

---

# Web development with Eclipse Europa, Part 1: The Java EE for Eclipse

How to use Eclipse Europa for Web development using Java technology, PHP, and Ruby

Skill Level: Intermediate

[Michael Galpin \(mike.sr@gmail.com\)](mailto:mike.sr@gmail.com)

Developer  
eBay

20 Nov 2007

It's a good time to be a Web developer. You've never had more choices in terms of technologies. There are so many great open source Web servers, databases, programming languages, and development frameworks. No matter what combination of technologies you prefer to work with, there is an integrated development environment (IDE) that can increase your productivity: Eclipse. In this tutorial, Part 1 of a three-part "[Web development with Eclipse Europa](#)" series on how to use Eclipse for Web development with Java™ technology, PHP, and Ruby, we'll see how the latest release of Eclipse — Europa — can be used to rapidly develop Java Web applications. We'll use Java Platform, Enterprise Edition 5 (Java EE) for Eclipse to build a Web application for tracking and calculating baseball statistics.

## Section 1. Before you start

Eclipse developers have worked hard to make Web developers' lives easier. The Europa release of Eclipse provides several tailored editions of Eclipse, including editions for Web developers.

### About this series

In this "[Web development with Eclipse Europa](#)" series, you'll see how no matter what your language of choice is, Eclipse is the platform for Web development. Its flexible plug-in system makes it easy to create versions of Eclipse customized for Web

development with Java technology, PHP, and Ruby. You'll see how different plug-ins give Eclipse unique capabilities for each language. You'll also see some of the common features that all Web developers can take advantage of. Throughout this series, we will be creating a sample baseball Web application, which will allow us to enter in game data for baseball players and calculate statistics for those players.

## About this tutorial

In this tutorial, Part 1 of a three-part series, we'll use the Eclipse Java EE version for Java Web development. We'll use Eclipse to connect to a database, create tables, and create test data in our database. We'll configure Eclipse to use a Web server, then use Eclipse's wizards to create a Web application, Web pages, and data access code. Finally, we'll use Eclipse to control our Web server, publish our application, and even debug it as it runs on the server.

## Prerequisites

A background in Java programming is essential for getting the most out of this tutorial. Familiarity with the Eclipse IDE is helpful, but not necessary.

## System requirements

You'll need the following:

### Eclipse Europa

This tutorial uses [Eclipse V3.3](#) (Europa).

### Java Development Kit (JDK)

This tutorial shows you how to develop Web applications using Java technology, so you'll need the Java Development Kit (JDK) 5.0 or higher. Download [V5.0](#) or [V6.0](#).

### Eclipse IDE for Java EE Developers

You'll also need the [Eclipse IDE for Java EE Developers](#).

### Java Runtime Environment (JRE)

To run Eclipse, you must have a [JRE](#).

### Apache Tomcat

The application uses a [Apache Tomcat](#) as its container.

### MySQL V5.0

The application uses [MySQL V5.0](#) as its database.

### Java Persistence API

You will also need the [Java Persistence API](#) and, in particular, the [OpenJPA](#) implementation.

---

## Section 2. Eclipse: Which edition?

If you check out the downloads section of Eclipse.org, you'll see several distributions of Eclipse available. We will use the Eclipse IDE for Java EE Developers.

### Java EE

The Eclipse IDE for Java EE Developers gives you exactly what you need for Java Web development: a Java compiler and debugger, support for Java application servers, a database client, wizards for Web applications, Web services, and Enterprise JavaBeans (EJBs), among others. Notable is a graphical editor for HTML files and Java Server Pages (JSPs.) Most of what's included was available in previous versions of Eclipse. Before the release of Europa, we needed to download what is now referred to as the Classic version of Eclipse and install additional plug-ins. This has all been simplified with the Europa release, and the Java EE package is perfectly tailored for Java Web development.

**Tip:** The Java EE package is feature-packed. To support all these features, we need to dedicate some resources on our development machines. I recommend you increase the memory usage for Europa by editing the eclipse.ini configuration file. This can be found in different places depending on the operating system. To increase the memory usage, edit `-vmargs`. The memory settings used during the development of the application in this tutorial are shown below.

#### Listing 1. eclipse.ini -vmargs

```
-Xms256m  
-Xmx512m  
-Xmn128m  
-XX:+UseParallelGC  
-XX:MaxPermSize=128m
```

Install the Java EE Edition of Europa, then configure a database and Web server. We review those steps next.

### Infrastructure

Eclipse makes Web development easier than ever, but there are certain pieces of infrastructure we still need. You will need a database for persisting and retrieving data. Eclipse offers great tools for working with databases, so all we'll need to do is create an empty database, or you can reuse one you already have. The MySQL command shown below creates an empty database.

#### Listing 2. Create database

```
mysql>
Newton:~ michael$ mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 9
Server version: 5.0.41 MySQL Community Server (GPL)

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> create database baseball;
Query OK, 1 row affected (0.04 sec)

mysql>
```

We completed one part of our infrastructure. Now we need a Web server. We can use any Java Web container, including an application server, such as Apache Geronimo or IBM® WebSphere®. We will keep things on the light side and just use a Web container — in this case, Apache Tomcat V6.0. You don't need to do anything special to Tomcat. Now that our infrastructure is in place, let's move on to using Eclipse to develop our Web application.

---

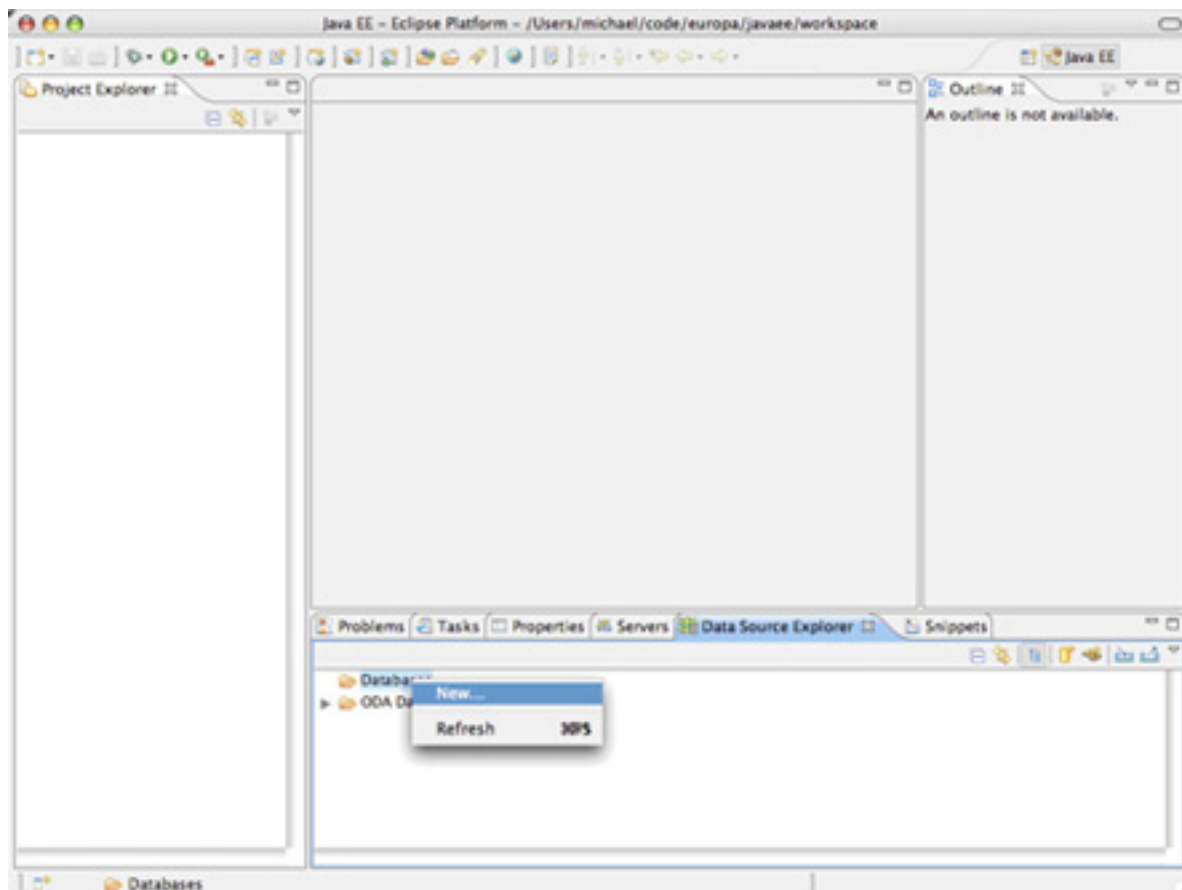
## Section 3. The Baseball application: Working with databases

To demonstrate the power of Eclipse, we will build a Java Web application for tracking baseball statistics. It's a pretty simple application, allowing us to enter in game data for baseball players and calculate statistics for those players. First thing we'll need to do is create a database schema.

### Creating the schema

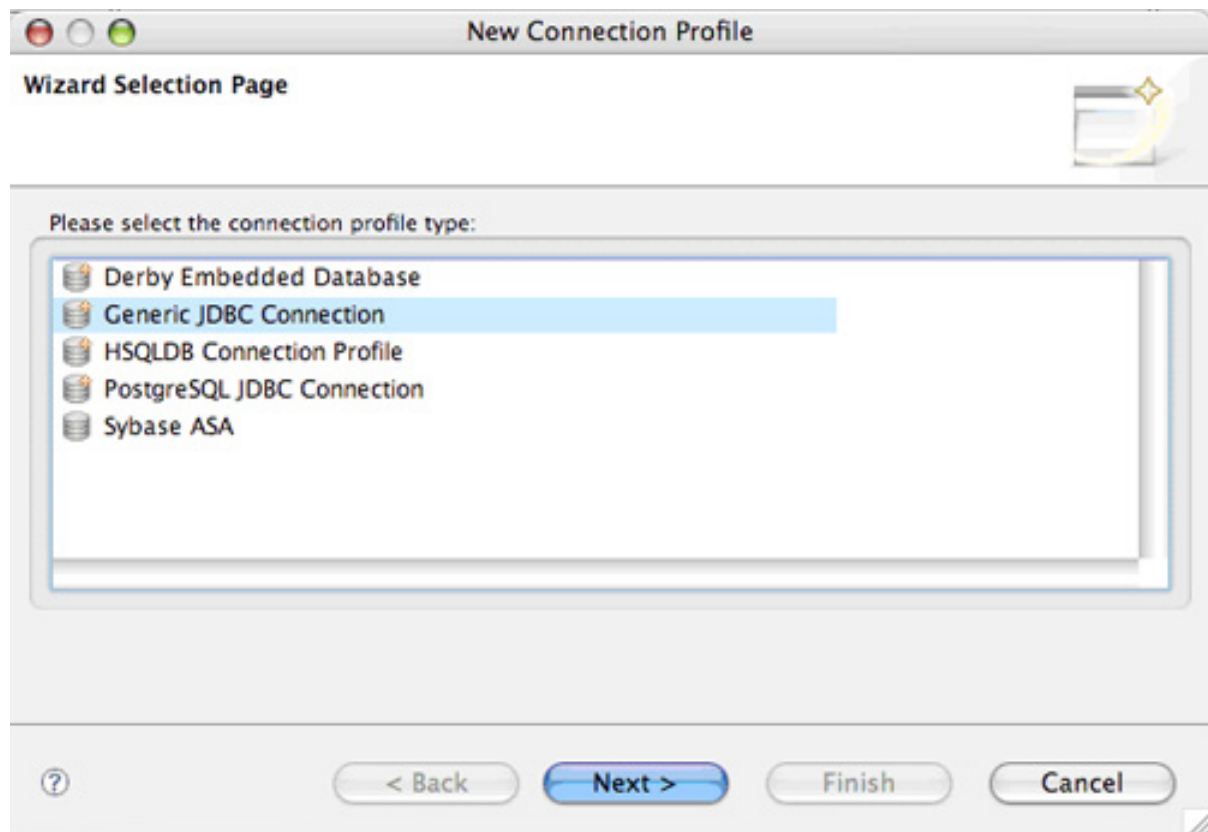
When you open up Eclipse, you'll want to open up the Data Source Explorer window. Right-click on the **Databases** folder and select **New**, as shown below.

#### Figure 1. Data Source Explorer



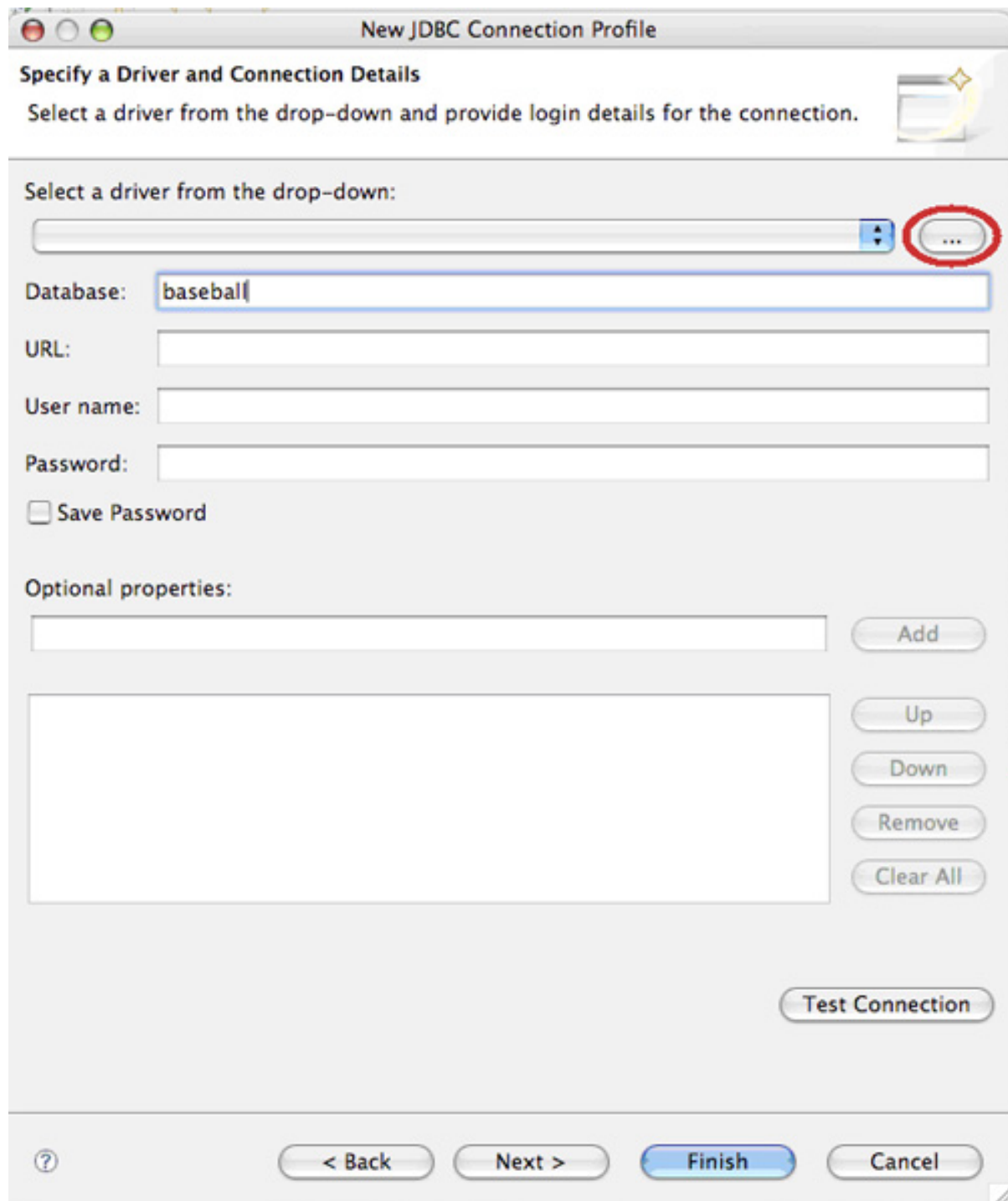
Eclipse will ask you what kind of connection it should use for the database. Select **Generic JDBC Connection**.

**Figure 2. Connection type**



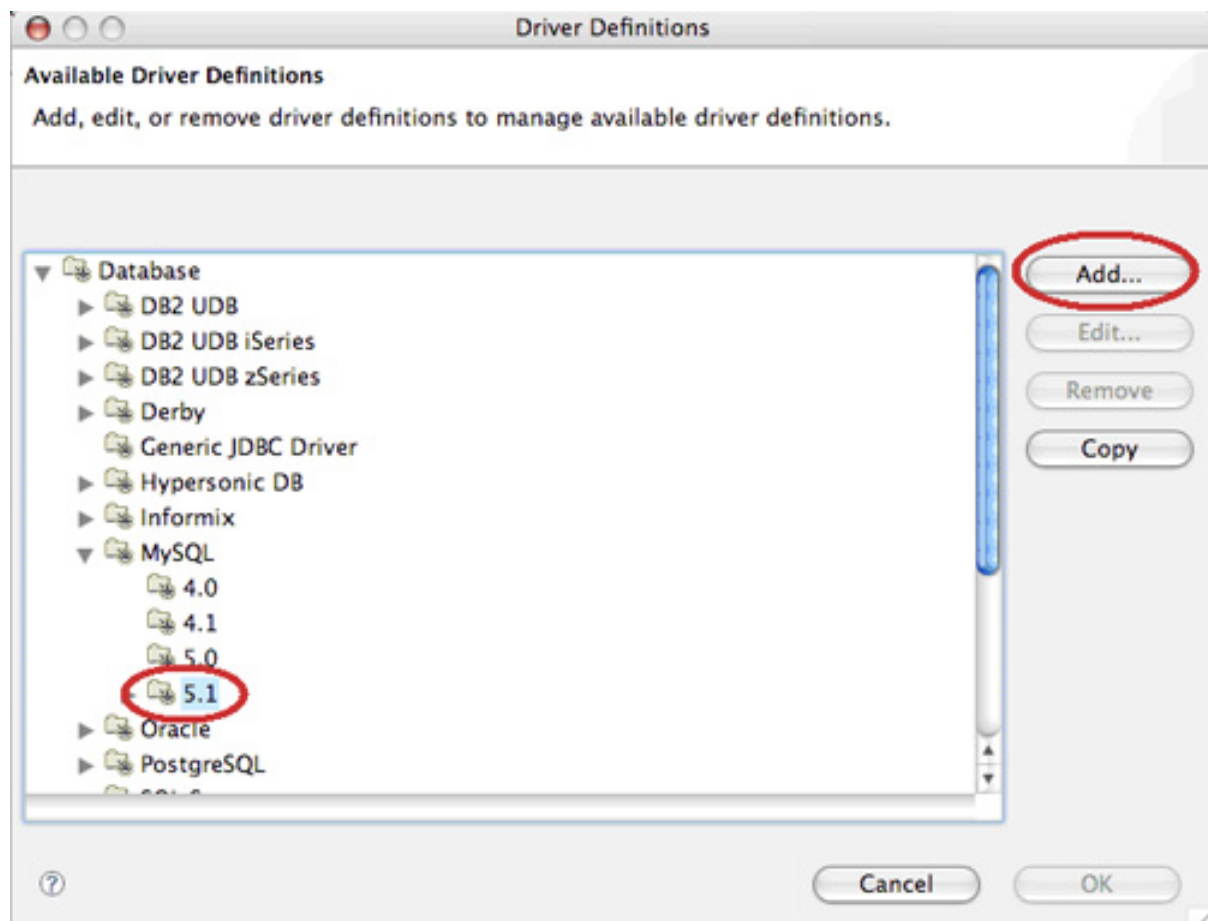
Now we configure our connection. We need to tell it what database to connect to. We created a baseball database in Listing 2, so we'll use that.

**Figure 3. JDBC Connection Profile**



We need to create a new driver by clicking on the **Browse** button, as shown above. This will bring up a list of various database types.

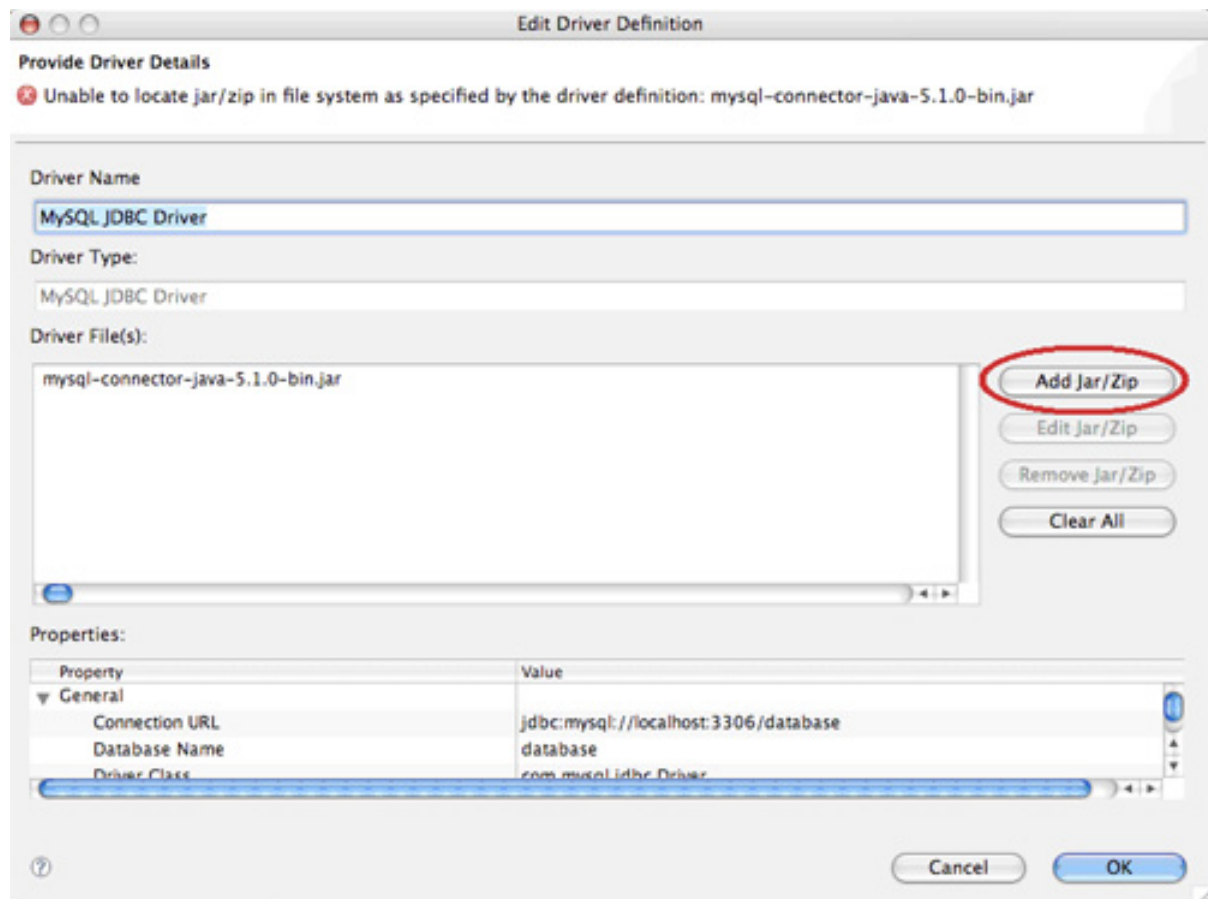
**Figure 4. Database types**



Scroll down to find MySQL V5.1 (or whatever type is appropriate for your database). Click **Add** to bring up the driver definition dialog.

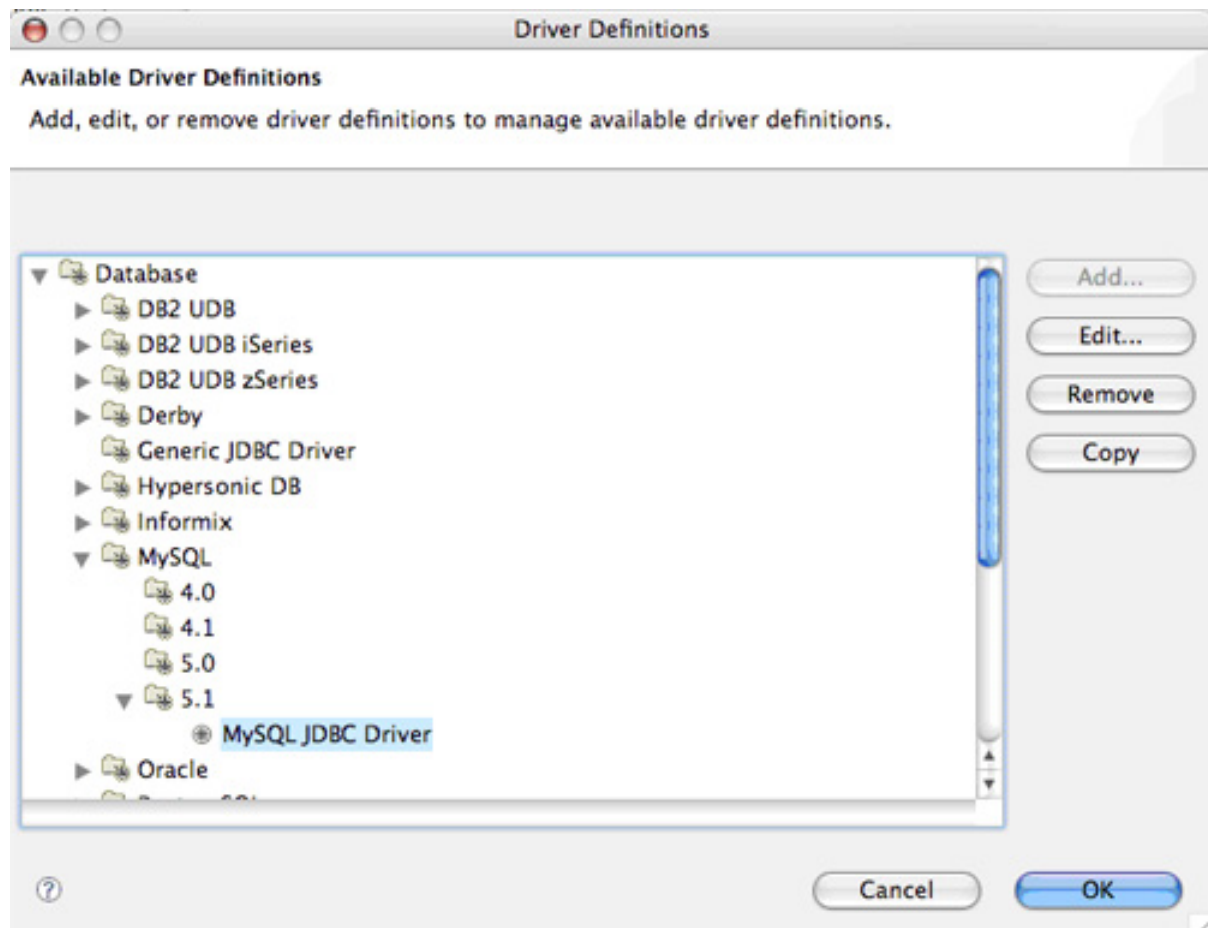
**Figure 5. JDBC driver definition**





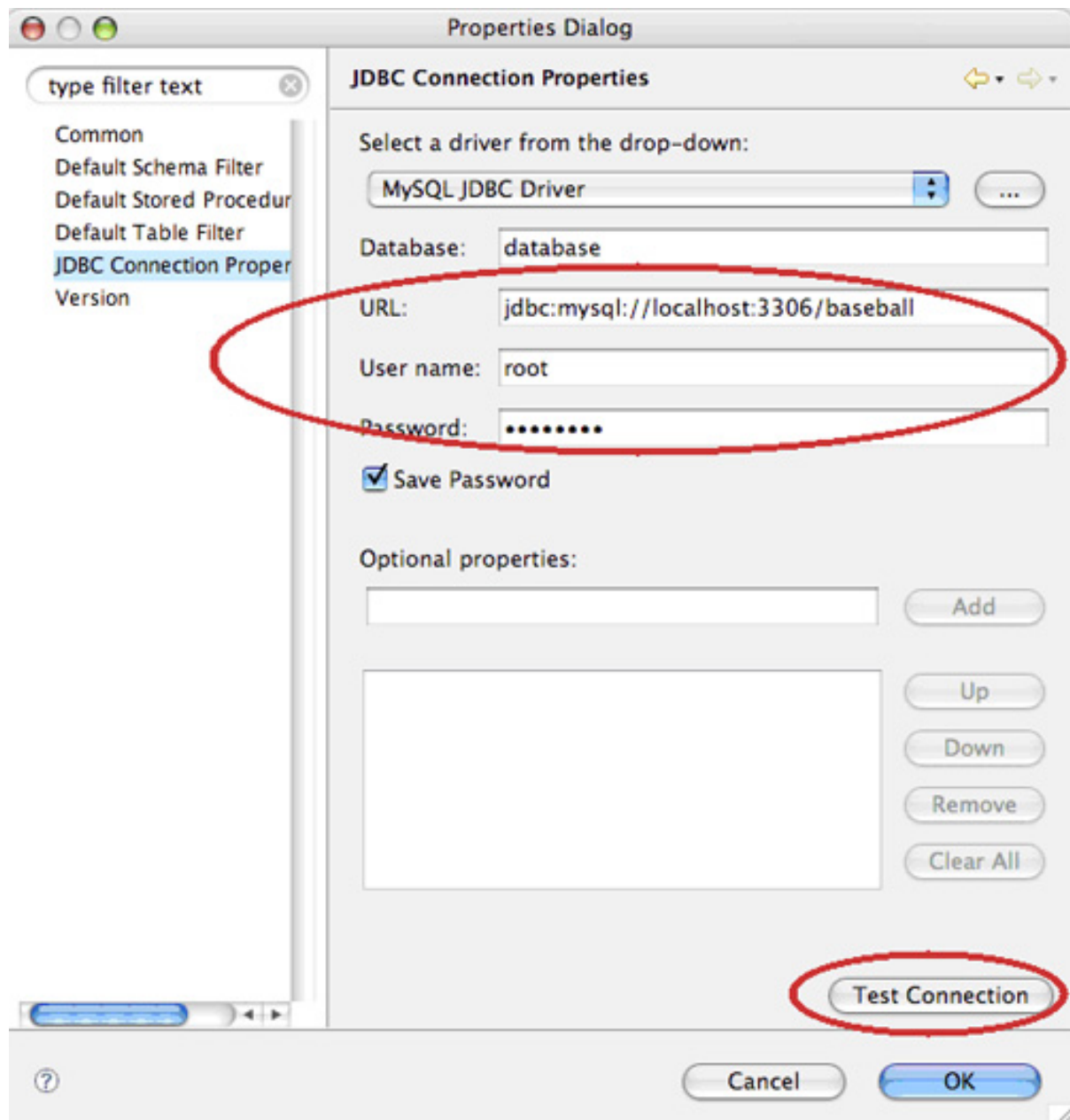
Click **Add Jar/Zip**. This will bring up a simple file browser, where you can navigate to the JAR file for the MySQL driver you downloaded. Once you've selected your JDBC JAR, the list of driver definitions should be refreshed.

**Figure 6. Update driver definitions list**



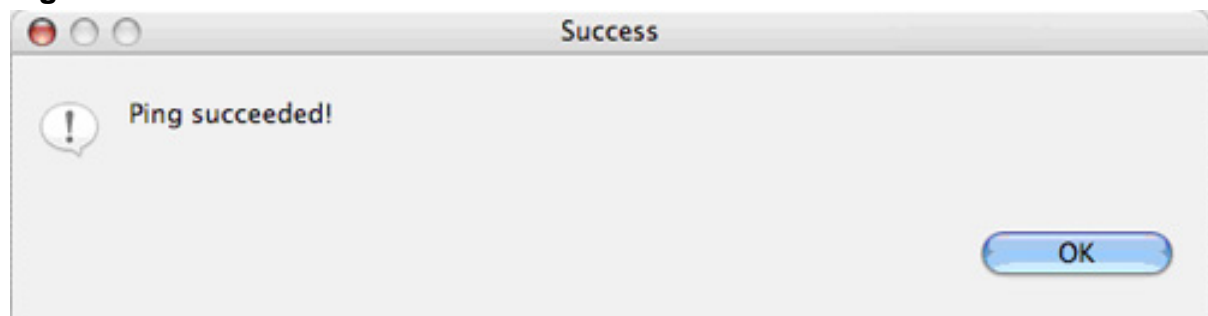
Now you can select the driver definition you just created and click **OK**. This will bring up your connection properties. You may need to edit this to reflect the user name and password you will use.

**Figure 7. JDBC Connection Properties**



Once you have put in the appropriate connection info, you can test your connection. It should bring up a success dialog.

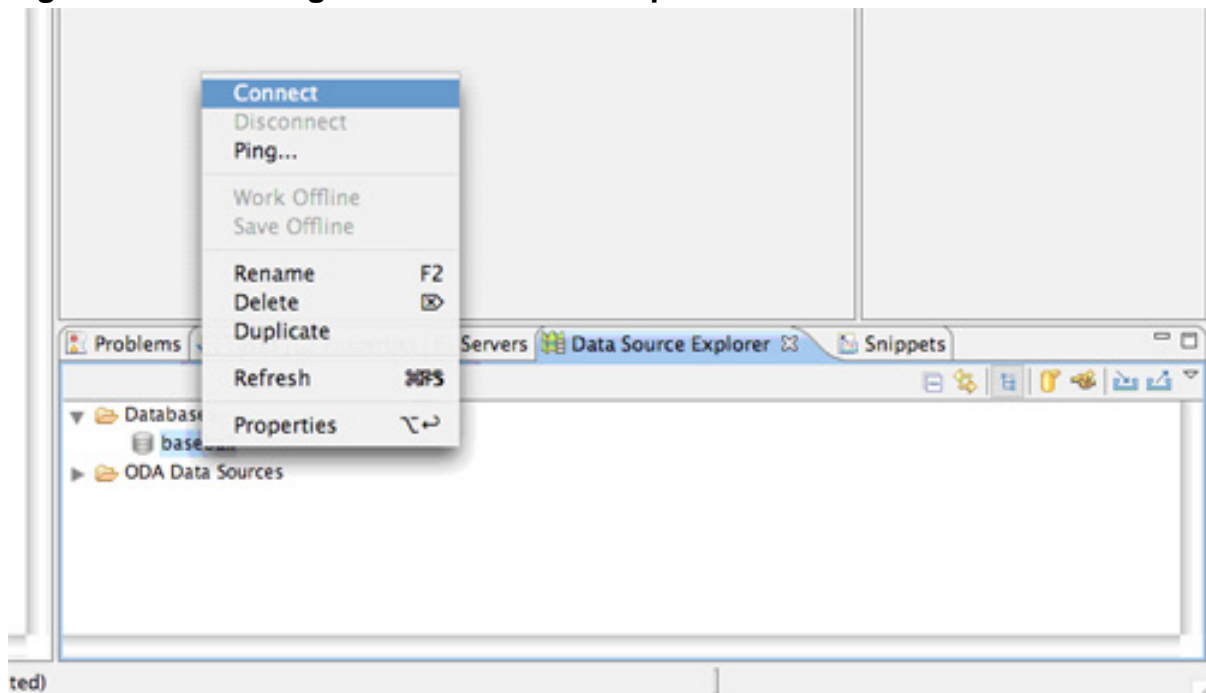
**Figure 8. Test connection**



If you didn't get a success message like the one shown above, you need to tweak

your settings. Once your connection test passes, you should be able to start a connection to the database in the Data Source Explorer.

**Figure 9. Connecting from Data Source Explorer**

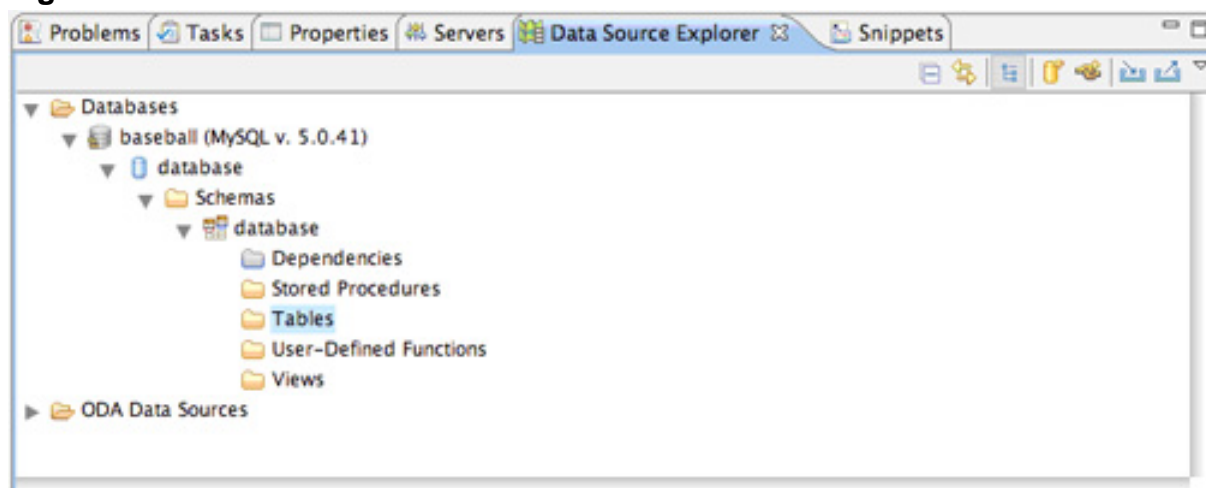


Now you have created a connection to your database. It's time to create some tables for the data we're going to store.

## Creating the tables

To create our tables, we'll start by drilling to the list of tables in the Data Source Explorer.

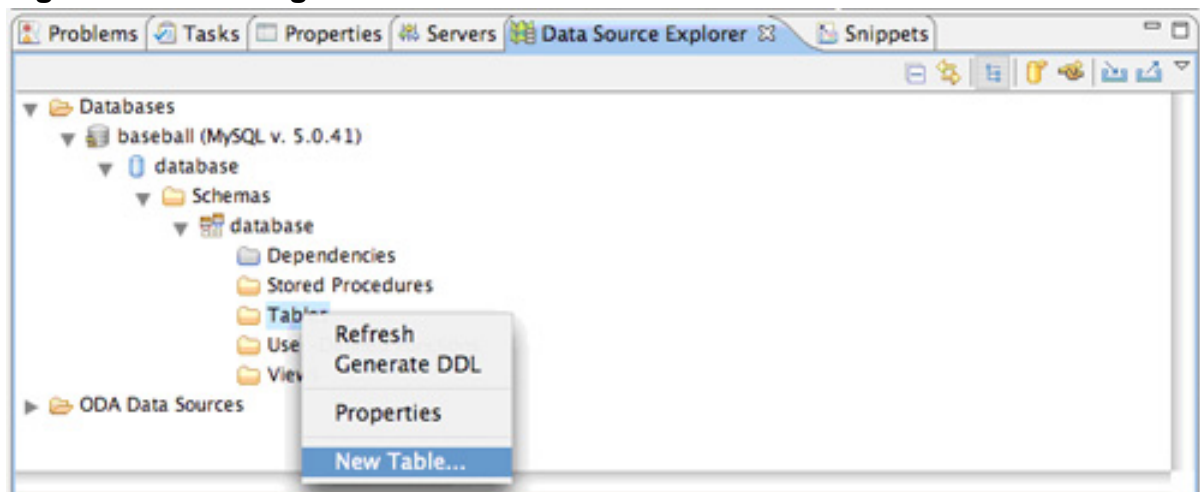
**Figure 10. Database tables listed**



There are no tables yet, since we haven't created any. If you reused an existing database, you should see list of tables. Either way, right-click on the **Tables** folder

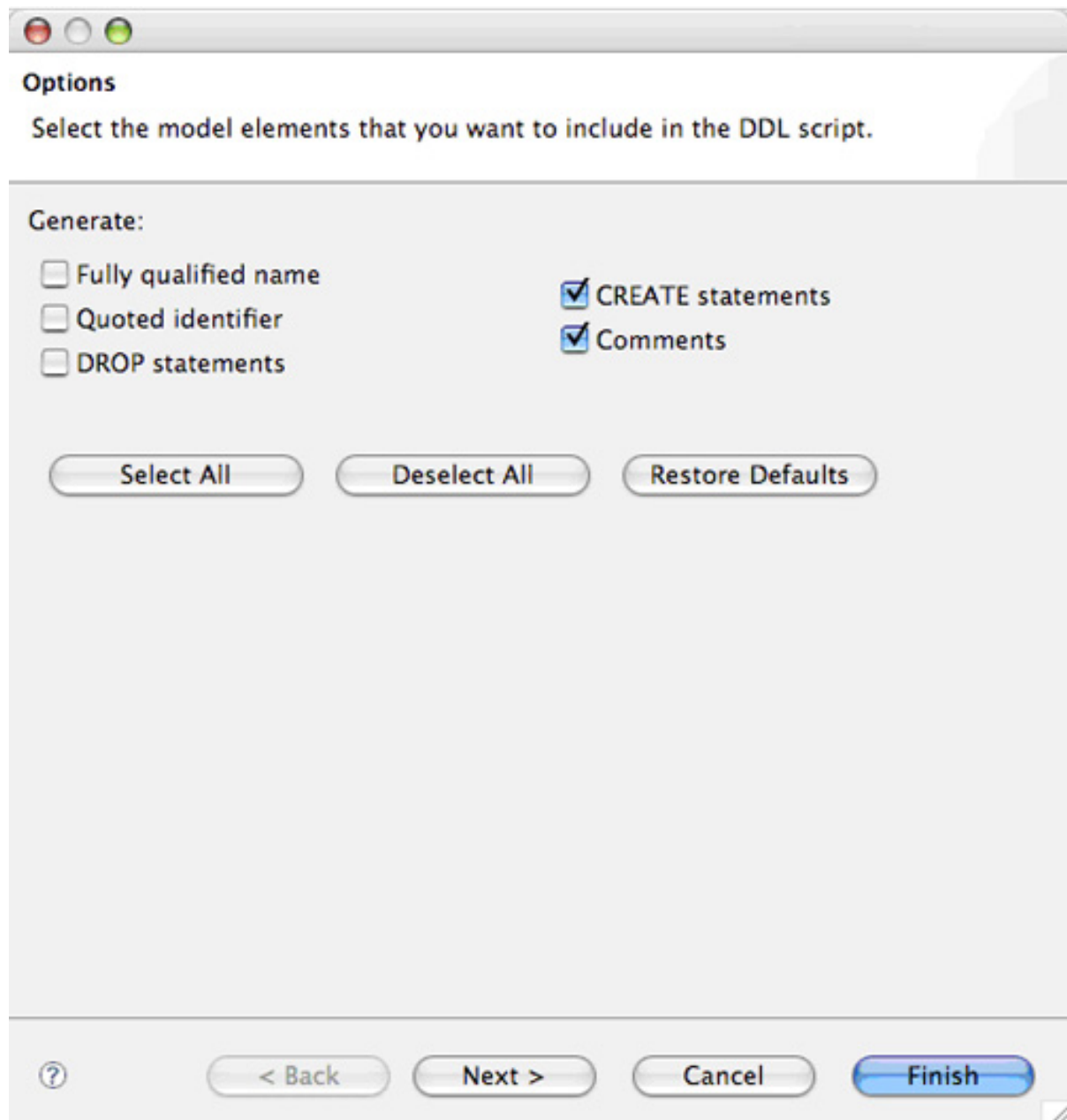
and select **New Table**.

**Figure 11. Selecting a new table**



This will bring up a list of create options.

**Figure 12. Create table options**



This will bring up the New Table dialog.

**Figure 13. New table**

**Columns**  
Set the columns that you want to include this table.

Table Name

Name	DataType	Nullable	Default Value
Id	INT	false	

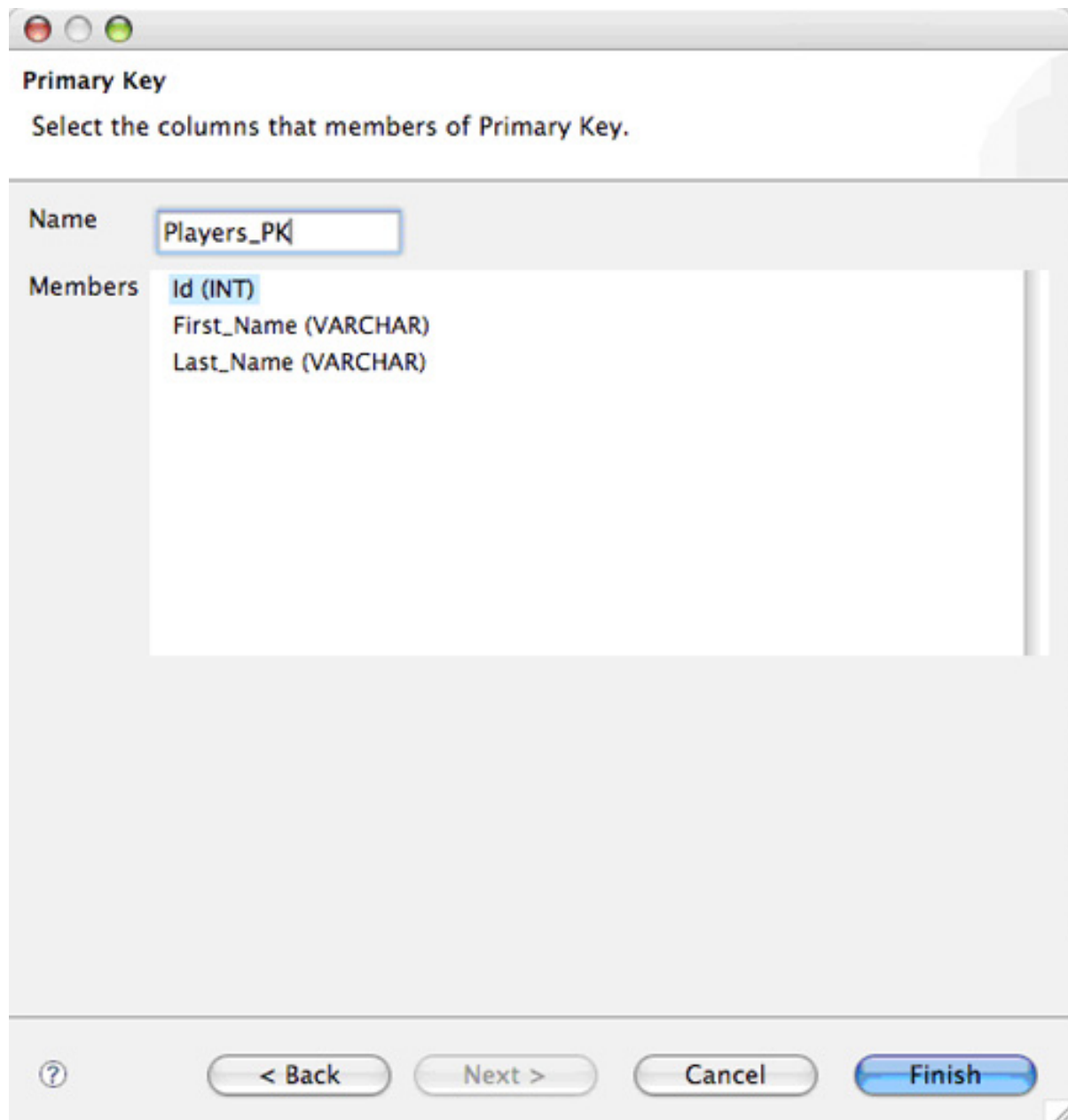
**Column Detail**

Name  DataType

☐ Nullable Default Value

We'll create a table called `Players` and give it three columns: `Id`, `First_Name`, and `Last_Name`. We'll make all three columns required, and we'll make `Id` the primary key.

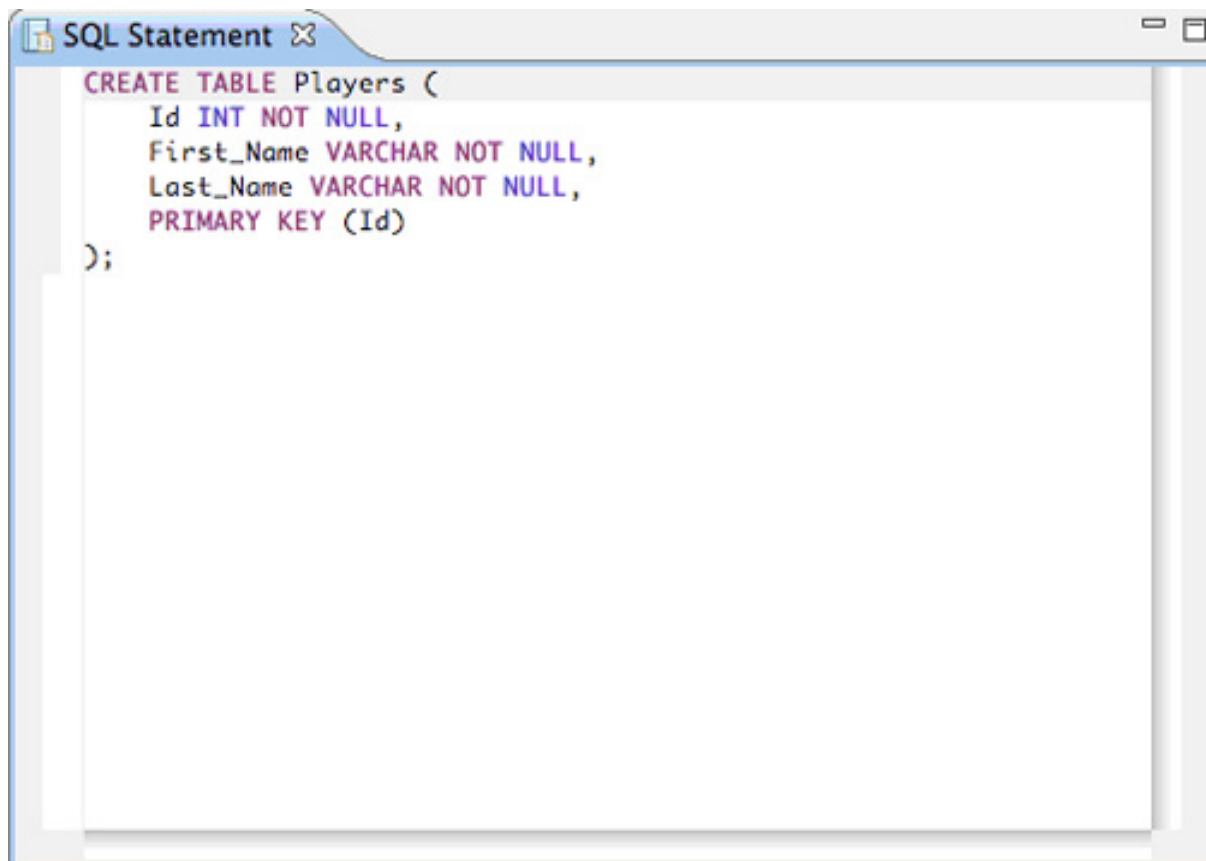
**Figure 14. Primary key definition**



Click **Finish**. This doesn't execute any SQL DDL against the database. Instead, it generates the SQL statement for you.

**Figure 15. Generated SQL statement**





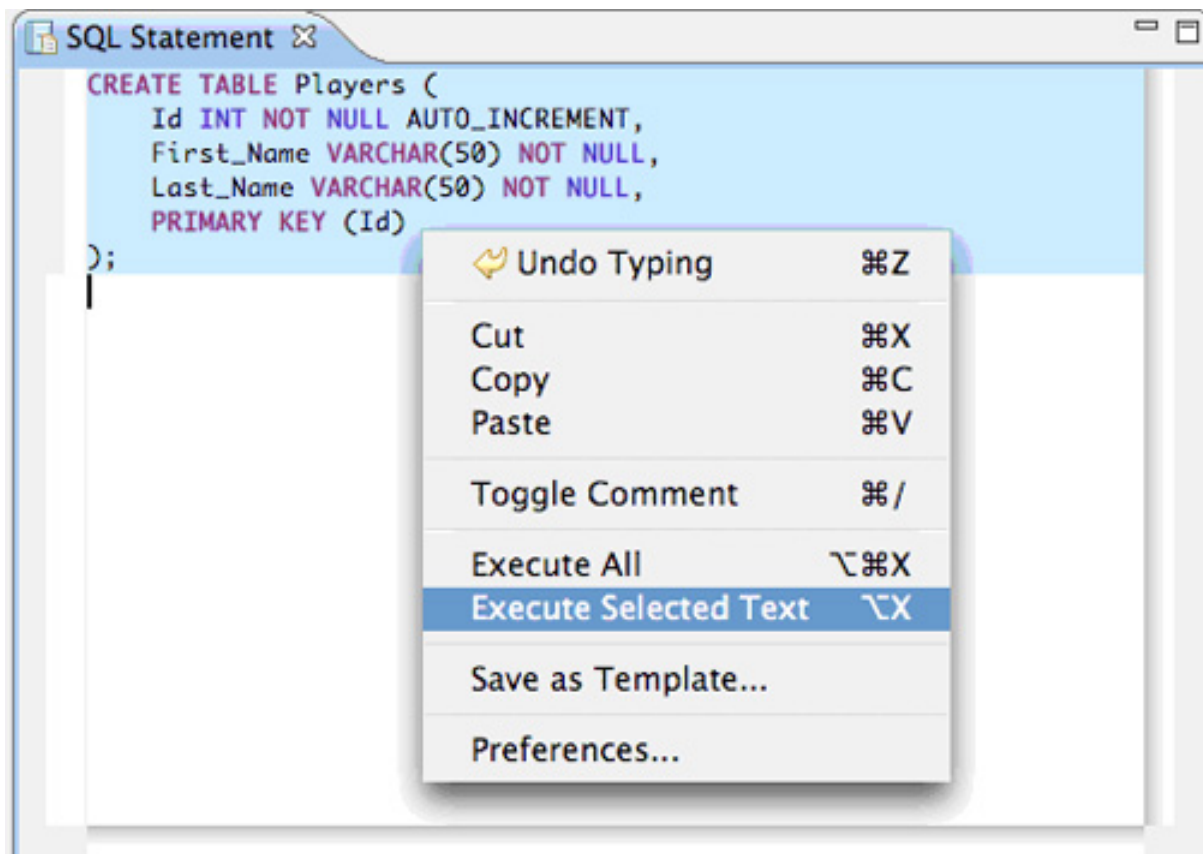
The nice thing here is that you can tweak the SQL yourself before executing it. We'll do exactly that. We'll make the `Id` column an auto-increment column, and we'll put a 50-character limit on both the first- and last-name columns. The SQL will look like Listing 3.

### Listing 3. Players table SQL

```
CREATE TABLE Players (  
    Id INT NOT NULL AUTO_INCREMENT,  
    First_Name VARCHAR(50) NOT NULL,  
    Last_Name VARCHAR(50) NOT NULL,  
    PRIMARY KEY (Id)  
);
```

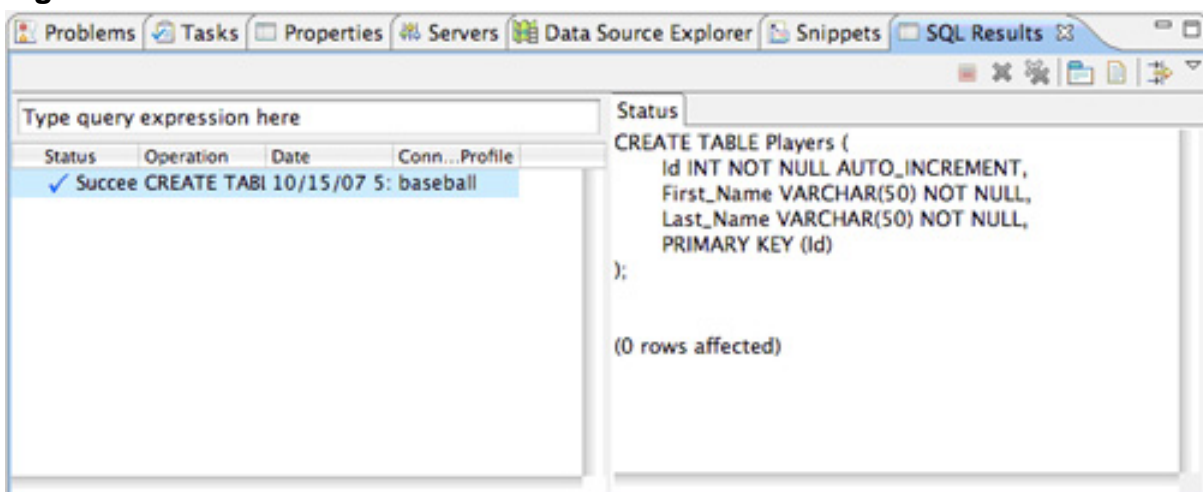
To execute the SQL, highlight and right-click **Execute Selected Text**.

### Figure 16. Execute SQL



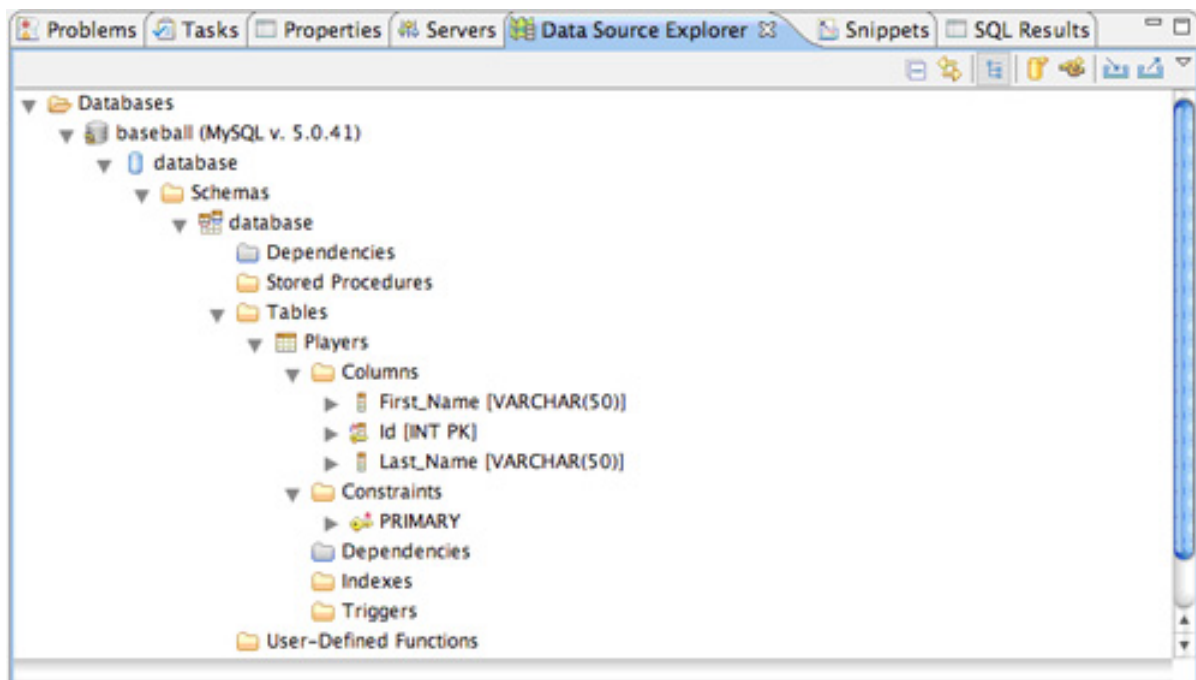
Executing the SQL should produce a success confirmation message in the SQL Results window.

**Figure 17. SQL execution success confirmation**



You've created the Players table from inside Eclipse. You can also verify the fruits of your labor by opening up the Data Source Explorer.

**Figure 18. Viewing Players table from Data Source Explorer**



We just need one more table, which will hold game data for our baseball application. You can create the same way as above. The final SQL statement you'll want to execute is shown in Listing 4.

#### Listing 4. Games table SQL

```
CREATE TABLE Games (
    Id INT NOT NULL AUTO_INCREMENT,
    Player_Id INT DEFAULT '' NOT NULL,
    AB INT DEFAULT 0 NOT NULL,
    H INT DEFAULT 0 NOT NULL,
    2B INT DEFAULT 0 NOT NULL,
    3B INT DEFAULT 0 NOT NULL,
    HR INT DEFAULT 0 NOT NULL,
    BB INT DEFAULT 0 NOT NULL,
    R INT DEFAULT 0 NOT NULL,
    RBI INT DEFAULT 0 NOT NULL,
    PRIMARY KEY (Id)
);

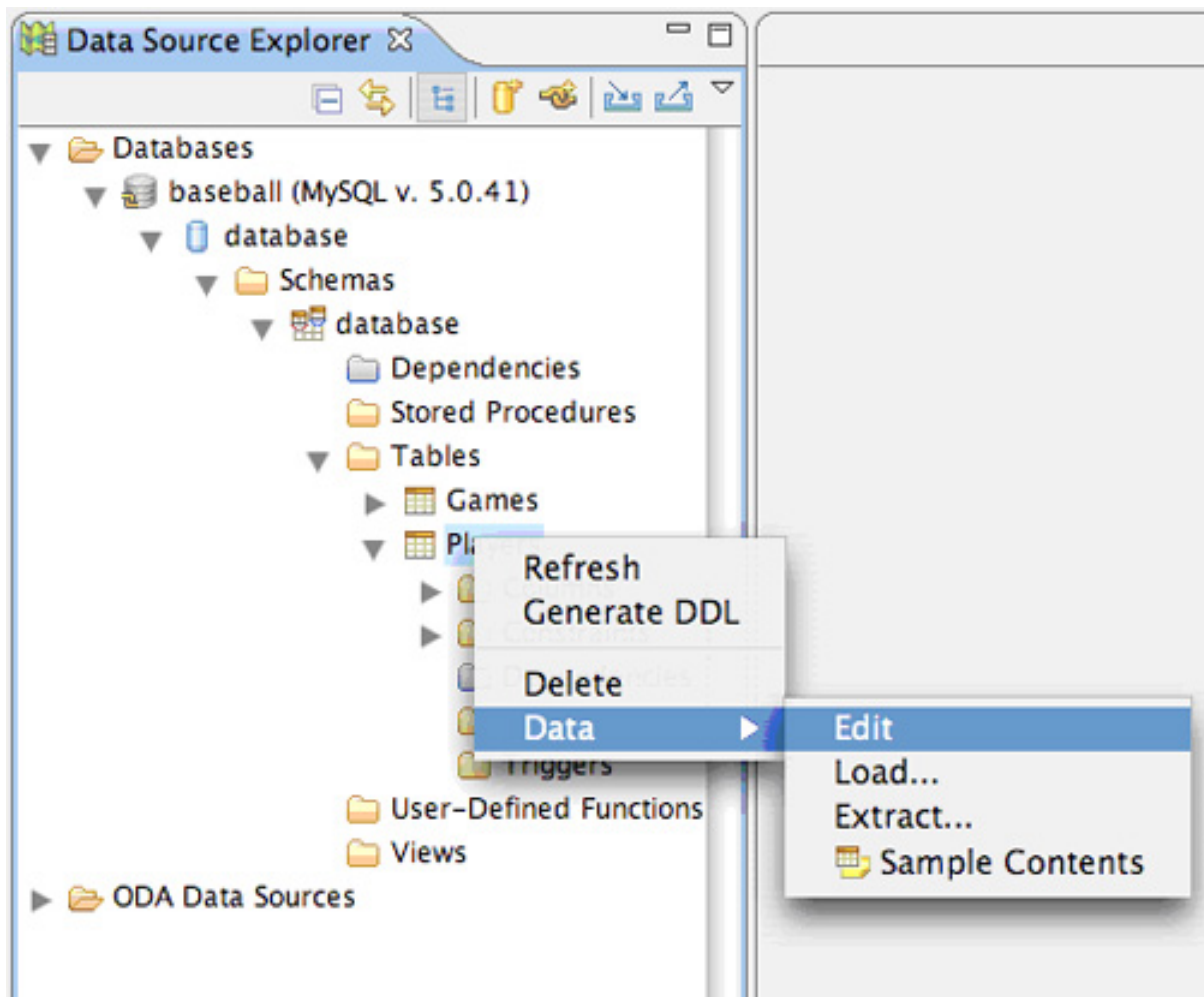
CREATE INDEX Player_Id ON Games (Player_Id ASC);
```

Once we've executed that SQL statement, we will have created both tables we need for the application. While we're working with the database, let's go ahead and enter in some test data.

## Entering in test data

Creating tables is pretty easy with Eclipse. Managing the data in those tables is even easier. You can start adding data by right-clicking on the name of the table in the Data Source Explorer and selecting **Data > Edit**.

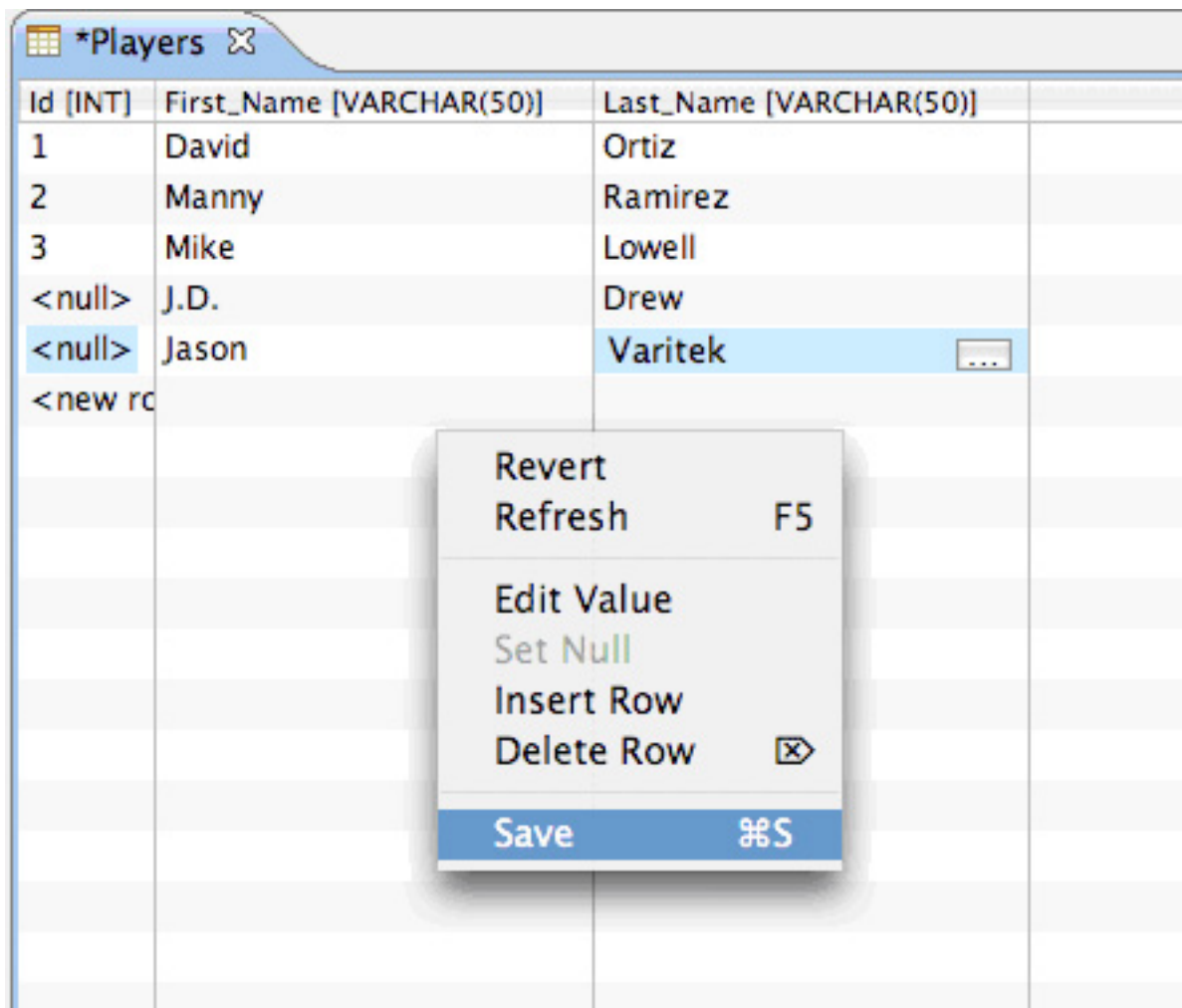
#### Figure 19. Adding data to Players table



This brings up a simple graphical interface for entering in the tabular data.

**Figure 20. Entering data**





Id [INT]	First_Name [VARCHAR(50)]	Last_Name [VARCHAR(50)]
1	David	Ortiz
2	Manny	Ramirez
3	Mike	Lowell
<null>	J.D.	Drew
<null>	Jason	Varitek
<new row>		

Revert

Refresh F5

Edit Value

Set Null

Insert Row

Delete Row

Save ⌘S

In the SQL results window, you'll see the actual SQL being executed and the return messages from the database. You can right-click in the Players window and select **Refresh**. This will re-query the data from the database. You'll see the IDs of the rows, as these were created by the database. Now we can do the same thing for the Games table. Create as much or as little test data as you want. When you're done, we're ready to set up Eclipse to use our Web server.

---

## Section 4. Working with Web servers

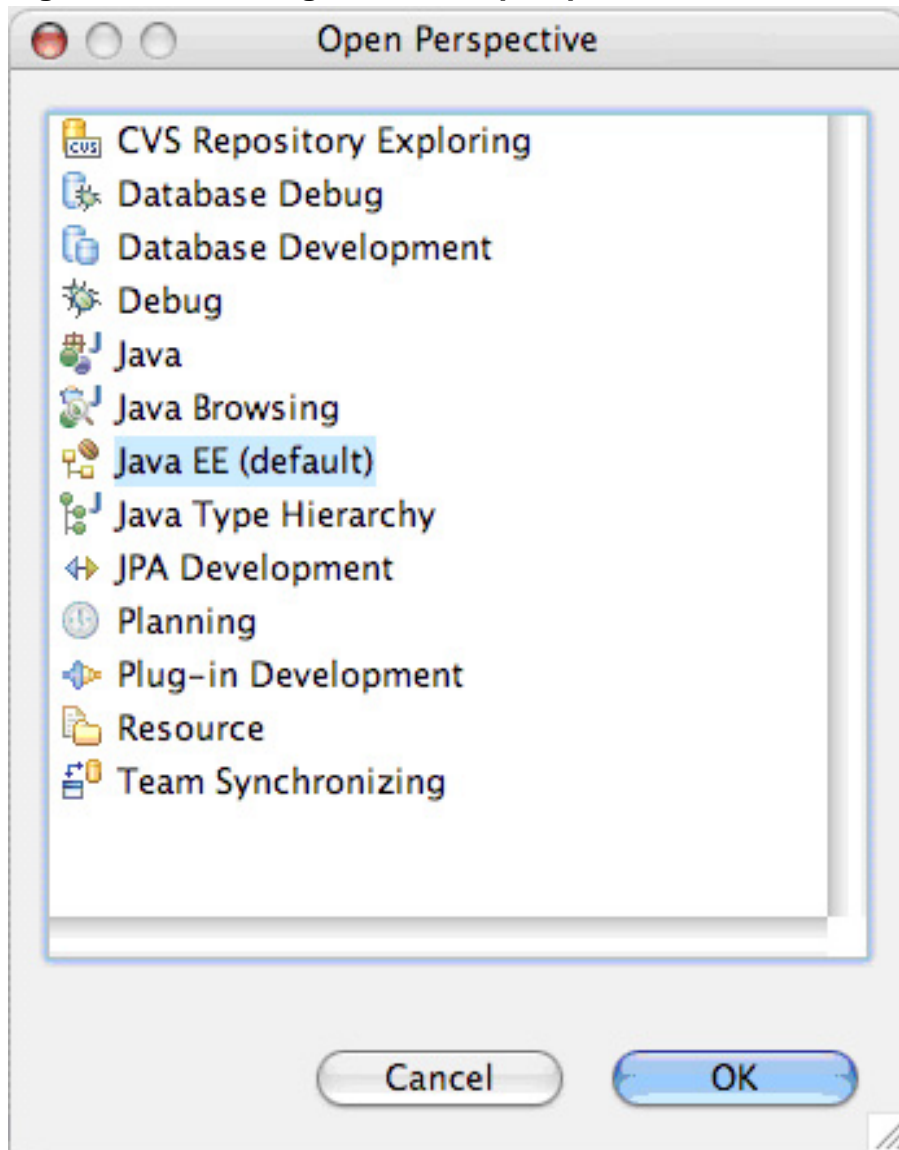
In this section, we look at basic Eclipse concepts and perspectives before setting up Eclipse to use our Web server. Once that's done, we will begin the process of actually writing the application.

### Setting up Tomcat with Eclipse

One of the key things with Eclipse is its notion of perspectives. Different

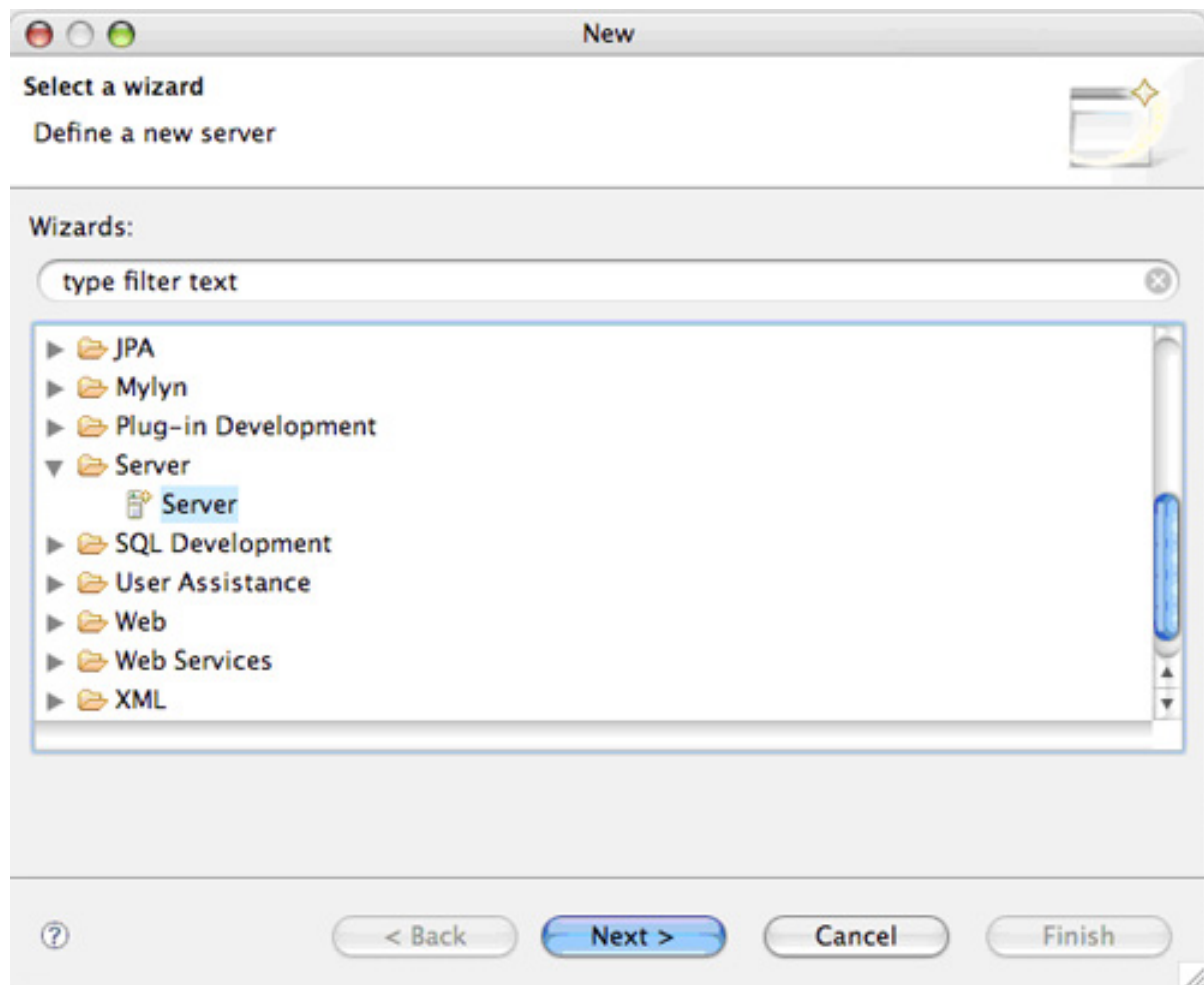
perspectives give you groups of windows and commands tailored for specific types of development. Let's switch to the Java EE perspective by selecting **Window > Open Perspective > Other**, then selecting Java EE.

**Figure 22. Switching to Java EE perspective**



Now select **File > New > Other** and pick Server from the list.

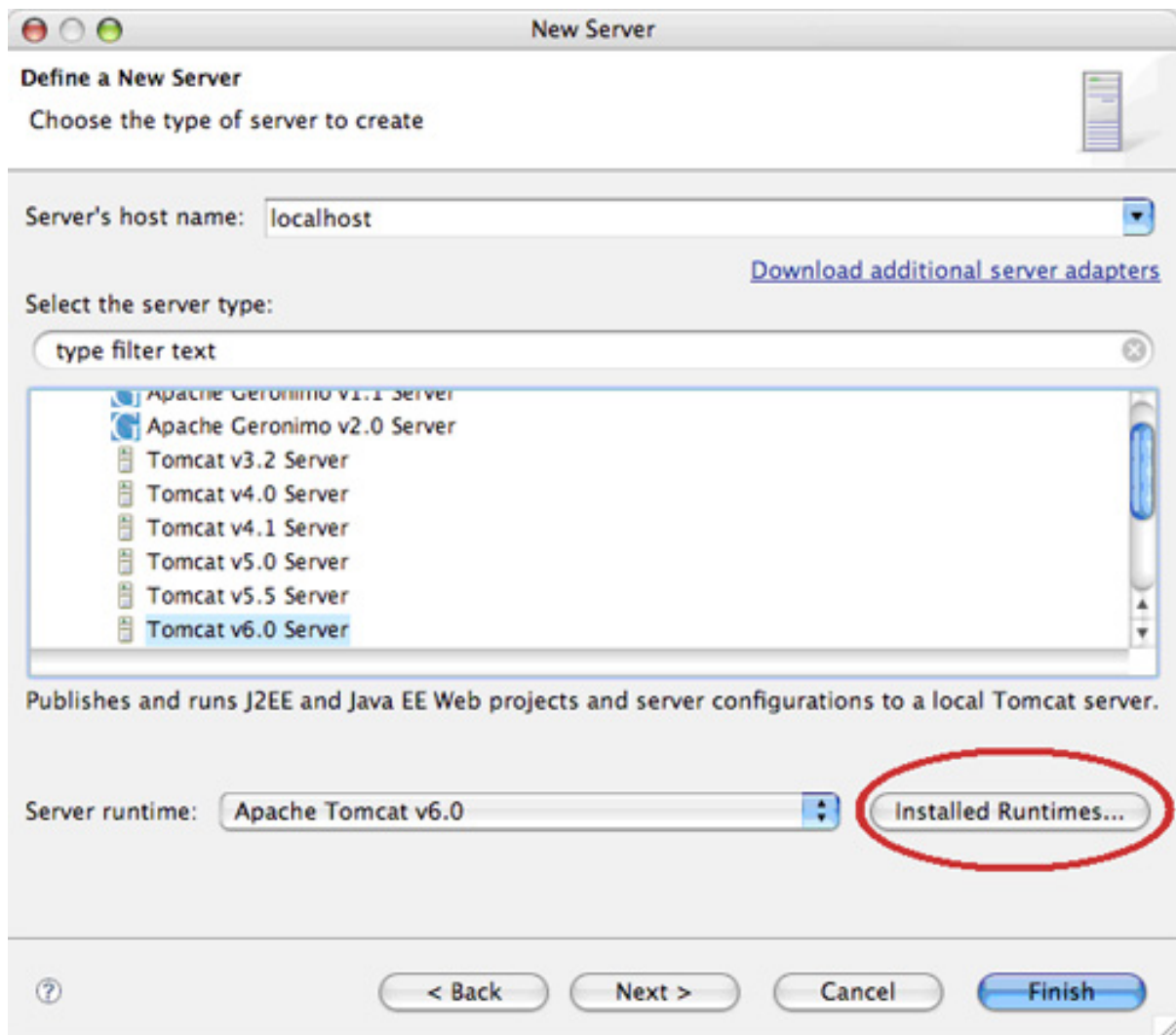
**Figure 23. New server**



Click **Next** and select **Tomcat v6.0 Server** for the server definition. You'll need to select a run time.

**Figure 24. Select server run time**



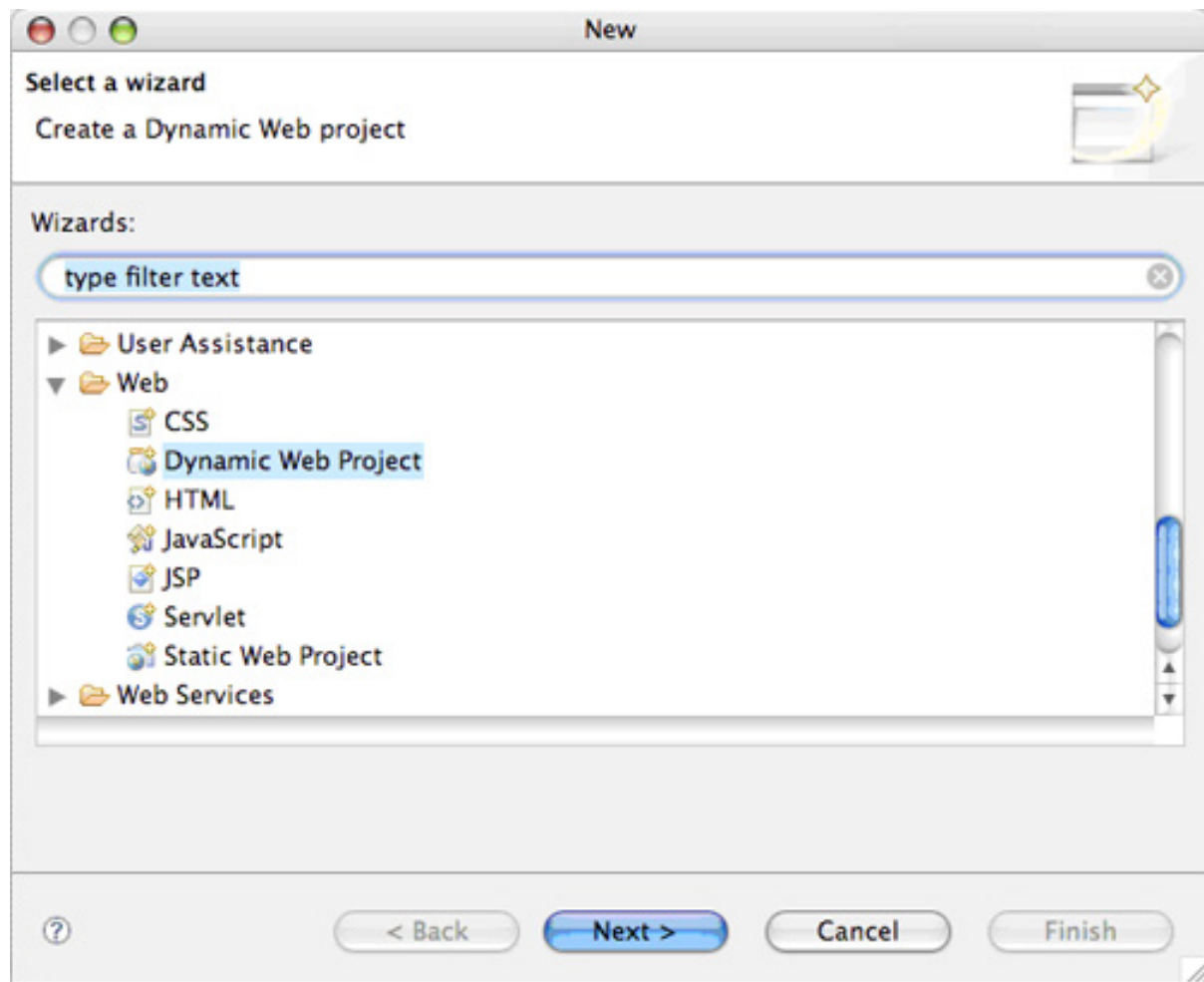


Navigate to the Tomcat installation directory in the file browser. Once Eclipse knows where your Tomcat directory is, click **Finish**. Now that we've set up all the infrastructure for the application, it's time to start writing some code.

## Creating the Web applications

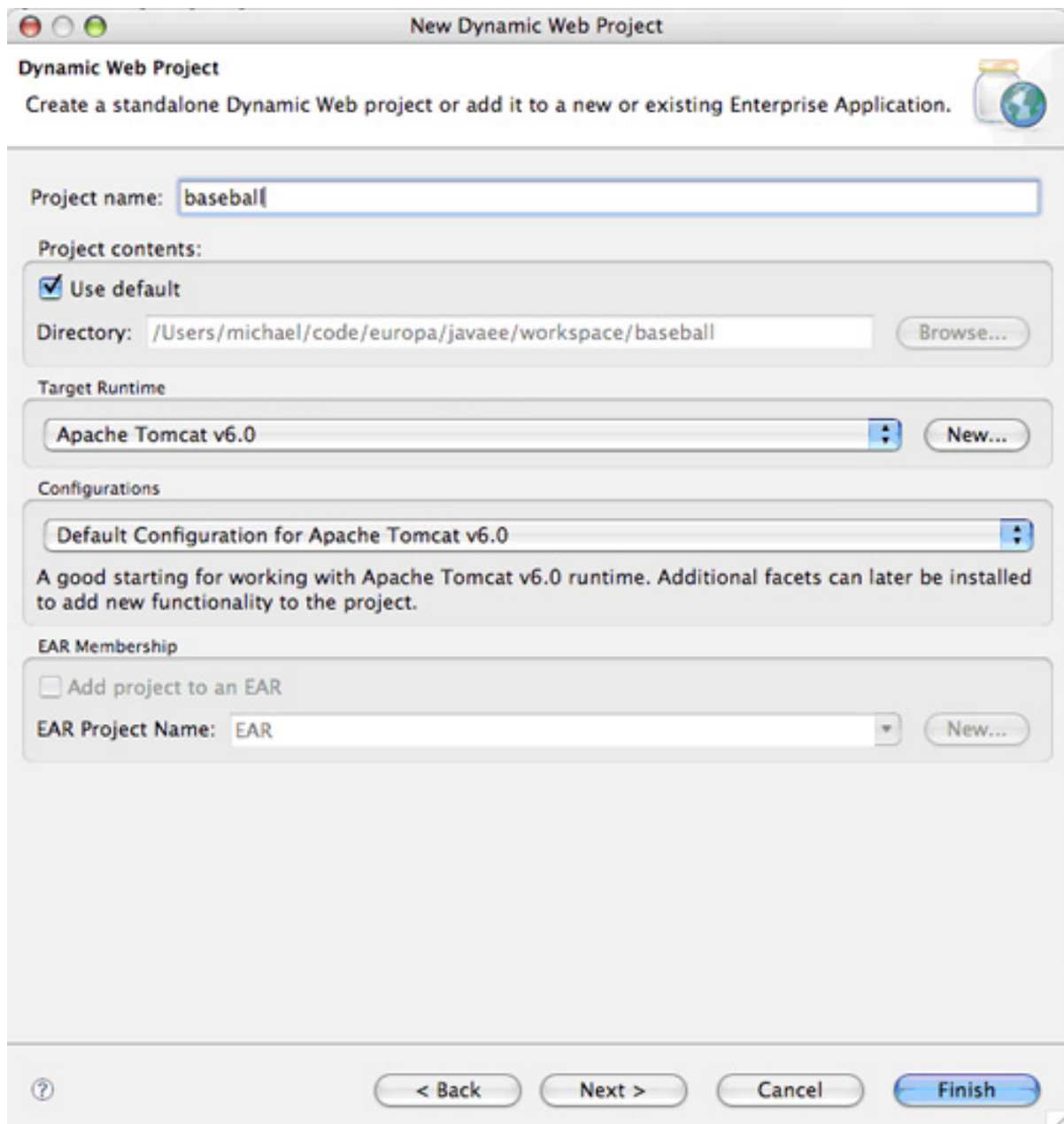
Time to get started with the development of this Web application. Here's where Eclipse is really going to help us. Let's create a new Web application by selecting **File > New > Other** and picking Dynamic Web Application.

**Figure 25. New Dynamic Web application**



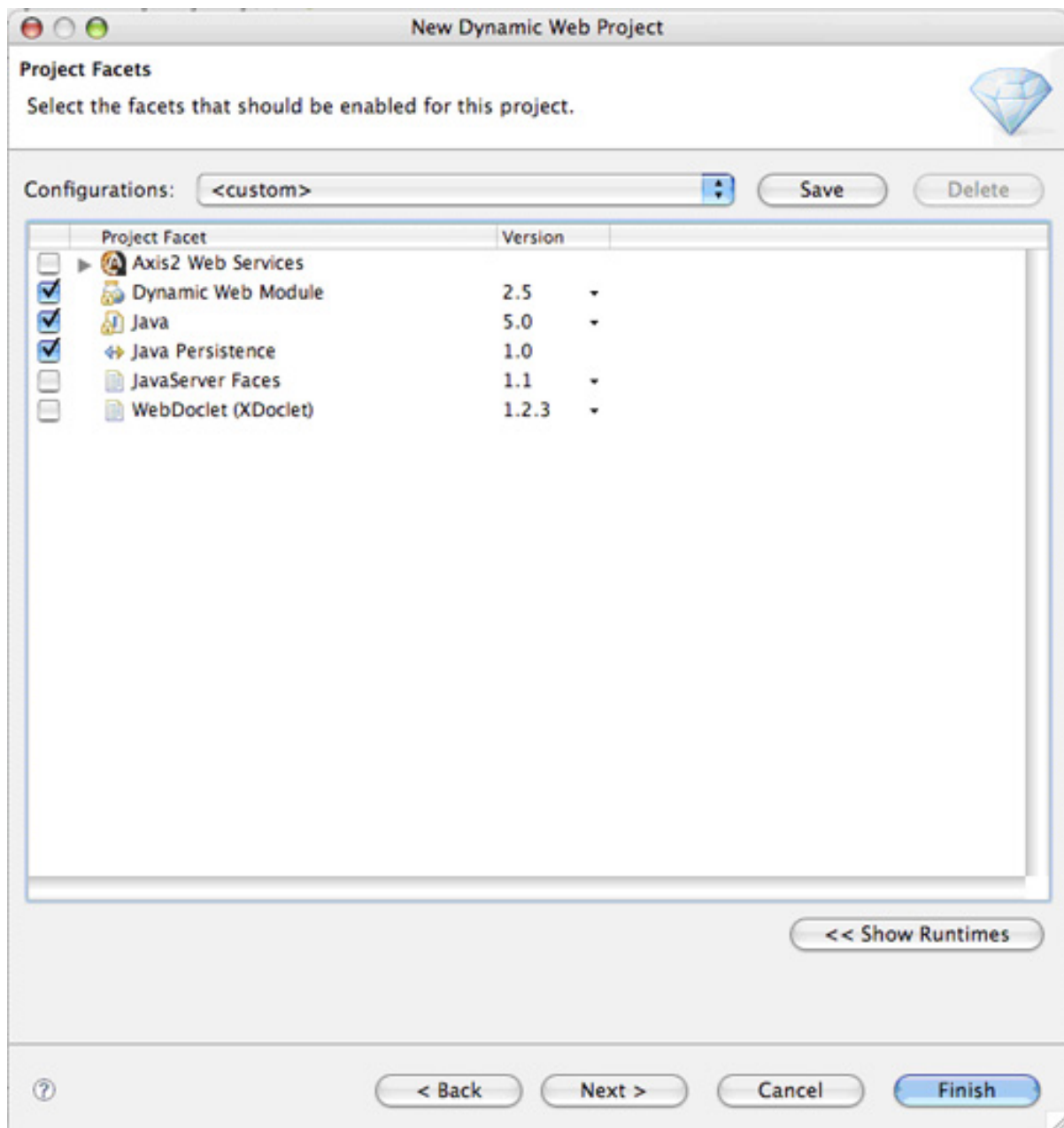
Next, we fill in the name of our project — we'll call it baseball. We'll also set the target run time as the Tomcat V6.0 server we just created.

**Figure 26. Project information**



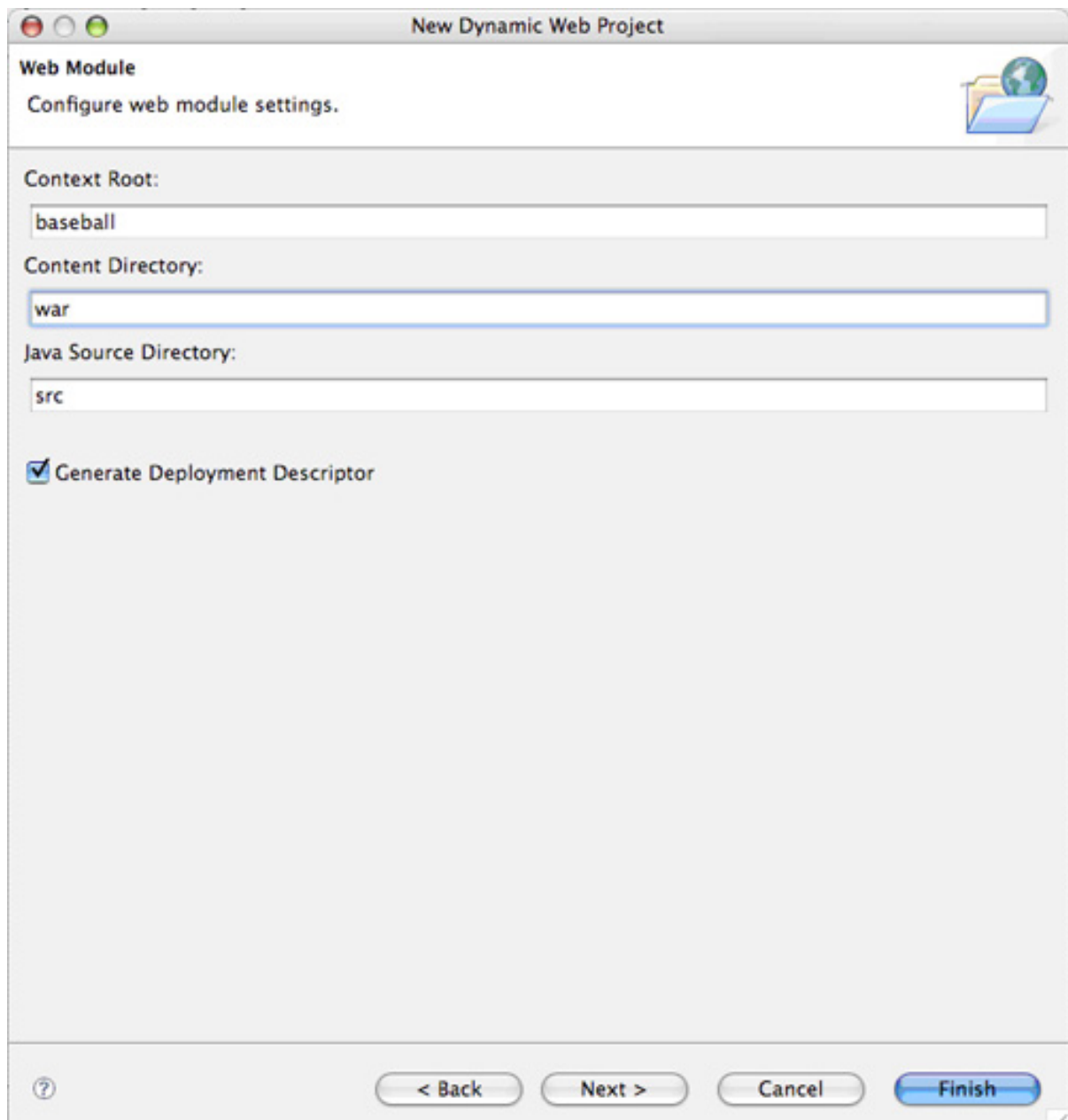
When you're done, it's time to start developing the domain model of the application. Click **Next** to bring up the Project Facets. We're going to use the Java Persistence API for managing data access (more on that in the next section), so make sure that is selected.

**Figure 27. Project facets**



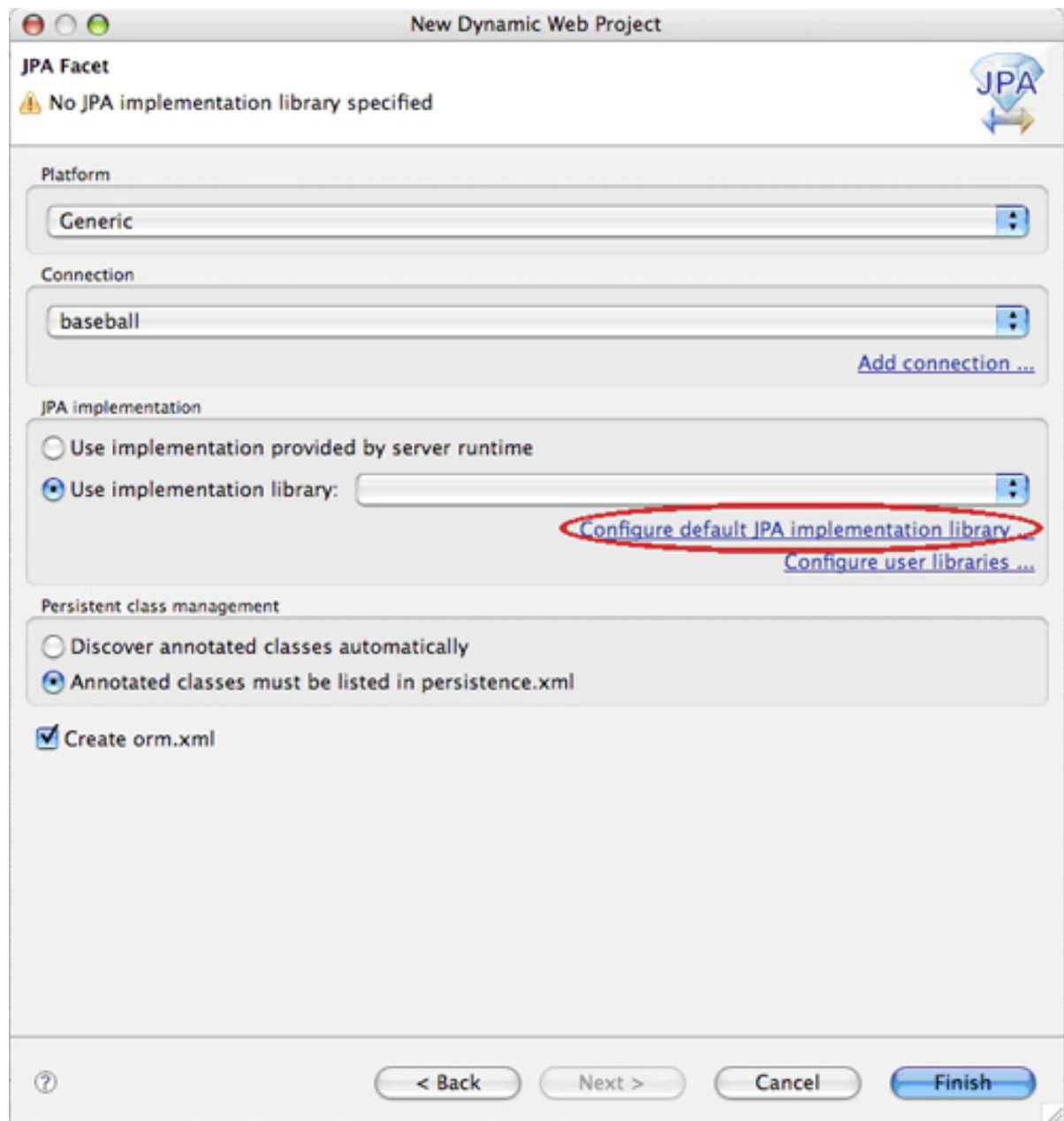
Click **Next**. This will bring up the Web module metadata. You can use the defaults here, but I like to call the Web module directory "war" as a reminder to what is being built.

**Figure 28. Web module information**



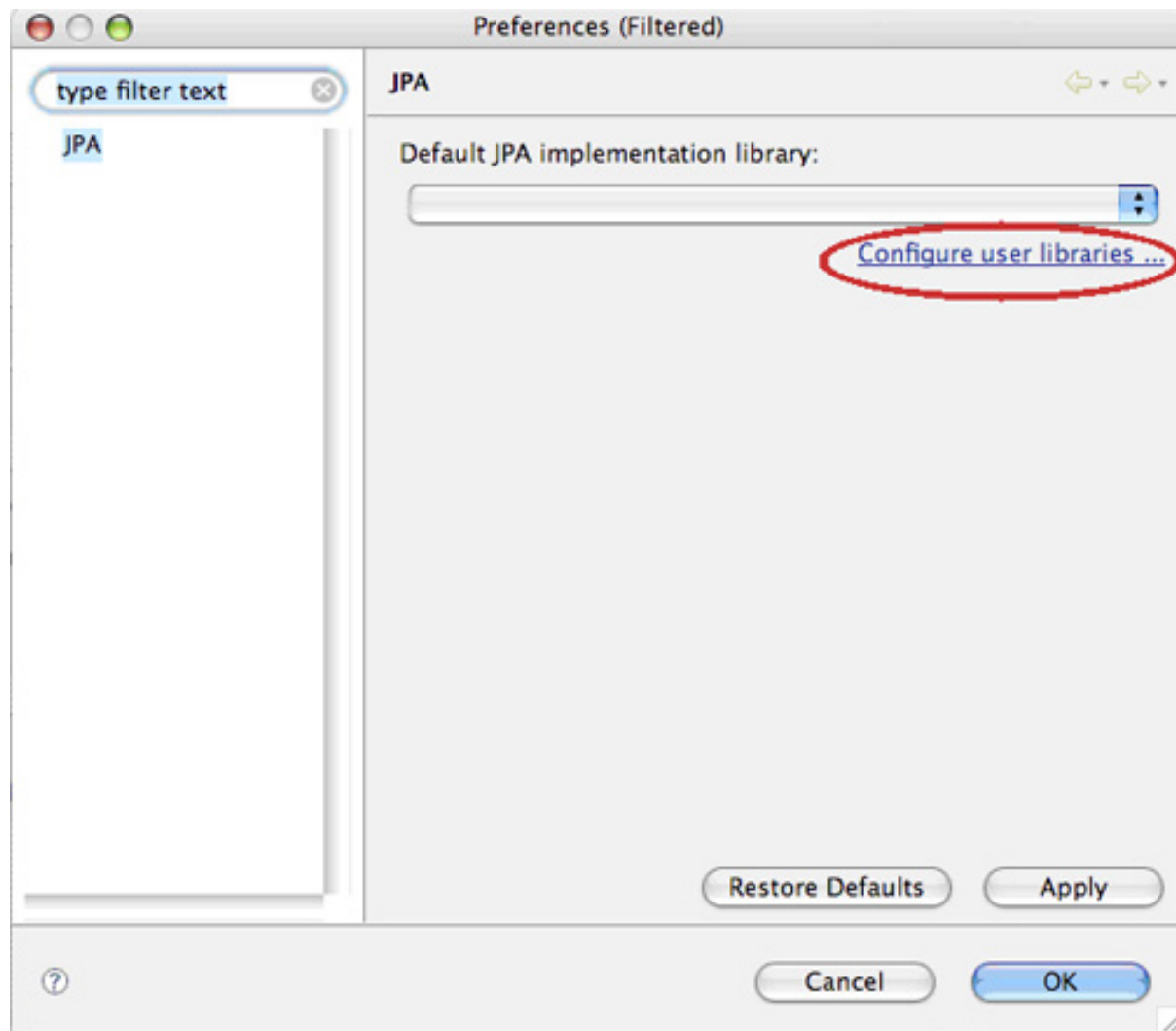
Clicking **Next** will bring up the JPA configuration. Since we are using Tomcat, we don't have a "built-in" JPA implementation. As mentioned in System requirements, we use OpenJPA. We need to select the **Configure default JPA implementation library**.

**Figure 29. JPA configuration**



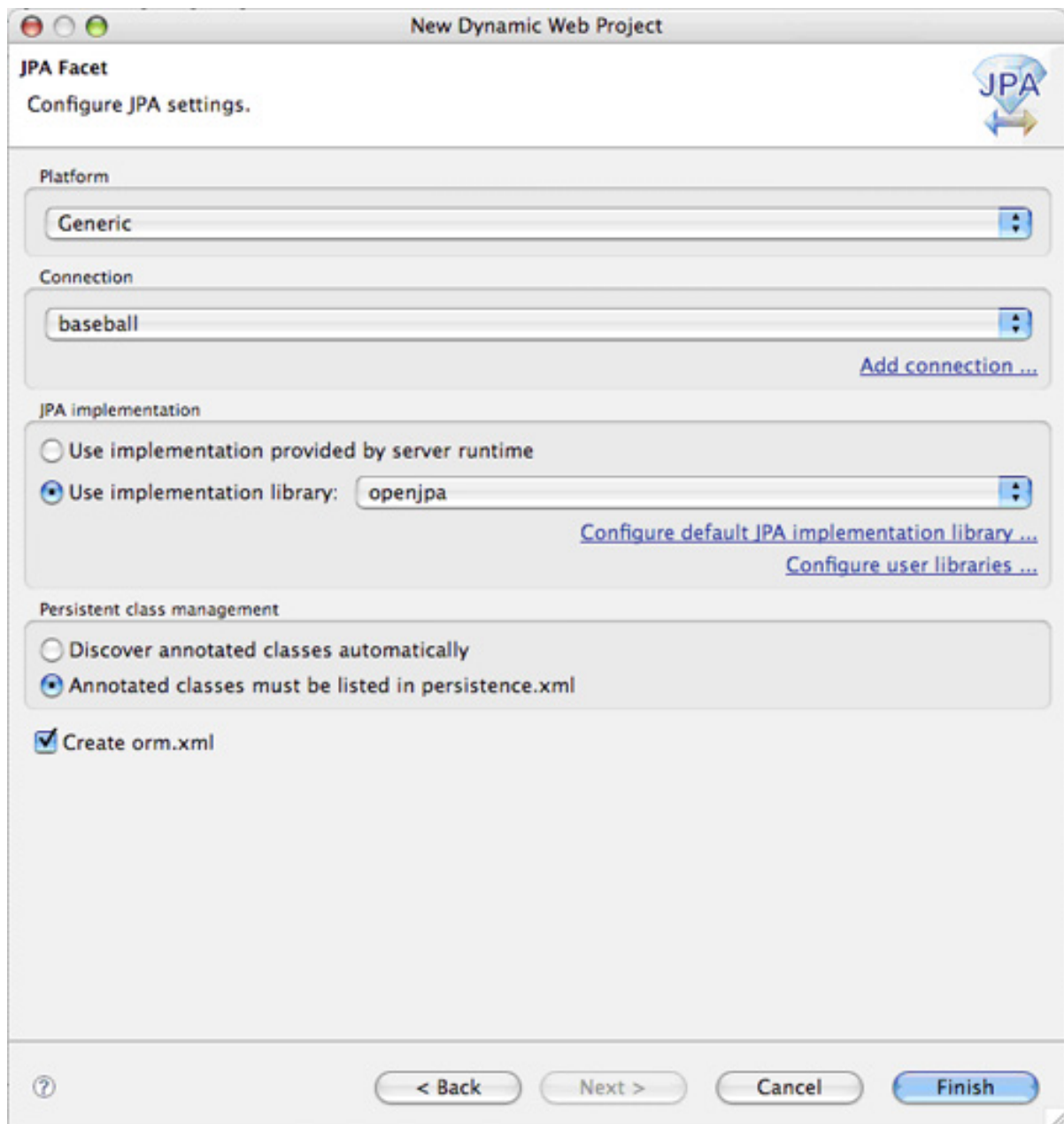
This will bring up the Default JPA Implementation Library preference. Click on **Configure user libraries**.

**Figure 30. Setting user library for default JPA implementation library**



This will let you select the OpenJPA (or other implementation, such as TopLink or Hibernate) JAR file. Once you've configured your JPA implementation, you'll be taken back to the JPA configuration screen, where you can select your new library, then click **Finish**.

**Figure 31. JPA configuration complete**



Click **Finish** and Eclipse will get to work. It will create the directory structure we need for our Web application, as well as many artifacts we'll need, such as the web.xml and persistence.xml files. Now that our application is set up, let's start creating its domain model.

---

## Section 5. Eclipse and data access code: Using JPA

It's time to look at how the pain of working with databases in the Java language has been eased by the Java Persistence API and at how Java EE for Eclipse supports it



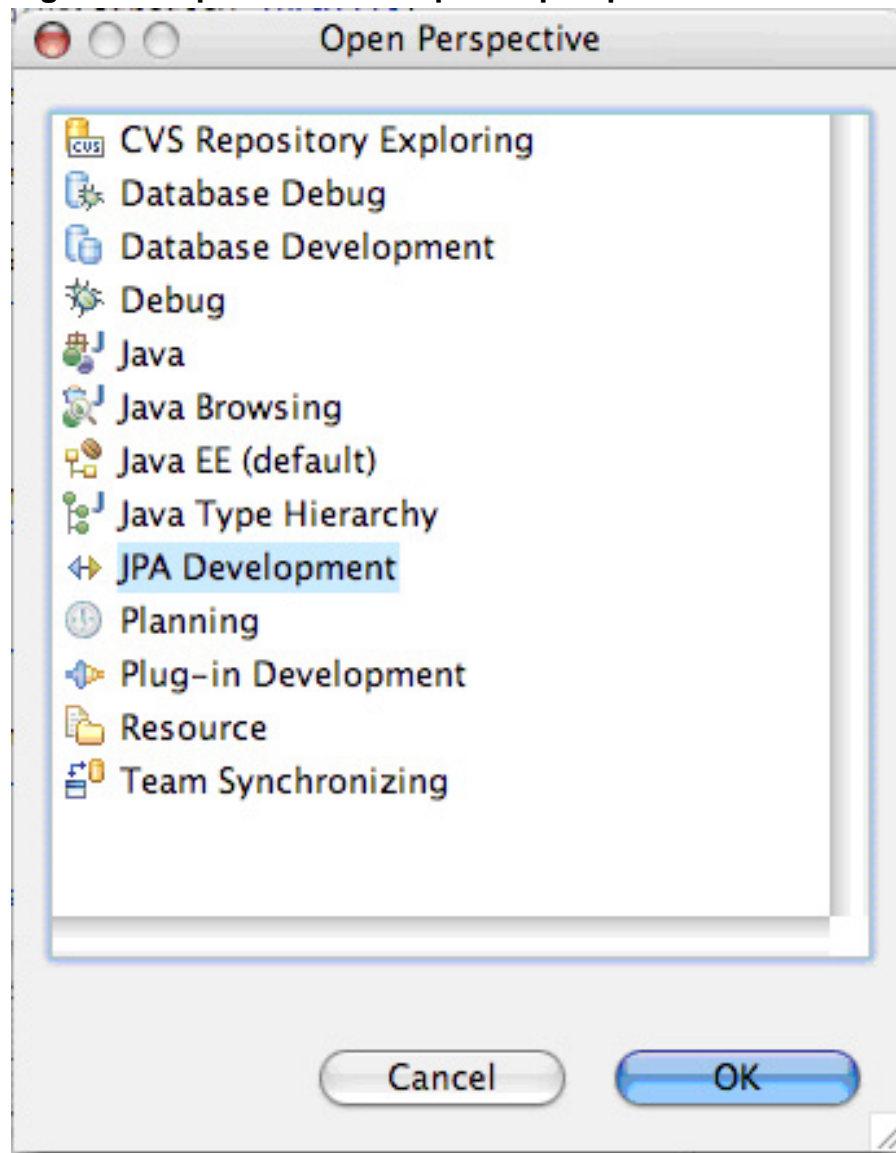
as we begin to create the domain model for our application.

## Creating the domain model: Using JPA

If you were an early Java developer, you know how painful working with databases could be. It usually involved either creating lots of JDBC code and long hand-coded SQL strings or it involved creating entity EJBs with many interfaces and writing deployment descriptors. Those days are long gone, thanks to the advent of EJB V3.0 and the JPA. Not only are EJBs easy to use now but also we don't need a heavyweight application server to use them.

Java EE for Eclipse has great support for JPA. There's even a JPA perspective. To switch to it, select **Window > Open Perspective > Other** and select **JPA Development**.

**Figure 32. Open JPA development perspective**



To create a domain model, we just need to create simple Java classes. We'll start by creating a class for the Players table, which we'll call `org.developerworks.baseball.Player`. The code for it is shown in Listing 5.

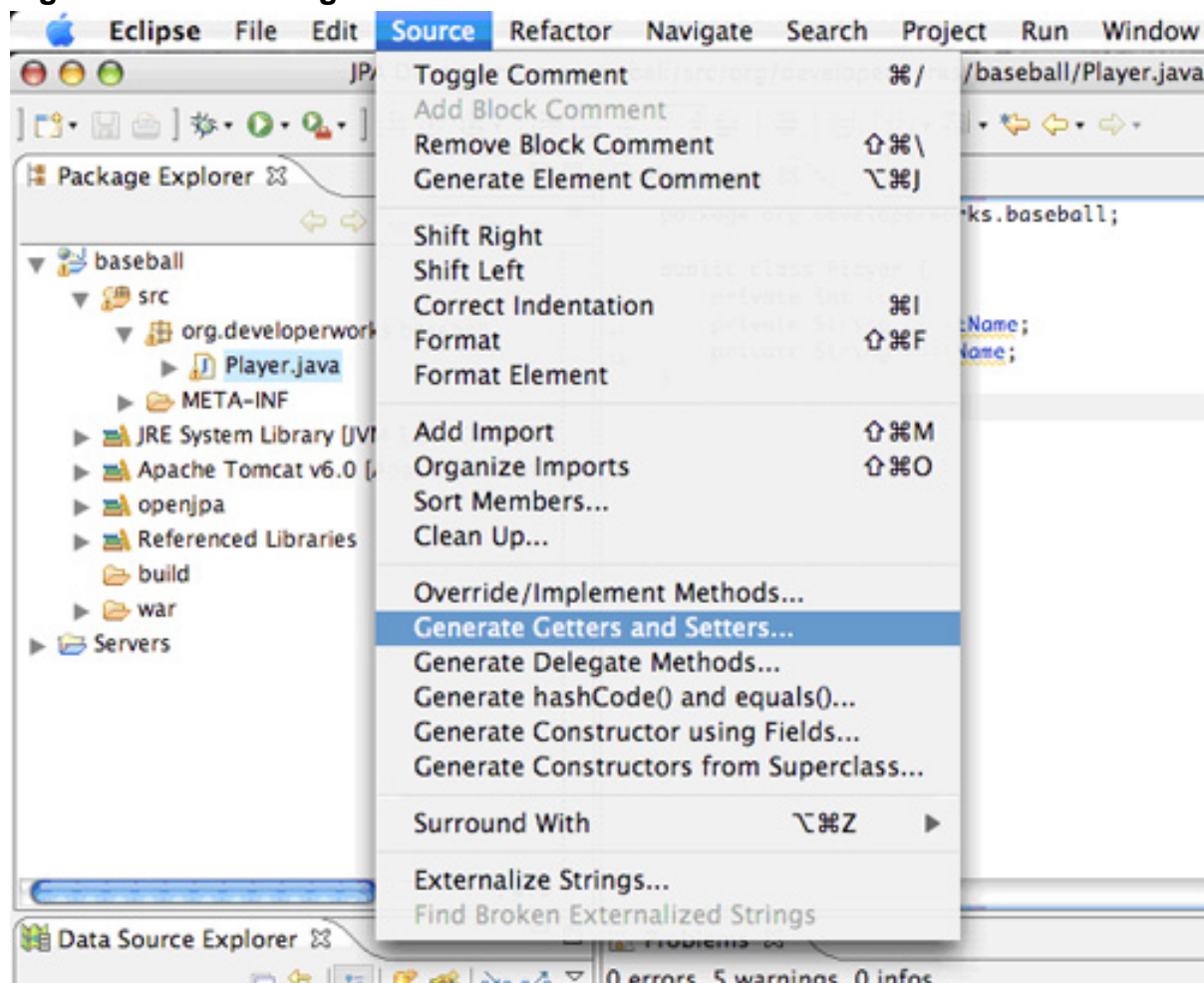
### Listing 5. Player class

```
package org.developerworks.baseball;

public class Player {
    private int id;
    private String firstName;
    private String lastName;
}
```

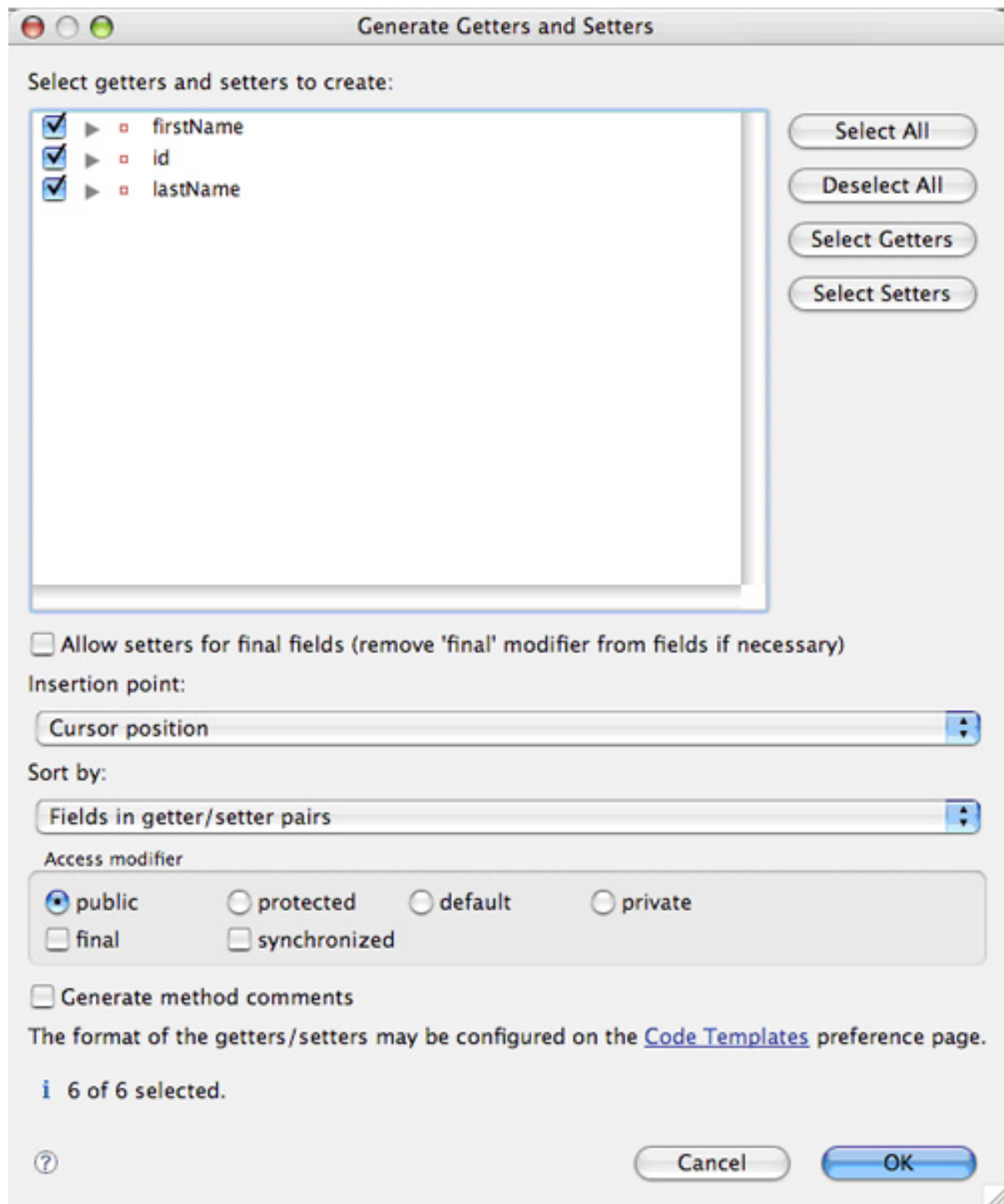
You'll need getters and Setters for this class. Luckily, Eclipse makes this easy. Select **Source > Generate Getters and Setters**.

Figure 33. Generating code



This will bring up the Generate Getters and Setters dialog. Click **Select All** and **OK**.

Figure 34. Generating getters and setters



The resulting code is shown in Listing 6.

#### Listing 6. Player class with getters and setters

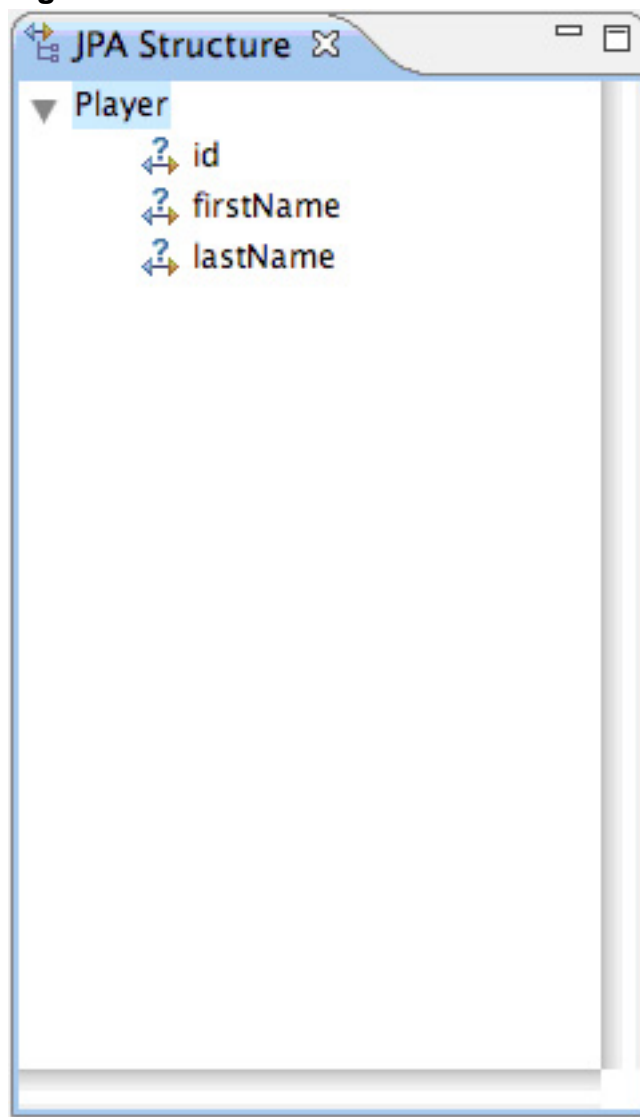
```
package org.developerworks.baseball;

public class Player {
    private int id;
    private String firstName;
    private String lastName;
    public int getId() {
```

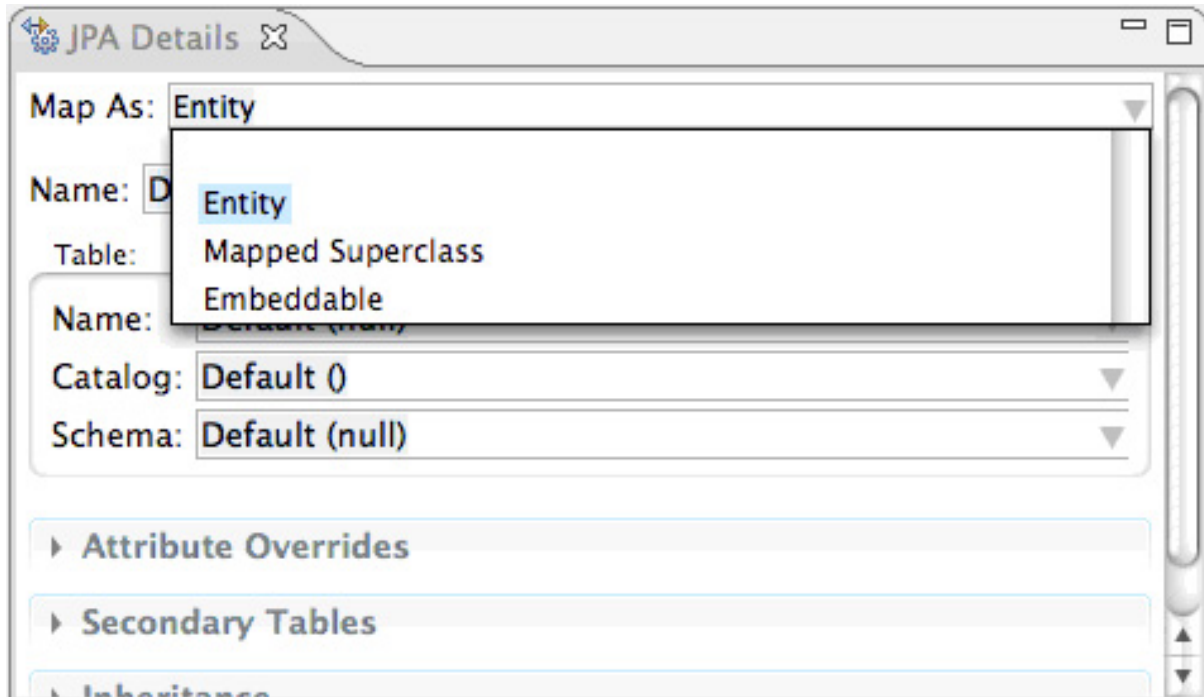
```
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

To associate our class with the database table, click on the `Player` class in the JPA Structure window.

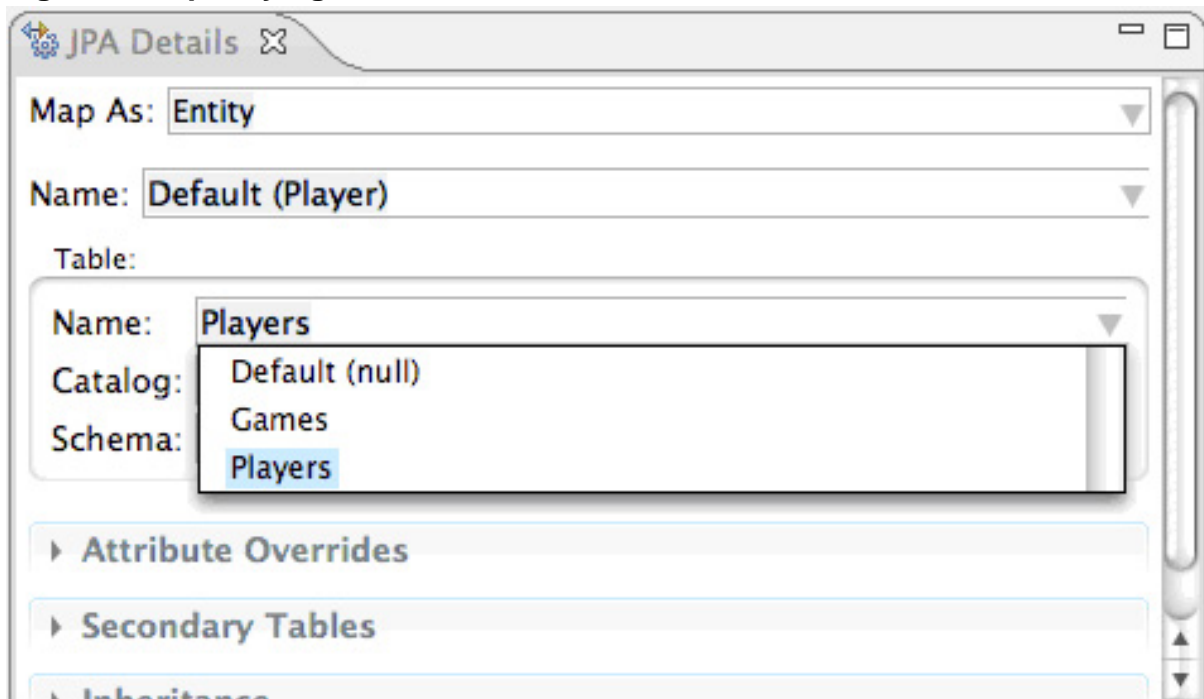
**Figure 35. JPA structure**



In **JPA Details**, select **Map As > Entity**.

**Figure 36. JPA details**

In the **Table** section, select the schema, and you'll be able to select the Players table from the **Name** drop-down.

**Figure 37. Specifying the table**

This should cause your code to change.

**Figure 38. Generated player code**

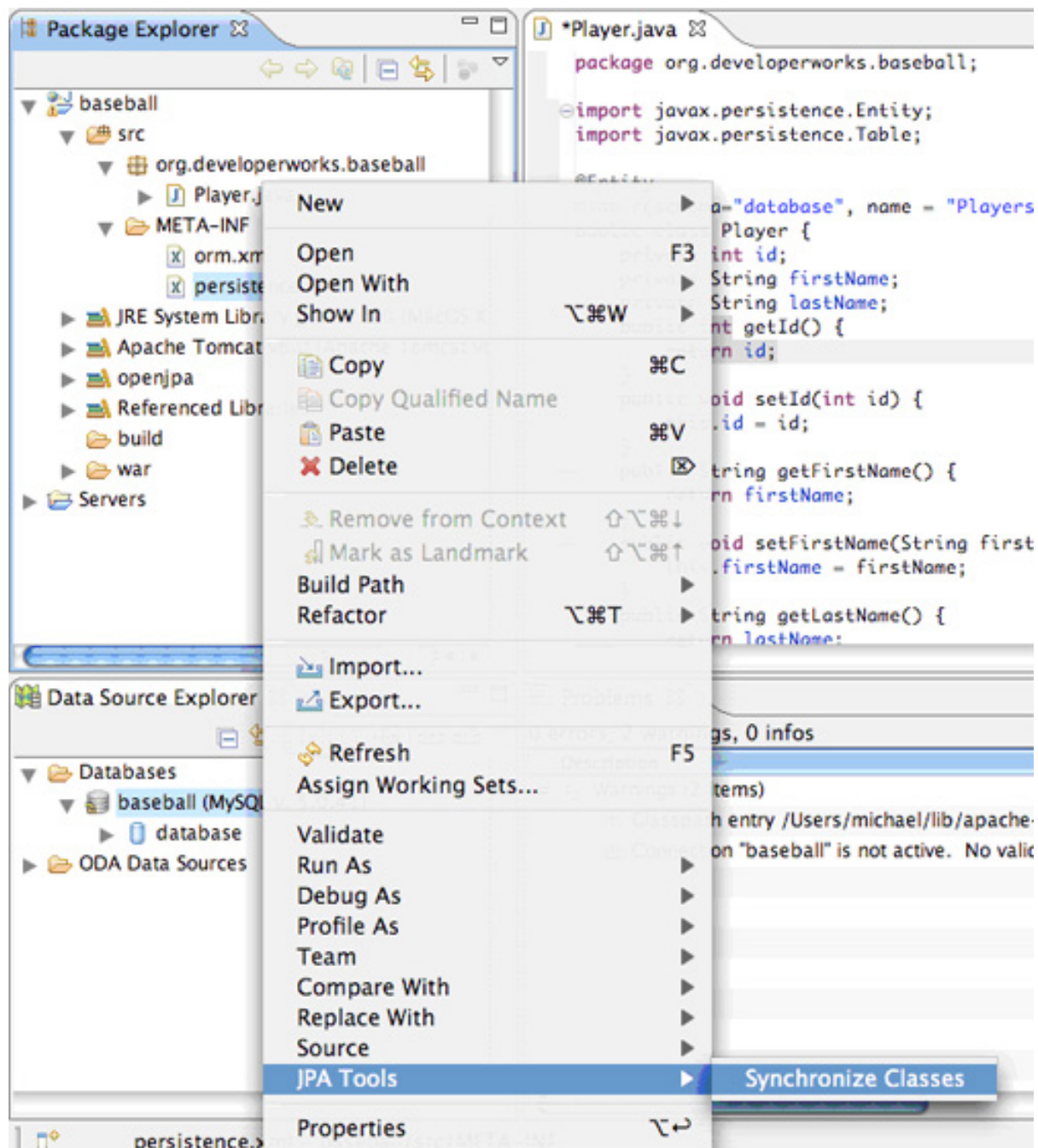


```
*Player.java ✕  
  
package org.developerworks.baseball;  
  
import javax.persistence.Entity;  
import javax.persistence.Table;  
  
@Entity  
@Table(schema="database", name = "Players")  
public class Player {  
    private int id;  
    private String firstName;  
    private String lastName;  
}
```

Select the persistence.xml file in the Package Explorer, right-click on it, and select **JPA Tools > Synchronize Classes**.

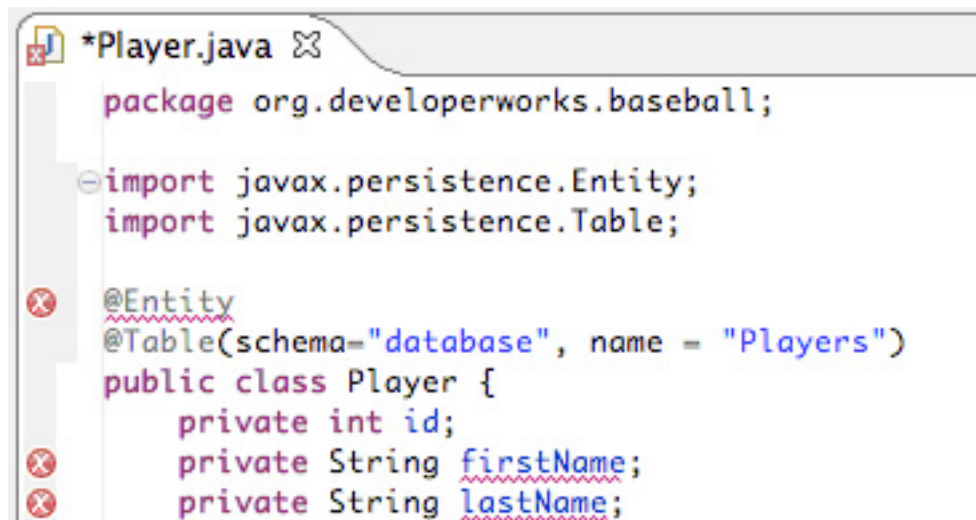
**Figure 39. Synchronize with database**





This will cause Eclipse to validate your code against the database. You'll notice that your code is not valid.

**Figure 40. Invalid Player class**



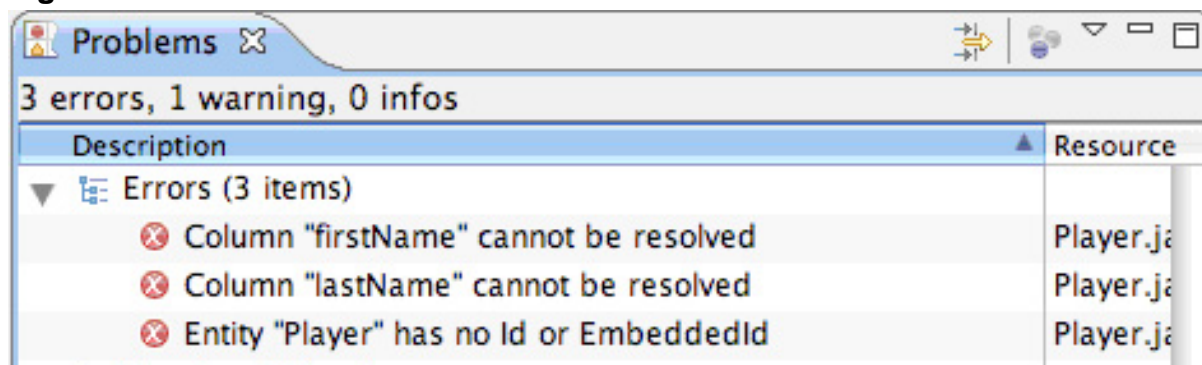
```
*Player.java
package org.developerworks.baseball;

import javax.persistence.Entity;
import javax.persistence.Table;

@Entity
@Table(schema="database", name = "Players")
public class Player {
    private int id;
    private String firstName;
    private String lastName;
```

What's going on here? Just take a look at the Problems window.

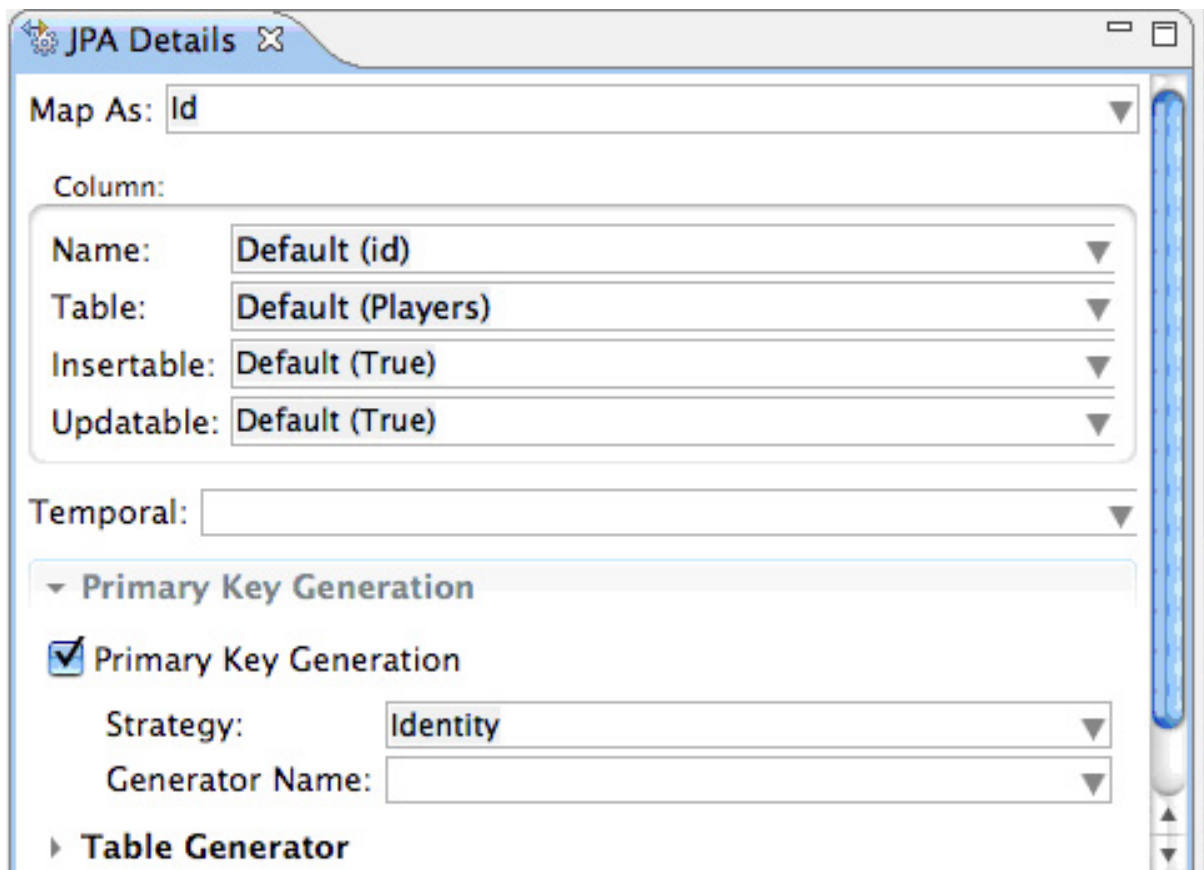
**Figure 41. Problems window**



The first problem is that our class has no primary key specified. To fix this, click on the Id property in the JPA Structure window. In the **JPA Details** window, select **Map As > Id**.

**Figure 42. JPA details for ID property**

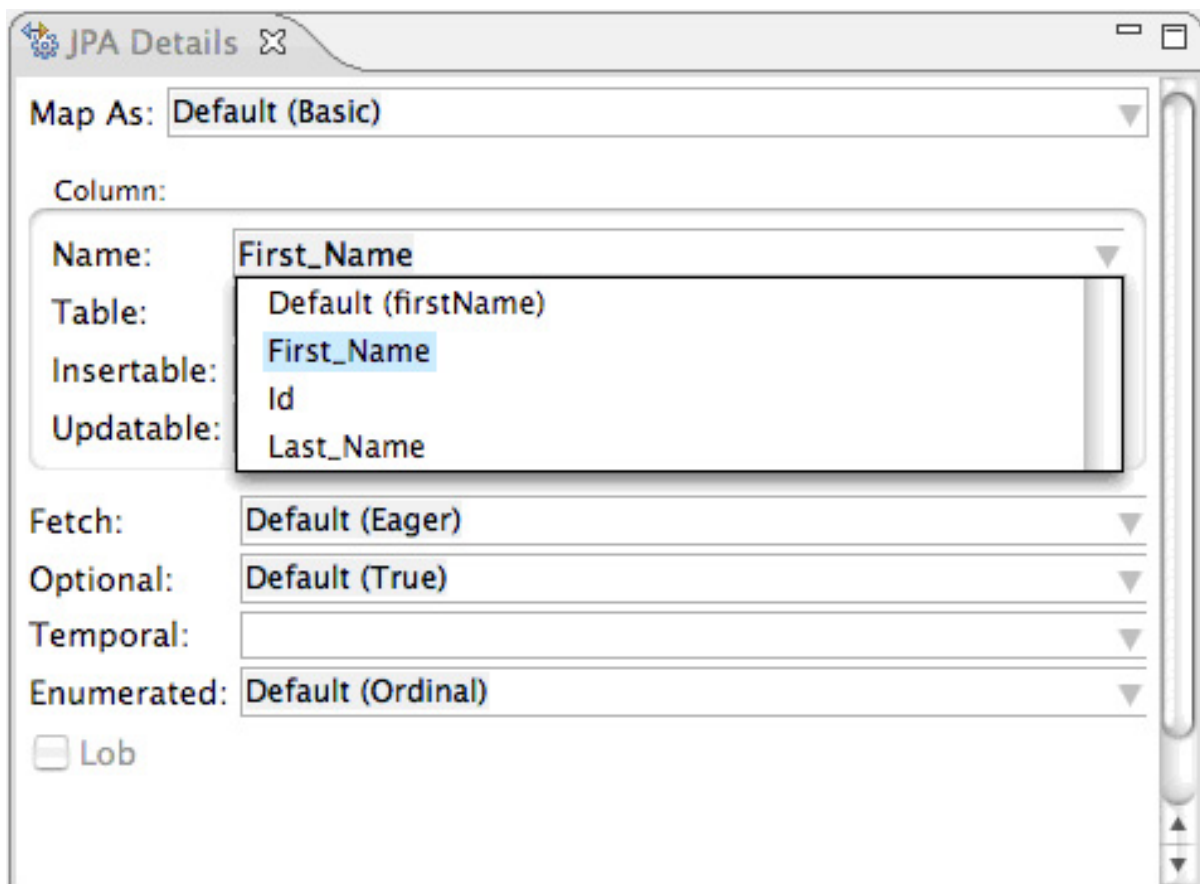




Also notice that since we made our ID column an auto-increment column, we need to specify that our primary key uses an Identity key-generation strategy. If you perform another JPA synchronization, you should see that we've fixed one of our problems.

The other problems come from JPA not being able to map our `firstName` and `lastName` fields to columns on the `Players` table. That's because we didn't name our Java fields with the exact same names as our table columns. Select the `firstName` field in the JPA Structure window. In the **JPA Details**, pick the appropriate column from the drop-down list.

**Figure 43. Mapping `firstName` to `First_Name`**



Do the same thing with the `lastName` field. Perform one more JPA synchronization, and all problems should be fixed. The code for the `Player` class should be updated and is shown in Listing 7.

### Listing 7. Finished Player class

```
package org.developerworks.baseball;

import javax.persistence.Entity;
import javax.persistence.Table;
import javax.persistence.Id;
import javax.persistence.GeneratedValue;
import static javax.persistence.GenerationType.IDENTITY;
import javax.persistence.Column;

@Entity
@Table(schema="baseball", name = "Players")
public class Player {
    @Id
    @GeneratedValue(strategy=IDENTITY)
    private int id;
    @Column(name="First_Name")
    private String firstName;
    @Column(name="Last_Name")
    private String lastName;
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getFirstName() {
        return firstName;
    }
}
```

```

    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}

```

Now we can follow the same process and create a `Game` class mapped to the `Games` table. The code for the `Game` class is shown in Listing 8.

### Listing 8. Game class

```

package org.developerworks.baseball;

import javax.persistence.Entity;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.Table;
import javax.persistence.Id;
import javax.persistence.GeneratedValue;
import static javax.persistence.GenerationType.IDENTITY;
import javax.persistence.Column;

@Entity
@Table(schema="baseball", name = "Games")
public class Game {
    @Id
    @GeneratedValue(strategy=IDENTITY)
    private int id;
    @ManyToOne(optional=false)
    @JoinColumn(name="Player_Id", nullable=false, updatable=false)
    private Player player;
    @Column(name="H")
    private int hits;
    @Column(name="2B")
    private int doubles;
    @Column(name="3B")
    private int triples;
    @Column(name="HR")
    private int homeRuns;
    @Column(name="BB")
    private int walks;
    @Column(name="R")
    private int runs;
    @Column(name="RBI")
    private int rbis;
    @Column(name="AB")
    private int atBats;
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public Player getPlayer() {
        return player;
    }
    public void setPlayer(Player player) {
        this.player = player;
    }
    public int getHits() {
        return hits;
    }
    public void setHits(int hits) {
        this.hits = hits;
    }
}

```

```

    }
    public int getDoubles() {
        return doubles;
    }
    public void setDoubles(int doubles) {
        this.doubles = doubles;
    }
    public int getTriples() {
        return triples;
    }
    public void setTriples(int triples) {
        this.triples = triples;
    }
    public int getHomeRuns() {
        return homeRuns;
    }
    public void setHomeRuns(int homeRuns) {
        this.homeRuns = homeRuns;
    }
    public int getWalks() {
        return walks;
    }
    public void setWalks(int walks) {
        this.walks = walks;
    }
    public int getRuns() {
        return runs;
    }
    public void setRuns(int runs) {
        this.runs = runs;
    }
    public int getRbis() {
        return rbis;
    }
    public void setRbis(int rbis) {
        this.rbis = rbis;
    }
    public int getAtBats() {
        return atBats;
    }
    public void setAtBats(int atBats) {
        this.atBats = atBats;
    }
}

```

Our classes are mapped. The last thing we need to do is edit our persistence.xml. This is the key metadata class for the Java Persistence API and contains connection information.

### Listing 9. persistence.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
        http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
    <persistence-unit name="baseball">
        <class>org.developerworks.baseball.Game</class>
        <class>org.developerworks.baseball.Player</class>
    <properties>
        <property name=\
"openjpa.ConnectionURL" value="jdbc:mysql://localhost:3306/baseball"/>
        <property name="openjpa.ConnectionDriverName" value="com.mysql.jdbc.Driver"/>
        <property name="openjpa.ConnectionUserName" value="root"/>
        <property name="openjpa.ConnectionPassword" value="password"/>
        <property name="openjpa.Log" value="DefaultLevel=WARN, Tool=INFO"/>
    </properties>
    </persistence-unit>
</persistence>

```

Eclipse should have already put the classes in this file for you. You'll only need to put in the connection information for your database. Now that we've used Eclipse to set up the Java Persistence API for our data access, we're ready to create a UI for this data.

---

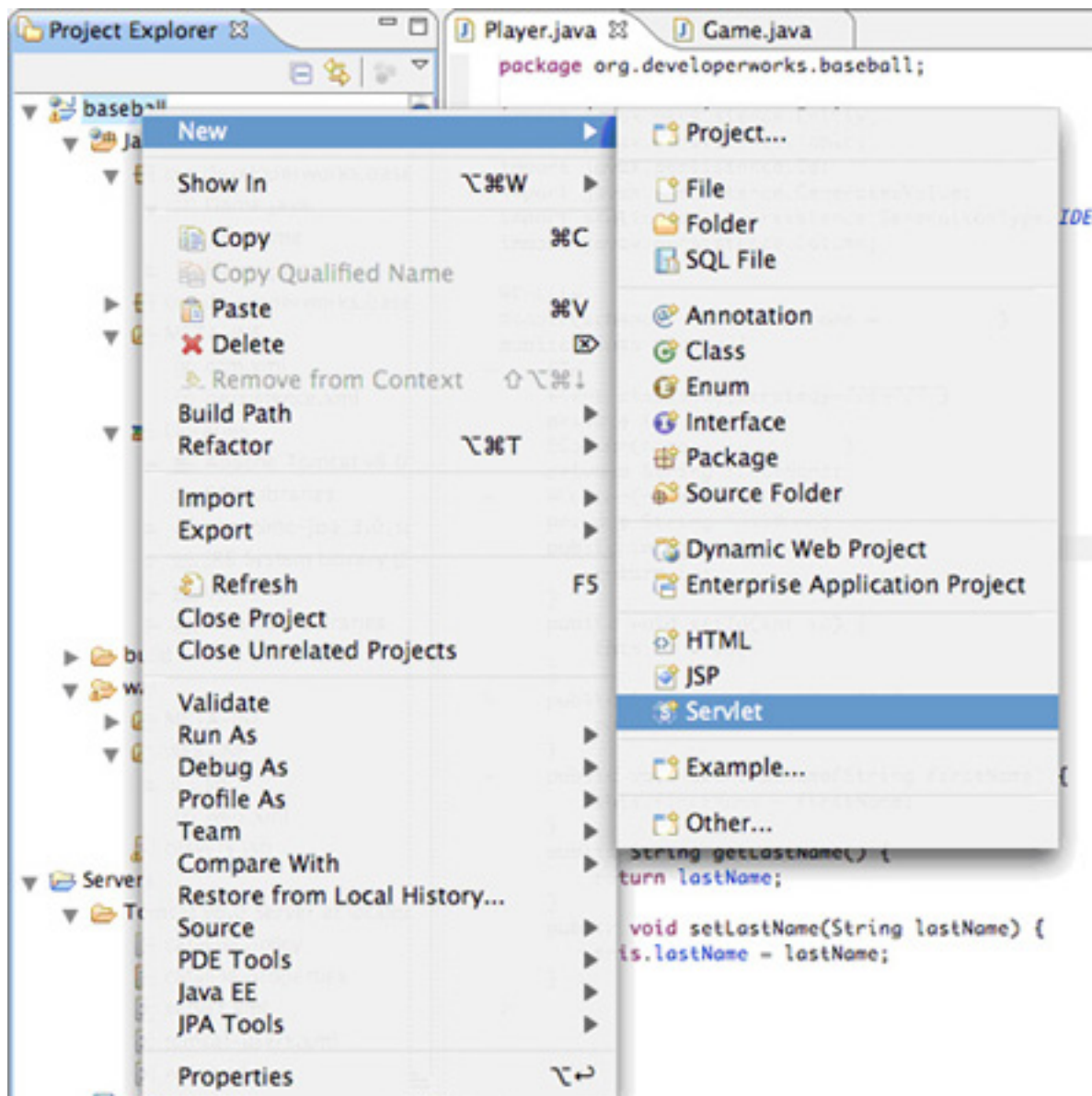
## Section 6. Eclipse and Web pages

In this section, we will create the UI for the data that you will be accessing. We'll use a JSP Model 2 architecture for our application. We'll start by creating a servlet for handling the business logic, then forwarding it to a JSP for rendering the output.

### Creating the servlet

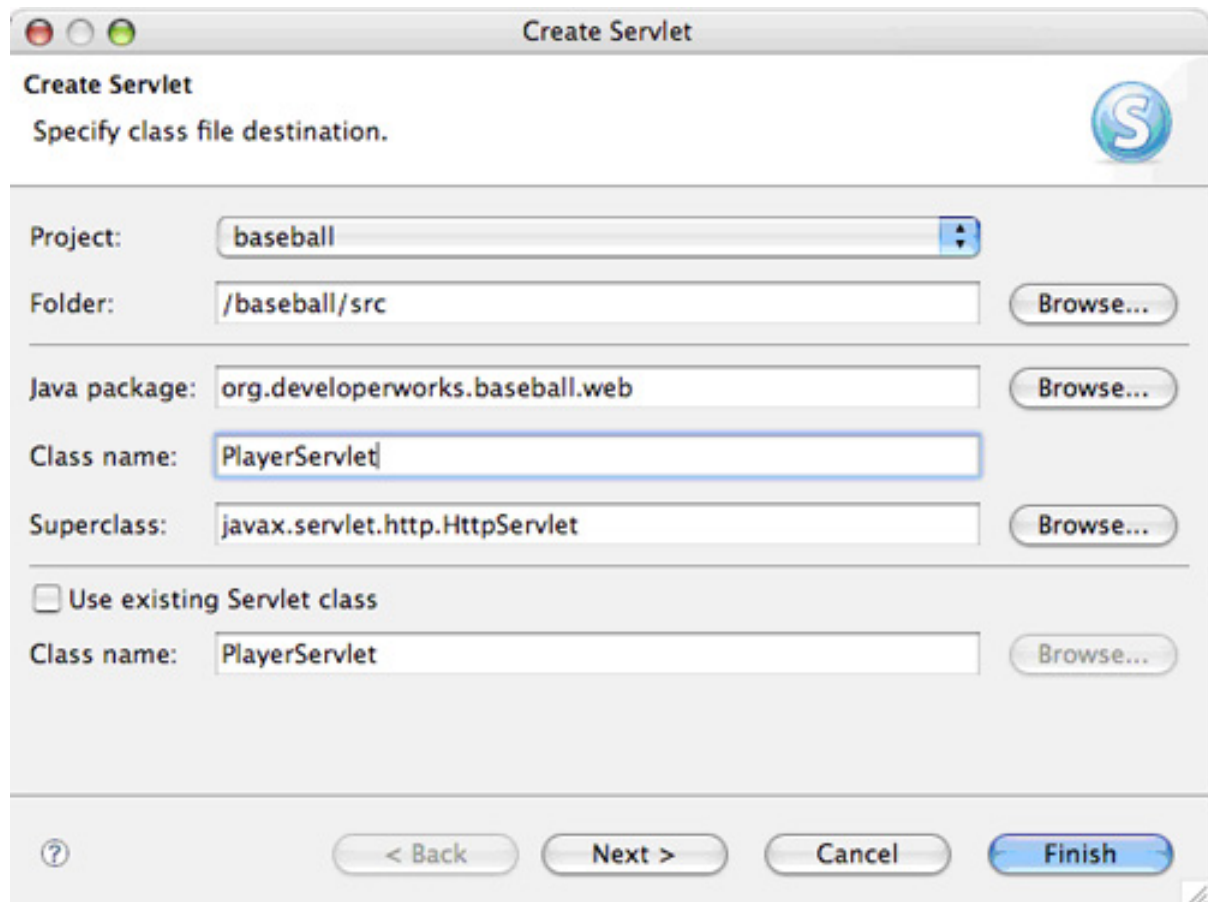
We'll use Eclipse to create our servlet. Go back to the Java EE perspective, right-click on the baseball project and select **New > Servlet**.

#### Figure 44. New servlet



This will bring up the Create Servlet dialog. We'll fill in the **Java package** and **Class name** for our servlet and click **Next**.

**Figure 45. Create servlet**



We don't need to edit anything here, but take note of the URL mapping to our servlet. This is the relative URL we need to access the servlet. Eclipse will set up this servlet mapping for us in our web.xml file automatically. We won't need to touch it at all. Take note of it and click **Finish**.

**Figure 46. Servlet info**

**Create Servlet**  
Enter servlet deployment descriptor specific information.

Name:

Description:

Initialization Parameters:

Add... Edit... Remove

URL Mappings:

Add... Edit... Remove

? < Back Next > Cancel Finish

In our `Servlet` class, we'll use the Java Persistence API to get a list of all the players and forward this to a JSP. The code for the servlet is shown in Listing 10.

### Listing 10. PlayerServlet code

```
package org.developerworks.baseball.web;

import java.io.IOException;
import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.developerworks.baseball.Player;

/**
 * Servlet implementation class for Servlet: PlayersServlet
 */
public class PlayersServlet extends javax.servlet.http.HttpServlet
    implements javax.servlet.Servlet {
    static final long serialVersionUID = 1L;
}
```



```

private EntityManagerFactory factory;
private EntityManager em;
/* (non-Java-doc)
 * @see javax.servlet.http.HttpServlet#HttpServlet()
 */
public PlayersServlet() {
    super();
    factory = Persistence.createEntityManagerFactory("baseball");
    em = factory.createEntityManager();
}

/* (non-Java-doc)
 * @see javax.servlet.http.HttpServlet#doGet(HttpServletRequest request,
 *                                           HttpServletResponse response)
 */
@SuppressWarnings("unchecked")
protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
{
    List<Player> players =
        em.createQuery("select p from Player p").getResultList();
    request.setAttribute("players", players);
    request.getRequestDispatcher("/players.jsp").forward(request, response);
}

/* (non-Java-doc)
 * @see javax.servlet.http.HttpServlet#doPost(HttpServletRequest request,
 *                                           HttpServletResponse response)
 */
protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
{
    this.doGet(request, response);
}
}

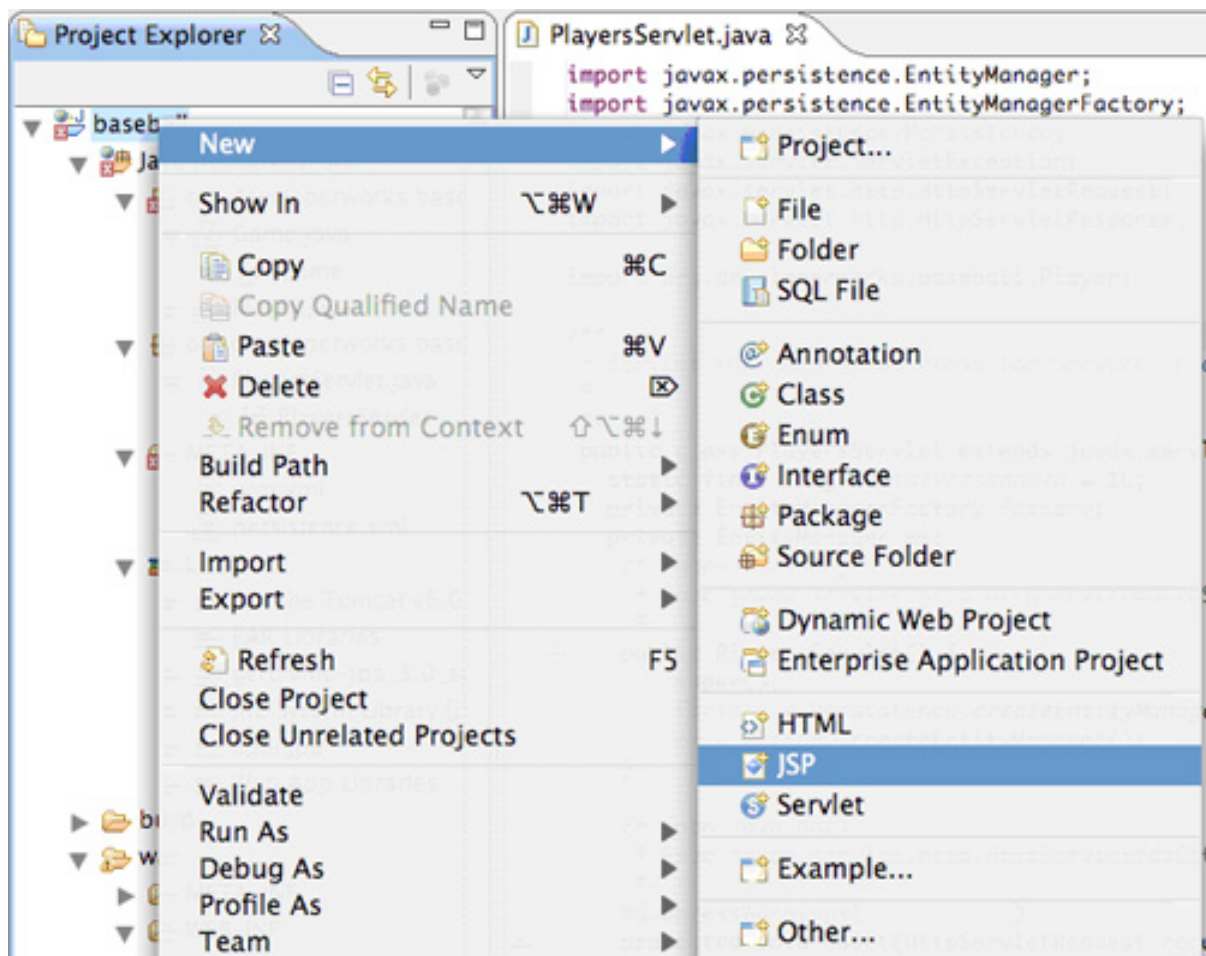
```

Notice that in the constructor to the servlet, we use the Java Persistence API to create an `EntityManager` instance referencing the "baseball" persistence unit. This comes from our `persistence.xml`, as shown in Listing 9.

## Creating a JSP for a view

Creating a JSP is easy with Eclipse. Just right-click on your baseball project and select **New > JSP**.

### Figure 47. New JSP



We start by just listing our baseball players. The code for that is shown in Listing 11.

### Listing 11. players.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ 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=ISO-8859-1">
<title>Baseball Players</title>
</head>
<body>
<table>
<thead>
<tr>
<td>First Name</td>
<td>Last Name</td>
</tr>
</thead>
<c:forEach items="${players}" var="player">
<tr>
<td>${player.firstName}</td>
<td>${player.lastName}</td>
</tr>
</c:forEach>
</table>
</body>
</html>
```

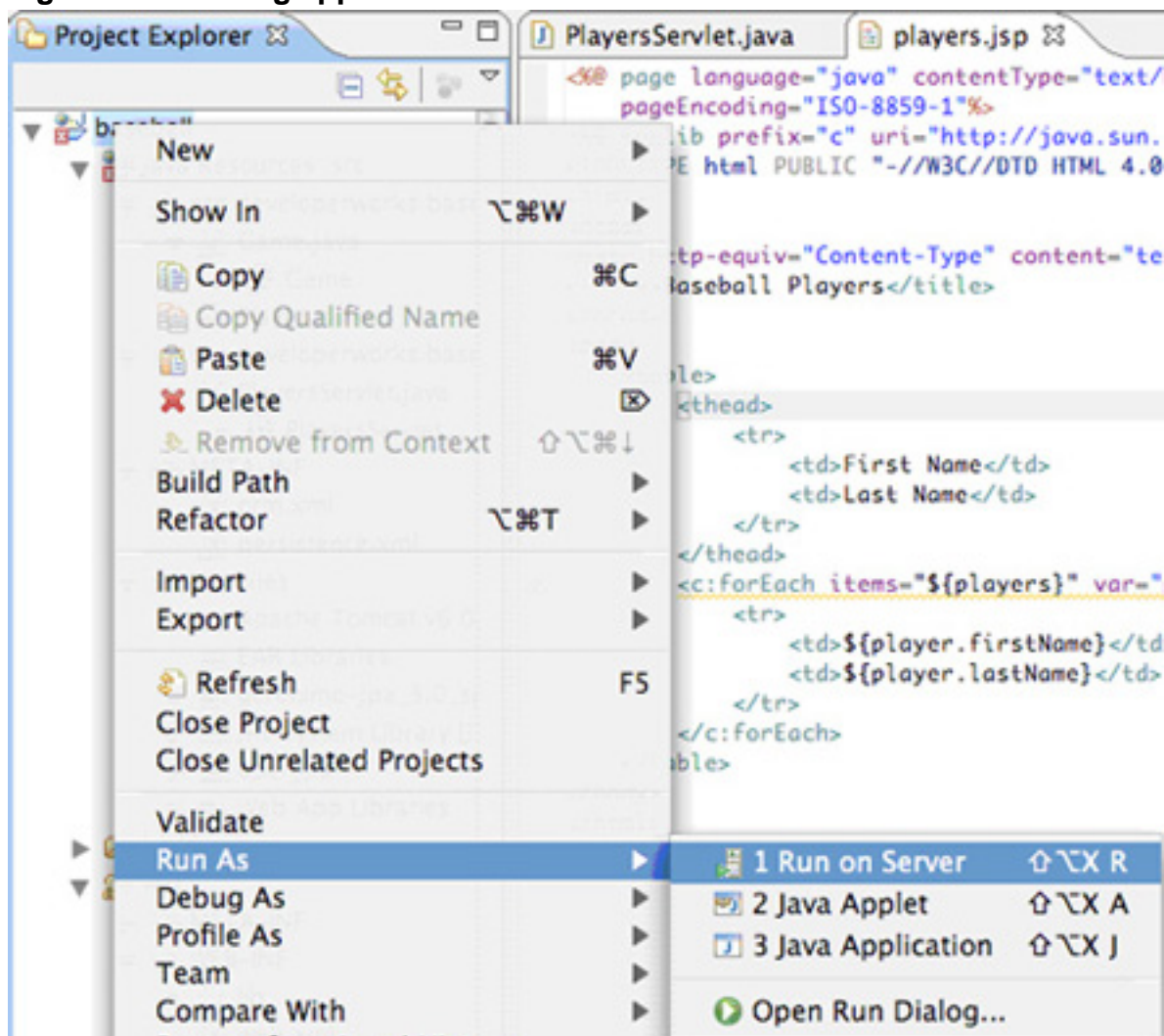
## Section 7. Deployment with Eclipse

We created a basic application, and we're ready to test it. At this point, you might think you need to write a deployment script maybe with Ant or some scripting language like Perl or Groovy. You could do that — Eclipse has good support for Ant and Maven — but there's an even easier way.

### Running from Eclipse

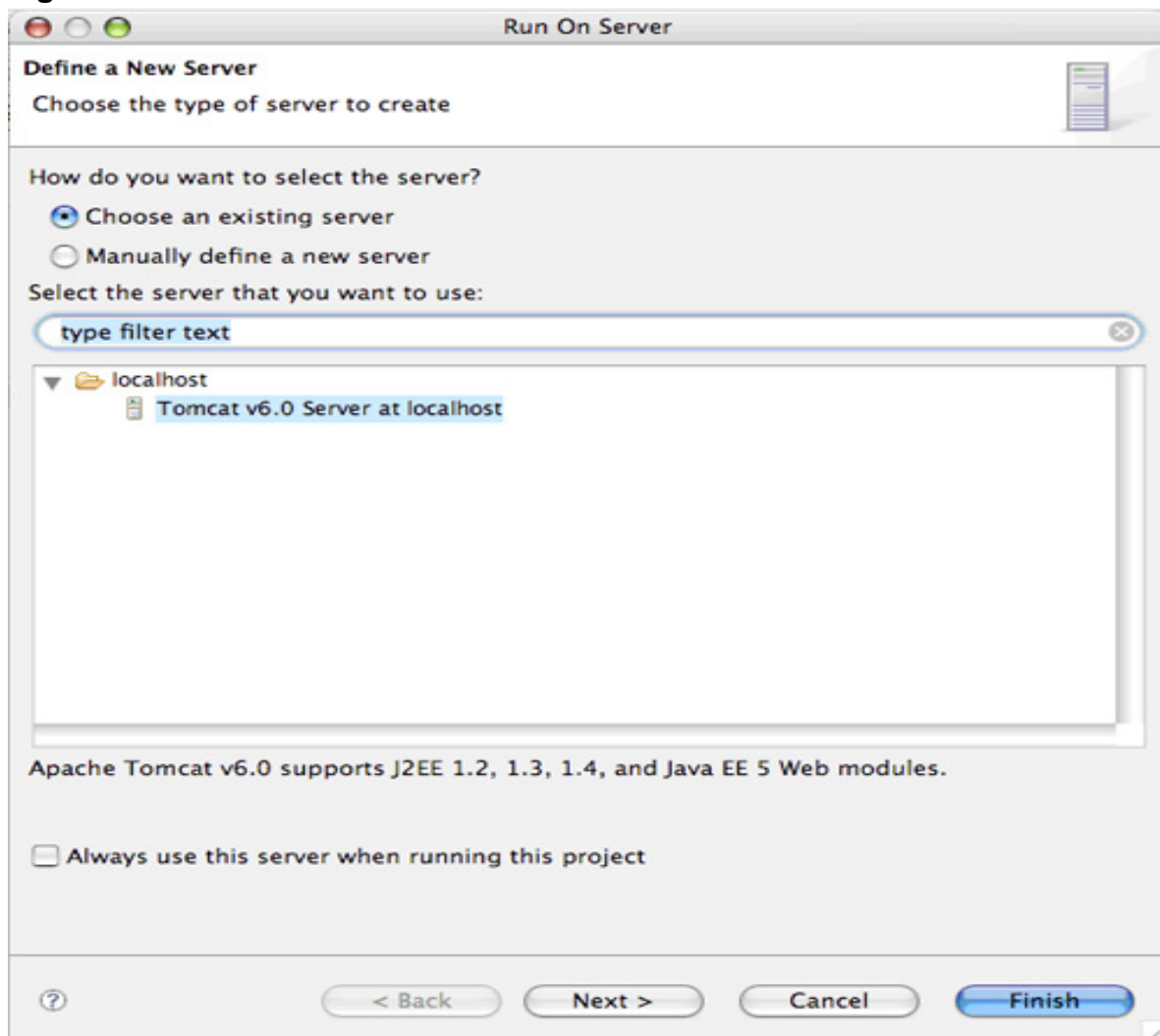
To deploy the application from Eclipse, simply right-click on the baseball project and select **Run As > Run On Server**.

**Figure 48. Running application on server**



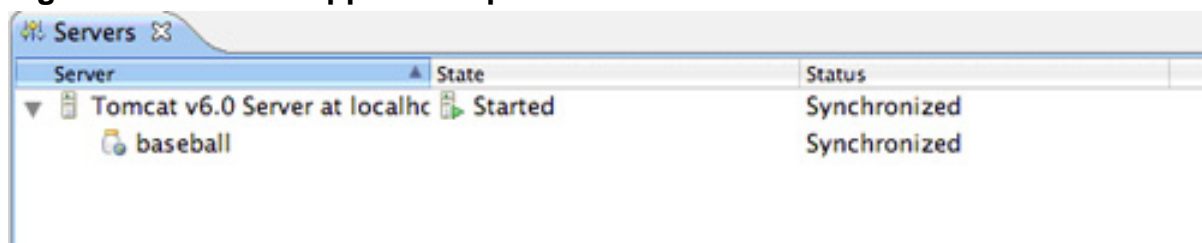
Eclipse will ask which server we want to deploy to. Simply select the Tomcat instance we created earlier and click **Finish**.

**Figure 49. Select server**



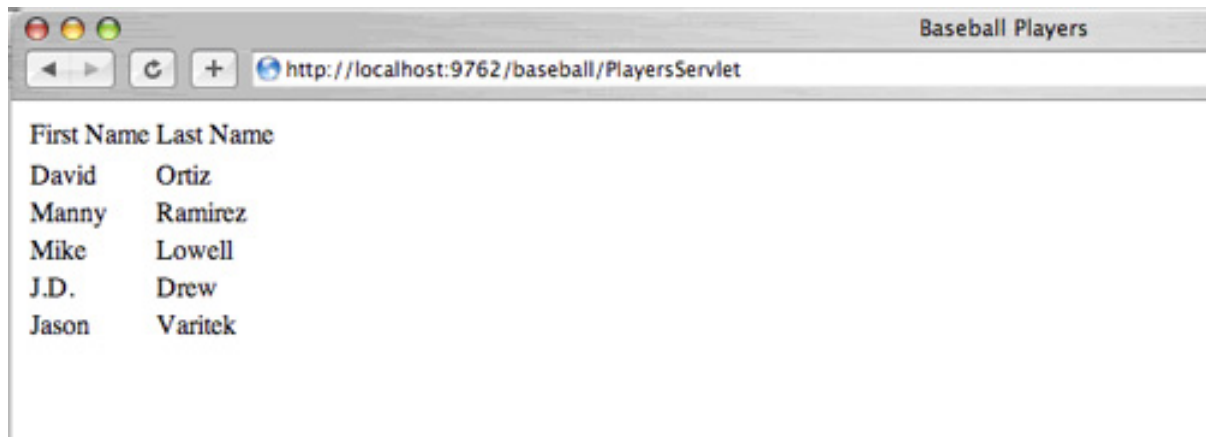
This will open the Servers window and show that the baseball application is published to Tomcat.

**Figure 50. Baseball application published to Tomcat**



Now you can bring up the application in a Web browser at <http://localhost:9762/baseball/PlayersServlet>, and it will show you the sample data you created earlier.

**Figure 51. Baseball application**



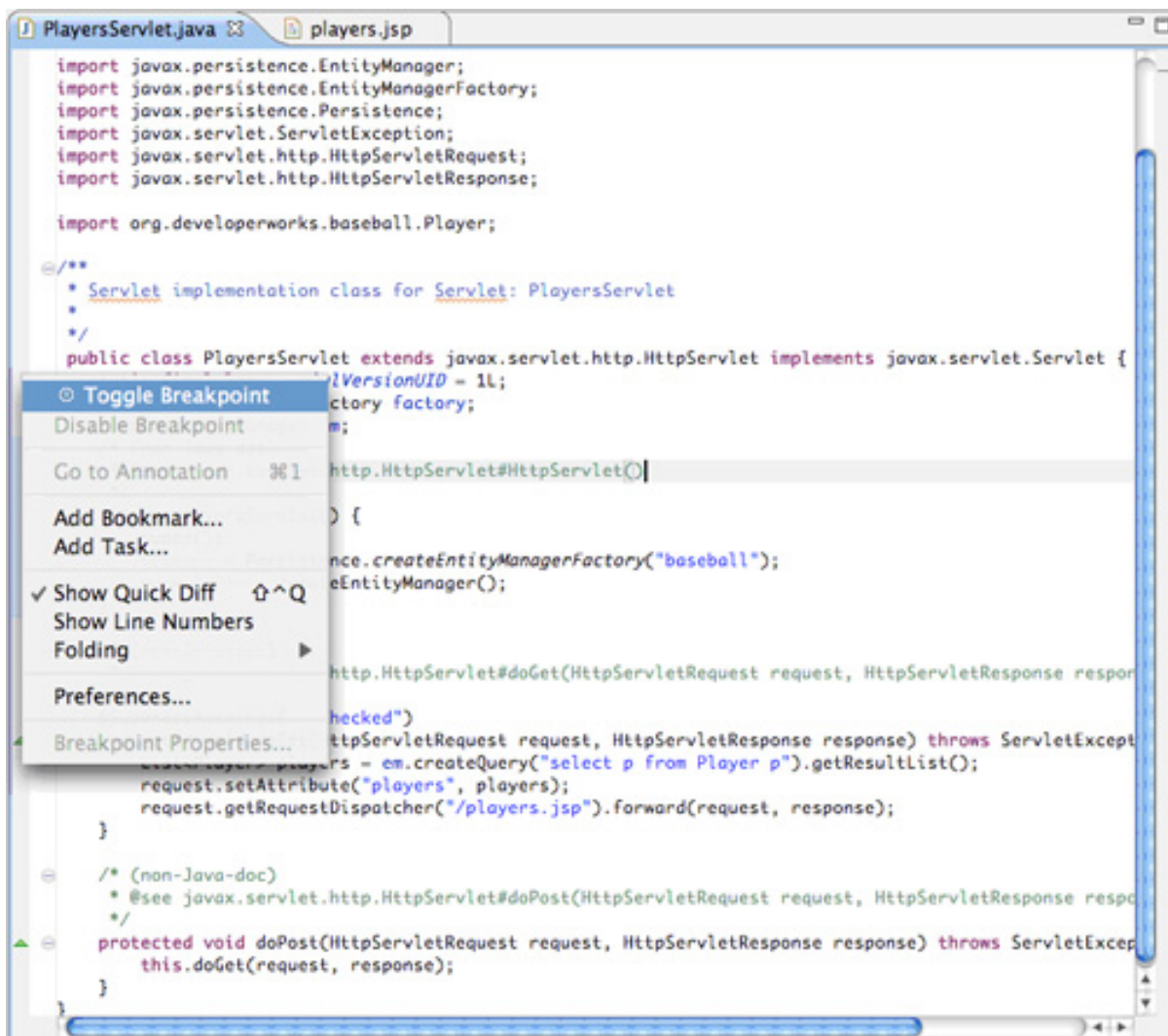
The list should be the same as the sample data created earlier. If you get any errors, you might want to make sure your connection properties are correct and that all the appropriate JARs have been copied to your /WEB-INF/lib directory. For example, the OpenJPA JARs (and dependencies) are needed, and your JDBC driver JAR should be there. We used some JSTL, so you'll need a JSTL JAR, as well. Remember that debugging is also easy using Eclipse.

## Debugging with Eclipse

Let's go back to our application and do some debugging. The debugging power of Eclipse is one of its major features, and being able to debug live code running in a Web server is invaluable. Let's go back to our servlet class and put a breakpoint where we query the database using the Java Persistence API.

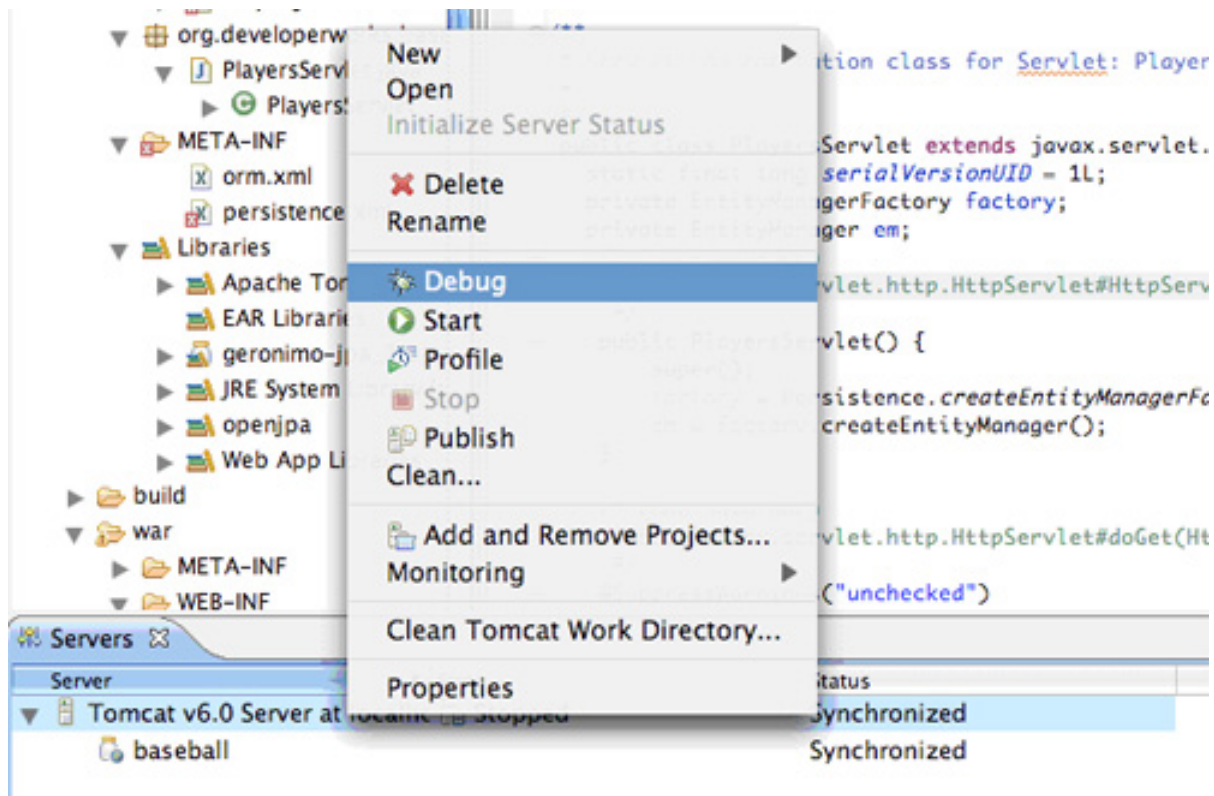
### Figure 52. Setting a breakpoint





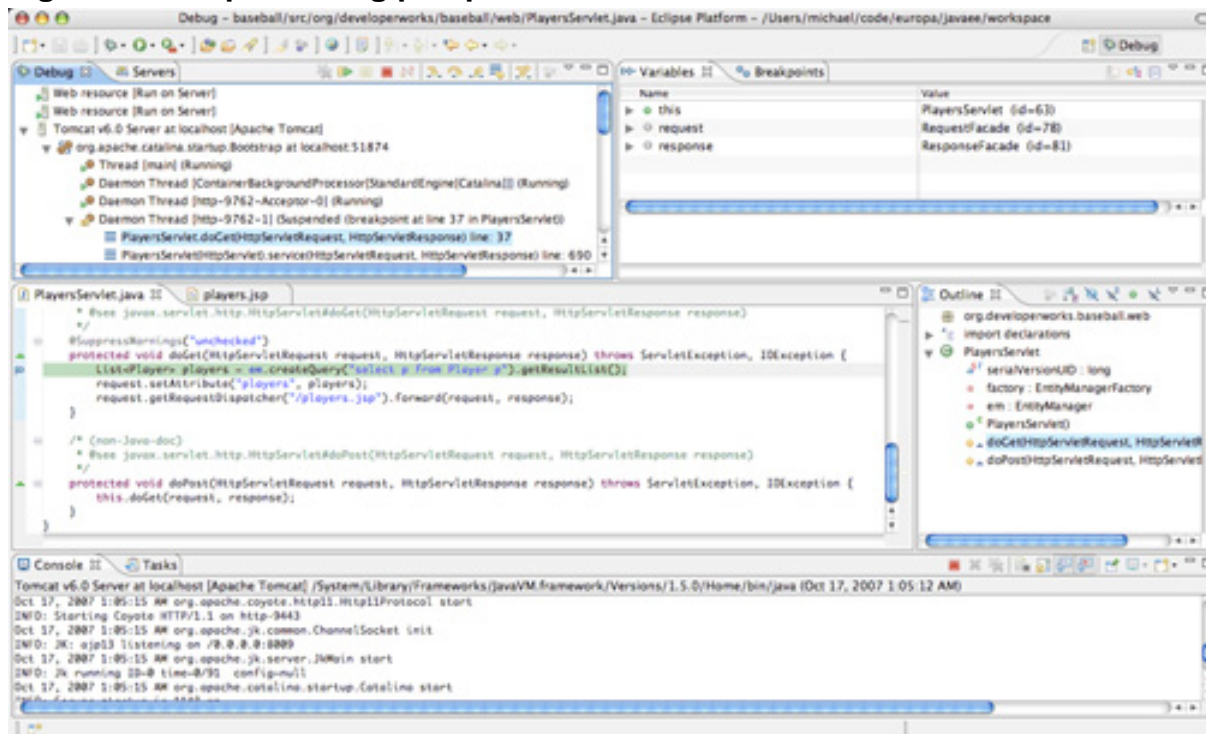
Now stop our server by right-clicking on it in the Server window and selecting **Stop**. We're going to restart it, but by selecting **Debug** this time.

**Figure 53. Debugging the server**



Now you can reload the Web page, and Eclipse should switch over to the Debug perspective.

**Figure 54. Eclipse Debug perspective**



From here, you can step through the code, set watch expressions, etc. We didn't have to do anything special to our server. We are debugging the Web application just like it was an executable class with a Java main method.

---

## Section 8. Summary

We've seen how Eclipse can be used for setting up everything you need for Web development. It integrates with databases, Web servers, and whatever else you need. It has numerous tools for helping you quickly leverage the Java EE technology stack. We used Eclipse to connect to a database to create tables and populate data, then turned around and leveraged the same tools to create Java Persistence API mappings to simplify our data access code. We used Eclipse to create JSPs and servlets. We were then able to deploy our Web application and even debug directly from Eclipse. From here, we can add more Web pages for creating players and games, and calculating statistics, which we do in [Part 2](#) of this three-part "[Web development with Eclipse Europa](#)" series.



## Downloads

Description	Name	Size	Download method
Sample code	os-eclipse-europa7MB	Baseball	<a href="#">http</a> <a href="#">zip</a>

[Information about download methods](#)

# Resources

## Learn

- Find all about the Europa release of Eclipse in the developerWorks article "[A whirlwind tour of Eclipse Europa](#)."
- Learn about the Java Persistence API in the developerWorks tutorial "[Design enterprise applications with the EJB 3.0 Persistence API](#)."
- Upgrade your legacy code to use the Java Persistence API by reading the developerWorks article "[Migrating legacy Hibernate applications to OpenJPA and EJB 3.0](#)."
- The Java EE Edition builds on top of the Web Tool Project. See some more great examples of its abilities in the developerWorks article "[Build Web applications with Eclipse, WTP, and Derby](#)."
- Check out the "[Recommended Eclipse reading list](#)."
- Browse all the [Eclipse content](#) on developerWorks.
- New to Eclipse? Read the developerWorks article "[Get started with Eclipse Platform](#)" to learn its origin and architecture, and how to extend Eclipse with plug-ins.
- Expand your Eclipse skills by checking out IBM developerWorks' [Eclipse project resources](#).
- To listen to interesting interviews and discussions for software developers, check out [developerWorks podcasts](#).
- For an introduction to the Eclipse platform, see "[Getting started with the Eclipse Platform](#)."
- Stay current with developerWorks' [Technical events and webcasts](#).
- Watch and learn about IBM and open source technologies and product functions with the no-cost [developerWorks On demand demos](#).
- Check out upcoming conferences, trade shows, webcasts, and other [Events](#) around the world that are of interest to IBM open source developers.
- Visit the developerWorks [Open source zone](#) for extensive how-to information, tools, and project updates to help you develop with open source technologies and use them with IBM's products.

## Get products and technologies

- Download [Eclipse Europa](#).
- Download [Java Development Kit \(JDK\) V5.0](#) or [V6.0](#).
- You'll also need the [Eclipse IDE for Java EE Developers](#).
- To run Eclipse, you must have a [JRE](#).

- The application in this tutorial uses a [Apache Tomcat](#) as its container and [MySQL V5.0](#) as its database.
- You will also need the [Java Persistence API](#) and, in particular, the [OpenJPA](#) implementation.
- Check out the latest [Eclipse technology downloads](#) at IBM [alphaWorks](#).
- Download [Eclipse Platform and other projects](#) from the Eclipse Foundation.
- Download [IBM product evaluation versions](#), and get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.
- Innovate your next open source development project with [IBM trial software](#), available for download or on DVD.

## Discuss

- The [Eclipse Platform newsgroups](#) should be your first stop to discuss questions regarding Eclipse. (Selecting this will launch your default Usenet news reader application and open eclipse.platform.)
- The [Eclipse newsgroups](#) has many resources for people interested in using and extending Eclipse.
- Participate in [developerWorks blogs](#) and get involved in the developerWorks community.

## About the author

Michael Galpin

Michael Galpin has been developing Java software professionally since 1998. He currently works for eBay. He holds a degree in mathematics from the California Institute of Technology.