University of Coimbra

Department of Informatics Engineering

Systems Integration

Report for Project 2

Author:
António Lima
2011166926

Author:
Pedro Janeiro
2012143629

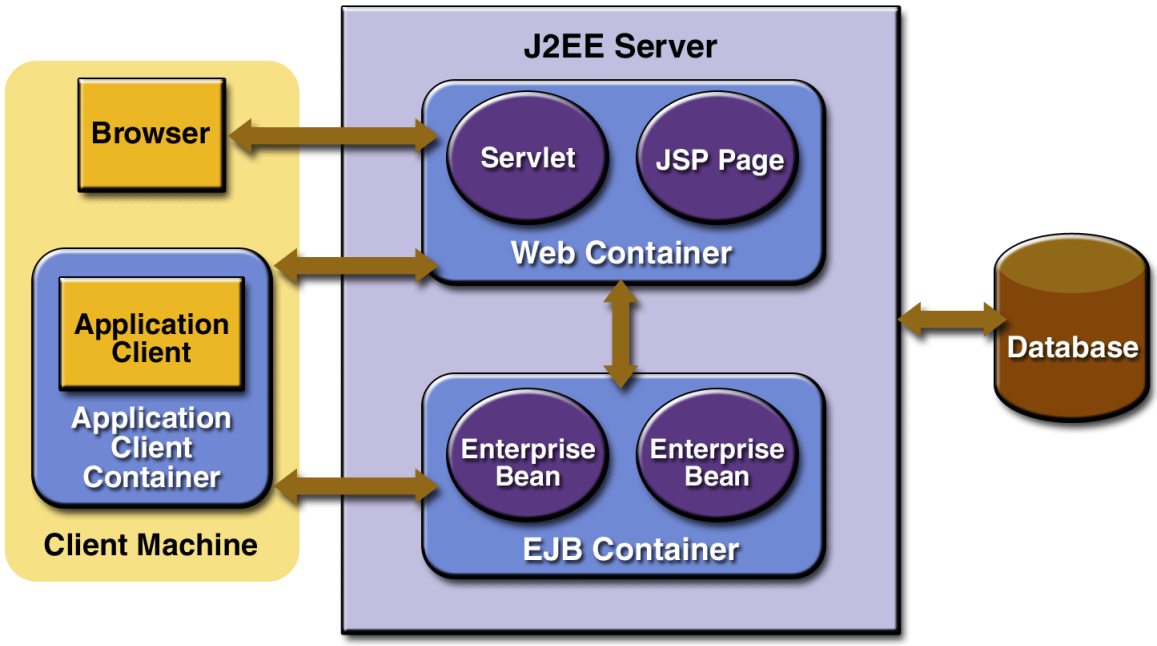
November 13, 2015

Contents

1 Introduction

This assignment is comprised of a single application, with three layers. The first one, the Java Persistence API layer, is responsible for the maintenance of entities, and for its persistence. The Enterprise Java Beans layer encapsulates the business logic of the whole application. And the Servlet layer is responsible for presenting the data to the user, in the form of a website.

The following figure roughly illustrates the previously described scenario.



The three layers are described in greater detail in the following sections.

2 Java Persistence API

We used three classes in this layer:

One for the User related data;

One for the Music related data;

One for the Playlist related data.

To achieve this, the @Entity annotation is necessary, as well as the relational annotations (@ManyToOne, @OneToMany, @OneToOne, @ManyToMany). These establish the connections between entities of the same kind. For example, a User may have uploaded several Music's. Therefore, the User will have List (or other Collection) of Music's, and tag them with the @OneToMany annotation.

On the other hand, using a bidirectional relationship, the Music will have a User field, tagged with the @ManyToMany annotation. The same goes with the relationship between User and Playlist. Between Playlist and Music, though, this type of relationship is suitable. A Music may be present in many Playlist's, and a Playlist will contain many Music's (well, that's kind of the point of a playlist). Thus, either entity has a List with members of the other type (Playlist has a List of Music's, Music has a List of Playlist), and both are tagged with the @ManyToMany annotation.

2.1 User

The User Entity is associated with each client of the application. Therefore, it requires a mean of being associated with a real person in the real world, in this case through the e-mail (which can only be used

by one User, though that is controlled in another layer). There's also a password, to ensure the user's information is kept safe, and the user's name, to better identify him within the application.

2.2 Music

The Music Entity is the core of the application. Users upload musics, users list musics, users create playlists with musics. The music is identified by its title, the name of the artist, the name of the album to which it belongs, and the year the given album was published. There's also a path field, though the application doesn't actually allow the upload of songs, so it's mostly a legacy field.

2.3 Playlist

There's not much about the Playlist Entity. It has a name, a creator, and a list with the songs it contains.

3 Enterprise Java Beans

The EJB layer is the one that makes the connection between the Entities previously mentioned. The Beans used are Stateless, though, so there's not really a reliable connection between a Bean instance and a client, at a given moment.

There are also three types of Beans, as there are three differente Entities, each one with a Remote Interface. There's also a series of Enums that we included, in hopes of simplifying the return values from Bean methods. Unfortunately, some methods require a completely different type of object to work (the User.login() mehod, for example, returns an instantiation of the User class, with the data retrieved during the login operation), so some of the Enums aren't properly used.

To allow for a easier connection to the Entities (and to the underlying database), one can use the @PersistenceContext() tag in conjunction with an EntityManager, to automatically perform some actions that would otherwise be somewhat difficult for a developer to perform. There's no need, for example, for the implicit use of a Transaction, since the annotation used deals with that internally.

3.1 UserBean

The UserBean has a set of methods necessary to perform the operations related to User management, be it registering a new User, logging in, changing User information or even deleting a User.

3.2 MusicBean

As with the UserBean, the MusicBean has the methods necessary for the creation, deletion, and update of a music from the application. There are also some methods that allow listing and searching, necessary for the application to work.

3.3 Playlist

The PlaylistBean follows the same scheme of the other two Beans, with the addition of the methods necessary for the relationship between Playlist and Music.

4 Servlets

We use a single servlet in this application. There are pros and cons for this approach, but we decided to go with it. There are two annotations rather important to note, being them @WebServlet and @EJB. The first one is used to indicate that the given class is, in fact a Servlet, and should be instantiated as one. The second, is used with the Beans mentioned in the previous section. This way, one doesn't have to implicitly instantiate these objets, as this is done internally.

This section of the applications makes use of sixteen .jsp files. These serve as a template for the different views used in the application. For example, the error.jsp file is used with any error occured during runtime, as one must only change the "error" variable in the session to change the displayed message. Other .jsp files with listing functionalities make use of the "forEach" directive (from the jstl library) to traverse to a list of objects, no matter how many elements the list will have.

5 Logging

We user a very simple logging model. We only utilize the INFO and ERROR levels, the former for pretty much every operation available (logging the user we attempted the operation, the songs involved, etc.) and the latter for general errors caused by external problems, namely exceptions.

6 WildFly configuration

As in the first assignment, some configuration has to be done. Firstly, one must create a new Datasource, to allow for the connection to the database. In our case, the datasource created is as follows:

```
<datasource jta="true" jndi-name="java:/SpotifyEAR" pool-name="spotify_pool" enabled="true"
      use-java-context="true" use-ccm="true">
    <connection-url>jdbc:mysql://localhost:3306/Spotify</connection-url>
    <driver>mysql</driver>
    <security>
        <user-name>pjaneiro</user-name>
        <password>|Sisc001</password>
    </security>
    <statement>
        <prepared-statement-cache-size>100</prepared-statement-cache-size>
        <share-prepared-statements>true</share-prepared-statements>
    </statement>
</datasource>
```

This was manually added to the standalone-full.xml file, in the $JBOSS_HOME/standalone/config directory, in the "datasources" subsection.

One must also add a mysql driver (the DBMS we used). To do so, start by downloading the newest version of the MySQL connector from http://dev.mysql.com/downloads/connector/j/. Onde downloaded and extracted, the directory modules/com/mysql/main is created in $JBOSS_HOME. The extracted .jar may be copied to this directory. Then, the following lines must be added to the (at this point, non-existent) $JBOSS_HOME/modules/com/mysql/main/module.xml file:

```
<?xml version="1.0" encoding="UTF-8"?>

<module xmlns="urn:jboss:module:1.0" name="com.mysql">
  <resources>
    <resource-root path="mysql-connector-java-#V.#M.#m-bin.jar"/>
  </resources>
  <dependencies>
    <module name="javax.api"/>
  </dependencies>
</module>
```

The file $JBOSS_HOME/standalone/config/standalone-full.xml must be edited once more. Find the "drivers" section, and add the following line:

```
<driver name="mysql" module="com.mysql"/>
```

At this point, running $JBOSS_HOME/bin/standalone.sh -c standalone-full.xml is enough to run the server.

7 MySQL configuration

The following commands must be executed before running the application for the first time

```
1 create user 'pjaneiro'@'localhost' identified by '|Sisc00l';
2 create database Spotify;
3 grant all privileges on Spotify.* to pjaneiro@localhost;
4 flush privileges;
```

8 Running the application

After starting the server ($JBOSS_HOME/bin/standalone.sh -c standalone-full.xml), the file Spotify-EAR.ear must be copied to the $JBOSS_HOME/standalone/deployments directory. The creaton of a file named Spotify-EAR.ear.deployed in the directory, by the WildFly server, indicates that the application is ready to be used.

To do so, open a web browser and go to http://localhost:8080/Spotify-WEB/Main