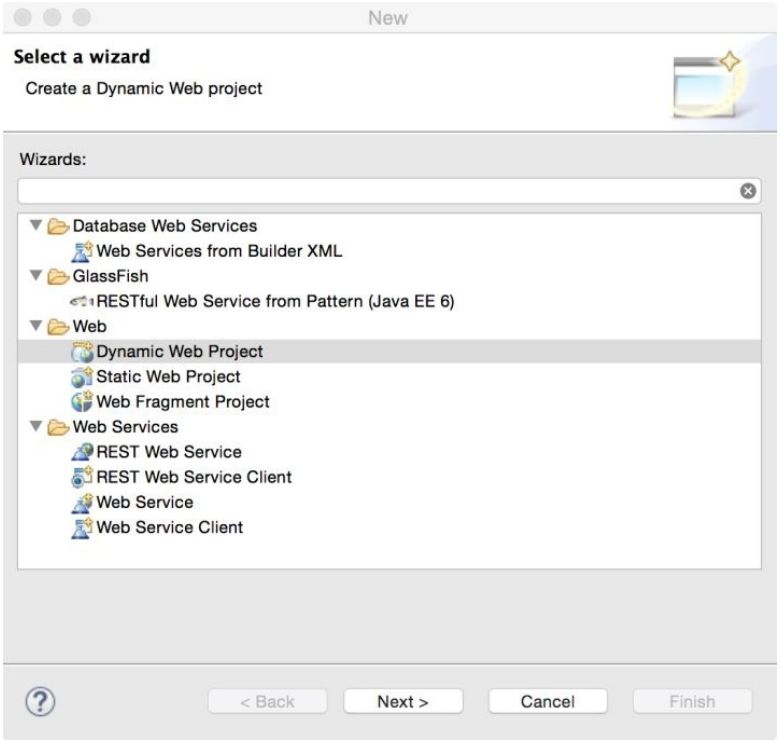

Figure 2.7: New selection wizard

3. Select Dynamic Web Project and click Next to open the Dynamic Web Project wizard. Enter project name, for example, `LoginSampleWebApp`. Note that the Dynamic web module version field in this page lists Servlet API version numbers. Select version 3.0 or greater. Click Next.

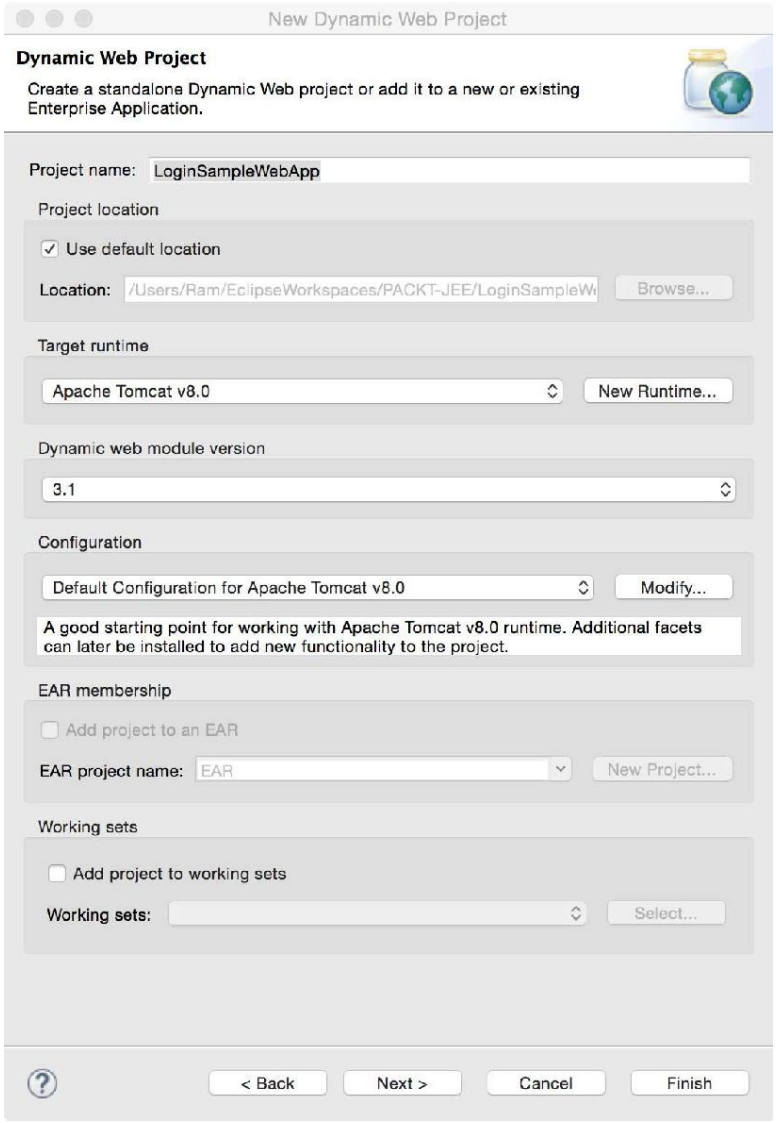

Figure 2.8: New Dynamic Web Project wizard

4. Click Next in the following pages and click Finish on the last page to create a `LoginSimpleWebApp` project. This project is also added to Project Explorer.

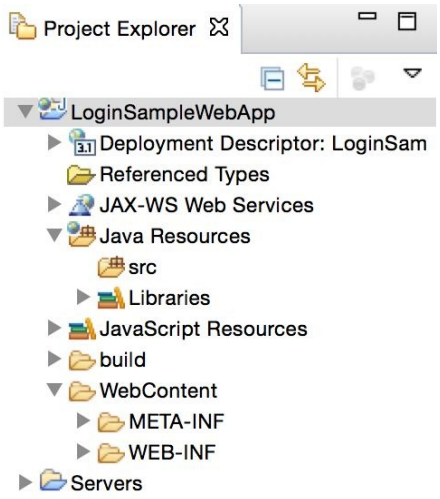

Figure 2.9: New web project

Java source files go in the `src` folder under `Java Resources`. Web resources such as the HTML, JS, and CSS files go in the `WebContent` folder.

In the next section, we will create a JSP page for login.

To keep the page simple in the first JSP, we will not follow many of the best practices. We will have the UI code mixed with the application business code. Such design is not recommended in real applications, but could be useful for quick prototyping. We will see how to write better JSP code with clear separation of the UI and business logic later in the chapter.

Creating JSP

We will perform the following steps to create the JSP:

1. Right-click on the `WebContent` folder and select New | JSP File. Name it `index.jsp`. The file will open in the editor with the split view. The top part shows the design view, and the bottom part shows the code. If the file is not opened in the split editor, right-click on `index.jsp` in the Project Explorer and select Open With | Web Page Editor.

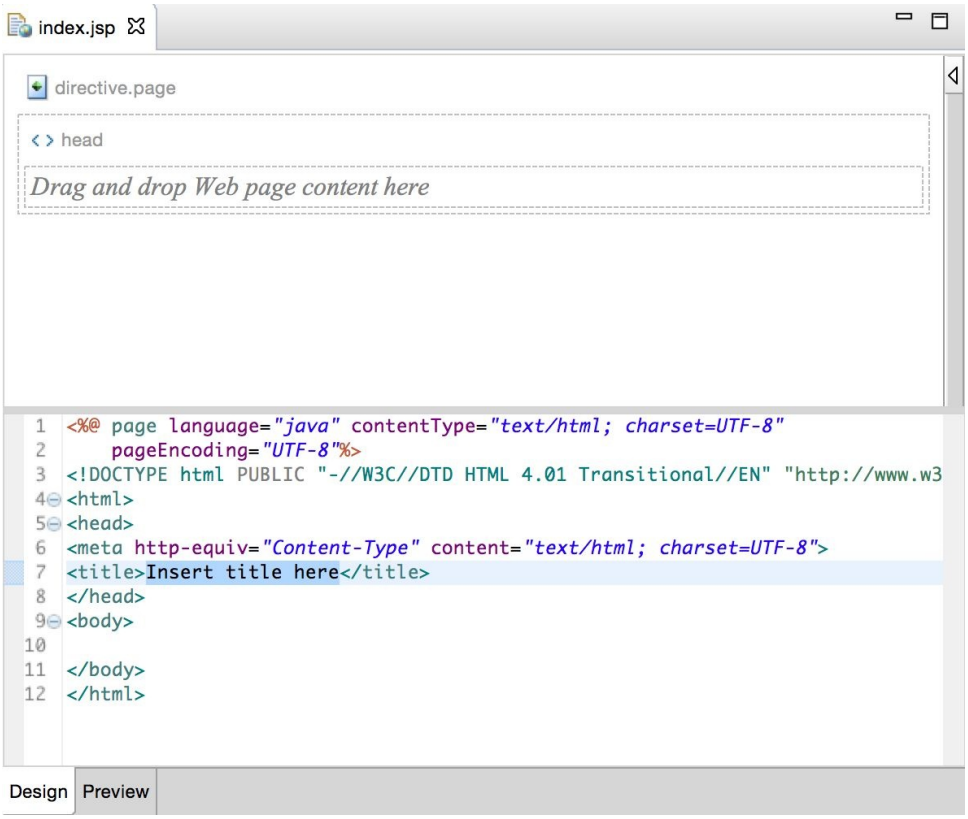

Figure 2.10: The JSP editor

2. If you do not like the split view and want to see either the full design view or the full code view, then use appropriate toolbar buttons at the top right, as shown in the following screenshot:

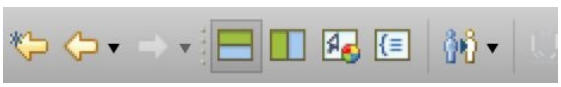

Figure 2.11: The JSP editor display buttons

3. Change the title from `Insert title here` to `Login`.
4. Let's now see how Eclipse provides code assistance for HTML tags. Note that input fields must be in a `form` tag. We will add a `form` tag later. Inside the `body` tag, type the `User Name:` label. Then, type `<`. If you wait for a moment, Eclipse pops up the code assist window showing options for all the valid HTML tags. You can also invoke code assist manually.
5. Place a caret just after `<` and press *Ctrl* + *Spacebar*.

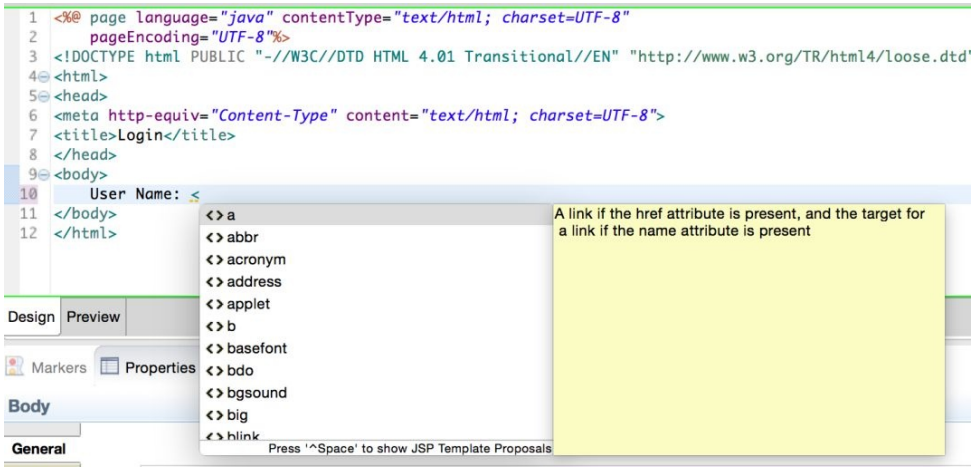

Figure 2.12: HTML code assist in JSP

Code assist works on partial text too; for example, if you invoke code assist after text `<i`, you will see a list of HTML tags starting with `i` (`i`, `iframe`, `img`, `input`, and so on). You can also use code assist for tag attributes and attribute values.
For now, we want to insert the `input` field for username.

6. Select `input` from the code assist proposals, or type it.
7. After the `input` element is inserted, move the caret inside the closing `>` and invoke code assist again (*Ctrl*/*Cmd* + *Spacebar*). You will see the list of proposals for the attributes of the `input` tag.

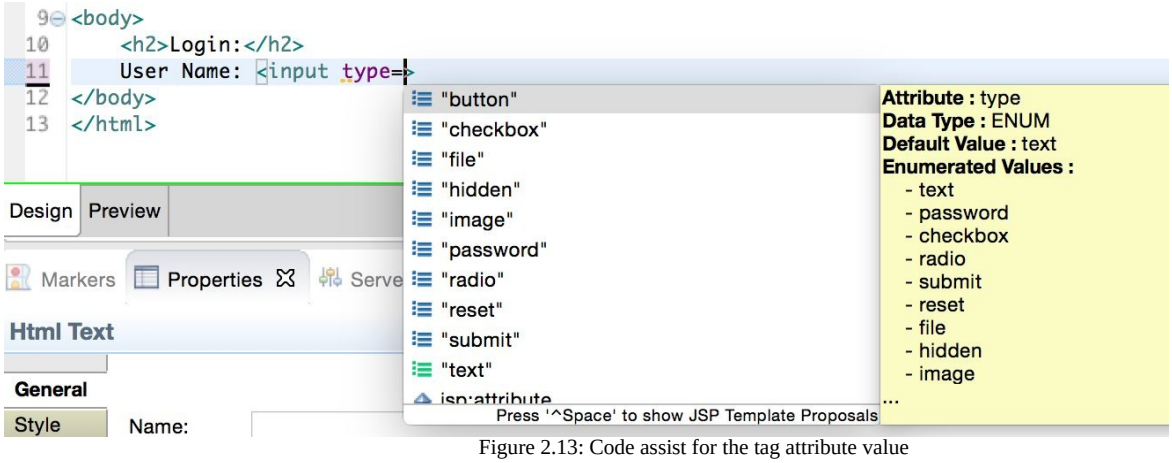

Figure 2.13: Code assist for the tag attribute value

8. Type the following code to create a login form:

```
<body>
  <h2>Login:</h2>
  <form method="post">
    User Name: <input type="text" name="userName"><br>
    Password: <input type="password" name="password"><br>
    <button type="submit" name="submit">Submit</button>
    <button type="reset">Reset</button>
  </form>
</body>
```

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

If you are using the split editor (design and source pages), you can see the login form rendered in the design view. If you want to see how the page would look in the web browser, click the Preview tab at the bottom of the editor. You will see that the web page is displayed in the browser view inside the editor. Therefore, you don't need to move out of Eclipse to test your web pages.

Figure 2.14: Design and Source views

If you click on any user interface control in the design view, you will see its properties in the Properties view (see *Figure 2.14*). You can edit properties, such as Name and Value of the selected element. Click on the Style tab of the Properties window to edit CSS styles of the element.

We have not specified the `action` *attribute in the previous form. This attribute specifies a URL to which the form data is to be posted when the user clicks the Submit button. If this attribute is not specified, then the request or the form data would be submitted to the same page; in this case, the form data would be submitted to* `index.jsp`*. We will now write the code to handle form data.*

As mentioned in `Chapter 1`, *Introducing JEE and Eclipse*, you can write Java code and the client-side code (HTML, CSS, and JavaScript) in the same JSP. It is not considered good practice to mix Java code with HTML code, but we will do that anyway in this example to keep the code simpler. Later in the book, we will see how to make our code modular.

Java code is written in JSP between `<%` and `%>`; such Java code blocks in JSP are called **scriptlets**. You can also set page-level attributes in JSP. They are called **page directives** and are included between `<%@` and `%>`. The JSP that we created already has a page directive to set the content type of the page. The content type tells the browser the type of response (in this case, `html/text`) returned by the server. The browser displays an appropriate response based on the content type:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
```

In JSP you have access to a number of objects to help you process and generate the response, as described in the following table:

Object name	**Type**
`request`	`HttpServletRequest` (`http://docs.oracle.com/javaee/7/api/javax/servlet/http/HttpServletRequest.html`). Use this to get request parameters and other request-related data.
`response`	`HttpServletResponse` (`http://docs.oracle.com/javaee/7/api/javax/servlet/http/HttpServletResponse.html`). Use this to send a response.
`out`	`JSPWriter` (`http://docs.oracle.com/javaee/7/api/javax/servlet/jsp/JspWriter.html`). Use this to generate a text response.
`session`	`HttpSession` (`http://docs.oracle.com/javaee/7/api/javax/servlet/http/HttpSession.html`). Use this to get or put objects in the session.
`application`	`ServletContext` (`http://docs.oracle.com/javaee/7/api/javax/servlet/ServletContext.html`). Use this to get or put objects in the context, which

	are shared across all JSPs and servlets in the same application.

In this example, we are going to make use of `request` and `out` objects. We will first check whether the form is submitted using the `POST` method. If true, we will get values of username and password fields. If the credentials are valid (in this example, we are going to hardcode username and the password as `admin`), we will print a welcome message:

```
<%
  String errMsg = null;
  //first check whether the form was submitted
  if ("POST".equalsIgnoreCase(request.getMethod()) &&
   request.getParameter("submit") != null)
  {
    //form was submitted
    String userName = request.getParameter("userName");
    String password = request.getParameter("password");
    if ("admin".equalsIgnoreCase(userName) &&
     "admin".equalsIgnoreCase(password))
    {
      //valid user
      System.out.println("Welcome admin !");
    }
    else
    {
      //invalid user. Set error message
      errMsg = "Invalid user id or password. Please try again";
    }
  }
%>
```

We have used two built-in objects in the preceding code—`request` and `out`. We first check whether the form was submitted —`"POST".equalsIgnoreCase(request.getMethod()`. Then, we check whether the submit button was used to post the form—`request.getParameter("submit") != null`.

We then get the username and the password by calling the `request.getParameter` method. To keep the code simple, we compare them with the hardcoded values. In the real application, you would most probably validate credentials against a database or some naming and folder service. If the credentials are valid, we print a message by using the `out` (`JSPWriter`) object. If the credentials are not valid, we set an error message. We will print the error message, if any, just before the login form:

```
<h2>Login:</h2>
  <!-- Check error message. If it is set, then display it -->
  <%if (errMsg != null) { %>
    <span style="color: red;"><%=;"><%=;"><%=errMsg %></span>
  <%} %>
  <form method="post">
  ...
  </form>
```

Here, we start another Java code block by using `<%%>`. If an error message is not null, we display it by using the `span` tag. Notice how the value of the error message is printed—`<%=errMsg %>`. This is a short syntax for `<%out.print(errMsg);%>`. Also notice that the curly brace that started in the first Java code block is completed in the next and separate Java code block. Between these two code blocks you can add any HTML code and it will be included in the response only if the conditional expression in the `if` statement is evaluated to true.

Here is the complete code of the JSP we created in this section:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!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>Login</title>
</head>
<%
  String errMsg = null;
  //first check whether the form was submitted
  if ("POST".equalsIgnoreCase(request.getMethod()) &&
   request.getParameter("submit") != null)
  {
    //form was submitted
    String userName = request.getParameter("userName");
    String password = request.getParameter("password");
    if ("admin".equalsIgnoreCase(userName) &&
     "admin".equalsIgnoreCase(password))
    {
      //valid user
      out.println("Welcome admin !");
      return;
    }
    else
    {
      //invalid user. Set error message
      errMsg = "Invalid user id or password. Please try again";
    }
  }
%>
<body>
  <h2>Login:</h2>
  <!-- Check error message. If it is set, then display it -->
  <%if (errMsg != null) { %>
    <span style="color: red;"><%out.print(errMsg); %></span>
```

```
  <%} %>
  <form method="post">
    User Name: <input type="text" name="userName"><br>
    Password: <input type="password" name="password"><br>
    <button type="submit" name="submit">Submit</button>
    <button type="reset">Reset</button>
  </form>
</body>
</html>
```

Running JSP in Tomcat

To run the JSP we created in the previous section in the web browser, you will need to deploy the application in a servlet container. We have already seen how to configure Tomcat in Eclipse. Make sure that Tomcat is running by checking its status in the Servers view of Eclipse:

Figure 2.15: Tomcat started in the Servers view

There are two ways to add a project to a configured server so that the application can be run on the server:

1. Right-click on the server in the Servers view and select the Add and Remove option. Select your project from the list on the left (Available resources) and click Add to move it to the Configured list. Click Finish.

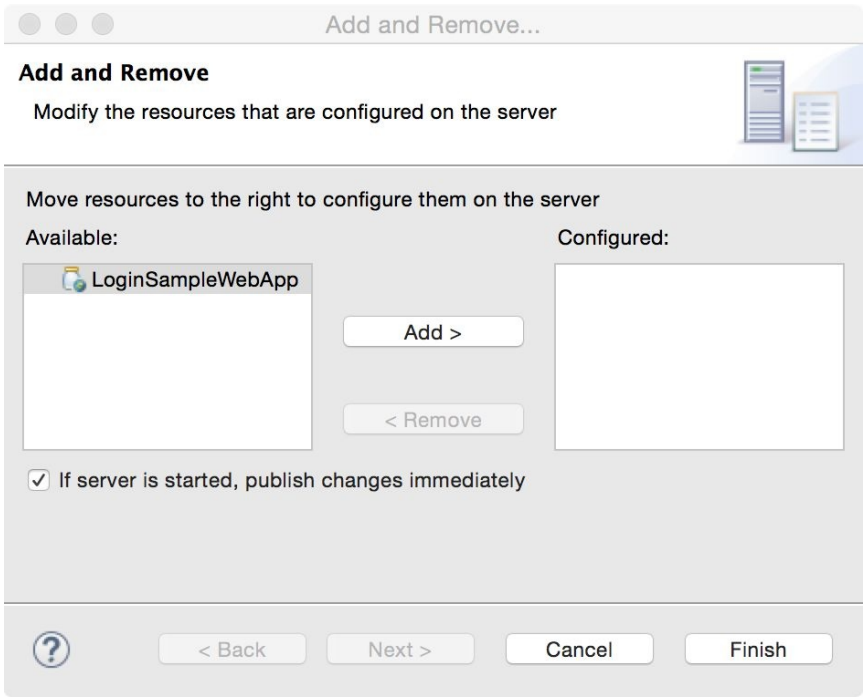

Figure 2.16: Add a project to the server

2. The other method to add a project to the server is to right-click on the project in Project Explorer and select Properties. This opens the Project Properties dialog box. Click on Server in the list and select the server in which you want to deploy this project. Click OK or Apply.

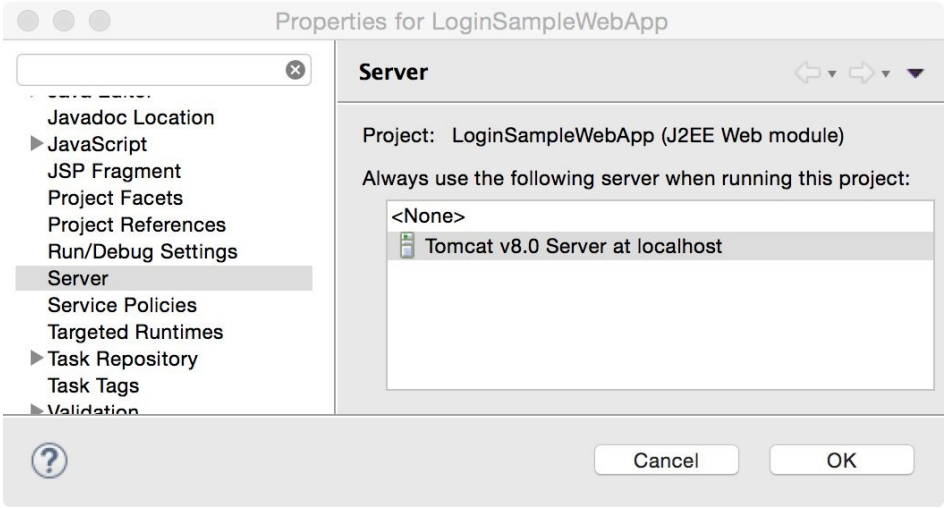

Figure 2.17: Select server in project properties

In the first method, the project is immediately deployed in the server. In the second method, it will be deployed only when you run the project in the server.

3. To run the application, right-click on the project in Project Explorer and select Run As | Run on Server. The first time you will be prompted to restart the server. Once the application is deployed, you will see it under the selected server in the Servers view:

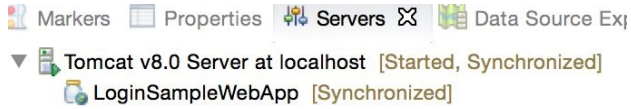

Figure 2.18: Project deployed on the server

4. Enter some text other than admin in the username and password boxes and click Submit. You should see the error message and the same form should be displayed again.

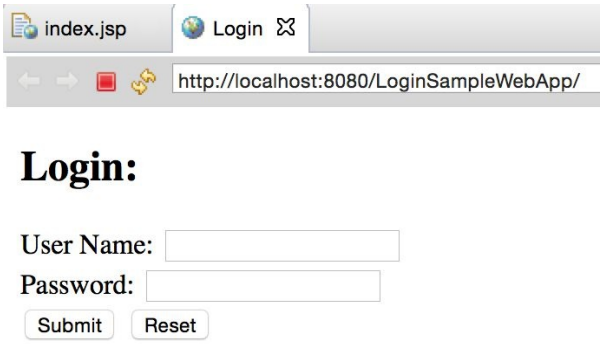

Figure 2.19: Project running in the built-in browser in Eclipse

5. Now enter `admin` as username and password and then submit the form. You should see the welcome message.

JSPs are compiled dynamically to Java classes, so if you make any changes in the page, in most cases, you do not have to restart the server; just refresh the page, and Tomcat will recompile the page if it has changed and the modified page will be displayed. In cases when you need to restart the server to apply your changes, Eclipse will prompt you if you want to restart the server.

Using JavaBeans in JSP

The JSP that we created previously does not follow JSP best practices. In general, it is a bad idea to have scriptlets (Java code) in JSP. In most large organizations, UI designer and programmer are different roles performed by different people. Therefore, it is recommended that JSP contains mostly markup tags so that it is easy for designers to work on the page design. Java code should be in separate classes. It also makes sense from a reusability point of view to move Java code out of JSP.

You can delegate the processing of the business logic to JavaBeans from JSP. JavaBeans are simple Java objects with attributes and getters and setters methods. The naming convention for getter/setter methods in JavaBeans is the prefix `get`/`set` followed by the name of the attribute, with the first letter of each word in uppercase, also known as CamelCase. For example, if you have a class attribute named `firstName`, then the getter method will be `getFirstName` and the setter will be `setFirstName`.

JSP has a special tag for using JavaBeans—`jsp:useBean`:

```
<jsp:useBean id="name_of_variable" class="name_of_bean_class"
 scope="scope_of_bean"/>
```

Scope indicates the lifetime of the bean. Valid values are `application`, `page`, `request`, and `session`.

Scope name	**Description**
`page`	Bean can be used only in the current page.
`request`	Bean can be used in any page in the processing of the same request. One web request can be handled by multiple JSPs if one

	page forwards the request to another page.
`session`	Bean can be used in the same HTTP session. The session is useful if your application wants to save the user data per interaction with the application, for example, to save items in the shopping cart in an online store application.
`application`	Bean can be used in any page in the same web application. Typically, web applications are deployed in a web application container as **web application archive** (**WAR**) files. In the application scope, all JSPs in the WAR file can use JavaBeans.

We will move the code to validate users in our login example to the `JavaBean` class. First, we need to create a `JavaBean` class:

1. In Project Explorer, right-click on the `src` folder New | Package menu option.
2. Create a package named `packt.book.jee_eclipse.ch2.bean`.
3. Right-click on the package and select the New | Class menu option.
4. Create a class named `LoginBean`.

5. Create two private `String` members as follows:

```
public class LoginBean {
  private String userName;
  private String password;
}
```

6. Right-click anywhere inside the class (in the editor) and select the Source | Generate Getters and Setters menu option:

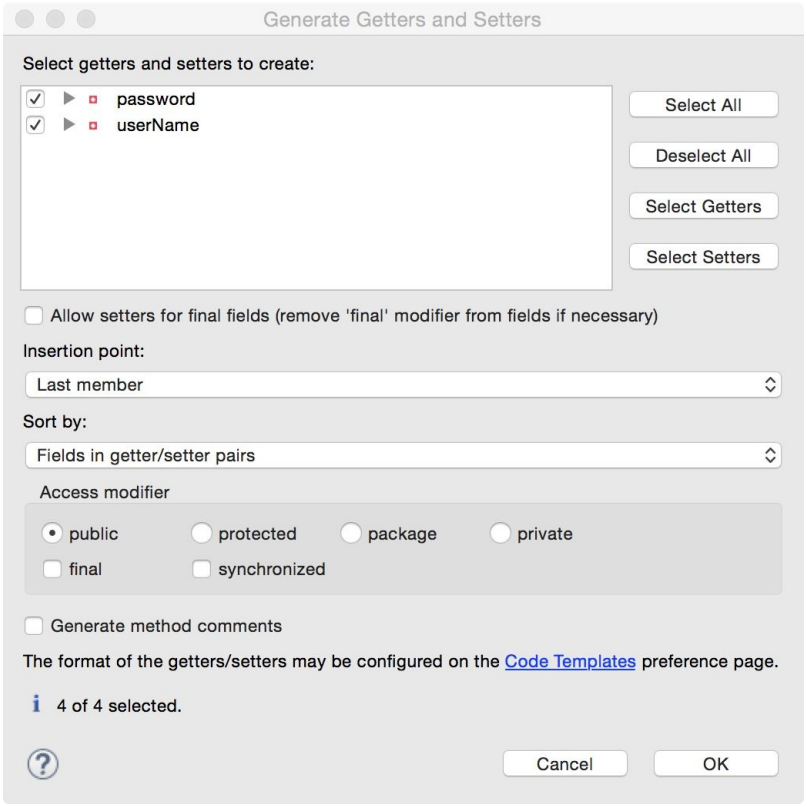

Figure 2.20: Generate getters and setters

7. We want to generate getters and setters for all members of the class. Therefore, click the Select All button and select Last member from the drop-down list for Insertion point, because we want to insert the getters and setters after declaring all member variables.

The `LoginBean` class should now be as follows:

```
public class LoginBean {
private String userName;
  private String password;
  public String getUserName() {
    return userName;
  }
  public void setUserName(String userName) {
    this.userName = userName;
  }
  public String getPassword() {
    return password;
  }
  public void setPassword(String password) {
    this.password = password;
  }
}
```

8. We will add one more method to it, to validate username and password:

```
        public boolean isValidUser()
          {
            //Validation can happen here from a number of sources
            //for example, database and LDAP
            //We are just going to hardcode a valid username and
            //password here.
            return "admin".equals(this.userName) &&
                    "admin".equals(this.password);
          }
```

This completes our JavaBean for storing user information and validation.
We will now use this bean in our JSP and delegate the task of validating users to this bean. Open `index.jsp`. Replace the Java scriptlet just above the `<body>` tag in the preceding code with the following:

```
<%String errMsg = null; %>
<%if ("POST".equalsIgnoreCase(request.getMethod()) && request.getParameter("submit") !=
null) {%>
  <jsp:useBean id="loginBean"
   class="packt.book.jee_eclipse.ch2.bean.LoginBean">
    <jsp:setProperty name="loginBean" property="*"/>
  </jsp:useBean>
  <%
    if (loginBean.isValidUser())
    {
      //valid user
      out.println("<h2>Welcome admin !</h2>");
      out.println("You are successfully logged in");
    }
    else
    {

      errMsg = "Invalid user id or password. Please try again";

    }
  %>
<%} %>
```

Before we discuss what has changed in the preceding code, note that you can invoke and get code assist for the attributes and values of `<jsp:*>` tags too. If you are not sure whether code assist is available, just press *Ctrl*/*Cmd* + *C*.

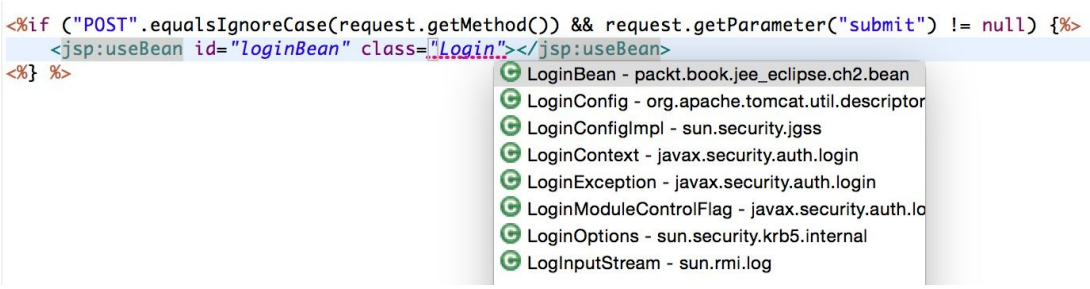

Figure 2.21: Code assist in JSP tags

Notice that Eclipse displays code assist for the JavaBean that we just added.

Let's now understand what we changed in the JSP:

- We created multiple scriptlets, one for declaration of the `errMsg` variable and two more for separate `if` blocks.
- We added a `<jsp:useBean` tag in the first `if` condition. The bean is created when a condition in the `if` statement is true, that is, when the form is posted by clicking the Submit button.

- We used the `<jsp:setProperty>` tag to set attributes of the bean:

```
<jsp:setProperty name="loginBean" property="*"/>
```

  We are setting values of member variables of `loginBean`. Furthermore, we are setting values of all the member variables by specifying `property="*"`. However, where do we specify values? The values are specified implicitly because we have named members of `LoginBean` to be the same as the fields in the form. So, the JSP runtime gets parameters from the `request` object and assigns values to the JavaBean members with the same name.
  If names of the members of JavaBean do not match the request parameters, then you need to set the values explicitly:

```
<jsp:setProperty name="loginBean" property="userName"
  value="<%=request.getParameter("userName")%>"/>
<jsp:setProperty name="loginBean" property="password"
  value="<%=request.getParameter("password")%>"/>
```

- We then checked whether the user is valid by calling `loginBean.isValidUser()`. The code to handle error messages hasn't changed.

To test the page, perform the following steps:

1. Right-click on `index.jsp` in Project Explorer.
2. Select the Run As | Run on Server menu option. Eclipse will prompt you to restart the Tomcat server.
3. Click the OK button to restart the server.

The page will be displayed in the internal Eclipse browser. It should behave in the same way as in the previous example.

Although we have moved validation of users to `LoginBean`, we still have a lot of code in Java scriptlets. Ideally, we should have as few Java scriptlets as possible in JSP. We still have scriptlets for checking conditions and for variable assignments. We can write the same code by using tags so that it is consistent with the remaining tag-based code in JSP and will be easier for web designers to work with it. This can be achieved using **JSP Standard Tag Library** (**JSTL**).

Using JSTL

JSTL tags can be used to replace much of the Java scriptlets in JSP. JSTL tags are classified in five broad groups:

- **Core**: Covers flow control and variable support among other things
- **XML**: Tags to process XML documents
- **i18n**: Tags to support internationalization
- **SQL**: Tags to access database
- **Functions**: Tags to perform some of the common string operations

 *See* `http://docs.oracle.com/javaee/5/tutorial/doc/bnake.html` *for more details on JSTL.*

We will modify the login JSP to use JSTL, so that there are no Java scriptlets in it:

1. Download JSTL libraries for APIs and their implementation. At the time of writing, the latest `.jar` files are `javax.servlet.jsp.jstl-api-1.2.1.jar` (`http://search.maven.org/remotecontent?filepath=javax/servlet/jsp/jstl/javax.servlet.jsp.jstl-api/1.2.1/javax.servlet.jsp.jstl-api-1.2.1.jar`) and `javax.servlet.jsp.jstl-1.2.1.jar` (`http://search.maven.org/remotecontent?filepath=org/glassfish/web/javax.servlet.jsp.jstl/1.2.1/javax.servlet.jsp.jstl-1.2.1.jar`). Make sure that these files are copied to `WEB-INF/lib`. All `.jar` files in this folder are added to the `classpath` of the web application.

2. We need to add a declaration for JSTL in our JSP. Add the following `taglib` declaration below the first page declaration (`<%@ page language="java" ...>`):

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

   The `taglib` declaration contains the URL of the `tag` library and `prefix`. All tags in the `tag` library are accessed using `prefix` in JSP.

3. Replace `<%String errMsg = null; %>` with the `set` tag of JSTL:

```
<c:set var="errMsg" value="${null}"/> <c:set var="displayForm"
value="${true}"/>
```

We have enclosed the value in `${}`. This is called **Expression Language** (**EL**). You enclose the Java expression in JSTL in `${}`.

4. Replace the following code:

```
<%if ("POST".equalsIgnoreCase(request.getMethod()) &&
request.getParameter("submit") != null) {%>
```

With the `if` tag of JSTL: <c:if test="${"POST".equalsIgnoreCase(pageContext.request .method) && pageContext.request.getParameter("submit") != null}">

The `request` object is accessed in the JSTL tag via `pageContext`.

5. JavaBean tags go within the `if` tag. There is no change in this code:

```
<jsp:useBean id="loginBean"


  class="packt.book.jee_eclipse.ch2.bean.LoginBean"> <jsp:setProperty
name="loginBean" property="*"/> </jsp:useBean>
```

6. We then add tags to call `loginBean.isValidUser()` and based on its return value, to set messages. However, we can't use the `if` tag of JSTL here, because we need to write the `else` statement too. JSTL does not have a tag for `else`. Instead, for multiple `if...else` statements, you need to use the `choose` statement, which is somewhat similar to the `switch` statement:

```
<c:choose>


  <c:when test="${!loginBean.isValidUser()}"> <c:set var="errMsg"
value="Invalid user id or password. Please
     try again"/> </c:when>


  <c:otherwise>


    <h2><c:out value="Welcome admin !"/></h2> <c:out value="You are
successfully logged in"/> <c:set var="displayForm" value="${false}"/>
</c:otherwise>
```

```
    </c:choose>
```

If the user credentials are not valid, we set the error message. Or (in the `c:otherwise` tag), we print the welcome message and set the `displayForm` flag to `false`. We don't want to display the login form if the user is successfully logged in.

7. We will now replace another `if` scriptlet code by `<%if%>` tag. Replace the following code snippet:

```
<%if (errMsg != null) { %> <span style="color: red;"><%out.print(errMsg); %>
</span> <%} %>
```

With the following code:

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


    <c:out value="${errMsg}"></c:out>


  </span>


</c:if>
```

Note that we have used the `out` tag to print an error message.

8. Finally, we enclose the entire `<body>` content in another JSTL `if` tag:

```
<c:if test="${displayForm}"> <body>


   ...


</body>


</c:if>
```

Here is the complete source code of the 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>Login</title>

</head>



<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="loginBean"
class="packt.book.jee_eclipse.ch2.bean.LoginBean"> <jsp:setProperty
name="loginBean" property="*"/> </jsp:useBean>

<c:choose>

<c:when test="${!loginBean.isValidUser()}"> <c:set var="errMsg"
value="Invalid user id or password.
Please try again"/> </c:when>

<c:otherwise>

<h2><c:out value="Welcome admin !"/></h2> <c:out value="You are
successfully logged in"/> <c:set var="displayForm" value="${false}"/>
</c:otherwise>

</c:choose>

</c:if>
```

<c:if test="${displayForm}">

<body>

<h2>Login:</h2>

<!-- Check error message. If it is set, then display it --> <c:if test="${errMsg != null}">

<c:out value="${errMsg}"></c:out>

</c:if>

<form method="post">

User Name: <input type="text" name="userName">
 Password: <input type="password" name="password">
 <button type="submit" name="submit">Submit</button> <button type="reset">Reset</button> </form>

</body>

</c:if>

</html>

As you can see, there are no Java scriptlets in the preceding code. All of them, from the previous code, are replaced by tags. This makes it easy for web designers to edit the page without worrying about Java scriptlets.

One last note before we leave the topic of JSP. In real-world applications, you would probably forward the request to another page after the user successfully logs in, instead of just displaying a welcome message on the same page. You could use the `<jsp:forward>` tag to achieve this.

Java Servlet

We will now see how to implement a login application using Java Servlet. Create a new **Dynamic Web Application** in Eclipse as described in the previous section. We will call this `LoginServletApp`:

1. Right-click on the `src` folder under `Java Resources` for the project in Project Explorer. Select the New | Servlet menu option.
2. In the Create Servlet wizard, enter package name as `packt.book.jee_eclipse.book.servlet` and class name as `LoginServlet`. Then, click Finish.

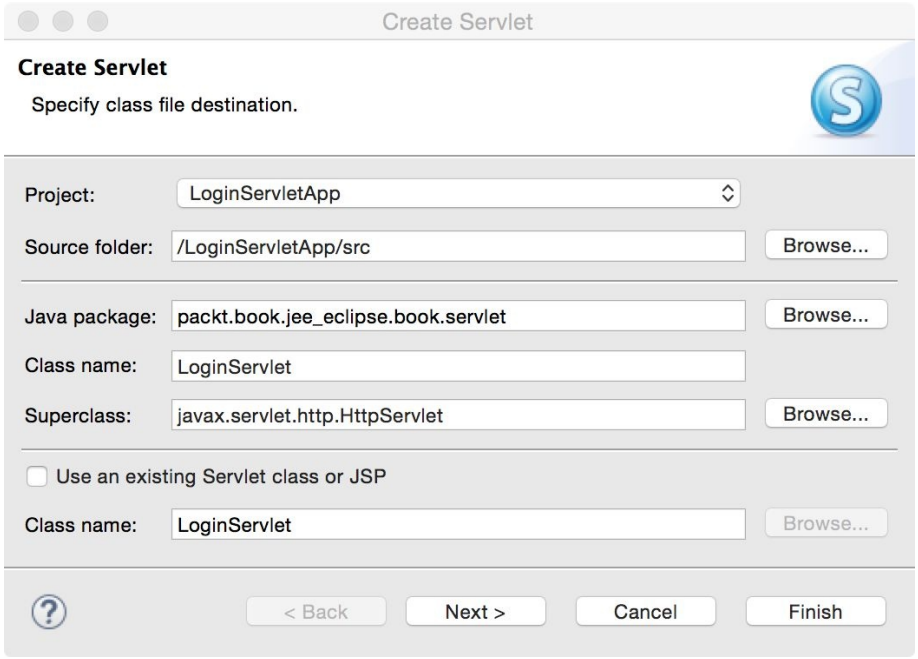

Figure 2.22: Create Servlet wizard

3. The servlet wizard creates the class for you. Notice the `@WebServlet("/LoginServlet")` annotation just above the class declaration. Before JEE 5, you had to declare servlets in `web.xml` in the `WEB-INF` folder. You can still do that, but you can skip this declaration if you use proper annotations. Using `WebServlet`, we are telling the servlet container that `LoginServlet` is a

servlet, and we are mapping it to the `/LoginServlet` URL path. Thus, we are avoiding the following two entries in `web.xml` by using this annotation: `<servlet>` and `<servlet-mapping>`.

We will now change the mapping from `/LoginServlet` to just `/login`. Therefore, we will modify the annotation as follows: @WebServlet("/login")

public class LoginServlet extends HttpServlet {...}

4. The wizard also created the `doGet` and `doPost` methods. These methods are overridden from the following base class: `HttpServlet`. The `doGet` method is called to create response for the `Get` request and `doPost` is called to create a response for the `Post` request.

   We will create a login form in the `doGet` method and process the form data (`Post`) in the `doPost` method. However, because `doPost` may need to display the form, in case user credentials are invalid, we will write a `createForm` method, which could be called from both `doGet` and `doPost`.

5. Add a `createForm` method as follows:

```
protected String createForm(String errMsg) {


 StringBuilder sb = new StringBuilder("<h2>Login</h2>"); //check whether error
message is to be displayed if (errMsg != null) {


    sb.append("<span style='color: red;'>") .append(errMsg)


      .append("</span>");


  }


  //create form


  sb.append("<form method='post'>n") .append("User Name: <input type='text'
     name='userName'><br>n")    .append("Password: <input type='password'
     name='password'><br>n")    .append("<button type='submit'
     name='submit'>Submit</button>n") .append("<button
type='reset'>Reset</button>n") .append("</form>");
```

```
  return sb.toString();


}
```

6. We will now modify a `doGet` method to call a `createForm` method and return the response:

```
protected void doGet(HttpServletRequest request,
 HttpServletResponse response) throws ServletException, IOException {


  response.getWriter().write(createForm(null)); }
```

We call the `getWrite` method on the `response` object and write the form content to it by calling the `createForm` function. Note that when we display the form, initially, there is no error message, so we pass a `null` argument to `createForm`.

7. We will modify `doPost` to process the form content when the user posts the form by clicking the Submit button:

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)


  throws ServletException, IOException {


    String userName = request.getParameter("userName"); String password =
request.getParameter("password");


  //create StringBuilder to hold response string StringBuilder responseStr =
new StringBuilder(); if ("admin".equals(userName) && "admin".equals(password))
{


    responseStr.append("<h2>Welcome admin !</h2>") .append("You are
successfully logged in"); }


  else {


    //invalid user credentials
```

```
            responseStr.append(createForm("Invalid user id or password.
             Please try again")); }




        response.getWriter().write(responseStr.toString()); }
```

We first get username and password from the `request` object by calling the `request.getParameter` method. If the credentials are valid, we add a welcome message to the `response` string; or else, we call `createForm` with an error message and add a return value (markup for the form) to the `response` string.

Finally, we get the `Writer` object from the `response` string and write the response.

8. Right-click on the `LoginServlet.java` file in Project Explorer and select the Run As | Run on Server option. We have not added this project to the Tomcat server. Therefore, Eclipse will ask if you want to use the configured server to run this servlet. Click the Finish button of the wizard.
9. Tomcat needs to restart because a new web application is deployed in the server. Eclipse will prompt you to restart the server. Click OK.

When the servlet is run in the internal browser of Eclipse, notice the URL; it ends with `/login`, which is the mapping that we specified in the servlet annotation. However, you will observe that instead of rendering the HTML form, the page displays the markup text. This is because we missed an important setting on the `response` object. We did not tell the browser the type of content we are returning, so the browser assumed it to be text and rendered it as plain text. We need to tell the browser that it is HTML content. We do this by calling `response.setContentType("text/html")` in both the `doGet` and the `doPost` methods. Here is the complete source code: package packt.book.jee_eclipse.book.servlet;
// skipping imports to save space

/**

* Servlet implementation class LoginServlet */

```
@WebServlet("/login")

public class LoginServlet extends HttpServlet {

private static final long serialVersionUID = 1L;

public LoginServlet() {

super();

}

//Handles HTTP Get requests

protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {

response.setContentType("text/html");

response.getWriter().write(createForm(null)); }

//Handles HTTP POST requests

protected void doPost(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {

String userName = request.getParameter("userName"); String password =
request.getParameter("password");

//create StringBuilder to hold response string StringBuilder responseStr = new
StringBuilder(); if ("admin".equals(userName) && "admin".equals(password)) {

responseStr.append("<h2>Welcome admin !</h2>") .append("You're are
successfully logged in"); } else {

//invalid user credentials

responseStr.append(createForm("Invalid user id or password.
Please try again")); }

response.setContentType("text/html");
```

```
response.getWriter().write(responseStr.toString()); }


//Creates HTML Login form

protected String createForm(String errMsg) {

StringBuilder sb = new StringBuilder("<h2>Login</h2>"); //check if error
message to be displayed if (errMsg != null) {

sb.append("<span style='color: red;'>") .append(errMsg)

.append("</span>");

}

//create form

sb.append("<form method='post'>n") .append("User Name: <input type='text'
name='userName'><br>n") .append("Password: <input type='password'
name='password'><br>n") .append("<button type='submit'
name='submit'>Submit</button>n") .append("<button
type='reset'>Reset</button>n") .append("</form>");

return sb.toString();

}

}
```

As you can see, it is not very convenient to write HTML markup in servlet. Therefore, if you are creating a page with a lot of HTML markup, then it is better to use JSP or plain HTML. Servlets are good to process requests that do not need to generate too much markup, for example, controllers in **Model-View-Controller** (**MVC**) frameworks, for processing requests that generate a non-text response, or for creating a web service or WebSocket endpoints.

Creating WAR

Thus far, we have been running our web application from Eclipse, which does all the work of deploying the application to the Tomcat server. This works fine during development, but when you want to deploy it to test or production servers, you need to create a **web application archive** (**WAR**). We will see how to create a WAR from Eclipse. However, first we will un-deploy the existing applications from Tomcat.

1. Go to the Servers view, select the application, and right-click and select the Remove option:

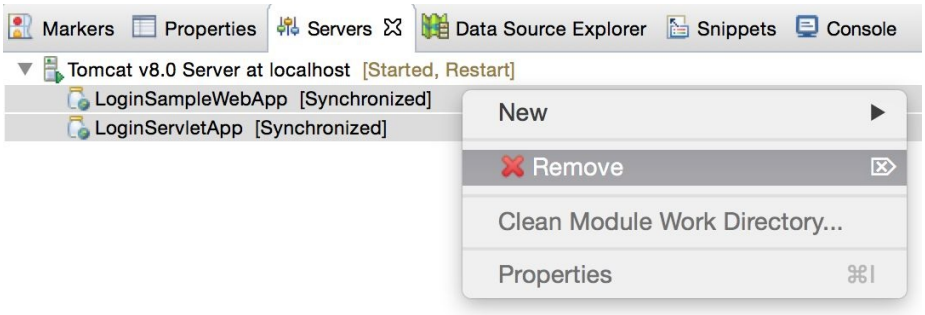

Figure 2.23 Un-deploy a web application from the server

2. Then, right-click on the project in Project Explorer and select Export | WAR file. Select the destination for the WAR file:

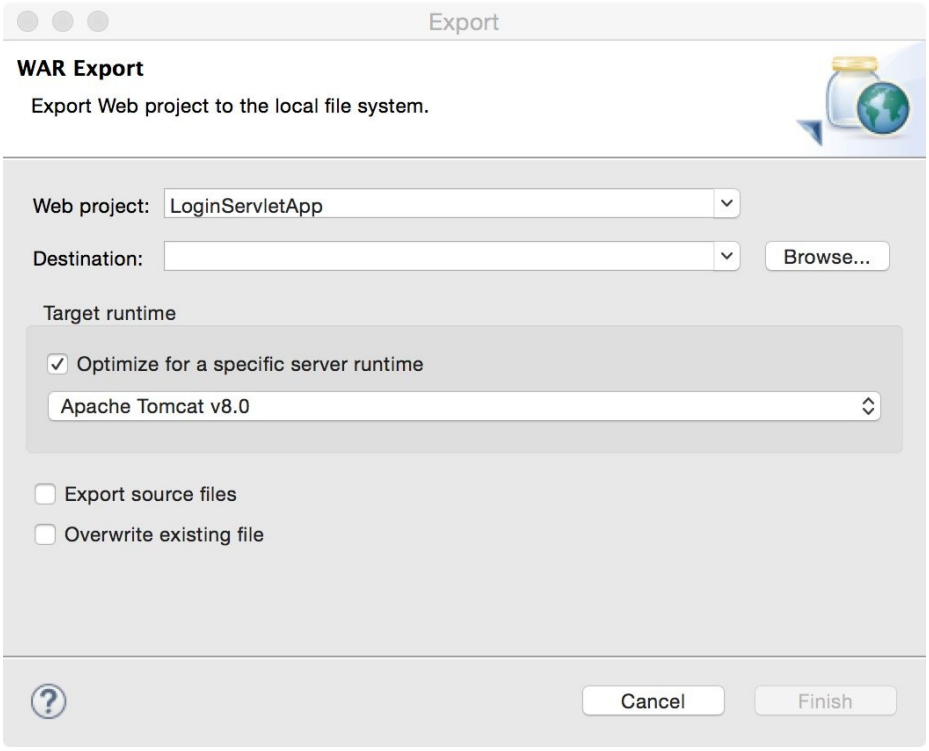

Figure 2.24 Export WAR

To deploy the WAR file to Tomcat, copy it to the `<tomcat_home>/webapps` folder. Then start the server if it is not already running. If Tomcat is already running, you don't need to restart it.

Tomcat monitors the `webapps` folder and any WAR file copied to it is automatically deployed. You can verify this by opening the URL of your application in the browser, for example, `http://localhost:8080/LoginServletApp/login`.

JavaServer Faces

When working with JSP, we saw that it is not a good idea to mix scriptlets with the HTML markup. We solved this problem by using JavaBean. JavaServer Faces takes this design further. In addition to supporting JavaBeans, JSF provides built-in tags for HTML user controls, which are context aware, can perform validation, and can preserve the state between requests. We will now create the login application using JSF:

1. Create a dynamic web application in Eclipse; let's name it `LoginJSFApp`. In the last page of the wizard, make sure that you check the Generate web.xml deployment descriptor box.
2. Download JSF libraries from `https://maven.java.net/content/repositories/releases/org/glassfish/javax.faces/2.2.9/javax.faces-2.2.9.jar` and copy them to the `WEB-INF/lib` folder in your project.
3. JSF follows the MVC pattern. In the MVC pattern, the code to generate user interface (view) is separate from the container of the data (model). The controller acts as the interface between the view and the model. It selects the model for processing a request on the basis of the configuration, and once the model processes the request, it selects the view to be generated and returned to the client, on the basis of the result of the processing in the model. The advantage of MVC is that there is a clear separation of the UI and the business logic (which requires a different set of expertise) so that they can be developed independently, to a large extent. In JSP the implementation of MVC is optional, but JSF enforces the MVC design.

Views are JSF created as `xhtml` files. The controller is a servlet from the JSF library and models are **managed beans** (JavaBeans).

We will first configure a controller for JSF. We will add the servlet configuration and mapping in `web.xml`. Open `web.xml` from the `WEB-INF` folder of the project (`web.xml` should have been created for you by the project wizard if you checked the Generate web.xml deployment descriptor box; step 1). Add the following XML snippet before `</web-app>`: <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>

Note that you can get code assist when creating the preceding elements by pressing *Ctrl*/*Cmd* + *C*.

You can specify any name as `servlet-name`; just make sure that you use the same name in `servlet-mapping`. The class for the servlet is `javax.faces.webapp.FacesServlet`, which is in the JAR file that we downloaded as the JSF library and copied to `WEB-INF/lib`. Furthermore, we have mapped any request ending with `.xhtml` to this servlet.

Next, we will create a managed bean for our login page. This is the same as JavaBean that we had created earlier, but with the addition of JSF-specific annotations:

1. Right-click on the `src` folder under `Java Resources` for the project in Project Explorer.
2. Select the New | Class menu option.
3. Create JavaBean, `LoginBean`, as described in the *Using JavaBeans in JSP* section of this chapter.
4. Create two members for `userName` and `password`.
5. Create the getters and setters for them. Then, add two annotations as follows:

```
package packt.book.jee_eclipse.bean; import javax.faces.bean.ManagedBean;
import javax.faces.bean.RequestScoped;


@ManagedBean(name="loginBean")


@RequestScoped
```

```
public class LoginBean {

  private String userName;

  private String password;

  public String getUserName() {

    return userName;

  }

  public void setUserName(String userName) {

    this.userName = userName;

  }

  public String getPassword() {

    return password;

  }

  public void setPassword(String password) {

    this.password = password;

  }

}
```

(You can get code assist for annotations too. Type `@` and press *Ctrl*/*Cmd* + *C*. Code assist works for the annotation `key-value` attribute pairs too, for example, for the `name` attribute of the `ManagedBean` annotation).

6. Create a new file called `index.xhtml` inside the `WebContent` folder of the project by selecting the File | New | File menu option. When using JSF, you need to add a few namespace declarations at the top of the file:

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

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

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

   Here, we are declaring namespaces for JSF built-in `tag` libraries. We will access tags in the core JSF `tag` library with the prefix `f` and HTML tags with the prefix `h`.

7. Add the title and start the `body` tag:

```
<head> <title>Login</title> </head>

<body>

  <h2>Login</h2>
```

   There are corresponding JSF tags for the `head` and the `body`, but we do not use any attributes specific to JSF; therefore, we have used simple HTML tags.

8. We then add the code to display the error message, if it is not null:

```
<h:outputText value="#{loginBean.errorMsg}"

              rendered="#{loginBean.errorMsg != null}"

              style="color:red;"/>
```

   Here, we use a tag specific to JSF and expression language to display the value of the error message. The `OutputText` tag is similar to the `c:out` tag that we saw in JSTL. We have also added a condition to render it only if the error message in the managed bean is not `null`. Additionally, we have set the color of this output text.

9. We have not added the `errorMsg` member to the managed bean yet. Therefore, let's add the declaration, the getter, and the setter. Open the `LoginBean` class and add the following code:

```
private String errorMsg; public String getErrorMsg() {


  return errorMsg;


}


public void setErrorMsg(String errorMsg) {


  this.errorMsg = errorMsg;


}
```

   Note that we access the managed bean in JSF by using value of the `name` attribute of the `ManagedBean` annotation. Furthermore, unlike JavaBean in JSP, we do not create it by using the `<jsp:useBean>` tag. The JSF runtime creates the bean if it is not already there in the required scope, in this case, the `Request` scope.

10. Let's go back to editing `index.xhtml`. We will now add the following form:

```
<h:form> User Name: <h:inputText id="userName"


                value="#{loginBean.userName}"/><br/> Password: <h:inputSecret
id="password"


                value="#{loginBean.password}"/><br/> <h:commandButton
value="Submit"
   action="#{loginBean.validate}"/> </h:form>
```

   Many things are happening here. First, we have used the `inputText` tag of JSF to create textboxes for username and password. We have set their values with the corresponding members of `loginBean`. We have used the `commandButton` tag of JSF to create a Submit button. When the user clicks the Submit button, we have set it to call the `loginBean.validate` method (using the `action` attribute).

11. We haven't defined a `validate` method in `loginBean`, so let's add that. Open the `LoginBean` class and add the following code:

```
public String validate() {

  if ("admin".equals(userName) && "admin".equals(password)) {

    errorMsg = null;

    return "welcome";

  } else {

    errorMsg = "Invalid user id or password. Please try
     again"; return null;

  }

}
```

    Note that the `validate` method returns a string. How is the return value used? It is used for navigation purposes in JSF. The JSF runtime looks for the JSF file with the same name as the string value returned after evaluating the expression in the `action` attribute of `commandButton`. In the `validate` method, we return `welcome` if the user credentials are valid. In this case we are telling the JSF runtime to navigate to `welcome.xhtml`. If the credentials are invalid, we set the error message and return `null`, in which case, the JSF runtime displays the same page.

12. We will now add the `welcome.xhml` page. It simply contains the welcome message:

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

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

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

```
            <h2>Welcome admin !</h2> You are successfully logged in </body>


        </html>
```

Here is the complete source code of `index.html`: <html xmlns="http://www.w3.org/1999/xhtml"

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

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

<head>

<title>Login</title> </head>

<body>

<h2>Login</h2>

<h:outputText value="#{loginBean.errorMsg}"

rendered="#{loginBean.errorMsg != null}"

style="color:red;"/>

<h:form>

User Name: <h:inputText id="userName" value="#{loginBean.userName}"/>
 Password: <h:inputSecret id="password" value="#{loginBean.password}"/>
 <h:commandButton value="Submit" action="#{loginBean.validate}"/> </h:form>

</body>

</html>

Here is the source code of the `LoginBean` class: package packt.book.jee_eclipse.bean; import javax.faces.bean.ManagedBean; import javax.faces.bean.RequestScoped;

```
@ManagedBean(name="loginBean")
@RequestScoped
public class LoginBean {
private String userName;
private String password;
private String errorMsg;
public String getUserName() {
return userName;
}
public void setUserName(String userName) {
this.userName = userName;
}
public String getPassword() {
return password;
}
public void setPassword(String password) {
this.password = password;
}
public String getErrorMsg() {
return errorMsg;
}
```

```
public void setErrorMsg(String errorMsg) {

this.errorMsg = errorMsg;

}

public String validate()

{

if ("admin".equals(userName) && "admin".equals(password)) {

errorMsg = null;

return "welcome";

}

else {

errorMsg = "Invalid user id or password. Please try again"; return null;

}

}

}
```

To run the application, right-click on `index.xhtml` in Project Explorer and select the Run As | Run on Server option.

JSF can do much more than what we have seen in this small example—it has the support to validate an input and create page templates too. However, these topics are beyond the scope of this book.

Visit `http://docs.oracle.com/cd/E11035_01/workshop102/webapplications/jsf/jsf-app-tutorial/Introduction.html` for a tutorial on JSF.

Using Maven for project management

In the projects that we have created thus far in this chapter, we have managed many project management tasks, such as downloading libraries on which our project depends, adding them to the appropriate folder so that the web application can find it, and exporting the project to create the WAR file for deployment. These are just some of the project management tasks that we have performed so far, but there are many more, which we will see in the subsequent chapters. It helps to have a tool do many of the project management tasks for us so that we can focus on application development. There are some well-known build management tools available for Java, for example, Apache Ant (`http://ant.apache.org/`) and Maven (`http://maven.apache.org/`).

In this section, we will see how to use Maven as a project management tool. By following the convention for creating the project structure and allowing projects to define the hierarchy, Maven makes project management easier than Ant. Ant is primarily a build tool, whereas Maven is a project management tool, which does build management too. See `http://maven.apache.org/what-is-maven.html` to understand what Maven can do.

In particular, Maven simplifies dependency management. In the JSF project earlier in this chapter, we first downloaded the appropriate `.jar` files for JSF and copied them to the `lib` folder. Maven can automate this. You can configure Maven settings in `pom.xml`. **POM** stands for **Project Object Model**.

Before we use Maven, it is important to understand how it works. Maven uses repositories. Repositories contain plugins for many well-known libraries/projects. A plugin includes the project configuration information, `.jar` files required to use this project in your own project, and any other supporting artifacts. The default Maven repository is a collection of plugins. You can find the list of plugins in the default Maven repository at `http://maven.apache.org/plugins/index.html`. You can also browse the content of the Maven repository at `http://search.maven.org/#browse`. Maven also maintains a local repository on your machine. This local repository contains only those plugins that your projects have specified dependencies on. On Windows, you will find the local repository at

`C:/Users /<username>.m2`, and on macOS X, it is located at `~/.m2`.

You define plugins on which your project depends in the `dependencies` section of `pom.xml` (we will see the structure of `pom.xml` shortly when we create a Maven project). For example, we can specify a dependency on JSF. When you run the Maven tool, it first inspects all dependencies in `pom.xml`. It then checks whether the dependent plugins with the required versions are already downloaded in the local repository. If not, it downloads them from the central (remote) repository. You can also specify repositories to look in. If you do not specify any repository, then dependencies are searched in the central Maven repository.

We will create a Maven project and explore `pom.xml` in more detail. However, if you are curious to know what `pom.xml` is, then visit `http://maven.apache.org/pom.html#What_is_the_POM`.

Eclipse JEE version has Maven built-in, so you don't need to download it. However, if you plan to use Maven from outside Eclipse, then download it from `http://maven.apache.org/download.cgi`.

Maven views and preferences in Eclipse JEE

Before we create a Maven project, let's explore the views and preferences specific to Maven in Eclipse:

1. Select the Window | Show View | Other... menu.
2. Type `Maven` in the filter box. You will see two views for Maven:

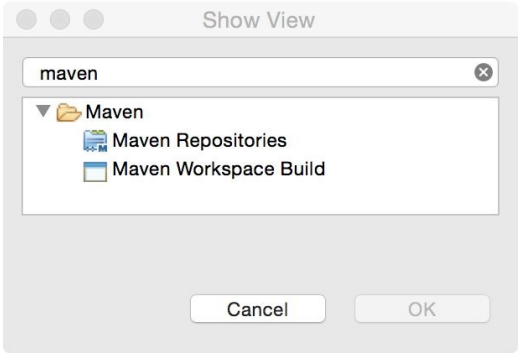

Figure 2.25: Maven views

3. Select Maven Repositories view and click OK. This view is opened in the bottom tab window of Eclipse. You can see the location of the local and remote repositories.
4. Right-click on a global repository to see the options to index the repository:

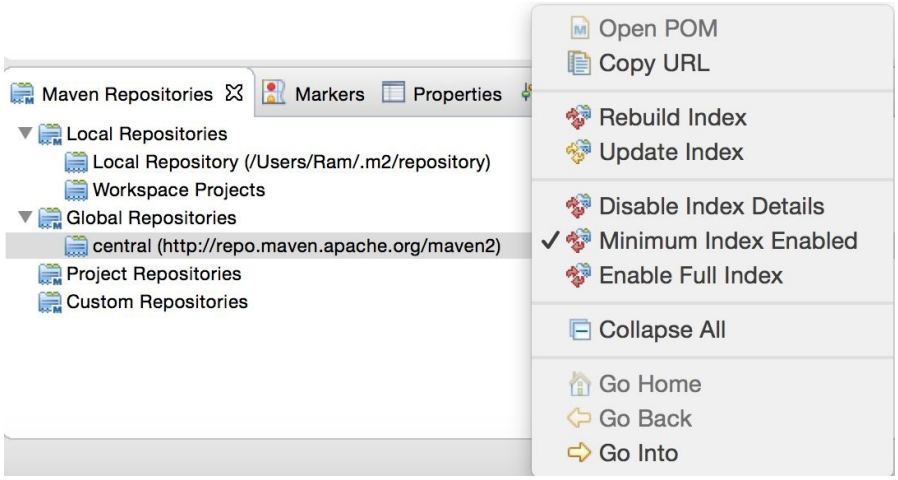

Figure 2.26: The Maven Repositories view

5. Open Eclipse Preferences and type `Maven` in the filter box to see all the Maven preferences:

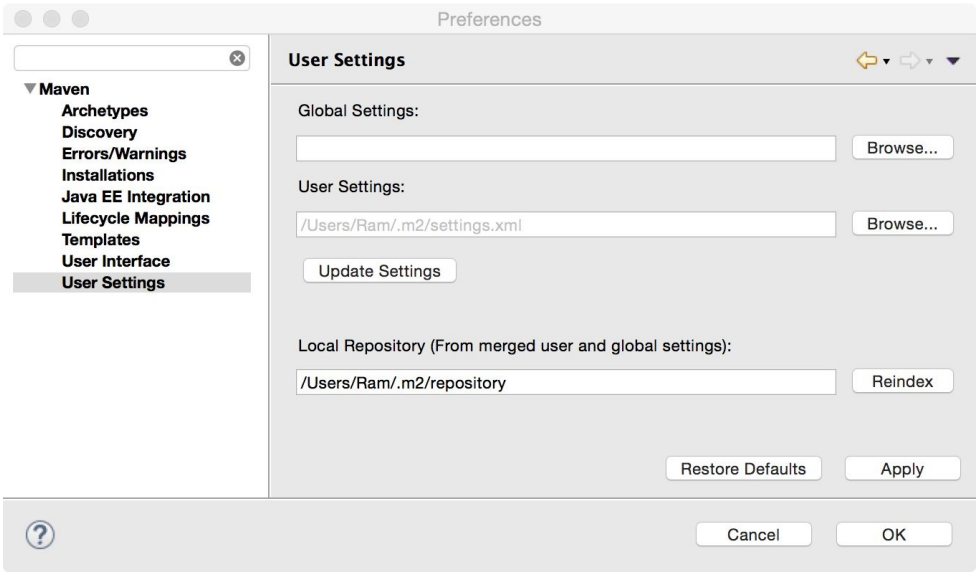

Figure 2.27: Maven preferences

You should set the Maven preferences to refresh repository indexes on startup, so that the latest libraries are available when you add dependencies to your project (we will learn how to add dependencies shortly).

6. Click on the Maven node in Preferences and set the following options:

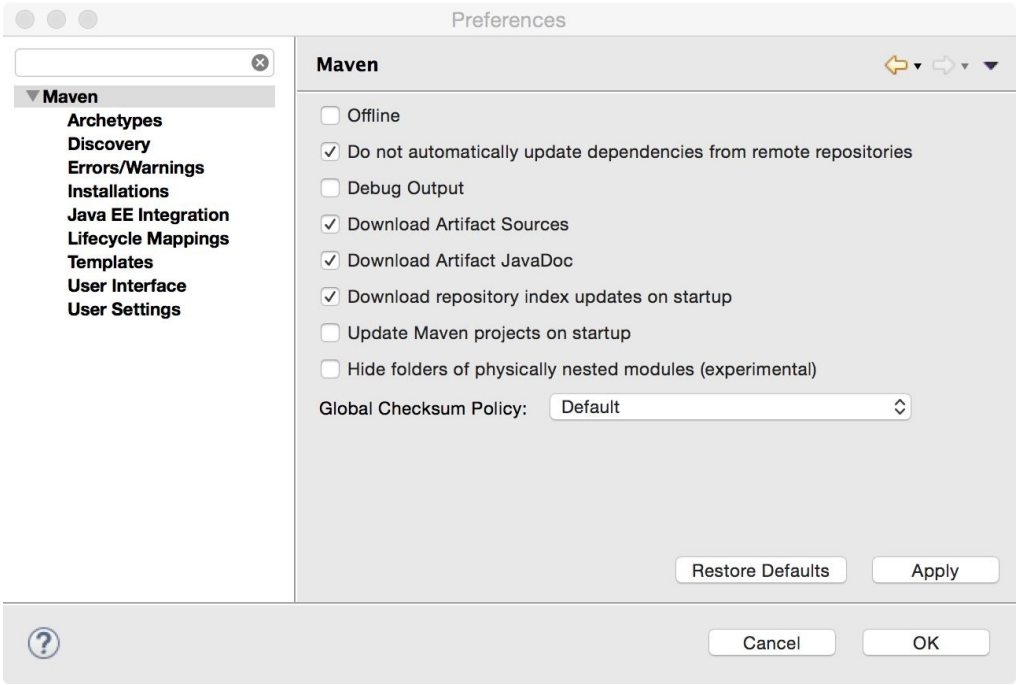

Figure 2.28: Maven preferences for updating indexes on startup

Creating a Maven project

In the following steps, we will see how to create a Maven project in Eclipse:

1. Select the New | Maven Project menu:

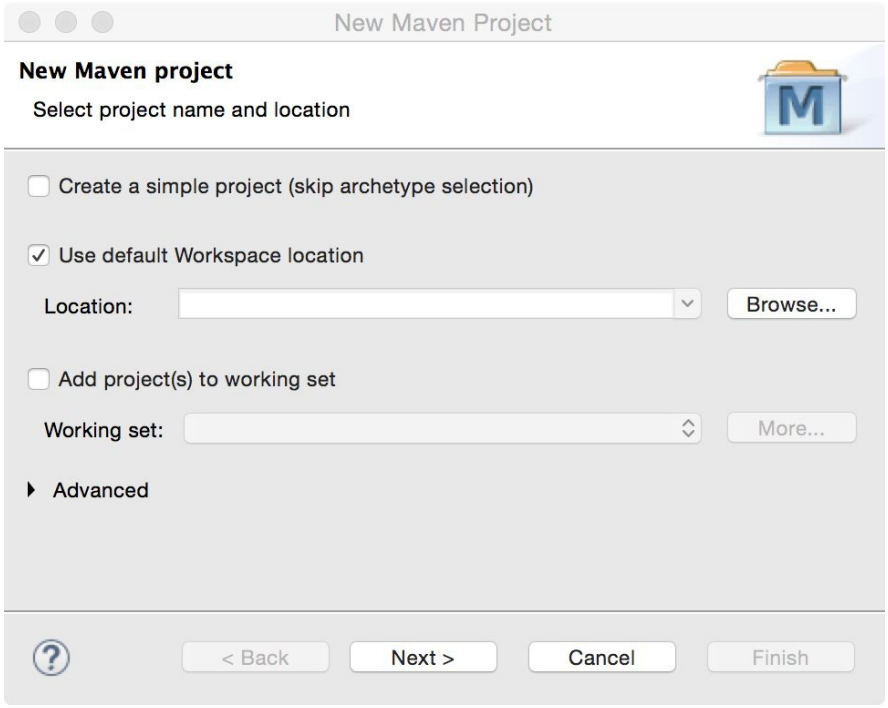

Figure 2.29: Maven New Project wizard

2. Accept all default options and click Next. Type `webapp` in the filter box and select maven-archetype-webapp:

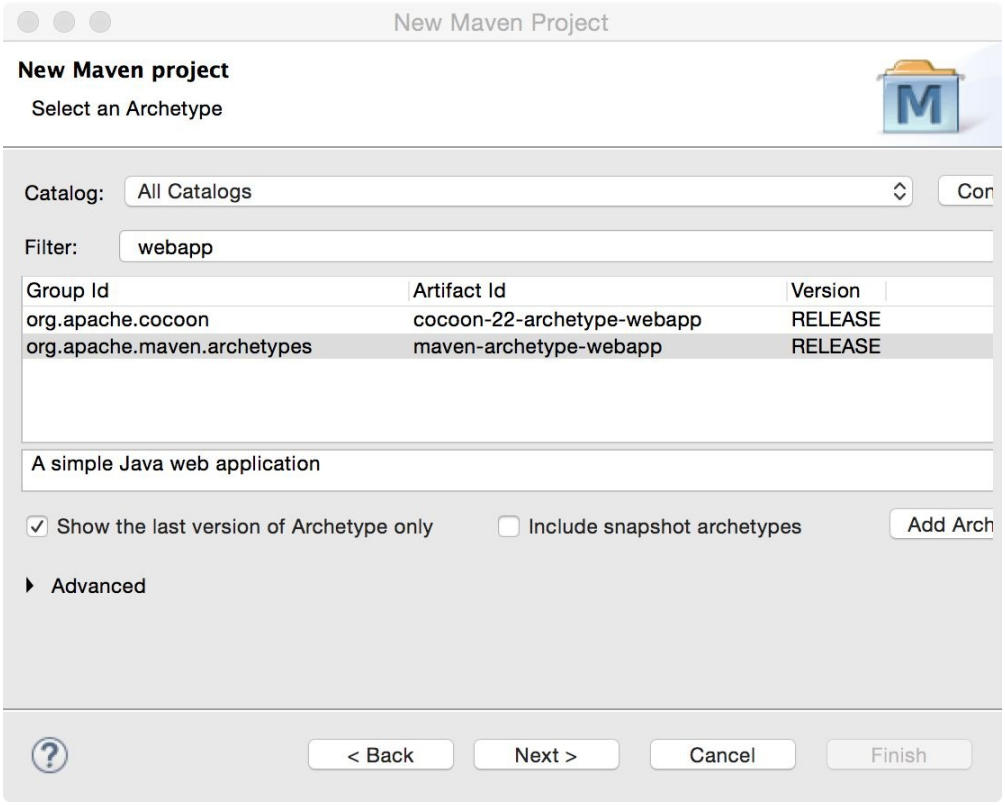

Figure 2.30: New Maven project - select archetype

Maven archetype

We selected maven-archetype-webapp in the preceding wizard. An archetype is a project template. When you use an archetype for your project, all the dependencies and other Maven project configurations defined in the template (archetype) are imported into your project.

See more information about Maven archetype at `http://maven.apache.org/guides/introduction/introduction-to-archetypes.html`.

1. Continuing with the New Maven Project wizard, click on Next. In the Group Id field, enter `packt.book.jee_eclipse`. In the Artifact Id field, enter `maven_jsf_web_app`:

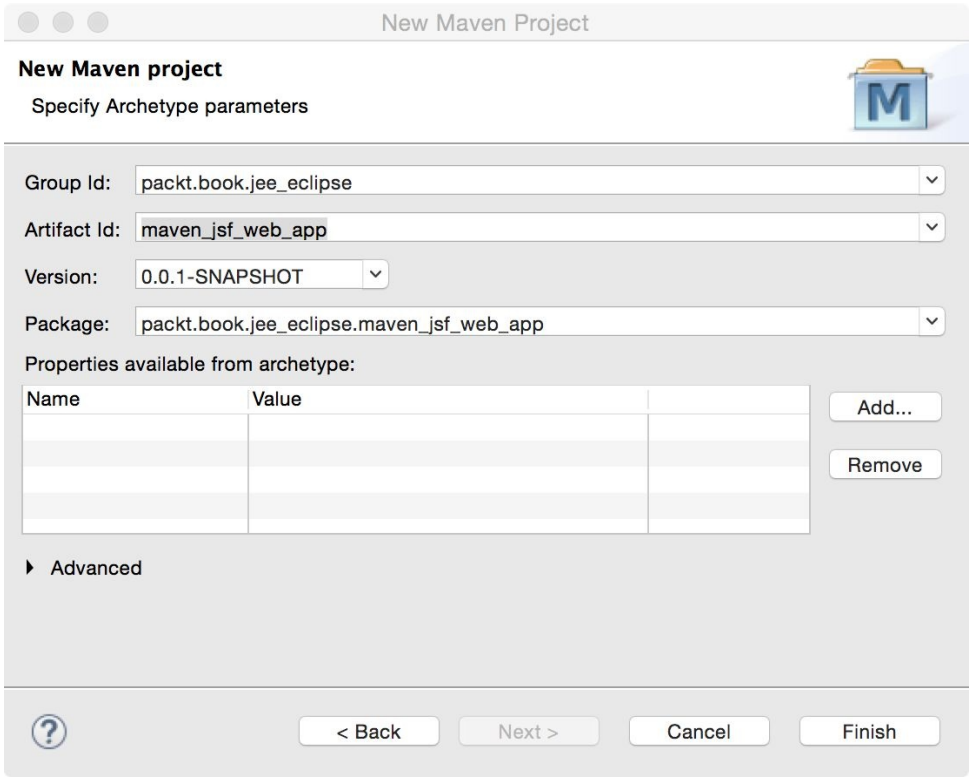

Figure 2.31: New Maven project - archetype parameters

2. Click on Finish. A `maven_jsf_web_app` project is added in Project Explorer.

Exploring the POM

Open `pom.xml` in the editor and go to the pom.xml tab. The file should have the following content: <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd"> <modelVersion>4.0.0</modelVersion> <groupId>packt.book.jee_eclipse</groupId> <artifactId>maven_jsf_web_app</artifactId> <packaging>war</packaging>

<version>0.0.1-SNAPSHOT</version> <name>maven_jsf_web_app Maven Webapp</name> <url>http://maven.apache.org</url> <dependencies>

<dependency>

<groupId>junit</groupId> <artifactId>junit</artifactId> <version>3.8.1</version> <scope>test</scope>

</dependency>

</dependencies>

<build>

<finalName>maven_jsf_web_app</finalName> </build>

</project>

Let's have a look at the different tags in detail, that are used in the preceding code snippet:

- `modelVersion`: This in the `pom.xml` file is the version of Maven.
- `groupId`: This is the common ID used in the business unit or organization under which projects are grouped together. Although it is not necessary to

use the package structure format for group ID, it is generally used.

- `artifactId`: This is the project name.
- `version`: This is version number of the project. Version numbers are important when specifying dependencies. You can have multiple versions of a project, and you can specify different version dependencies in different projects. Maven also appends the version number to JAR, WAR, or EAR files that it creates for the project.
- `packaging`: This tells Maven what kind of final output we want when the project is built. In this book, we will be using JAR, WAR, and EAR packaging types, although more types exist.
- `name`: This is actually the name of the project, but Eclipse shows `artifactid` as the project name in Project Explorer.
- `url`: This is the URL of your project if you are hosting the project information on the web. The default is Maven's URL.
- `dependencies`: This section is where we specify the libraries (or other Maven artifacts) that the project depends on. The archetype that we selected for this project has added the default dependency of JUnit to our project. We will learn more about JUnit in `Chapter 5`, *Unit Testing*.
- `finalName`: This tag in the `build` tag indicates the name of the output file (JAR, WAR, or EAR) that Maven generates for your project.

Adding Maven dependencies

The archetype that we selected for the project does not include some of the dependencies required for a JEE web project. Therefore, you might see error markers in `index.jsp`. We will fix this by adding dependencies for the JEE libraries:

1. With `pom.xml` open in the editor, click on the Dependencies tab.
2. Click the Add button. This opens the Select Dependency dialog.
3. In the filter box, type `javax.servlet` (we want to use servlet APIs in the project).

4. Select the latest version of the API and click the OK button.

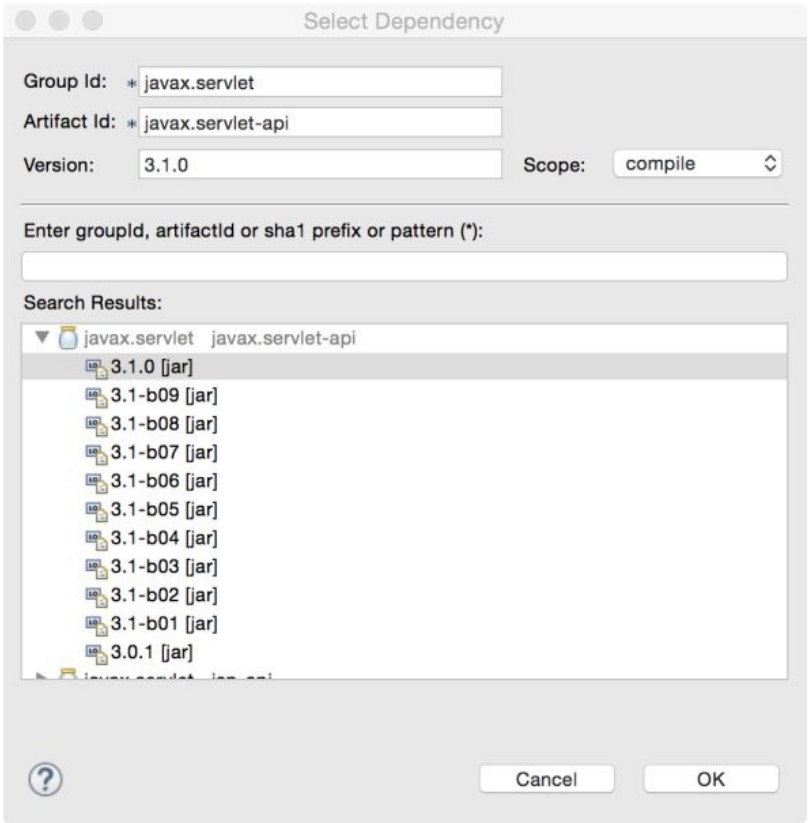

Figure 2.32: Adding servlet API dependency

However, we need JAR files for servlet APIs only at the compile time; at runtime, these APIs are provided by Tomcat. We can indicate this by spec

the scope of the dependency; in this case, setting it to provided, which tell Maven to evaluate this dependency for compilation only and not to packa the WAR file. See `http://maven.apache.org/guides/introduction/introduction-to-d y-mechanism.html` for more information on dependency scopes.

5. To set scope of the dependency, select dependency from the Dependencies tab of the POM editor.
6. Click the Properties button. Then, select the provided scope from the drop-down list:

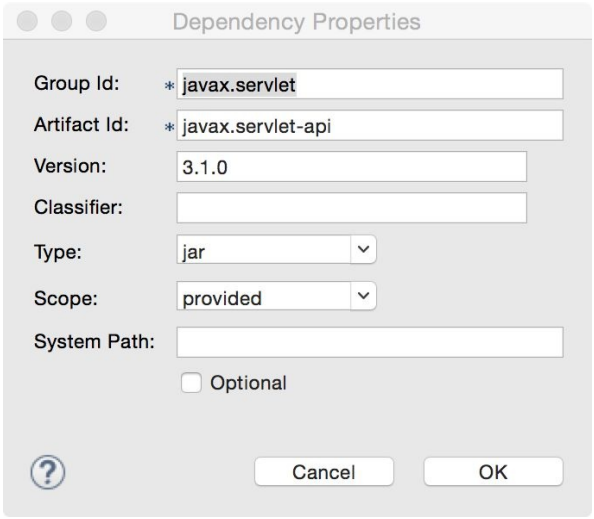

Figure 2.33: Setting the Maven dependency scope

7. Now we need to add dependencies for JSF APIs and their implementation. Click the Add button again and type `jsf` in the search box.

8. From the list, select `jsf-api` with Group Id `com.sun.faces` and click the OK button:

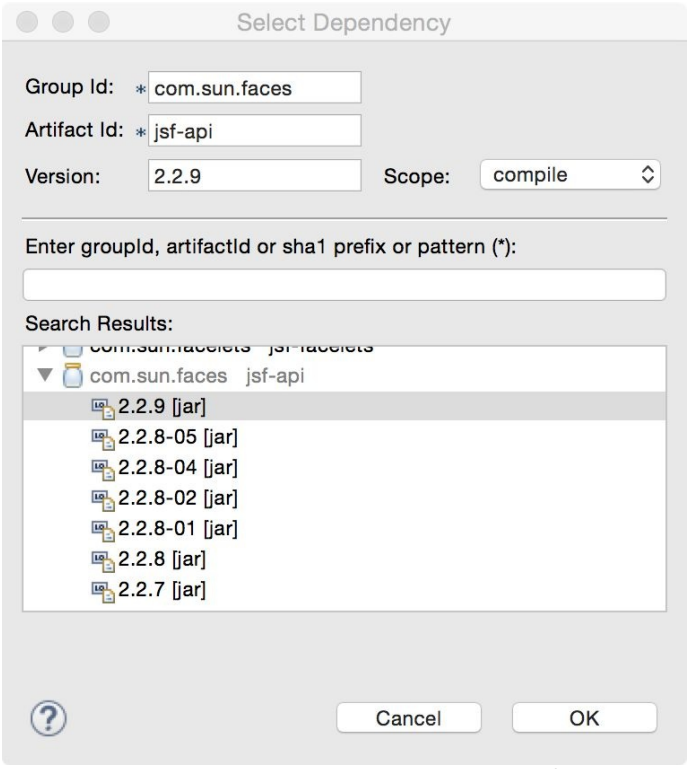

Figure 2.34: Adding Maven dependencies for JSF

9. Similarly, add a dependency for `jsf-impl` with Group Id `com.sun.faces`. The dependencies section in your `pom.xml` should look as follows:

```
<dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
    <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>
```

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.*

Maven project structure

The Maven project wizard creates `src` and `target` folders under the main project folder. As the name suggests, all source files go under `src`. However, Java package structure starts under the `main` folder. By convention, Maven expects Java source files under the `java` folder. Therefore, create a `java` folder under `src/main`. The Java package structure starts from the `java` folder, that is, `src/main/java/<java-packages>`. Web content such as HTML, JS, CSS, and JSP goes in the `webapp` folder under `src/main`. Compiled classes and other output files generated by the Maven build process are stored in the `target` folder:

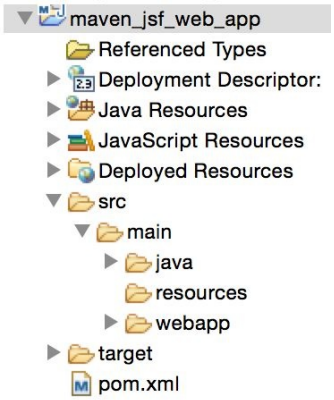

Figure 2.35: Maven web application project structure

The source code for our login JSF page is the same as in the previous example of `LoginJSFApp`. Therefore, copy the `packt` folder from the `src` folder of that project to the `src/main/java` folder of this Maven project. This adds `LoginBean.java` to the project. Then, copy `web.xml` from the `WEB-INF` folder to the `src/main/webapp/WEB-INF` folder of this project. Copy `index.xhtml` and `welcome.xhtml` to the `src/main/webapp` folder:

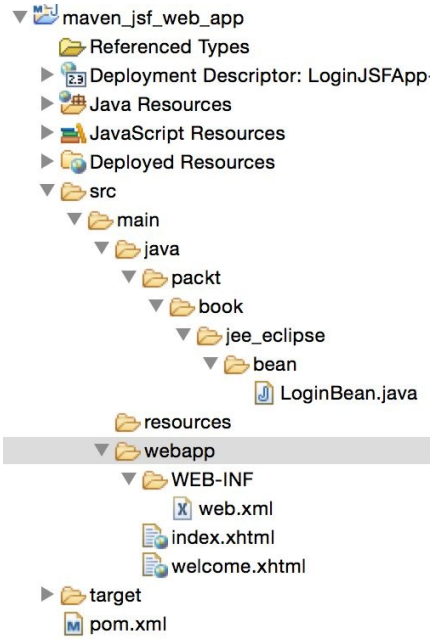

Figure 2.36: Project structure after adding source files

No change is required in the source code. To run the application, right-click on `index.xhtml` and select Run As | Run on Server.

We will be using Maven for project management in the rest of this book.

Creating a WAR file using Maven

In a previous example, we created the WAR file using the Export option of Eclipse. In a Maven project you can create a WAR by invoking the Maven Install plugin. Right-click on the project and select the Run As | Maven install option. The WAR file is created in the `target` folder. You can then deploy the WAR file in Tomcat by copying it to the `webapps` folder of Tomcat.