Ticket Monster Tutorial

Contents

What is TicketMonster?

TicketMonster is an example application that focuses on Java EE6 - JSF 2, JPA 2, CDI and JAX-RS along with HTML5, jQuery Mobile, JSF and GWT. It is a moderately complex application that demonstrates how to build modern web applications optimized for mobile & desktop. TicketMonster is representative of an online ticketing broker - providing access to events (e.g. concerts, shows, etc) with an online booking application.

Apart from being a demo, TicketMonster provides an already existing application structure that you can use as a starting point for your app. You could try out your use cases, test your own ideas, or, contribute improvements back to the community.

Fork us on GitHub!

The accompanying tutorials walk you through the various tools & technologies needed to build TicketMonster on your own. Alternatively you can download TicketMonster as a completed application and import it into your favorite IDE.

Before we dive into the code, let's discuss the requirements for the application.

Use cases

We have grouped the current use cases in two major categories: end user oriented, and administrative.

What can end users do?

The end users of the application want to attend some cool events. They will try to find shows, create bookings, or cancel bookings. The use cases are:

- look for current events;
- look for venues;
- select shows (events taking place at specific venues) and choose a performance time;
- book tickets;
- view current bookings;
- cancel bookings;

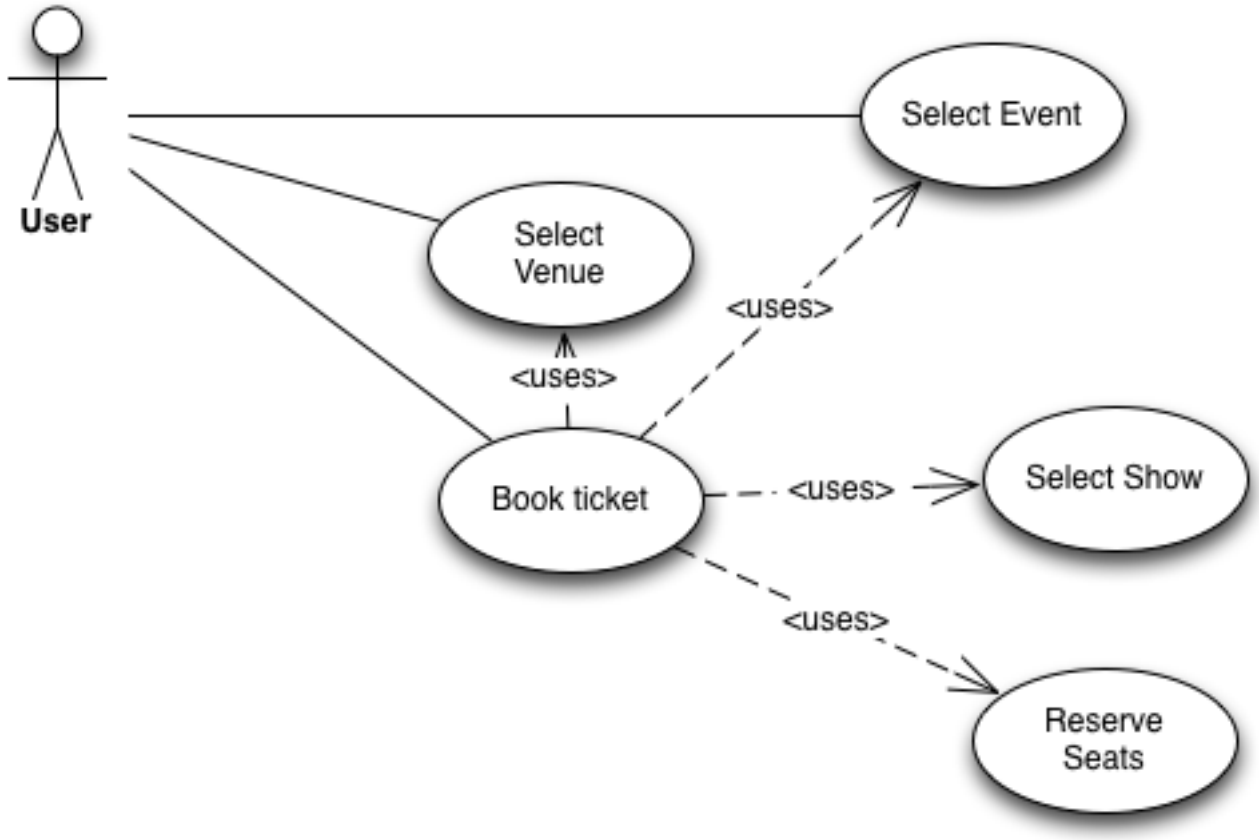

Figure 1: End user use cases

What can administrators do?

Administrators are more concerned the operation of the business. They will manage the *master data*: information about venues, events and shows, and will want to see how many tickets have been sold. The use cases are:

- add, remove and update events;
- add, remove and update venues (including venue layouts);
- add, remove and update shows and performances;
- monitor ticket sales for current shows;

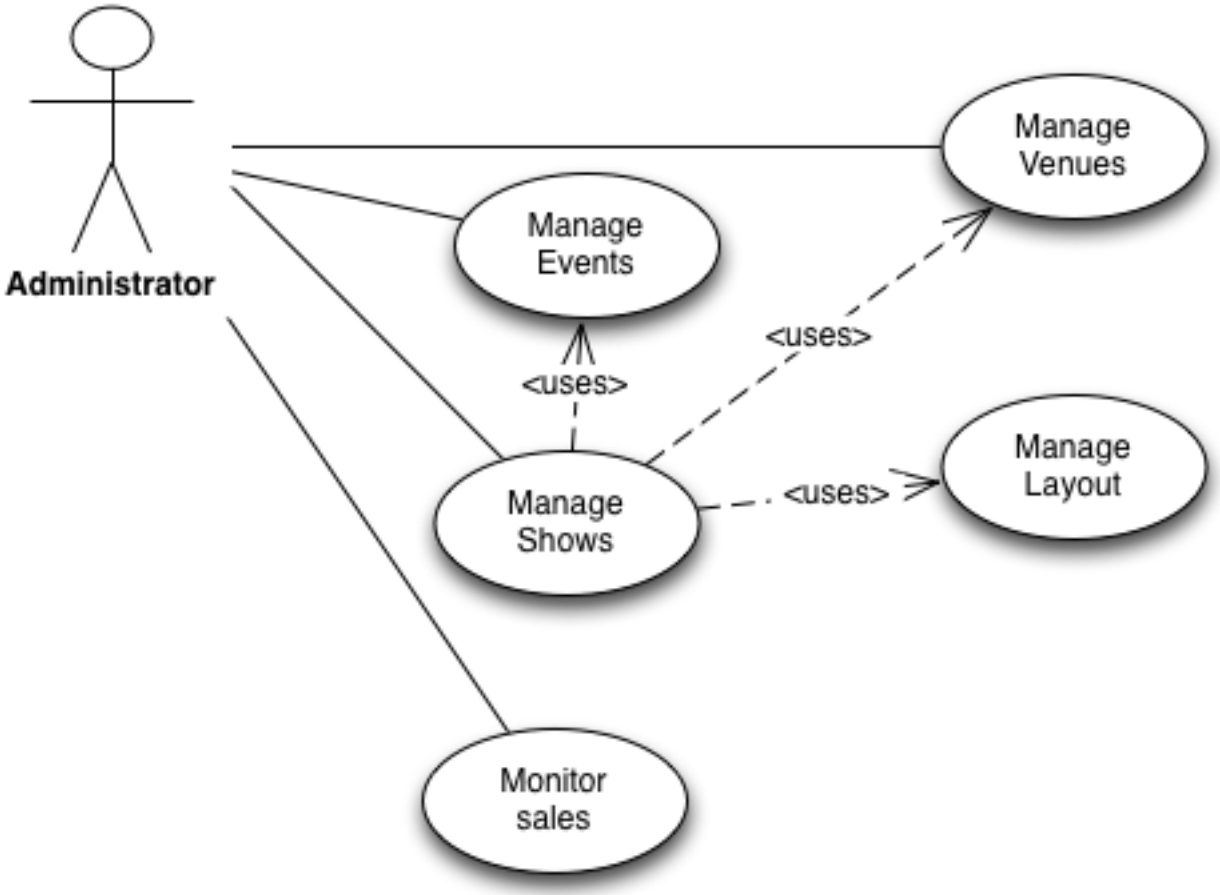

Figure 2: Administration use cases

Architecture

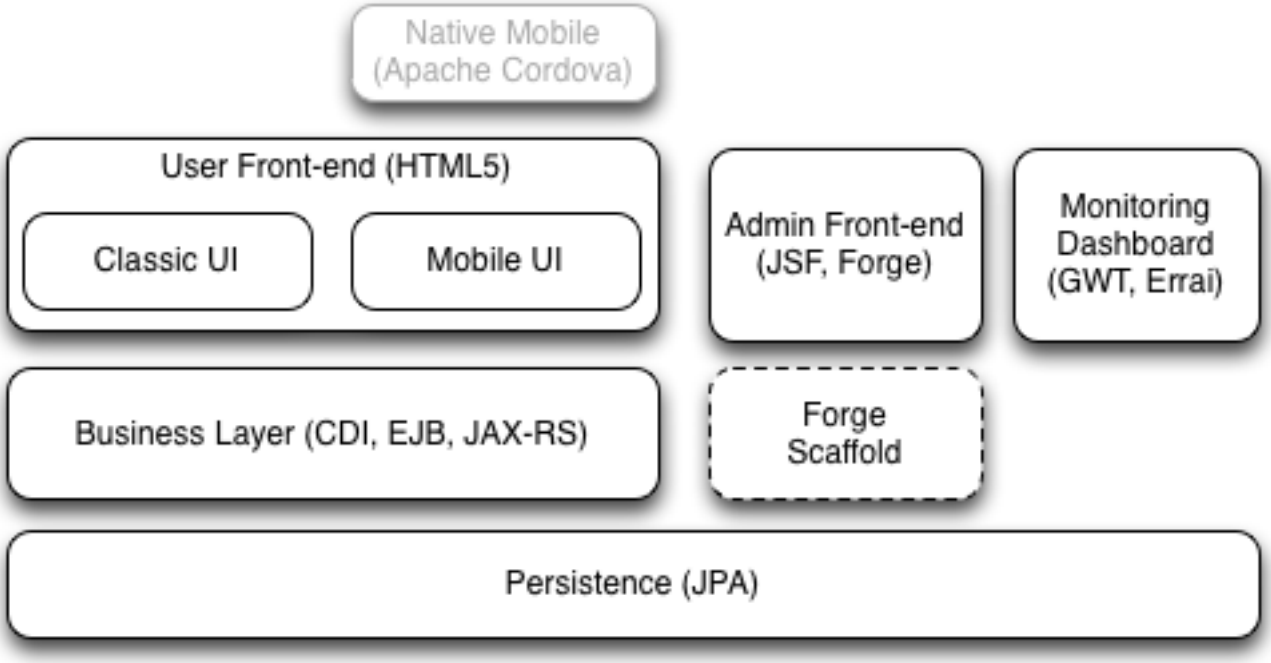

Figure 3: TicketMonster architecture

The application uses Java EE 6 services to provide business logic and persistence, utilizing technologies such as CDI, EJB 3.1 and JAX-RS, JPA 2. These services back the user-facing booking process, which is implemented using HTML5 and JavaScript, with support for mobile devices through jQuery Mobile.

The administration site is centered around CRUD use cases, so instead of writing everything manually, the business layer and UI are generated by Forge, using EJB 3.1, CDI and JSF. For a better user experience, Richfaces UI components are used.

Monitoring sales requires staying in touch with the latest changes on the server side, so this part of the application will be developed in GWT and showcases Errai's support for real-time updates via client-server CDI eventing.

How can you run it?

Before building and running TicketMonster, you must generate the administration site with Forge. See the tutorial for details.

Building TicketMonster

TicketMonster can be built from Maven, by runnning the following Maven command:

```
mvn clean package
```

If you want to run the Arquillian tests as part of the build, you can enable one of the two available Arquillian profiles.

For running the tests in an *already running* application server instance, use the `arq-jbossas-remote` profile.

```
mvn clean package -Parq-jbossas-remote
```

If you want the test runner to *start* an application server instance, use the `arq-jbossas-managed` profile. You must set up the `JBOSS_HOME` property to point to the server location, or update the `src/main/test/resources/arquillian.xml` file.

```
mvn clean package -Parq-jbossas-managed
```

If you intend to deploy into OpenShift, you can use the `postgresql-openshift` profile:

```
mvn clean package -Ppostgresql-openshift
```

Running TicketMonster

You can run TicketMonster into a local JBoss AS7 instance or on OpenShift.

Running TicketMonster locally

First, *start JBoss Enterprise Application Platform 6 or JBoss AS 7 with the Web Profile*.

1. Open a command line and navigate to the root of the JBoss server directory.
2. The following shows the command line to start the server with the web profile:

   ```
   For Linux:   JBOSS_HOME/bin/standalone.sh
   For Windows: JBOSS_HOME\bin\standalone.bat
   ```

Then, *deploy TicketMonster*.

1. Make sure you have started the JBoss Server as described above.
2. Type this command to build and deploy the archive into a running server instance.

   ```
   mvn clean package jboss-as:deploy

   (You can use the 'arq-jbossas-remote' profile for running tests as well)
   ```

3. This will deploy `target/ticket-monster.war` to the running instance of the server.
4. Now you can see the application running at http://localhost:8080/ticket-monster.

Running TicketMonster in OpenShift

First, *create an OpenShift project*.

1. Make sure that you have an OpenShift domain and you have created an application using the `jbossas-7` cartridge (for more details, get started [here](https://openshift.redhat.com/app/getting_started)). If you want to use PostgreSQL, add the `postgresql-8.4` cartridge too.
2. Ensure that the Git repository of the project is checked out.

Then, *build and deploy it*.

1. Build TicketMonster using either:

   - the default profile (with H2 database support)

```
mvn clean package
```

- the `postgresql-openshift` profile (with PostgreSQL support) if the PostgreSQL cartrdige is enabled in OpenShift.

```
mvn clean package -Ppostgresql-openshift
```

2. Copy the `target/ticket-monster.war`file in the OpenShift Git repository(located at `<root

```
cp target/ticket-monster.war <root-of-openshift-application-git-repository>/ ↩
   deployments/ROOT.war
```

3. Navigate to `<root-of-openshift-application-git-repository>` folder
4. Remove the existing `src` folder and `pom.xml` file.

```
git rm -r src
        git rm pom.xml
```

5. Add the copied file to the repository, commit and push to Openshift

```
git add deployments/ROOT.war
git commit -m "Deploy TicketMonster"
git push
```

6. Now you can see the application running at at `http://<app-name>-<domain-name>.rhcloud.com`

Learn more

The example is accompanied by a series of tutorials that will walk you through the process of creating the TicketMonster application from end to end.

After reading this series you will understand how to:

- set up your project;
- define the persistence layer of the application;
- design and implement the business layer and expose it to the front-end via RESTful endpoints;
- implement a mobile-ready front-end using HTML 5, JSON, JavaScript and jQuery Mobile;
- develop a JSF-based administration interface rapidly using JSF and JBoss Forge;
- thoroughly test your project using JUnit and Arquillian;

Throughout the series, you will be shown how to achieve these goals using JBoss Developer Studio.

Part I

Introduction & Getting Started

Chapter 1

Purpose and Target Audience

The target audience for this tutorial are those individuals who do not yet have a great deal of experience with:

- Eclipse + JBoss Tools (JBoss Developer Studio)
- JBoss Enterprise Application 6 or JBoss AS 7
- Java EE 6 features like JAX-RS
- HTML5 & jQuery for building an mobile web front-end.

This tutorial sets the stage for the creation of TicketMonster - our sample application that illustrates how to bring together the best features of **Java EE 6 + HTML5 + JBoss** to create a rich, mobile-optimized and dynamic application.

TicketMonster is developed as an open source application, and you can find it at github.

If you prefer to watch instead of read, a large portion of this content is also covered in video form.

In this tutorial, we will cover the following topics:

- Working with JBoss Developer Studio (Eclipse + JBoss Tools)
- Creating of a Java EE 6 project via a Maven archetype
- Leveraging m2e and m2e-wtp
- Using Forge to create a JPA entity
- Using Hibernate Tools
- Database Schema Generation
- Deployment to a local JBoss Server
- Adding a JAX-RS endpoint
- Adding a jQuery Mobile client
- Using the Mobile BrowserSim

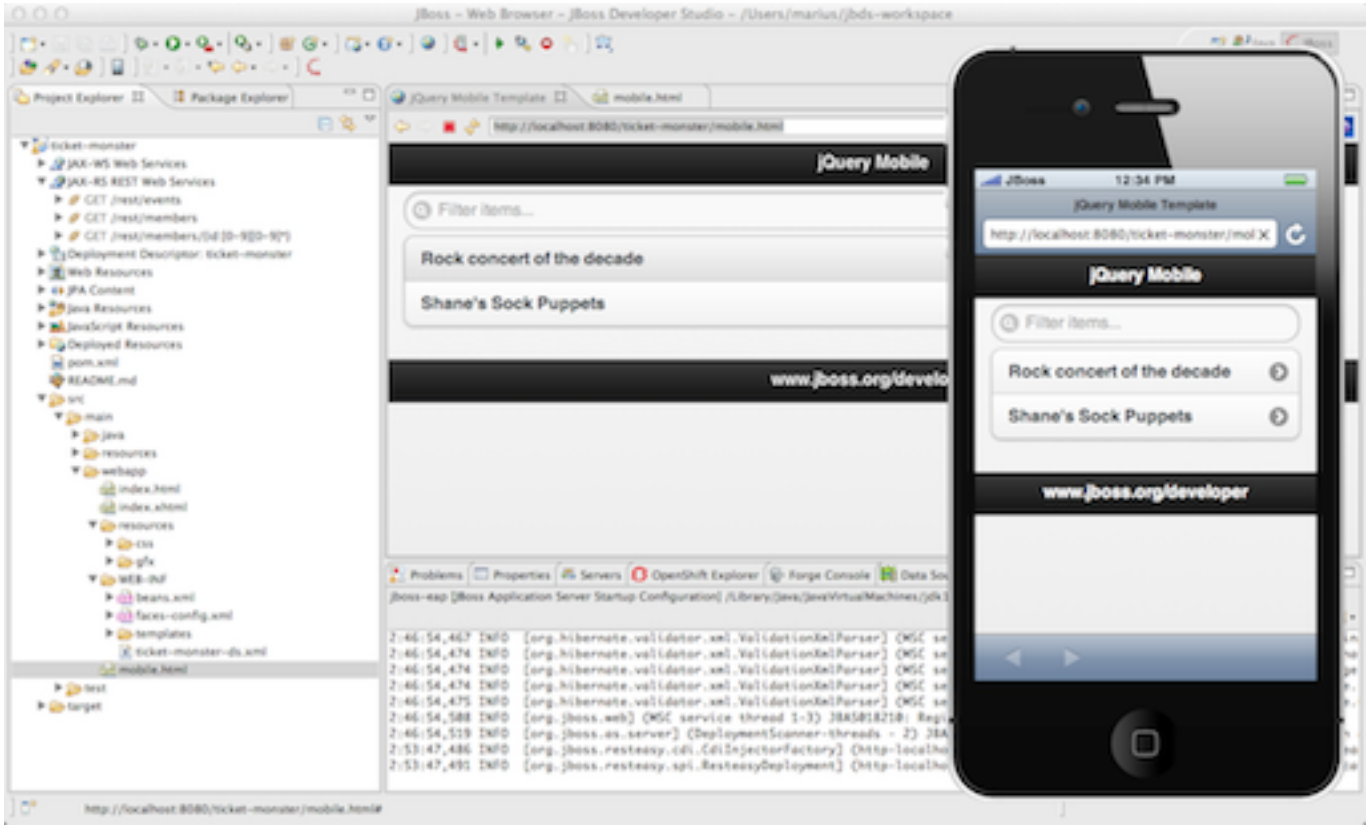

Figure 1.1: JBoss Developer Studio 5 with Mobile BrowserSim

Chapter 2

Installation

The first order of business is to get your development environment setup and JBoss Developer Studio v5 installed. JBoss Developer Studio is Eclipse Indigo (3.7.2) for Java EE Developers plus select JBoss Tools and is available for free. Visit http://www.jboss.org/tools/download/ to download it. You may also choose to install JBoss Tools 3.3 into your existing Eclipse Indigo for Java EE Developers. This document uses screenshots depicting JBoss Developer Studio.

You must have a Java Development Kit (JDK) installed, either v6 or v7 will work - whilst a JVM runtime will work for most use cases, for a developer environment it is normally best to have the full JDK. System requirements for JBoss Developer Studio are listed in the System Requirements chapter of the JBoss Developer Studio 5.0 Getting Started Guide online documentation.

Tip

If you prefer to see JBoss Developer studio being installed, then check out this video.
To see JBoss Tools being intalled into Eclipse Indigo, see this video.

The JBoss Developer Studio installer has a (very long!) name such as `jbdevstudio-product-universal-5.0.0.v20120430` where the latter portion of the file name relates to build date and version information and the text near the front related to the target operating system. The "universal" installer is for any operating system. To launch the installer you may simply be able to double-click on the .jar file name or you may need to issue the following from the operating system command line:

```
java -jar jbdevstudio-product-universal-5.0.0.v201204301418M-H119.jar
```

We recommend using the "universal" installer as it handles Windows, Mac OS X and Linux - 32-bit and 64-bit versions.

Note

Even if you are installing on a 64-bit OS, you may still wish to use the 32-bit JVM for the JBoss Developer Studio (or Eclipse + JBoss Tools). Only the 32-bit version provides the Visual Page Editor - a split-pane editor that gives you a glimpse of what your HTML/XHTML (JSF, JSP, etc) will look like. Also, the 32-bit version uses less memory than the 64-bit version. You may still run your application server in 64-bit JVMs if needed to insure compatibility with the production environment whilst keeping your IDE in 32-bit mode.

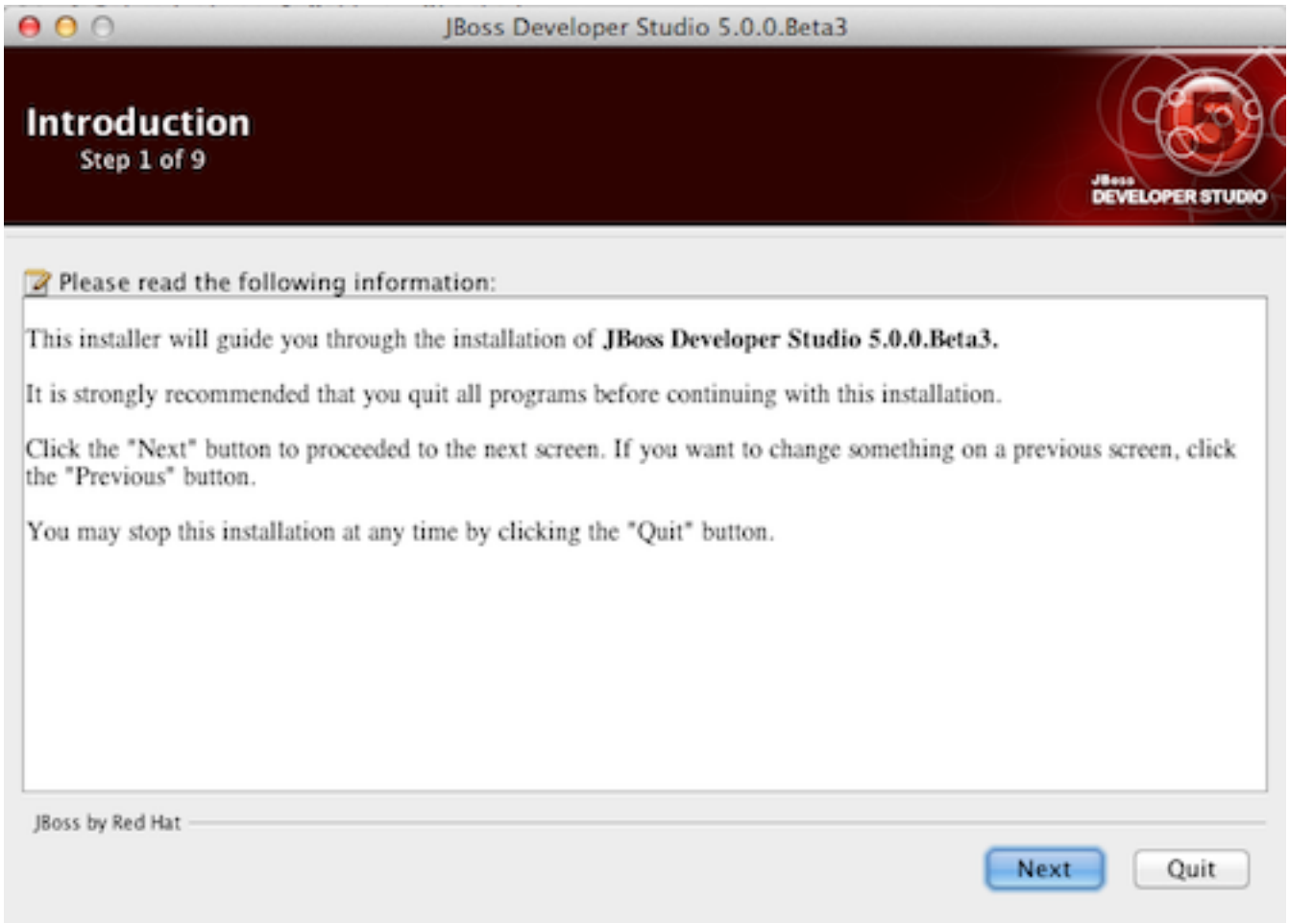

Figure 2.1: Installation Wizard, Step 1 of 9

The rest of the steps are fairly self explanatory. If you run into trouble, please consult the videos above as they explore a few troubleshooting tips related to JRE/JDK setup.

Please make sure to say **Yes** to the prompt that says "Will you allow JBoss Tools team to receive anonymous usage statistics for this Eclipse instance with JBoss Tools?". This information is very helpful to us when it comes to prioritizing our QA efforts in terms of operating system platforms. More information concerning our usage tracking can be found at http://www.jboss.org/-tools/usage

You can skip the step in the installation wizard that allows you to install JBoss Enterprise Application Platform 6 or JBoss AS 7 as we will do this in the next step.

Chapter 3

Creating a new Java EE 6 project with Maven

Tip
For a deeper dive into the world of Maven and how it is used with JBoss Developer Studio 5 and JBoss Enterprise Application Platform 6 (or JBoss Tools 3.3 and JBoss AS 7) review link: this video.

Now that everything is properly installed, configured, running and verified to work, let's build something "from scratch".

We recommend that you switch to the JBoss Perspective if you have not already.

Tip
If you close JBoss Central, it is only a click away - simply click on the JBoss icon in the Eclipse toolbar - it is normally the last icon, on the last row - assuming you are in the JBoss Perspective.

First, select **Create Projects** → **Java EE Web Project** in JBoss Central. Under the covers, this uses a Maven archetype which creates a Java EE 6 web application (.war), based around Maven. The project can be built outside of the IDE, and in continuous integration solutions like Hudson/Jenkins.

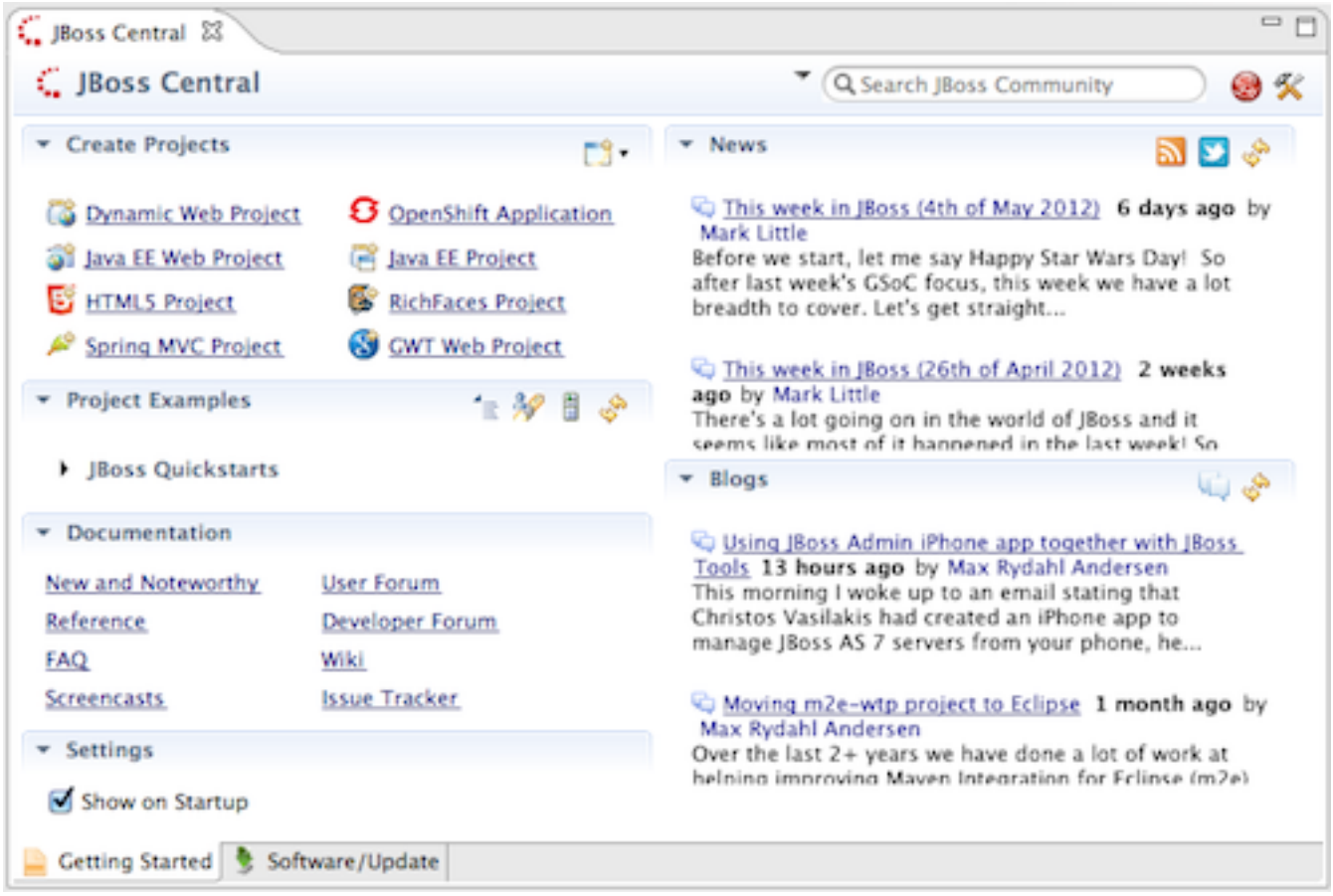

Figure 3.1: JBoss Central

You will be prompted with a dialog box that verifies that JBoss Developer Studio is configured correctly. If you are in a brand new workspace, the application server will not be configured yet and you will notice the lack of a check mark on the server/runtime row.

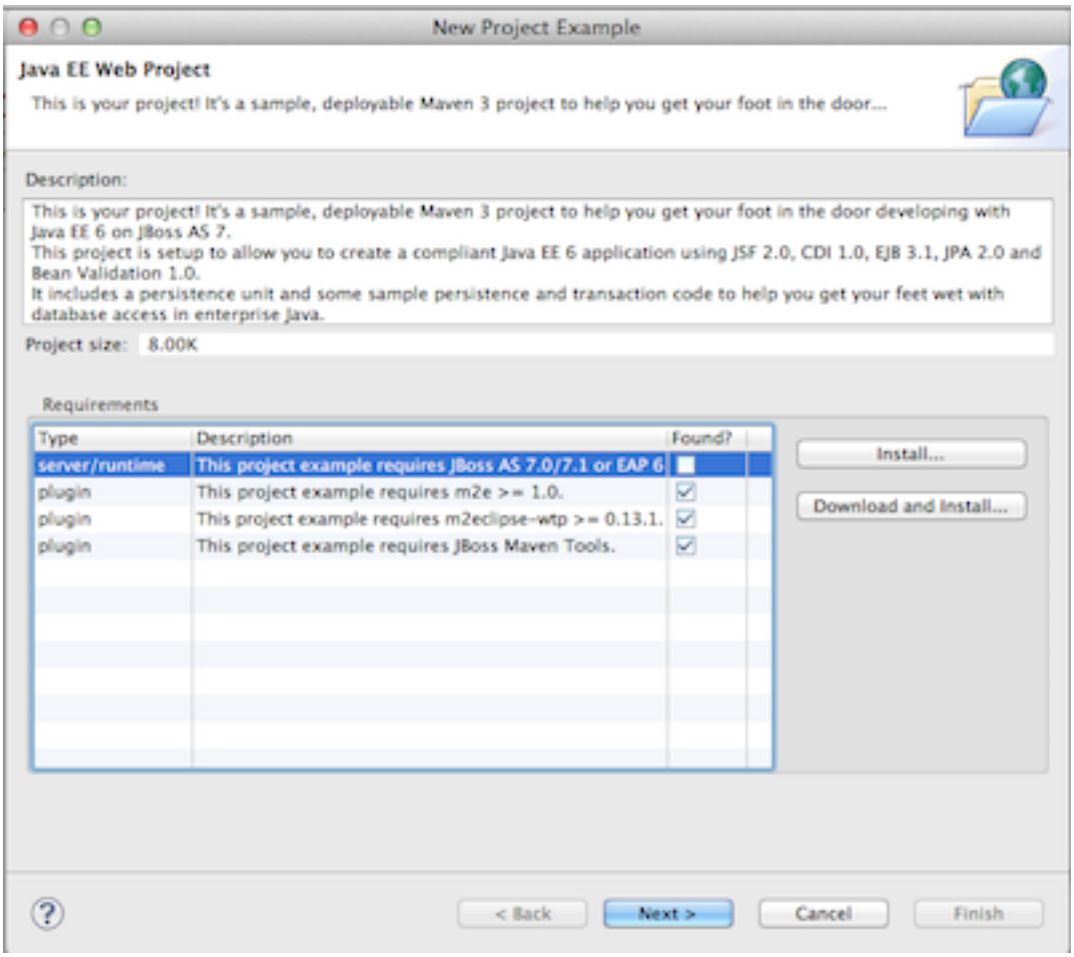

Figure 3.2: New Project Wizard

Note
There are several ways to add JBoss Enterprise Application Platform 6 or JBoss AS 7 to JBoss Developer Studio. The **Install...** button on the new project wizard is probably the easiest, but you can use any of the methods you are familiar with!

To add JBoss Enterprise Application Platform or JBoss AS 7, click on the **Install...** button, or if you have not yet downloaded and unzipped the server, click on the **Download and Install...** button.

Caution
The download option only works with the community application server.

Selecting **Install...** will pop up the JBoss Runtime Detection section of Preferences. You can always get back to this dialog by selecting **Preferences** → **JBoss Tools** → **JBoss Tools Runtime Detection**.

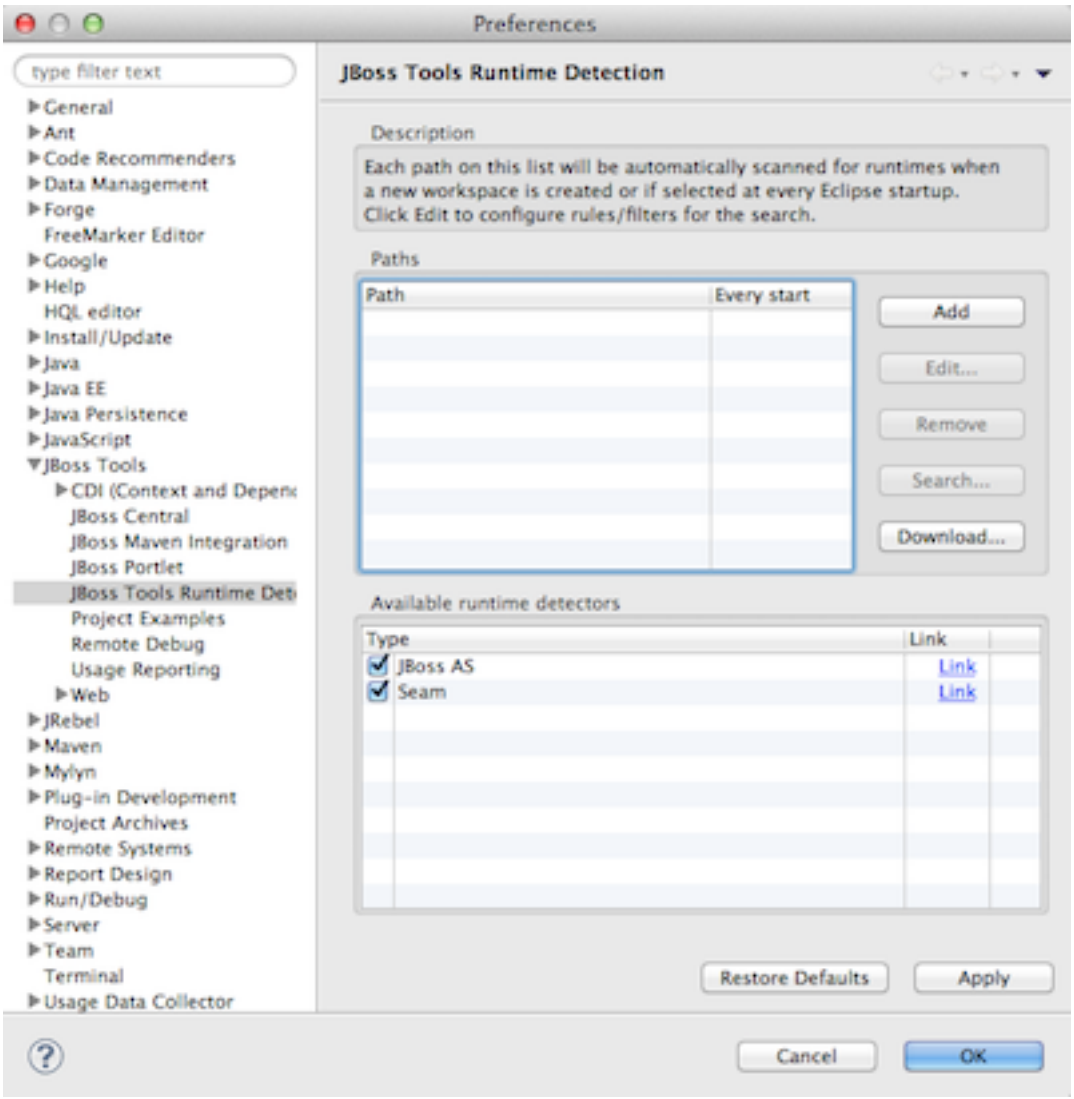

Figure 3.3: JBoss Tools Runtime Detection

Select the **Add** button which will take you to a file browser dialog where you should locate your unzipped JBoss server.

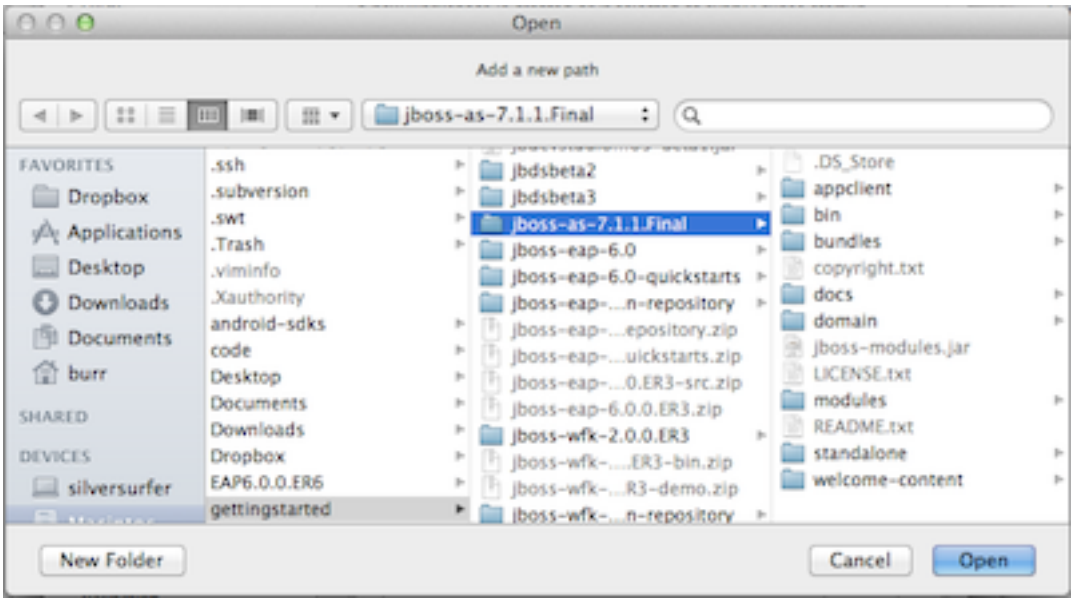

Figure 3.4: Runtime Open Dialog

Select **Open** and JBoss Developer Studio will pop up the **Searching for runtimes…** window.

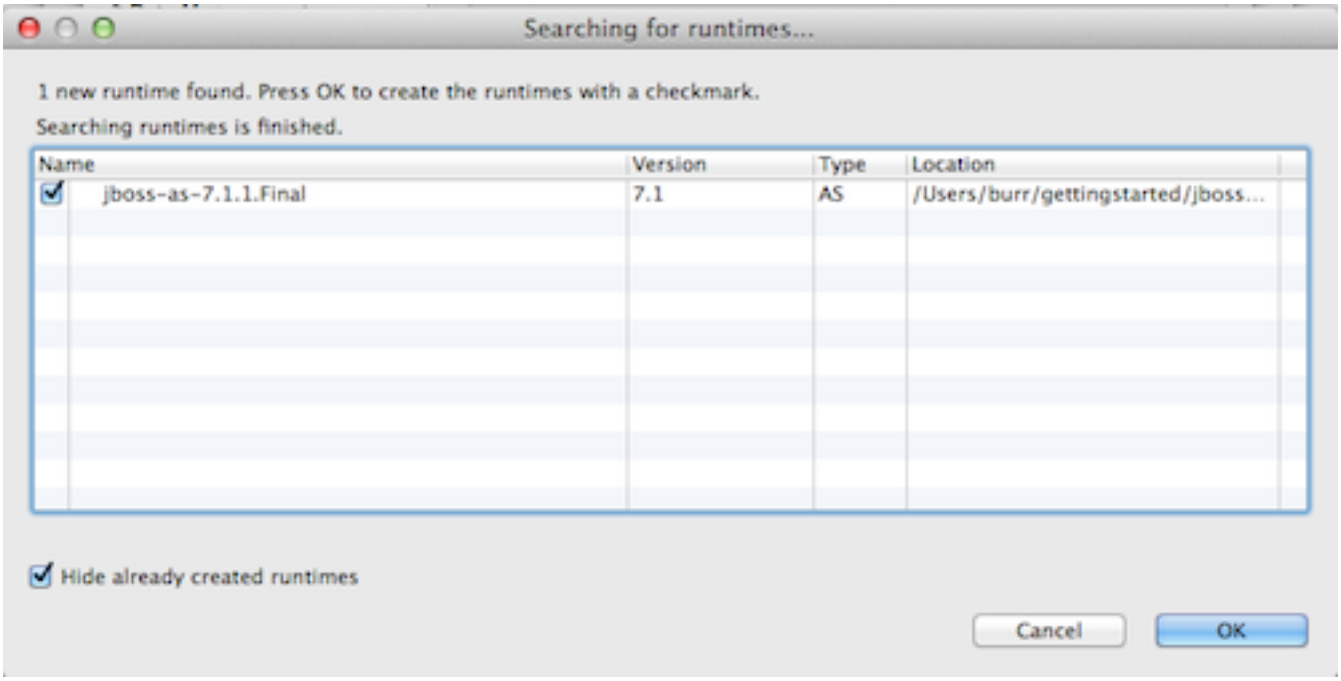

Figure 3.5: Searching for runtimes window

Simply select **OK**. You should see the added runtime in the Paths list.

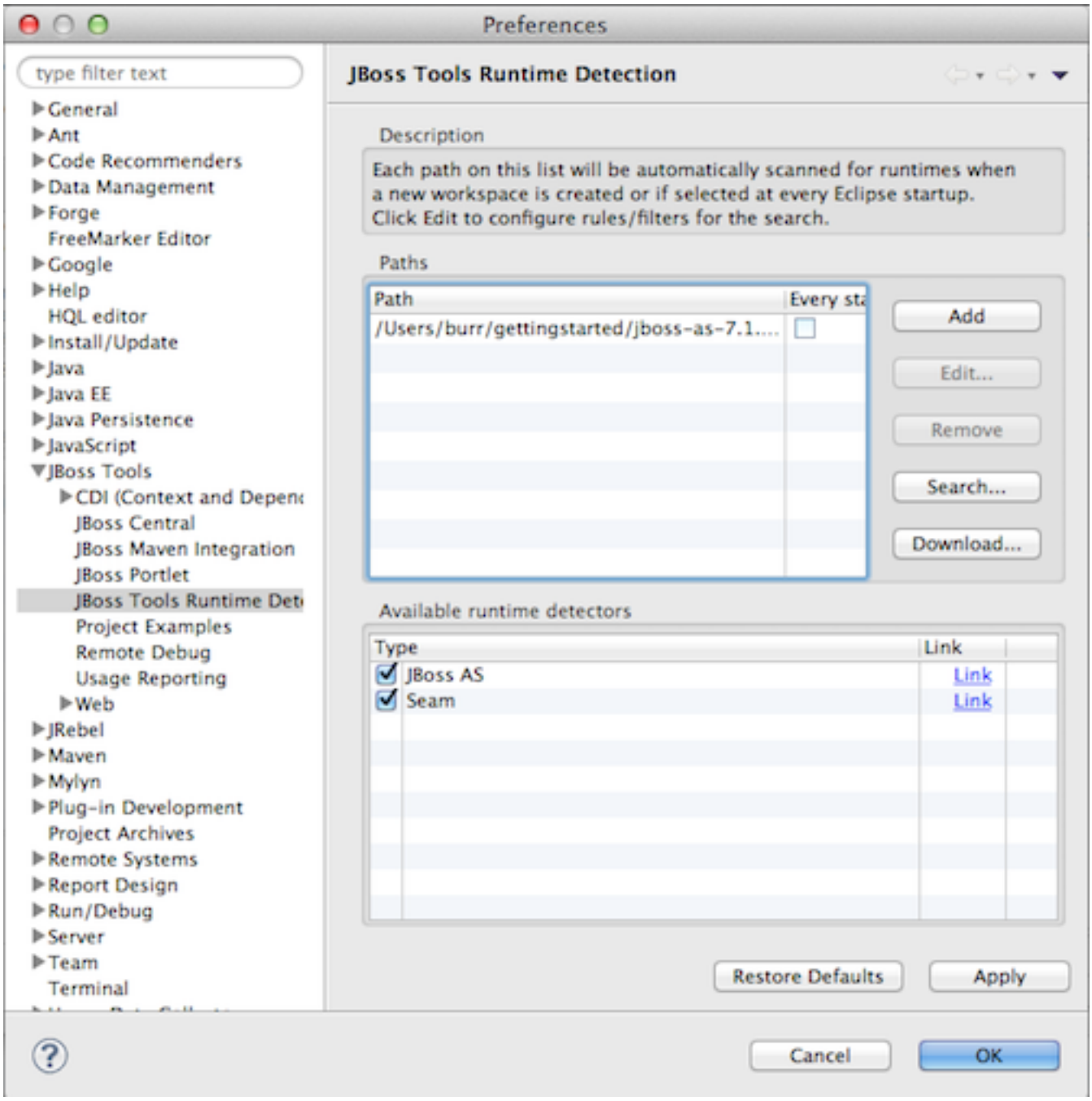

Figure 3.6: JBoss Tools Runtime Detection Completed

Select **OK** to close the **Preferences** dialog, and you will be returned to the **New Project Example** dialog, with the the server/runtime found.

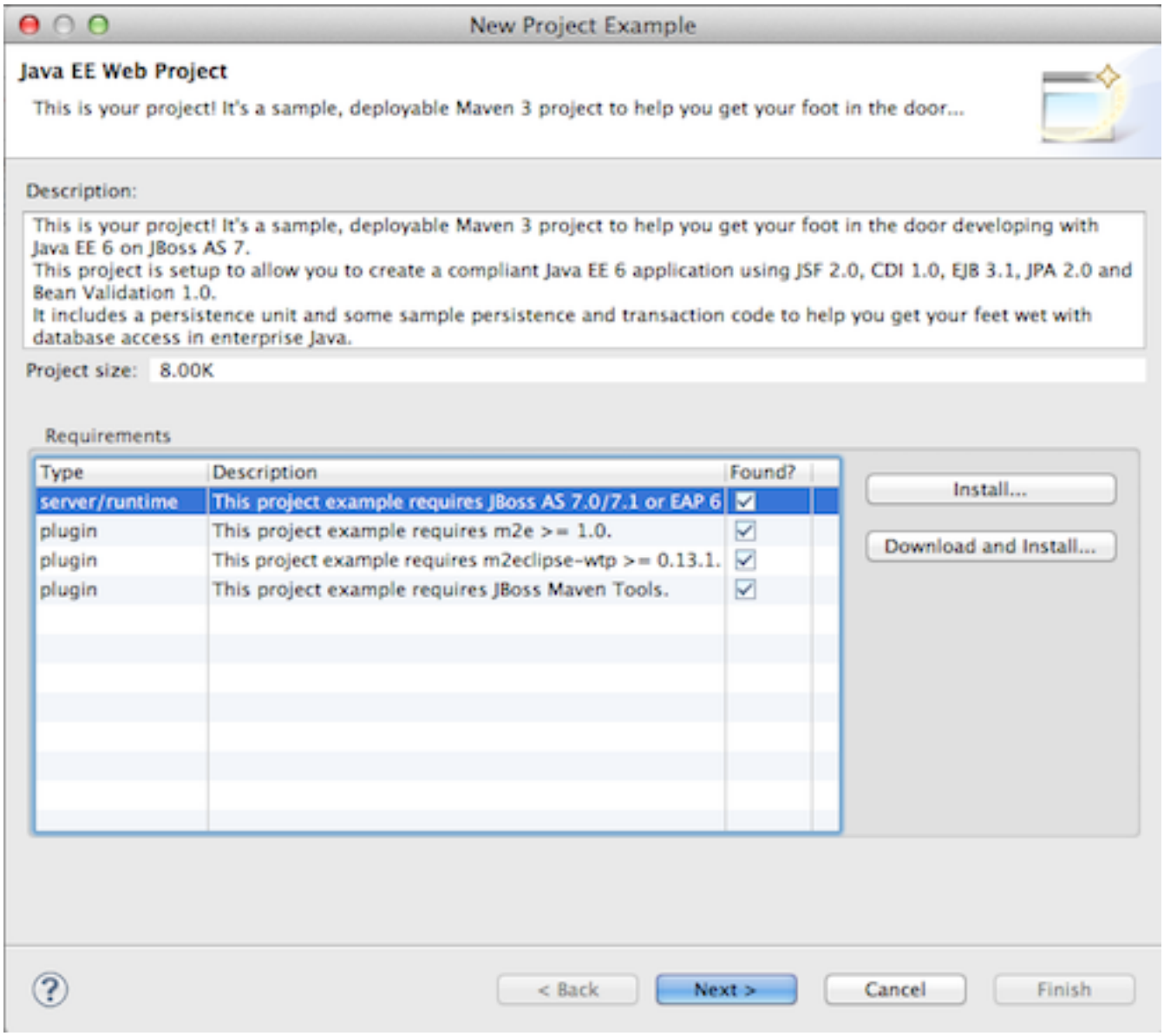

Figure 3.7: JBoss AS 7.0/7.1 or EAP 6 Found

Select **Next**.

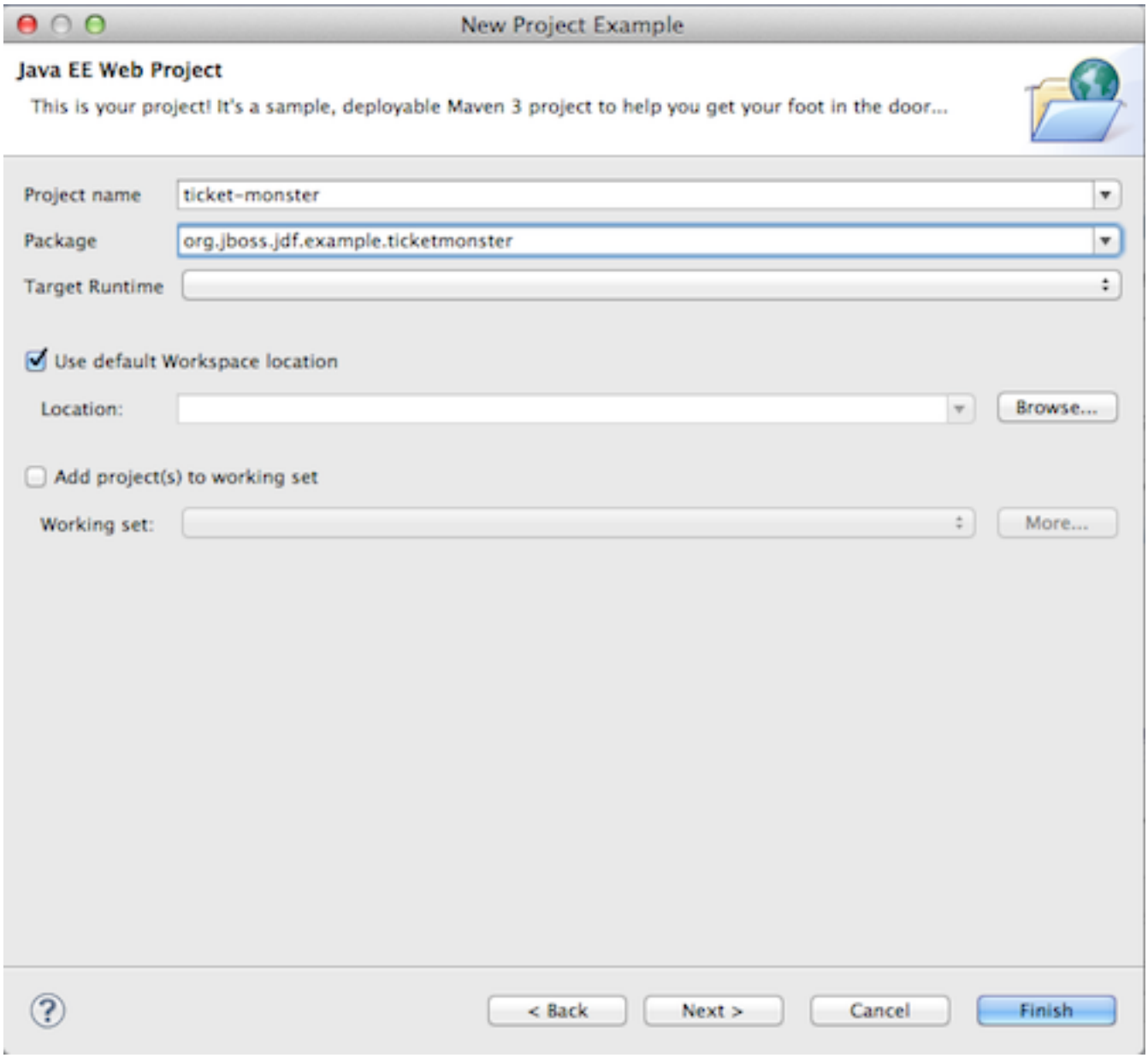

Figure 3.8: New Project Wizard Step 2

The default **Project name** is `jboss-javaee6-webapp`. If this field appears blank, it is because your workspace already contains a "jboss-javaee6-webapp" in which case just provide another name for your project. Change the project name to `ticket-monster`, and the package name to `org.jboss.jdf.example.ticketmonster`.

The **Target Runtime** allows you to choose between JBoss Enterprise Application Platform and JBoss AS 7. If it is left empty, JBoss AS 7 will be elected.

Select **Finish**.

JBoss Tools/JBoss Developer Studio will now generate the template project and import it into the workspace. You will see it pop up into the Project Explorer and a message that asks if you would like to review the readme file.

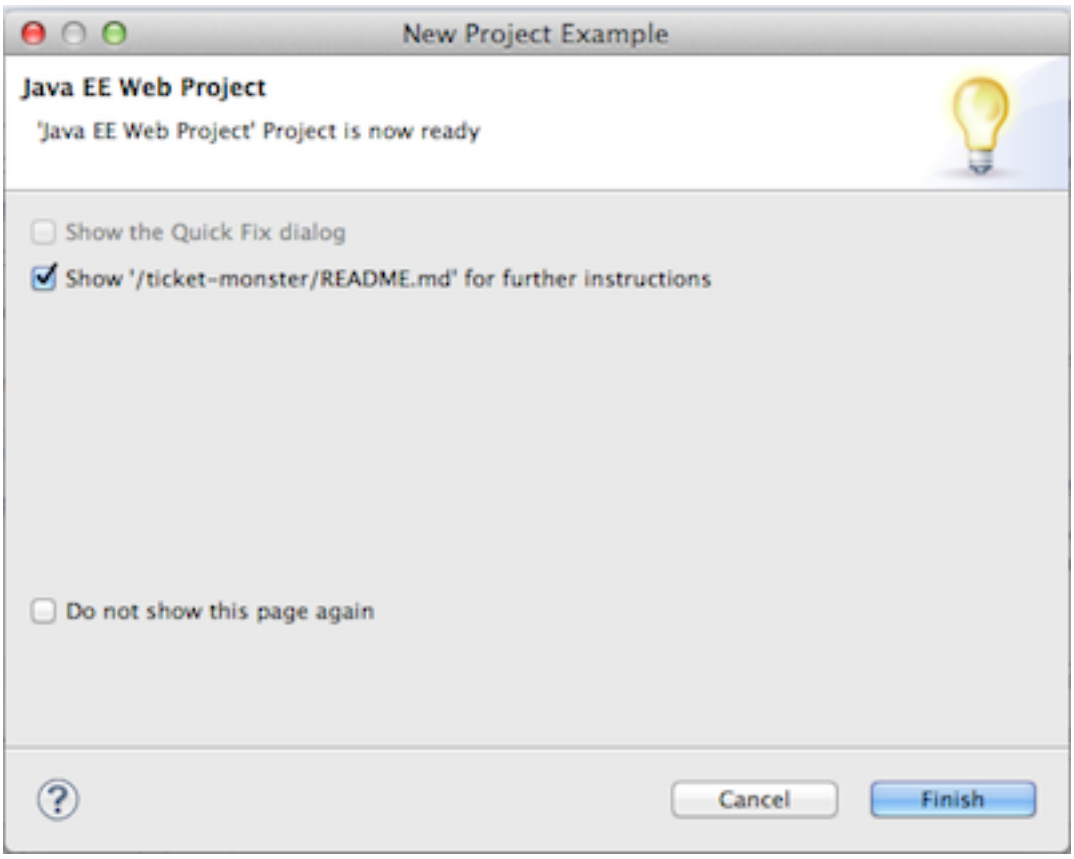

Figure 3.9: New Project Wizard Step 3

Select **Finish**

Chapter 4

Exploring the newly generated project

Using the **Project Explorer**, open up the generated project, and double-click on the `pom.xml`.

The generated project is a Maven-based project with a `pom.xml` in its root directory.

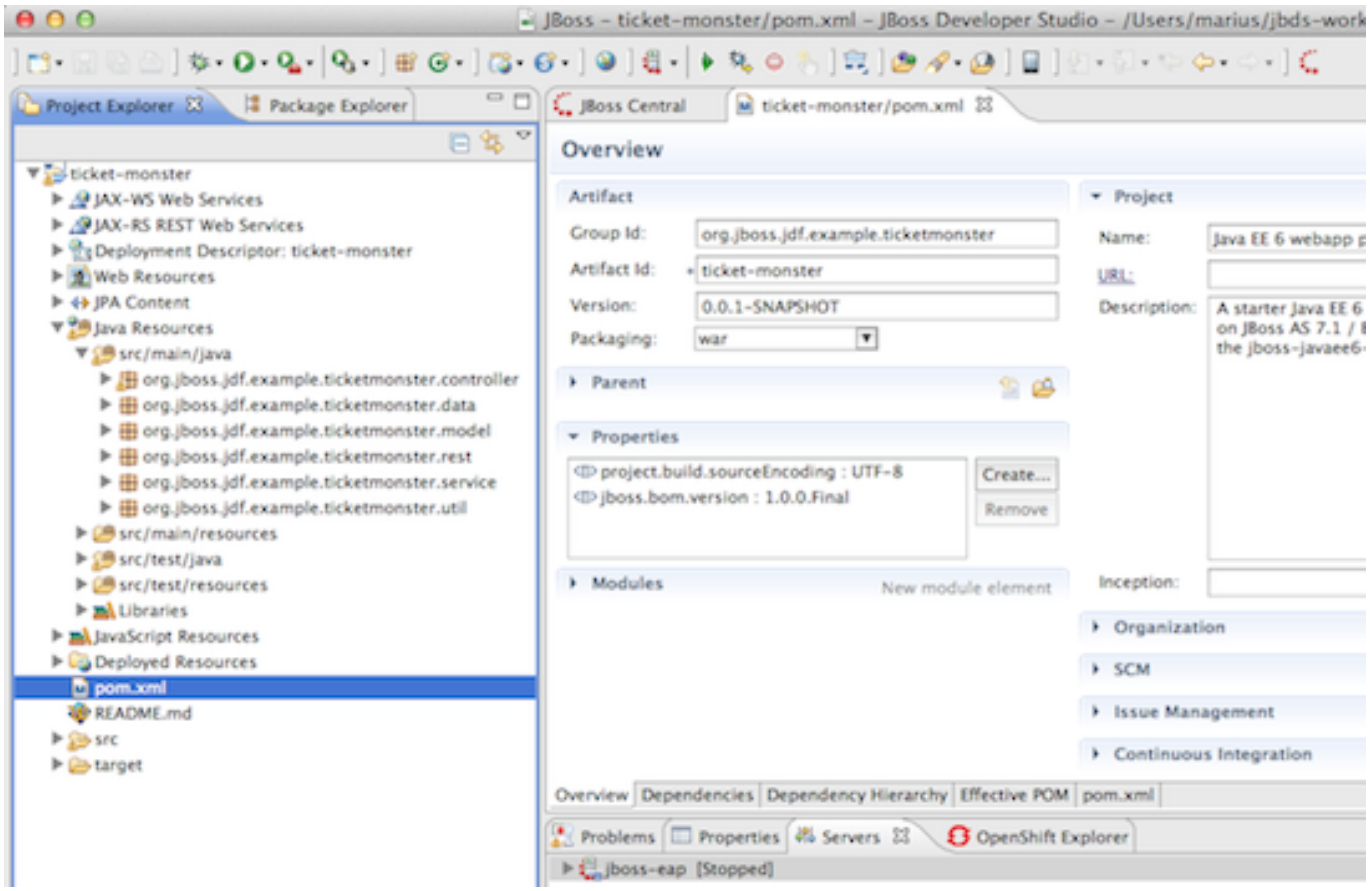

Figure 4.1: Project Explorer

JBoss Developer Studio and JBoss Tools include m2e and m2e-wtp. m2e is the Maven Eclipse plug-in and provides a graphical editor for editing `pom.xml` files, along with the ability to run maven goals directly from within Eclipse. m2e-wtp allows you to deploy your Maven-based project directly to any Web Tools Project (WTP) compliant application server. This means you can drag & drop, use **Run As** → **Run on Server** and other mechanisms to have the IDE deploy your application.

The `pom.xml` editor has several tabs along its bottom edge.

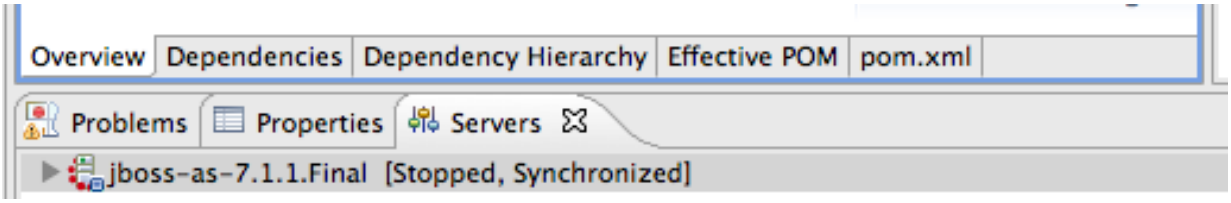

Figure 4.2: pom.xml Editor Tabs

For this tutorial, we do not need to edit the `pom.xml` as it already provides the Java EE 6 APIs that we will need (e.g. JPA, JAX-RS, CDI). You should spend some time exploring the **Dependencies** and the **pom.xml** (source view) tabs.

One key element to make note of is `<jboss.bom.version>1.0.0.M6</jboss.bom.version>` which establishes if this project uses JBoss Enterprise Application Platform or JBoss AS dependencies. The BOM (Bill of Materials) specifies the versions of the Java EE (and other) APIs defined in the dependency section.

If you are using JBoss Enterprise Application Platform 6 and you selected that as your Target Runtime, you will find a `-redhat-1` suffix on the version string.

Caution
The specific version (e.g. `1.0.0.M6`) is very likely to change, so do not be surprised if the version is slightly different.

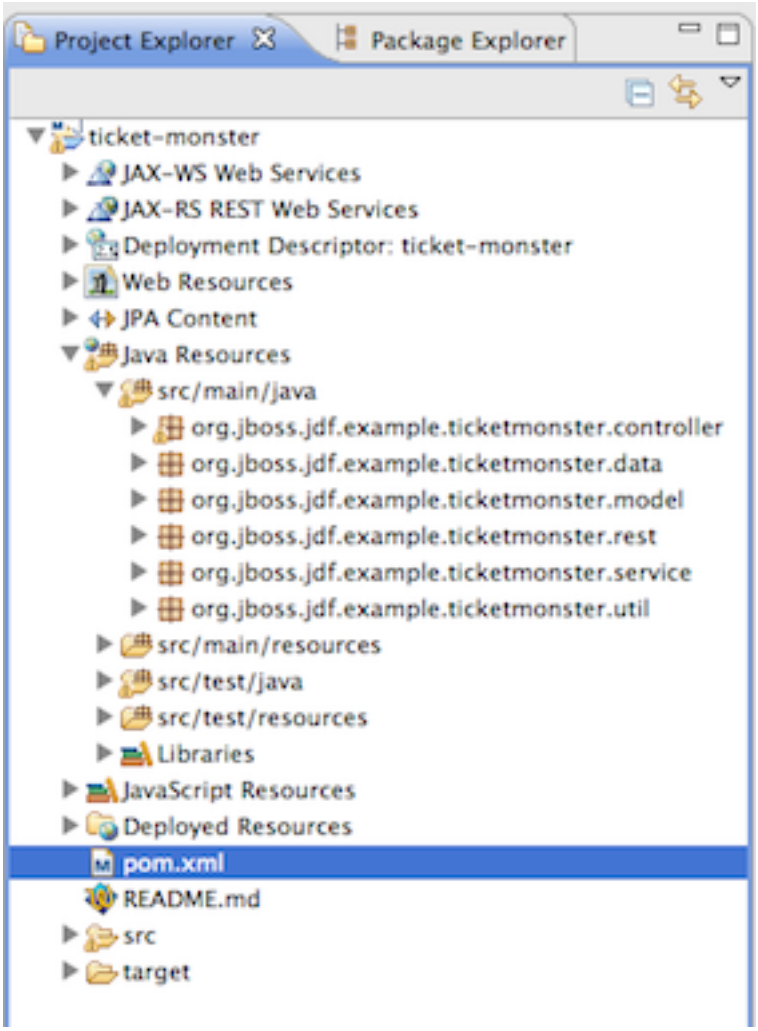

Figure 4.3: Project Explorer Java Packages

Using the **Project Explorer**, drill-down into `src/main/java` under **Java Resources**.

The initial project includes the following Java packages:

`.controller`
: contains the backing beans for `#{newMember}` and `#{memberRegistration}` in the JSF page `index.xhtml`

`.data`
: contains a class which uses `@Produces` and `@Named` to return the list of members for `index.xhtml`

`.model`
: contains the JPA entity class, a POJO annotated with `@Entity`, annodated with Bean Validation (JSR 303) constraints

`.rest`
: contains the JAX-RS endpoints, POJOs annotated with `@Path`

`.service`
: handles the registration transaction for new members

`.util`
: contains Resources.java which sets up an alias for `@PersistenceContext` to be injectable via `@Inject`

Now, let's explore the resources in the project.

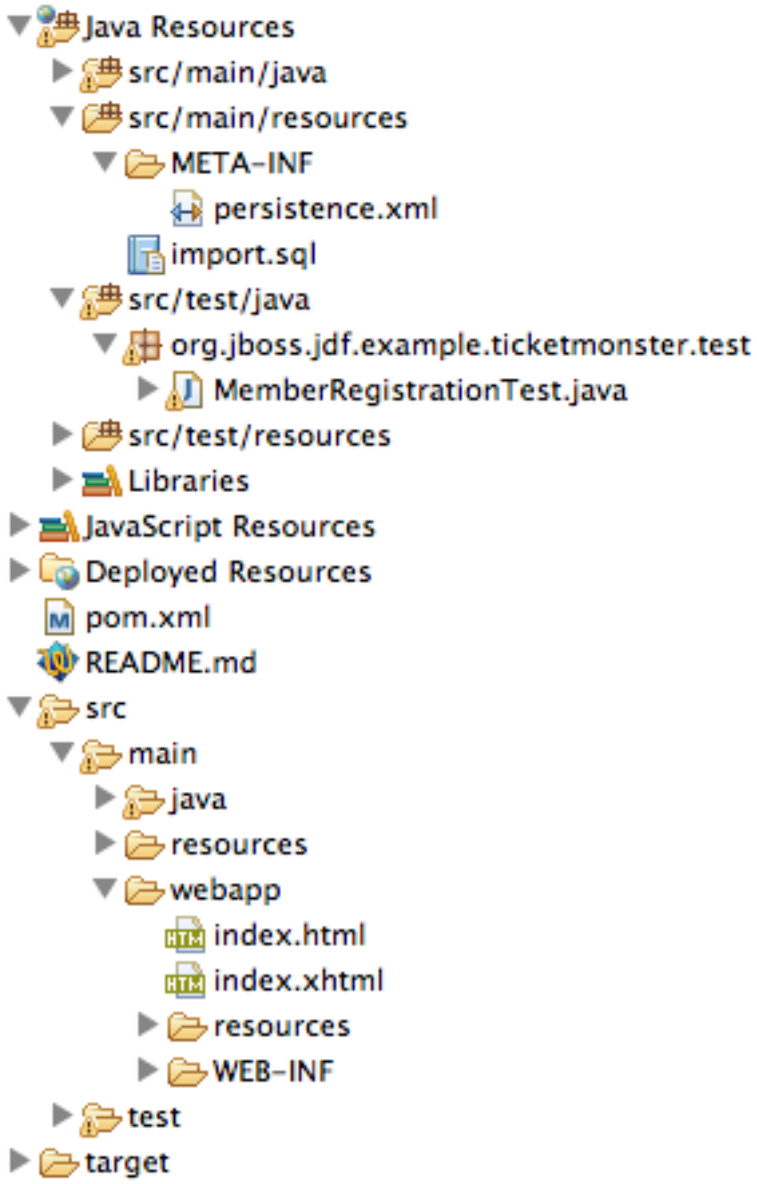

Figure 4.4: Project Explorer Resources

Under src you will find:

`main/resources/import.sql`
: contains insert statements that provides initial database data. This is particularly useful when `hibernate.hbm2dll.auto=cr` is set in `persistence.xml`. `hibernate.hbm2dll.auto=create-drop` causes the schema to be recreated each time the application is deployed.

`main/resources/META-INF/persistence.xml`
: establishes that this project contains JPA entities and it identifies the datasource, which is deployed alongside the project. It also includes the `hibernate.hbm2dll.auto` property set to `create-drop` by default.

`test/java/test`
: provides the `.test` package that contains `MemberRegistrationTest.java`, an Arquillian based test that runs both fomm within JBoss Developer Studio via **Run As** → **JUnit Test** and at the command line:

```
mvn test -Parq-jbossas-remote
```

src/main/webapp
contains `index.xhtml`, the JSF-based user interface for the sample application. If you double-click on that file you will see Visual Page Editor allows you to visually navigate through the file and see the source simultaneously. Changes to the source are immediately reflected in the visual pane.

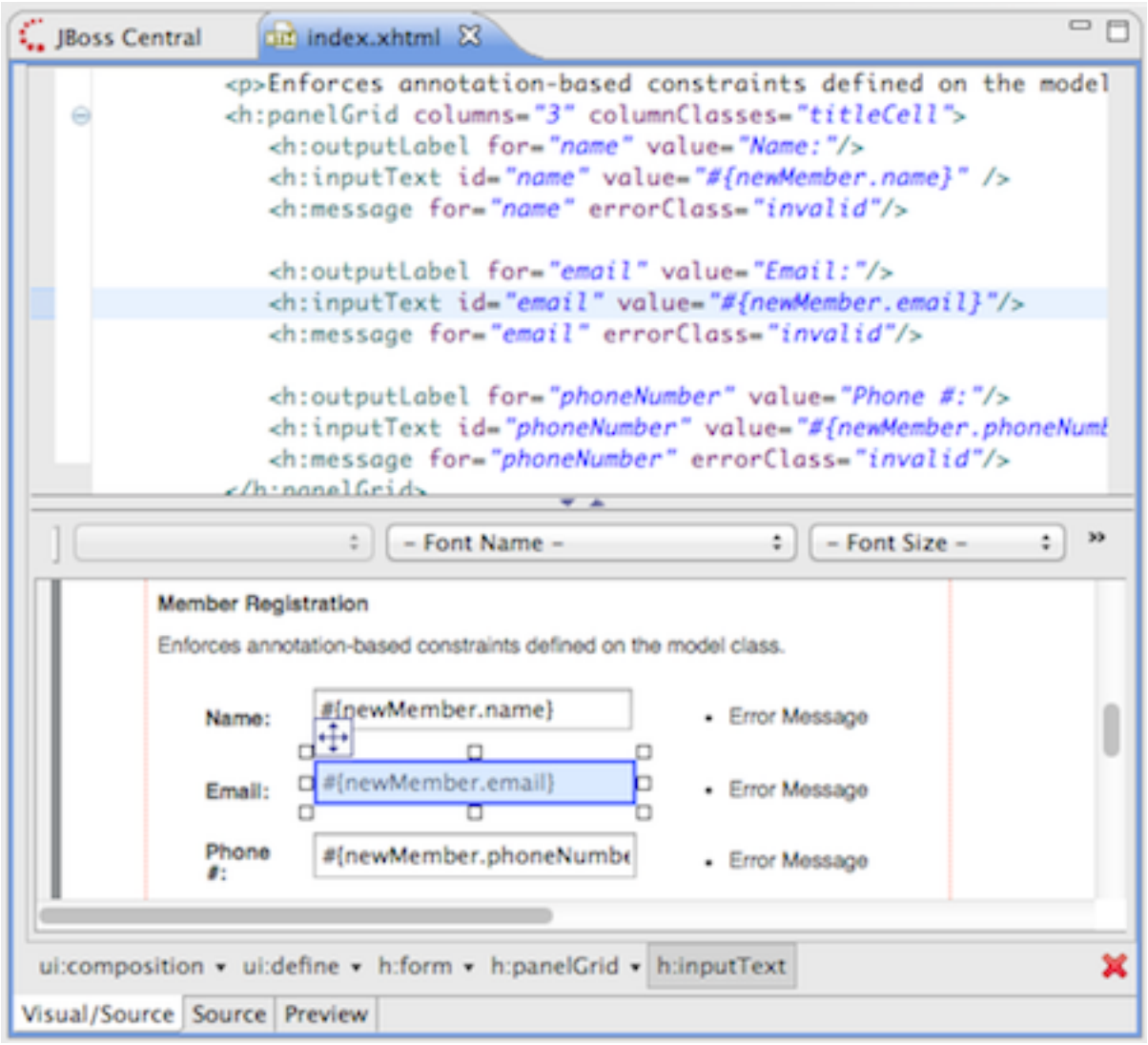

Figure 4.5: Visual Page Editor

In `src/main/webapp/WEB-INF`, you will find three key files:

beans.xml
is an empty file that indicates this is a CDI capable EE6 application

faces-config.xml
is an empty file that indicates this is a JSF capable EE6 application

ticket-monster-ds.xml
when deployed, creates a new datasource within the JBoss container

Chapter 5

Adding a new entity using Forge

There are several ways to add a new JPA entity to your project:

Starting from scratch
Right-click on the `.model` package and select **New** → **Class**. JPA entities are annotated POJOs so starting from a simple class is a common approach.

Reverse Engineering
Right-click on the "model" package and select New → JPA Entities from Tables. For more information on this technique see this video

Using Forge
to create a new entity for your project using a CLI (we will explore this in more detail below)

Reverse Engineering with Forge
Forge has a Hibernate Tools plug-in that allows you to script the conversion of RDBMS schema into JPA entities. For more information on this technique see this video.

For the purposes of this tutorial, we will take advantage of Forge to add a new JPA entity. This requires the least keystrokes, and we do not yet have a RDBMS schema to reverse engineer. There is also an optional section for adding an entity using **New** → **Class**.

Right-click on the `.model` package in the **Project Explorer** and select **Show In** → **Forge Console**.

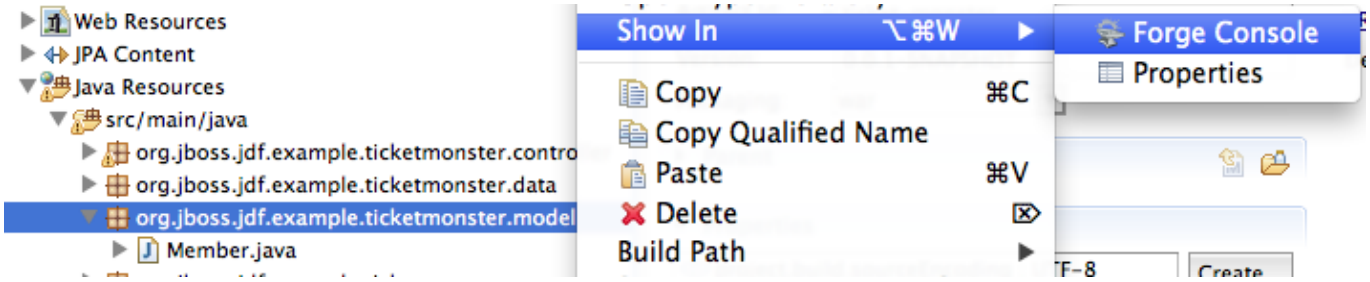

Figure 5.1: Show In Forge Console

Tip
Alternative methods to activate Forge include:

- **Window** → **Show View** → **Forge Console**
- **Ctrl 4** (Windows) or **Cmd 4** (Mac).

Note: the Show In method will issue a "pick-up" command to switch you to the right location within your project.

The first time you start Forge, you will be prompted with a **Forge Not Running** dialog, select **Yes**.

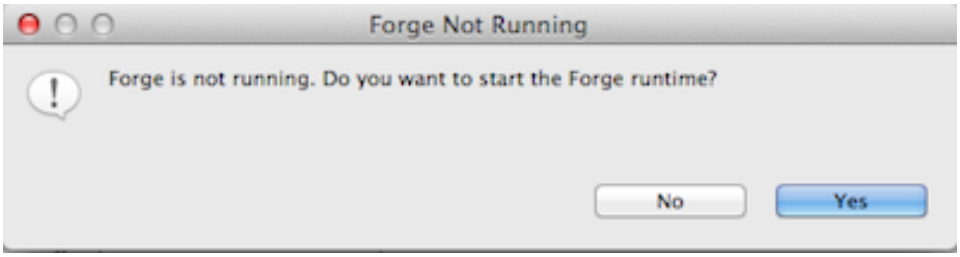

Figure 5.2: Show Forge Not Running

Tip
If you are not prompted you can always start Forge using the green arrow (or stop via the red square) in the Forge Console tab.

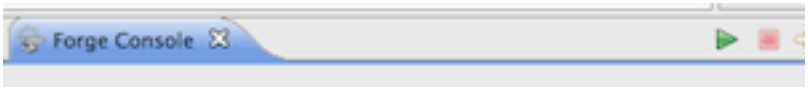

Figure 5.3: Show Forge Start/Stop

Figure 5.4: Show Forge Console

Forge is a command-oriented rapid application development tool that allows you to enter commands that generate classes and code. It will automatically update the IDE for you. A key feature is "content assist" or "tab completion", activated by pressing **tab**.

To generate an entity, use these commands:

```
entity --named Event --package org.jboss.jdf.example.ticketmonster.model
field string --named name
validation setup
constraint NotNull --onProperty name
constraint Size --onProperty name --min 5 --max 50 --message "Must be > 5 and <  ↩
    50"
field string --named description
constraint Size --onProperty description --min 20 --max 1000 --message "Must be >  ↩
    20 and < 1000"
field boolean --named major
```

```
field string --named picture
```

Let's work through this, step by step.

At the `[ticket-monster] model $` prompt, type `en` and hit the tab key on your keyboard. `entity` will fill in. Hit tab again and `entity --named` will appear. Type in `Event` and add a space — Forge can not anticipate the name of your new entity!

Hit tab again and select `--package`. Now, hit tab repeatedly to fill in `org.jboss.jdf.example.ticketmonster`. Since there are multiple entries underneath examples, Forge will display those options. Type in `m` and hit tab to select `model`.

Now hit the Enter/Return key to watch the command execute. The Event entity will be generated into the "model" package and open up inside of Eclipse.

```
[no project] jbds-workspace $ pick-up /Users/marius/jbds-workspace/ticket-monster/src/main/java/
org/jboss/jdf/example/ticketmonster/model
Picked up type <DirectoryResource>: model
[ticket-monster] model $ entity --named Event --package org.jboss.jdf.example.ticketmonster.
controller   data         model        rest         service      util
[ticket-monster] model $ entity --named Event --package org.jboss.jdf.example.ticketmonster.model
Created @Entity [org.jboss.jdf.example.ticketmonster.model.Event]
Picked up type <JavaResource>: org.jboss.jdf.example.ticketmonster.model.Event
Wrote /Users/marius/jbds-workspace/ticket-monster/src/main/java/org/jboss/jdf/example/
ticketmonster/model/Event.java
[ticket-monster] Event.java $
```

Figure 5.5: Forge new entity

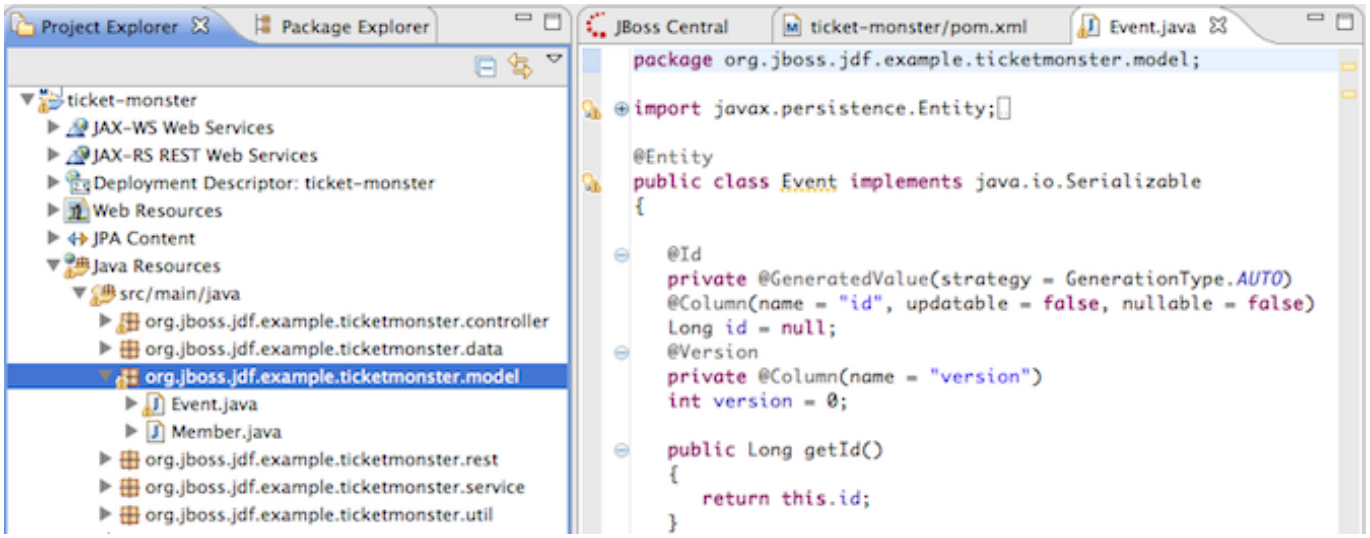

Figure 5.6: Event Entity

Note

`@Entity public class` is placed on the same line as ` import java.lang.Override` by Forge. Using the formatter your IDE provides on the entity will make this look more like you would expect!

Forge has automatically changed the context of the CLI to `Event.java`, and typing `ls` will provide a listing of the fields and methods.

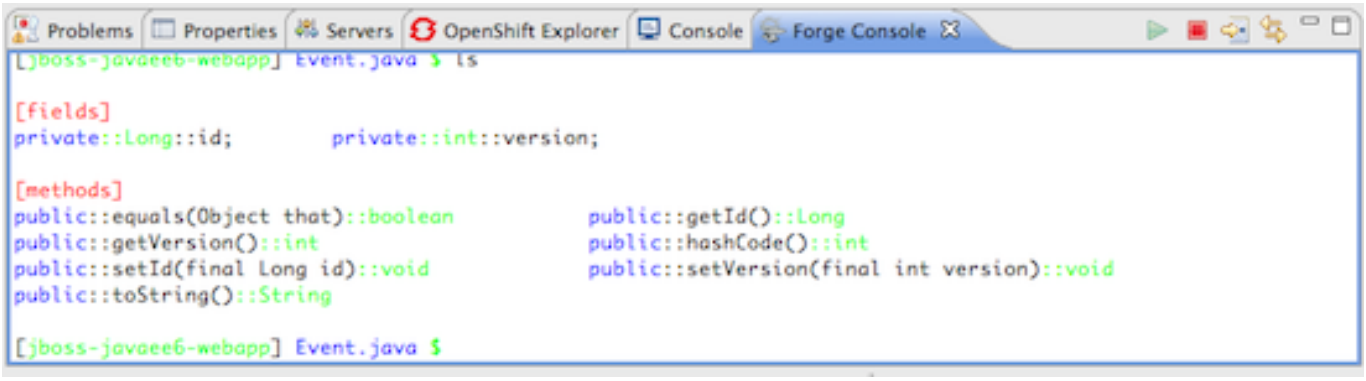

Figure 5.7: Forge `ls`

Now that the base `Event` entity has been created, let's add the fields and their JSR 303 Bean Validation constraints.

This next step involves adding a `name` property for the Event entity so that an event could hold data like "Rock Concert".

Type `fie` and hit tab to fill in `field`, if you hit **tab** again, Forge will list out the possible field types. Type in `s` and hit **tab**, Forge will respond with `string`. Hit **tab** again to get `--named` and type in `name`. You should end up with the command `field string --named name`, to execute it, press enter. This will add a `private String name;` field, and the appropriate accessor and mutator (getter and setter) methods. You should also notice that the toString method is tweaked to include `name` as well.

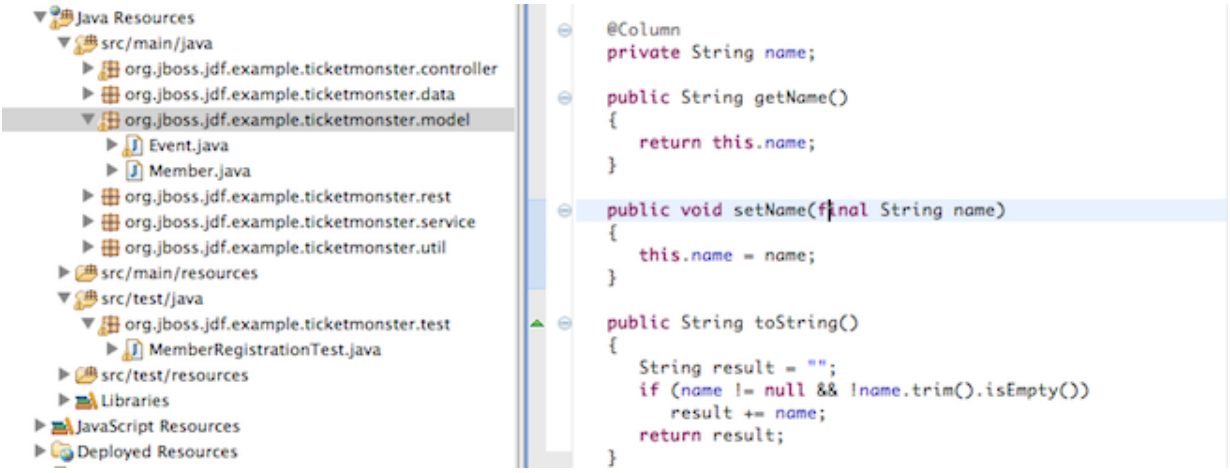

Figure 5.8: @Column name

From this point forward, we will assume you have the basics of using Forge's interactive command line. The remaining commands to run are:

```
validation setup
constraint NotNull --onProperty name
constraint Size --onProperty name --min 5 --max 50 --message "Must be > 5 and <  ↩
    50"
field string --named description
constraint Size --onProperty description --min 20 --max 1000 --message "Must be >  ↩
    20 and < 1000"
field boolean --named major
field string --named picture
```

The easiest way to see the results of Forge operating on the `Event.java` JPA Entity is to use the **Outline View** of JBoss Developer Studio. It is normally on the right-side of the IDE when using the JBoss Perspective.

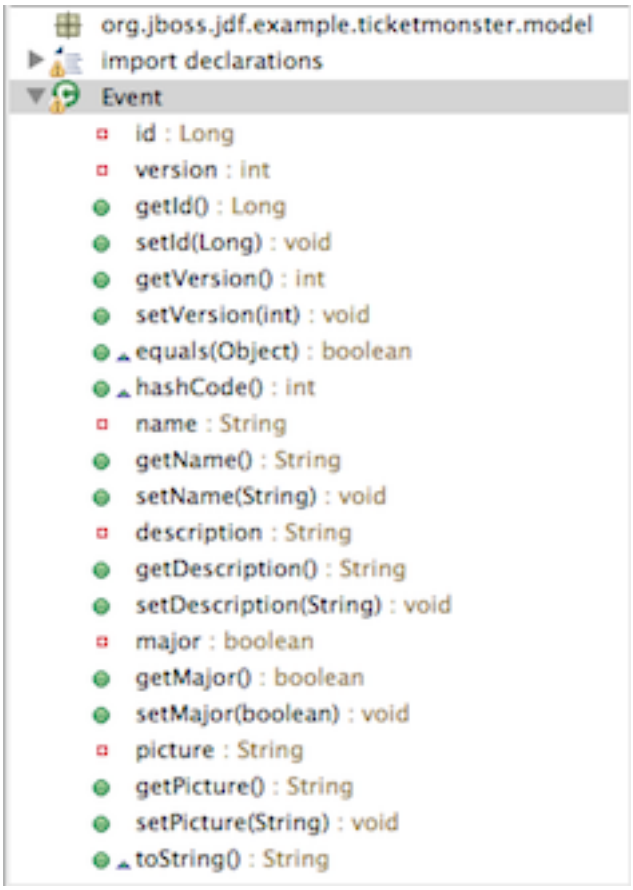

Figure 5.9: Outline View

Chapter 6

Reviewing persistence.xml & updating import.sql

By default, the entity classes generate the database schema, and is controlled by `src/main/resources/persistence.xml`.

The two key settings are the `<jta-data-source>` and the `hibernate.hbm2ddl.auto` property. The datasource maps to the datasource defined in `src\main\webapp\ticket-monster-ds.xml`.

The `hibernate.hbm2ddl.auto=create-drop` property indicates that all database tables will be dropped when an application is undeployed, or redeployed, and created when the application is deployed.

The `import.sql` file contains SQL statements that will inject sample data into your initial database structure. Add the following insert statements:

```
insert into Event (id, name, description, major, picture, version) values (1, ' ↩
    Shane''s Sock Puppets', 'This critically acclaimed masterpiece...', true, 'http ↩
    ://dl.dropbox.com/u/65660684/640px-Carnival_Puppets.jpg', 1);
insert into Event (id, name, description, major, picture, version) values (2, ' ↩
    Rock concert of the decade', 'Get ready to rock...', true, 'http://dl.dropbox. ↩
    com/u/65660684/640px-Weir%2C_Bob_(2007)_2.jpg', 1);
```

Chapter 7

Adding a new entity using JBoss Developer Studio

Alternatively, we can add an entity with JBoss Developer Studio or JBoss Tools.

First, right-click on the `.model` package and select **New** → **Class**. Enter the class name as `Venue` - our concerts & shows happen at particular stadiums, concert halls and theaters.

First, add some private fields representing the entities properties, which translate to the columns in the database table.

```
package org.jboss.jdf.example.ticketmonster.model;

public class Venue {
        private Long id;
        private String name;
        private String description;
        private int capacity;
}
```

Now, right-click on the editor itself, and from the pop-up, context menu select **Source** → **Generate Getters and Setters**.

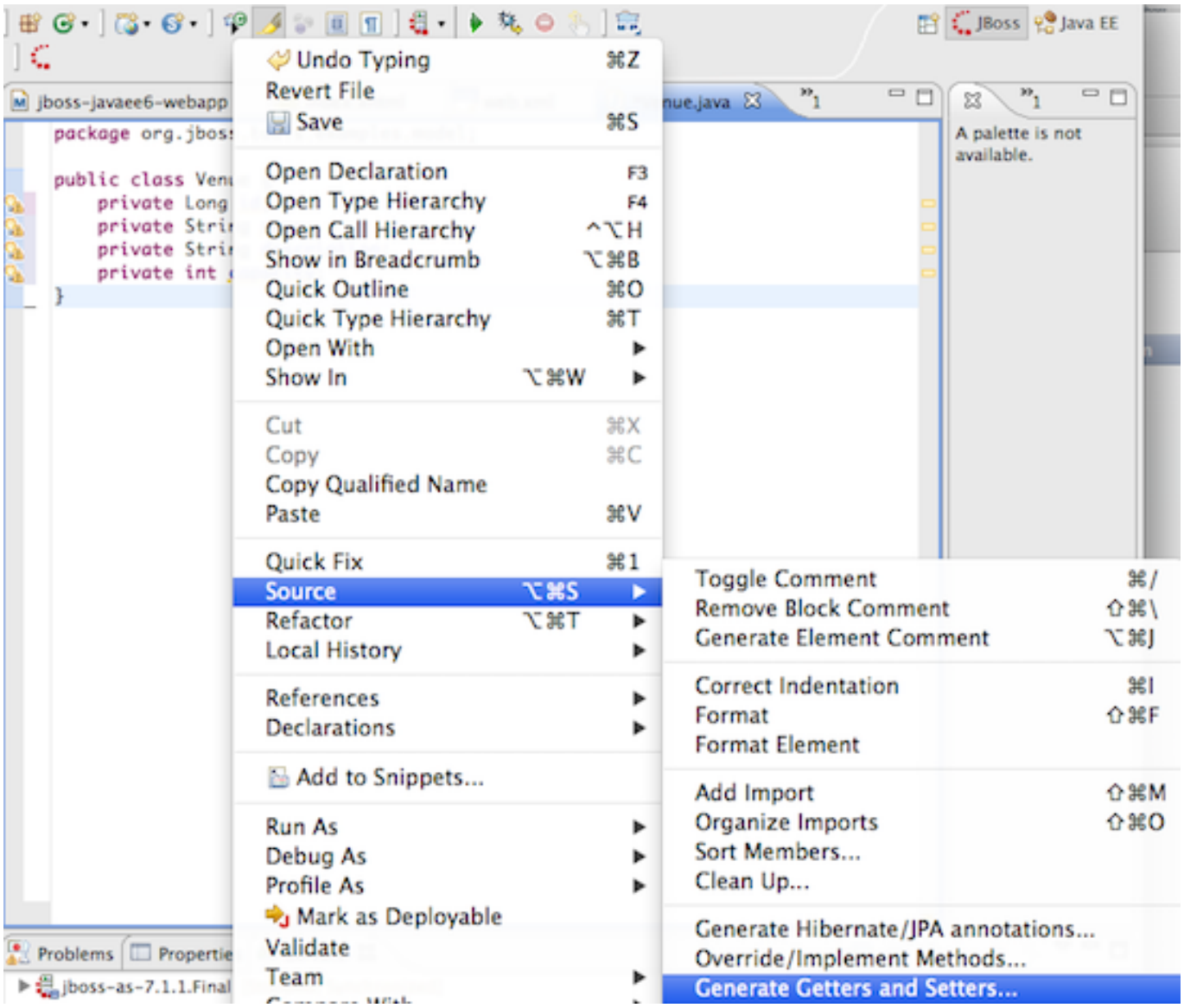

Figure 7.1: Generate Getters and Setters Menu

This will create accessor and mutator methods for all your fields, making them accessible properties for the entity class.

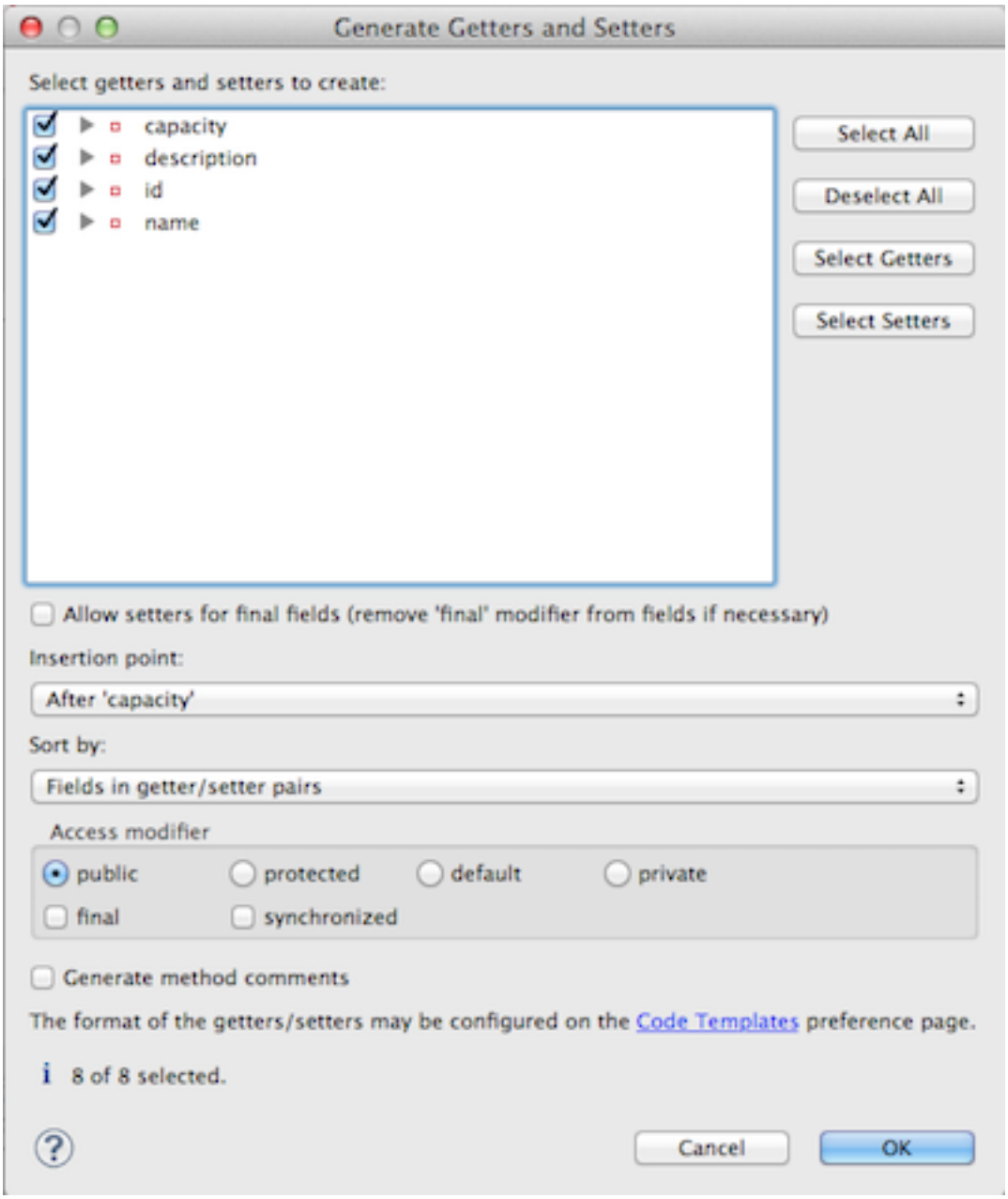

Figure 7.2: Generate Getters and Setters Dialog

Click **Select All** and then **OK**.

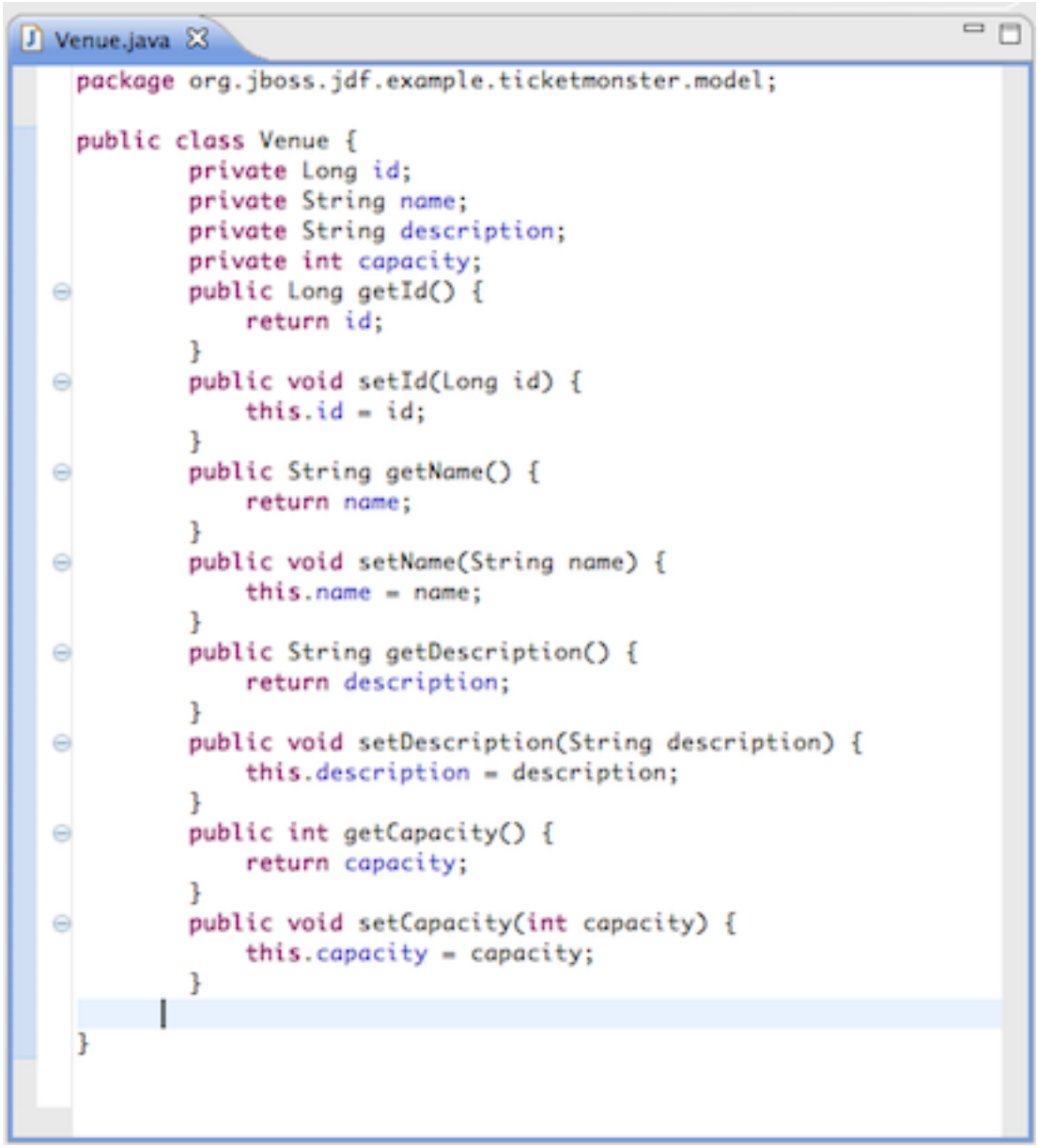

Figure 7.3: Venue.java with gets/sets

Now, right-click on the editor, from the pop-up context menu select **Source** → **Generate Hibernate/JPA Annotations**.

If you are prompted to save `Venue.java`, simply select OK.

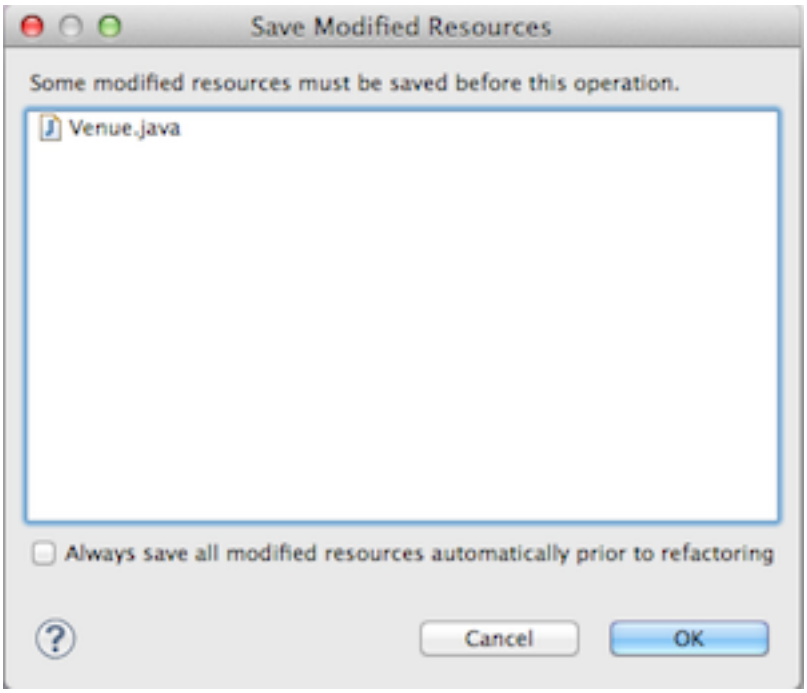

Figure 7.4: Save Modified Resources

The **Hibernate: add JPA annotations** wizard will start up. First, verify that `Venue` is the class you are working on.

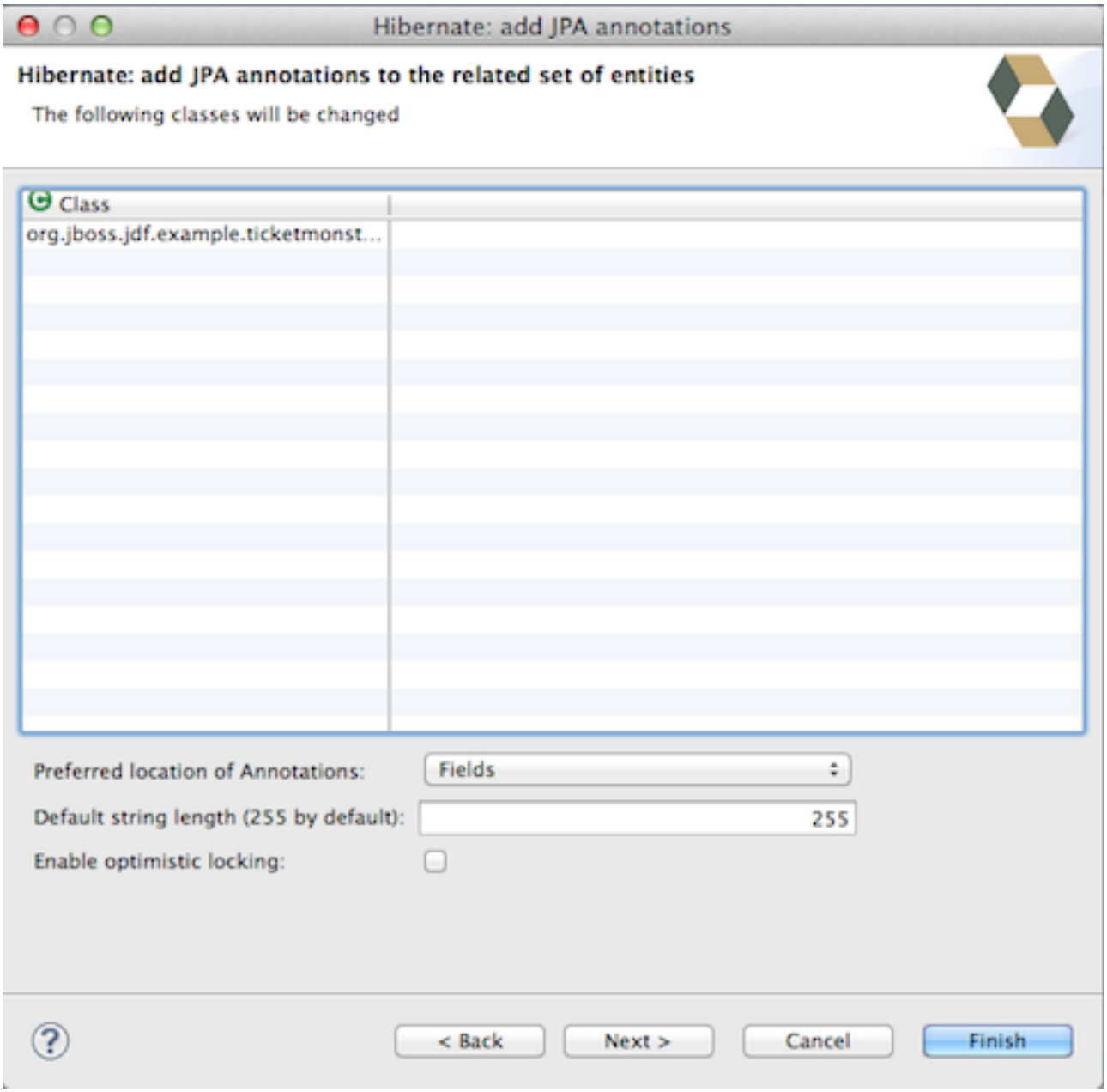

Figure 7.5: Hibernate: add JPA annotations

Select **Next**.

The next step in the wizard will provide a sampling of the refactored sources – describing the basic changes that are being made to `Venue`.

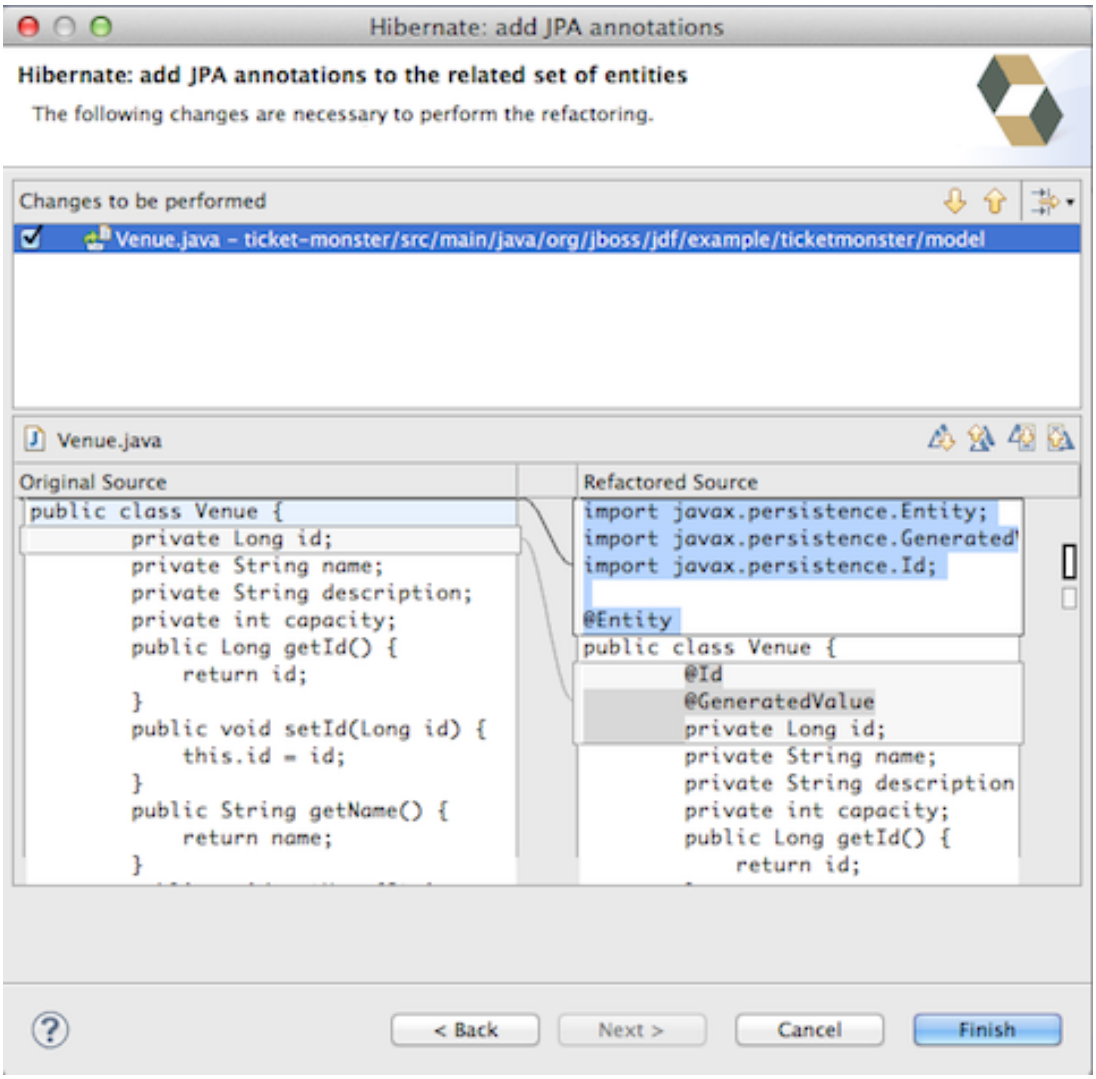

Figure 7.6: Hibernate: add JPA annotations Step 2

Select **Finish**.

Now you may wish to add the Bean Validation constraint annotations, such as `@NotNull` to the fields.

Chapter 8

Deployment

At this point, if you have not already deployed the application, right click on the project name in the Project Explorer and select **Run As** → **Run on Server**. If needed, this will startup the application server instance, compile & build the application and push the application into the `JBOSS_HOME/standalone/deployments` directory. This directory is scanned for new deployments, so simply placing your war in the directory will cause it to be deployed.

Caution
If you have been using another application server or web server such as Tomcat, shut it down now to avoid any port conflicts.

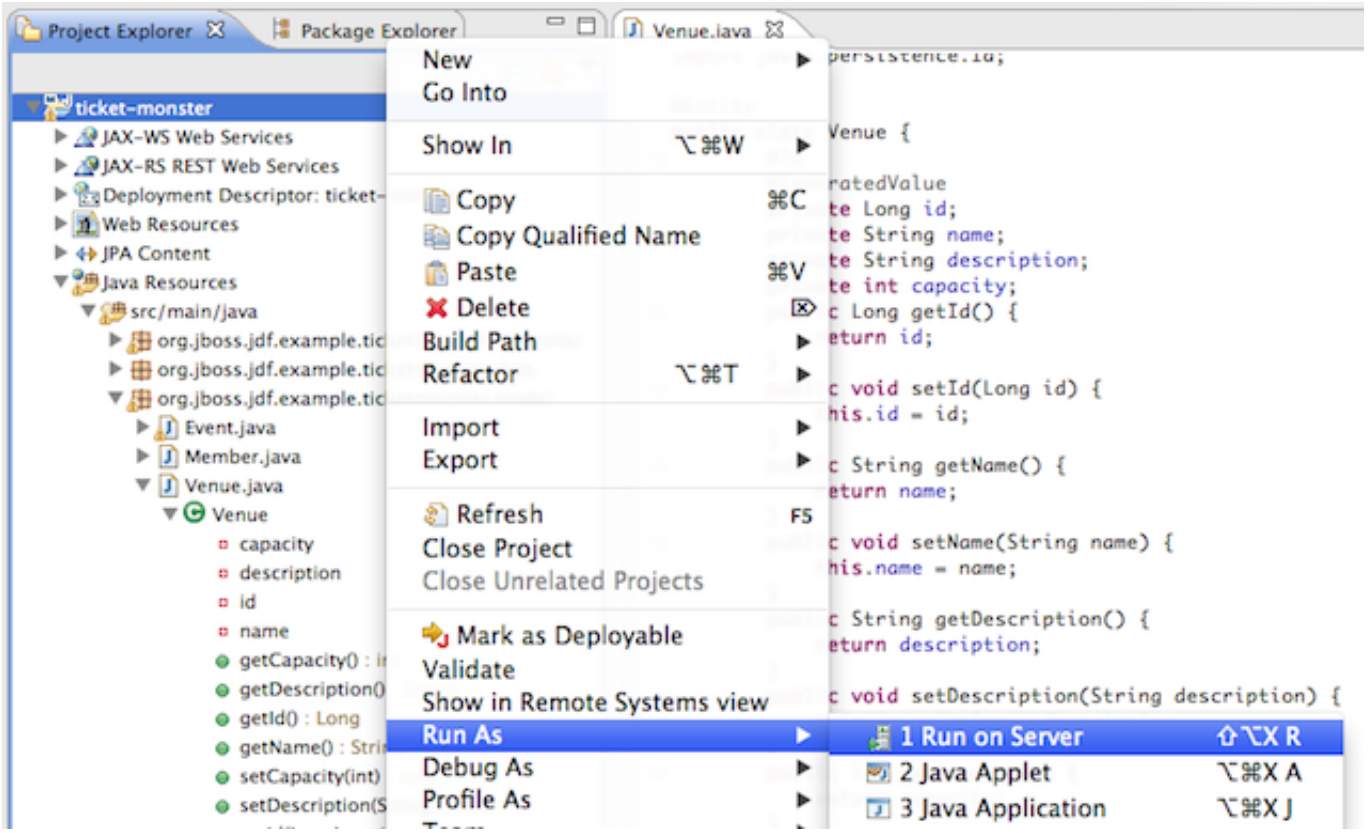

Figure 8.1: Run As → Run on Server

Now, deploy the h2console webapp. You can read how to do this in the h2console quickstart.

The **Run As** → **Run on Server** option will also launch the internal Eclipse browser with the appropriate URL so that you can immediately begin interacting with the application.

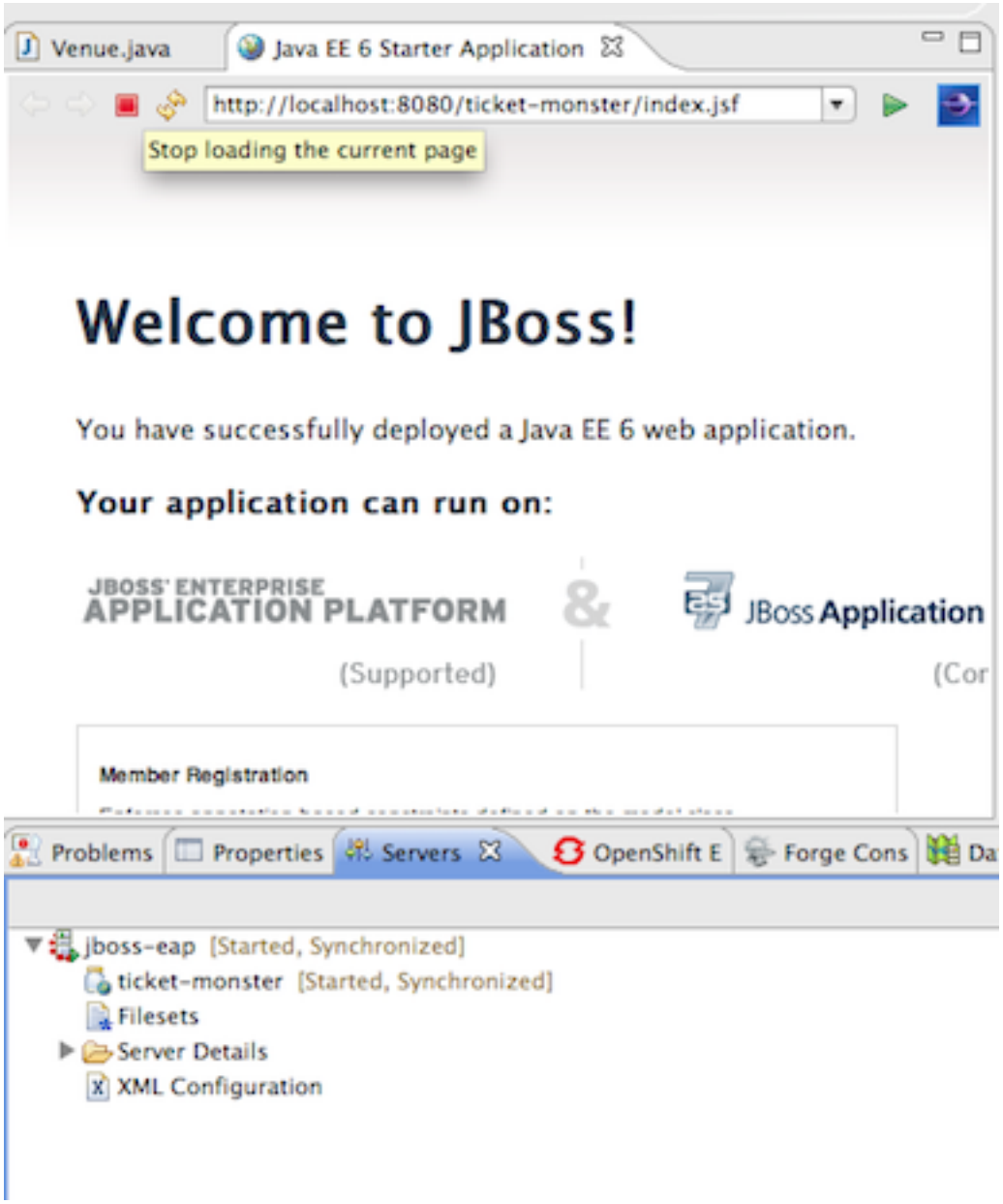

Figure 8.2: Eclipse Browser after Run As → Run on Server

Now, go to http://localhost:8080/h2console to start up the h2 console.

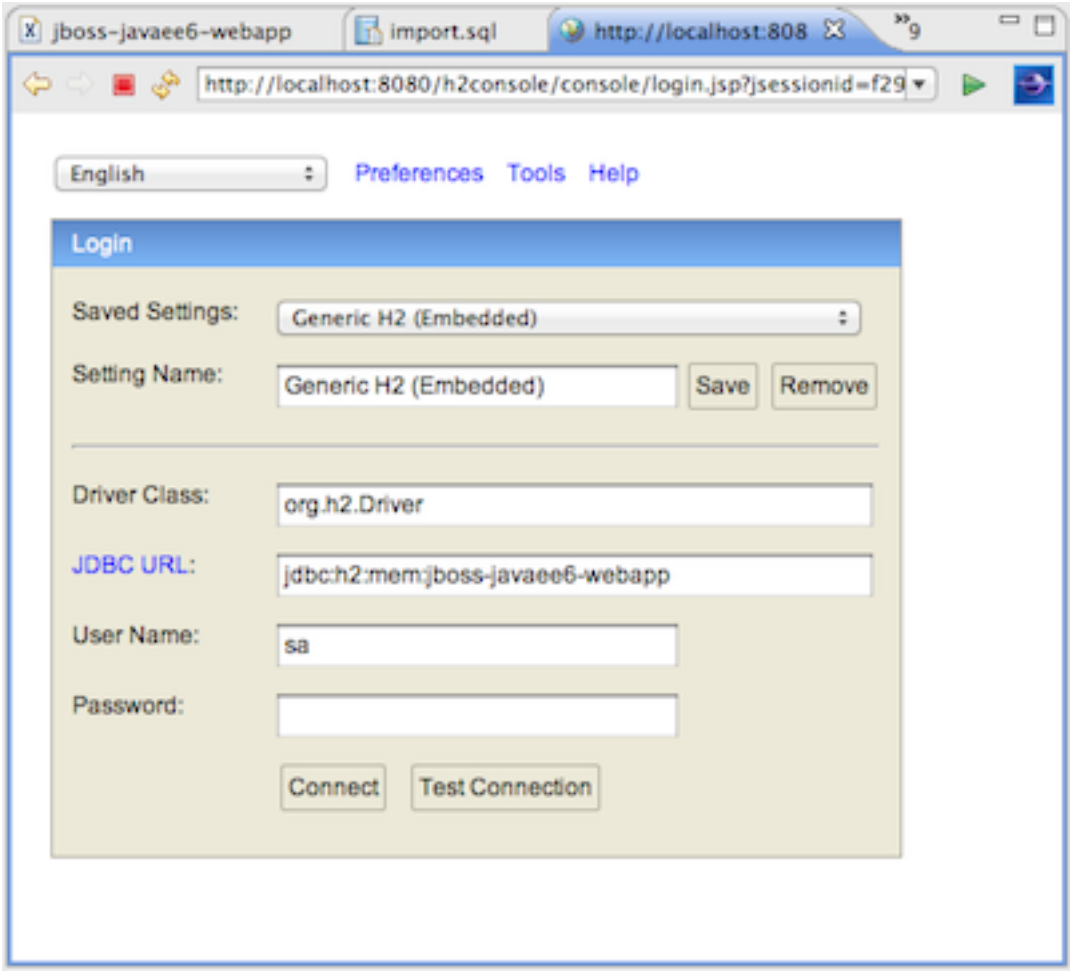

Figure 8.3: h2console in browser

Use `jdbc:h2:mem:ticket-monster` as the JDBC URL (this is defined in `src/main/webapp/WEB-INF/ticket-monste` `sa` as the username and `sa` as the password.

Click **Connect**

You will see both the `EVENT` table, the `VENUE` table and the `MEMBER` tables have been added to the H2 schema.

And if you enter the SQL statement: `select * from event` and select the **Run** (Ctrl-Enter) button, it will display the data you entered in the `import.sql` file in a previous step. With these relatively simple steps, you have verified that your new EE 6 JPA entities have been added to the system and deployed successfully, creating the supporting RDBMS schema as needed.

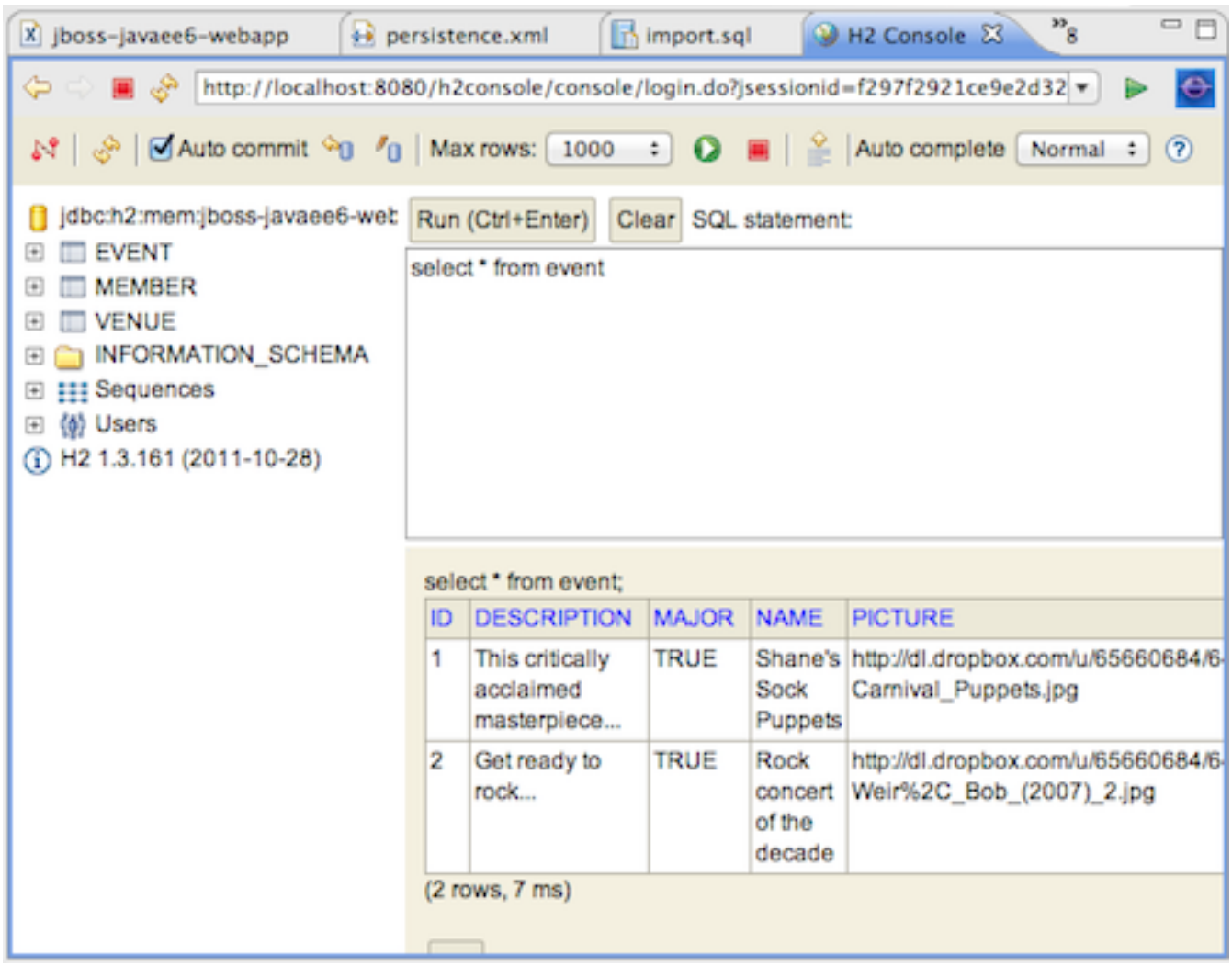

Figure 8.4: h2console Select * from Event

Chapter 9

Adding a JAX-RS RESTful web service

The goal of this section of the tutorial is to walk you through the creation of a POJO with the JAX-RS annotations.

Right-click on the `.rest` package, select **New** → **Class** from the context menu, and enter `EventService` as the class name.

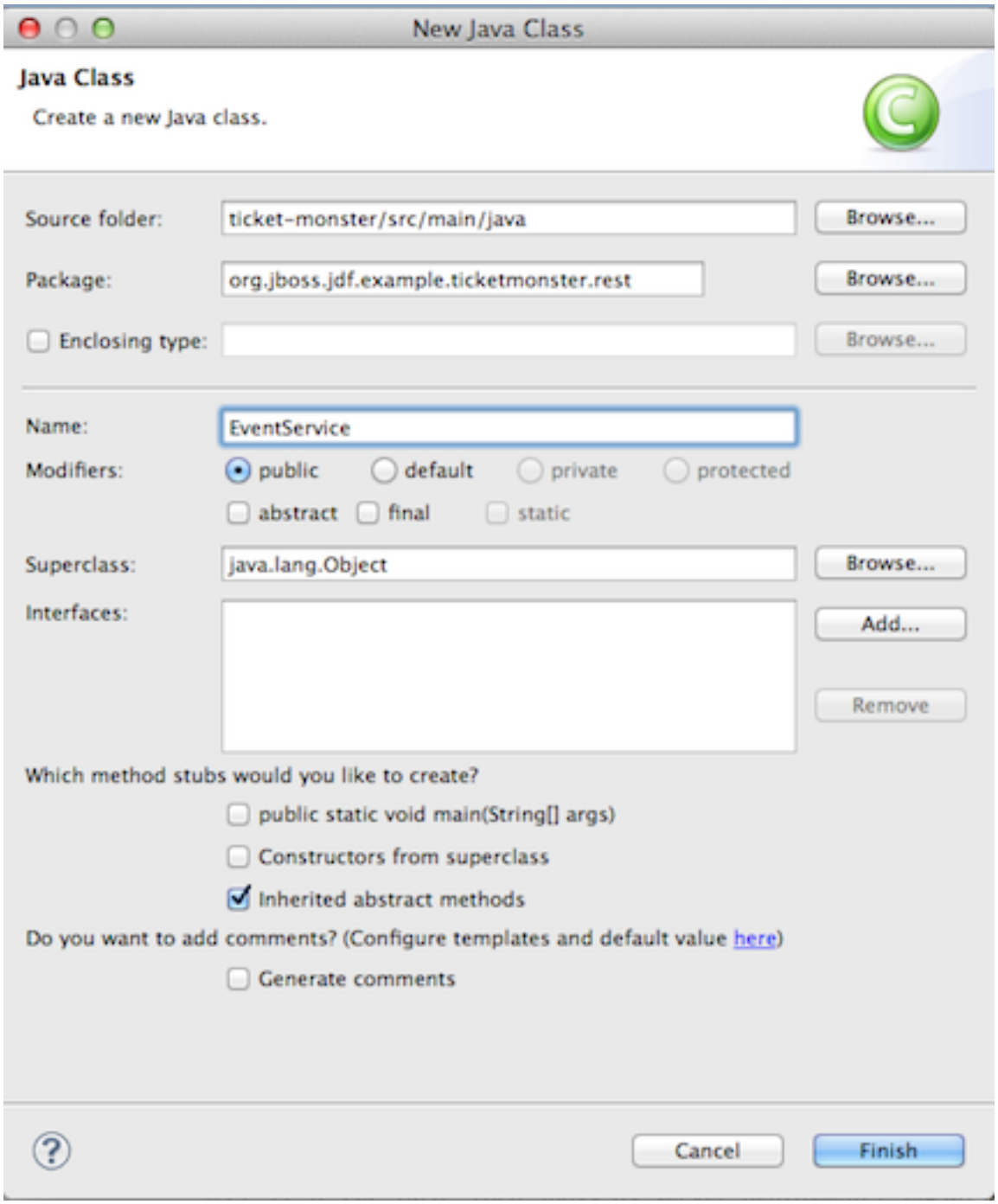

Figure 9.1: New Class EventService

Select **Finish**.

Replace the contents of the class with this sample code:

```
package org.jboss.jdf.example.ticketmonster.rest;

@Path("/events")
@RequestScoped
public class EventService {
        @Inject
        private EntityManager em;
```

```
        @GET
        @Produces(MediaType.APPLICATION_JSON)
        public List<Event> getAllEvents() {
                final List<Event> results =
                        em.createQuery(
                        "select e from Event e order by e.name").getResultList();
                return results;
        }
}
```

This class is a JAX-RS endpoint that returns all Events.

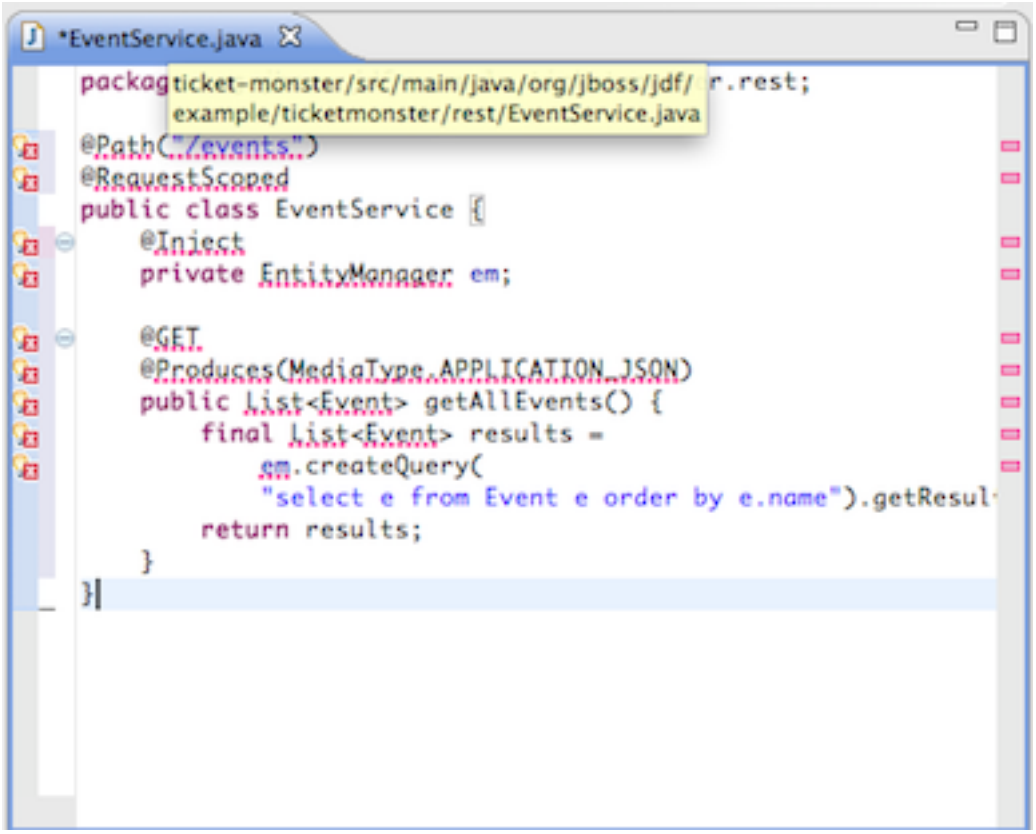

Figure 9.2: EventService after Copy and Paste

You'll notice a lot of errors, relating to missing imports. The easiest way to solve this is to right-click inside the editor and select **Source** → **Organize Imports** from the context menu.

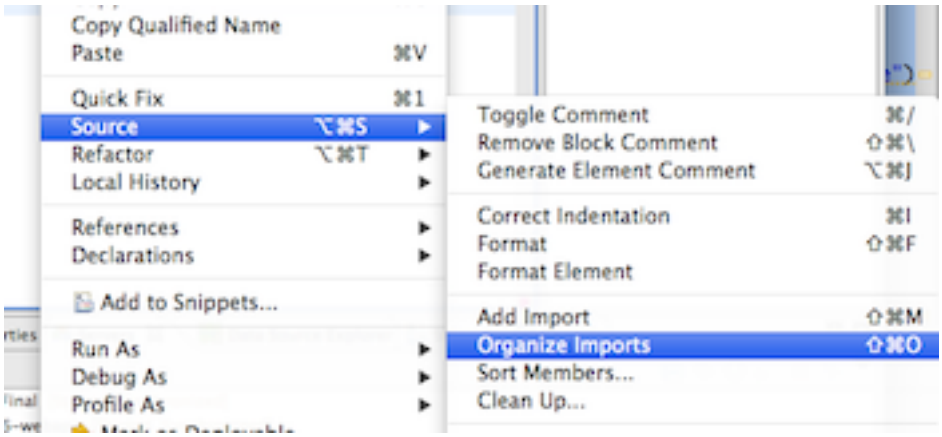

Figure 9.3: Source → Organize → Imports

Some of the class names are not unique. Eclipse will prompt you with any decisions around what class is intended. Select the following:

- `javax.ws.rs.core.MediaType`
- `org.jboss.tools.examples.Event`
- `javax.ws.rs.Produces`
- `java.util.List`
- `java.inject.Inject`
- `java.enterprise.context.RequestScoped`

The following screenshots illustrate how you handle these decisions. The Figure description indicates the name of the class you should select.

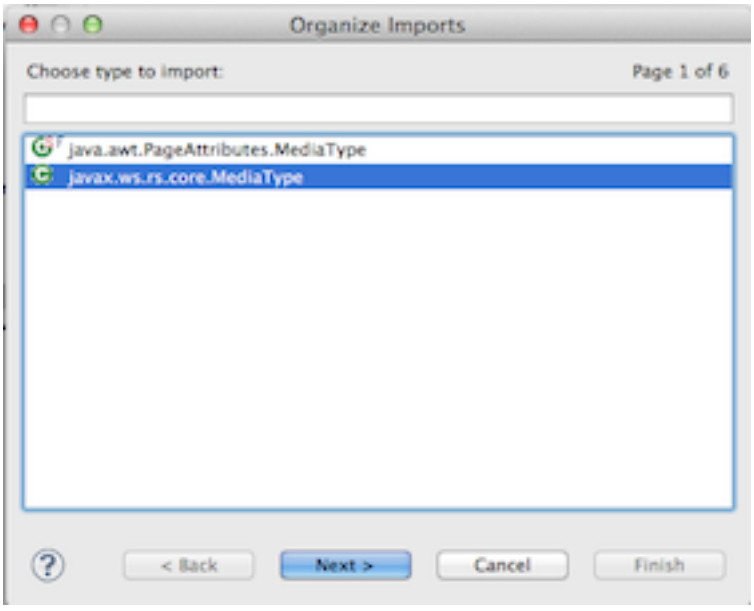

Figure 9.4: javax.ws.rs.core.MediaType

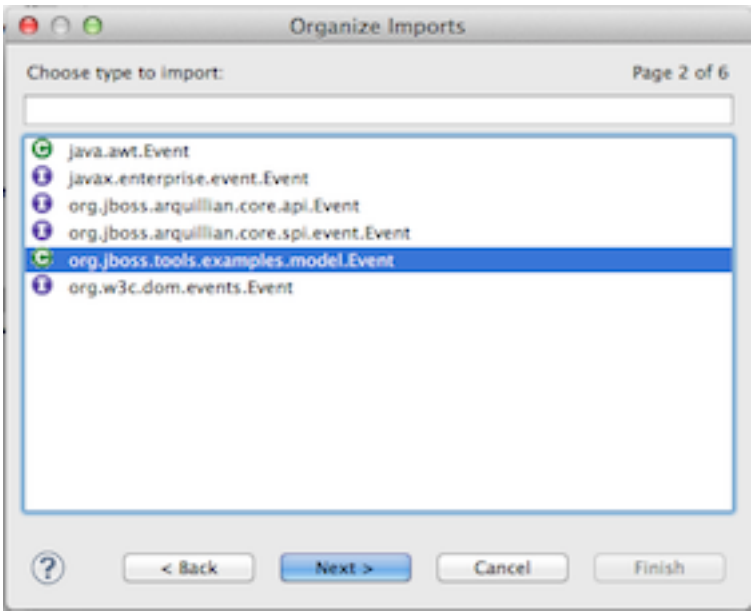

Figure 9.5: org.jboss.tools.examples.Event

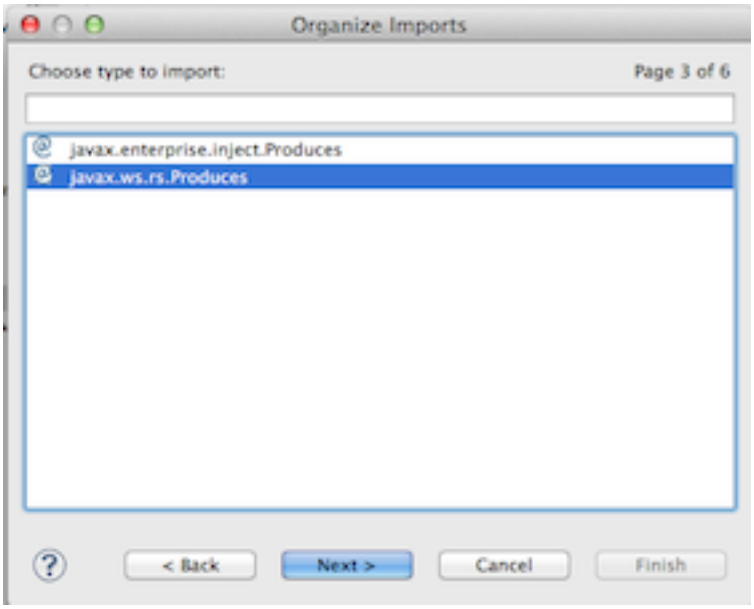

Figure 9.6: javax.ws.rs.Produces

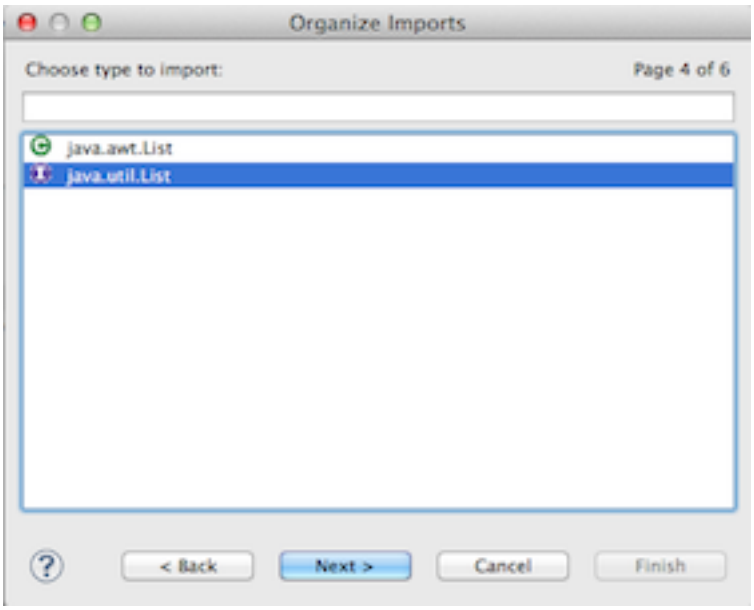

Figure 9.7: java.util.List

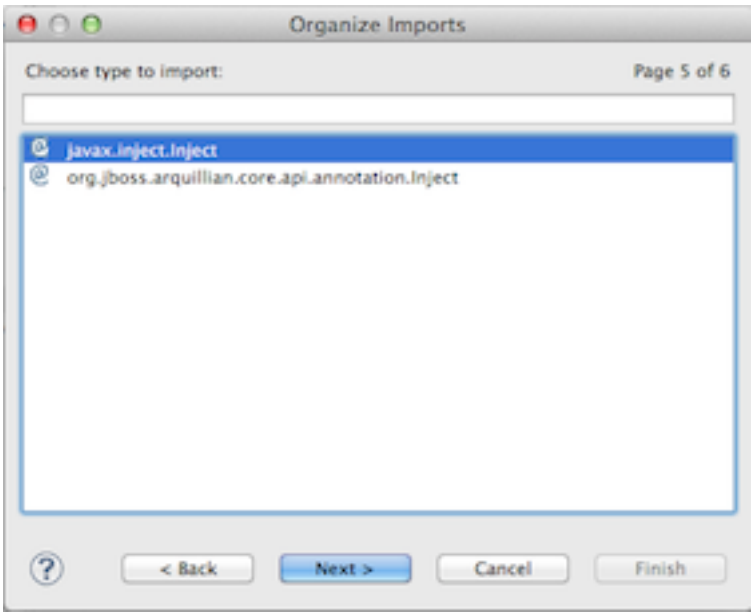

Figure 9.8: javax.inject.Inject

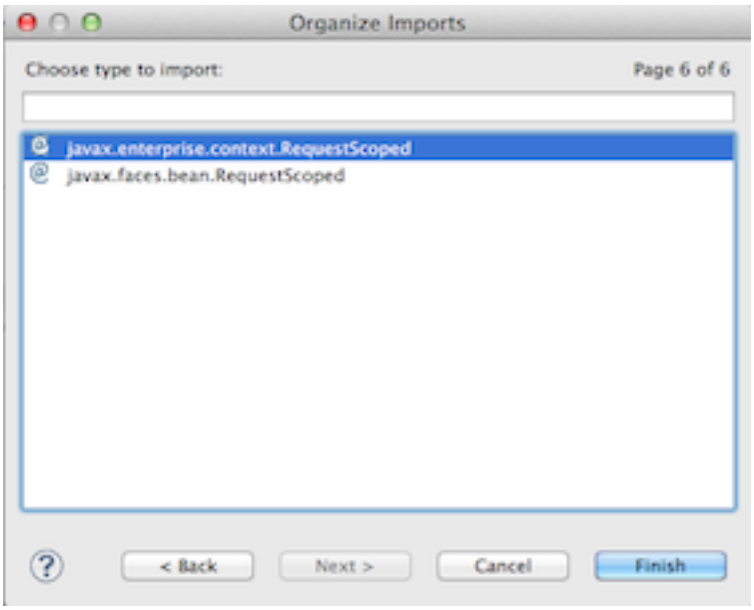

Figure 9.9: javax.enterprise.context.RequestScoped

You should end up with these imports:

```
import java.util.List;

import javax.enterprise.context.RequestScoped;
import javax.inject.Inject;
import javax.persistence.EntityManager;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

import org.jboss.jdf.example.ticketmonster.model.Event;
```

Once these import statements are in place you should have no more compilation errors. When you save `EventService.java`, you will see it listed in JAX-RS REST Web Services in the Project Explorer.

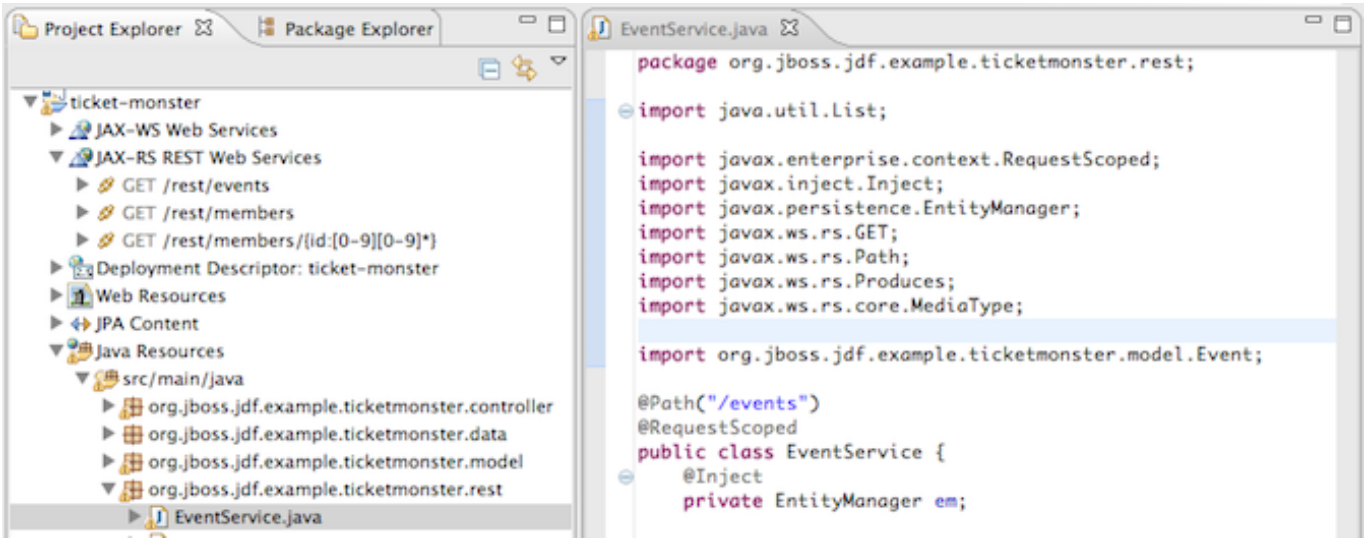

Figure 9.10: Project Explorer JAX-RS Services

This feature of JBoss Developer Studio and JBoss Tools provides a nice visual indicator that you have successfully configured your JAX-RS endpoint.

You should now redeploy your project via **Run As** → **Run on Server**, or by right clicking on the project in the **Servers** tab and select **Full Publish**.

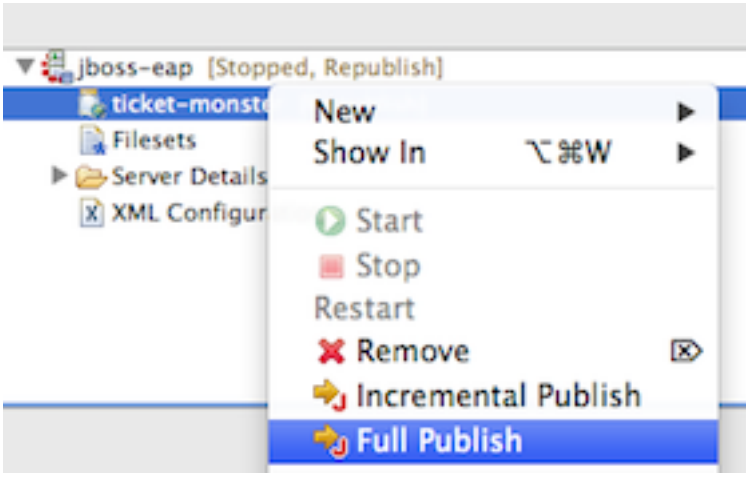

Figure 9.11: Full Publish

Using a browser, visit http://localhost:8080/ticket-monster/rest/events to see the results of the query, formatted as JSON (JavaScript Object Notation).

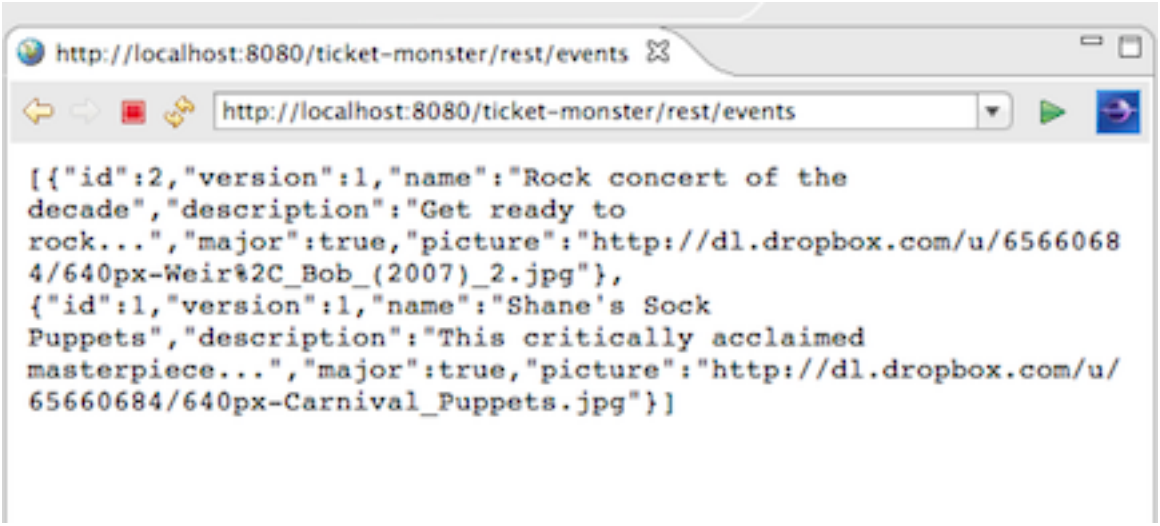

Figure 9.12: JSON Response

Note

The `rest` prefix is setup in a file called `JaxRsActivator.java` which contains a small bit of code that sets up the application for JAX-RS endpoints.

Chapter 10

Adding a jQuery Mobile client application

Now, it is time to add a HTML5, jQuery based client application that is optimized for the mobile web experience.

There are numerous JavaScript libraries that help you optimize the end-user experience on a mobile web browser. We have found that jQuery Mobile is one of the easier ones to get started with but as your skills mature, you might investigate solutions like Sencha Touch, Zepto or Jo. This tutorial focuses on jQuery Mobile as the basis for creating the UI layer of the application.

The UI components interact with the JAX-RS RESTful services (e.g. `EventService.java`).

Tip
For more information on building HTML5 + REST applications with JBoss technologies, check out Aerogear.

These next steps will guide you through the creation of a file called `mobile.html` that provides a mobile friendly version of the application, using jQuery Mobile.

First, using the Project Explorer, navigate to `src/main/webapp`, and right-click on `webapp`, and choose **New HTML file**.

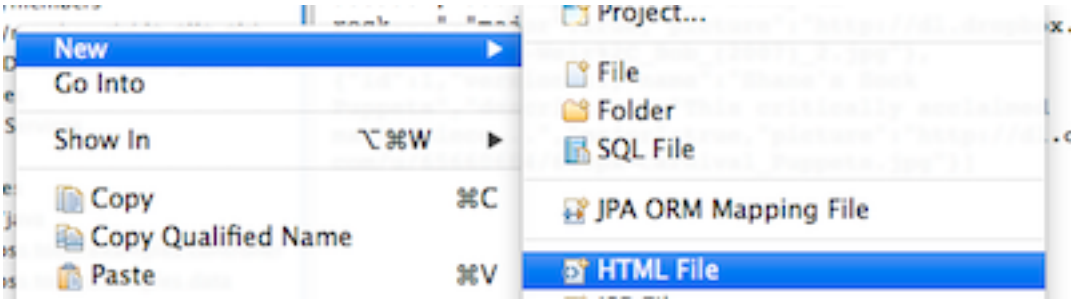

Figure 10.1: New HTML File

Caution
The New HTML File Wizard starts off with your target location being `m2e-wtp/web-resources`, this is an incorrect location and it is a bug, JBIDE-11472.
It is possible it may already be corrected by the time you read through this document.

Change directory to `ticket-monster/src/main/webapp` and enter name the file `mobile.html`.

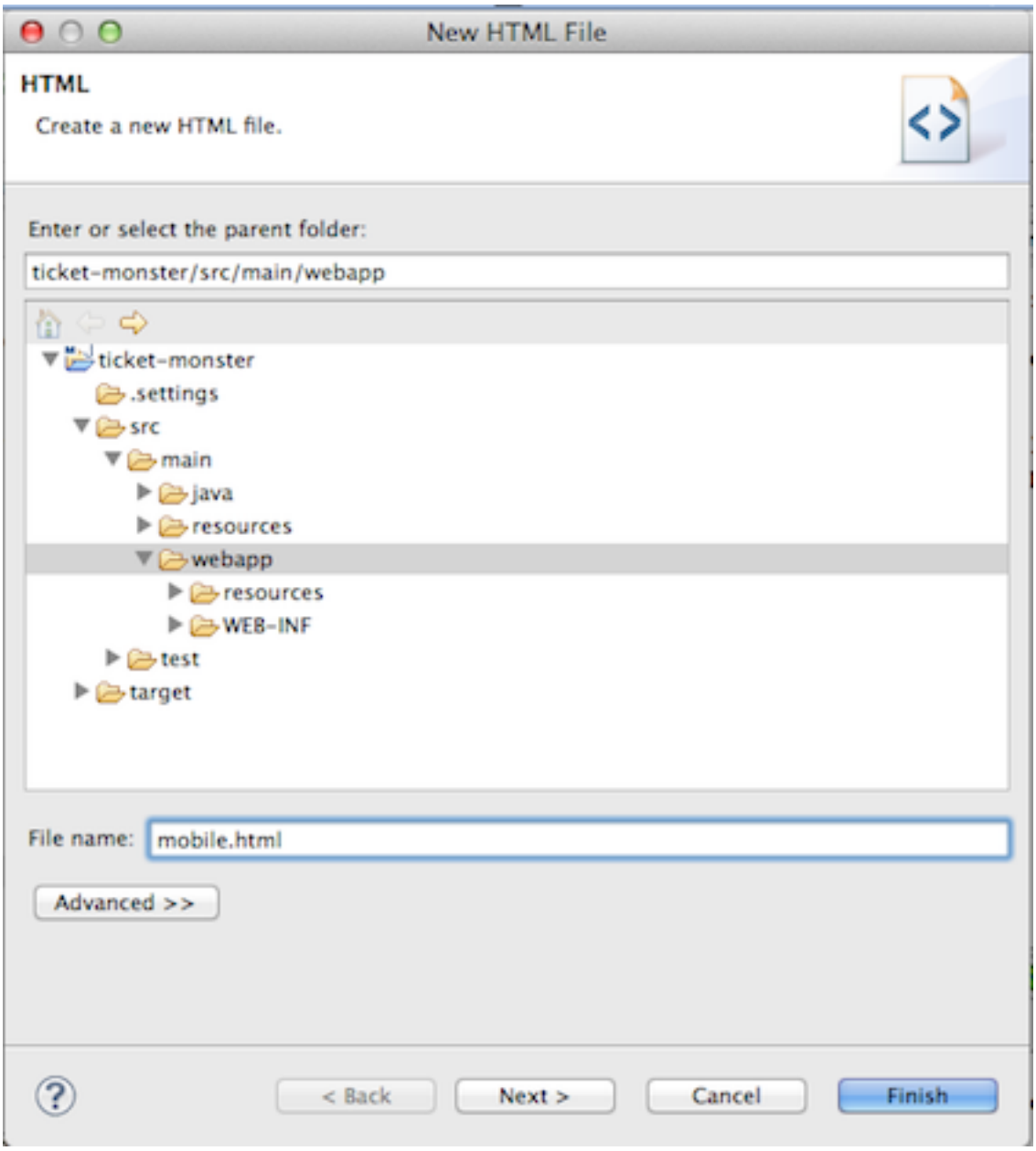

Figure 10.2: New HTML File src/main/webapp

Select **Next**.

On the **Select HTML Template** page of the **New HTML File** wizard, select **HTML5 jQuery Mobile Page**. This template will get you off to a fast start using jQuery Mobile.

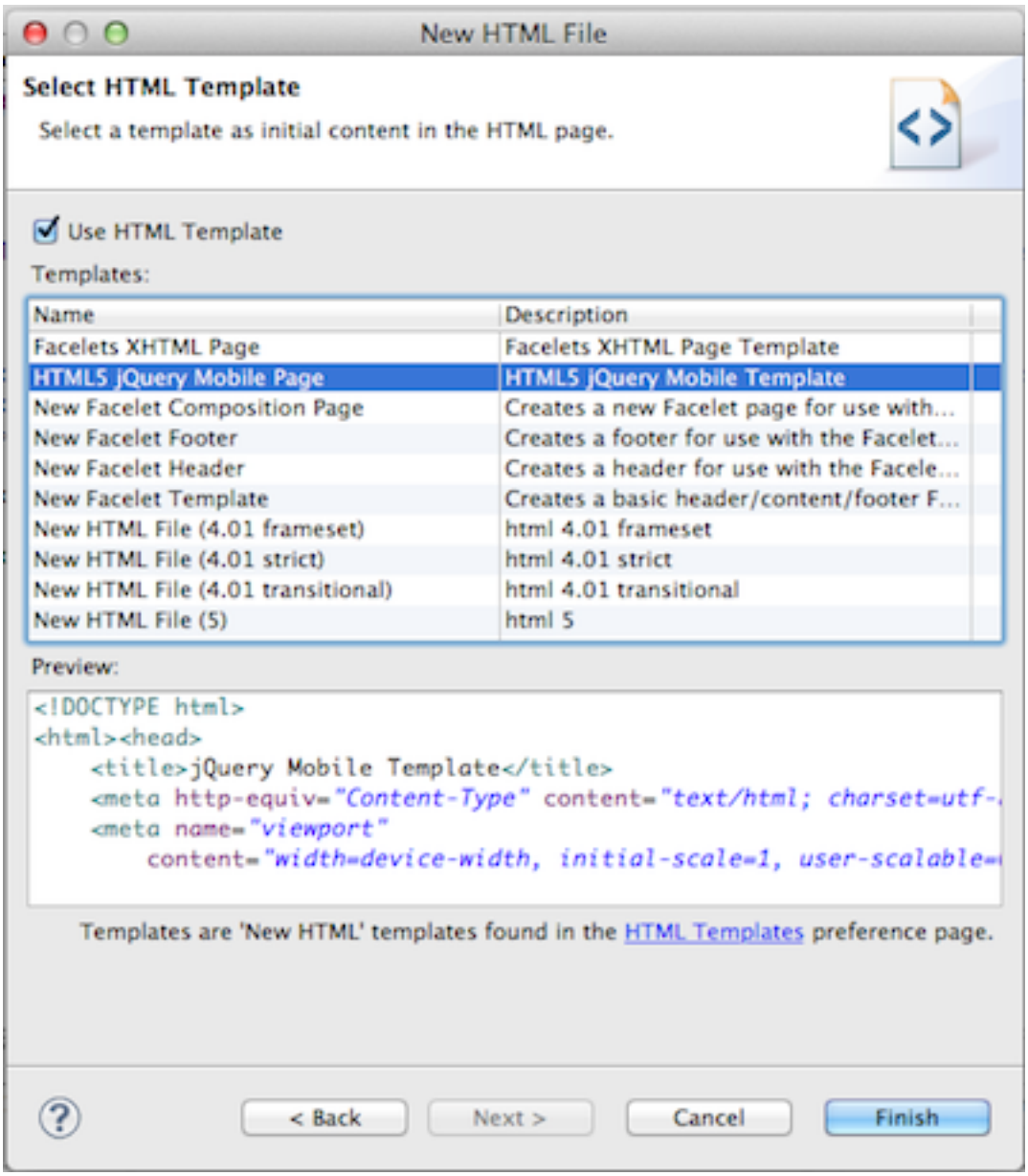

Figure 10.3: Select HTML5 jQuery Mobile Template

Select **Finish**.

The document must start with `<!DOCTYPE html>` as this identifies the page as HTML 5 based. For this particular phase of the tutorial, we are not introducing a bunch of HTML 5 specific concepts like the new form fields (type=email), websockets or the new CSS capabilities. For now, we simply wish to get our mobile application completed as soon as possible. The good news is that jQuery and jQuery Mobile make the consumption of a RESTful endpoint very simple.

You might notice that in the **Visual Page Editor**, the visual portion is not that attractive, this is because the majority of jQuery Mobile magic happens at runtime and our visual page editor simply displays the HTML without embellishment.

Visit http://localhost:8080/ticket-monster/mobile.html.

Note

Note: Normally HTML files are deployed automatically, if you find it missing, just use Full Publish or Run As Run on Server as demonstrated in previous steps.

As soon as the page loads, you will be prompted with an alert box with "Ready to Go". This alert box is generated from JavaScript that is associated with the pageinit event.

Figure 10.4: jQuery Mobile Template

One side benefit of using a HTML5 + jQuery-based front-end to your application is that it allows for fast turnaround in development. Simply edit the HTML file, save the file and refresh your browser.

Now the secret sauce to connecting your front-end to your back-end is simply editing the pageinit JavaScript event and including an invocation of the previously created Events JAX-RS service.

Insert the following block of code directly below the alert()

```
 $.getJSON("rest/events", function(events) {
// console.log("returned are " + events);
var listOfEvents = $("#listOfItems");
listOfEvents.empty();
$.each(events, function(index, event) {
        // console.log(event.name);
        listOfEvents.append("<li><a href='#'>" + event.name + "</a>");
});
listOfEvents.listview("refresh");
});
```

Note:

- using `$.getJSON("rest/events")` to hit the `EventService.java`
- a commented out `// console.log`, causes problems in IE

- Getting a reference to `listOfItems` which is declared in the HTML using an `id` attribute
- Calling `.empty` on that list - removing the exiting `One, Two, Three` items
- For each event - based on what is returned in step 1

- another commented out `// console.log`
- `append` the found event to the UL in the HTML
- `refresh` the `listOfItems`

Note
You may find the `.append("<li>...")` syntax unattractive, embedding HTML inside of the JS .append method, this can be corrected using various JS templating techniques.

The result is ready for the average mobile phone. Simply refresh your browser to see the results.

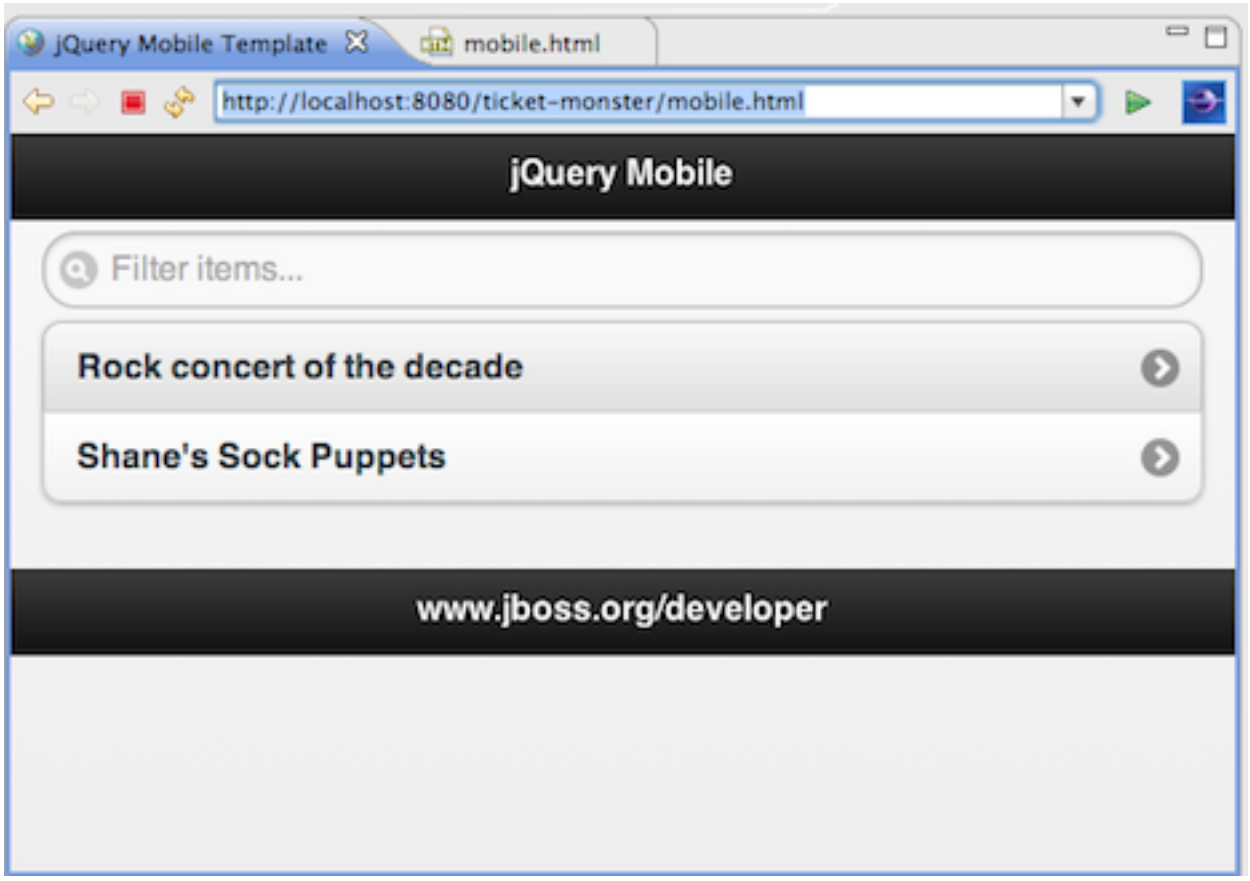

Figure 10.5: jQuery Mobile REST Results

JBoss Developer Studio and JBoss Tools includes BrowerSim to help you better understand what your mobile application will look like. Look for a "phone" icon in the toolbar, visible in the JBoss Perspective.

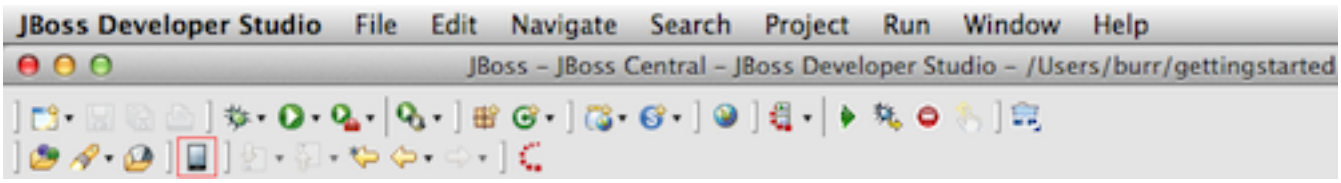

Figure 10.6: Mobile BrowserSim icon in Eclipse Toolbar

Note

The BrowserSim tool takes advantage of a locally installed Safari (Mac & Windows) on your workstation. It does not package a whole browser by itself. You will need to install Safari on Windows to leverage this feature – but that is more economical than having to purchase a MacBook to quickly look at your mobile-web focused application!

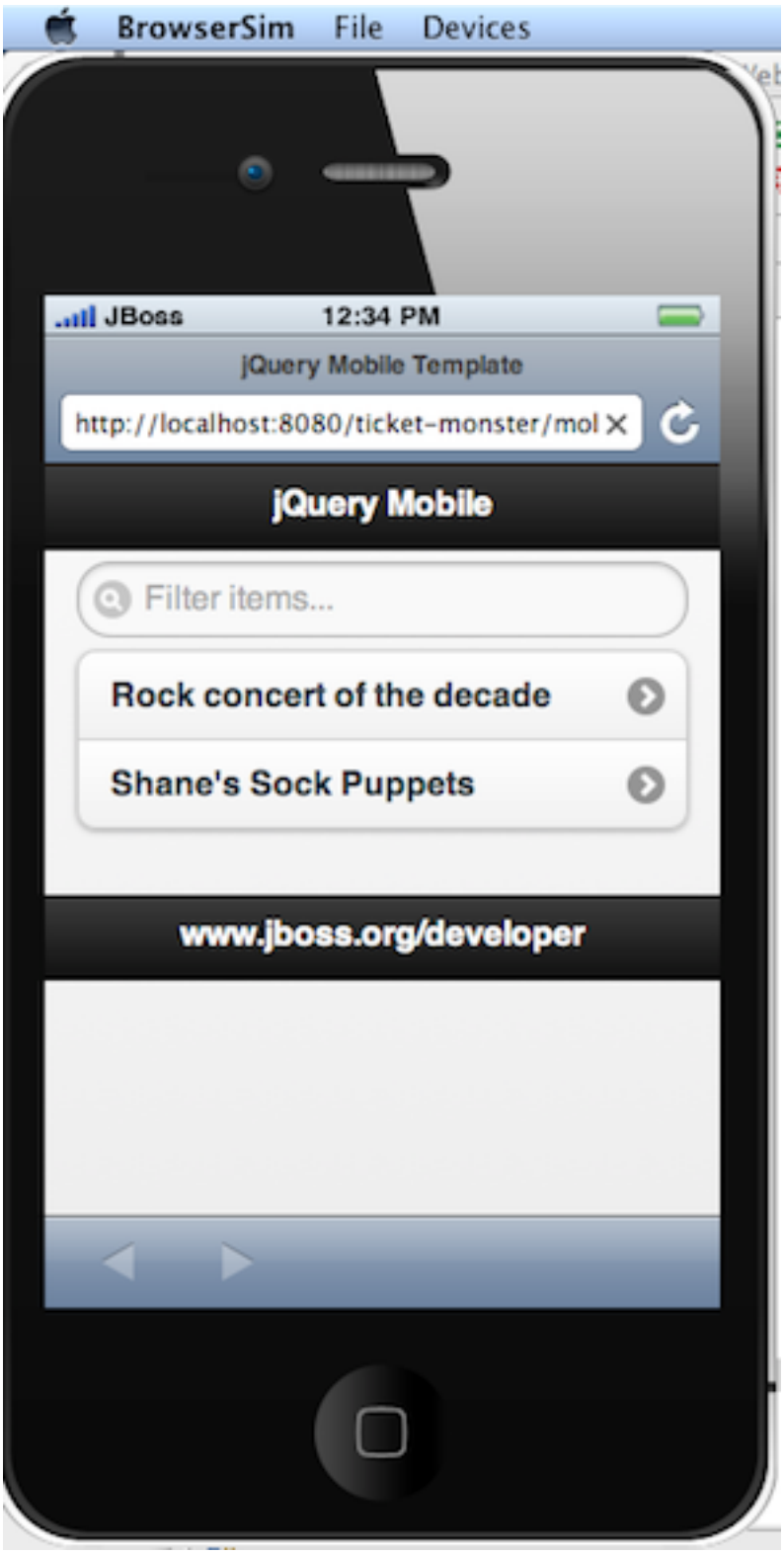

Figure 10.7: Mobile BrowserSim

The Mobile BrowserSim has a Devices menu, on Mac it is in the top menu bar and on Windows it is available via right-click as a pop-up menu. This menu allows you to change user-agent and dimensions of the browser, plus change the orientation of the device.

Figure 10.8: Mobile BrowserSim Devices Menu

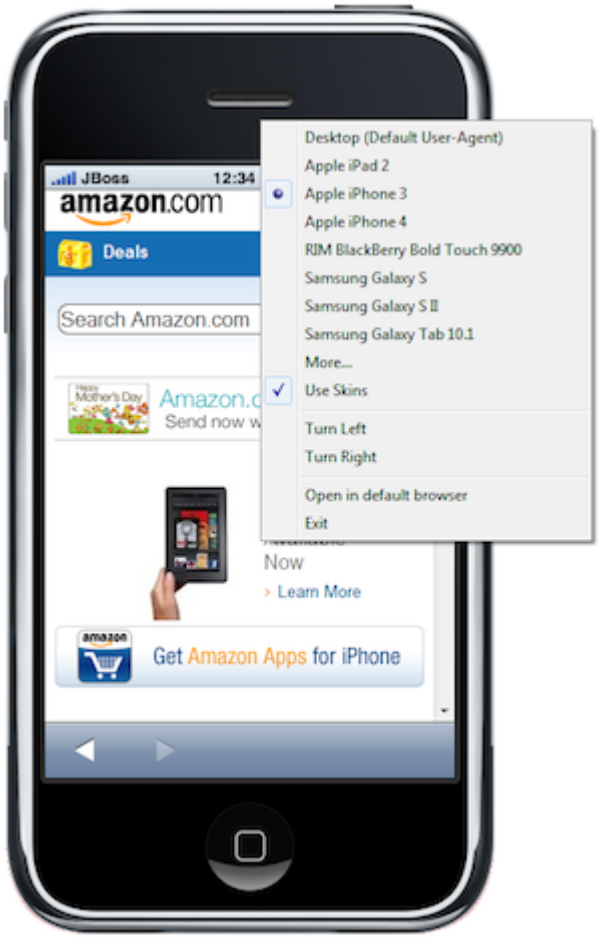

Figure 10.9: Mobile BrowserSim on Windows 7

You can also add your own custom device/browser types.

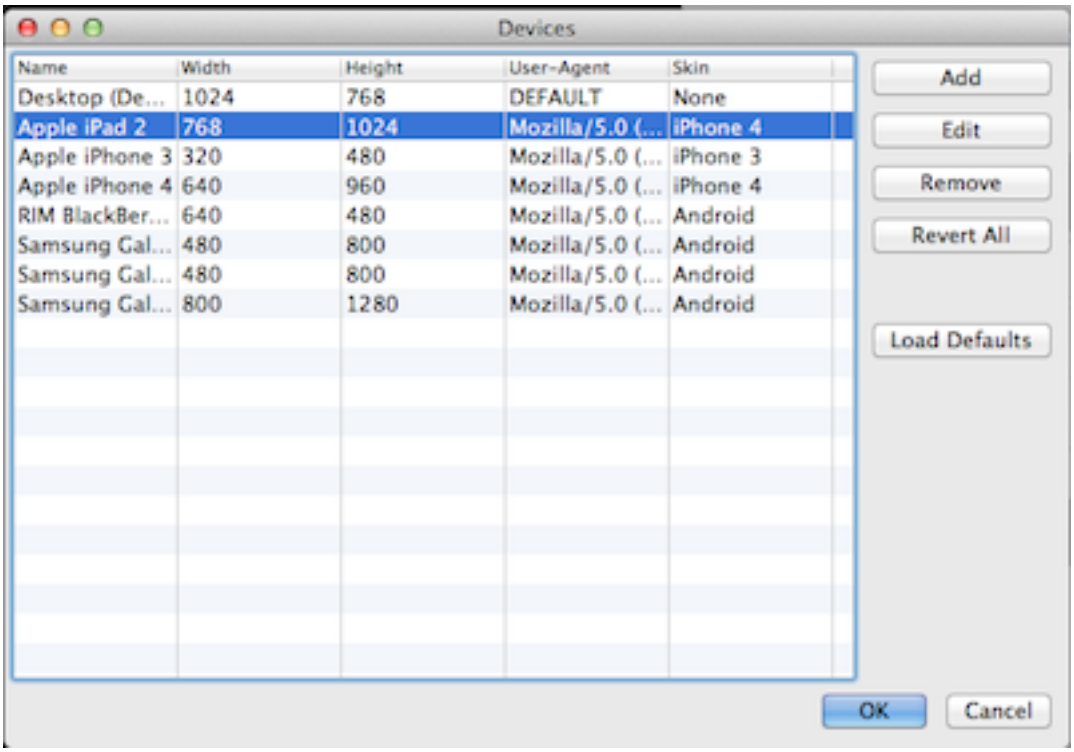

Figure 10.10: Mobile BrowserSim Custom Devices Window

Under the **File** menu, you will find a **View Page Source** option that will open up the mobile-version of the website's source code inside of JBoss Developer Studio. This is a very useful feature for learning how other developers are creating their mobile web presence.

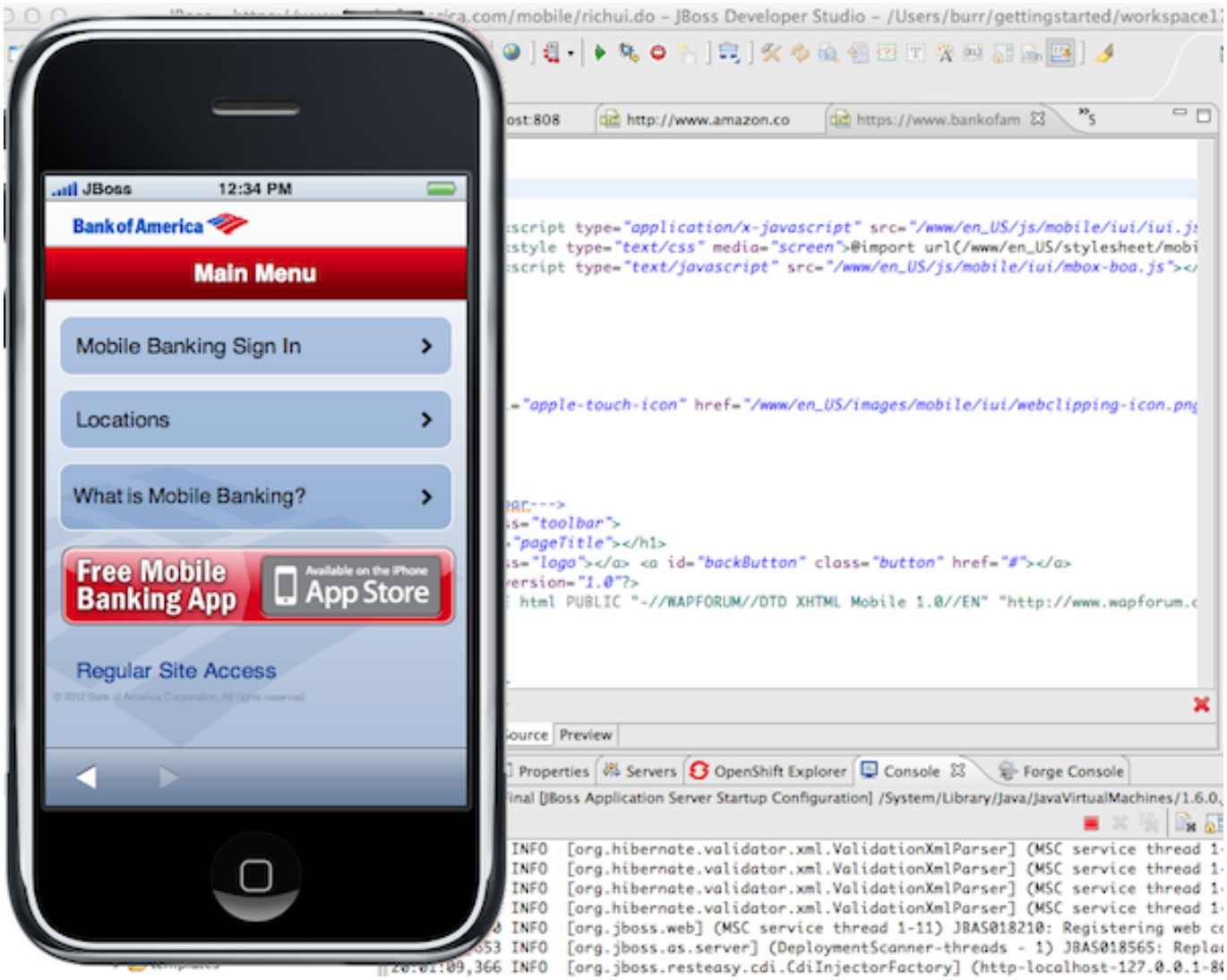

Figure 10.11: Mobile BrowserSim View Source

Chapter 11

Conclusion

This concludes our introduction to building HTML5 Mobile Web applications using Java EE 6 with Forge and JBoss Developer Studio. At this point, you should feel confident enough to tackle any of the additional exercises to learn how the TicketMonster sample application is constructed.

11.1 Cleaning up the generated code

Before we proceed with the tutorial and implement TicketMonster, we need to clean up some of the archetype-generated code. The Member management code, while useful for ilustrating the general setup of a Java EE 6 web application, will not be part of TicketMonster, so we can safely remove some packages, classes, and resources:

- All the Member-related persistence and business code:
  - `src/main/java/org/jboss/jdf/example/ticketmonster/controller`
  - `src/main/java/org/jboss/jdf/example/ticketmonster/data`
  - `src/main/java/org/jboss/jdf/example/ticketmonster/model/Member.java`
  - `src/main/java/org/jboss/jdf/example/ticketmonster/rest/MemberResourceRESTService.java`
  - `src/main/java/org/jboss/jdf/example/ticketmonster/service/MemberRegistration.java`
- Generated web content
  - `src/main/webapp/index.html`
  - `src/main/webapp/index.xhtml`
  - `src/main/webapp/WEB-INF/templates/default.xhtml`
- JSF configuration (we will re-add it via Forge)
  - `src/main/webapp/WEB-INF/faces-config.xml`
- Prototype mobile application (we will generate a proper mobile interface)
  - `src/main/webapp/mobile.html`

Also, we will update the `src/main/resources/import.sql` file and remove the `Member` entity insertion:

```
insert into Member (id, name, email, phone_number) values (0, 'John Smith', 'john. ↩
    smith@mailinator.com', '2125551212'
```

The data file should contain only the Event data import:

```
insert into Event (id, name, description, major, picture, version) values (1, ' ↩
    Shane''s Sock Puppets', 'This critically acclaimed masterpiece...', true, 'http ↩
    ://dl.dropbox.com/u/65660684/640px-Carnival_Puppets.jpg', 1);
insert into Event (id, name, description, major, picture, version) values (2, ' ↩
    Rock concert of the decade', 'Get ready to rock...', true, 'http://dl.dropbox. ↩
    com/u/65660684/640px-Weir%2C_Bob_(2007)_2.jpg', 1);
```

Part II

Building the persistence layer with JPA2 and Bean Validation

Chapter 12

What will you learn here?

You have set up your project successfully. Now it is time to begin working on the TicketMonster application, and the first step is adding the persistence layer. After reading this guide, you'll understand what design and implementation choices to make. Topics covered include:

- RDBMS design using JPA entity beans
- How to validate your entities using Bean Validation
- How to populate test data
- Basic unit testing using JUnit

We'll round out the guide by revealing the required, yet short and sweet, configuration.

The tutorial will show you how to perform all these steps in JBoss Developer Studio, including screenshots that guide you through. For those of you who prefer to watch and learn, the included videos show you how we performed all the steps.

TicketMonster contains 14 entities, of varying complexity. In the introduction, you have seen the basic steps for creating a couple of entities (`Event` and `Venue`) and interacting with them. In this tutorial we'll go deeper into domain model design, we'll classify the entities, and walk through designing and creating one of each group.

Chapter 13

Your first entity

The simplest kind of entities are often those representing lookup tables. `TicketCategory` is a classic lookup table that defines the ticket types available (e.g. Adult, Child, Pensioner). A ticket category has one property - *description*.

What's in a name?
Using a consistent naming scheme for your entities can help another developer get up to speed with your code base. We've named all our lookup tables XXXCategory to allow us to easily spot them.

Let's start by creating a JavaBean to represent the ticket category:

src/main/java/org/jboss/jdf/example/ticketmonster/model/TicketCategory.java

```
public class TicketCategory {

    /* Declaration of fields */

    /**
     * <p>
     * The description of the of ticket category.
     * </p>
     *
     */
    private String description;

    /* Boilerplate getters and setters */

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    @Override
    public String toString() {
        return description;
    }
}
```

We're going to want to keep the ticket category in collections (for example, to present it as part of drop down in the UI), so it's important that we properly implement `equals()` and `hashCode()`. At this point, we need to define a property (or group of properties) that uniquely identifies the ticket category. We refer to these properties as the "entity's natural identity".

Defining an entity's natural identity

Using an ORM introduces additional constraints on object identity. Defining the properties that make up an entity's natural identity can be tricky, but is very important. Using the object's identity, or the synthetic identity (database generated primary key) identity can introduce unexpected bugs into your application, so you should always ensure you use a natural identity. You can read more about the issue at https://community.jboss.org/wiki/EqualsAndHashCode.

For ticket category, the choice of natural identity is easy and obvious - it must be the one property, *description* that the entity has! Having identified the natural identity, adding an `equals()` and `hashCode()` method is easy. In Eclipse, choose *Source* → *Generate hashCode() and equals()...*

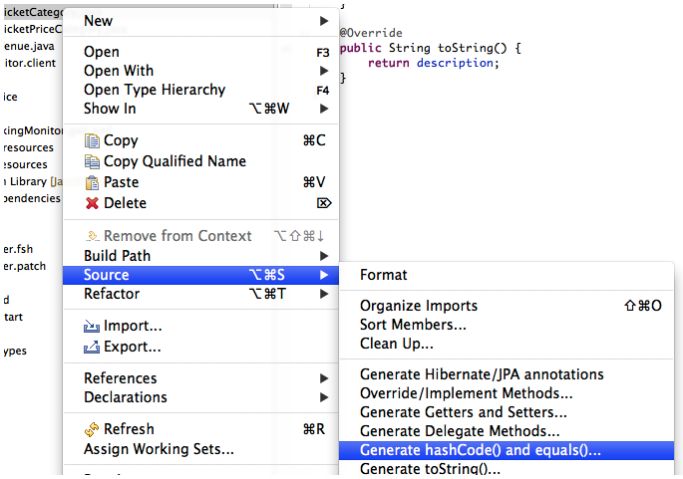

Figure 13.1: Generate hashCode() and equals() in Eclipse

Now, select the properties to include:

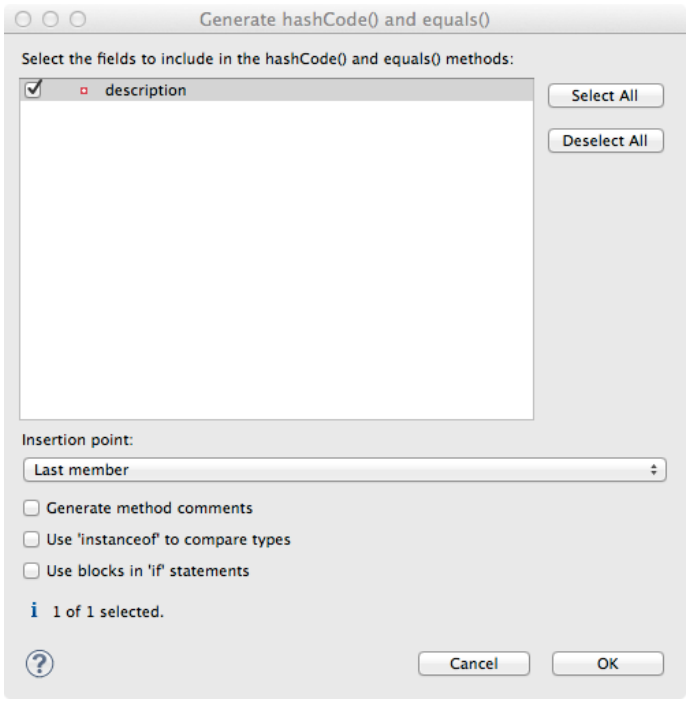

Figure 13.2: Generate hashCode() and equals() in Eclipse

Now that we have a JavaBean, let's proceed to make it an entity. First, add the `@Entity` annotation to the class:

src/main/java/org/jboss/jdf/example/ticketmonster/model/TicketCategory.java

```
@Entity
public class TicketCategory {

   ...

}
```

And, add the synthetic id:

src/main/java/org/jboss/jdf/example/ticketmonster/model/TicketCategory.java

```
@Entity
public class TicketCategory {

    /* Declaration of fields */

    /**
     * The synthetic id of the object.
     */
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    ...

    /* Boilerplate getters and setters */

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    ...

}
```

As we decided that our natural identifier was the description, we should introduce a unique constraint on the property:

src/main/java/org/jboss/jdf/example/ticketmonster/model/TicketCategory.java

```
@Entity
public class TicketCategory {

    /* Declaration of fields */

    ...

    /**
     * <p>
     * The description of the of ticket category.
     * </p>
     *
     * <p>
     * The description forms the natural id of the ticket category, and so must be unique.
     * </p>
     *
     */
```

```
    @Column(unique = true)
    private String description;

    ...
}
```

It's very important that any data you place in the database is of the highest quality - this data is probably one of your organisations most valuable assets! To ensure that bad data doesn't get saved to the database by mistake, we'll use Bean Validation to enforce constraints on our properties.

What is Bean Validation?
Bean Validation (JSR 303) is a Java EE specification which:

- provides a unified way of declaring and defining constraints on an object model.
- defines a runtime engine to validate objects

Bean Validation includes integration with other Java EE specifications, such as JPA. Bean Validation constraints are automatically applied before data is persisted to the database, as a last line of defence against bad data.

The *description* of the ticket category should not be empty for two reasons. Firstly, an empty ticket category description is no use to a person trying to book a ticket - it doesn't convey any information. Secondly, as the description forms the natural identity, we need to make sure the property is always populated.

Let's add the Bean Validation constraint `@NotEmpty`:

src/main/java/org/jboss/jdf/example/ticketmonster/model/TicketCategory.java

```
@Entity
public class TicketCategory {

    /* Declaration of fields */

    ...

    /**
     * <p>
     * The description of the of ticket category.
     * </p>
     *
     * <p>
     * The description forms the natural id of the ticket category, and so must be unique.
     * </p>
     *
     * <p>
     * The description must not be null and must be one or more characters, the Bean  ↩
         Validation constraint <code>@NotEmpty</code>
     * enforces this.
     * </p>
     *
     */
    @Column(unique = true)
    @NotEmpty
    private String description;

    ...
}
```

And that is our first entity! Here is the complete entity:

src/main/java/org/jboss/jdf/example/ticketmonster/model/TicketCategory.java

```
/**
 * <p>
 * A lookup table containing the various ticket categories. E.g. Adult, Child, Pensioner,  ↩
     etc.
 * </p>
 */
@Entity
public class TicketCategory {

    /* Declaration of fields */

    /**
     * The synthetic id of the object.
     */
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    /**
     * <p>
     * The description of the of ticket category.
     * </p>
     *
     * <p>
     * The description forms the natural id of the ticket category, and so must be unique.
     * </p>
     *
     * <p>
     * The description must not be null and must be one or more characters, the Bean  ↩
         Validation constraint <code>@NotEmpty</code>
     * enforces this.
     * </p>
     *
     */
    @Column(unique = true)
    @NotEmpty
    private String description;

    /* Boilerplate getters and setters */

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    /* toString(), equals() and hashCode() for TicketCategory, using the natural identity  ↩
        of the object */

    @Override
    public boolean equals(Object o) {
```

```
        if (this == o)
            return true;
        if (o == null || getClass() != o.getClass())
            return false;

        TicketCategory that = (TicketCategory) o;

        if (description != null ? !description.equals(that.description) : that.description ↩
            != null)
            return false;

        return true;
    }

    @Override
    public int hashCode() {
        return description != null ? description.hashCode() : 0;
    }

    @Override
    public String toString() {
        return description;
    }
}
```

TicketMonster contains another lookup tables, `EventCategory`. It's pretty much identical to `TicketCategory`, so we leave it as an exercise to the reader to investigate, and understand. If you are building the application whilst following this tutorial, copy the source over from the TicketMonster example.

Chapter 14

Database design & relationships

First, let's understand the the entity design.

An `Event` may occur at any number of venues, on various days and at various times. The intersection between an event and a venue is a `Show`, and each show can have a `Performance` which is associated with a date and time.

Venues are a separate grouping of entities, which, as mentioned, intersect with events via shows. Each venue consists of groupings of seats, each known as a `Section`.

Every section, in every show is associated with a ticket category via the `TicketPrice` entity.

Users must be able to book tickets for performances. A `Booking` is associated with a performance, and contains a collection of tickets.

Finally, both events and venues can have "media items", such as images or videos attached.

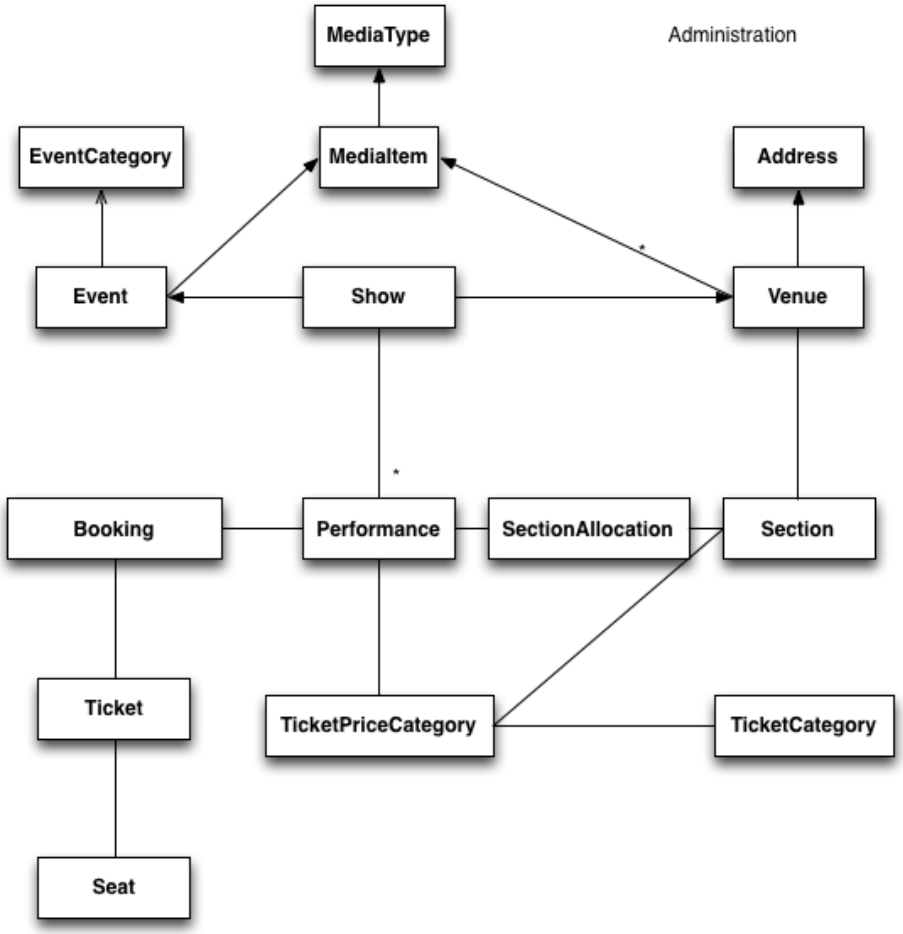

Figure 14.1: Entity-Relationship Diagram

14.1 Media items

Storing large binary objects, such as images or videos in the database isn't advisable (as it can lead to performance issues), and playback of videos can also be tricky, as it depends on browser capabilities. For TicketMonster, we decided to make use of existing services to host images and videos, such as YouTube or Flickr. All we store in the database is the URL the application should use to access the media item, and the type of the media item (note that the URL forms a media items natural identifier). We need to know the type of the media item in order to render the media correctly in the view layer.

In order for a view layer to correctly render the media item (e.g. display an image, embed a media player), it's likely that special code has had to have been added. For this reason we represent the types of media that TicketMonster understands as a closed set, unmodifiable at runtime. An enum is perfect for this!

Luckily, JPA has native support for enums, all we need to do is add the `@Enumerated` annotation:

src/main/java/org/jboss/jdf/example/ticketmonster/model/MediaItem.java

```
    ...

    /**
     * <p>
     * The type of the media, required to render the media item corectly.
     * </p>
     *
```

```
     * <p>
     * The media type is a <em>closed set</em> - as each different type of media requires ↩
         support coded into the view layers, it
     * cannot be expanded upon without rebuilding the application. It is therefore ↩
         represented by an enumeration. We instruct
     * JPA to store the enum value using it's String representation, so that we can later ↩
         reorder the enum members, without
     * changing the data. Of course, this does mean we can't change the names of media ↩
         items once the app is put into
     * production.
     * </p>
     */
    @Enumerated(STRING)
    private MediaType mediaType;

    ...
```

@Enumerated(STRING) or @Enumerated(ORDINAL)?
JPA can store an enum value using it's ordinal (position in the list of declared enums) or it's STRING (the name it is given). If you choose to store an ordinal, you musn't alter the order of the list. If you choose to store the name, you musn't change the enum name. The choice is yours!

The rest of `MediaItem` shouldn't present a challenge to you. If you are building the application whilst following this tutorial, copy both `MediaItem` and `MediaType` from the TicketMonster project.

14.2 Events

In Chapter 13 we saw how to build simple entites with properties, identify and apply constraints using Bean Validation, identify the natural id and add a synthetic id. From now on we'll assume you know how to build simple entities - for each new entity that we build, we will start with it's basic structure and properties filled in.

So, here is our starting point for Event (where we left at the end of the introduction, and including some comments reflecting the explanations above):

src/main/java/org/jboss/jdf/example/ticketmonster/model/Event.java

```
@Entity
public class Event {

    /* Declaration of fields */

    /**
     * The synthetic ID of the object.
     */
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    /**
     * <p>
     * The name of the event.
     * </p>
     *
     * <p>
     * The name of the event forms it's natural identity and cannot be shared between ↩
         events.
     * </p>
```

```
     *
     * <p>
     * Two constraints are applied using Bean Validation
     * </p>
     *
     * <ol>
     * <li><code>@NotNull</code> &mdash; the name must not be null.</li>
     * <li><code>@Size</code> &mdash; the name must be at least 5 characters and no more than 50 characters. This allows for
     * better formatting consistency in the view layer.</li>
     * </ol>
     */
    @Column(unique = true)
    @NotNull
    @Size(min = 5, max = 50, message = "An event's name must contain between 5 and 50 characters")
    private String name;

    /**
     * <p>
     * A description of the event.
     * </p>
     *
     * <p>
     * Two constraints are applied using Bean Validation
     * </p>
     *
     * <ol>
     * <li><code>@NotNull</code> &mdash; the description must not be null.</li>
     * <li><code>@Size</code> &mdash; the name must be at least 20 characters and no more than 1000 characters. This allows for
     * better formatting consistency in the view layer, and also ensures that event organisers provide at least some description
     * - a classic example of a business constraint.</li>
     * </ol>
     */
    @NotNull
    @Size(min = 20, max = 1000, message = "An event's name must contain between 20 and 1000 characters")
    private String description;


    /* Boilerplate getters and setters */

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getDescription() {
        return description;
```

```
    }

    public void setDescription(String description) {
        this.description = description;
    }

    /* toString(), equals() and hashCode() for Event, using the natural identity of the  ↩
        object */

    @Override
    public boolean equals(Object o) {
        if (this == o)
            return true;
        if (o == null || getClass() != o.getClass())
            return false;

        Event event = (Event) o;

        if (name != null ? !name.equals(event.name) : event.name != null)
            return false;

        return true;
    }

    @Override
    public int hashCode() {
        return name != null ? name.hashCode() : 0;
    }

    @Override
    public String toString() {
        return name;
    }
}
```

First, let's add a media item to `Event`. As multiple events (or venues) could share the same media item, we'll model the relationship as *many-to-one* - many events can reference the same media item.

Relationships supported by JPA

JPA can model four types of relationship between entities - one-to-one, one-to-many, many-to-one and many-to-many. A relationship may be bi-directional (both sides of the relationship know about each other) or uni-directional (only one side knows about the relationship).

Many database models are hierarchical (parent-child), as is TicketMonster's. As a result, you'll probably find you mostly use one-to-many and many-to-one relationships, which allow building parent-child models.

Creating a many-to-one relationship is very easy in JPA. Just add the `@ManyToOne` annotation to the field. JPA will take care of the rest. Here's the property for `Event`:

src/main/java/org/jboss/jdf/example/ticketmonster/model/Event.java

```
    ...

    /**
     * <p>
     * A media item, such as an image, which can be used to entice a browser to book a  ↩
         ticket.
     * </p>
     *
     * <p>
```

```
     * Media items can be shared between events, so this is modeled as a <code>@ManyToOne</ ↩
        code> relationship.
     * </p>
     *
     * <p>
     * Adding a media item is optional, and the view layer will adapt if none is provided.
     * </p>
     *
     */
    @ManyToOne
    private MediaItem mediaItem;

    ...

    public MediaItem getMediaItem() {
        return mediaItem;
    }

    public void setMediaItem(MediaItem picture) {
        this.mediaItem = picture;
    }

    ...
```

There is no need for a media item to know who references it (in fact, this would be a poor design, as it would reduce the reusability of `MediaItem`), so we can leave this as a uni-directional relationship.

An event will also have a category. Once again, many events can belong to the same event category, and there is no need for an event category to know what events are in it. To add this relationship, we add the `eventCategory` property, and annotate it with `@ManyToOne`, just as we did for `MediaItem`.

And that's Event created. Here is the full source:

src/main/java/org/jboss/jdf/example/ticketmonster/model/Event.java

```
/**
 * <p>
 * Represents an event, which may have multiple performances with different dates and  ↩
    venues.
 * </p>
 *
 * <p>
 * Event's principle members are it's relationship to {@link EventCategory} - specifying  ↩
    the type of event it is - and
 * {@link MediaItem} - providing the ability to add media (such as a picture) to the event  ↩
    for display. It also contains
 * meta-data about the event, such as it's name and a description.
 * </p>
 *
 */
@Entity
public class Event {

    /* Declaration of fields */

    /**
     * The synthetic ID of the object.
     */
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    /**
```

```
     * <p>
     * The name of the event.
     * </p>
     *
     * <p>
     * The name of the event forms it's natural identity and cannot be shared between ↩
         events.
     * </p>
     *
     * <p>
     * Two constraints are applied using Bean Validation
     * </p>
     *
     * <ol>
     * <li><code>@NotNull</code> &mdash; the name must not be null.</li>
     * <li><code>@Size</code> &mdash; the name must be at least 5 characters and no more ↩
         than 50 characters. This allows for
     * better formatting consistency in the view layer.</li>
     * </ol>
     */
    @Column(unique = true)
    @NotNull
    @Size(min = 5, max = 50, message = "An event's name must contain between 5 and 50 ↩
        characters")
    private String name;

    /**
     * <p>
     * A description of the event.
     * </p>
     *
     * <p>
     * Two constraints are applied using Bean Validation
     * </p>
     *
     * <ol>
     * <li><code>@NotNull</code> &mdash; the description must not be null.</li>
     * <li><code>@Size</code> &mdash; the name must be at least 20 characters and no more ↩
         than 1000 characters. This allows for
     * better formatting consistency in the view layer, and also ensures that event ↩
         organisers provide at least some description
     * - a classic example of a business constraint.</li>
     * </ol>
     */
    @NotNull
    @Size(min = 20, max = 1000, message = "An event's name must contain between 20 and 1000 ↩
         characters")
    private String description;

    /**
     * <p>
     * A media item, such as an image, which can be used to entice a browser to book a ↩
         ticket.
     * </p>
     *
     * <p>
     * Media items can be shared between events, so this is modeled as a <code>@ManyToOne</ ↩
         code> relationship.
     * </p>
     *
     * <p>
     * Adding a media item is optional, and the view layer will adapt if none is provided.
```

```
     * </p>
     *
     */
    @ManyToOne
    private MediaItem mediaItem;

    /**
     * <p>
     * The category of the event
     * </p>
     *
     * <p>
     * Event categories are used to ease searching of available of events, and hence this ↩
        is modeled as a relationship
     * </p>
     *
     * <p>
     * The Bean Validation constraint <code>@NotNull</code> indicates that the event ↩
        category must be specified.
     */
    @ManyToOne
    @NotNull
    private EventCategory category;

    /* Boilerplate getters and setters */

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public MediaItem getMediaItem() {
        return mediaItem;
    }

    public void setMediaItem(MediaItem picture) {
        this.mediaItem = picture;
    }

    public EventCategory getCategory() {
        return category;
    }

    public void setCategory(EventCategory category) {
        this.category = category;
    }

    public String getDescription() {
        return description;
    }
```

```
    public void setDescription(String description) {
        this.description = description;
    }

    /* toString(), equals() and hashCode() for Event, using the natural identity of the  ↩
        object */

    @Override
    public boolean equals(Object o) {
        if (this == o)
            return true;
        if (o == null || getClass() != o.getClass())
            return false;

        Event event = (Event) o;

        if (name != null ? !name.equals(event.name) : event.name != null)
            return false;

        return true;
    }

    @Override
    public int hashCode() {
        return name != null ? name.hashCode() : 0;
    }

    @Override
    public String toString() {
        return name;
    }
}
```

14.3 Shows

A show is an event at a venue. It consists of a set of performances of the show. A show also contains the list of ticket prices available.

Let's start building Show. Here's is our starting point:

src/main/java/org/jboss/jdf/example/ticketmonster/model/Show.java

```
/**
 * <p>
 * A show is an instance of an event taking place at a particular venue. A show can have  ↩
    multiple performances.
 * </p>
 */
@Entity
public class Show {

    /* Declaration of fields */

    /**
     * The synthetic id of the object.
     */
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
```

```
    /**
     * <p>
     * The event of which this show is an instance. The <code>@ManyToOne<code> JPA mapping ↩
        establishes this relationship.
     * </p>
     *
     * <p>
     * The <code>@NotNull</code> Bean Validation constraint means that the event must be ↩
        specified.
     * </p>
     */
    @ManyToOne
    @NotNull
    private Event event;

    /**
     * <p>
     * The venue where this show takes place. The <code>@ManyToOne<code> JPA mapping ↩
        establishes this relationship.
     * </p>
     *
     * <p>
     * The <code>@NotNull</code> Bean Validation constraint means that the venue must be ↩
        specified.
     * </p>
     */
    @ManyToOne
    @NotNull
    private Venue venue;

    /* Boilerplate getters and setters */

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public Event getEvent() {
        return event;
    }

    public void setEvent(Event event) {
        this.event = event;
    }

    public Venue getVenue() {
        return venue;
    }

    public void setVenue(Venue venue) {
        this.venue = venue;
    }

    /* toString(), equals() and hashCode() for Show, using the natural identity of the ↩
        object */
    @Override
    public boolean equals(Object o) {
        if (this == o)
            return true;
```

```
        if (o == null || getClass() != o.getClass())
            return false;

        Show show = (Show) o;

        if (event != null ? !event.equals(show.event) : show.event != null)
            return false;
        if (venue != null ? !venue.equals(show.venue) : show.venue != null)
            return false;

        return true;
    }

    @Override
    public int hashCode() {
        int result = event != null ? event.hashCode() : 0;
        result = 31 * result + (venue != null ? venue.hashCode() : 0);
        return result;
    }

    @Override
    public String toString() {
        return event + " at " + venue;
    }
}
```

If you've been paying attention, you'll notice that there is a problem here. We've identified that the natural identity of this entity is formed of two properties - the *event* and the *venue*, and we've correctly coded the `equals()` and `hashCode()` methods (or had them generated for us!). However, we haven't told JPA that these two properties, in combination, must be unique. As there are two properties involved, we can no longer use the `@Column` annotation (which operates on a single property/table column), but now must use the class level `@Table` annotation (which operates on the whole entity/table). Change the class definition to read:

src/main/java/org/jboss/jdf/example/ticketmonster/model/Show.java

```
...

@Entity
@Table(uniqueConstraints = @UniqueConstraint(columnNames = { "event_id", "venue_id" }))
public class Show {

    ...
}
```

You'll notice that JPA requires us to use the column names, rather than property names here. The column names used in the `@UniqueConstraint` annotation are those generated by default for properties called `event` and `venue`.

Now, let's add the set of performances to the event. Unlike previous relationships we've seen, the relationship between a show and it's performances is bi-directional. We chose to model this as a bi-directional relationship in order to improve the generated database schema (otherwise you end with complicated mapping tables which makes updates to collections hard). Let's add the set of performances:

src/main/java/org/jboss/jdf/example/ticketmonster/model/Show.java

```
    ...

    /**
     * <p>
     * The set of performances of this show.
     * </p>
     *
     * <p>
```

```
     * The <code>@OneToMany<code> JPA mapping establishes this relationship. Collection members
     * are fetched eagerly, so that they can be accessed even after the entity has become detached.
     * This relationship is bi-directional (a performance knows which show it is part of), and the <code>mappedBy</code>
     * attribute establishes this.
     * </p>
     *
     */
    @OneToMany(fetch=EAGER, mappedBy = "show", cascade = ALL)
    @OrderBy("date")
    private Set<Performance> performances = new HashSet<Performance>();

    ...

    public Set<Performance> getPerformances() {
        return performances;
    }

    public void setPerformances(Set<Performance> performances) {
        this.performances = performances;

    ...
```

As the relationship is bi-directional, we specify the `mappedBy` attribute on the `@OneToMany` annotation, which informs JPA to create a bi-directional relationship. The value of the attribute is name of property which forms the other side of the relationship - in this case, not unsuprisingly `show`!

As `Show` is the owner of `Performance` (and without a show, a performance cannot exist), we add the `cascade = ALL` attribute to the `@OneToMany` annotation. As a result, any persistence operation that occurs on a show, will be propagated to it's performances. For example, if a show is removed, any associated performances will be removed as well.

When retrieving a show, we will also retrieve its associated performances by adding the `fetch = EAGER` attribute to the `@OneToMany` annotation. This is a design decision which required careful consideration. In general, you should favour the default lazy initialization of collections: their content should be accessible on demand. However, in this case we intend to marshal the contents of the collection and pass it across the wire in the JAX-RS layer, after the entity has become detached, and cannot initialize its members on demand.

We'll also need to add the set of ticket prices available for this show. Once more, this is a bi-directional relationship, owned by the show. It looks just like the set of performances.

Here's the full source for `Show`:

src/main/java/org/jboss/jdf/example/ticketmonster/model/Show.java

```
/**
 * <p>
 * A show is an instance of an event taking place at a particular venue. A show can have multiple performances.
 * </p>
 *
 * <p>
 * A show contains a set of performances, and a set of ticket prices for each section of the venue for this show.
 * </p>
 *
 * <p>
 * The event and venue form the natural id of this entity, and therefore must be unique. JPA requires us to use the class level
 * <code>@Table</code> constraint.
 * </p>
 *
```

```
 */
/*
 * We suppress the warning about not specifying a serialVersionUID, as we are still  ↩
    developing this app, and want the JVM to
 * generate the serialVersionUID for us. When we put this app into production, we'll  ↩
    generate and embed the serialVersionUID
 */
@SuppressWarnings("serial")
@Entity
@Table(uniqueConstraints = @UniqueConstraint(columnNames = { "event_id", "venue_id" }))
public class Show implements Serializable {

    /* Declaration of fields */

    /**
     * The synthetic id of the object.
     */
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    /**
     * <p>
     * The event of which this show is an instance. The <code>@ManyToOne<code> JPA mapping  ↩
        establishes this relationship.
     * </p>
     *
     * <p>
     * The <code>@NotNull</code> Bean Validation constraint means that the event must be  ↩
        specified.
     * </p>
     */
    @ManyToOne
    @NotNull
    private Event event;

    /**
     * <p>
     * The event of which this show is an instance. The <code>@ManyToOne<code> JPA mapping  ↩
        establishes this relationship.
     * </p>
     *
     * <p>
     * The <code>@NotNull</code> Bean Validation constraint means that the event must be  ↩
        specified.
     * </p>
     */
    @ManyToOne
    @NotNull
    private Venue venue;

    /**
     * <p>
     * The set of performances of this show.
     * </p>
     *
     * <p>
     * The <code>@OneToMany<code> JPA mapping establishes this relationship. TODO Explain  ↩
        EAGER fetch.
     * This relationship is bi-directional (a performance knows which show it is part of),  ↩
        and the <code>mappedBy</code>
     * attribute establishes this. We cascade all persistence operations to the set of  ↩
```

```
     performances, so, for example if a show
     * is removed, then all of it's performances will also be removed.
     * </p>
     *
     * <p>
     * Normally a collection is loaded from the database in the order of the rows, but here ↩
         we want to make sure that
     * performances are ordered by date - we let the RDBMS do the heavy lifting. The
     * <code>@OrderBy<code> annotation instructs JPA to do this.
     * </p>
     */
    @OneToMany(fetch = EAGER, mappedBy = "show", cascade = ALL)
    @OrderBy("date")
    private Set<Performance> performances = new HashSet<Performance>();

    /**
     * <p>
     * The set of ticket prices available for this show.
     * </p>
     *
     * <p>
     * The <code>@OneToMany<code> JPA mapping establishes this relationship.
     * This relationship is bi-directional (a ticket price category knows which show it is ↩
         part of), and the <code>mappedBy</code>
     * attribute establishes this. We cascade all persistence operations to the set of ↩
         performances, so, for example if a show
     * is removed, then all of it's ticket price categories are also removed.
     * </p>
     */
    @OneToMany(mappedBy = "show", cascade = ALL, fetch = EAGER)
    private Set<TicketPrice> ticketPrices = new HashSet<TicketPrice>();

    /* Boilerplate getters and setters */

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public Event getEvent() {
        return event;
    }

    public void setEvent(Event event) {
        this.event = event;
    }

    public Set<Performance> getPerformances() {
        return performances;
    }

    public void setPerformances(Set<Performance> performances) {
        this.performances = performances;
    }

    public Venue getVenue() {
        return venue;
    }
```

```
    public void setVenue(Venue venue) {
        this.venue = venue;
    }

    public Set<TicketPrice> getTicketPrices() {
        return ticketPrices;
    }

    public void setTicketPrices(Set<TicketPrice> ticketPrices) {
        this.ticketPrices = ticketPrices;
    }

    /* toString(), equals() and hashCode() for Show, using the natural identity of the  ↩
       object */
    @Override
    public boolean equals(Object o) {
        if (this == o)
            return true;
        if (o == null || getClass() != o.getClass())
            return false;

        Show show = (Show) o;

        if (event != null ? !event.equals(show.event) : show.event != null)
            return false;
        if (venue != null ? !venue.equals(show.venue) : show.venue != null)
            return false;

        return true;
    }

    @Override
    public int hashCode() {
        int result = event != null ? event.hashCode() : 0;
        result = 31 * result + (venue != null ? venue.hashCode() : 0);
        return result;
    }

    @Override
    public String toString() {
        return event + " at " + venue;
    }
}
```

14.4 Performances

Finally, let's create the `Performance` class, which represents an instance of a `Show`. Performance is pretty straightforward. It contains the date and time of the performance, and the show of which it is a performance. Together, the show, and the date and time, make up the natural identity of the performance. Here's the source for `Performance`:

src/main/java/org/jboss/jdf/example/ticketmonster/model/Performance.java

```
/**
 * <p>
 * A performance represents a single instance of a show.
 * </p>
 *
 * <p>
 * The show and date form the natural id of this entity, and therefore must be unique. JPA  ↩
    requires us to use the class level
```

```
 * <code>@Table</code> constraint.
 * </p>
 *
 */
@Entity
@Table(uniqueConstraints = @UniqueConstraint(columnNames = { "date", "show_id" }))
public class Performance {

    /* Declaration of fields */

    /**
     * The synthetic id of the object.
     */
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    /**
     * <p>
     * The date and start time of the performance.
     * </p>
     *
     * <p>
     * A Java {@link Date} object represents both a date and a time, whilst an RDBMS splits ↩
         out Date, Time and Timestamp.
     * Therefore we instruct JPA to store this date as a timestamp using the <code> ↩
        @Temporal(TIMESTAMP)</code> annotation.
     * </p>
     *
     * <p>
     * The date and time of the performance is required, and the Bean Validation constraint ↩
         <code>@NotNull</code> enforces this.
     * </p>
     */
    @Temporal(TIMESTAMP)
    @NotNull
    private Date date;

    /**
     * <p>
     * The show of which this is a performance. The <code>@ManyToOne<code> JPA mapping ↩
        establishes this relationship.
     * </p>
     *
     * <p>
     * The show of which this is a performance is required, and the Bean Validation ↩
        constraint <code>@NotNull</code> enforces
     * this.
     * </p>
     */
    @ManyToOne
    @NotNull
    private Show show;

    /* Boilerplate getters and setters */

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
```

```
    }

    public void setShow(Show show) {
        this.show = show;
    }

    public Show getShow() {
        return show;
    }

    public Date getDate() {
        return date;
    }

    public void setDate(Date date) {
        this.date = date;
    }

    /* equals() and hashCode() for Performance, using the natural identity of the object */

    @Override
    public boolean equals(Object o) {
        if (this == o)
            return true;
        if (o == null || getClass() != o.getClass())
            return false;

        Performance that = (Performance) o;

        if (date != null ? !date.equals(that.date) : that.date != null)
            return false;
        if (show != null ? !show.equals(that.show) : that.show != null)
            return false;

        return true;
    }

    @Override
    public int hashCode() {
        int result = date != null ? date.hashCode() : 0;
        result = 31 * result + (show != null ? show.hashCode() : 0);
        return result;
    }
}
```

Of interest here is the storage of the date and time.

A Java `Date` represents "a specific instance in time, with millisecond precision" and is the recommended construct for representing date and time in the JDK. A RDBMS's *DATE* type typically has day precision only, and uses the *DATETIME* or *TIMESTAMP* types to represent an instance in time, and often only to second precision.

As the mapping between Java date and time, and database date and time isn't straightforward, JPA requires us to use the `@Temporal` annotation on any property of type `Date`, and to specify whether the `Date` should be stored as a date, a time or a timestamp (date and time).

14.5 Venue

Now, let's build out the entities to represent the venue.

We start by adding an entity to represent the venue. A venue needs to have a name, a description, a capacity, an address, an

associated media item and a set sections in which people can sit. If you completed the introduction chapter, you should already have some of these properties set, so we will update the `Venue` class to look like in the definition below.

src/main/java/org/jboss/jdf/example/ticketmonster/model/Venue.java

```
/**
 * <p>
 * Represents a single venue
 * </p>
 *
 */
@Entity
public class Venue {

    /* Declaration of fields */

    /**
     * The synthetic id of the object.
     */
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    /**
     * <p>
     * The name of the event.
     * </p>
     *
     * <p>
     * The name of the event forms it's natural identity and cannot be shared between ↩
        events.
     * </p>
     *
     * <p>
     * The name must not be null and must be one or more characters, the Bean Validation
     * constraint <code>@NotEmpty</code> enforces this.
     * </p>
     */
    @Column(unique = true)
    @NotEmpty
    private String name;

    /**
     * The address of the venue
     */
    @Embedded
    private Address address = new Address();

    /**
     * A description of the venue
     */
    private String description;

    /**
     * <p>
     * A set of sections in the venue
     * </p>
     *
     * <p>
     * The <code>@OneToMany<code> JPA mapping establishes this relationship.
     * Collection members are fetched eagerly, so that they can be accessed even after the
     * entity has become detached. This relationship is bi-directional (a section knows ↩
        which
```

```
     * venue it is part of), and the <code>mappedBy</code> attribute establishes this. We
     * cascade all persistence operations to the set of performances, so, for example if a venue
     * is removed, then all of it's sections will also be removed.
     * </p>
     */
    @OneToMany(cascade = ALL, fetch = EAGER, mappedBy = "venue")
    private Set<Section> sections = new HashSet<Section>();

    /**
     * The capacity of the venue
     */
    private int capacity;

    /**
     * An optional media item to entice punters to the venue. The <code>@ManyToOne</code> establishes the relationship.
     */
    @ManyToOne
    private MediaItem mediaItem;

    /* Boilerplate getters and setters */

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Address getAddress() {
        return address;
    }

    public void setAddress(Address address) {
        this.address = address;
    }

    public MediaItem getMediaItem() {
        return mediaItem;
    }

    public void setMediaItem(MediaItem description) {
        this.mediaItem = description;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }
```

```
    public Set<Section> getSections() {
        return sections;
    }

    public void setSections(Set<Section> sections) {
        this.sections = sections;
    }

    public int getCapacity() {
        return capacity;
    }

    public void setCapacity(int capacity) {
        this.capacity = capacity;
    }

    /* toString(), equals() and hashCode() for Venue, using the natural identity of the  ↩
       object */

    @Override
    public boolean equals(Object o) {
        if (this == o)
            return true;
        if (o == null || getClass() != o.getClass())
            return false;

        Venue venue = (Venue) o;

        if (address != null ? !address.equals(venue.address) : venue.address != null)
            return false;
        if (name != null ? !name.equals(venue.name) : venue.name != null)
            return false;

        return true;
    }

    @Override
    public int hashCode() {
        int result = name != null ? name.hashCode() : 0;
        result = 31 * result + (address != null ? address.hashCode() : 0);
        return result;
    }

    @Override
    public String toString() {
        return name;
    }
}
```

In creating this entity, we've followed all the design and implementation decisions previously discussed, with one new concept. Rather than add the properties for street, city, postal code etc. to this object, we've extracted them into the `Address` object, and included it in the `Venue` object using composition. This would allow us to reuse the Address object in other places (such as a customer's address).

A RDBMS doesn't have a similar concept to composition, so we need to choose whether to represent the address as a separate entity, and create a relationship between the venue and the address, or whether to map the properties from `Address` to the table for the owning entity, in this case `Venue`. It doesn't make much sense for an address to be a full entity - we're not going to want to run queries against the address in isolation, nor do we want to be able to delete or update an address in isolation - in essence, the address doesn't have a standalone identity outside of the object into which it is composed.

To *embed* the `Address` into `Venue` we add the `@Embeddable` annotation to the `Address` class. However, unlike a full

entity, there is no need to add an identifier. Here's the source for `Address`:

src/main/java/org/jboss/jdf/example/ticketmonster/model/Address.java

```
/**
 * <p>
 * A reusable representation of an address.
 * </p>
 *
 * <p>
 * Addresses are used in many places in an application, so to observe the DRY principle, we ↩
     model Address as an embeddable
 * entity. An embeddable entity appears as a child in the object model, but no relationship ↩
     is established in the RDBMS..
 * </p>
 */
@Embeddable
public class Address {

    /* Declaration of fields */
    private String street;
    private String city;
    private String country;

    /* Declaration of boilerplate getters and setters */

    public String getStreet() {
        return street;
    }

    public void setStreet(String street) {
        this.street = street;
    }

    public String getCity() {
        return city;
    }

    public void setCity(String city) {
        this.city = city;
    }

    public String getCountry() {
        return country;
    }

    public void setCountry(String country) {
        this.country = country;
    }

    /* toString(), equals() and hashCode() for Address, using the natural identity of the ↩
        object */

    @Override
    public boolean equals(Object o) {
        if (this == o)
            return true;
        if (o == null || getClass() != o.getClass())
            return false;

        Address address = (Address) o;

        if (city != null ? !city.equals(address.city) : address.city != null)
```

```
            return false;
        if (country != null ? !country.equals(address.country) : address.country != null)
            return false;
        if (street != null ? !street.equals(address.street) : address.street != null)
            return false;

        return true;
    }

    @Override
    public int hashCode() {
        int result = street != null ? street.hashCode() : 0;
        result = 31 * result + (city != null ? city.hashCode() : 0);
        result = 31 * result + (country != null ? country.hashCode() : 0);
        return result;
    }

    @Override
    public String toString() {
        return street + ", " + city + ", " + country;
    }
}
```

14.6 Sections

A venue consists of a number of seating sections. Each seating section has a name, a description, the number of rows in the section, and the number of seats in a row. It's natural identifier is the name of section combined with the venue (a venue can't have two sections with the same name). `Section` doesn't introduce any new concepts, so go ahead and copy the source in, if you are building the application whilst following this tutorial.

14.7 Booking, Ticket & Seat

There aren't many new concepts to explore in `Booking`, `Ticket` and `Seat`, so if you are following along with the tutorial, you should copy in the `Booking`, `Ticket` and `Seat` classes.

Once the user has selected an event, identified the venue, and selected a performance, they have the opportunity to request a number of seats in a given section, and select the category of tickets required. Once they chosen their seats, and entered their email address, a `Booking` is created.

A booking consists of the date the booking was created, an email address (as TicketMonster doesn't yet have fully fledged user management), a set of tickets and the associated performance. The set of tickets shows us how to create a uni-directional one-to-many relationship:

src/main/java/org/jboss/jdf/example/ticketmonster/model/Booking.java

```
    ...

    /**
     * <p>
     * The set of tickets contained within the booking. The <code>@OneToMany<code> JPA ↩
        mapping establishes this relationship.
     * </p>
     *
     * <p>
     * The set of tickets is eagerly loaded because FIXME . All operations are cascaded to ↩
        each ticket, so for example if a
     * booking is removed, then all associated tickets will be removed.
     * </p>
```

```
     *
     * <p>
     * This relationship is uni-directional, so we need to inform JPA to create a foreign ↩
         key mapping. The foreign key mapping
     * is not visible in the {@link Ticket} entity despite being present in the database.
     * </p>
     *
     */
    @OneToMany(fetch = EAGER, cascade = ALL)
    @JoinColumn    @NotEmpty
    @Valid
    private Set<Ticket> tickets = new HashSet<Ticket>();

    ...
```

We add the `@JoinColumn` annotation, which sets up a foreign key in `Ticket`, but doesn't expose the booking on Ticket. This prevents the use of messy mapping tables, whilst preserving the integrity of the entity model.

A ticket embeds the seat allocated, and contains a reference to the category under which it was sold. It also contains the price at which it was sold.

Chapter 15

Connecting to the database

In this example, we are using the in-memory H2 database, which is very easy to set up on JBoss AS. JBoss AS allows you deploy a datasource inside your application's `WEB-INF` directory. You can locate the source in `src/main/webapp/WEB-INF/ticket-mon`

src/main/webapp/WEB-INF/ticket-monster-ds.xml

```
<datasources xmlns="http://www.jboss.org/ironjacamar/schema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.jboss.org/ironjacamar/schema http://docs.jboss.org/ ↩
        ironjacamar/schema/datasources_1_0.xsd">
    <!-- The datasource is bound into JNDI at this location. We reference
        this in META-INF/persistence.xml -->
    <datasource jndi-name="java:jboss/datasources/ticket-monsterDS"
        pool-name="ticket-monster" enabled="true" use-java-context="true">
        <connection-url>jdbc:h2:mem:ticket-monster;DB_CLOSE_ON_EXIT=FALSE;DB_CLOSE_DELAY=-1 ↩
            </connection-url>
        <driver>h2</driver>
        <security>
            <user-name>sa</user-name>
            <password>sa</password>
        </security>
    </datasource>
</datasources>
```

The datasource configures an H2 in-memory database, called *ticket-monster*, and registeres a datasource in JNDI at the address:

`java:jboss/datasources/ticket-monsterDS`

Now we need to configure JPA to use the datasource. This is done in `src/main/resources/META-INF/persistence.xml`:

src/main/resources/persistence.xml

```
<persistence version="2.0"
   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_2_0.xsd">
   <persistence-unit name="primary">
      <!-- If you are running in a production environment, add a managed
         data source, this example data source is just for development and testing! -->
      <!-- The datasource is deployed as WEB-INF/ticket-monster-ds.xml, you
         can find it in the source at src/main/webapp/WEB-INF/ticket-monster-ds.xml -->
      <jta-data-source>java:jboss/datasources/ticket-monsterDS</jta-data-source>
      <properties>
         <!-- Properties for Hibernate -->
```

```
         <property name="hibernate.hbm2ddl.auto" value="create-drop" />
         <property name="hibernate.show_sql" value="false" />
      </properties>
   </persistence-unit>
</persistence>
```

As our application has only one datasource, and hence one persistence unit, the name given to the persistence unit doesn't really matter. We call ours `primary`, but you can change this as you like. We tell JPA about the datasource bound in JNDI.

Hibernate includes the ability to generate tables from entities, which here we have configured. We don't recommend using this outside of development. Updates to databases in production should be done manually.

Chapter 16

Populating test data

Whilst we develop our application, it's useful to be able to populate the database with test data. Luckily, Hibernate makes this easy. Just add a file called `import.sql` onto the classpath of your application (we keep it in `src/main/resources/import.sql`). In it, we just write standard sql statements suitable for the database we are using. To do this, you need to know the generated column and table names for your entities. The best way to work these out is to look at the h2console.

The h2console is included in the JBoss AS quickstarts, along with instructions on how to use it. For more information, see http://jboss.org/jdf/quickstarts/jboss-as-quickstart/h2-console/

Where do I look for my data?
The database URL is `jdbc:h2:mem:ticket-monster`. After you have downloaded `h2console.war` and deployed it on the server, make sure that the application is running on the server and use this value to connect to your running application's database.

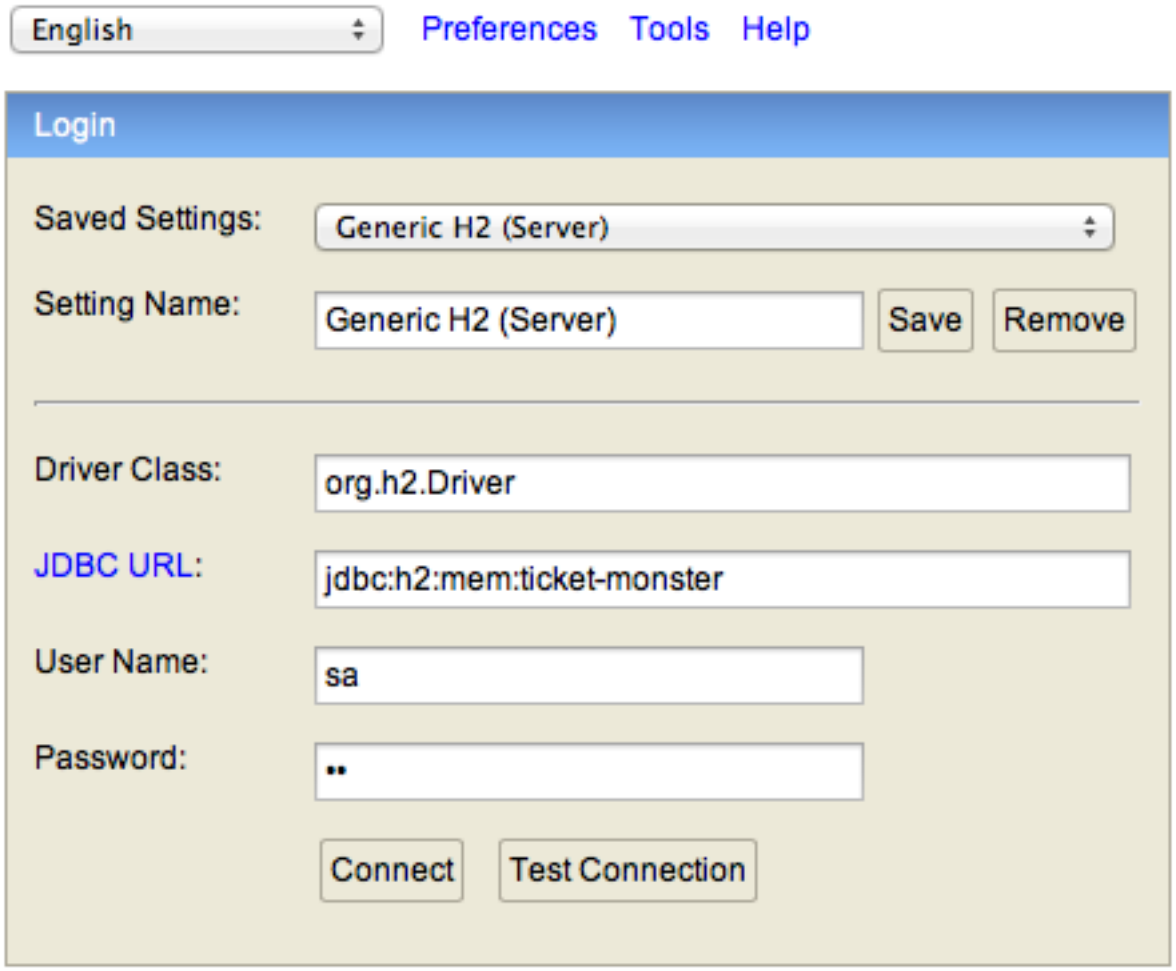

Figure 16.1: h2console settings

Chapter 17

Conclusion

You now have a working data model for your TicketMonster application, our next tutorial will show you how to create the business services layer or something like that - it seems to end abruptly.

Part III

Building The Business Services With JAX-RS

Chapter 18

What Will You Learn Here?

We've just defined the domain model of the application and created its persistence layer. Now we need to define the services that implement the business logic of the application and expose them to the front-end. After reading this, you'll understand how to design the business layer and what choices to make while developing it. Topics covered include:

- Encapsulating business logic in services and integrating with the persistence tier
- Using CDI for integrating individual services
- Integration testing using Arquillian
- Exposing RESTful services via JAX-RS

The tutorial will show you how to perform all these steps in JBoss Developer Studio, including screenshots that guide you through.

Chapter 19

Business Services And Their Relationships

TicketMonster's business logic is implemented by a number of classes, with different responsibilities:

- managing media items
- allocating tickets
- handling information on ticket availability
- remote access through a RESTful interface

The services are consumed by various other layers of the application:

- the media management and ticket allocation services encapsulate complex functionality, which in turn is exposed externally by RESTful services that wrap them
- RESTful services are mainly used by the HTML5 view layer
- the ticket availability service is used by the Errai-based view layer

---

Where to draw the line?
A business service is an encapsulated, reusable logical component that groups together a number of well-defined cohesive business operations. Business services perform business operations, and may coordinate infrastructure services such as persistence units, or even other business services as well. The boundaries drawn between them should take into account whether the newly created services represent , potentially reusable components.

As you can see, some of the services are intended to be consumed within the business layer of the application, while others provide an external interface as JAX-RS services. We will start by implementing the former, and we'll finish up with the latter. During this process, you will discover how CDI, EJB and JAX-RS make it easy to define and wire together our services.

Chapter 20

Preparations

20.1 Enabling CDI

The first step for setting up our service architecture is to enable CDI in the deployment by creating a `beans.xml` file in the `WEB-INF` folder of the web application.

src/main/webapp/WEB-INF/beans.xml

```
<beans xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                           http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
</beans>
```

If you used the Maven archetype
If you used the Maven archetype to create the project, this file will exist already in the project - it is added automatically.

You may wonder why the file is empty! Whilst `beans.xml` can specify various deployment-time configuration (e.g. activation of interceptors, decorators or alternatives), it can also act as a marker file, telling the container to enable CDI for the deployment (which it doesn't do, unless `beans.xml` is present).

Contexts and Dependency Injection (CDI)
As it's name suggests, CDI is the contexts and dependency injection standard for Java EE. By enabling CDI in your application, deployed classes become managed components and their lifecycle and wiring becomes the responsibility of the Java EE server. In this way, we can reduce coupling between components, which is a requirement o a well-designed architecture. Now, we can focus on implementing the responsibilities of the components and describing their dependencies in a declarative fashion. The runtime will do the rest for you: instantiating and wiring them together, as well as disposing of them as needed.

20.2 Adding utility classes

Next, we add some helper classes providing low-level utilities for the application. We won't get in their implementation details here, but you can study their source code for details.

Copy the following classes from the original example to `src/main/java/org/jboss/jdf/example/ticketmonster/uti`

- `Base64`

- `ForwardingMap`
- `MultivaluedHashMap`
- `Reflections`
- `Resources`

Chapter 21

Internal Services

We begin the service implementation by implementing some helper services.

21.1 The Media Manager

First, let's add support for managing media items, such as images. The persistence layer siimply stores URLs, referencing media items stored by online services. The URL look like http://dl.dropbox.com/u/65660684/640px-Roy_Thomson_Hall_Toronto.jpg.

Now, we could use the URLs in our application, and retrieve these media items from the provider. However, we would prefer to cache these media items in order to improve application performance and increase resilience to external failures - this will allow us to run the application successfully even if the provider is down. The `MediaManager` is a good illustration of a business service; it performs the retrieval and caching of media objects, encapsulating the operation from the rest of the application.

We begin by creating `MediaManager`:

src/main/java/org/jboss/jdf/example/ticketmonster/service/MediaManager.java

```
/**
 * <p>
 * The media manager is responsible for taking a media item, and returning either the URL
 * of the cached version (if the application cannot load the item from the URL), or the
 * original URL.
 * </p>
 *
 * <p>
 * The media manager also transparently caches the media items on first load.
 * </p>
 *
 * <p>
 * The computed URLs are cached for the duration of a request. This provides a good balance
 * between consuming heap space, and computational time.
 * </p>
 *
 */
public class MediaManager {

    /**
     * Locate the tmp directory for the machine
     */
    private static final File tmpDir;

    static {
        tmpDir = new File(System.getProperty("java.io.tmpdir"),
                                    "org.jboss.jdf.examples.ticket-monster");
```

```
        if (tmpDir.exists()) {
            if (tmpDir.isFile())
                throw new IllegalStateException(tmpDir.getAbsolutePath()
                                                + " already exists, and is a file. Remove it ↩
                                                    .");
        } else {
            tmpDir.mkdir();
        }
    }

    /**
     * A request scoped cache of computed URLs of media items.
     */
    private final Map<MediaItem, MediaPath> cache;

    public MediaManager() {

        this.cache = new HashMap<MediaItem, MediaPath>();
    }

    /**
     * Load a cached file by name
     *
     * @param fileName
     * @return
     */
    public File getCachedFile(String fileName) {
        return new File(tmpDir, fileName);
    }

    /**
     * Obtain the URL of the media item. If the URL h has already been computed in this
         * request, it will be looked up in the request scoped cache, otherwise it will be
         * computed, and placed in the request scoped cache.
     */
    public MediaPath getPath(MediaItem mediaItem) {
        if (cache.containsKey(mediaItem)) {
            return cache.get(mediaItem);
        } else {
            MediaPath mediaPath = createPath(mediaItem);
            cache.put(mediaItem, mediaPath);
            return mediaPath;
        }
    }

    /**
     * Compute the URL to a media item. If the media item is not cacheable, then, as long
         * as the resource can be loaded, the original URL is returned. If the resource is ↩
             not
         * available, then a placeholder image replaces it. If the media item is cachable, ↩
             it
         * is first cached in the tmp directory, and then path to load it is returned.
     */
    private MediaPath createPath(MediaItem mediaItem) {
        if (!mediaItem.getMediaType().isCacheable()) {
            if (checkResourceAvailable(mediaItem)) {
                return new MediaPath(mediaItem.getUrl(), false, mediaItem.getMediaType());
            } else {
                return createCachedMedia(Reflections.getResource("not_available.jpg"). ↩
                    toExternalForm(), IMAGE);
            }
        } else {
```

```
            return createCachedMedia(mediaItem);
        }
    }

    /**
     * Check if a media item can be loaded from it's URL, using the JDK URLConnection ↩
         classes.
     */
    private boolean checkResourceAvailable(MediaItem mediaItem) {
        URL url = null;
        try {
            url = new URL(mediaItem.getUrl());
        } catch (MalformedURLException e) {
        }

        if (url != null) {
            try {
                URLConnection connection = url.openConnection();
                if (connection instanceof HttpURLConnection) {
                    return ((HttpURLConnection) connection).getResponseCode() == ↩
                        HttpURLConnection.HTTP_OK;
                } else {
                    return connection.getContentLength() > 0;
                }
            } catch (IOException e) {
            }
        }
        return false;
    }

    /**
     * The cached file name is a base64 encoded version of the URL. This means we don't ↩
         need to maintain a database of cached
     * files.
     */
    private String getCachedFileName(String url) {
        return Base64.encodeToString(url.getBytes(), false);
    }

    /**
     * Check to see if the file is already cached.
     */
    private boolean alreadyCached(String cachedFileName) {
        File cache = getCachedFile(cachedFileName);
        if (cache.exists()) {
            if (cache.isDirectory()) {
                throw new IllegalStateException(cache.getAbsolutePath() + " already exists, ↩
                     and is a directory. Remove it.");
            }
            return true;
        } else {
            return false;
        }
    }

    /**
     * To cache a media item we first load it from the net, then write it to disk.
     */
    private MediaPath createCachedMedia(String url, MediaType mediaType) {
        String cachedFileName = getCachedFileName(url);
        if (!alreadyCached(cachedFileName)) {
            URL _url = null;
```

```
            try {
                _url = new URL(url);
            } catch (MalformedURLException e) {
                throw new IllegalStateException("Error reading URL " + url);
            }

            try {
                InputStream is = null;
                OutputStream os = null;
                try {
                    is = new BufferedInputStream(_url.openStream());
                    os = new BufferedOutputStream(getCachedOutputStream(cachedFileName));
                    while (true) {
                        int data = is.read();
                        if (data == -1)
                            break;
                        os.write(data);
                    }
                } finally {
                    if (is != null)
                        is.close();
                    if (os != null)
                        os.close();
                }
            } catch (IOException e) {
                throw new IllegalStateException("Error caching " + mediaType.getDescription ↩
                    (), e);
            }
        }
        return new MediaPath(cachedFileName, true, mediaType);
    }

    private MediaPath createCachedMedia(MediaItem mediaItem) {
        return createCachedMedia(mediaItem.getUrl(), mediaItem.getMediaType());
    }

    private OutputStream getCachedOutputStream(String fileName) {
        try {
            return new FileOutputStream(getCachedFile(fileName));
        } catch (FileNotFoundException e) {
            throw new IllegalStateException("Error creating cached file", e);
        }
    }

}
```

The service delegates to a number of internal methods that do the heavy lifting, but exposes a simple API, to the external observer it simply converts the `MediaItem` entities into `MediaPath` data structures, that can be used by the application to load the binary data of the media item. The service will retrieve and cache the data locally in the filesystem, if possible (e.g. streamed videos aren't cachable!).

src/main/java/org/jboss/jdf/example/ticketmonster/service/MediaPath.java

```
public class MediaPath {

    private final String url;
    private final boolean cached;
    private final MediaType mediaType;

    public MediaPath(String url, boolean cached, MediaType mediaType) {
        this.url = url;
        this.cached = cached;
```

```
        this.mediaType = mediaType;
    }

    public String getUrl() {
        return url;
    }

    public boolean isCached() {
        return cached;
    }

    public MediaType getMediaType() {
        return mediaType;
    }

}
```

The service can be injected by type into the components that depend on it. However, in order to make it available to JSF views, we add a `@Named` annotation, which means the bean can be referenced as `mediaManager` as well.

We should also control the lifecycle of this service. The `MediaManager` stores request-specific state, so should be scoped to the web request, the CDI `@RequestScoped` is perfect.

src/main/java/org/jboss/jdf/example/ticketmonster/service/MediaManager.java

```
   ...
@Named
@RequestScoped
public class MediaManager {
   ...
}
```

21.2 The Seat Allocation Service

The seat allocation service finds free seats at booking time, in a given section of the venue. It is a good example of how a service can coordinate infrastructure services (using the injected persistence unit to get access to the `ServiceAllocation` instance) and domain objects (by invoking the `allocateSeats` method on a concrete allocation instance).

Isolating this functionality in a service class makes it possible to write simpler, self-explanatory code in the layers above and opens the possibility of replacing this code at a later date with a more advanced implementation (for example one using an in-memory cache).

src/main/java/org/jboss/jdf/example/ticketmonster/service/SeatAllocationService.java

```
@SuppressWarnings("serial")
public class SeatAllocationService implements Serializable {

    @Inject
    EntityManager entityManager;

    public AllocatedSeats allocateSeats(Section section, Performance performance, int ↩
        seatCount, boolean contiguous) {
        SectionAllocation sectionAllocation = retrieveSectionAllocationExclusively(section, ↩
            performance);
        List<Seat> seats = sectionAllocation.allocateSeats(seatCount, contiguous);
        return new AllocatedSeats(sectionAllocation, seats);
    }

    public void deallocateSeats(Section section, Performance performance, List<Seat> seats) ↩
        {
```

```
        SectionAllocation sectionAllocation = retrieveSectionAllocationExclusively(section, ↩
            performance);
        for (Seat seat : seats) {
            if (!seat.getSection().equals(section)) {
                throw new SeatAllocationException("All seats must be in the same section!") ↩
                    ;
            }
            sectionAllocation.deallocate(seat);
        }
    }

    private SectionAllocation retrieveSectionAllocationExclusively(Section section, ↩
        Performance performance) {
        SectionAllocation sectionAllocationStatus = (SectionAllocation) entityManager. ↩
            createQuery(
```

```
        entityManager.lock(sectionAllocationStatus, LockModeType.PESSIMISTIC_WRITE);
        return sectionAllocationStatus;
    }
}
```

Next, we define the `AllocatedSeats` class that we use for storing seat reservations for a booking, before they are made persistent.

src/main/java/org/jboss/jdf/example/ticketmonster/service/AllocatedSeats.java

```
public class AllocatedSeats {

    private final SectionAllocation sectionAllocation;

    private final List<Seat> seats;

    public AllocatedSeats(SectionAllocation sectionAllocation, List<Seat> seats) {
        this.sectionAllocation = sectionAllocation;
        this.seats = seats;
    }

    public SectionAllocation getSectionAllocation() {
        return sectionAllocation;
    }

    public List<Seat> getSeats() {
        return seats;
```

```
    }

    public void markOccupied() {
        sectionAllocation.markOccupied(seats);
    }
}
```

21.3 Booking Monitor Service

The last service that we create provides data about the current shows and their ticket availability status. It is accessed remotely by Errai through a dedicated RPC mechanism, which requires us to define and implement a service interface. We begin by adding the interface first, using the `@Remote` annotation from Errai to indicate its purpose.

src/main/java/org/jboss/jdf/example/ticketmonster/monitor/client/shared/BookingMonitorService.java

```
/**
 * A service used by the booking monitor for retrieving status information.
 *
 * Errai's @Remote annotation indicates that the Service implementation can
 * be used as an RPC endpoint and that this interface can be used on the
 * client for type safe method invocation on this endpoint.
 */
@Remote
public interface BookingMonitorService {

    /**
     * Lists all active {@link Show}s (shows with future performances).
     *
     * @return list of shows found.
     */
    public List<Show> retrieveShows();

    /**
     * Constructs a map of performance IDs to the total number of sold tickets.
     *
     * @return map of performance IDs to the total number of sold tickets.
     */
    public Map<Long, Long> retrieveOccupiedCounts();
}
```

After doing so, we create the service implementation, using the `@Service` annotation to indicate that it should be exposed externally by Errai.

src/main/java/org/jboss/jdf/example/ticketmonster/service/BookingMonitorServiceImpl.java

```
/**
 * Implementation of {@link BookingMonitorService}.
 *
 * Errai's @Service annotation exposes this service as an RPC endpoint.
 */
@ApplicationScoped
@Service
@SuppressWarnings("unchecked")
public class BookingMonitorServiceImpl implements BookingMonitorService {

    @Inject
    private EntityManager entityManager;

    @Override
    public List<Show> retrieveShows() {
```

```
        Query showQuery = entityManager.createQuery(
                "select DISTINCT s from Show s JOIN s.performances p " +
                "WHERE p.date > current_timestamp");
        return showQuery.getResultList();
    }

    @Override
    public Map<Long, Long> retrieveOccupiedCounts() {
        Map <Long, Long> occupiedCounts = new HashMap<Long, Long>();

        Query occupiedCountsQuery = entityManager.createQuery(
                        "select s.performance.id, SUM(s.occupiedCount) from  ↩
                            SectionAllocation s " +
                        "where s.performance.date > current_timestamp GROUP BY s. ↩
                            performance.id");

        List<Object[]> results = occupiedCountsQuery.getResultList();
        for (Object[] result : results) {
            occupiedCounts.put((Long) result[0], (Long) result[1]);
        }

        return occupiedCounts;
    }
}
```

Implement an interface or not?

You will find yourself very often facing a dilemma: add an interface for a service or not? As you have seen so far and will continue to see next, most of the services in TicketMonster do not implement interfaces, except wherever it is a requirement of the framework in use (e.g. Errai in this case). In Java EE 6 the requirements for business services to implement interfaces have been relaxed significantly, therefore unless there are valid reasons for creating an abstraction (such as multiple possible implementations), we skipped adding interfaces to our services.

Chapter 22

JAX-RS Services

The majority of services in the application are JAX-RS web services. They are critical part of the design, as they next service is used for provide communication with the HTML5 view layer. The JAX-RS services range from simple CRUD to processing bookings and media items.

To pass data across the wire we use JSON as the data marshalling format, as it is less verbose and easier to process than XML by the JavaScript client-side framework.

22.1 Initializing JAX-RS

To activate JAX-RS we add the class below, which instructs the container to look for JAX-RS annotated classes and install them as endpoints. This class should exist already in your project, as it is generated by the archetype, so make sure that it is there and it contains the code below:

src/main/java/org/jboss/jdf/example/ticketmonster/rest/JaxRsActivator.java

```
@ApplicationPath("/rest")
public class JaxRsActivator extends Application {
   /* class body intentionally left blank */
}
```

All the JAX-RS services are mapped relative to the `/rest` path, as defined by the `@ApplicationPath` annotation.

22.2 A Base Service For Read Operations

Most JAX-RS services must provide both a (filtered) list of entities or individual entity (e.g. events, venues and bookings). Instead of duplicating the implementation into each individual service we create a base service class and wire the helper objects in.

src/main/java/org/jboss/jdf/example/ticketmonster/rest/BaseEntityService.java

```
/**
 * <p>
 *   A number of RESTful services implement GET operations on a particular type of entity.  ↩
    For
 *   observing the DRY principle, the generic operations are implemented in the <code> ↩
    BaseEntityService</code>
 *   class, and the other services can inherit from here.
 * </p>
 *
 * <p>
```

```
 *    Subclasses will declare a base path using the JAX-RS {@link Path} annotation, for example:
 * </p>
 *
 * <pre>
 * <code>
 * &#064;Path("/widgets")
 * public class WidgetService extends BaseEntityService<Widget> {
 * ...
 * }
 * </code>
 * </pre>
 *
 * <p>
 *   will support the following methods:
 * </p>
 *
 * <pre>
 * <code>
 *   GET /widgets
 *   GET /widgets/:id
 *   GET /widgets/count
 * </code>
 * </pre>
 *
 *  <p>
 *     Subclasses may specify various criteria for filtering entities when retrieving a list of them, by supporting
 *     custom query parameters. Pagination is supported by default through the query parameters <code>first</code>
 *     and <code>maxResults</code>.
 * </p>
 *
 * <p>
 *     The class is abstract because it is not intended to be used directly, but subclassed by actual JAX-RS
 *     endpoints.
 * </p>
 *
 */
public abstract class BaseEntityService<T> {

    @Inject
    private EntityManager entityManager;

    private Class<T> entityClass;

    public BaseEntityService() {}

    public BaseEntityService(Class<T> entityClass) {
        this.entityClass = entityClass;
    }

    public EntityManager getEntityManager() {
        return entityManager;
    }

}
```

Now we add a method to retrieve all entities of a given type:

src/main/java/org/jboss/jdf/example/ticketmonster/rest/BaseEntityService.java

```
public abstract class BaseEntityService<T> {

    ...

    /**
     * <p>
     *   A method for retrieving all entities of a given type. Supports the query parameters
     *  <code>first</code>
     *   and <code>maxResults</code> for pagination.
     * </p>
     *
     *  @param uriInfo application and request context information (see {@see UriInfo} class
     *  information for more details)
     *  @return
     */
    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<T> getAll(@Context UriInfo uriInfo) {
        return getAll(uriInfo.getQueryParameters());
    }

    public List<T> getAll(MultivaluedMap<String, String> queryParameters) {
        final CriteriaBuilder criteriaBuilder = entityManager.getCriteriaBuilder();
        final CriteriaQuery<T> criteriaQuery = criteriaBuilder.createQuery(entityClass);
        Root<T> root = criteriaQuery.from(entityClass);
        Predicate[] predicates = extractPredicates(queryParameters, criteriaBuilder, root);
        criteriaQuery.select(criteriaQuery.getSelection()).where(predicates);
        criteriaQuery.orderBy(criteriaBuilder.asc(root.get("id")));
        TypedQuery<T> query = entityManager.createQuery(criteriaQuery);
        if (queryParameters.containsKey("first")) {
                Integer firstRecord = Integer.parseInt(queryParameters.getFirst("first")) -1;
                query.setFirstResult(firstRecord);
        }
        if (queryParameters.containsKey("maxResults")) {
                Integer maxResults = Integer.parseInt(queryParameters.getFirst("maxResults"));
                query.setMaxResults(maxResults);
        }
                return query.getResultList();
    }

    /**
     * <p>
     *     Subclasses may choose to expand the set of supported query parameters (for adding more filtering
     *     criteria) by overriding this method.
     * </p>
     * @param queryParameters - the HTTP query parameters received by the endpoint
     * @param criteriaBuilder - @{link CriteriaBuilder} used by the invoker
     * @param root  @{link Root} used by the invoker
     * @return a list of {@link Predicate}s that will added as query parameters
     */
    protected Predicate[] extractPredicates(MultivaluedMap<String, String> queryParameters,
                                            CriteriaBuilder criteriaBuilder, Root<T> root) {
        return new Predicate[]{};
    }
```

```
}
```

The newly added method ‘getAll` is annotated with `@GET` which instructs JAX-RS to call it when a `GET` HTTP requests on the JAX-RS’ endpoint base URL */rest/<entityRoot>* is performed. But remember, this is not a true JAX-RS endpoint. It is an abstract class and it is not mapped to a path. The classes that extend it are JAX-RS endpoints, and will have to be mapped to a path, and are able to process requests.

The `@Produces` annotation defines that the response sent back by the server is in JSON format. The JAX-RS implementation will automatically convert the result returned by the method (a list of entities) into JSON format.

As well as configuring the marshaling strategy, the annotation affects content negotiation and method resolution. If the client requests JSON content specifically, this method will be invoked.

Note

Even though it is not shown in this example, you may have multiple methods that handle a specific URL and HTTP method, whilst consuming and producing different types of content (JSON, HTML, XML or others).

Subclasses can also override the `extractPredicates` method and add own support for additional query parameters to `GET /rest/<entityRoot>` which can act as filter criteria.

The `getAll` method supports retrieving a range of entities, which is especially useful when we need to handle very large sets of data, and use pagination. In those cases, we need to support counting entities as well, so we add a method that retrieves the entity count:

src/main/java/org/jboss/jdf/example/ticketmonster/rest/BaseEntityService.java

```
public abstract class BaseEntityService<T> {

    ...

    /**
     * <p>
     *   A method for counting all entities of a given type
     * </p>
     *
     * @param uriInfo application and request context information (see {@see UriInfo} class ↩
          information for more details)
     * @return
     */
    @GET
    @Path("/count")
    @Produces(MediaType.APPLICATION_JSON)
    public Map<String, Long> getCount(@Context UriInfo uriInfo) {
        CriteriaBuilder criteriaBuilder = entityManager.getCriteriaBuilder();
        CriteriaQuery<Long> criteriaQuery = criteriaBuilder.createQuery(Long.class);
        Root<T> root = criteriaQuery.from(entityClass);
        criteriaQuery.select(criteriaBuilder.count(root));
        Predicate[] predicates = extractPredicates(uriInfo.getQueryParameters(),  ↩
            criteriaBuilder, root);
        criteriaQuery.where(predicates);
        Map<String, Long> result = new HashMap<String, Long>();
        result.put("count", entityManager.createQuery(criteriaQuery).getSingleResult());
        return result;
    }

}
```

We use the `@Path` annotation to map the new method to a sub-path of */rest/<entityRoot>*. *Now all the JAX-RS endpoints that subclass `BaseEntityService` will be able to get entity counts from ’/rest/<entityRoot>/count*. Just like `getAll`, this method also delegates to `extractPredicates`, so any customizations done there by subclasses

Next, we add a method for retrieving individual entities.

src/main/java/org/jboss/jdf/example/ticketmonster/rest/BaseEntityService.java

```
   ...
public abstract class BaseEntityService<T> {

    ...

    /**
     * <p>
     *     A method for retrieving individual entity instances.
     * </p>
     * @param id entity id
     * @return
     */
    @GET
    @Path("/{id:[0-9][0-9]*}")
    @Produces(MediaType.APPLICATION_JSON)
    public T getSingleInstance(@PathParam("id") Long id) {
        final CriteriaBuilder criteriaBuilder = entityManager.getCriteriaBuilder();
        final CriteriaQuery<T> criteriaQuery = criteriaBuilder.createQuery(entityClass);
        Root<T> root = criteriaQuery.from(entityClass);
        Predicate condition = criteriaBuilder.equal(root.get("id"), id);
        criteriaQuery.select(criteriaBuilder.createQuery(entityClass).getSelection()).where ↩
            (condition);
        return entityManager.createQuery(criteriaQuery).getSingleResult();
    }
}
```

This method is similar to `getAll` and `getCount`, and we use the `@Path` annotation to map it to a sub-path of */rest/<entityRoot>*. The annotation attribute identifies the expected format of the URL (here, the last segment has to be a number) and binds a portion of the URL to a variable (here named `id`). The `@PathParam` annotation allows the value of the variable to be passed as amethod argument. Data conversion is performed automatically.

Now, all the JAX-RS endpoints that subclass `BaseEntityService` will get two operations for free:

`GET /rest/<entityRoot>`
retrieves all entities of a given type

`GET /rest/<entityRoot>/<id>`
retrieves an entity with a given id

22.3 Retrieving Venues

Adding support for retrieving venues is now extremely simple. We refactor the class we created during the introduction, and make it extend `BaseEntityService`, passing the entity type to the superclass constructor. We remove the old retrieval code, which is not needed anymore.

src/main/java/org/jboss/jdf/example/ticketmonster/rest/VenueService.java

```
/**
 * <p>
 *     A JAX-RS endpoint for handling {@link Venue}s. Inherits the actual
 *     methods from {@link BaseEntityService}.
 * </p>
 */
@Path("/venues")
/**
 * <p>
 *     This is a stateless service, so a single shared instance can be used in this case.
```

```
 * </p>
 */
@Stateless
public class VenueService extends BaseEntityService<Venue> {

    public VenueService() {
        super(Venue.class);
    }

}
```

We add the `@Path` annotation to the class, to indicate that this is a JAX-RS resource which can serve URLs starting with `/rest/venues`.

We define this service (along with all the other JAX-RS services) as an EJB (see how simple is that in Java EE 6!) to benefit from automatic transaction enrollment. Since the service is fundamentally stateless, we take advantage of the new EJB 3.1 singleton feature.

Now, we can retrieve venues from URLs like `/rest/venues` or `rest/venues/1`.

22.4 Retrieving Events

Just like `VenueService`, the `EventService` implementation we use for TicketMonster is a direct subclass of `BaseEntityServi` Refactor the existing class, remove the old retrieval code and make it extend `BaseEntityService`.

One additional functionality we will implement is querying events by category. We can use URLs like `/rest/events?category=1` to retrieve all concerts, for example (`1` is the category id of concerts). This is done by overriding the `extractPredicates` method to handle any query parameters (in this case, the `category` parameter).

src/main/java/org/jboss/jdf/example/ticketmonster/rest/EventService.java

```
/**
 * <p>
 *     A JAX-RS endpoint for handling {@link Event}s. Inherits the actual
 *     methods from {@link BaseEntityService}, but implements additional search
 *     criteria.
 * </p>
 */
@Path("/events")
/**
 * <p>
 *     This is a stateless service, we declare it as an EJB for transaction demarcation
 * </p>
 */
@Stateless
public class EventService extends BaseEntityService<Event> {

    public EventService() {
        super(Event.class);
    }

    /**
     * <p>
     *    We override the method from parent in order to add support for additional search
     *    criteria for events.
     * </p>
     * @param queryParameters - the HTTP query parameters received by the endpoint
     * @param criteriaBuilder - @{link CriteriaBuilder} used by the invoker
     * @param root  @{link Root} used by the invoker
     * @return
     */
```

```
    @Override
    protected Predicate[] extractPredicates(
            MultivaluedMap<String, String> queryParameters,
            CriteriaBuilder criteriaBuilder,
            Root<Event> root) {
        List<Predicate> predicates = new ArrayList<Predicate>() ;

        if (queryParameters.containsKey("category")) {
            String category = queryParameters.getFirst("category");
            predicates.add(criteriaBuilder.equal(root.get("category").get("id"), category)) ↩
                ;
        }

        return predicates.toArray(new Predicate[]{});
    }
}
```

The `ShowService` and `BookingService` follow the same pattern and we leave the implementation as an exercise to the reader (knowing that its contents can always be copied over to the appropriate folder).

Of course, we also want to change data with our services - we want to create and delete bookings as well!

22.5 Creating and deleting bookings

To create a booking, we add a new method, which handles `POST` requests to `/rest/bookings`. This is not a simple CRUD method, as the client does not send a booking, but a booking request. It is the responsibility of the service to process the request, reserve the seats and return the full booking details to the invoker.

src/main/java/org/jboss/jdf/example/ticketmonster/rest/BookingService.java

```
/**
 * <p>
 *     A JAX-RS endpoint for handling {@link Booking}s. Inherits the GET
 *     methods from {@link BaseEntityService}, and implements additional REST methods.
 * </p>
 */
@Path("/bookings")
/**
 * <p>
 *     This is a stateless service, we declare it as an EJB for transaction demarcation
 * </p>
 */
@Stateless
public class BookingService extends BaseEntityService<Booking> {

    @Inject
    SeatAllocationService seatAllocationService;

    @Inject @Created
    private Event<Booking> newBookingEvent;

    public BookingService() {
        super(Booking.class);
    }

   /**
     * <p>
     *   Create a booking. Data is contained in the bookingRequest object
     * </p>
     * @param bookingRequest
```

```
     * @return
     */
    @SuppressWarnings("unchecked")
    @POST
    /**
     * <p> Data is received in JSON format. For easy handling, it will be unmarshalled in ↩
         the support
     * {@link BookingRequest} class.
     */
    @Consumes(MediaType.APPLICATION_JSON)
    public Response createBooking(BookingRequest bookingRequest) {
        try {
            // identify the ticket price categories in this request
            Set<Long> priceCategoryIds = bookingRequest.getUniquePriceCategoryIds();

            // load the entities that make up this booking's relationships
            Performance performance = getEntityManager().find(Performance.class, ↩
                bookingRequest.getPerformance());

            // As we can have a mix of ticket types in a booking, we need to load all of ↩
                them that are relevant,
            // id
            Map<Long, TicketPrice> ticketPricesById = loadTicketPrices(priceCategoryIds);

            // Now, start to create the booking from the posted data
            // Set the simple stuff first!
            Booking booking = new Booking();
            booking.setContactEmail(bookingRequest.getEmail());
            booking.setPerformance(performance);
            booking.setCancellationCode("abc");

            // Now, we iterate over each ticket that was requested, and organize them by ↩
                section and category
            // we want to allocate ticket requests that belong to the same section ↩
                contiguously
            Map<Section, Map<TicketCategory, TicketRequest>> ticketRequestsPerSection
                    = new TreeMap<Section, java.util.Map<TicketCategory, TicketRequest>>( ↩
                        SectionComparator.instance());
            for (TicketRequest ticketRequest : bookingRequest.getTicketRequests()) {
                final TicketPrice ticketPrice = ticketPricesById.get(ticketRequest. ↩
                    getTicketPrice());
                if (!ticketRequestsPerSection.containsKey(ticketPrice.getSection())) {
                    ticketRequestsPerSection
                            .put(ticketPrice.getSection(), new HashMap<TicketCategory, ↩
                                TicketRequest>());
                }
                ticketRequestsPerSection.get(ticketPrice.getSection()).put(
                        ticketPricesById.get(ticketRequest.getTicketPrice()). ↩
                            getTicketCategory(), ticketRequest);
            }

            // Now, we can allocate the tickets
            // Iterate over the sections, finding the candidate seats for allocation
            // The process will acquire a write lock for a given section and performance
            // Use deterministic ordering of sections to prevent deadlocks
            Map<Section, AllocatedSeats> seatsPerSection =
                                new TreeMap<Section, org.jboss.jdf.example.ticketmonster. ↩
                                    service.AllocatedSeats>(SectionComparator.instance());
            List<Section> failedSections = new ArrayList<Section>();
            for (Section section : ticketRequestsPerSection.keySet()) {
                int totalTicketsRequestedPerSection = 0;
                // Compute the total number of tickets required (a ticket category doesn't ↩
```

```
                impact the actual seat!)
            final Map<TicketCategory, TicketRequest> ticketRequestsByCategories =
                ticketRequestsPerSection.get(section);
            // calculate the total quantity of tickets to be allocated in this section
            for (TicketRequest ticketRequest : ticketRequestsByCategories.values()) {
                totalTicketsRequestedPerSection += ticketRequest.getQuantity();
            }
            // try to allocate seats

            AllocatedSeats allocatedSeats =
                                seatAllocationService.allocateSeats(section,
                                    performance, totalTicketsRequestedPerSection, true
                                    );
            if (allocatedSeats.getSeats().size() == totalTicketsRequestedPerSection) {
                seatsPerSection.put(section, allocatedSeats);
            } else {
                failedSections.add(section);
            }
        }
        if (failedSections.isEmpty()) {
            for (Section section : seatsPerSection.keySet()) {
                // allocation was successful, begin generating tickets
                // associate each allocated seat with a ticket, assigning a price
                    category to it
                final Map<TicketCategory, TicketRequest> ticketRequestsByCategories =
                    ticketRequestsPerSection.get(section);
                AllocatedSeats allocatedSeats = seatsPerSection.get(section);
                allocatedSeats.markOccupied();
                int seatCounter = 0;
                // Now, add a ticket for each requested ticket to the booking
                for (TicketCategory ticketCategory : ticketRequestsByCategories.keySet
                    ()) {
                    final TicketRequest ticketRequest = ticketRequestsByCategories.get(
                        ticketCategory);
                    final TicketPrice ticketPrice = ticketPricesById.get(ticketRequest.
                        getTicketPrice());
                    for (int i = 0; i < ticketRequest.getQuantity(); i++) {
                        Ticket ticket =
                                                        new Ticket(allocatedSeats.
                                                            getSeats().get(seatCounter
                                                             + i), ticketCategory,
                                                            ticketPrice.getPrice());
                        // getEntityManager().persist(ticket);
                        booking.getTickets().add(ticket);
                    }
                    seatCounter += ticketRequest.getQuantity();
                }
            }
            // Persist the booking, including cascaded relationships
            booking.setPerformance(performance);
            booking.setCancellationCode("abc");
            getEntityManager().persist(booking);
            newBookingEvent.fire(booking);
            return Response.ok().entity(booking).type(MediaType.APPLICATION_JSON_TYPE).
                build();
        } else {
            Map<String, Object> responseEntity = new HashMap<String, Object>();
            responseEntity.put("errors", Collections.singletonList("Cannot allocate the
                 requested number of seats!"));
            return Response.status(Response.Status.BAD_REQUEST).entity(responseEntity).
                build();
        }
```

```
        } catch (ConstraintViolationException e) {
            // If validation of the data failed using Bean Validation, then send an error
            Map<String, Object> errors = new HashMap<String, Object>();
            List<String> errorMessages = new ArrayList<String>();
            for (ConstraintViolation<?> constraintViolation : e.getConstraintViolations())  ↩
                {
                errorMessages.add(constraintViolation.getMessage());
            }
            errors.put("errors", errorMessages);
            // A WebApplicationException can wrap a response
            // Throwing the exception causes an automatic rollback
            throw new WebApplicationException(Response.status(Response.Status.BAD_REQUEST). ↩
                entity(errors).build());
        } catch (Exception e) {
            // Finally, handle unexpected exceptions
            Map<String, Object> errors = new HashMap<String, Object>();
            errors.put("errors", Collections.singletonList(e.getMessage()));
            // A WebApplicationException can wrap a response
            // Throwing the exception causes an automatic rollback
            throw new WebApplicationException(Response.status(Response.Status.BAD_REQUEST). ↩
                entity(errors).build());
        }
    }

    /**
     * Utility method for loading ticket prices
     * @param priceCategoryIds
     * @return
     */
    private Map<Long, TicketPrice> loadTicketPrices(Set<Long> priceCategoryIds) {
        List<TicketPrice> ticketPrices = (List<TicketPrice>) getEntityManager()
                .createQuery("select p from TicketPrice p where p.id in :ids")
                .setParameter("ids", priceCategoryIds).getResultList();
        // Now, map them by id
        Map<Long, TicketPrice> ticketPricesById = new HashMap<Long, TicketPrice>();
        for (TicketPrice ticketPrice : ticketPrices) {
            ticketPricesById.put(ticketPrice.getId(), ticketPrice);
        }
        return ticketPricesById;
    }
}
```

We won't get into the details of the inner workings of the method - it implements a fairly complex algorithm - but we'd like to draw attention to a few particular items.

We use the `@POST` annotation to indicate that this method is executed on inbound HTTP POST requests. When implementing a set of RESTful services, it is important that the semantic of HTTP methods are observed in the mappings. Creating new resources (e.g. bookings) is typically associated with HTTP POST invocations. The `@Consumes` annotation indicates that the type of the request content is JSON and identifies the correct unmarshalling strategy, as well as content negotiation.

The `BookingService` delegates to the `SeatAllocationService` to find seats in the requested section, the required `SeatAllocationService` instance is initialized and supplied by the container as needed. The only thing that our service does is to specify the dependency in form of an injection point - the field annotated with `@Inject`.

We would like other parts of the application to be aware of the fact that a new booking has been created, therefore we use the CDI to fire an event. We do so by injecting an `Event<Booking>` instance into the service (indicating that its payload will be a booking). In order to individually identify this event as referring to event creation, we use a CDI qualifier, which we need to add:

src/main/java/org/jboss/jdf/example/ticketmonster/monitor/client/shared/qualifier/Created.java

```
/**
 * {@link Qualifier} to mark a Booking as new (created).
 */
```

```
@Qualifier
@Target({ElementType.FIELD,ElementType.PARAMETER,ElementType.METHOD,ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface Created {

}
```

What are qualifiers?
CDI uses a type-based resolution mechanism for injection and observers. In order to distinguish between implementations of an interface, you can use qualifiers, a type of annotations, to disambiguate. Injection points and event observers can use qualifiers to narrow down the set of candidates

We also need allow the removal of bookings, so we add a method:

src/main/java/org/jboss/jdf/example/ticketmonster/rest/BookingService.java

```
@Singleton
public class BookingService extends BaseEntityService<Booking> {
        ...

    @Inject @Cancelled
    private Event<Booking> cancelledBookingEvent;
    ...
    /**
     * <p>
     * Delete a booking by id
     * </p>
     * @param id
     * @return
     */
    @DELETE
    @Path("/{id:[0-9][0-9]*}")
    public Response deleteBooking(@PathParam("id") Long id) {
        Booking booking = getEntityManager().find(Booking.class, id);
        if (booking == null) {
            return Response.status(Response.Status.NOT_FOUND).build();
        }
        getEntityManager().remove(booking);
        cancelledBookingEvent.fire(booking);
        return Response.ok().build();
    }
}
```

We use the `@DELETE` annotation to indicate that it will be executed as the result of an HTTP DELETE request (again, the use of the DELETE HTTP verb is a matter of convention).

We need to notify the other components of the cancellation of the booking, so we fire an event, with a different qualifier.

src/main/java/org/jboss/jdf/example/ticketmonster/monitor/client/shared/qualifier/Cancelled.java

```
/**
 * {@link Qualifier} to mark a Booking as cancelled.
 */
@Qualifier
@Target({ElementType.FIELD,ElementType.PARAMETER,ElementType.METHOD,ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface Cancelled {

}
```

The other services, including the `MediaService` that handles media items follow roughly the same patterns as above, so we leave them as an exercise to the reader.

Chapter 23

Testing the services

We've now finished implementing the services and there is a significant amount of functionality in the application. Before taking any step forward, you need to make sure the services work correctly: we need to test them.

Testing enterprise services be a complex task as the implementation is based on services provided by a container: dependency injection, access to infrastructure services such as persistence, transactions etc.. Unit testing frameworks, whilst offering a valuable infrastructure for running tests, do not provide these capabilities.

One of the traditional approaches has been the use of mocking frameworks to simulate *what will happen* in the runtime environment. While certainly providing a solution mocking brings its own set of problems (e.g. the additional effort required to provide a proper simulation or the risk of introducing errors in the test suite by incorrectly implemented mocks.

Fortunately, Arquillian provides the means to testing your application code within the container, with access to all the services and container features. In this section we will show you how to create a few Arquillian tests for your business services.

What to test?
A common asked question is: how much application functionality should we test? The truth is, you can never test too much. That being said, resources are always limited and tradeoffs are part of an engineer's work. Generally speaking, trivial functionality (setters/getters/toString methods) is a big concern compared to the actual business code, so you probably want to focus your efforts on the business code. Testing should include individual parts (unit testing), as well as aggregates (integration testing).

23.1 A Basic Deployment Class

In order to create Arquillian tests, we need to define the deployment. The code under test, as well as its dependencies is packaged and deployed in the container.

Much of the deployment contents is common for all tests, so we create a helper class with a method that creates the base deployment with all the general content.

src/test/java/org/jboss/jdf/ticketmonster/test/TicketMonsterDeployment.java

```
public class TicketMonsterDeployment {

    public static WebArchive deployment() {
        return ShrinkWrap
                .create(WebArchive.class, "test.war")
                .addPackage(Resources.class.getPackage())
                .addAsResource("META-INF/test-persistence.xml", "META-INF/persistence.xml")
                .addAsResource("import.sql")
                .addAsWebInfResource(EmptyAsset.INSTANCE, "beans.xml")
                // Deploy our test datasource
                .addAsWebInfResource("test-ds.xml");
```

```
    }
}
```

Arquillian uses Shrinkwrap to define the contents of the deployment.

23.2 Writing RESTful service tests

For testing our JAX-RS RESTful services, we need to add the corresponding application classes to the deployment. Since we need to do that for each test we create, we abide by the DRY principles and create a utility class.

src/test/java/org/jboss/jdf/ticketmonster/test/rest/RESTDeployment.java

```
public class RESTDeployment {

    public static WebArchive deployment() {
        return TicketMonsterDeployment.deployment()
                .addPackage(Booking.class.getPackage())
                .addPackage(BaseEntityService.class.getPackage())
                .addPackage(MockMultivaluedMap.class.getPackage())
                .addClass(SeatAllocationService.class)
                .addClass(AllocatedSeats.class)
                .addClass(MediaPath.class)
                .addClass(MediaManager.class);
    }

}
```

Now, we create the first test to validate the proper retrieval of individual events.

src/test/java/org/jboss/jdf/ticketmonster/test/rest/VenueServiceTest.java

```
@RunWith(Arquillian.class)
public class VenueServiceTest {

    @Deployment
    public static WebArchive deployment() {
        return RESTDeployment.deployment();
    }

    @Inject
    private VenueService venueService;

    @Test
    public void testGetVenueById() {

        // Test loading a single venue
        Venue venue = venueService.getSingleInstance(1l);
        assertNotNull(venue);
        assertEquals("Roy Thomson Hall", venue.getName());
    }

}
```

In the class above we specify the deployment, and we define the test method. The test supports CDI injection - one of the strengths of Arquillian is the ability to inject the object being tested.

Now, we test a more complicated use cases, query parameters for pagination.

src/test/java/org/jboss/jdf/ticketmonster/test/rest/VenueServiceTest.java

```
...
@RunWith(Arquillian.class)
public class VenueServiceTest {

    ...

    @Test
    public void testPagination() {

        // Test pagination logic
        MultivaluedMap<String, String> queryParameters = new MultivaluedHashMap<String,  ↩
            String>();

        queryParameters.add("first", "2");
        queryParameters.add("maxResults", "1");

        List<Venue> venues = venueService.getAll(queryParameters);
        assertNotNull(venues);
        assertEquals(1, venues.size());
        assertEquals("Sydney Opera House", venues.get(0).getName());
    }

}
```

We add another test method (`testPagination`), which tests the retrieval of all venues, passing the search criteria as parameters. We use a Map to simulate the passing of query parameters.

Now, we test more advanced use cases such as the creation of a new booking. We do so by adding a new test for bookings

src/test/java/org/jboss/jdf/ticketmonster/test/rest/BookingServiceTest.java

```
@RunWith(Arquillian.class)
public class BookingServiceTest {

    @Deployment
    public static WebArchive deployment() {
        return RESTDeployment.deployment();
    }

    @Inject
    private BookingService bookingService;

    @Inject
    private ShowService showService;

    @Test
    @InSequence(1)
    public void testCreateBookings() {
        BookingRequest br = createBookingRequest(1l, 0, 0, 1, 3);
        bookingService.createBooking(br);

        BookingRequest br2 = createBookingRequest(2l, 1, 2, 4, 9);
        bookingService.createBooking(br2);

        BookingRequest br3 = createBookingRequest(3l, 0, 0, 1);
        bookingService.createBooking(br3);
    }

    @Test
    @InSequence(10)
    public void testGetBookings() {
        checkBooking1();
```

```
        checkBooking2();
        checkBooking3();
    }

    private void checkBooking1() {
        Booking booking = bookingService.getSingleInstance(1l);
        assertNotNull(booking);
        assertEquals("Roy Thomson Hall", booking.getPerformance().getShow().getVenue(). ↩
            getName());
        assertEquals("Rock concert of the decade", booking.getPerformance().getShow(). ↩
            getEvent().getName());
        assertEquals("bob@acme.com", booking.getContactEmail());

        // Test the ticket requests created

        assertEquals(3 + 2 + 1, booking.getTickets().size());

        List<String> requiredTickets = new ArrayList<String>();
        requiredTickets.add("A @ 219.5 (Adult)");
        requiredTickets.add("A @ 219.5 (Adult)");
        requiredTickets.add("D @ 149.5 (Adult)");
        requiredTickets.add("C @ 179.5 (Adult)");
        requiredTickets.add("C @ 179.5 (Adult)");
        requiredTickets.add("C @ 179.5 (Adult)");

        checkTickets(requiredTickets, booking);
    }

    private void checkBooking2() {
        Booking booking = bookingService.getSingleInstance(2l);
        assertNotNull(booking);
        assertEquals("Sydney Opera House", booking.getPerformance().getShow().getVenue(). ↩
            getName());
        assertEquals("Rock concert of the decade", booking.getPerformance().getShow(). ↩
            getEvent().getName());
        assertEquals("bob@acme.com", booking.getContactEmail());

        assertEquals(3 + 2 + 1, booking.getTickets().size());

        List<String> requiredTickets = new ArrayList<String>();
        requiredTickets.add("S2 @ 197.75 (Adult)");
        requiredTickets.add("S6 @ 145.0 (Child 0-14yrs)");
        requiredTickets.add("S6 @ 145.0 (Child 0-14yrs)");
        requiredTickets.add("S4 @ 145.0 (Child 0-14yrs)");
        requiredTickets.add("S6 @ 145.0 (Child 0-14yrs)");
        requiredTickets.add("S4 @ 145.0 (Child 0-14yrs)");

        checkTickets(requiredTickets, booking);
    }

    private void checkBooking3() {
        Booking booking = bookingService.getSingleInstance(3l);
        assertNotNull(booking);
        assertEquals("Roy Thomson Hall", booking.getPerformance().getShow().getVenue(). ↩
            getName());
        assertEquals("Shane's Sock Puppets", booking.getPerformance().getShow().getEvent(). ↩
            getName());
        assertEquals("bob@acme.com", booking.getContactEmail());

        assertEquals(2 + 1, booking.getTickets().size());

        List<String> requiredTickets = new ArrayList<String>();
```

```
        requiredTickets.add("B @ 199.5 (Adult)");
        requiredTickets.add("D @ 149.5 (Adult)");
        requiredTickets.add("B @ 199.5 (Adult)");

        checkTickets(requiredTickets, booking);
    }

    @Test
    @InSequence(10)
    public void testPagination() {

        // Test pagination logic
        MultivaluedMap<String, String> queryParameters = new MultivaluedHashMap<java.lang. ↩
            String, java.lang.String>();

        queryParameters.add("first", "2");
        queryParameters.add("maxResults", "1");

        List<Booking> bookings = bookingService.getAll(queryParameters);
        assertNotNull(bookings);
        assertEquals(1, bookings.size());
        assertEquals("Sydney Opera House", bookings.get(0).getPerformance().getShow(). ↩
            getVenue().getName());
        assertEquals("Rock concert of the decade", bookings.get(0).getPerformance().getShow ↩
            ().getEvent().getName());
    }

    @Test
    @InSequence(20)
    public void testDelete() {
        bookingService.deleteBooking(2l);
        checkBooking1();
        checkBooking3();
        try {
            bookingService.getSingleInstance(2l);
        } catch (Exception e) {
            if (e.getCause() instanceof NoResultException) {
                return;
            }
        }
        fail("Expected NoResultException did not occur.");
    }

    private BookingRequest createBookingRequest(Long showId, int performanceNo, int...  ↩
        ticketPriceNos) {
        Show show = showService.getSingleInstance(showId);

        Performance performance = new ArrayList<Performance>(show.getPerformances()).get( ↩
            performanceNo);

        BookingRequest bookingRequest = new BookingRequest(performance, "bob@acme.com");

        List<TicketPrice> possibleTicketPrices = new ArrayList<TicketPrice>(show. ↩
            getTicketPrices());
        int i = 1;
        for (int index : ticketPriceNos) {
            bookingRequest.addTicketRequest(new TicketRequest(possibleTicketPrices.get( ↩
                index), i));
            i++;
        }

        return bookingRequest;
```

```
    }

    private void checkTickets(List<String> requiredTickets, Booking booking) {
        List<String> bookedTickets = new ArrayList<String>();
        for (Ticket t : booking.getTickets()) {
            bookedTickets.add(new StringBuilder().append(t.getSeat().getSection()).append(" ↩
                 @ ").append(t.getPrice()).append(" (").append(t.getTicketCategory()).append ↩
                (")").toString());
        }
        System.out.println(bookedTickets);
        for (String requiredTicket : requiredTickets) {
            Assert.assertTrue("Required ticket not present: " + requiredTicket,  ↩
                bookedTickets.contains(requiredTicket));
        }
    }

}
```

First we test booking creation in a test method of its own (`testCreateBookings`). Then, we test that the previously created bookings are retrieved correctly (`testGetBookings` and `testPagination`). Finally, we test that deletion takes place correctly (`testDelete`).

The other tests in the application follow roughly the same pattern and are left as an exercise to the reader.

23.3 Running the tests

If you have followed the instructions in the introduction and used the Maven archetype to generate the project structure, you should have two profiles already defined in your application.

/pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/ ↩
    XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/ ↩
             maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>

        ...
        <profile>
            <!-- An optional Arquillian testing profile that executes tests
                 in your JBoss AS instance -->
            <!-- This profile will start a new JBoss AS instance, and execute
                 the test, shutting it down when done -->
            <!-- Run with: mvn clean test -Parq-jbossas-managed -->
            <id>arq-jbossas-managed</id>
            <dependencies>
                <dependency>
                    <groupId>org.jboss.as</groupId>
                    <artifactId>jboss-as-arquillian-container-managed</artifactId>
                    <scope>test</scope>
                </dependency>
            </dependencies>
        </profile>

        <profile>
            <!-- An optional Arquillian testing profile that executes tests
                 in a remote JBoss AS instance -->
            <!-- Run with: mvn clean test -Parq-jbossas-remote -->
            <id>arq-jbossas-remote</id>
```

```
            <dependencies>
                <dependency>
                    <groupId>org.jboss.as</groupId>
                    <artifactId>jboss-as-arquillian-container-remote</artifactId>
                    <scope>test</scope>
                </dependency>
            </dependencies>
        </profile>

    </profiles>
</project>
```

If you haven't used the archetype, or the profiles don't exist, create them.

Each profile defines a different Arquillian container. In both cases the tests execute in an application server instance. In one case (`arq-jbossas-managed`) the server instance is started and stopped by the test suite, whils in the other (`arq-jbossas-remote`), the test suite expects an already started server instance.

Once these profiles are defined, we can execute the tests in two ways:

- from the command-line build
- from an IDE

23.3.1 Executing tests from the command line

You can now execute the test suite from the command line by running the Maven build with the appropriate target and profile, as in one of the following examples.

After ensuring that the `JBOSS_HOME` environment variable is set to a valid JBoss AS7 installation directory), you can run the following command:

```
mvn clean test -Parq-jbossas-managed
```

Or, after starting a JBoss AS7 instance, you can run the following command

```
mvn clean test -Parq-jbossas-remote
```

These tests execute as part of the Maven build and can be easily included in an automated build and test harness.

23.3.2 Running Arquillian tests from within Eclipse

Running the entire test suite as part of the build is an important part of the development process - you may want to make sure that everything is working fine before releasing a new milestone, or just before committing new code. However, running the entire test suite all the time can be a productivity drain, especially when you're trying to focus on a particular problem. Also, when debugging, you don't want to leave the comfort of your IDE for running the tests.

Running Arquillian tests from JBoss Developer Studio or JBoss tools is very simple as Arquillian builds on JUnit (or TestNG).

First enable one of the two profiles in the project. In Eclipse, open the project properties, and from the *Maven* tab, add the profile as shown in the picture below.

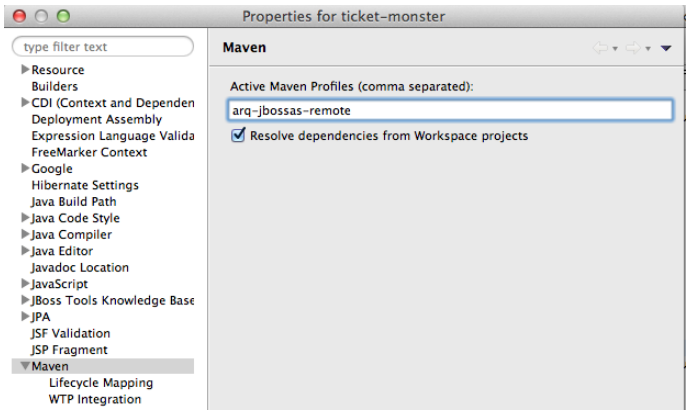

Figure 23.1: Update Maven profiles in Eclipse

The project configuration will be updated automatically.

Now, you can click right on one of your test classes, and select **Run As** → **JUnit Test**.

The test suite will run, deploying the test classes to the application server, executing the tests and finally producing the much coveted green bar.

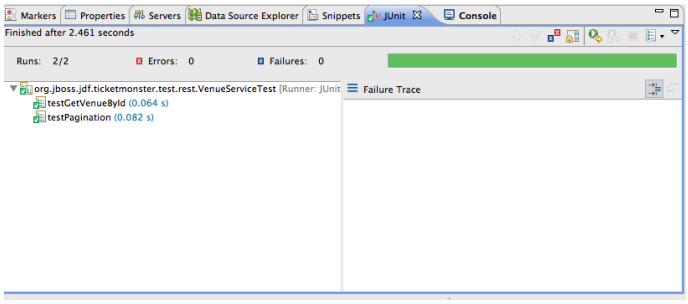

Figure 23.2: Running the tests

Part IV

Building The User UI Using HTML5

Chapter 24

What Will You Learn Here?

We've just implemented the business services of our application, and exposed them through RESTful endpoints. Now we need to implement a flexible user interface that can be easily used with both desktop and mobile clients. After reading this tutorial, you will understand our front-end design and the choices that we made in its implementation. Topics covered include:

- Creating single-page applications using HTML5, JavaScript and JSON
- Using JavaScript frameworks for invoking RESTful endpoints and manipulating page content
- Feature and device detection
- Implementing a version of the user interface that is optimized for mobile clients using JavaScript frameworks such as jQuery mobile

The tutorial will show you how to perform all these steps in JBoss Developer Studio, including screenshots that guide you through.

Chapter 25

First, the basics

In this tutorial, we will build a single-page application. All the necessary code: HTML, CSS and JavaScript is retrieved within a single page load. Rather than refreshing the page every time the user changes a view, the content of the page will be redrawn by manipulating the DOM in JavaScript. The application uses REST calls to retrieve data from the server.

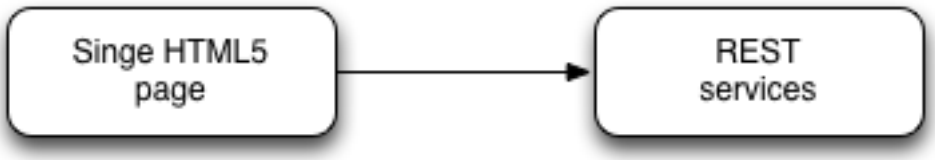

Figure 25.1: Single page application

25.1 Client-side MVC Support

Because this is a moderately complex example, which involves multiple views and different types of data, we will use a client-side MVC framework to structure the application, which provides amongst others:

- routing support within the single page application;
- event-driven interaction between views and data;
- simplified CRUD invocations on RESTful services.

In this application we use the client-side MVC framework "backbone.js".

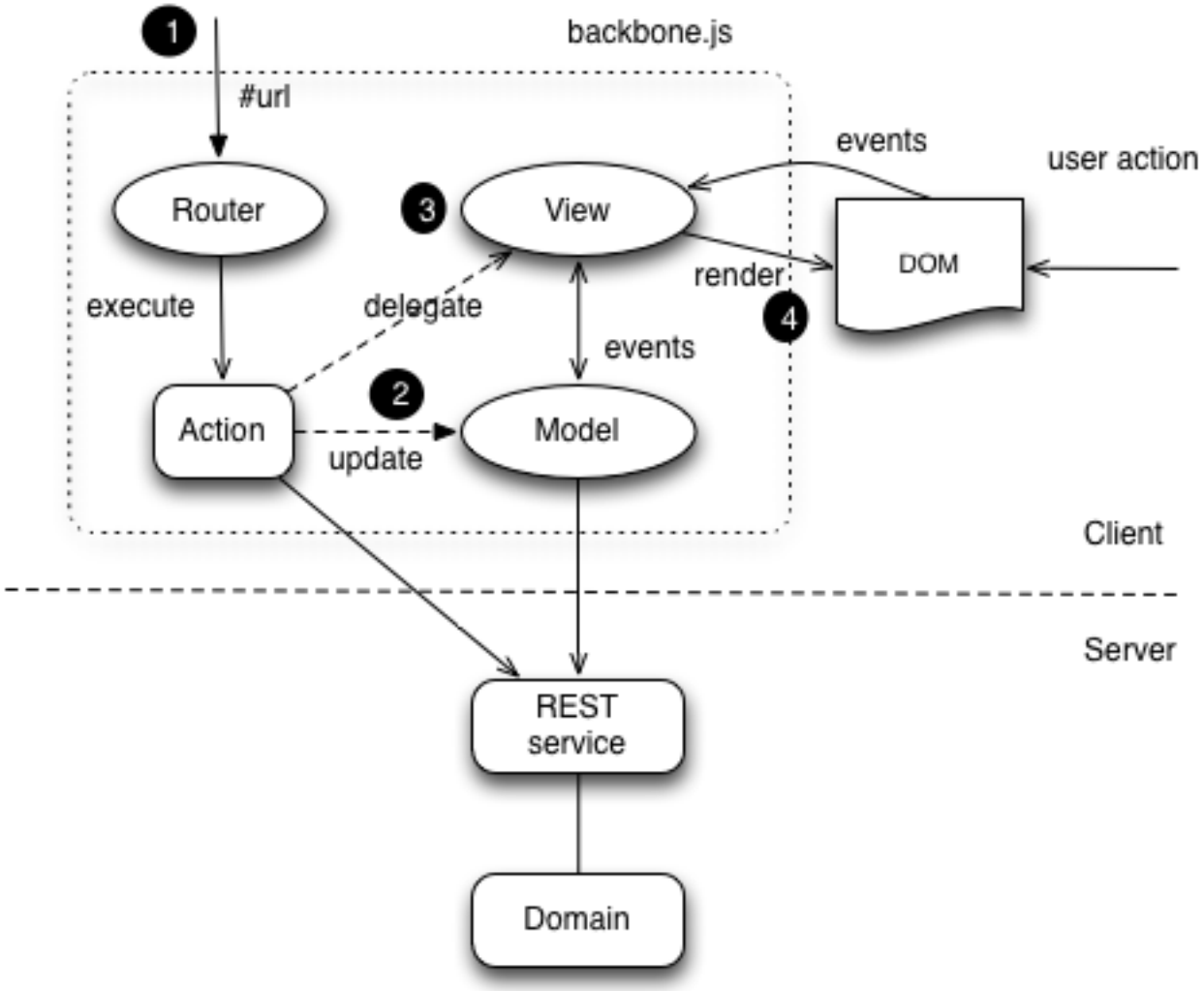

Figure 25.2: Backbone architecture

25.2 Modularity

In order to provide good separation of concerns, we split the JavaScript code into modules. Ensuring that all the modules of the application are loaded properly at runtime becomes a complex task, as the application size increases. To conquer this complexity, we use the Asynchronous Module Definition mechanism as implemented by the "require.js" library.

Asynchronous Module Definition
The Asynchronous Module Definition (AMD) API specifies a mechanism for defining modules such that the module, and its dependencies, can be asynchronously loaded. This is particularly well suited for the browser where synchronous loading of modules incurs performance, usability, debugging, and cross-domain access problems.

25.3 Templating

Instead of manipulating the DOM directly, and mixing up HTML with the JavaScript code, we create HTML markup fragments separately as templates which are applied when the application views are rendered.

In this application we use the templating support provided by "underscore.js".

25.4 Mobile and desktop versions

The page flow and structure, as well as feature set, are slightly different for mobile and desktop, and therefore we will build two variants of the single-page-application, one for desktop and one for mobile. As the application variants are very similar, we will cover the desktop version of the application first, and then we will explain what is different in the mobile version.

Chapter 26

Setting up the structure

Before we start developing the user interface, we need to set up the general application structure and add the JavaScript libraries. First, we create the directory structure:

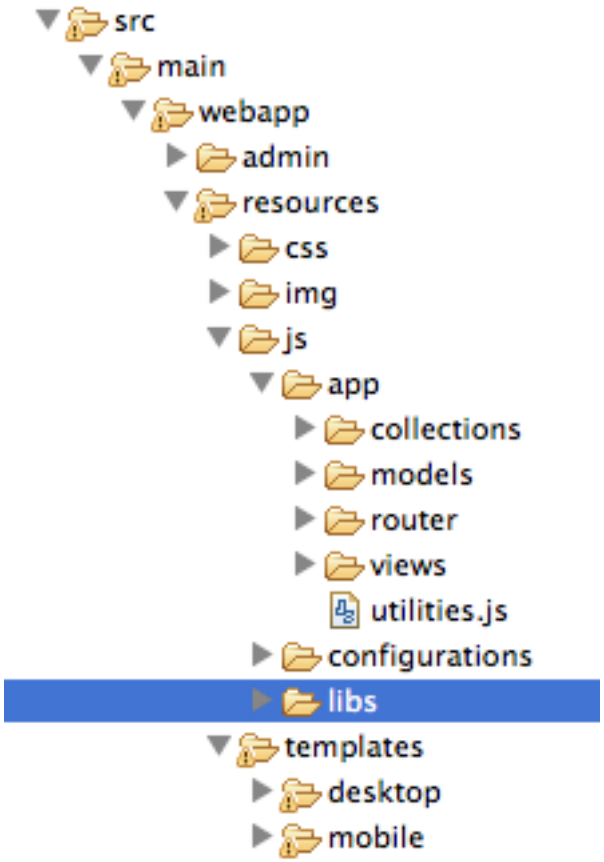

Figure 26.1: File structure for our web application

We put stylesheets in `resources/css` folder, images in `resources/img`, and HTML view templates in `resources/template` `resources/js` contains the JavaScript code, split between `resources/js/lib` - which contains the libraries used by the application, `resources/js/app` - which contains the application code, and `resources/configurations` which contains module definitions for the different versions of the application - i.e. mobile and desktop. The `resources/js/app` folder will contain the application modules, in subsequent subdirectories, for models, collections, routers and views.

The first step in implementing our solution is adding the stylesheets and JavaScript libraries to the `resources/css` and `resources/js/lib`:

require.js
: AMD support, along with the plugins:

- text - for loading text files, in our case the HTML templates
- order - for enforcing load ordering if necessary

jQuery
: general purpose library for HTML traversal and manipulation

Underscore
: JavaScript utility library (and a dependency of Backbone)

Backbone
: Client-side MVC framework

Bootstrap
: UI components and stylesheets for page structuring

Now, we create the main page of the application (which is the URL loaded by the browser):

src/main/webapp/index.html

```
<!DOCTYPE html>
<html>
<head>
    <title>Ticket Monster</title>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
    <meta name="viewport" content="width=device-width, initial-scale=1, user-scalable=0"/>

    <script type="text/javascript" src="resources/js/libs/modernizr-2.0.6.js"></script>

    <!--
         A simple check on the client. For touch devices or small-resolution screens)
         show the mobile client. By enabling the mobile client on a small-resolution screen
         we allow for testing outside a mobile device (like for example the Mobile Browser
         simulator in JBoss Tools and JBoss Developer Studio).
      -->
    <script type="text/javascript">
        var loader = document.createElement("script")
        loader.setAttribute("src", "resources/js/libs/require.js");
        if (Modernizr.touch || Modernizr.mq("only all and (max-width: 480px)")) {
           environment = "mobile"
        } else {
           environment = "desktop"
        };
        loader.setAttribute("data-main","resources/js/configurations/" + environment);
        document.head.appendChild(loader)
    </script>
</head>
<body>
</body>
</html>
```

As you can see, the page does not contain much. It detects the current client (mobile or destktop) based on its capabilities (touch or not) and loads the JavaScript code of the page from an AMD module defined in the `resources/js/configurations` folder using `require.js`. In the case of the desktop client, the code is loaded from `resources/js/configurations/deskto`

src/main/webapp/resources/js/configurations/desktop.js

```
/**
 * Shortcut alias definitions - will come in handy when declaring dependencies
 * Also, they allow you to keep the code free of any knowledge about library
 * locations and versions
 */
requirejs.config({
    baseUrl: "resources/js",
    paths: {
        jquery:'libs/jquery-1.7.1',
        underscore:'libs/underscore',
        text:'libs/text',
        order:'libs/order',
        bootstrap: 'libs/bootstrap',
        backbone: 'libs/backbone',
        utilities: 'app/utilities'
    },
    // We shim Backbone since it doesn't declare an AMD module
    shim: {
        'backbone': {
            deps: ['jquery', 'underscore'],
            exports: 'Backbone'
        }
    }
});

define("initializer", ["jquery"],
    function ($) {
    $('head').append('<link type="text/css" rel="stylesheet" href="resources/css/screen.css ↩
        "/>');
    $('head').append('<link rel="stylesheet" href="resources/css/bootstrap.css" type="text/ ↩
        css" media="all"/>');
    $('head').append('<link rel="stylesheet" href="resources/css/custom.css" type="text/css ↩
        " media="all">');
    $('head').append('<link href="http://fonts.googleapis.com/css?family=Rokkitt" rel=" ↩
        stylesheet" type="text/css">');
});

// Now we declare all the dependencies
require([
    'order!initializer',
    'order!underscore',
    'order!backbone'
], function(){
});

define("configuration", {
    baseUrl : ""
});
```

The module loads all the utility libraries, converting them to AMD modules where necessary (like it is the case for Backbone). It also defines two modules of its own - an initializer that loads the application stylesheets for the page, and the `configuration` module that allows customizing the REST service URLs (this will become in handy in a further tutorial).

Before we add any functionality, let us create a first landing page. We will begin by setting up a critical piece of the application, the router.

26.1 Routing

The outer allows for navigation in our application via bookmarkable URLs, and we will define it as follows:

src/main/webapp/resources/js/app/router/desktop/router.js

```
/**
 * A module for the router of the desktop application
 */
define("router", [
    'jquery',
    'underscore',
    'configuration',
    'utilities',
    'text!../templates/desktop/main.html'
],function ($,
            _,
            config,
            utilities,
            MainTemplate) {

    $(document).ready(new function() {
       utilities.applyTemplate($('body'), MainTemplate)
    })

    /**
     * The Router class contains all the routes within the application -
     * i.e. URLs and the actions that will be taken as a result.
     *
     * @type {Router}
     */

    var Router = Backbone.Router.extend({
        routes:{
        }
    });

    // Create a router instance
    var router = new Router();

    //Begin routing
    Backbone.history.start();

    return router;
});
```

Remember, this is a single page application. You can either navigate using urls such as `http://localhost:8080/ticket-monst` or using relative urls (from within the application, this being exactly what the main menu does). The fragment after the hash sign represents the url within the single page, on which the router will act, according to the mappings set up in the `routes` property.

The main module needs to load it. Because the router depends on all the other components (models, collections and views) of the application, directly or indirectly, it is the only component that is explicitly loaded in the `main-desktop` definition, which we change as follows:

src/main/webapp/resources/js/main-desktop.js

```
requirejs.config({
    baseUrl: "resources/js",
    paths: {
        jquery:'libs/jquery-1.7.1',
        underscore:'libs/underscore',
        text:'libs/text',
        order:'libs/order',
        bootstrap: 'libs/bootstrap',
        backbone: 'libs/backbone',
        utilities: 'app/utilities',
        router:'app/router/desktop/router'
```

```
    },
    // We shim Backbone since it doesn't declare an AMD module
    shim: {
        'backbone': {
            deps: ['jquery', 'underscore'],
            exports: 'Backbone'
        }
    }
});
  ...

require([
    'order!initializer',
    'order!underscore',
    'order!backbone',
    'order!router'
], function(){
});
```

During the router set up, we load the page template for the entire application. TicketMonster uses a templating library in order to separate application logic from it's actual graphical content. The actual HTML is described in template files, which are applied by the application, when necessary, on a DOM element - effectively populating it's content. So the general content of the page, as described in the `body` element is described in a template file too. Let us define it.

/src/main/webapp/resources/templates/desktop/main.html

```
<!--
    The main layout of the page - contains the menu and the 'content' &lt;div/&gt; in which ↩
         all the
    views will render the content.
-->
<div id="logo"><div class="wrap"><h1>Ticket Monster</h1></div></div>
<div id="container">
    <div id="menu">
        <div class="navbar">
            <div class="navbar-inner">
                <div class="container">
                    <ul class="nav">
                        <li><a href="#about">About</a></li>
                        <li><a href="#events">Events</a></li>
                        <li><a href="#venues">Venues</a></li>
                        <li><a href="#bookings">Bookings</a></li>
                        <li><a href="booking-monitor.html">Monitor</a></li>
                        <li><a href="admin">Administration</a></li>
                    </ul>
                </div>
            </div>
        </div>
    </div>
    <div id="content" class="container-fluid">
    </div>
</div>

<footer style="">
    <div style="text-align: center;"><img src="resources/img/dualbrand_as7eap.png" alt=" ↩
        HTML5"/></div>
</footer>
```

The actual HTML code of the template contains a menu definition which will be present on all the pages, as well as an empty element named `content`, which is the placeholder for the application views. When a view is displayed, it will apply a template and populate the `content` element.

Chapter 27

Setting up the initial views

Let us complete our application setup by creating an initial landing page. The first thing that we will need to do is to add a view component.

src/main/resources/js/app/views/desktop/home.js

```
/**
 * The About view
 */
define([
    'utilities',
    'text!../../../../templates/desktop/home.html'
], function (utilities, HomeTemplate) {

    var HomeView = Backbone.View.extend({
        render:function () {
            utilities.applyTemplate($(this.el),HomeTemplate,{});
            return this;
        }
    });

    return HomeView;
});
```

Functionally, this is a very basic component - it only renders the splash page of the application, but it helps us introduce a new concept that will be heavily used throughout the application views. One main role of a view is to describe the logic for manipulating the page content. It will do so by defining a function named `render` which will be invoked by the application. In this very simple case, all that the view does is to create the content of the splash page. You can proceed by copying the content of `src/main/webapp/resources/templates/desktop/home.html` to your project.

Backbone Views
Views are logical representations of user interface elements that can interact with data components, such as models in an event-driven fashion. Apart from defining the logical structure of your user interface, views handle events resulting from the user interaction (e.g. clicking a DOM element or selecting an element into a list), translating them into logical actions inside the application.

Once we defined a view, we must tell the router to navigate to it whenever requested. We will add the following mapping to the router:

src/main/webapp/resources/js/app/router/desktop/router.js

```
    var Router = Backbone.Router.extend({
        routes : {
```

```
            "":"home",
            "about":"home"
        },
        home : function () {
            utilities.viewManager.showView(new HomeView({el:$("#content")}));
        }
    });
```

We have just told the router to invoke the `home` function whenever the user navigates to the root of the application or uses a `#about` hash. The method will simply cause the `HomeView` defined above to render.

Now you can navigate to `http://localhost:8080/ticket-monster/#about` or `http://localhost:8080/ticket-` and see the results.

Chapter 28

Displaying Events

The first use case that we implement is event navigation. The users will be able to view the list of events and select the one that they want to attend. After doing so, they will select a venue, and will be able to choose a performance date and time.

28.1 The Event model

We a Backbone model for holding event data. Nearly all domain entities (booking, event, venue) are represented by a corresponding Backbone model:

src/main/webapp/resources/js/app/models/event.js

```
/**
 * Module for the Event model
 */
define([
    'configuration'
], function (config) {
    /**
     * The Event model class definition
     * Used for CRUD operations against individual events
     */
    var Event = Backbone.Model.extend({
        urlRoot: config.baseUrl + 'rest/events' // the URL for performing CRUD operations
    });
    // export the Event class
    return Event;
});
```

The `Event` model can perform CRUD operations against the REST services we defined earlier.

Backbone Models
Backbone models contain data as well as much of the logic surrounding it: conversions, validations, computed properties, and access control. They also perform CRUD operations with the REST service.

28.2 The Events collection

We define a Backbone collection for handling groups of events (like the events list):

src/main/webapp/resources/js/app/collections/events.js

```
/**
 * Module for the Events collection
 */
define([
    // BThe collection element type and configuration are dependencies
    'app/models/event',
    'configuration'
], function (Event, config) {
    /**
     *  Here we define the Bookings collection
     *  We will use it for CRUD operations on Bookings
     */
    var Events = Backbone.Collection.extend({
        url: config.baseUrl + "rest/events", // the URL for performing CRUD operations
        model: Event,
        id:"id", // the 'id' property of the model is the identifier
        comparator:function (model) {
            return model.get('category').id;
        }
    });
    return Events;
});
```

By mapping the model and collection to a REST endpoint you can perform CRUD operations without having to invoke the services explicitly. You will see how that works a bit later.

Backbone Collections
Collections are ordered sets of models. They can handle events which are fired as a result of a change to a individual member, and can perform CRUD operations for syncing up contents against RESTful services.

28.3 The EventsView view

Now that we have implemented the data components of the example, we need to create the view that displays them.

src/main/webapp/resources/js/app/views/desktop/events.js

```
define([
    'utilities',
    'text!../../../../templates/desktop/events.html'
], function (
    utilities,
    eventsTemplate) {

    var EventsView = Backbone.View.extend({
        events:{
            "click a":"update"
        },
        render:function () {
            var categories = _.uniq(
                _.map(this.model.models, function(model){
                    return model.get('category')
                }), false, function(item){
                    return item.id
                });
            utilities.applyTemplate($(this.el), eventsTemplate, {categories:categories,  ↩
                model:this.model})
            $(this.el).find('.item:first').addClass('active');
```

```
            $(".carousel").carousel()
            $(".collapse").collapse()
            $("a[rel='popover']").popover({trigger:'hover'});
            return this;
        },
        update:function () {
            $("a[rel='popover']").popover('hide')
        }
    });

    return  EventsView;
});
```

As we explained, earlier, the view is attached to a DOM element (the `el` property). When the `render` method is invoked, it manipulates the DOM and renders the view. We could have achieved this by writing these instructions directly in the method, but that would make it hard to change the page design later on. Instead, we create a template and apply it, thus separating the HTML view code from the view implementation.

src/main/webapp/resources/templates/desktop/events.html

```
<div class="row-fluid">
    <div class="span3">
        <div id="itemMenu">

            <%
            _.each(categories, function (category) {
            %>
            <div class="accordion-group">
                <div class="accordion-heading">
                    <a class="accordion-toggle"
                       data-target="#category-<%=category.id%>-collapsible" data-toggle=" ↩
                           collapse"
                       data-parent="#itemMenu"><%= category.description %></a>
                </div>
                <div id="category-<%=category.id%>-collapsible" class="collapse in  ↩
                    accordion-body">
                    <div id="category-<%- category.id%>" class="accordion-inner">

                        <%
                        _.each(model.models, function (model) {
                        if (model.get('category').id == category.id) {
                        %>
                        <p><a href="#events/<%- model.attributes.id%>" rel="popover"
                              data-content="<%- model.attributes.description%>"
                              data-original-title="<%- model.attributes.name%>"><%=model. ↩
                                  attributes.name%></a></p>
                        <% }
                        });
                        %>
                    </div>
                </div>
            </div>
            <% }); %>
        </div>
    </div>

    <div id='itemSummary' class="span9">
        <div class="row-fluid">
            <div class="span11">
                <div id="eventCarousel" class="carousel">
                    <!-- Carousel items -->
                    <div class="carousel-inner">
```

```
                        <%_.each(model.models, function(model) { %>
                        <div class="item">
                            <img src='rest/media/<%=model.attributes.mediaItem.id%>'/>

                            <div class="carousel-caption">
                                <h4><%=model.attributes.name%></h4>

                                <p><%=model.attributes.description%></p>
                                <a class="btn btn-danger" href="#events/<%=model.id%>">Book ↩
                                    tickets</a>
                            </div>
                        </div>
                        <% }) %>
                    </div>
                    <!-- Carousel nav -->
                    <a class="carousel-control left" href="#eventCarousel" data-slide="prev ↩
                        ">&lsaquo;</a>
                    <a class="carousel-control right" href="#eventCarousel" data-slide=" ↩
                        next">&rsaquo;</a>
                </div>
            </div>
        </div>
    </div>
</div>
```

As well as applying the template and preparing the data that will be used to fill it in (the `categories` and `model` entries in the map), the `render` method also performs the JavaScript calls that are required to initialize the UI components (in this case the Bootstrap carousel and popover).

A view can also listen to events fired by the children of it's root element (`el`). In this case, the `update` method is configured to listen to clicks on anchors. The configuration occurs within the `events` property of the class.

Now that the views are in place, we need to add another routing rule to the application.

src/main/webapp/resources/js/app/router/desktop/router.js

```
    var Router = Backbone.Router.extend({
        routes : {
            ...,
            "events":"events"
        },
        ...,
        events:function () {
            var events = new Events();
            var eventsView = new EventsView({model:events, el:$("#content")});
            events.bind("reset",
                function () {
                    utilities.viewManager.showView(eventsView);
                }).fetch();
        }
    });
```

The `events` function handles the `#events` fragment and will retrieve the events in our application via a REST call. We don't manually perform the REST call as it is triggered the by invocation of `fetch` on the `Events` collection, as discussed earlier.

The `reset` event on the collection is invoked when the data from the server is received, and the collection is populated. This triggers the rendering of the events view (which is bound to the `#content` div).

The whole process is event orientated - the models, views and controllers interact through events.

Chapter 29

Viewing a single event

With the events list view now in place, we can add a view to display the details of each individual event, allowing the user to select a venue and performance time.

We already have the models in place so all we need to do is to create the additional view and expand the router. First, we'll implement the view:

src/main/webapp/resources/js/app/views/desktop/event-detail.js

```
define([
    'utilities',
    'require',
    'text!../../../../templates/desktop/event-detail.html',
    'text!../../../../templates/desktop/media.html',
    'text!../../../../templates/desktop/event-venue-description.html',
    'configuration',
    'bootstrap'
], function (
    utilities,
    require,
    eventDetailTemplate,
    mediaTemplate,
    eventVenueDescriptionTemplate,
    config,
    Bootstrap) {

    var EventDetail = Backbone.View.extend({

        events:{
            "click input[name='bookButton']":"beginBooking",
            "change select[id='venueSelector']":"refreshShows",
            "change select[id='dayPicker']":"refreshTimes"
        },

        render:function () {
            $(this.el).empty()
            utilities.applyTemplate($(this.el), eventDetailTemplate, this.model.attributes) ↩
                ;
            $("#bookingOption").hide();
            $("#venueSelector").attr('disabled', true);
            $("#dayPicker").empty();
            $("#dayPicker").attr('disabled', true)
            $("#performanceTimes").empty();
            $("#performanceTimes").attr('disabled', true)
            var self = this
```

```
            $.getJSON(config.baseUrl + "rest/shows?event=" + this.model.get('id'), function (shows) {
                self.shows = shows
                $("#venueSelector").empty().append("<option value='0' selected>Select a venue</option>");
                $.each(shows, function (i, show) {
                    $("#venueSelector").append("<option value='" + show.id + "'>" + show.venue.address.city + " : " + show.venue.name + "</option>")
                });
                $("#venueSelector").removeAttr('disabled')
            })
            return this;
        },
        beginBooking:function () {
            require("router").navigate('/book/' + $("#venueSelector option:selected").val() + '/' + $("#performanceTimes").val(), true)
        },
        refreshShows:function (event) {
            event.stopPropagation();
            $("#dayPicker").empty();

            var selectedShowId = event.currentTarget.value;

            if (selectedShowId != 0) {
                var selectedShow = _.find(this.shows, function (show) {
                    return show.id == selectedShowId
                });
                this.selectedShow = selectedShow;
                utilities.applyTemplate($("#eventVenueDescription"), eventVenueDescriptionTemplate, {venue:selectedShow.venue});
                var times = _.uniq(_.sortBy(_.map(selectedShow.performances, function (performance) {
                    return (new Date(performance.date).withoutTimeOfDay()).getTime()
                }), function (item) {
                    return item
                }));
                utilities.applyTemplate($("#venueMedia"), mediaTemplate, selectedShow.venue)
                $("#dayPicker").removeAttr('disabled')
                $("#performanceTimes").removeAttr('disabled')
                _.each(times, function (time) {
                    var date = new Date(time)
                    $("#dayPicker").append("<option value='" + date.toYMD() + "'>" + date.toPrettyStringWithoutTime() + "</option>")
                });
                this.refreshTimes()
                $("#bookingWhen").show(100)
            } else {
                $("#bookingWhen").hide(100)
                $("#bookingOption").hide()
                $("#dayPicker").empty()
                $("#venueMedia").empty()
                $("#eventVenueDescription").empty()
                $("#dayPicker").attr('disabled', true)
                $("#performanceTimes").empty()
                $("#performanceTimes").attr('disabled', true)
            }

        },
        refreshTimes:function () {
            var selectedDate = $("#dayPicker").val();
            $("#performanceTimes").empty()
```

```
            if (selectedDate) {
                $.each(this.selectedShow.performances, function (i, performance) {
                    var performanceDate = new Date(performance.date);
                    if (_.isEqual(performanceDate.toYMD(), selectedDate)) {
                        $("#performanceTimes").append("<option value='" + performance.id +
                            "'>" + performanceDate.getHours().toZeroPaddedString(2) + ":" +
                            performanceDate.getMinutes().toZeroPaddedString(2) + "</option>"
                            )
                    }
                })
            }
            $("#bookingOption").show()
        }

    });

    return  EventDetail;
});
```

This view is more complex than the global events view, as portions of the page need to be updated when the user chooses a venue.

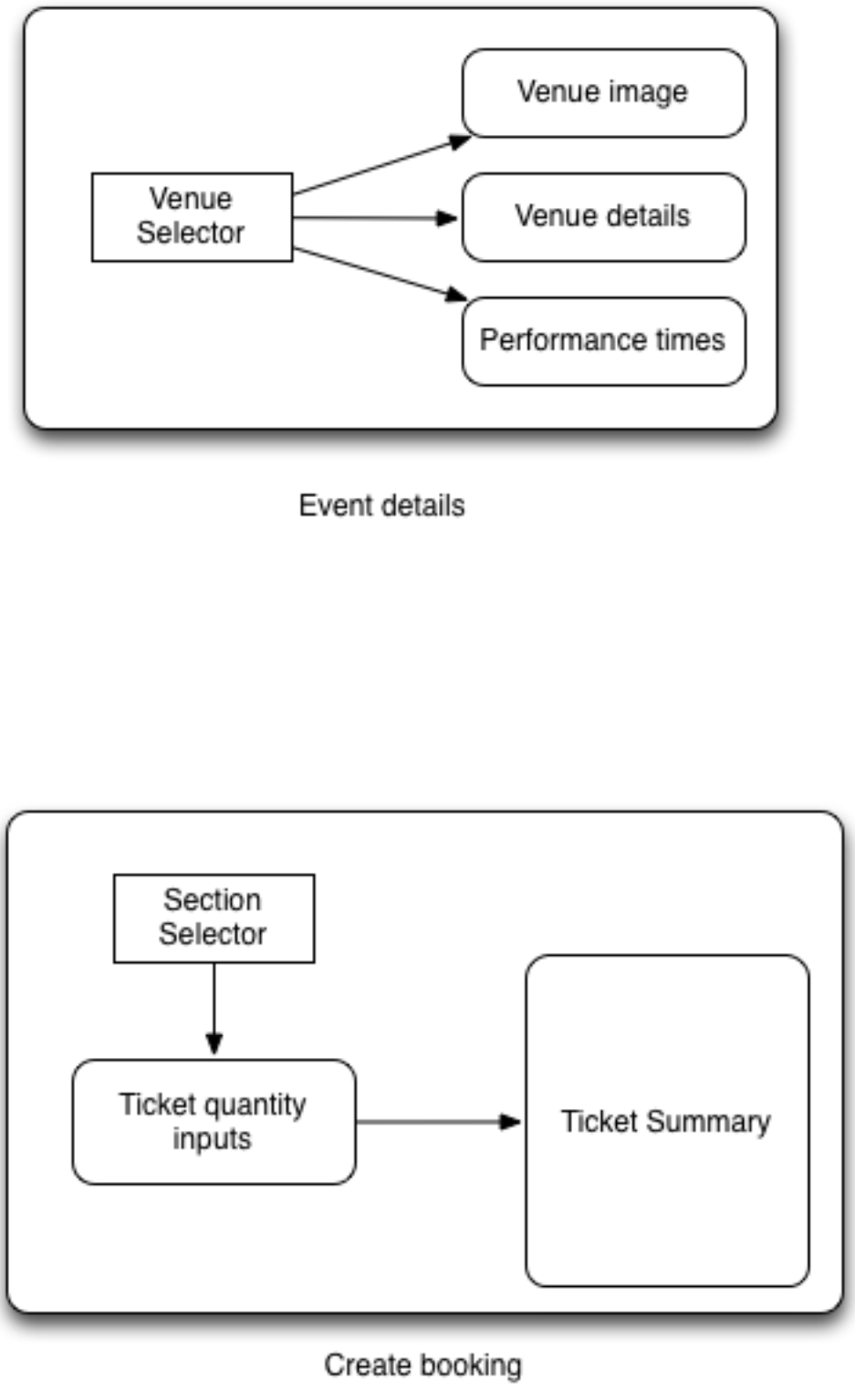

Figure 29.1: On the event details page some fragments are re-rendered when the user selects a venue

The view responds to three different events:

- changing the current venue triggers a reload of the venue details and the venue image, as well as the performance times. The

application retrieves the performance times through a REST call.

- changing the day of the performance causes the performance time selector to reload.
- once the venue and performance date and time have been selected, the user can navigate to the booking page.

The corresponding templates for the three fragments rendered above are:

src/main/webapp/resources/templates/desktop/event-detail.html

```
<div class="row-fluid" xmlns="http://www.w3.org/1999/html">
    <h2 class="page-header special-title light-font"><%=name%></h2>
</div>
<div class="row-fluid">
    <div class="span4 well">
        <div class="row-fluid"><h3 class="page-header span6">What?</h3>
            <img width="100" src='rest/media/<%=mediaItem.id%>'/></div>
        <div class="row-fluid">
            <p> </p>

            <div class="span12"><%= description %></div>
        </div>
    </div>
    <div class="span4 well">
        <div class="row-fluid"><h3 class="page-header span6">Where?</h3>
            <div class="span6" id='venueMedia'/>
        </div>
        <div class='row-fluid'><select id='venueSelector'/>
            <div id="eventVenueDescription"/>
        </div>
    </div>
    <div id='bookingWhen' style="display: none;" class="span2 well">
        <h3 class="page-header">When?</h3>
        <select class="span2" id="dayPicker"/>
        <select class="span2" id="performanceTimes"/>


        <div id='bookingOption'><input name="bookButton" class="btn btn-primary" type=" ↩
            button"
                                    value="Order tickets"></div>
    </div>
</div>
```

src/main/webapp/resources/templates/desktop/event-venue-description.html

```
<address>
    <p><%= venue.description %></p>
    <p><strong>Address:</strong></p>
    <p><%= venue.address.street %></p>
    <p><%= venue.address.city %>, <%= venue.address.country %></p>
</address>
```

Now that the view exists, we add it to the router:

src/main/webapp/resources/js/app/router/desktop/router.js

```
/**
 * A module for the router of the desktop application
 */
define("router", [
    ...
    'app/models/event',
        ...,
```

```
    'app/views/desktop/event-detail',
    ...
],function (
                        ...
            Event,
            ...
            EventDetailView,
            ...) {

    var Router = Backbone.Router.extend({
        routes:{
            ...
            "events/:id":"eventDetail",
        },
        ...
        eventDetail:function (id) {
            var model = new Event({id:id});
            var eventDetailView = new EventDetailView({model:model, el:$("#content")});
            model.bind("change",
                function () {
                    utilities.viewManager.showView(eventDetailView);
                }).fetch();
        }
});
```

As you can see, this is very similar to the previous view and route, except that now the application can accept parameterized URLs (e.g. `http://localhost:8080/ticket-monster/desktop-index#events/1`). This URL can be entered directly into the browser, or it can be navigated to as a relative path (e.g. `#events/1`) from within the applicaton.

With this in place, all that remains is to implement the final view of this use case, creating the bookings.

Chapter 30

Creating Bookings

The user has chosen the event, the venue and the performance time, and must now create the booking. Users can select one of the available sections for the show's venue, and then enter the number of tickets requiredfor each category available for this show (Adult, Child, etc.). They then add the tickets to the current order, which causes the summary view to be updated. Users can also remove tickets from the order. When the order is complete, they enter their contact information (e-mail address) and submit the order to the server.

First, we add the new view:

src/main/webapp/resources/js/app/views/desktop/create-booking.js

```
define([
    'utilities',
    'require',
    'configuration',
    'text!../../../../templates/desktop/booking-confirmation.html',
    'text!../../../../templates/desktop/create-booking.html',
    'text!../../../../templates/desktop/ticket-categories.html',
    'text!../../../../templates/desktop/ticket-summary-view.html',
    'bootstrap'
],function (
    utilities,
    require,
    config,
    bookingConfirmationTemplate,
    createBookingTemplate,
    ticketEntriesTemplate,
    ticketSummaryViewTemplate){


    var TicketCategoriesView = Backbone.View.extend({
        id:'categoriesView',
        events:{
            "keyup input":"onChange"
        },
        render:function () {
            if (this.model != null) {
                var ticketPrices = _.map(this.model, function (item) {
                    return item.ticketPrice;
                });
                utilities.applyTemplate($(this.el), ticketEntriesTemplate, {ticketPrices: ↩
                    ticketPrices});
            } else {
                $(this.el).empty();
            }
            return this;
```

```
        },
        onChange:function (event) {
            var value = event.currentTarget.value;
            var ticketPriceId = $(event.currentTarget).data("tm-id");
            var modifiedModelEntry = _.find(this.model, function (item) {
                return item.ticketPrice.id == ticketPriceId
            });
            // update model
            if ($.isNumeric(value) && value > 0) {
                modifiedModelEntry.quantity = parseInt(value);
            }
            else {
                delete modifiedModelEntry.quantity;
            }
            // display error messages
            if (value.length > 0 &&
                    (!$.isNumeric(value)  // is a non-number, other than empty string
                         || value <= 0 // is negative
                         || parseFloat(value) != parseInt(value))) { // is not an integer
                $("#error-input-"+ticketPriceId).empty().append("Please enter a positive ↩
                    integer value");
                $("#ticket-category-fieldset-"+ticketPriceId).addClass("error")
            } else {
                $("#error-input-"+ticketPriceId).empty();
                $("#ticket-category-fieldset-"+ticketPriceId).removeClass("error")
            }
            // are there any outstanding errors after this update?
            // if yes, disable the input button
            if (
               $("div[id^='ticket-category-fieldset-']").hasClass("error") ||
                   _.isUndefined(modifiedModelEntry.quantity) ) {
              $("input[name='add']").attr("disabled", true)
            } else {
              $("input[name='add']").removeAttr("disabled")
            }
        }
    });

    var TicketSummaryView = Backbone.View.extend({
        tagName:'tr',
        events:{
            "click i":"removeEntry"
        },
        render:function () {
            var self = this;
            utilities.applyTemplate($(this.el), ticketSummaryViewTemplate, this.model. ↩
                bookingRequest);
        },
        removeEntry:function () {
            this.model.bookingRequest.tickets.splice(this.model.index, 1);
        }
    });

    var CreateBookingView = Backbone.View.extend({

        events:{
            "click input[name='submit']":"save",
            "change select[id='sectionSelect']":"refreshPrices",
            "keyup #email":"updateEmail",
            "change #email":"updateEmail",
            "click input[name='add']":"addQuantities",
            "click i":"updateQuantities"
```

```
        },
        render:function () {

            var self = this;
            $.getJSON(config.baseUrl + "rest/shows/" + this.model.showId, function ( ↩
                selectedShow) {

                self.currentPerformance = _.find(selectedShow.performances, function (item) ↩
                     {
                    return item.id == self.model.performanceId;
                });

                var id = function (item) {return item.id;};
                // prepare a list of sections to populate the dropdown
                var sections = _.uniq(_.sortBy(_.pluck(selectedShow.ticketPrices, 'section' ↩
                    ), id), true, id);
                utilities.applyTemplate($(self.el), createBookingTemplate, {
                    sections:sections,
                    show:selectedShow,
                    performance:self.currentPerformance});
                self.ticketCategoriesView = new TicketCategoriesView({model:{}, el:$("# ↩
                    ticketCategoriesViewPlaceholder") });
                self.ticketSummaryView = new TicketSummaryView({model:self.model, el:$("# ↩
                    ticketSummaryView")});
                self.show = selectedShow;
                self.ticketCategoriesView.render();
                self.ticketSummaryView.render();
                $("#sectionSelector").change();
            });
            return this;
        },
        refreshPrices:function (event) {
            var ticketPrices = _.filter(this.show.ticketPrices, function (item) {
                return item.section.id == event.currentTarget.value;
            });
            var sortedTicketPrices = _.sortBy(ticketPrices, function(ticketPrice) {
                return ticketPrice.ticketCategory.description;
            });
            var ticketPriceInputs = new Array();
            _.each(sortedTicketPrices, function (ticketPrice) {
                ticketPriceInputs.push({ticketPrice:ticketPrice});
            });
            this.ticketCategoriesView.model = ticketPriceInputs;
            this.ticketCategoriesView.render();
        },
        save:function (event) {
            var bookingRequest = {ticketRequests:[]};
            var self = this;
            bookingRequest.ticketRequests = _.map(this.model.bookingRequest.tickets,  ↩
                function (ticket) {
                return {ticketPrice:ticket.ticketPrice.id, quantity:ticket.quantity}
            });
            bookingRequest.email = this.model.bookingRequest.email;
            bookingRequest.performance = this.model.performanceId
            $("input[name='submit']").attr("disabled", true)
            $.ajax({url: (config.baseUrl + "rest/bookings"),
                data:JSON.stringify(bookingRequest),
                type:"POST",
                dataType:"json",
                contentType:"application/json",
                success:function (booking) {
                    this.model = {}
```

```
                $.getJSON(config.baseUrl +'rest/shows/performance/' + booking.performance.id, function (retrievedPerformance) {
                    utilities.applyTemplate($(self.el), bookingConfirmationTemplate, { booking:booking, performance:retrievedPerformance })
                });
            }}).error(function (error) {
                if (error.status == 400 || error.status == 409) {
                    var errors = $.parseJSON(error.responseText).errors;
                    _.each(errors, function (errorMessage) {
                        $("#request-summary").append('<div class="alert alert-error"><a class="close" data-dismiss="alert">$\times$</a><strong>Error!</strong> ' + errorMessage + '</div>')
                    });
                } else {
                    $("#request-summary").append('<div class="alert alert-error"><a class="close" data-dismiss="alert">$\times$</a><strong>Error! </strong>An error has occured</div>')
                }
                $("input[name='submit']").removeAttr("disabled");
            })

    },
    addQuantities:function () {
        var self = this;
        _.each(this.ticketCategoriesView.model, function (model) {
            if (model.quantity != undefined) {
                var found = false;
                _.each(self.model.bookingRequest.tickets, function (ticket) {
                    if (ticket.ticketPrice.id == model.ticketPrice.id) {
                        ticket.quantity += model.quantity;
                        found = true;
                    }
                });
                if (!found) {
                    self.model.bookingRequest.tickets.push({ticketPrice:model.ticketPrice, quantity:model.quantity});
                }
            }
        });
        this.ticketCategoriesView.model = null;
        $('option:selected', 'select').removeAttr('selected');
        this.ticketCategoriesView.render();
        this.updateQuantities();
    },
    updateQuantities:function () {
        // make sure that tickets are sorted by section and ticket category
        this.model.bookingRequest.tickets.sort(function (t1, t2) {
            if (t1.ticketPrice.section.id != t2.ticketPrice.section.id) {
                return t1.ticketPrice.section.id - t2.ticketPrice.section.id;
            }
            else {
                return t1.ticketPrice.ticketCategory.id - t2.ticketPrice.ticketCategory.id;
            }
        });

        this.model.bookingRequest.totals = _.reduce(this.model.bookingRequest.tickets, function (totals, ticketRequest) {
            return {
                tickets:totals.tickets + ticketRequest.quantity,
                price:totals.price + ticketRequest.quantity * ticketRequest.ticketPrice.price
```

```
                };
            }, {tickets:0, price:0.0});

            this.ticketSummaryView.render();
            this.setCheckoutStatus();
        },
        updateEmail:function (event) {
            if ($(event.currentTarget).is(':valid')) {
                this.model.bookingRequest.email = event.currentTarget.value;
                $("#error-email").empty();
            } else {
                $("#error-email").empty().append("Please enter a valid e-mail address");
                delete this.model.bookingRequest.email;
            }
            this.setCheckoutStatus();
        },
        setCheckoutStatus:function () {
            if (this.model.bookingRequest.totals != undefined && this.model.bookingRequest. ↩
                totals.tickets > 0 && this.model.bookingRequest.email != undefined && this. ↩
                model.bookingRequest.email != '') {
                $('input[name="submit"]').removeAttr('disabled');
            }
            else {
                $('input[name="submit"]').attr('disabled', true);
            }
        }
    });

    return CreateBookingView;
});
```

The code above may be surprising! After all, we said that we were going to add a single view, but instead, we added three! This view makes use of two subviews (`TicketCategoriesView` and `TicketSummaryView`) for re-rendering parts of the main view. Whenever the user changes the current section, the list of available tickets is updated. Whenever the user adds the tickets to the booking, the booking summary is re-rendered. Changes in quantities or the target email may enable or disable the submission button - the booking is validated whenever changes to it are made. We do not create separate modules for the subviews, since they are not referenced outside the module itself.

The booking submission is handled by the `save` method which constructs a JSON object, as required by a POST to `http://localhos` and performs the AJAX call. In case of a successful response, a confirmation view is rendered. On failure, a warning is displayed and the user may continue to edit the form.

The corresponding templates for the views above are shown below:

src/main/webapp/resources/templates/desktop/booking-confirmation.html

```
<div class="row-fluid">
    <h2 class="special-title light-font">Booking #<%=booking.id%> confirmed!</h2>
</div>
<div class="row-fluid">
    <div class="span5 well">
        <h4 class="page-header">Checkout information</h4>
        <p><strong>Email: </strong><%= booking.contactEmail %></p>
        <p><strong>Event: </strong> <%= performance.event.name %></p>
        <p><strong>Venue: </strong><%= performance.venue.name %></p>
        <p><strong>Date: </strong><%= new Date(booking.performance.date).toPrettyString() % ↩
            ></p>
        <p><strong>Created on: </strong><%= new Date(booking.createdOn).toPrettyString() %> ↩
            </p>
    </div>
    <div class="span5 well">
        <h4 class="page-header">Ticket allocations</h4>
```

```
        <table class="table table-striped table-bordered" style="background-color: #fffffa; ↩
            ">
            <thead>
            <tr>
                <th>Ticket #</th>
                <th>Category</th>
                <th>Section</th>
                <th>Row</th>
                <th>Seat</th>
            </tr>
            </thead>
            <tbody>
            <% $.each(_.sortBy(booking.tickets, function(ticket) {return ticket.id}),  ↩
                function (i, ticket) { %>
            <tr>
                <td><%= ticket.id %></td>
                <td><%=ticket.ticketCategory.description%></td>
                <td><%=ticket.seat.section.name%></td>
                <td><%=ticket.seat.rowNumber%></td>
                <td><%=ticket.seat.number%></td>
            </tr>
            <% }) %>
            </tbody>
        </table>
    </div>
</div>
<div class="row-fluid" style="padding-bottom:30px;">
    <div class="span2"><a href="#">Home</a></div>
</div>
```

src/main/webapp/resources/templates/desktop/create-booking.html

```
<div class="row-fluid">
    <div class="span12">
        <h2 class="special-title light-font"><%=show.event.name%>
            <small><%=show.venue.name%>, <%=new Date(performance.date).toPrettyString()%></ ↩
                p></small>
        </h2>
    </div>
</div>
<div class="row-fluid">
    <div class="span6 well">
       <h3 class="page-header">Select tickets</h3>
        <form class="form-horizontal">
        <div id="sectionSelectorPlaceholder">
            <div class="control-group">
                <label class="control-label" for="sectionSelect"><strong>Section</strong></ ↩
                    label>
                <div class="controls">
                    <select id="sectionSelect">
                        <option value="-1" selected="true">Choose a section</option>
                        <% _.each(sections, function(section) { %>
                        <option value="<%=section.id%>"><%=section.name%> - <%=section. ↩
                            description%></option>
                        <% }) %>
                    </select>
                </div>
            </div>
        </div>
        </form>
        <div id="ticketCategoriesViewPlaceholder"></div>
    </div>
```

```
    <div id="request-summary" class="span5 offset1 well">
        <h3 class="page-header">Order summary</h3>
        <div id="ticketSummaryView" class="row-fluid"/>
        <h3 class="page-header">Checkout</h3>
        <div class="row-fluid">
            <form class="form-search">
            <input type='email' id="email" placeholder="Email" required/>
            <input type='button' class="btn btn-primary" name="submit" value="Checkout"
                   disabled="true"/>
            <p class="help-block error-notification"  id="error-email"></p>
            </form>
        </div>
    </div>
</div>
```

src/main/webapp/resources/templates/desktop/ticket-categories.html

```
<% if (ticketPrices.length > 0) { %>
<form class="form-horizontal">
    <% _.each(ticketPrices, function(ticketPrice) { %>
    <div class="control-group" id="ticket-category-fieldset-<%=ticketPrice.id%>">
        <label class="control-label"><strong><%=ticketPrice.ticketCategory.description%></ ↩
            strong></label>

        <div class="controls">
            <div class="input-append">
                <input class="span2" rel="tooltip" title="Enter value"
                       data-tm-id="<%=ticketPrice.id%>"
                       placeholder="Number of tickets"
                       name="tickets-<%=ticketPrice.ticketCategory.id%>"/>
                <span class="add-on">@ $<%=ticketPrice.price%></span>

                <p class="help-block" id="error-input-<%=ticketPrice.id%>"></p>
            </div>
        </div>
    </div>
    <% }) %>

<p> </p>

<div class="control-group">
    <label class="control-label"/>

    <div class="controls">
        <input type="button" class="btn btn-primary" disabled="true" name="add" value="Add  ↩
            tickets"/>
    </div>
</div>
</div>
</form>
<% } %>
```

src/main/webapp/resources/templates/desktop/ticket-summary-view.html

```
<div class="span12">
    <% if (tickets.length>0) { %>
    <table class="table table-bordered table-condensed row-fluid" style="background-color:  ↩
        #fffffa;">
        <thead>
        <tr>
            <th colspan="5"><strong>Requested tickets</strong></th>
        </tr>
```

```
        <tr>
            <th>Section</th>
            <th>Category</th>
            <th>Quantity</th>
            <th>Price</th>
            <th></th>
        </tr>
        </thead>
        <tbody id="ticketRequestSummary">
        <% _.each(tickets, function (ticketRequest, index, tickets) { %>
        <tr>
            <td><%= ticketRequest.ticketPrice.section.name %></td>
            <td><%= ticketRequest.ticketPrice.ticketCategory.description %></td>
            <td><%= ticketRequest.quantity %></td>
            <td>$<%=ticketRequest.ticketPrice.price%></td>
            <td><i class="icon-trash"/></td>
        </tr>
        <% }); %>
        </tbody>
    </table>
    <p/>
    <div class="row-fluid">
        <div class="span5"><strong>Total ticket count:</strong> <%= totals.tickets %></div>
        <div class="span5"><strong>Total price:</strong> $<%=totals.price%></div></div>
    <% } else { %>
    No tickets requested.
    <% } %>
</div>
```

Finally, once the view is available, we can add it's corresponding routing rule:

src/main/webapp/resources/js/app/router/desktop/router.js

```
/**
 * A module for the router of the desktop application
 */
define("router", [
    ...
    'app/views/desktop/create-booking',
        ...
],function (
                        ...
            CreateBooking
            ...
            ) {

    var Router = Backbone.Router.extend({
        routes:{
            ...
            "book/:showId/:performanceId":"bookTickets",
        },
        ...
        bookTickets:function (showId, performanceId) {
            var createBookingView =
                new CreateBookingView({
                    model:{ showId:showId,
                            performanceId:performanceId,
                            bookingRequest:{tickets:[]}},
                            el:$("#content")
                           });
            utilities.viewManager.showView(createBookingView);
        }
```

```
});
```

This concludes the implementation of the booking use case. We started by listing the available events, continued by selecting a venue and performance time, and ended by choosing tickets and completing the order.

The other use cases: a booking starting from venues and vi existing bookings are conceptually similar, so you can just copy the remaining files in the `src/main/webapp/resources/js/app/models`, `src/main/webapp/resources/js/app/colle` `src/main/webapp/resources/js/app/views/desktop` and the remainder of `src/main/webapp/resources/js/a`

Chapter 31

Mobile view

The mobile version of the application uses approximately the same architecture as the desktop version. Any differences are due to the functional changes in the mobile version and the use of jQuery mobile.

31.1 Setting up the structure

The first step in implementing our solution is to copy the CSS and JavaScript libraries to `resources/css` and `resources/js/lib`:

require.js
AMD support, along with the plugins:

- text - for loading text files, in our case the HTML templates
- order - for enforcing load ordering if necessary

jQuery
general purpose library for HTML traversal and manipulation

Underscore
JavaScript utility library (and a dependency of Backbone)

Backbone
Client-side MVC framework

jQuery Mobile
user interface system for mobile devices;

(If you have already built the desktop application, some files may already be in place.)

For mobile clients, the main page will display the mobile version of the application, by loading the mobile AMD module of the application. Let us create it.

/src/main/webapp/resources/js/configurations/mobile.js

```
/**
 * Shortcut alias definitions - will come in handy when declaring dependencies
 * Also, they allow you to keep the code free of any knowledge about library
 * locations and versions
 */
require.config({
    baseUrl:"resources/js",
    paths: {
        jquery:'libs/jquery-1.7.1',
        jquerymobile:'libs/jquery.mobile-1.1.0',
```

```
        text:'libs/text',
        underscore:'libs/underscore',
        backbone: 'libs/backbone',
        order: 'libs/order',
        utilities: 'app/utilities'
    },
    // We shim Backbone since it doesn't declare an AMD module
    shim: {
        'backbone': {
            deps: ['underscore', 'jquery'],
            exports: 'Backbone'
        }
    }
});

define("configuration", function() {
    if (window.TicketMonster != undefined && TicketMonster.config != undefined) {
        return {
            baseUrl: TicketMonster.config.baseRESTUrl
        };
    } else {
        return {
            baseUrl: ""
        }
    }
})
define("initializer", [
    'jquery',
    'utilities',
    'text!../templates/mobile/main.html'
], function ($,
             utilities,
             MainTemplate) {
    $('head').append('<link rel="stylesheet" href="resources/css/jquery.mobile-1.1.0.css"/> ↩
        ');
    $('head').append('<link rel="stylesheet" href="resources/css/m.screen.css"/>');
    $(document).bind("mobileinit", function () {
        utilities.applyTemplate($('body'), MainTemplate)
    });
});


// Now we declare all the dependencies
require(['order!initializer',
         'order!underscore',
         'order!backbone'],
    function(){
});

define(["configuration"],function(configuration){
    return {config: configuration };
})
```

In this application, we combine Backbone and jQuery Mobile. Each framework has its own strengths; jQuery Mobile provides UI components and touch support, whilst Backbone provides MVC support. There is some overlap between the two, as jQuery Mobile provides its own navigation mechanism which we disable.

We also define a special initializer module (`initializer`) that, when loaded, adds the stylesheets and applies the template for the general structure of the page in the `body` element. Let us add the template definition.

We also define a `configuration` module which allows the customizaton of the base URLs for RESTful invocations. This module does not play any role in the mobile web version. We will come to it, however, when discussing hybrid applications.

src/main/webapp/resources/templates/mobile/main.html

```
<!--
    The main layout of the page - contains the menu and the 'content' &lt;div/&gt; in which ↩
        all the
    views will render the content.
-->
<div id="container" data-role="page" data-ajax="false"></div>
```

Next, we create the application router.

src/main/webapp/resources/js/app/router/mobile/router.js

```
/**
 * A module for the router of the desktop application.
 *
 */
define("router",[
    'jquery',
    'jquerymobile',
    'underscore',
    'utilities',
    'text!../templates/mobile/home-view.html'
],function ($,
            jqm,
            _,
            utilities,
            HomeViewTemplate) {

    // prior to creating an starting the router, we disable jQuery Mobile's own routing ↩
        mechanism
    $.mobile.hashListeningEnabled = false;
    $.mobile.linkBindingEnabled = false;
    $.mobile.pushStateEnabled = false;

    /**
     * The Router class contains all the routes within the application - i.e. URLs and the ↩
         actions
     * that will be taken as a result.
     *
     * @type {Router}
     */
    var Router = Backbone.Router.extend({
        defaultHandler:function (actions) {
            if ("" != actions) {
                $.mobile.changePage("#" + actions, {transition:'slide', changeHash:false, ↩
                    allowSamePageTransition:true});
            }
    });

    // Create a router instance
    var router = new Router();

    // Begin routing
    Backbone.history.start();

    return router;
});
```

In the router code we make customizations in order to get the two frameworks working together - disabling the jQuery Mobile navigation and adding the `defaultHandler` to the router for handling jQuery Mobile transitions between internal pages (such as the ones generated by a nested listview).

Next, we need to create a first page.

31.2 The landing page

The first page in our application is the landing page. First, we add the template for it:

src/main/webapp/resources/templates/mobile/home-view.html

```
<div data-role="header">
    <h3>Ticket Monster</h3>
</div>
<div data-role="content" align="center">
    <img src="resources/img/dualbrand_as7eap.png" width="300px"/>
    <h4 align="left">Find events</h4>
    <ul data-role="listview">
        <li>
            <a href="#events">By Category</a>
        </li>
        <li>
            <a href="#venues">By Location</a>
        </li>
    </ul>
</div>
```

Now we have to add the page to the router:

src/main/webapp/resources/js/app/router/mobile/router.js

```
/**
 * A module for the router of the desktop application.
 *
 */
define("router",[
    ...
    'text!../templates/mobile/home-view.html'
],function (
                ...
        HomeViewTemplate) {

        ...
    var Router = Backbone.Router.extend({
        routes:{
            "":"home"
        },
        ...
        home:function () {
            utilities.applyTemplate($("#container"), HomeViewTemplate);
            try {
                $("#container").trigger('pagecreate');
            } catch (e) {
                // workaround for a spurious error thrown when creating the page initially
            }
        }
    });
    ...
});
```

Because jQuery Mobile navigation is disabled, we must tell jQuery Mobile explicitly to enhance the page content in order to create the mobile view. Here, we trigger the jQuery Mobile `pagecreate` event explicitly to ensure that the page gets the appropriate look and feel.

31.3 The events view

First, we display a list of events (just as in the desktop view). Since mobile interfaces are more constrained, we will just show a simple list view:

src/main/webapp/resources/js/app/views/mobile/events.js

```
define([
    'utilities',
    'text!../../../../templates/mobile/events.html'
], function (
    utilities,
    eventsView) {

    var EventsView = Backbone.View.extend({
        render:function () {
            var categories = _.uniq(
                _.map(this.model.models, function(model){
                    return model.get('category')
                }), false, function(item){
                    return item.id
                });
            utilities.applyTemplate($(this.el), eventsView,  {categories:categories, model: ↩
                this.model})
            $(this.el).trigger('pagecreate');
            return this;
        }
    });

    return EventsView;
});
```

As you can see, the view is very similar to the desktop view, the main difference being the explicit hint to jQuery mobile through the `pagecreate` event invocation.

Next, we add the template for rendering the view:

src/main/webapp/resources/templates/mobile/events.html

```
<div data-role="header">
    <a data-role="button" data-icon="home" href="#">Home</a>
    <h3>Categories</h3>
</div>
<div data-role="content" id='itemMenu'>
    <div id='categoryMenu' data-role='listview' data-filter='true' data-filter-placeholder ↩
        ='Event category name ...'>
        <%
        _.each(categories, function (category) {
        %>
        <li>
            <a href="#"><%= category.description %></a>
            <ul id="category-<%=category.id%>">
                <%
                _.each(model.models, function (model) {
                if (model.get('category').id == category.id) {
                %>
                <li>
                    <a href="#events/<%=model.attributes.id%>"><%=model.attributes.name%></ ↩
                        a>
                </li>
                <% }
                });
```

```
                %>
            </ul>
        </li>
        <% }); %>
    </div>
</div>
```

And finally, we need to instruct the router to invoke the page:

src/main/webapp/resources/js/app/router/mobile/router.js

```
/**
 * A module for the router of the desktop application.
 *
 */
define("router",[
    ...
        'app/collections/events',
        ...
        'app/views/mobile/events'
        ...
],function (
        ...,
        Events,
        ...,
        EventsView,
        ...) {

        ...
    var Router = Backbone.Router.extend({
        routes:{
                ...
            "events":"events"
            ...
        },
        ...
        events:function () {
            var events = new Events;
            var eventsView = new EventsView({model:events, el:$("#container")});
            events.bind("reset",
                function () {
                    utilities.viewManager.showView(eventsView);
                }).fetch();
        }
        ...
    });
    ...
});
```

Just as in the case of the desktop application, the list of events will be accessible at `#events` (i.e. `http://localhost:8080/tick`

31.4 Displaying an individual event

Now, we create the view to display an event:

src/main/webapp/resources/js/app/views/mobile/event-detail.js

```
define([
    'utilities',
    'require',
```

```
    'configuration',
    'text!../../../../templates/mobile/event-detail.html',
    'text!../../../../templates/mobile/event-venue-description.html'
], function (
    utilities,
    require,
    config,
    eventDetail,
    eventVenueDescription) {

    var EventDetailView = Backbone.View.extend({
        events:{
            "click a[id='bookButton']":"beginBooking",
            "change select[id='showSelector']":"refreshShows",
            "change select[id='performanceTimes']":"performanceSelected",
            "change select[id='dayPicker']":'refreshTimes'
        },
        render:function () {
            $(this.el).empty()
            utilities.applyTemplate($(this.el), eventDetail, this.model.attributes)
            $(this.el).trigger('create')
            $("#bookButton").addClass("ui-disabled")
            var self = this;
            $.getJSON(config.baseUrl + "rest/shows?event=" + this.model.get('id'), function ↩
                 (shows) {
                self.shows = shows;
                $("#showSelector").empty().append("<option data-placeholder='true'>Choose a ↩
                     venue ...</option>");
                $.each(shows, function (i, show) {
                    $("#showSelector").append("<option value='" + show.id + "'>" + show. ↩
                        venue.address.city + " : " + show.venue.name + "</option>");
                });
                $("#showSelector").selectmenu('refresh', true)
                $("#dayPicker").selectmenu('disable')
                $("#dayPicker").empty().append("<option data-placeholder='true'>Choose a  ↩
                    show date ...</option>")
                $("#performanceTimes").selectmenu('disable')
                $("#performanceTimes").empty().append("<option data-placeholder='true'> ↩
                    Choose a show time ...</option>")
            });
            $("#dayPicker").empty();
            $("#dayPicker").selectmenu('disable');
            $("#performanceTimes").empty();
            $("#performanceTimes").selectmenu('disable');
            $(this.el).trigger('pagecreate');
            return this;
        },
        performanceSelected:function () {
            if ($("#performanceTimes").val() != 'Choose a show time ...') {
                $("#bookButton").removeClass("ui-disabled")
            } else {
                $("#bookButton").addClass("ui-disabled")
            }
        },
        beginBooking:function () {
            require('router').navigate('book/' + $("#showSelector option:selected").val() + ↩
                '/' + $("#performanceTimes").val(), true)
        },
        refreshShows:function (event) {

            var selectedShowId = event.currentTarget.value;
```

```
            if (selectedShowId != 'Choose a venue ...') {
                var selectedShow = _.find(this.shows, function (show) {
                    return show.id == selectedShowId
                });
                this.selectedShow = selectedShow;
                var times = _.uniq(_.sortBy(_.map(selectedShow.performances, function ( ↩
                    performance) {
                    return (new Date(performance.date).withoutTimeOfDay()).getTime()
                }), function (item) {
                    return item
                }));
                utilities.applyTemplate($("#eventVenueDescription"), eventVenueDescription, ↩
                     {venue:selectedShow.venue});
                $("#detailsCollapsible").show()
                $("#dayPicker").removeAttr('disabled')
                $("#performanceTimes").removeAttr('disabled')
                $("#dayPicker").empty().append("<option data-placeholder='true'>Choose a  ↩
                    show date ...</option>")
                _.each(times, function (time) {
                    var date = new Date(time)
                    $("#dayPicker").append("<option value='" + date.toYMD() + "'>" + date. ↩
                        toPrettyStringWithoutTime() + "</option>")
                });
                $("#dayPicker").selectmenu('refresh')
                $("#dayPicker").selectmenu('enable')
                this.refreshTimes()
            } else {
                $("#detailsCollapsible").hide()
                $("#eventVenueDescription").empty()
                $("#dayPicker").empty()
                $("#dayPicker").selectmenu('disable')
                $("#performanceTimes").empty()
                $("#performanceTimes").selectmenu('disable')
            }


        },
        refreshTimes:function () {
            var selectedDate = $("#dayPicker").val();
            $("#performanceTimes").empty().append("<option data-placeholder='true'>Choose a ↩
                 show time ...</option>")
            if (selectedDate) {
                $.each(this.selectedShow.performances, function (i, performance) {
                    var performanceDate = new Date(performance.date);
                    if (_.isEqual(performanceDate.toYMD(), selectedDate)) {
                        $("#performanceTimes").append("<option value='" + performance.id +  ↩
                            "'>" + performanceDate.getHours().toZeroPaddedString(2) + ":" +  ↩
                            performanceDate.getMinutes().toZeroPaddedString(2) + "</option>" ↩
                            )
                    }
                })
                $("#performanceTimes").selectmenu('enable')
            }
            $("#performanceTimes").selectmenu('refresh')
            this.performanceSelected()
        }

    });

    return EventDetailView;
});
```

Once again, this is very similar to the desktop version. Now we add the page templates:

src/main/webapp/resources/templates/mobile/event-detail.html

```
<div data-role="header">
    <h3>Book tickets</h3>
</div>
<div data-role="content">
    <h3><%=name%></h3>
    <img width='100px' src='rest/media/<%=mediaItem.id%>'/>
    <p><%=description%></p>
    <div data-role="fieldcontain">
        <label for="showSelector"><strong>Where</strong></label>
        <select id='showSelector' data-mini='true'/>
    </div>

    <div data-role="collapsible" data-content-theme="c" style="display: none;"
         id="detailsCollapsible">
        <h3>Venue details</h3>

        <div id="eventVenueDescription">
        </div>
    </div>

    <div data-role='fieldcontain'>
        <fieldset data-role='controlgroup'>
            <legend><strong>When</strong></legend>
            <label for="dayPicker">When:</label>
            <select id='dayPicker' data-mini='true'/>

            <label for="performanceTimes">When:</label>
            <select id="performanceTimes" data-mini='true'/>

        </fieldset>
    </div>

</div>
<div data-role="footer" class="ui-bar ui-grid-c">
    <div class="ui-block-a"></div>
    <div class="ui-block-b"></div>
    <div class="ui-block-c"></div>
    <a id='bookButton' class="ui-block-e" data-theme='b' data-role="button" data-icon="
       check">Book</a>
</div>
```

src/main/webapp/resources/templates/mobile/event-venue-description.html

```
<img width="100" src="rest/media/<%=venue.mediaItem.id%>"/></p>
<%= venue.description %>
<address>
    <p><strong>Address:</strong></p>
    <p><%= venue.address.street %></p>
    <p><%= venue.address.city %>, <%= venue.address.country %></p>
</address>
```

Finally, we add this to the router, explicitly indicating to jQuery Mobile that a transition has to take place after the view is rendered - in order to allow the page to render correctly after it has been invoked from the listview.

src/main/webapp/resources/js/app/router/mobile/router.js

```
/**
 * A module for the router of the desktop application.
 *
```

```
 */
define("router",[
    ...
        'app/model/event',
        ...
        'app/views/mobile/event-detail'
        ...
],function (
        ...,
        Event,
        ...,
        EventDetailView,
        ...) {

        ...
    var Router = Backbone.Router.extend({
        routes:{
                ...
            "events/:id":"eventDetail",
            ...
        },
        ...
        eventDetail:function (id) {
            var model = new Event({id:id});
            var eventDetailView = new EventDetailView({model:model, el:$("#container")});
            model.bind("change",
                function () {
                    utilities.viewManager.showView(eventDetailView);
                    $.mobile.changePage($("#container"), {transition:'slide', changeHash: ↩
                        false});
                }).fetch();
        }
        ...
    });
    ...
});
```

Just as the desktop version, the mobile event detail view allows users to choose a venue and a performance time. The next step is to allow the user to book some tickets.

31.5 Booking tickets

The views to book tickets are simpler than the desktop version. Users can select a section and enter the number of tickets for each category however, there is no way to add or remove tickets from an order. Once the form is filled out, the user can only submit it.

First, we create the views:

src/main/webapp/resources/js/app/views/mobile/create-booking.js

```
define([
    'utilities',
    'configuration',
    'require',
    'text!../../../../templates/mobile/booking-details.html',
    'text!../../../../templates/mobile/create-booking.html',
    'text!../../../../templates/mobile/confirm-booking.html',
    'text!../../../../templates/mobile/ticket-entries.html',
    'text!../../../../templates/mobile/ticket-summary-view.html'
], function (
```

```
    utilities,
    config,
    require,
    bookingDetailsTemplate,
    createBookingTemplate,
    confirmBookingTemplate,
    ticketEntriesTemplate,
    ticketSummaryViewTemplate) {

    var TicketCategoriesView = Backbone.View.extend({
        id:'categoriesView',
        events:{
            "change input":"onChange"
        },
        render:function () {
            var views = {};

            if (this.model != null) {
                var ticketPrices = _.map(this.model, function (item) {
                    return item.ticketPrice;
                });
                utilities.applyTemplate($(this.el), ticketEntriesTemplate, {ticketPrices: ↩
                    ticketPrices});
            } else {
                $(this.el).empty();
            }
            $(this.el).trigger('pagecreate');
            return this;
        },
        onChange:function (event) {
            var value = event.currentTarget.value;
            var ticketPriceId = $(event.currentTarget).data("tm-id");
            var modifiedModelEntry = _.find(this.model, function(item) { return item. ↩
                ticketPrice.id == ticketPriceId});
            if ($.isNumeric(value) && value > 0) {
                modifiedModelEntry.quantity = parseInt(value);
            }
            else {
                delete modifiedModelEntry.quantity;
            }
        }
    });

     var TicketSummaryView = Backbone.View.extend({
        render:function () {
            utilities.applyTemplate($(this.el), ticketSummaryViewTemplate, this.model. ↩
                bookingRequest)
        }
    });

    var ConfirmBookingView = Backbone.View.extend({
        events:{
            "click a[id='saveBooking']":"save",
            "click a[id='goBack']":"back"
        },
        render:function () {
            utilities.applyTemplate($(this.el), confirmBookingTemplate, this.model)
            this.ticketSummaryView = new TicketSummaryView({model:this.model, el:$("# ↩
                ticketSummaryView")});
            this.ticketSummaryView.render();
            $(this.el).trigger('pagecreate')
        },
```

```
        back:function () {
            require("router").navigate('book/' + this.model.bookingRequest.show.id + '/' +  ↩
                this.model.bookingRequest.performance.id, true)

        }, save:function (event) {
            var bookingRequest = {ticketRequests:[]};
            var self = this;
            _.each(this.model.bookingRequest.tickets, function (collection) {
                _.each(collection, function (model) {
                    if (model.quantity != undefined) {
                        bookingRequest.ticketRequests.push({ticketPrice:model.ticketPrice. ↩
                            id, quantity:model.quantity})
                    };
                })
            });

            bookingRequest.email = this.model.email;
            bookingRequest.performance = this.model.performanceId;
            $.ajax({url:(config.baseUrl + "rest/bookings"),
                data:JSON.stringify(bookingRequest),
                type:"POST",
                dataType:"json",
                contentType:"application/json",
                success:function (booking) {
                    utilities.applyTemplate($(self.el), bookingDetailsTemplate, booking)
                    $(self.el).trigger('pagecreate');
                }}).error(function (error) {
                    alert(error);
                });
            this.model = {};
        }
    });


    var CreateBookingView = Backbone.View.extend({

        events:{
            "click a[id='confirmBooking']":"checkout",
            "change select":"refreshPrices",
            "blur input[type='number']":"updateForm",
            "blur input[name='email']":"updateForm"
        },
        render:function () {

            var self = this;

            $.getJSON(config.baseUrl + "rest/shows/" + this.model.showId, function ( ↩
                selectedShow) {
                self.model.performance = _.find(selectedShow.performances, function (item)  ↩
                    {
                    return item.id == self.model.performanceId;
                });
                var id = function (item) {return item.id;};
                // prepare a list of sections to populate the dropdown
                var sections = _.uniq(_.sortBy(_.pluck(selectedShow.ticketPrices, 'section' ↩
                    ), id), true, id);

                utilities.applyTemplate($(self.el), createBookingTemplate, { show: ↩
                    selectedShow,
                    performance:self.model.performance,
                    sections:sections});
                $(self.el).trigger('pagecreate');
```

```
                self.ticketCategoriesView = new TicketCategoriesView({model:{}, el:$("# ↩
                    ticketCategoriesViewPlaceholder") });
                self.model.show = selectedShow;
                self.ticketCategoriesView.render();
                $('a[id="confirmBooking"]').addClass('ui-disabled');
                $("#sectionSelector").change();
            });

        },
        refreshPrices:function (event) {
            if (event.currentTarget.value != "Choose a section") {
                var ticketPrices = _.filter(this.model.show.ticketPrices, function (item) {
                    return item.section.id == event.currentTarget.value;
                });
                var ticketPriceInputs = new Array();
                _.each(ticketPrices, function (ticketPrice) {
                    var model = {};
                    model.ticketPrice = ticketPrice;
                    ticketPriceInputs.push(model);
                });
                $("#ticketCategoriesViewPlaceholder").show();
                this.ticketCategoriesView.model = ticketPriceInputs;
                this.ticketCategoriesView.render();
                $(this.el).trigger('pagecreate');
            } else {
                $("#ticketCategoriesViewPlaceholder").hide();
                this.ticketCategoriesView.model = new Array();
                this.updateForm();
            }
        },
        checkout:function () {
            this.model.bookingRequest.tickets.push(this.ticketCategoriesView.model);
            this.model.performance = new ConfirmBookingView({model:this.model, el:$("# ↩
                container")}).render();
            $("#container").trigger('pagecreate');
        },
        updateForm:function () {

            var totals = _.reduce(this.ticketCategoriesView.model, function (partial, model ↩
                ) {
                if (model.quantity != undefined) {
                    partial.tickets += model.quantity;
                    partial.price += model.quantity * model.ticketPrice.price;
                    return partial;
                }
            }, {tickets:0, price:0.0});
            this.model.email = $("input[type='email']").val();
            this.model.bookingRequest.totals = totals;
            if (totals.tickets > 0 && $("input[type='email']").val()) {
                $('a[id="confirmBooking"]').removeClass('ui-disabled');
            } else {
                $('a[id="confirmBooking"]').addClass('ui-disabled');
            }
        }
    });
    return CreateBookingView;
});
```

The views follow the structure the desktop application, except that the summary view is not rendered inline but after a page transition.

Next, we create the page fragment templates. First, the actual page:

src/main/webapp/resources/templates/mobile/create-booking.html

```
<div data-role="header">
    <h1>Book tickets</h1>
</div>
<div data-role="content">
    <p>
       <h3><%=show.event.name%></h3>
    </p>
    <p>
      <%=show.venue.name%>
    <p>

    <p>
      <small><%=new Date(performance.date).toPrettyString()%></small>
    </p>
    <div id="sectionSelectorPlaceholder">
        <div data-role="fieldcontain">
            <label for="sectionSelect">Section</label>
            <select id="sectionSelect">
                <option value="-1" selected="true">Choose a section</option>
                <% _.each(sections, function(section) { %>
                <option value="<%=section.id%>"><%=section.name%> - <%=section.description% ↩
                    ></option>
                <% }) %>
            </select>
        </div>

    </div>
    <div id="ticketCategoriesViewPlaceholder" style="display:none;"/>

    <div class="fieldcontain">
        <label>Contact email</label>
        <input type='email' name='email' placeholder="Email"/>
    </div>
</div>

<div data-role="footer" class="ui-bar">
    <a href="#" data-role="button" data-icon="delete">Cancel</a>
    <a id="confirmBooking" data-icon="check" data-role="button" disabled>Checkout</a>
</div>
```

Next, the fragment that contains the input form for tickets, which is re-rendered whenever the section is changed:

src/main/webapp/resources/templates/mobile/ticket-entries.html

```
<% if (ticketPrices.length > 0) { %>
    <form name="ticketCategories">
    <h4>Select tickets by category</h4>
    <% _.each(ticketPrices, function(ticketPrice) { %>
      <div id="ticket-category-input-<%=ticketPrice.id%>"/>

      <fieldset data-role="fieldcontain">
         <label for="ticket-<%=ticketPrice.id%>"><%=ticketPrice.ticketCategory.description% ↩
             >($<%=ticketPrice.price%>)</label>
        <input id="ticket-<%=ticketPrice.id%>" data-tm-id="<%=ticketPrice.id%>" type=" ↩
            number" placeholder="Enter value"
               name="tickets"/>
      </fieldset>
   <% }) %>
   </form>
<% } %>
```

Before submitting the request to the server, the order is confirmed:

src/main/webapp/resources/templates/mobile/confirm-booking.html

```
<div data-role="header">
    <h1>Confirm order</h1>
</div>
<div data-role="content">
    <h3><%=show.event.name%></h3>
    <p><%=show.venue.name%></p>
    <p><small><%=new Date(performance.date).toPrettyString()%></small></p>
    <p><strong>Buyer:</strong>  <emphasis><%=email%></emphasis></p>
    <div id="ticketSummaryView"/>

</div>

<div data-role="footer" class="ui-bar">
    <div class="ui-grid-b">
        <div class="ui-block-a"><a id="cancel" href="#" data-role="button" data-icon=" ↩
            delete">Cancel</a></div>
        <div class="ui-block-b"><a id="goBack" data-role="button" data-icon="back">Back</a> ↩
            </div>
        <div class="ui-block-c"><a id="saveBooking" data-icon="check" data-role="button"> ↩
            Buy!</a></div>
    </div>
</div>
```

The confirmation page contains a summary subview:

src/main/webapp/resources/templates/mobile/ticket-summary-view.html

```
<table>
    <thead>
    <tr>
        <th>Section</th>
        <th>Category</th>
        <th>Price</th>
        <th>Quantity</th>
    </tr>
    </thead>
    <tbody>
    <% _.each(tickets, function(ticketRequest) { %>
    <% _.each(ticketRequest, function(model) { %>
    <% if (model.quantity != undefined) { %>
    <tr>
        <td><%= model.ticketPrice.section.name %></td>
        <td><%= model.ticketPrice.ticketCategory.description %></td>
        <td>$<%= model.ticketPrice.price %></td>
        <td><%= model.quantity %></td>
    </tr>
    <% } %>
    <% }) %>
    <% }) %>
    </tbody>
</table>
<div data-theme="c">
    <h4>Totals</h4>
    <p><strong>Total tickets: </strong><%= totals.tickets %></p>
    <p> <strong>Total price: $</strong><%= totals.price %></p>
</div>
```

Finally, we create the page that displays the booking confirmation:

src/main/webapp/resources/templates/mobile/booking-details.html

```
<div data-role="header">
    <h1>Booking complete</h1>
</div>
<div data-role="content">
    <table id="confirm_tbl">
        <thead>
        <tr>
            <td colspan="5" align="center"><strong>Booking <%=id%></strong></td>
        <tr>
        <tr>
            <th>Ticket #</th>
            <th>Category</th>
            <th>Section</th>
            <th>Row</th>
            <th>Seat</th>
        </tr>
        </thead>
        <tbody>
        <% $.each(_.sortBy(tickets, function(ticket) {return ticket.id}), function (i,  ↩
           ticket) { %>
        <tr>
            <td><%= ticket.id %></td>
            <td><%=ticket.ticketCategory.description%></td>
            <td><%=ticket.seat.section.name%></td>
            <td><%=ticket.seat.rowNumber%></td>
            <td><%=ticket.seat.number%></td>
        </tr>
        <% }) %>
        </tbody>
    </table></div>
<div data-role="footer" class="ui-bar">

    <div class="ui-block-b"><a id="back" href="#" data-role="button" data-icon="back">Back< ↩
        /a></div>

</div>
```

The last step is registering the view with the router:

src/main/webapp/resources/js/app/router/desktop/router.js

```
/**
 * A module for the router of the desktop application
 */
define("router", [
        ...
    'app/views/mobile/create-booking',
    ...
],function (
                        ...
            CreateBookingView
            ...) {

    var Router = Backbone.Router.extend({
        routes:{
            ...
            "book/:showId/:performanceId":"bookTickets",
            ...
        },
        ...
        bookTickets:function (showId, performanceId) {
            var createBookingView =
```

```
                new CreateBookingView(
                      { model: {
                            showId:showId,
                            performanceId:performanceId,
                            bookingRequest:{tickets:[]}},
                            el:$("#container")
                      });
            utilities.viewManager.showView(createBookingView);
        },
        ...
        );
});
```

Chapter 32

More Resources

To learn more about writing HTML5 + REST applications with JBoss, take a look at the Aerogear project.

Part V

Building the Administration UI using Forge

Chapter 33

What Will You Learn Here?

You've just defined the domain model of your application, and all the entities managed directly by the end-users. Now it's time to build an administration GUI for the TicketMonster application using JSF and RichFaces. After reading this guide, you'll understand how to use JBoss Forge to create the views from the entities and how to "soup up" the UI using RichFaces.

We'll round out the guide by revealing the required, yet short and sweet, configuration.

The tutorial will show you how to perform all these steps in JBoss Developer Studio, including screenshots that guide you through. For those of you who prefer to watch and learn, the included video shows you how we performed all the steps.

Chapter 34

Setting up Forge

34.1 JBoss Enterprise Application Platform 6

If you are using JBoss Enterprise Application Platform 6, Forge is available in JBoss Developer Studio 5 (Beta1 or newer).

To show the Forge Console, navigate to *Window* → *Show View* → *Other*, locate *Forge Console* and click *OK*. Then click the *Start* button in top right corner of the view.

34.2 JBoss AS 7

If you are using JBoss AS 7, you should install JBoss Forge version 1.0.2.Final or higher. Follow the instructions at Installing Forge.

Open a command line and navigate to the root directory of this quickstart.

Launch Forge by typing the following command:

```
forge
```

34.3 Required Forge Plugins

Forge comes with a number of built in plugins, including the "scaffold" plugin, which is able to generate a full CRUD UI from JPA entities. The generated UI uses JSF as the view layer, backed by CDI beans. Internally, Forge uses Metawidget to create the CRUD screens.

Forge also includes a powerful plugin management system. The RichFaces plugin isn't bundled with Forge, but it's easy to install. First use the `forge find-plugin` command to locate it

```
forge find-plugin richfaces
```

In this case, the plugin is just called `richfaces` - easy! We can install it using the `forge install-plugin` command:

```
forge install-plugin richfaces
```

This will download, compile and install the RichFaces plugin.

Chapter 35

Getting started with Forge

Forge is a powerful rapid application development (aimed at Java EE 6) and project comprehension tool. It can operate both on projects it creates, and on existing projects, such as TicketMonster. If you want to learn more about Forge ...

When you `cd` into a project with Forge, it inspects the project, and detects what technologies you are using in the project. Let's see this in action:

```
project list-facets
```

Those facets detected are colored green.

```
[ticket-monster] ticket-monster-1 $ project list-facets
forge.spec.jaxrs.webxml
forge.spec.jms
forge.spec.jpa
forge.maven.JavaSourceFacet
forge.spec.jsf
forge.maven.ResourceFacet
forge.spec.validation
forge.spec.cdi
forge.maven.MavenCoreFacet
forge.maven.MavenDependencyFacet
forge.spec.jta
forge.spec.servlet
forge.spec.jstl
forge.maven.MetadataFacet
forge.spec.jaxrs
forge.spec.ejb
forge.spec.jaxrs.applicationclass
org.richfaces
forge.maven.MavenPluginFacet
faces
forge.api
forge.spec.jaxws
forge.spec.jsf.api
forge.maven.PackagingFacet
forge.maven.JavaExecutionFacet
forge.maven.WebResourceFacet
[ticket-monster] ticket-monster-1 $
```

Figure 35.1: Output of `project list-facets`

As you can see, Forge has detected all the technologies we are using, such as JPA, JAX-RS, CDI and Bean Validation.

Chapter 36

Generating the CRUD UI

Forge Scripts

Forge supports the execution of scripts. The generation of the CRUD UI is provided as a Forge script in TicketMonster, so you don't need to type the commands everytime you want to regenerate the Admin UI. The script will also prompt you to applyTo run the script:

```
run admin_layer.fsh
```

36.1 Update the project

First, we need to add Scaffold to the project. Run:

```
scaffold setup --targetDir admin
```

to instruct Forge to generate the css, images and templates used by the scaffolded UI. Forge also adds an error page to be used when a 404 or a 500 error is encountered.

```
[ticket-monster] ticket-monster-1 $ scaffold setup --targetDir admin
***INFO*** Using currently installed scaffold [faces]
Warning:  The encoding 'UTF-8' is not supported by the Java runtime.
Warning:  The encoding 'UTF-8' is not supported by the Java runtime.
Warning:  The encoding 'UTF-8' is not supported by the Java runtime.
Warning:  The encoding 'UTF-8' is not supported by the Java runtime.
Warning:  The encoding 'UTF-8' is not supported by the Java runtime.
Warning:  The encoding 'UTF-8' is not supported by the Java runtime.
Wrote /Users/pmuir/workspace/ticket-monster-1/src/main/webapp/resources/scaffold/paginator.xh
Wrote /Users/pmuir/workspace/ticket-monster-1/src/main/webapp/resources/scaffold/pageTemplate
Wrote /Users/pmuir/workspace/ticket-monster-1/src/main/webapp/admin/index.html
Wrote /Users/pmuir/workspace/ticket-monster-1/src/main/webapp/admin/index.xhtml
Wrote /Users/pmuir/workspace/ticket-monster-1/src/main/webapp/error.xhtml
Wrote /Users/pmuir/workspace/ticket-monster-1/src/main/webapp/resources/add.png
Wrote /Users/pmuir/workspace/ticket-monster-1/src/main/webapp/resources/background.gif
Wrote /Users/pmuir/workspace/ticket-monster-1/src/main/webapp/resources/false.png
Wrote /Users/pmuir/workspace/ticket-monster-1/src/main/webapp/resources/favicon.ico
Wrote /Users/pmuir/workspace/ticket-monster-1/src/main/webapp/resources/forge-logo.png
Wrote /Users/pmuir/workspace/ticket-monster-1/src/main/webapp/resources/forge-style.css
Wrote /Users/pmuir/workspace/ticket-monster-1/src/main/webapp/resources/jboss-community.png
Wrote /Users/pmuir/workspace/ticket-monster-1/src/main/webapp/resources/remove.png
Wrote /Users/pmuir/workspace/ticket-monster-1/src/main/webapp/resources/search.png
Wrote /Users/pmuir/workspace/ticket-monster-1/src/main/webapp/resources/true.png
Wrote /Users/pmuir/workspace/ticket-monster-1/src/main/webapp/WEB-INF/web.xml
```

Figure 36.1: Output of `scaffold setup`

Now, we need to add RichFaces to the project. Run:

```
richfaces setup
```

You'll be prompted for the version of RichFaces to use. Choose version `4.0.0.Final` (the default), by pressing **Enter**.

```
[ticket-monster] ticket-monster-1 $ richfaces setup

Which version of RichFaces?

  1 - [RichFaces 4.0.0.Final]*
  2 - [RichFaces 3.3.3.Final]

 ? Choose an option by typing the number of the selection [*-default]  [0]
Warning:  The encoding 'UTF-8' is not supported by the Java runtime.
Warning:  The encoding 'UTF-8' is not supported by the Java runtime.
***SUCCESS*** Installed [org.richfaces] successfully.
***SUCCESS*** RichFacesFacet is configured.
Wrote /Users/pmuir/workspace/ticket-monster-1/pom.xml
Wrote /Users/pmuir/workspace/ticket-monster-1/src/main/webapp/WEB-INF/web.xml
```

Figure 36.2: Output of `richfaces setup`

36.2 Scaffold the view from the JPA entities

You can either scaffold the entities one-by-one, which allows to control which UIs are generated, or you can generate a CRUD UI for all the entities. We'll do the latter:

```
scaffold from-entity org.jboss.jdf.example.ticketmonster.model.* --targetDir admin ↩
    --overwrite
```

> Forge asks us whether we want to overwrite every file - which get's a bit tedious! Specifying `--overwrite` allows Forge to overwrite files without prompt - much better!

We now have a CRUD UI for all the entities used in TicketMonster!

Chapter 37

Test the CRUD UI

Let's test our UI on our local JBoss AS instance. As usual, we'll build and deploy using Maven:

```
mvn clean package jboss-as:deploy
```

Chapter 38

Make some changes to the UI

Let's add support for images to the Admin UI. TicketMonster doesn't provide support for storing images, but allows you to reference images from hosting sites on the internet. TicketMonster caches the images, so you can still use the application when you aren't connected to the internet.

We'll use JSF 2's composite components, which allow to easily create new components.

/src/main/webapp/resources/tm/image.xhtml

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:composite="http://java.sun.com/jsf/composite">
<head>
<title>Cached Image</title>
</head>
<body>

<composite:interface>
    <composite:attribute name="media" type="org.jboss.jdf.example.ticketmonster.services. ↩
        MediaPath"/>
    <composite:attribute name="id" type="java.lang.String" />
</composite:interface>

<composite:implementation>
    <h:graphicImage value="#{cc.attrs.media.url}" rendered="#{!cc.attrs.media.cached}"/>
    <h:graphicImage value="/rest/media/cache/#{cc.attrs.media.url}" rendered="#{cc.attrs. ↩
        media.cached}"/>
</composite:implementation>

</body>
</html>
```

The `image` composite component encapsulates the rendering of the image, pulling it from the remote location if the item is available and not cached, or pulling it from the cache if otherwise.

Adding this file to `/src/main/webapp/resources/tm/` automatically registers the component with JSF, using the namespace `xmlns:tm="http://java.sun.com/jsf/composite/tm`.

Let's go ahead and use this component to display the image in `src/main/webapp/admin/event/view.xhtml` - the page an admin uses to view an event before editing it. Open up the file in JBoss Developer Studio (or your favourite IDE or text editor). Forge has generated an entry in panel grid to display the image URL, so we can just add `<tm:image media="#{mediaManager.getPath(eventBean.event.picture)}" />` to the `<h:link>` with the id `eventBeanEv`
We need to register the namespace as well, so add `xmlns:tm="http://java.sun.com/jsf/composite/tm"` to the `<ui:composition>` tag. You should end up with a file that looks a bit like:

/src/main/webapp/admin/event/view.xhtml

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1 ↩
    /DTD/xhtml1-transitional.dtd">
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:tm="http://java.sun.com/jsf/composite/tm"
    template="/resources/scaffold/pageTemplate.xhtml">

    <f:metadata>
        <f:viewParam name="id" value="#{eventBean.id}" />
        <f:event type="preRenderView" listener="#{eventBean.retrieve}" />
    </f:metadata>

    <ui:param name="pageTitle" value="View Event" />

    <ui:define name="header">
                Event
        </ui:define>

    <ui:define name="subheader">
                View existing Event
        </ui:define>

    <ui:define name="footer" />

    <ui:define name="main">
        <h:panelGrid columnClasses="label,component,required"
            columns="3">
            <h:outputLabel for="eventBeanEventName" value="Name:" />
            <h:outputText id="eventBeanEventName"
                value="#{eventBean.event.name}" />
            <h:outputText />
            <h:outputLabel for="eventBeanEventPicture" value="Picture:" />
            <h:link id="eventBeanEventPicture"
                outcome="/admin/mediaItem/view"
                value="#{eventBean.event.picture}">
                <tm:image
                    media="#{mediaManager.getPath(eventBean.event.picture)}" />
                <f:param name="id" value="#{eventBean.event.picture.id}" />
            </h:link>
            <h:outputText />
            <h:outputLabel for="eventBeanEventCategory"
                value="Category:" />
            <h:link id="eventBeanEventCategory"
                outcome="/admin/eventCategory/view"
                value="#{eventBean.event.category}">
                <f:param name="id"
                    value="#{eventBean.event.category.id}" />

            </h:link>
            <h:outputText />
            <h:outputLabel for="eventBeanEventDescription"
                value="Description:" />
            <h:outputText id="eventBeanEventDescription"
                value="#{eventBean.event.description}" />
            <h:outputText />
            <h:outputLabel value="Major:" />
            <h:outputText
```

```
                styleClass="#{eventBean.event.major ? 'boolean-true' : 'boolean-false'}" />
            <h:outputText />
        </h:panelGrid>

        <div class="buttons">
            <h:link value="View All" outcome="search" />
            <h:link value="Edit" outcome="create"
                includeViewParams="true" />
            <h:link value="Create New" outcome="create" />
        </div>
    </ui:define>

</ui:composition>
```

We can test these changes by running

```
mvn clean package jboss-as:deploy
```

as usual.

Part VI

Building The Statistics Dashboard Using GWT And Errai

Chapter 39

What Will You Learn Here?

You've just built the administration view, and would like to collect real-time information about ticket sales and attendance. Now it would be good to implement a dashboard that can collect data and receive real-time updates. After reading this tutorial, you will understand our dashboard design and the choices that we made in its implementation. Topics covered include:

- Adding GWT to your application
- Setting up CDI server-client eventing using Errai
- Testing GWT applications

The tutorial will show you how to perform all these steps in JBoss Developer Studio, including screenshots that guide you through. For those of you who prefer to watch and learn, the included video shows you how we performed all the steps.

In this tutorial, we will create a booking monitor using Errai and GWT, and add it to the TicketMonster application. It will show live updates on the booking status of all performances and shows. These live updates are powered by CDI events crossing the client-server boundary, a feature provided by the Errai Framework.

39.1 Before we start

Let us quickly review the starting point of this chapter. If you are re-creating TicketMonster as part of reading this tutorial, this is a good time to check that all the prerequisites are in place. If you are not re-creating TicketMonster on your own, then you can skip this section.

Before everything, make sure that you have read and created the code described in chapter Part III.

Afterwards, make sure that Errai is properly configured in the application.

First, we check if `pom.xml` contains a reference to the Bill Of Materials (BOM) that describes the correct version for the Errai artifacts. Make sure that you have the following in the `dependencyManagement` section:

pom.xml

```
<project ...>
  ...
  <dependencyManagement>
    <dependencies>
      ...
      <dependency>
         <groupId>org.jboss.bom</groupId>
         <artifactId>jboss-javaee-6.0-with-errai</artifactId>
         <version>${jboss.bom.version}</version>
         <type>pom</type>
         <scope>import</scope>
```

```
      </dependency>
    </dependencies>
  </dependencyManagement>
</project>
```

Next, we check if the GWT and Errai artifacts are included in the project. .pom.xml

```
<project ...>
  ...
  <dependencies>

    <!-- The next set of dependencies are for Errai, which we use for
           the TicketMonster booking monitor -->
   <dependency>
       <groupId>org.jboss.errai</groupId>
       <artifactId>errai-bus</artifactId>
       <exclusions>
           <exclusion>
               <groupId>javax.inject</groupId>
               <artifactId>javax.inject</artifactId>
           </exclusion>
           <exclusion>
               <groupId>javax.annotation</groupId>
               <artifactId>jsr250-api</artifactId>
           </exclusion>
       </exclusions>
   </dependency>
   <dependency>
       <groupId>org.jboss.errai</groupId>
       <artifactId>errai-ioc</artifactId>
       <exclusions>
           <exclusion>
               <groupId>javax.inject</groupId>
               <artifactId>javax.inject</artifactId>
           </exclusion>
           <exclusion>
               <groupId>javax.annotation</groupId>
               <artifactId>jsr250-api</artifactId>
           </exclusion>
       </exclusions>
   </dependency>
   <dependency>
       <groupId>org.jboss.errai</groupId>
       <artifactId>errai-tools</artifactId>
   </dependency>
   <dependency>
       <groupId>org.mvel</groupId>
       <artifactId>mvel2</artifactId>
   </dependency>

   <!-- CDI/ Errai Integration Modules -->
   <dependency>
       <groupId>org.jboss.errai</groupId>
       <artifactId>errai-cdi-client</artifactId>
   </dependency>

   <dependency>
       <groupId>org.jboss.errai</groupId>
       <artifactId>errai-javax-enterprise</artifactId>
       <scope>provided</scope>
   </dependency>
```

```
   <dependency>
    <groupId>org.jboss.errai</groupId>
    <artifactId>errai-weld-integration</artifactId>
    <exclusions>
      <exclusion>
        <groupId>org.jboss.weld.servlet</groupId>
        <artifactId>weld-servlet</artifactId>
      </exclusion>
    </exclusions>
  </dependency>

  <dependency>
    <groupId>com.google.gwt</groupId>
    <artifactId>gwt-user</artifactId>
    <scope>provided</scope>
  </dependency>

    <dependency>
      <groupId>com.google.gwt</groupId>
      <artifactId>gwt-dev</artifactId>
      <scope>provided</scope>
    </dependency>
  </dependencies>

  ...
</project>
```

Make sure that the appropriate Maven plugins are configured too, and your build configuration contains the following:

pom.xml

```
<build>
    <!-- Maven will append the version to the finalName (which is the
   name given to the generated war, and hence the context root) -->
    <finalName>${project.artifactId}</finalName>
    <pluginManagement>

        <plugins>
            <!-- Compiler plugin enforces Java 1.6 compatibility and activates
          annotation processors -->
            <plugin>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>2.3.1</version>
                <configuration>
                    <source>1.6</source>
                    <target>1.6</target>
                </configuration>
            </plugin>
            <plugin>
                <artifactId>maven-war-plugin</artifactId>
                <version>2.1.1</version>
                <configuration>
                    <!-- We must exclude GWT client local classes from the
               deployment, or classpath scanners such as Hibernate and Weld get confused
               when the webapp is bootstrapping. -->
                    <packagingExcludes>**/javax/**/*.*,**/client/local/**/*.class</ ↩
                        packagingExcludes>
                    <archive>
                        <manifestEntries>
                            <Dependencies>org.jboss.as.naming,org.jboss.as.server,org.jboss ↩
                                .msc</Dependencies>
                        </manifestEntries>
```

```
                </archive>
            </configuration>
        </plugin>

        <!-- The JBoss AS plugin deploys your war to a local JBoss AS container -->
        <!-- To use run: mvn package jboss-as:deploy -->
        <plugin>
            <groupId>org.jboss.as.plugins</groupId>
            <artifactId>jboss-as-maven-plugin</artifactId>
            <version>7.1.1.Final</version>
        </plugin>

        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-clean-plugin</artifactId>
            <version>2.4.1</version>
            <configuration>
                <filesets>
                    <fileset>
                        <directory>.errai</directory>
                        <includes>
                            <include>**</include>
                        </includes>
                    </fileset>
                </filesets>
            </configuration>
        </plugin>
        <!-- m2e (Maven integration for Eclipse) requires the following
            configuration -->
        <plugin>
            <groupId>org.eclipse.m2e</groupId>
            <artifactId>lifecycle-mapping</artifactId>
            <version>1.0.0</version>
            <configuration>
                <lifecycleMappingMetadata>
                    <pluginExecutions>
                        <pluginExecution>
                            <pluginExecutionFilter>
                                <groupId>org.codehaus.mojo</groupId>
                                <artifactId>gwt-maven-plugin</artifactId>
                                <versionRange>[2.3.0,)</versionRange>
                                <goals>
                                    <goal>resources</goal>
                                </goals>
                            </pluginExecutionFilter>
                            <action>
                                <execute/>
                            </action>
                        </pluginExecution>
                    </pluginExecutions>
                </lifecycleMappingMetadata>
            </configuration>
        </plugin>
    </plugins>
</pluginManagement>
<plugins>
    <!-- GWT plugin to compile client-side java code to javascript
    and to run GWT development mode -->
    <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>gwt-maven-plugin</artifactId>
        <version>2.4.0</version>
```

```
            <configuration>
                <inplace>true</inplace>
                <logLevel>INFO</logLevel>
                <extraJvmArgs>-Xmx512m</extraJvmArgs>
                <draftCompile>true</draftCompile>
                <!-- Configure GWT's development mode (formerly known
                as hosted mode) to not start the default server (embedded jetty), but to
                download the HTML host page from the configured runTarget. -->
                <noServer>true</noServer>
                <runTarget>http://localhost:8080/ticket-monster/booking-monitor.html</ ↩
                    runTarget>
            </configuration>
            <executions>
                <execution>
                    <goals>
                        <goal>resources</goal>
                        <goal>compile</goal>
                    </goals>
                </execution>
                <execution>
                    <id>gwt-clean</id>
                    <phase>clean</phase>
                    <goals>
                        <goal>clean</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

If one or more of the above is not true, please make the appropriate changes.

Chapter 40

Module definition

The first step is to add a GWT module descriptor (a `.gwt.xml` file) which defines the GWT module, its dependencies and configures the client source paths. Only classes in these source paths will be compiled to JavaScript by the GWT compiler. Here's the `BookingMonitor.gwt.xml` file:

src/main/resources/org/jboss/jdf/example/ticketmonster/BookingMonitor.gwt.xml

```
<!DOCTYPE module PUBLIC "-//Google Inc.//DTD Google Web Toolkit 1.6//EN"
        "http://google-web-toolkit.googlecode.com/svn/releases/1.6/distro-source/core/src/ ↩
            gwt-module.dtd">

<!--
   This file declares the Errai/GWT module for the TicketMonster booking monitor,
   which shares the model classes with the user-facing part of the app, but defines
   its own user interface for TicketMonster administrators.
-->

<module rename-to="BookingMonitor">
    <inherits name="org.jboss.errai.common.ErraiCommon"/>
    <inherits name="org.jboss.errai.bus.ErraiBus"/>
    <inherits name="org.jboss.errai.ioc.Container"/>
    <inherits name="org.jboss.errai.enterprise.CDI"/>

    <!-- Model classes that are shared with the rest of the application -->
    <source path="model"/>

    <!-- Classes that are specific to 'booking monitor' features; not shared with rest of  ↩
        app -->
    <source path="monitor"/>

    <!-- Limit the supported browsers for the sake of this demo -->
    <set-property name="user.agent" value="ie8,safari,gecko1_8"/>
</module>
```

The `rename-to` attribute specifies the output directory and file name of the resulting JavaScript file. In this case we specified that the `BookingMonitor` module will be compiled into `BookingMonitor/BookingMonitor.nocache.js` in the project's output directory. The module further inherits the required Errai modules, and specifies the already existing `model` package as source path, as well as a new package named `monitor`, which will contain all the client source code specific to the booking monitor.

Chapter 41

Host page

In the next step we add a *host HTML page* which includes the generated JavaScript and all required CSS files for the booking monitor. It further specifies a `<div>` element with id `content` which will be used as a container for the booking monitor's user interface.

src/main/webapp/booking-monitor.html

```
<!DOCTYPE html>
<html>
<head>
    <title>Ticket Monster Administration</title>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

    <link rel="stylesheet" href="resources/bootstrap/css/bootstrap.css" />
    <link type="text/css" rel="stylesheet" href="resources/css/screen.css" />

    <script type="text/javascript" src="BookingMonitor/BookingMonitor.nocache.js"></script>
</head>

<body>
    <div id="container">
        <div id="menu">
            <div class="navbar">
                <div class="navbar-inner">
                    <div class="container">
                        <a class="brand">JBoss Ticket Monster Booking Monitor</a>
                    </div>
                </div>
            </div>
        </div>
        <h3 class="booking-status-header">Booking status</h3>
        <div id="content" class="container-fluid"></div>
    </div>

    <footer>
        <div style="text-align: center;">
            <img src="resources/img/logo.png" alt="Errai" />
        </div>
    </footer>
</body>
</html>
```

Chapter 42

Enabling Errai

For enabling Errai in our application we will add an `ErraiApp.properties` marker file. When it is detected inside a JAR or at the top of any classpath, the subdirectories are scanned for deployable components. As such, all Errai application modules in a project must contain an `ErraiApp.properties` at the root of all classpaths that you wish to be scanned, in this case `src/main/resources`.

We will also add an `ErraiService.properties` file, which contains basic configuration for the bus itself. Unlike `ErraiApp.pro` there should be at most one ErraiService.properties file on the classpath of a deployed application.

src/main/resources/ErraiService.properties

```
#
# Request dispatcher implementation (default is SimpleDispatcher)
#
errai.dispatcher_implementation=org.jboss.errai.bus.server.SimpleDispatcher
```

Chapter 43

Preparing the wire objects

One of the strengths of Errai is the ability to use domain objects for communication across the wire. In order for that to be possible, all model classes that are transferred using Errai RPC or Errai CDI need to be annotated with the Errai-specific annotation `@Portable`. We will begin by annotating the `Booking` class which used as an the event payload.

src/main/java/org/jboss/jdf/example/ticketmonster/model/Booking.java

```
...
import org.jboss.errai.common.client.api.annotations.Portable;
...
@Portable
public class Booking implements Serializable {
...
}
```

You should do the same for the other model classes.

Chapter 44

The EntryPoint

We are set up now and ready to start coding. The first class we need is the EntryPoint into the GWT application. Using Errai, all it takes is to create a POJO and annotate it with `@EntryPoint`.

src/main/java/org/jboss/jdf/example/ticketmonster/monitor/client/local/BookingMonitor.java

```
package org.jboss.jdf.example.ticketmonster.monitor.client.local;

import java.util.Collections;
import java.util.Comparator;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import javax.enterprise.event.Observes;
import javax.inject.Inject;

import org.jboss.errai.bus.client.api.RemoteCallback;
import org.jboss.errai.ioc.client.api.AfterInitialization;
import org.jboss.errai.ioc.client.api.Caller;
import org.jboss.errai.ioc.client.api.EntryPoint;
import org.jboss.jdf.example.ticketmonster.monitor.client.shared.BookingMonitorService;
import org.jboss.jdf.example.ticketmonster.monitor.client.shared.qualifier.Cancelled;
import org.jboss.jdf.example.ticketmonster.monitor.client.shared.qualifier.Created;
import org.jboss.jdf.example.ticketmonster.model.Booking;
import org.jboss.jdf.example.ticketmonster.model.Performance;
import org.jboss.jdf.example.ticketmonster.model.Show;

import com.google.gwt.user.client.ui.RootPanel;

/**
 * The entry point into the TicketMonster booking monitor.
 *
 * The {@code @EntryPoint} annotation indicates to the Errai framework that
 * this class should be instantiated inside the web browser when the web page
 * is first loaded.
 */
@EntryPoint
public class BookingMonitor {
    /**
     * This map caches the number of sold tickets for each {@link Performance} using
     * the performance id as key.
     */
    private static Map<Long, Long> occupiedCounts;

    /**
```

```
     * This is the client-side proxy to the {@link BookingMonitorService}.
     * The proxy is generated at build time, and injected into this field when the page ↩
        loads.
     */
    @Inject
    private Caller<BookingMonitorService> monitorService;

    /**
     * We store references to {@link ShowStatusWidget}s in this map, so we can update
     * these widgets when {@link Booking}s are received for the corresponding {@link Show}.
     */
    private Map<Show, ShowStatusWidget> shows = new HashMap<Show, ShowStatusWidget>();

    /**
     * This method constructs the UI.
     *
     * Methods annotated with Errai's {@link AfterInitialization} are only called once
     * everything is up and running, including the communication channel to the server.
     */
    @AfterInitialization
    public void createAndShowUI() {
        // Retrieve the number of sold tickets for each performance.
        monitorService.call(new RemoteCallback<Map<Long, Long>>() {
            @Override
            public void callback(Map<Long, Long> occupiedCounts) {
                BookingMonitor.occupiedCounts = occupiedCounts;
                listShows();
            }
        }).retrieveOccupiedCounts();
    }

    private void listShows() {
        // Retrieve all shows
        monitorService.call(new RemoteCallback<List<Show>>() {
            @Override
            public void callback(List<Show> shows) {
                // Sort based on event name
                Collections.sort(shows, new Comparator<Show>() {
                    @Override
                    public int compare(Show s0, Show s1) {
                        return s0.getEvent().getName().compareTo(s1.getEvent().getName());
                    }
                });

                // Create a show status widget for each show
                for (Show show : shows) {
                    ShowStatusWidget sw = new ShowStatusWidget(show);
                    BookingMonitor.this.shows.put(show, sw);
                    RootPanel.get("content").add(sw);
                }
            }
        }).retrieveShows();
    }

}
```

As soon as Errai has completed its initialization process, the `Booking Monitor#createAndShowUI()` method is invoked (`@AfterInitialization` tells Errai to call it). In this case the method will fetch initial data from the server using Errai RPC and construct the user interface. To carry out the remote procedure call, we use an injected `Caller` for the remote interface `BookingMonitorService` which is part of the `org.jboss.jdf.example.ticketmonster.monitor.client.share` package and whose implementation `BookingMonitorServiceImpl` has been explained in the previous chapter.

In order for the booking status to be updated in real-time, the class must be notified when a change has occured. If you have built the service layer already, you may remember that the JAX-RS `BookingService` class will fire CDI events whenever a booking has been created or cancelled. Now we need to listen to those events.

src/main/java/org/jboss/jdf/example/ticketmonster/monitor/client/local/BookingMonitor.java

```
public class BookingMonitor {

        /**
     * Responds to the CDI event that's fired on the server when a {@link Booking} is ↩
         created.
     *
     * @param booking  the create booking
     */
    public void onNewBooking(@Observes @Created Booking booking) {
        updateBooking(booking, false);
    }

    /**
     * Responds to the CDI event that's fired on the server when a {@link Booking} is ↩
         cancelled.
     *
     * @param booking  the cancelled booking
     */
    public void onCancelledBooking(@Observes @Cancelled Booking booking) {
        updateBooking(booking, true);
    }

    // update the UI widget to reflect the new or cancelled booking
    private void updateBooking(Booking booking, boolean cancellation) {
        ShowStatusWidget sw = shows.get(booking.getPerformance().getShow());
        if (sw != null) {
            long count = getOccupiedCountForPerformance(booking.getPerformance());
            count += (cancellation) ? -booking.getTickets().size() : booking.getTickets(). ↩
                size();

            occupiedCounts.put(booking.getPerformance().getId(), count);
            sw.updatePerformance(booking.getPerformance());
        }
    }

    /**
     * Retrieve the sold ticket count for the given {@link Performance}.
     *
     * @param p  the performance
     * @return number of sold tickets.
     */
    public static long getOccupiedCountForPerformance(Performance p) {
        Long count = occupiedCounts.get(p.getId());
        return (count == null) ? 0 : count.intValue();
    }

}
```

The newly created methods `onNewBooking` and `onCancelledBooking` are *event listeners*. They are identified as such by the `@Observes` annotation applied to their parameters. By using the `@Created` and `@Cancelled` qualifiers that we have defined in our application, we narrow down the range of events that they listen.

Chapter 45

The widgets

Next, we will define the widget classes that are responsible for rendering the user interface. First, we will create the widget class for an individual performance.

src/main/java/org/jboss/jdf/example/ticketmonster/monitor/client/local/PerformanceStatusWidget.java

```
package org.jboss.jdf.example.ticketmonster.monitor.client.local;

import org.jboss.jdf.example.ticketmonster.model.Performance;

import com.google.gwt.i18n.client.DateTimeFormat;
import com.google.gwt.i18n.client.DateTimeFormat.PredefinedFormat;
import com.google.gwt.user.client.ui.Composite;
import com.google.gwt.user.client.ui.HorizontalPanel;
import com.google.gwt.user.client.ui.Label;

/**
 * A UI component to display the status of a {@link Performance}.
 */
public class PerformanceStatusWidget extends Composite {

    private Label bookingStatusLabel = new Label();

    private HorizontalPanel progressBar = new HorizontalPanel();
    private Label soldPercentLabel;
    private Label availablePercentLabel;

    private Performance performance;
    private long soldTickets;
    private int capacity;

    public PerformanceStatusWidget(Performance performance) {
        this.performance = performance;

        soldTickets = BookingMonitor.getOccupiedCountForPerformance(performance);
        capacity = performance.getShow().getVenue().getCapacity();

        setBookingStatus();
        setProgress();

        HorizontalPanel performancePanel = new HorizontalPanel();
        String date = DateTimeFormat.getFormat(PredefinedFormat.DATE_TIME_SHORT).format( ↩
            performance.getDate());
        performancePanel.add(new Label(date));
        performancePanel.add(progressBar);
        performancePanel.add(bookingStatusLabel);
```

```
        performancePanel.setStyleName("performance-status");
        initWidget(performancePanel);
    }

    /**
     * Updates the booking status (progress bar and corresponding text) of the {@link ↩
         Performance}
     * associated with this widget based on the number of sold tickets cached in {@link ↩
         BookingMonitor}.
     */
    public void updateBookingStatus() {
        this.soldTickets = BookingMonitor.getOccupiedCountForPerformance(performance);
        setBookingStatus();
        setProgress();
    }

    private void setBookingStatus() {
        bookingStatusLabel.setText(soldTickets + " of " + capacity + " tickets booked");
    }

    private void setProgress() {
        int soldPercent = Math.round((soldTickets / (float) capacity) * 100);

        if (soldPercentLabel != null) {
            progressBar.remove(soldPercentLabel);
        }

        if (availablePercentLabel != null) {
            progressBar.remove(availablePercentLabel);
        }

        soldPercentLabel = new Label();
        soldPercentLabel.setStyleName("performance-status-progress-sold");
        soldPercentLabel.setWidth(soldPercent + "px");

        availablePercentLabel = new Label();
        availablePercentLabel.setStyleName("performance-status-progress-available");
        availablePercentLabel.setWidth((100 - soldPercent) + "px");

        progressBar.add(soldPercentLabel);
        progressBar.add(availablePercentLabel);
    }
}
```

A show has multiple performances, so we will create a `ShowStatusWidget` to contains a `PerformanceStatusWidget` for each performance.

src/main/java/org/jboss/jdf/example/ticketmonster/monitor/client/local/ShowStatusWidget.java

```
package org.jboss.jdf.example.ticketmonster.monitor.client.local;

import java.util.Date;
import java.util.HashMap;
import java.util.Map;

import org.jboss.jdf.example.ticketmonster.model.Performance;
import org.jboss.jdf.example.ticketmonster.model.Show;

import com.google.gwt.user.client.ui.Composite;
import com.google.gwt.user.client.ui.Label;
import com.google.gwt.user.client.ui.VerticalPanel;
```

```
/**
 * A UI component to display the status of a {@link Show}.
 */
public class ShowStatusWidget extends Composite {

    private Map<Long, PerformanceStatusWidget> performances = new HashMap<Long,  ↩
        PerformanceStatusWidget>();

    public ShowStatusWidget(Show show) {
        VerticalPanel widgetPanel = new VerticalPanel();
        widgetPanel.setStyleName("show-status");

        Label showStatusHeader = new Label(show.getEvent().getName() + " @ " + show. ↩
            getVenue());
        showStatusHeader.setStyleName("show-status-header");
        widgetPanel.add(showStatusHeader);

        // Add a performance status widget for each performance of the show
        for (Performance performance : show.getPerformances()) {
            if (performance.getDate().getTime() > new Date().getTime()) {
                PerformanceStatusWidget psw = new PerformanceStatusWidget(performance);
                performances.put(performance.getId(), psw);
                widgetPanel.add(psw);
            }
        }

        initWidget(widgetPanel);
    }

    /**
     * Triggers an update of the {@link PerformanceStatusWidget} associated with
     * the provided {@link Performance}.
     *
     * @param performance
     */
    public void updatePerformance(Performance performance) {
        PerformanceStatusWidget pw = performances.get(performance.getId());
        if (pw != null) {
            pw.updateBookingStatus();
        }
    }
}
```

This class is has two responsibilities. First, it will to display together all the performances that belong to a given show. Also, it will update its `PerformanceStatusWidget` children whenever a booking event is received on the client (through the observer method defined above).