

# Creating JEE Applications with EJB

In the last chapter, we learned some techniques to debug JEE applications from Eclipse. In this chapter, we will shift our focus back to JEE application development and learn how to create and use **Enterprise JavaBeans (EJB)**. If you recall the architecture of database applications in [Chapter 4, Creating JEE Database Applications](#), we had JSP or a JSF page calling a JSP bean or a managed bean. The beans then called DAOs to execute the data access code. This separated code for the user interface, the business logic, and the database nicely. This would work for small or medium applications, but may prove to be a bottleneck in large enterprise applications; the application may not scale very well. If processing of the business logic is time consuming then it would make more sense to distribute it on different servers for better scalability and resilience. If code for the user interface, the business logic, and the data access is all on the same machine, then it may affect scalability of the application; that is, it may not perform well under the load.

Using EJB for implementing the business logic is ideal in scenarios where you want components processing the business logic to be distributed across different servers. However, this is just one of the advantages of EJB. Even if you use EJBs on the same server as the web application, you may gain from a number of services that the EJB container provides; you can specify the security constraints for calling EJB methods declaratively (using annotations) and can easily specify transaction boundaries (specify a set of method calls from a part of one transaction) using annotations. Furthermore, the container handles the life cycle of EJBs, including pooling of certain types of EJB objects so that more objects can be created when load on the application increases.

In [Chapter 4, Creating JEE Database Applications](#), we created a *Course Management* web application using simple JavaBeans. In this chapter, we will create the same application using EJBs and deploy it on the GlassFish Server. However, before that we need to understand some basic concepts of EJBs.

We will cover the following broad topics:

- Understanding different types of EJBs and how they can be accessed from different client deployment scenarios
- Configuring GlassFish Server for testing EJB applications in Eclipse
- Creating and testing EJB projects from Eclipse with and without Maven

# Types of EJB

EJB can be of the following types according to the EJB3 specification:

- Session bean:
  - Stateful session bean
  - Stateless session bean
  - Singleton session bean
- Message-driven bean

We will discuss **message-driven bean (MDB)** in detail in a [Chapter 10](#), *Asynchronous Programming with JMS*, when we learn about asynchronous processing of requests in the JEE application. In this chapter, we will focus on session beans.

# Session beans

In general, session beans are meant to contain methods to execute the main business logic of the enterprise application. Any **Plain Old Java Object (POJO)** can be annotated with the appropriate EJB3-specific annotations to make it a session bean. Session beans come in three types.

# Stateful session beans

One stateful session bean serves requests for one client only. There is one-to-one mapping between the stateful session bean and the client. Therefore, stateful beans can hold the state data for the client between multiple method calls. In our *Course Management* application, we could use a stateful bean for holding student data (student profile and courses taken by her/him) after a student logs in. The state maintained by the stateful bean is lost when the server restarts or when the session times out. Since there is one stateful bean per client, using a stateful bean might impact scalability of the application.

We use the `@Stateful` annotation on the class to mark it as a stateful session bean.

# Stateless session beans

A stateless session bean does not hold any state information for the client. Therefore, one session bean can be shared across multiple clients. The EJB container maintains pools of stateless beans, and when a client request comes, it takes a bean out of the pool, executes methods, and returns the bean to the pool again. Stateless session beans provide excellent scalability because they can be shared and they need not be created for each client.

We use the `@Stateless` annotation on the class to mark it as a stateless session bean.

# Singleton session beans

As the name suggests, there is only one instance of a singleton bean class in the EJB container (this is true in the clustered environment too; each EJB container will have one instance of a singleton bean). This means that they are shared by multiple clients, and they are not pooled by EJB containers (because there can be only one instance). Since a singleton session bean is a shared resource, we need to manage concurrency in it. Java EE provides two concurrency management options for singleton session beans, namely container-managed concurrency and bean-managed concurrency. Container-managed concurrency can be easily specified by annotations.



*See <https://javaee.github.io/tutorial/ejb-basicexamples003.html#GIPSZ> for more information on managing concurrency in singleton session beans.*

The use of a singleton bean could have an impact on the scalability of the application if there are resource contentions.

We use the `@Singleton` annotation on the class to mark it as a singleton session bean.

# Accessing session beans from a client

Session beans can be designed to be accessed locally (client and bean in the same application), remotely (from a client running in a different application or JVM), or both. In the case of remote access, session beans are required to implement a remote interface. For local access, session beans can implement a local interface or implement no interface (no-interface view of a session bean). The remote and local interfaces that the session bean implements are sometimes also called **business interfaces** because they typically expose the primary business functionality.



# Creating a no-interface session bean

To create a session bean with the no-interface view, create a POJO and annotate it with the appropriate EJB annotation type and `@LocalBean`. For example, we can create a local stateful `Student` bean as follows:

```
import javax.ejb.LocalBean;

import javax.ejb.Singleton;

@Singleton
@LocalBean

public class Student {

    ...

}
```

# Accessing session beans using dependency injection

You can access session beans either using the `@EJB` annotation (which injects the bean in the client class) or by performing the **Java Naming and Directory Interface (JNDI)** lookup. EJB containers are required to make JNDI URLs of the EJBs available to clients.

Injecting session beans using `@EJB` works only for managed components, that is, components of the application whose life cycle is managed by the EJB container. When a component is managed by the container, it is created (instantiated) and destroyed by the container. You do not create managed components using the `new` operator. JEE-managed components that support direct injection of EJBs are servlets, managed beans of JSF pages, and EJBs themselves (one EJB can have another EJB injected into it). Unfortunately, you cannot have a web container inject EJBs in JSPs or JSP beans. Furthermore, you cannot have EJBs injected into any custom classes that you create and that are instantiated using the `new` operator. Later in the chapter, we will see how to use JNDI to access EJBs from objects that are not managed by the container.

We could use a student bean (created previously) from a managed bean of a JSF as follows:

```
import javax.ejb.EJB;
import javax.faces.bean.ManagedBean;

@ManagedBean
public class StudentJSFBean {
    @EJB
    private Student studentEJB;
}
```

Note that if you create an EJB with no-interface view, then all `public` methods in that EJB will be exposed to the client. If you want to control the methods that could be called by the client, then you should implement a business interface.

# Creating session beans using local business interface

Business interface for the EJB is a simple Java interface annotated either with `@Remote` OR `@Local`. Therefore, we can create a local interface for a student bean as follows: `import java.util.List; import javax.ejb.Local;`

`@Local`

```
public interface StudentLocal {  
  
    public List<Course> getCourses(); }  

```

Furthermore, we can implement a session bean as follows: `import java.util.List; import javax.ejb.Local; import javax.ejb.Stateful;`

`@Stateful`

`@Local`

```
public class Student implements StudentLocal {  

```

`@Override`

```
    public List<CourseDTO> getCourses() {  
  
        //get courses are return ...  
  
    }  
  
}
```

The client can access the `Student` EJB only through the local interface: `import javax.ejb.EJB; import javax.faces.bean.ManagedBean;`

@ManagedBean

```
public class StudentJSFBean {
```

@EJB

```
private StudentLocal student; }
```

A session bean can implement multiple business interfaces.

# Accessing session beans using JNDI lookup

Although accessing EJB using dependency injection is the easiest way, it works only if the container manages the class that accesses the EJB. If you want to access EJB from a POJO that is not a managed bean, then dependency injection will not work. Another scenario where dependency injection does not work is when EJB is deployed in a separate JVM (could be on a remote server). In such cases, you will have to access the EJB using JNDI lookup (visit <https://docs.oracle.com/javase/tutorial/jndi/> for more information on JNDI).

JEE applications could be packaged in **Enterprise Application aRchive (EAR)**, which contains a `.jar` file for EJBs and a `.war` file for web applications (and a `lib` folder containing libraries required for both). If, for example, the name of the EAR file is `CourseManagement.ear` and the name of the EJB JAR in it is `CourseManagementEJBs.jar`, then the name of the application is `CourseManagement` (name of the EAR file) and the module name is `CourseManagementEJBs`. The EJB container uses these names to create JNDI URL for looking up EJBs. A global JNDI URL for EJB is created as follows:

```
| "java:global/<application_name>/<module_name>/<bean_name>![<bean_interface>]"
```

Let's have a look at the different parameters used in the preceding code snippets:

- `java:global`: This indicates that it is a global JNDI URL.
- `<application_name>`: This is typically the name of the EAR file.
- `<module_name>`: This is the name of the EJB JAR.
- `<bean_name>`: This is the name of the EJB bean class.
- `<bean_interface>`: This is optional if EJB has a no-interface view, or if EJB implements only one business interface. Otherwise it is a fully qualified name of the business interface.

EJB containers are required to publish two more variations of JNDI URLs for each EJB. These are not global URLs, which means that they can't be used to access EJBs from clients that are not in the same JEE application (in the same

EAR):

- `java:app/[<module_name>]/<bean_name>! [<bean_interface>]`
- `java:module/<bean_name>! [<bean_interface>]`

The first URL can be used if the EJB client is in the same application, and the second URL can be used if the client is in the same module (the same `.jar` file as the EJB).

Before you look up any URL in a JNDI server, you need to create `InitialContext`, which includes, among other things, information such as the hostname of the JNDI server and the port on which it is running. If you create `InitialContext` in the same server, then there is no need to specify these attributes:

```
InitialContext initCtx = new InitialContext();
Object obj = initCtx.lookup("jndi_url");
```

We can use the following JNDI URLs to access a no-interface (`LocalBean`) `Student` EJB (assuming that the name of the EAR file is `CourseManagement` and the name of the `.jar` file for EJBs is `CourseManagementEJBs`):

URL	When to use
<code>java:global/CourseManagement/CourseManagementEJBs/Student</code>	The client can be anywhere in the EAR file, because we use the global URL. Note that we haven't specified the interface name because we are assuming that the student bean provides a no-interface view in this example.
<code>java:app/CourseManagementEJBs/Student</code>	The client can be anywhere in the EAR. We skipped application name because the client is expected to be in the same application, because the namespace of the URL is <code>java:app</code> .

java:module/Student	The client must be in the same .jar file as EJB.
---------------------	--

We can use the following JNDI URLs for accessing `Student` EJB that implemented a local interface called `StudentLocal`:

URL	When to use
java:global/CourseManagement/ CourseManagementEJBs/Student!packt.jee.book.ch6.StudentLocal	The client can be anywhere in the EAR file, because we use a global URL.
java:global/CourseManagement/ CourseManagementEJBs/Student	The client can be anywhere in the EAR. We skipped the interface name because the bean implements only one business

	<p>interface. Note that the object returned from this call will be of <code>StudentLocal</code> type, and not <code>Student</code> type.</p>
<p><code>java:app/CourseManagementEJBs/Student</code></p> <p>Or</p> <p><code>java:app/CourseManagementEJBs/Student!packt.jee.book.ch6.StudentLocal</code></p>	<p>The client can be anywhere in the EAR. We skipped the application name because the JNDI namespace is <code>java:app</code>.</p>
<p><code>java:module/Student</code></p> <p>Or</p> <p><code>java:module/Student!packt.jee.book.ch6.StudentLocal</code></p>	<p>The client must be in the same EAR as the EJB.</p>

Here is an example of how we can call the student bean with a local business interface from one of the objects (that is not managed by the web container) in our web application:



```
|InitialContext ctx = new InitialContext();  
|StudentLocal student = (StudentLocal) ctx.lookup  
|("java:app/CourseManagementEJBs/Student");  
|return student.getCourses(id) ; //get courses from Student EJB
```

# Creating session beans using remote business interface

If the session bean that you create is going to be accessed by a client object that is not in the same JVM as the bean, then the bean needs to implement a remote business interface. You create a remote business interface by annotating the class with `@Remote`:

```
import java.util.List;

import javax.ejb.Remote;

@Remote

public interface StudentRemote {

    public List<CourseDTO> getCourses();

}
```

The EJB implementing the remote interface is also annotated with `@Remote`:

```
@Stateful

@Remote

public class Student implements StudentRemote {

    @Override
```

```

    public List<CourseDTO> getCourses() {

        //get courses are return

        ...

    }

}

```

Remote EJBs can be injected into managed objects in the same application using the `@EJB` annotation. For example, a JSF bean can access the previously mentioned student bean (in the same application) as follows:

```

import javax.ejb.EJB;

import javax.faces.bean.ManagedBean;

@ManagedBean

public class StudentJSFBean {

    @EJB

    private StudentRemote student;

}

```

# Accessing remote session beans

For accessing a remote `student` EJB, we can use the following JNDI URLs:

URL	When to use
<code>java:global/CourseManagement/ CourseManagementEJBs/Student!packt.jee.book.ch6.StudentRemote</code>	The client can be in the same application or remote. In the case of a remote client, we need to set up proper <code>InitialContext</code> parameters.
<code>java:global/CourseManagement/CourseManagementEJBs/Student</code>	The client can be in the same application or remote. We skipped the interface name because the bean implements only one

	business interface.
<pre>java:app/CourseManagementEJBs/Student</pre> <p>Or</p> <pre>java:app/CourseManagementEJBs/Student!packt.jee.book.ch6.StudentRemote</pre>	<p>The client can be anywhere in the EAR. We skipped the application name because the JNDI namespace is <code>java:app</code>.</p>
<pre>java:module/Student</pre> <p>Or</p> <pre>java:module/Student!packt.jee.book.ch6.StudentRemote</pre>	<p>The client must be in the same EAR as the EJB.</p>

To access EJBs from a remote client, you need to use the JNDI lookup method. Furthermore, you need to set up `InitialContext` with certain properties; some of them are JEE application server specific. If the remote EJB and the client are both deployed in GlassFish (different instances of GlassFish), then you can look up the remote EJB as follows:

```
Properties jndiProperties = new Properties();
jndiProperties.setProperty("org.omg.CORBA.ORBInitialHost",
    "<remote_host>");
//target ORB port. default is 3700 in GlassFish
jndiProperties.setProperty("org.omg.CORBA.ORBInitialPort",
    "3700");
```

```
| InitialContext ctx = new InitialContext(jndiProperties);  
| StudentRemote student =  
|   (StudentRemote)ctx.lookup("java:app/CourseManagementEJBs/Student");  
| return student.getCourses();
```

# Configuring the GlassFish Server in Eclipse

We are going to use the GlassFish application server in this chapter. We have already seen how to install GlassFish in the *Installing GlassFish Server* section of [Chapter 1, Introducing JEE and Eclipse](#).

We will first configure the GlassFish Server in Eclipse JEE:

1. To configure the GlassFish Server in Eclipse EE, make sure that you are in the Java EE perspective in Eclipse. Right-click on the Servers view and select New | Server. If you do not see the GlassFish Server group in the list of server types, then expand Oracle node and select and install GlassFish Tools:

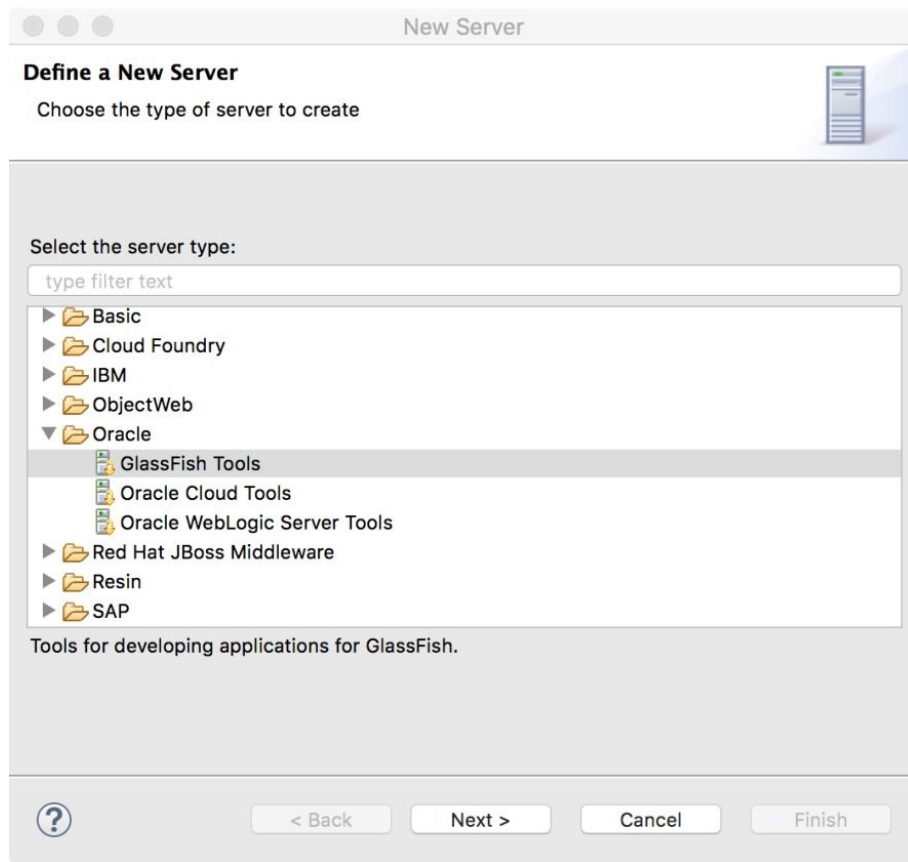


Figure 7.1: Installing GlassFish Tools

2. If you have already installed GlassFish Tools, or if GlassFish Server type is available in the list, then expand that and select the GlassFish option:



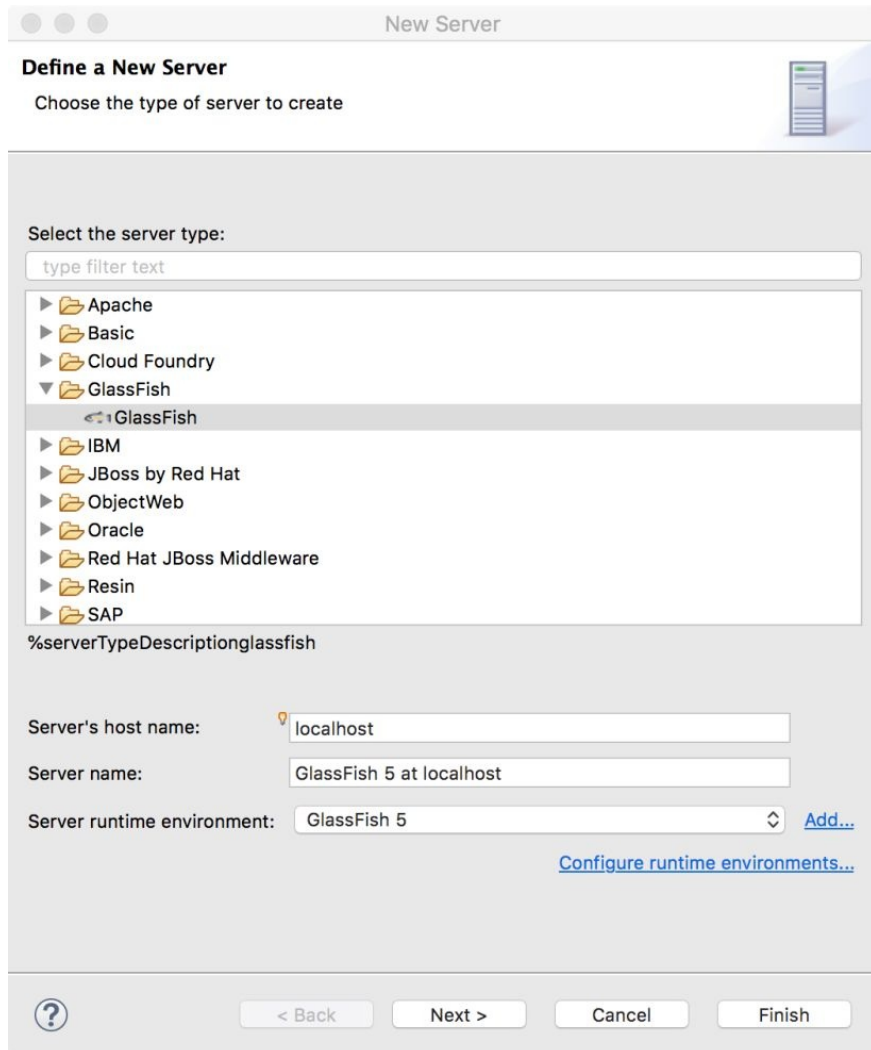


Figure 7.2: Creating GlassFish Server instance in Eclipse

3. Click Next. Enter the path of the GlassFish Server on your local machine in the Domain path field. Enter admin name and password, if applicable, and click Next:

**New Server**

**GlassFish**  
Define GlassFish Application Server properties.

Name: GlassFish 5 [domain1]

Host name: localhost

Domain path: ons/GlassFish/glassfish5/glassfish/domains/domain1

Admin name: admin

Admin password:

Debug port: 8009

☒ Preserve sessions across redeployment

☐ Use JAR archives for deployment

? < Back Next > Cancel Finish

Figure 7.3: Defining GlassFish Server properties

4. The next page allows you to deploy the existing Java EE projects in GlassFish. We don't have any projects to add at this point, so just click Finish.
5. The server is added to the Servers view. Right-click on the server and select Start. If the server is installed and configured properly, then the server status should change to Started.
6. To open the admin page of the server, right-click on the server and select GlassFish | View Admin Console. The admin page is opened in the built-in Eclipse browser. You can browse to the server home page by opening the `http://localhost:8080` URL. 8080 is the default GlassFish port.

# Creating a Course Management application using EJB

Let's now create the *Course Management* application that we created in [chapter 4, Creating JEE Database Applications](#), this time using EJBs. In [chapter 4, Creating JEE Database Applications](#), we created service classes (which were POJOs) for writing the business logic. We will replace them with EJBs. We will start by creating Eclipse projects for EJBs.

# Creating EJB projects in Eclipse

EJBs are packaged in a JAR file. Web applications are packaged in a **Web Application aRchive (WAR)**. If EJBs are to be accessed remotely, then the client needs to have access to business interfaces. Therefore, EJB business interfaces and shared objects are packaged in a separate JAR, called EJB client JAR. Furthermore, if EJBs and web applications are to be deployed as one single application, then they need to be packaged in an EAR.

So, in most cases the application with EJBs is not a single project, but four different projects:

- EJB project that creates EJB JAR
- EJB client project that contains business classes and shared (between EJB and client) classes
- Web project that generates WAR
- EAR project that generates EAR containing EBJ JAR, EJB client JAR, and WAR

You can create each of these projects independently and integrate them. However, Eclipse gives you the option to create EJB projects, EJB client projects, and EAR projects with one wizard:

1. Select File | New | EJB Project. Type `CourseManagementEJBs` in the Project name textbox:

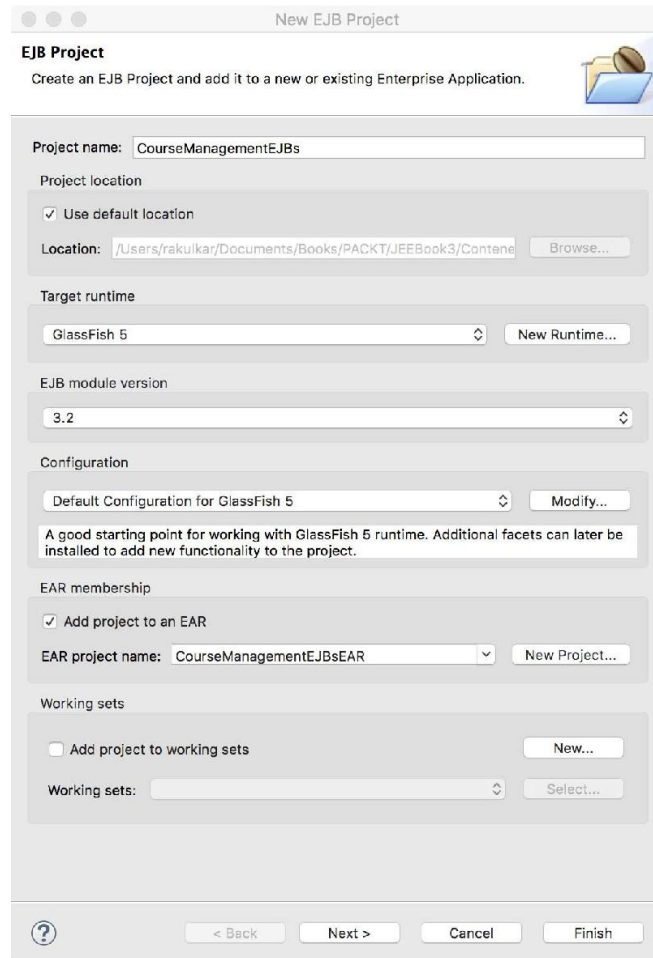


Figure 7.4: New EJB Project wizard

Make sure Target runtime is GlassFish 5 and EJB module version is 3.2 or later. From the Configuration drop-down list, select Default Configuration for GlassFish 5. In the EAR membership group, check the Add project to an EAR box.

2. Select Next. On the next page, specify source and output folders for the classes. Leave the defaults unchanged on this page:

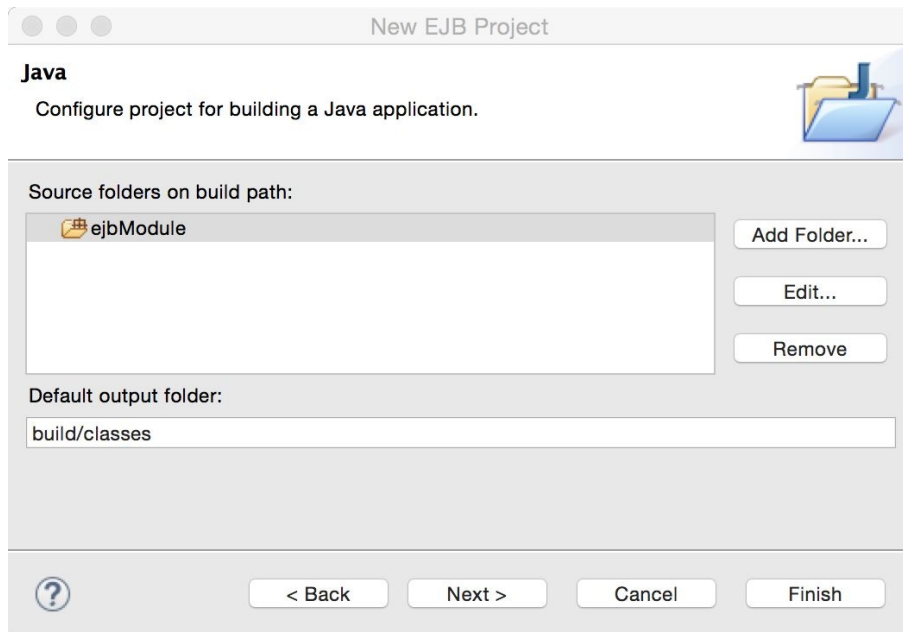


Figure 7.5: Select source and output folders

3. The source Java files in this project would be created in the `ejbModule` folder. Click Next:

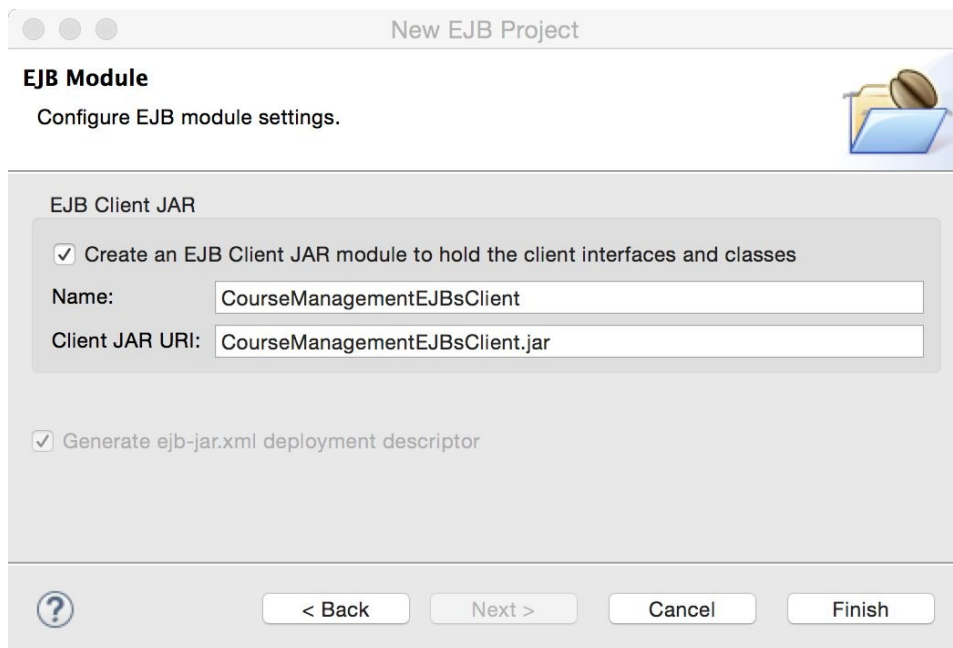


Figure 7.6: Creating an EJB client project

4. Eclipse gives you the option to create an EJB client project. Select the option and click Finish.

5. Since we are building a web application, we will create a web project. Select File | Dynamic Web Project. Set the project name as CourseManagementWeb:

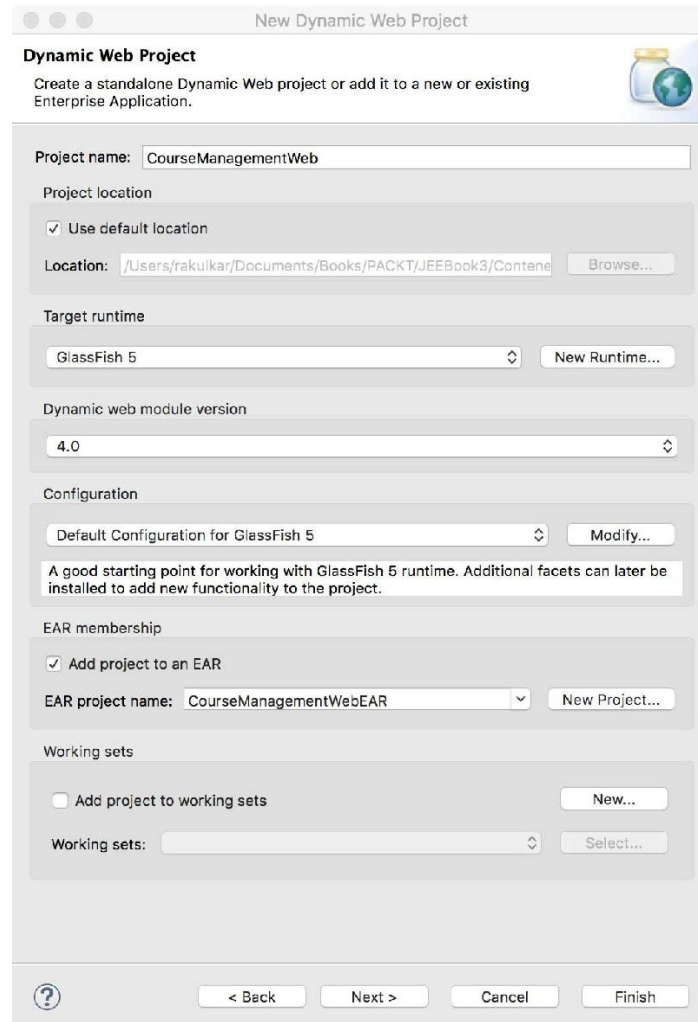


Figure 7.7: New Dynamic Web Project

6. Select the Add Project to an EAR checkbox. Since we have only one EAR project in the workspace, Eclipse selects this project from the drop-down list. Click Finish.

We now have the following four projects in the workspace:

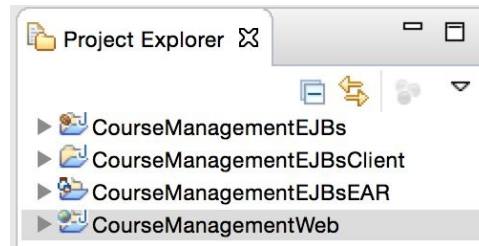


Figure 7.8: Course Management projects

In the course management application, we will create a stateless EJB called `CourseBean`. We will use **Java Persistence APIs (JPA)** for data access and create a `Course` entity. See [Chapter 4, \*Creating JEE Database Applications\*](#), for details on using JPAs. The `CourseManagementEJBClient` project will contain the EJB business interface and shared classes. In `CourseManagementWeb`, we will create a JSF page and a managed bean that will access the `Course` EJB in the `CourseManagementEJBs` project to get a list of courses.



# Configuring datasources in GlassFish

In [Chapter 4, Creating JEE Database Applications](#), we created the JDBC datasource locally in the application. In this chapter, we will create a JDBC datasource in GlassFish. GlassFish Server is not packaged with the JDBC driver for MySQL. So, we need to place the .jar file for `MySQLDriver` in the path where GlassFish can find it. You can place such external libraries in the `lib/ext` folder of the GlassFish domain in which you want to deploy your application. For this example, we will copy the JAR in `<glassfish_home>/glassfish/domains/domain1/lib/ext`.

If you do not have the MySQL JDBC driver, you can download it from

<http://dev.mysql.com/downloads/connector/j/>:

1. Open the GlassFish admin console, either by right-clicking on the server in the Servers view and selecting GlassFish | View Admin Console (this opens the admin console inside Eclipse) or browsing to `http://localhost:4848` (4848 is the default port to which the GlassFish admin console application listens). In the admin console, select Resources | JDBC | JDBC Connection Pools. Click the New button on the JDBC Connection Pool page:

## New JDBC Connection Pool (Step 1 of 2)

Identify the general settings for the connection pool.

### General Settings

Pool Name: *	<input type="text" value="MySQLconnectionPool"/>
Resource Type:	<input type="text" value="javax.sql.DataSource"/>
<small>Must be specified if the datasource class implements more than 1 of the interface.</small>	
Database Driver Vendor:	<input type="text" value="MySql"/>
<small>Select or enter a database driver vendor</small>	
Introspect:	<input type="checkbox"/> Enabled
<small>If enabled, data source or driver implementation class names will enable introspection.</small>	

Figure 7.9: Create JDBC Connection Pool in GlassFish

2. Set Pool Name as `MySQLconnectionPool` and select `javax.sql.DataSource` as Resource Type. Select `MySql` from the Database Driver Vendor list and click Next. In the next page, select the correct Datasource Classname (`com.mysql.jdbc.jdbc2.optional.MysqlDatasource`):

## New JDBC Connection Pool (Step 2 of 2)

Identify the general settings for the connection pool. Datasource Classname or Driver Classname must be

### General Settings

Pool Name:	MySQLconnectionPool
Resource Type:	javax.sql.DataSource
Database Driver Vendor:	MySql
Datasource Classname:	<input type="text" value="com.mysql.jdbc.jdbc2.optional.MysqlDataSource"/>
	<small>Select or enter vendor-specific classname that implements the DataSource</small>
Driver Classname:	<input type="text"/>
	<small>Select or enter vendor-specific classname that implements the java.sql.Driv</small>
Ping:	<input type="checkbox"/> <b>Enabled</b> <small>When enabled, the pool is pinged during creation or reconfiguration to ident</small>
Description:	<input type="text"/>

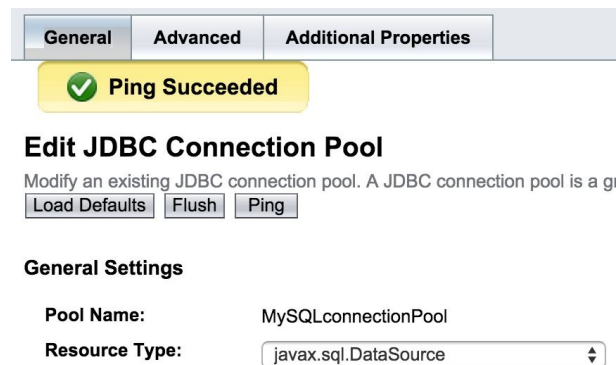
Figure 7.10: JDBC Connection Pool settings in GlassFish

3. We need to set the hostname, port, username, and password of MySQL. In the admin page, scroll down to the Additional Properties section and set the following properties:


Properties	Values
Port/PortNumber	3306
DatabaseName	<schemaname_of_coursemanagement>, for example, course_management. See <a href="#">Chapter 4, Creating JEE Database Applications</a> , for details on creating the MySQL schema for the <i>Course Management</i> database.
Password	MySQL database password.
URL/Url	jdbc:mysql://:3306/<database_name> , for example,

	jdbc:mysql://:3306/course_management
ServerName	localhost
User	MySQL username

- Click Finish. The new connection pool is added to the list in the left pane. Click on the newly added connection pool. In the General tab, click on the Ping button and make sure that the ping is successful:



General Advanced Additional Properties

 Ping Succeeded

### Edit JDBC Connection Pool

Modify an existing JDBC connection pool. A JDBC connection pool is a gr

Load Defaults Flush Ping

#### General Settings

Pool Name: MySQLconnectionPool

Resource Type: javax.sql.DataSource

Figure 7.11: Test JDBC Connection Pool in GlassFish

- Next, we need to create a JNDI resource for this connection pool so that it can be accessed from the client application. Select the Resources | JDBC | JDBC Resources node in the left pane. Click the New button to create a new JDBC resource:

## New JDBC Resource

Specify a unique JNDI name that identifies the JDBC resource you want to create.

JNDI Name: \* jdbc/CourseManagement

Pool Name: MySQLconnectionPool

Use the [JDBC Connection Pools](#) page to create new pools

Description:

Status: ☒ Enabled

Figure 7.12: Test JDBC Connection Pool in GlassFish

6. Set JNDI Name as `jdbc/CourseManagement`. From the Pool Name drop-down list, select the connection pool that we created for MySQL, `MySQLconnectionPool`. Click Save.

# Configuring JPA in an Eclipse project

We will now configure our EJB project to use JPA to access the MySQL database. We have already learned how to enable JPA for an Eclipse project in [chapter 4, Creating JEE Database Applications](#). However, we will briefly cover the steps again here:

1. Right-click on the `courseManagementEJBs` project in Project Explorer and select **Configure | Convert to JPA Project**. Eclipse opens the Project Facets window:

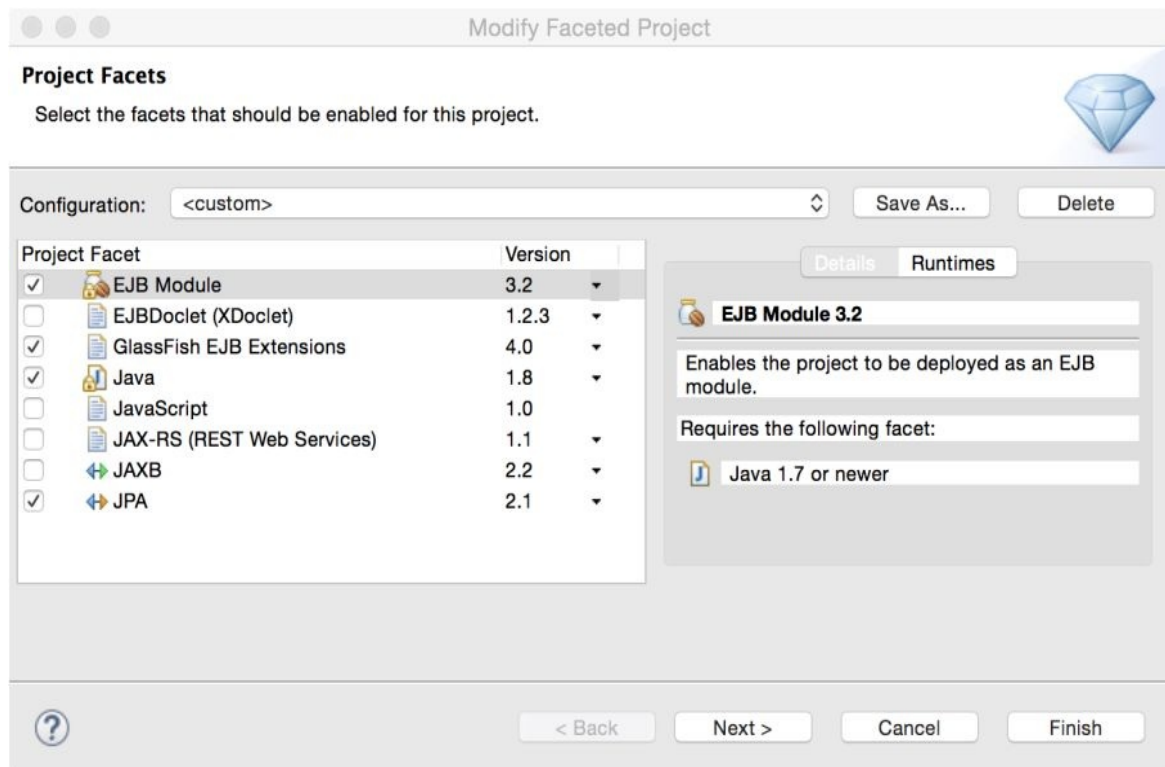


Figure 7.13: Eclipse Project Facets

2. Click Next to go to the JPA Facet page:

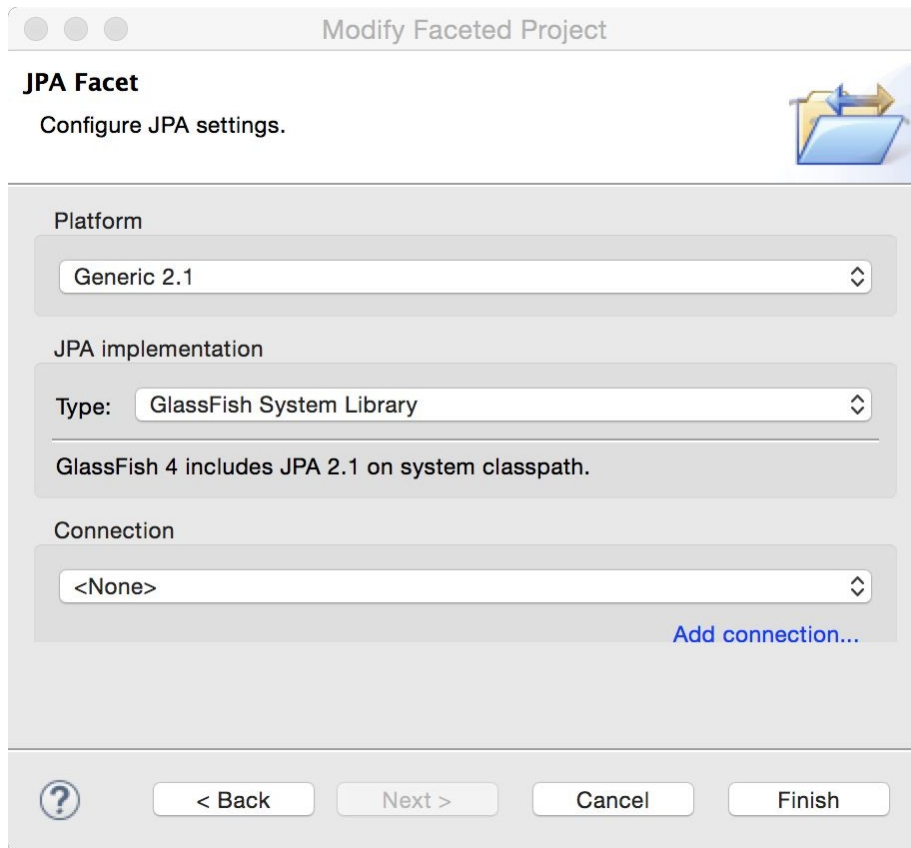


Figure 7.14: JPA Facet

Keep the default values unchanged, and click Finish. Eclipse adds `persistence.xml`, required by JPA, to the project under the JPA Content group in Project Explorer. We need to configure the JPA datasource in `persistence.xml`. Open `persistence.xml` and click on the Connection tab. Set Transaction Type to JTA. In the JTA datasource textbox, type the JNDI name that we set up for our MySQL database in the previous section, which was `jdbc/CourseManagement`. Save the file. Note that the actual location of `persistence.xml` is `ejbModule/META-INF`.

Let's now create a database connection in Eclipse and link it with JPA properties of the project so that we can create JPA entities from the database tables. Right-click on the `CourseManagementEJBs` project and select Properties. This opens the Project Properties window. Click on the JPA node to see the details page. Click on the Add connection link just below the Connection drop-down box. We have already seen how to set up a database connection in the *Using Eclipse Data Source Explorer* section of [chapter 4, Creating JEE Database Applications](#). However, we will quickly recap the steps:

1. In the Connection Profile window, select MySQL:

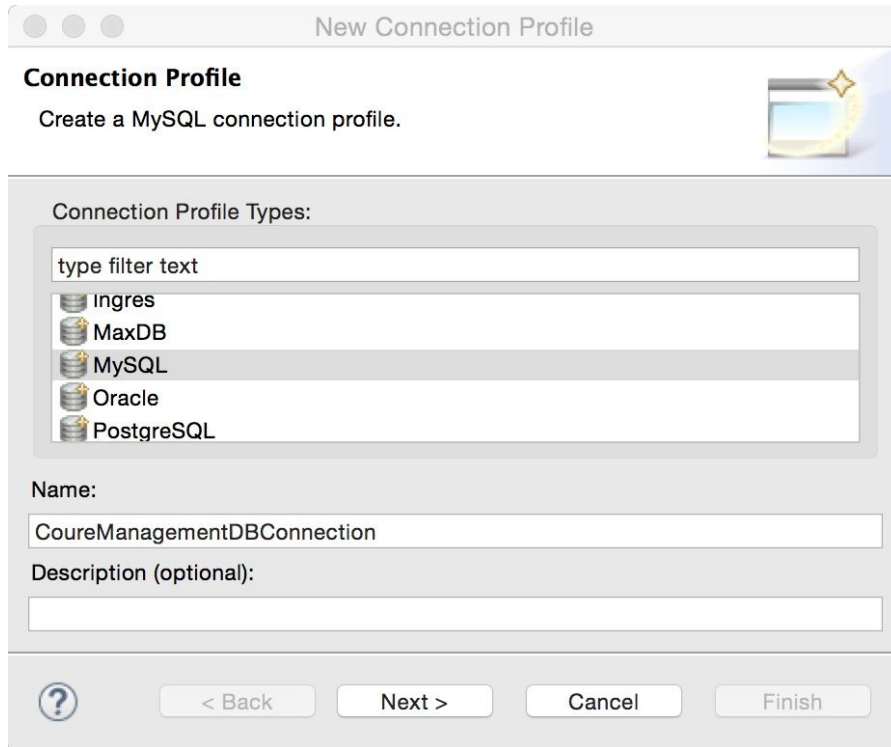


Figure 7.15: New DB Connection Profile

2. Type `CourseManagementDBConnection` in the name textbox and click Next. In the New Connection Profile window, click on the new connection profile button (the circle next to the Drivers drop-down box) to open the New Driver Definition window. Select the appropriate MySQL JDBC Driver version and click on the JAR List tab. In the case of any error, remove any existing `.jar` and click on the Add JAR/Zip button. Browse to the MySQL JDBC driver JAR that we saved in the `<glassfish_home>/glassfish/domains/domain1/lib/ext` folder. Click OK. Back in the New Connection Profile window, enter the database name, modify the connection URL, and enter User name and Password:

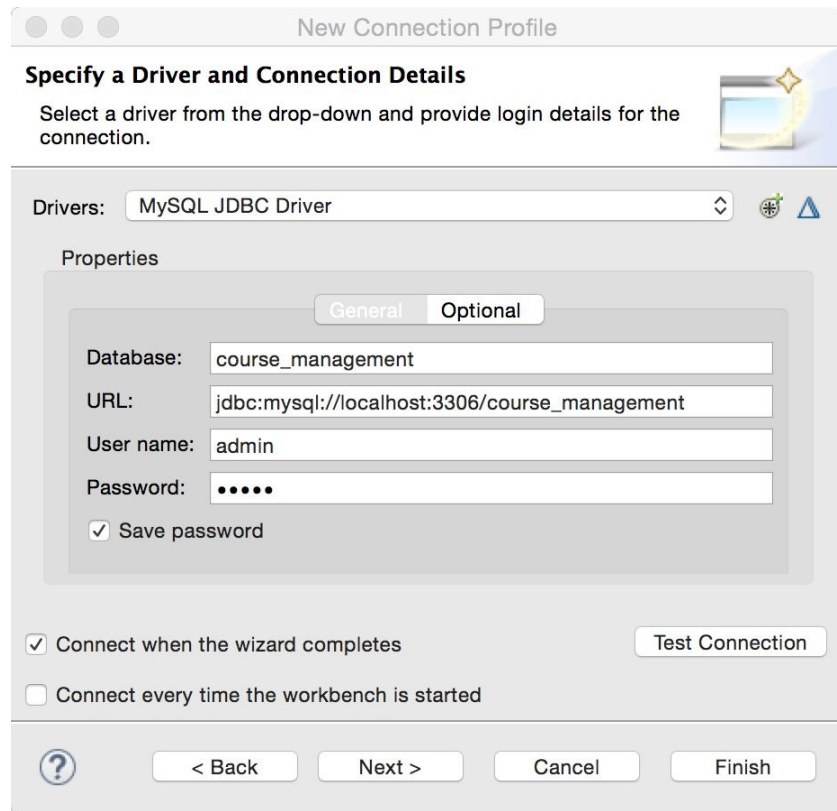


Figure 7.16: Configuring MySQL database connection

3. Select the Save password checkbox. Click the Test Connection button and make sure that the test is successful. Click the Finish button. Back in the JPA properties page, the new connection is added and appropriate schema is selected:



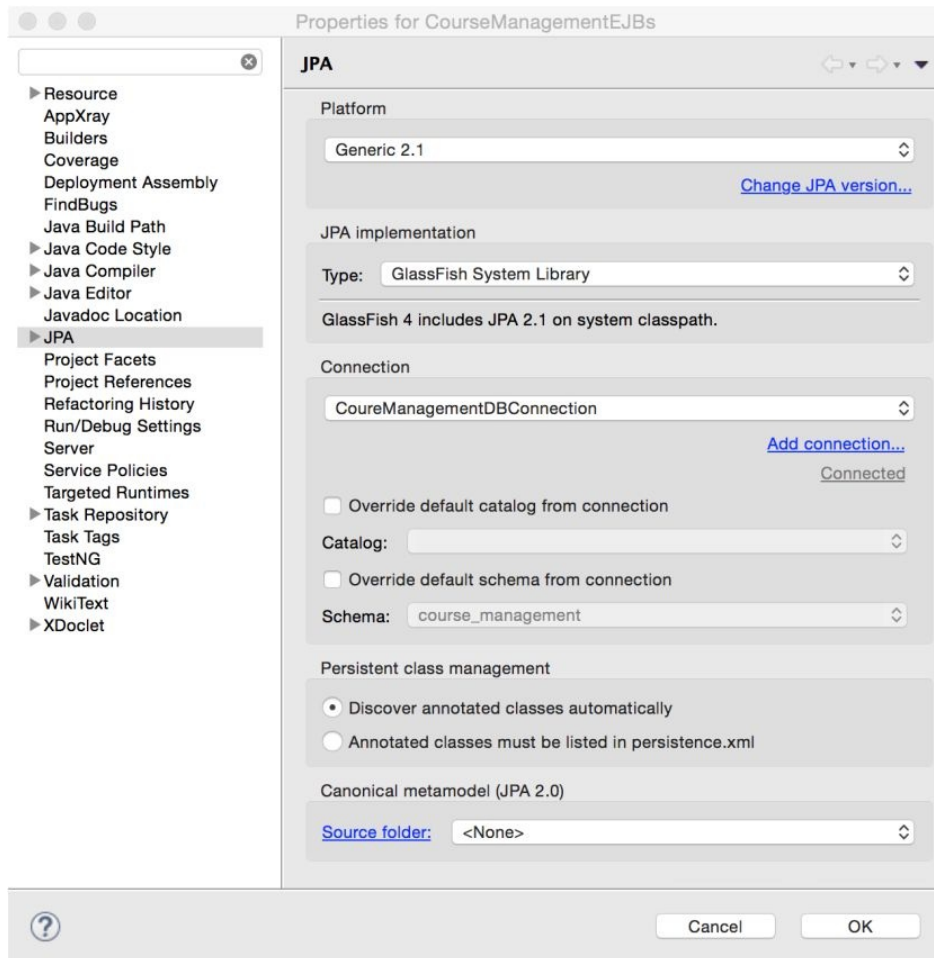


Figure 7.17: Connection added to JPA project properties

4. Click OK to save the changes.

# Creating a JPA entity

We will now create the entity class for `course`, using Eclipse JPA tools:

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

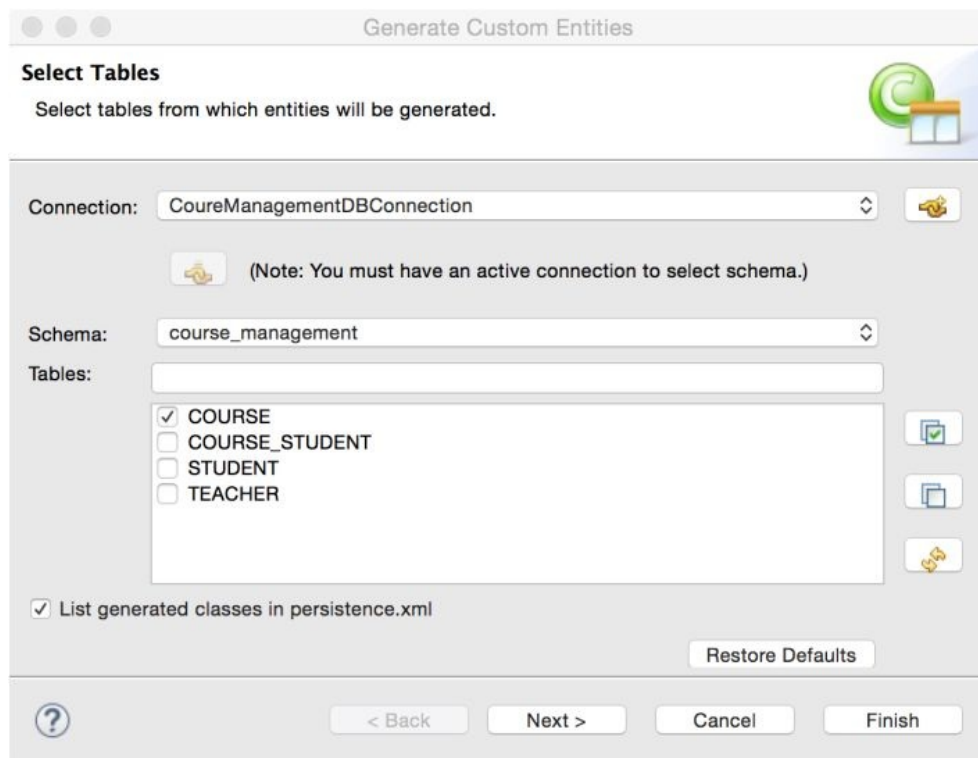


Figure 7.18: Creating entity from tables

2. Select the `Course` table and click Next. Click Next in the Table Associations window. On the next page, select `identity` as Key generator:

**Generate Custom Entities**

**Customize Defaults**

Optionally customize aspects of entities that will be generated by default from database tables. A Java package should be specified.

**Mapping defaults**

Key generator:

Sequence name:

You can use the patterns \$table and/or \$pk in the sequence name. These patterns will be replaced by the table name and the primary key column name when a table mapping is generated.

Entity access: ☒ Field ☐ Property

Associations fetch: ☒ Default ☐ Eager ☐ Lazy

Collection properties type: ☐ java.util.Set ☒ java.util.List

☐ Always generate optional JPA annotations and DDL parameters

**Domain java class**

Source folder:

Package:

Superclass:

Interfaces:

Figure 7.19: Customizing JPA entity details

3. Enter the package name. We do not want to change anything on the next page, so click Finish. Notice that the wizard creates a `findAll` query for the class that we can use to get all courses:

```
@Entity
@NamedQuery(name="Course.findAll", query="SELECT c FROM
Course c")
public class Course implements Serializable { ...}
```

# Creating stateless EJB

We will now create the stateless EJB for our application:

1. Right-click on the `ejbModule` folder in the `CourseManagementEJBs` project in Project Explorer and select **New | Session Bean (3.x)**. Type `packt.book.jee.eclipse.ch7.ejb` in the Java package textbox and `CourseBean` in Class name. Select the Remote checkbox:

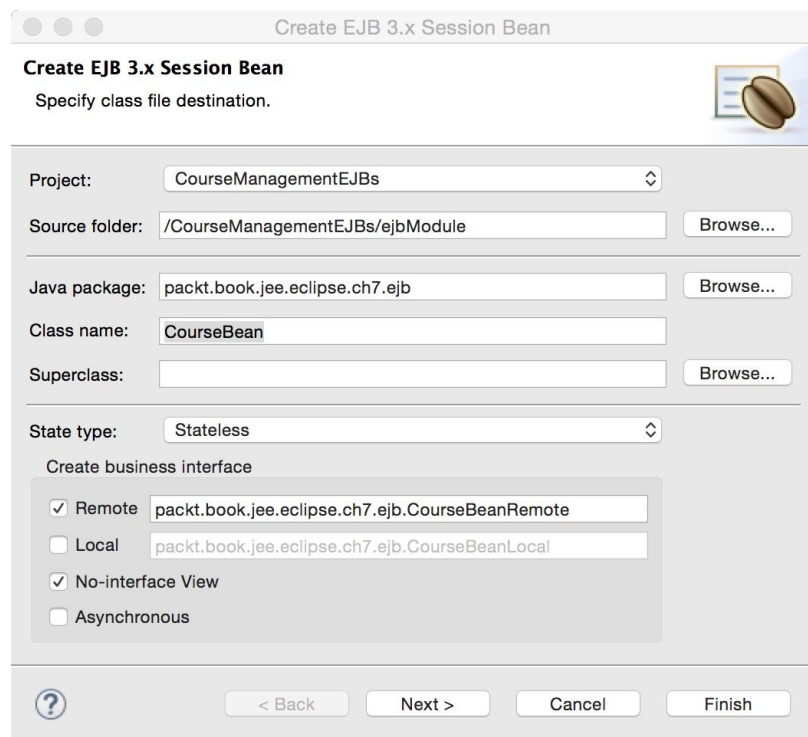


Figure 7.20: Creating a stateless session bean

2. Click Next. No change is required on the next page:

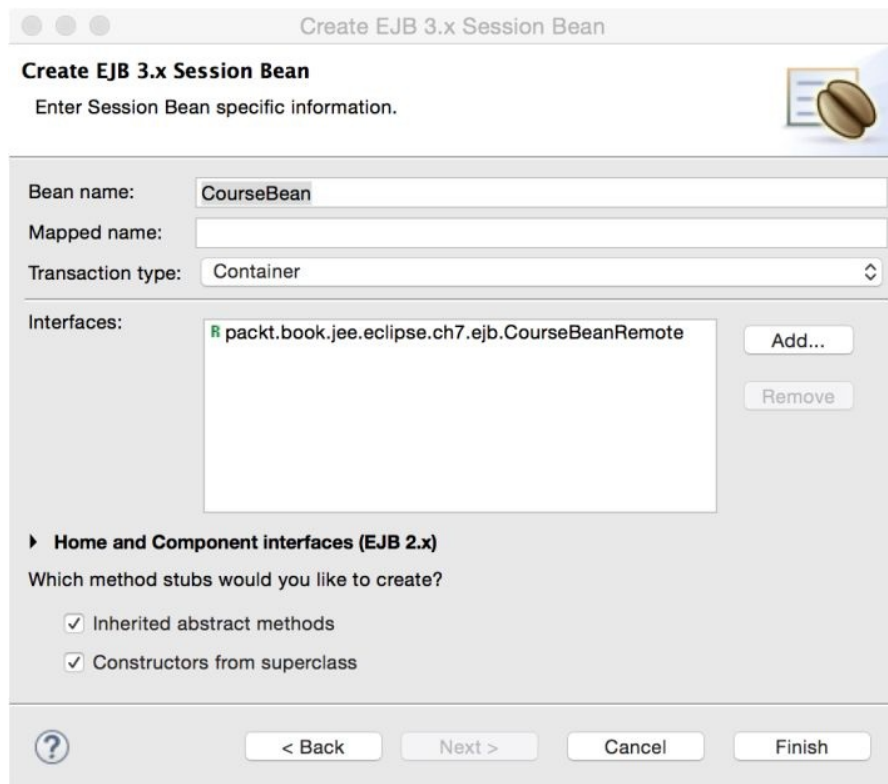


Figure 7.21: Stateless session bean information

3. Click Finish. A `CourseBean` class is created with `@Stateless` and `@LocalBean` annotations. The class also implements the `CourseBeanRemote` interface, which is defined in the `CourseManagementEJBClient` project. This interface is a shared interface (a client calling EJB needs to access this interface):

```
@Stateless
@LocalBean
public class CourseBean implements CourseBeanRemote {
    public CourseBean() {
    }
}
```

The interface is annotated with `@Remote`:

```
@Remote
public interface CourseBeanRemote {
}
```

Now, the question is how do we return `course` information from our EJB? The EJB will call JPA APIs to get instances of the `course` entity, but do we want EJB to return instances of the `course` entity or should it return instances of lightweight

**data transfer object (DTO)?** Each has its own advantages. If we return a `Course` entity, then we do not need to transfer data between objects; which we will have to do in the case of DTO (transfer data from the entity to the corresponding DTO). However, passing entities between layers may not be a good idea if the EJB client is not in the same application, and you may not want to expose your data model to external applications. Furthermore, by passing back JPA entities you are forcing the client application to depend on JPA libraries in its implementation.

DTOs are lightweight, and you can expose only those fields that you want your clients to use. However, you will have to transfer data between entities and DTOs.

If your EJBs are going to be used by the client in the same application, then it could be easier to transfer entities to the client from the EJBs. However, if your client is not part of the same EJB application, or when you want to expose the EJB as a web service (we will learn how to create web services in [Chapter 9, Creating Web Services](#)), then you may need to use DTOs.

In our application, we will see examples of both the approaches, that is, an EJB method returning JPA entities as well as DTOs. Remember that we have created `CourseBean` as a remote as well as a local bean (no-interface view). Implementation of the remote interface method will return DTOs and that of the local method will return JPA entities.

Let's now add the `getCourses` method to the EJB. We will create `CourseDTO`, a data transfer object, which is a POJO, and returns instances of the DTO from the `getCourses` method. This DTO needs to be in the `CourseManagementEJBsClient` project because it will be shared between the EJB and its client.

Create the following class in the `packt.book.jee.eclipse.ch7.dto` package in the `CourseManagementEJBsClient` project:

```
package packt.book.jee.eclipse.ch7.dto;

public class CourseDTO {
    private int id;
    private int credits;
    private String name;
    public int getId() {
        return id;
    }
}
```

```

    public void setId(int id) {
        this.id = id;
    }
    public int getCredits() {
        return credits;
    }
    public void setCredits(int credits) {
        this.credits = credits;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}

```

Add the following method to `CourseBeanRemote`:

```

public List<CourseDTO> getCourses();

```

We need to implement this method in `CourseBean` EJB. To get the courses from the database, the EJB needs to first get an instance of `EntityManager`. Recall that in [Chapter 4, Creating JEE Database Applications](#), we created `EntityManagerFactory` and got an instance of `EntityManager` from it. Then, we passed that instance to the service class, which actually got the data from the database using JPA APIs.

JEE application servers make injecting `EntityManager` very easy. You just need to create the `EntityManager` field in the EJB class and annotate it with

`@PersistenceContext(unitName="<name_as_specified_in_persistence.xml>")`. The `unitName` attribute is optional if there is only one persistence unit defined in `persistence.xml`. Open the `CourseBean` class and add the following declaration:

```

@PersistenceContext
EntityManager entityManager;

```

EJBs are managed objects, and the EJB container injects `EntityManager` after EJBs are created.



*Auto injection of objects is a part of JEE features called **Context and Dependency Injection (CDI)**. See <https://javaee.github.io/tutorial/cdi-basic.html#GIWHB> for information on CDI.*

Let's now add a method to `CourseBean` EJB that will return a list of `Course` entities. We will name this method `getCourseEntities`. This method will be called by the `getCourses` method in the same EJB, which will then convert the list of entities to DTOs. The method `getCourseEntities` can also be called by any web application,

because the EJB exposes no-interface view (using the `@LocalBean` annotation):

```
public List<Course> getCourseEntities() {  
    //Use named query created in Course entity using @NamedQuery  
    annotation. TypedQuery<Course> courseQuery =  
    entityManager.createNamedQuery("Course.findAll", Course.class);  
    return courseQuery.getResultList();  
}
```

After implementing the `getCourses` method (defined in our remote business interface called `CourseBeanRemote`), we have `CourseBean`, as follows:

```
@Stateless  
@LocalBean  
public class CourseBean implements CourseBeanRemote {  
    @PersistenceContext  
    EntityManager entityManager;  
  
    public CourseBean() {  
    }  
  
    public List<Course> getCourseEntities() {  
        //Use named query created in Course entity using @NamedQuery  
        annotation. TypedQuery<Course> courseQuery =  
        entityManager.createNamedQuery("Course.findAll", Course.class);  
        return courseQuery.getResultList();  
    }  
  
    @Override  
    public List<CourseDTO> getCourses() {  
        //get course entities first  
        List<Course> courseEntities = getCourseEntities();  
  
        //create list of course DTOs. This is the result we will  
        return  
        List<CourseDTO> courses = new ArrayList<CourseDTO>();  
  
        for (Course courseEntity : courseEntities) {  
            //Create CourseDTO from Course entity  
            CourseDTO course = new CourseDTO();  
            course.setId(courseEntity.getId());  
            course.setName(courseEntity.getName());  
            course.setCredits(courseEntity.getCredits());  
            courses.add(course);  
        }  
        return courses;  
    }  
}
```



# Creating JSF and managed beans

We will now create a JSF page to display courses in the `CourseManagementWeb` project. We will also create a managed bean that will call the `getCourses` method of `CourseEJB`. See the *Java Server Faces* section in [chapter 2, Creating a Simple JEE Web Application](#), for details about JSF.

As explained in [chapter 2, Creating a Simple JEE Web Application](#), we need to add JSF Servlet and mapping to `web.xml`. Open `web.xml` from the `CourseManagementWeb` project. You can open this file either by double-clicking the Deployment Descriptor: `CourseManagementWeb` node (under the project in Project Explorer) or from the `WebContent/Web-INF` folder (again, under the project in Project Explorer). Add the following servlet declaration and mapping (within the `web-app` node):

```
<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>
```

The `CourseManagementWeb` project needs to access the business interface of EJB, which is in `CourseManagementEJBsClient`. So, we need to add the reference of `CourseManagementEJBsClient` to `CourseManagementWeb`. Open the project properties of `CourseManagementWeb` (right-click on the `CourseManagementWeb` project and select Properties) and select Java Build Path. Click on the Projects tab, and then click the Add... button. Select `CourseManagementEJBsClient` from the list and click OK:

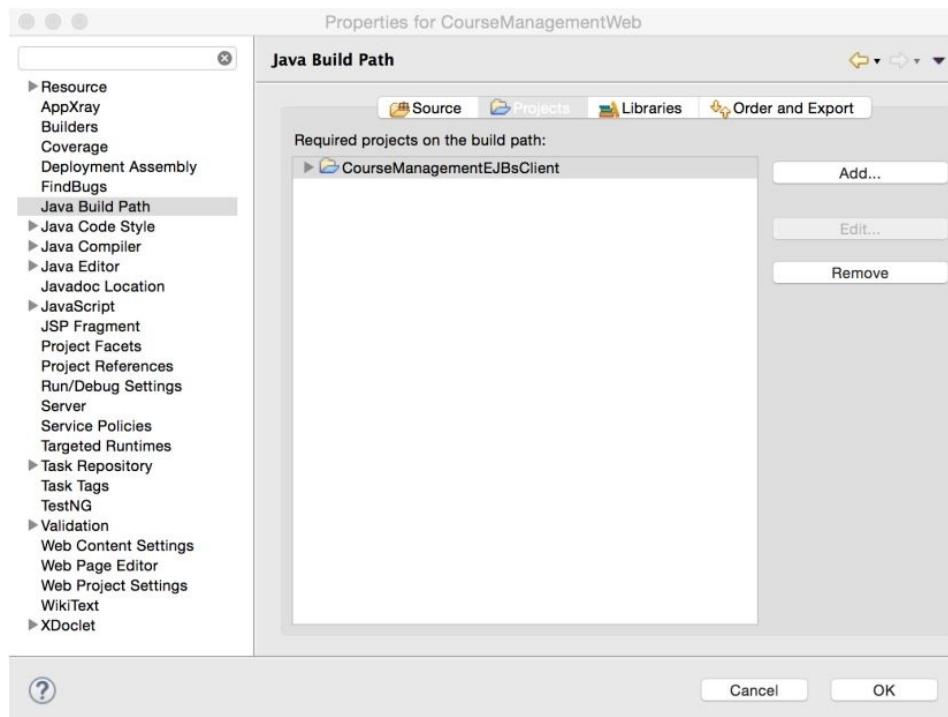


Figure 7.22: Adding project reference Now, let's create a managed bean for the JSF that we are going to create later. Create a `CourseJSFBean` class in the `packt.book.jee.eclipse.ch7.web.bean` package in the `CourseManagementWeb` project (Java source files go in the `src` folder under the Java Resources group): `import java.util.List; import javax.ejb.EJB; import javax.faces.bean.ManagedBean; import packt.book.jee.eclipse.ch7.dto.CourseDTO; import packt.book.jee.eclipse.ch7.ejb.CourseBeanRemote; @ManagedBean(name="Course") public class CourseJSFBean { @EJB CourseBeanRemote courseBean; public List<CourseDTO> getCourses() { return courseBean.getCourses(); } }`

JSF beans are managed beans, so we can have the container inject EJBs using the `@EJB` annotation. In the preceding code we have referenced `CourseBean` with its remote interface, `CourseBeanRemote`. We then created a method called `getCourses`, which calls the method with the same name on `Course` EJB and returns the list of `CourseDTO` objects.

Note that we have set the `name` attribute in the `@ManagedBean` annotation. This managed bean would be accessed from JSF as variable `Course`.

We will now create the JSF page, `course.xhtml`. Right-click on `WebContent` group in the `CourseManagementWeb` project, and select `New | File`. Create `courses.xhtml` with 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"> <head> <title>Courses</title> </head> <body> <h2>Courses</h2> <h:dataTable value="#{Course.courses}" var="course"> <h:column> <f:facet name="header">Name</f:facet> #{course.name} </h:column> <h:column> <f:facet name="header">Credits</f:facet> #{course.credits} </h:column> </h:dataTable> </body> </html>`

The page uses the `dataTable` tag (<https://docs.oracle.com/javaee/7/javadoc/jsp/h/dataTable.html>), which receives the data to populate from the `Course` managed bean (which is actually the `CourseJSFBean` class). `Course.courses` in the expression language syntax is a short-form for `Course.getCourses()`. This results in a call to the `getCourses` method of the `courseJSFBean` class.

Each element of the list returned by `Course.courses`, which is `List` of `CourseDTO`, is represented by the `course` variable (in the `var` attribute value). We then display the name and credits of each course in the table using the `column` child tag.

# Running the example

Before we can run the example, we need to start the GlassFish Server and deploy our JEE application in it:

1. Start the GlassFish Server.
2. Once it is started, right-click on the GlassFish Server in the Servers view and select the Add and Remove... menu option:

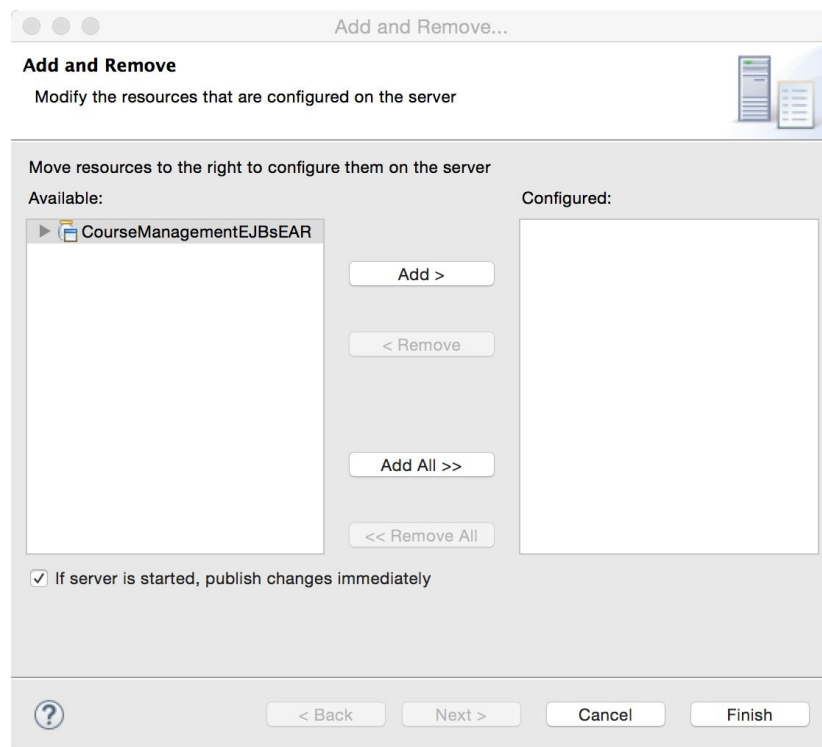


Figure 7.23: Adding a project to GlassFish for deployment

3. Select the EAR project and click on the Add button. Then, click Finish. The selected EAR application will be deployed in the server:



Figure 7.24: Application deployed in GlassFish

4. To run the JSF page, `course.xhtml`, right-click on it in Project Explorer and select Run As | Run on Server. The page will be opened in the internal Eclipse browser and courses in the MySQL database will be displayed on the page.

Note that we can use `courseBean` (EJB) as a local bean in `courseJSFBean`, because they are in the same application deployed on the same server. To do this, add a reference of the `CourseManagementEJBs` project in the build path of `CourseManagementWeb` (open the project properties of `CourseManagementWeb`, select Java Build Path, select the Projects tab, and click the Add... button. Select the `CourseManagementEJBs` project and add its reference).

Then, in the `courseJSFBean` class, remove the declaration of `courseBeanRemote` and add one for `courseBean`:

```
//@EJB
//CourseBeanRemote courseBean;

@EJB
CourseBean courseBean;
```

When you make any changes in the code, the EAR project needs to be redeployed in the GlassFish Server. In Servers view, you can see whether redeployment is needed by checking the status of the server. If it is [Started, Synchronized], then no redeployment is needed. However, if it is [Started, Republish], then redeployment is required. Just click on the server node and select the Publish menu option.

# Creating EAR for deployment outside Eclipse

In the last section, we learned how to deploy an application to GlassFish from Eclipse. This works fine during development, but finally you will need to create the EAR file for deployment to an external server. To create the EAR file from the project, right-click on the EAR project (in our example, it is CourseManagementEJBsEAR) and select Export | EAR file:

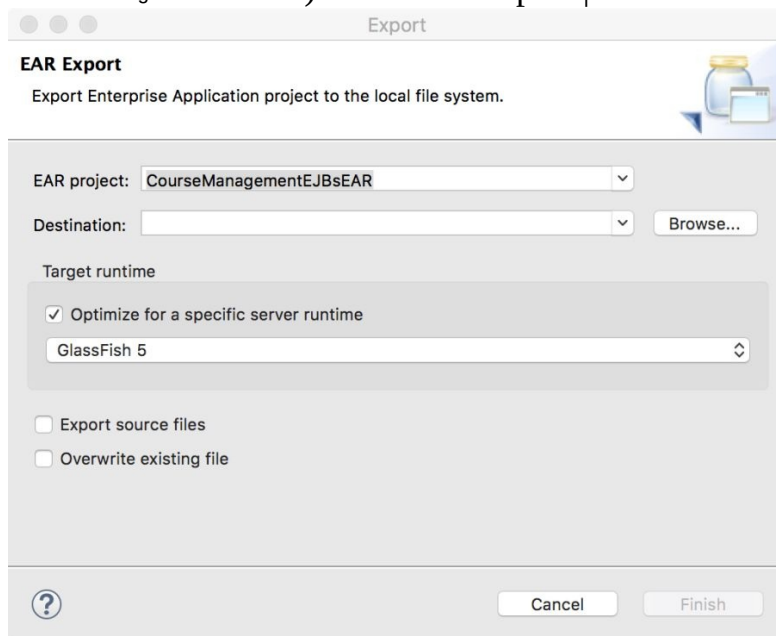


Figure 7.25: Exporting to EAR file Select the destination folder and click Finish. This file can then be deployed in GlassFish using the management console or by copying it to the `autodeploy` folder in GlassFish.

# Creating a JEE project using Maven

In this section, we will learn how to create JEE projects with EJBs using Maven. Creating Maven projects may be preferable to Eclipse JEE projects because builds can be automated. We have seen many details of creating EJBs, JPA entities, and other classes in the previous section, so we won't repeat all that information here. We have also learned how to create Maven projects in [Chapter 2, Creating a Simple JEE Web Application](#), and [Chapter 3, Source Control Management in Eclipse](#), so the basic details of creating a Maven project will not be repeated either. We will focus mainly on how to create EJB projects using Maven. We will create the following projects:

- `CourseManagementMavenEJBs`: This project contains EJBs
- `CourseManagementMavenEJBClient`: This project contains shared interfaces and objects between an EJB project and the client projects
- `CourseManagementMavenWAR`: This is a web project containing a JSF page and a managed bean
- `CourseManagementMavenEAR`: This project creates the EAR file that can be deployed in GlassFish
- `CourseManagement`: This project is the overall parent project that builds all the previously mentioned projects

We still start with `CourseManagementMavenEJBs`. This project should generate the EJB JAR file. Let's create a Maven project with the following details:

Field	Value
Group ID	packt.book.jee.eclipse.ch7.maven
Artifact ID	CourseManagementMavenEJBClient

Version	1
Packaging	jar

We need to add the dependency of JEE APIs to our EJB project. Let's add the dependency of `javax.javaee-api`. Since we are going to deploy this project in GlassFish, which comes with its own JEE implementation and libraries, we will scope this dependency as provided. Add the following in `pom.xml`:

```
<dependencies> <dependency> <groupId>javax</groupId> <artifactId>javaee-api</artifactId> <version>8.0</version> <scope>provided</scope>
</dependency> </dependencies>
```

When we create the EJBs in this project, we need to create local or remote business interfaces in a shared project (client project). Therefore, we will create `CourseManagementMavenEJBClient` with the following details:

Field	Values
Group ID	packt.book.jee.eclipse.ch7.maven
Artifact ID	CourseManagementMavenEJBs
Version	1
Packaging	jar

---

This shared project also needs to access EJB annotations. So, add the same dependency for `javax.javaee-api` that we added previously to the `pom.xml` file of the `CourseManagementMavenEJBClient` project.

We will create a `packt.book.jee.eclipse.ch7.ejb` package in this project and create a remote interface. Create a `CourseBeanRemote` interface (just as we created in the *Creating stateless EJB* section of this chapter). Furthermore, create a `CourseDTO` class in the `packt.book.jee.eclipse.ch7.dto` package. This class is the same as the one that we created in the *Creating stateless EJB* section.

We are going to create a `Course` JPA entity in the `CourseManagementMavenEJBs` project. Before we do that, let's convert this project to a JPA project. Right-click on the project in Package Explorer and select `Configure | Convert to JPA Project`. In the JPA configuration wizard, select the following JPA facet details:

Fields	Values
Platform	Generic 2.1
JPA Implementation	Disable Library Configuration

JPA wizard creates a `META-INF` folder in the `src` folder of the project and creates `persistence.xml`. Open `persistence.xml` and click on the `Connection` tab. We have already created the MySQL datasource in GlassFish (see the *Configuring datasource in GlassFish* section). Enter the JNDI name of the datasource, `jdbc/CourseManagement`, in the JTA data source field.



Create a `Course` entity in `packt.book.jee.eclipse.ch7.jpa`, as described in the *Creating JPA entity* section. Before we create the EJB in this project, let's add an EJB facet to this project. Right-click on the project and select Properties. Click on the Project Facets group and select the EJB Module checkbox. Set version to the latest one (at the time of writing, the latest version was 3.2). We will now create the implementation class of the remote session bean interface that we created previously. Right-click on the `CourseManagementMavenEJBs` project and select the New | Session Bean menu. Create the EJB class with the following details:

Fields	Values
Java package	<code>packt.book.jee.eclipse.ch7.ejb</code>
Class name	<code>CourseBean</code>
State type	Stateless

Do not select any business interface, because we have already created the business interface in the `CourseManagementMavenEJBClient` project. Click Next. On the next page, select `CourseBeanRemote`. Eclipse will show errors at this point because `CourseManagementMavenEJBs` does not know about `CourseManagementMavenEJBClient`, which contains the `CourseBeanRemote` interface, used by `CourseBean` in the EJB project. Adding the Maven dependency (in `pom.xml`) for `CourseManagementMavenEJBClient` in `CourseManagementMavenEJBs` and implementing the `getCourses` method in the EJB class should fix the compilation errors. Now complete the implementation of the `CourseBean` class as described in the *Creating stateless EJB* section of this chapter. Make sure that EJB is marked as Remote: `@Stateless @Remote public class CourseBean implements CourseBeanRemote { ... }`

Let's create a web application project for course management using Maven.

Create a Maven project with the following details:

Fields	Values
Group ID	packt.book.jee.eclipse.ch7.maven
Artifact ID	CourseManagementMavenWebApp
Version	1
Packaging	war

To create `web.xml` in this project, right-click on the project and select **Java EE Tools | Generate Deployment Descriptor Stub**. The `web.xml` file is created in the `src/main/webapp/WEB-INF` folder. Open `web.xml` and add the servlet definition and mapping for JSF (see the *Creating JSF and managed bean* section of this chapter). Add the dependency of the `CourseManagementMavenEJBClient` project and `javax.javaee-api` in `pom.xml` of the `CourseManagementMavenWebApp` project so that the web project has access to the EJB business interface declared in the shared project and also to EJB annotations.

Let's now create a `CourseJSFBean` class in the web project as described in the *Creating JSF and managed bean* section. Note that this will reference the remote interface of the EJB in the managed bean, as follows:

```
@ManagedBean(name="Course") public class CourseJSFBean { @EJB
CourseBeanRemote courseBean; public List<CourseDTO> getCourses() { return
courseBean.getCourses(); } }
```

Create `course.xhtml` in the `webapp` folder as described in the *Creating JSF and*

*managed bean* section.

Let's now create a `CourseManagementMavenEAR` project with the following details:

Fields	Values
Group ID	packt.book.jee.eclipse.ch7.maven
Artifact ID	CourseManagementMavenEAR
Version	1
Packaging	ear

You will have to type `ear` in the Packaging file; there is no `ear` option in the drop-down list. Add dependencies of `web`, `ejb`, and `client` projects to `pom.xml`, as follows:

```
<dependencies> <dependency>
<groupId>packt.book.jee.eclipse.ch7.maven</groupId>
<artifactId>CourseManagementMavenEJBClient</artifactId>
<version>1</version> <type>jar</type> </dependency> <dependency>
<groupId>packt.book.jee.eclipse.ch7.maven</groupId>
<artifactId>CourseManagementMavenEJBs</artifactId> <version>1</version>
<type>ejb</type> </dependency> <dependency>
<groupId>packt.book.jee.eclipse.ch7.maven</groupId>
<artifactId>CourseManagementMavenWebApp</artifactId>
<version>1</version> <type>war</type> </dependency> </dependencies>
```

Make sure to set `<type>` of each dependency properly. You also need to update JNDI URLs for any name changes.

Maven does not have built-in support to package EAR. However, there is a

Maven plugin for EAR. You can find details of this plugin at <https://maven.apache.org/plugins/maven-ear-plugin/> and <https://maven.apache.org/plugins/maven-ear-plugin/modules.html>. We need to add this plugin to our `pom.xml` and configure its parameters. Our EAR file will contain the JAR for the EJB project, the client project, and the WAR for the web project. Right-click on `pom.xml` of the EAR project, and select Maven | Add Plugin. Type `ear` in the Filter box, and select the latest plugin version under `maven-ear-plugin`. Make sure that you also install the `maven-acr-plugin` plugin. Configure the EAR plugin in the `pom.xml` details, as follows:

```
<build> <plugins> <plugin> <groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-acr-plugin</artifactId> <version>1.0</version>
<extensions>true</extensions> </plugin> <plugin>
<groupId>org.apache.maven.plugins</groupId> <artifactId>maven-ear-
plugin</artifactId> <version>2.10</version> <configuration>
<version>6</version> <defaultLibBundleDir>lib</defaultLibBundleDir>
<modules> <webModule>
<groupId>packt.book.jee.eclipse.ch7.maven</groupId>
<artifactId>CourseManagementMavenWebApp</artifactId> </webModule>
<ejbModule> <groupId>packt.book.jee.eclipse.ch7.maven</groupId>
<artifactId>CourseManagementMavenEJBs</artifactId> </ejbModule> <
jarModule > <groupId>packt.book.jee.eclipse.ch7.maven</groupId>
<artifactId>CourseManagementMavenEJBClient</artifactId> </ jarModule >
</modules> </configuration> </plugin> </plugins> </build>
```

After modifying `pom.xml`, sometimes Eclipse may display the following error: Project configuration is not up-to-date with `pom.xml`. Run Maven->Update Project or use Quick Fix...

In such cases, right-click on the project and select Maven | Update Project.

The last project that we create in this section is `CourseManagement`, which will be the container project for all other EJB projects. When this project is installed, it should build and install all the contained projects. Create a Maven project with the following details:

Fields	Values

Group ID	packt.book.jee.eclipse.ch7.maven
Artifact ID	CourseManagement
Version	1
Packaging	Pom

Open `pom.xml` and click on the Overview tab. Expand the Modules group, and add all the other projects as modules. The following modules should be listed in `pom.xml`: `<modules> <module>../CourseManagementMavenEAR</module> <module>../CourseManagementMavenEJBClient</module> <module>../CourseManagementMavenEJBs</module> <module>../CourseManagementMavenWebApp</module> </modules>`

Right-click on the `CourseManagement` project and select Run As | Maven Install. This builds all EJB projects, and an EAR file is created in the target folder of the `CourseManagementMavenEAR` project. You can deploy this EAR in GlassFish from its management console, or you can right-click on the configured GlassFish Server in the Servers view of Eclipse, select the Add and Remove... option, and deploy the EAR project from within Eclipse. Browse to `http://localhost:8080/CourseManagementMavenWebApp/course.xhtml` to see the list of courses displayed by the `course.xhtml` JSF page.