Integration between JavaEE and AngularJs (musicManagement-angular)

Davide Lissoni Mat.179878

06/01/2017

1.Introduction:

This assignment is about html5, javascript and in particular AngularJs, an open-source web framework used to develop single-page web applications.
Its goal was to develop a simple application which shows an integration between Java EE (used as REST web service) and AngularJS used in order to develop the user interface (front-end) part.
The project follows the line of the http://www.radcortez.com/java-ee-7-with-angular-js-crud-rest-validations-part-2 application and the functionalities are basically the same.
The application, in broad terms, is structured in order to manage a simple music project management software that allow user to perform all the basic CRUD operations on two entities: “MusicalGenre” and “Song” related together through a One to Many relationship.

2.Implementation:

Thought the project contains both back-end and front-end application, this chapter will be divided in order to make clearer the explanation of their implementations.

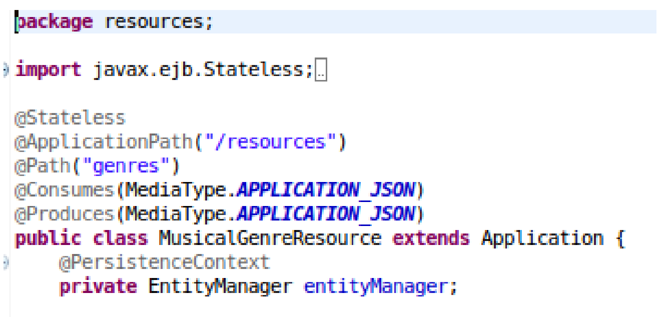
2.1 Java EE service:

The java EE service keeps a REST architecture that allows CRUD operation on the dataset.
The database is composed by two table: MusicalGenre and Song represented by the two java entities mentioned in the “introduction” chapter. Their persistence are managed through Hibernate (the persistence unit is defined in the peristence.xml file).
The server application contains also two resources classes that refers to the respective entities used in order to contain the resources that are exposed through the RESTful Api.

2.1.1 Entities:

The two entities are represented as a normal Java POJO using the Java EE entity annotations.
The structures are the following:

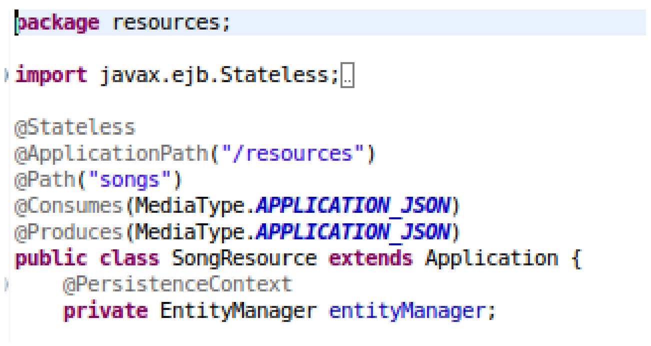
MusicalGenre



```
package resources;

import javax.ejb.Stateless;

@Stateless
@ApplicationPath("/resources")
@Path("genres")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public class MusicalGenreResource extends Application {
    @PersistenceContext
    private EntityManager entityManager;
```

- Long Id (primary key, auto-generated);
- String name;
- String description;
- List <Song>
- Getter and setter methods;

Song:



```
package resources;

import javax.ejb.Stateless;

@Stateless
@ApplicationPath("/resources")
@Path("songs")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public class SongResource extends Application {
    @PersistenceContext
    private EntityManager entityManager;
```

- Long Id (primary key, auto-generated),
- String name;
- String description;
- String youtubeUrl;
- MusicalGenre genre (secondary key).
- Getter and setter methods;

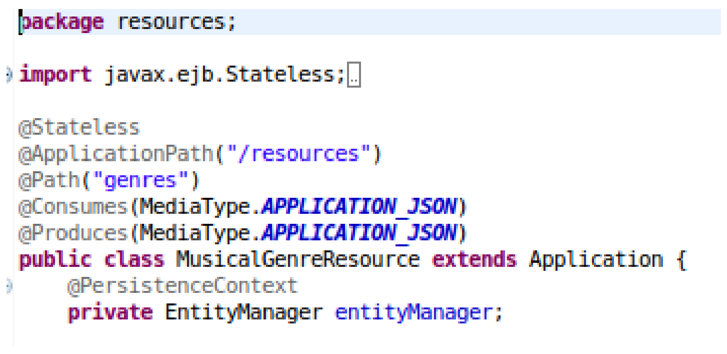
The two entities are referenced each other using a bidirectional One to Many relationship [figura]. This relationship has been chosen for comfort motivations (a many to one unidirectional relationship would be good as well).
The difference between the two relationship is that by using the bidirectional one all the songs related with a genre will be listed in the genre json resource representation, allowing in this way to list in a easier and faster way all the songs related to one genre.

2.1.2 Resources:

For each entity has been implemented its relative resource class.
The resource classes are stateless bean used in order to query the database and expose it as a REST

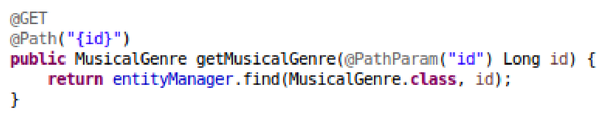
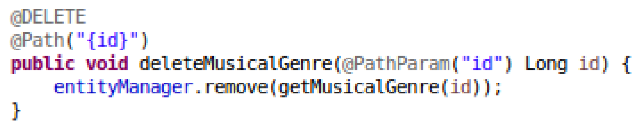
service.
The resource format supported by this service is Json.

MusicalGenreResource.java:

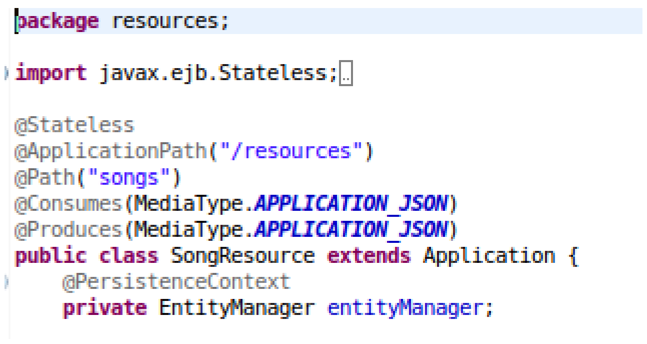


```java
package resources;

import javax.ejb.Stateless;

@Stateless
@ApplicationPath("/resources")
@Path("genres")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public class MusicalGenreResource extends Application {
    @PersistenceContext
    private EntityManager entityManager;
```

This class has been defined as the resource class for the MusiclaGenre entity and expose the RESTful service on the path */resources/genres*.
The CRUD operation implemented on this resource are:

@GET listMusicalGenre	will return the list of genres present in the dataset
@GET @Path("{id}") getMusicalGenre	 `@GET` `@Path("{id}")` `public MusicalGenre getMusicalGenre(@PathParam("id") Long id) {` `return entityManager.find(MusicalGenre.class, id);` `}` will return the genre that corresponds to the id specified in the path (i.e. *resources/genres/id*).
@POST saveMusicalGenre(MusicalGenre genre)	will create(persist) or update(merge) the genre passed in the request body.
@DELETE @Path("{id}")	 `@DELETE` `@Path("{id}")` `public void deleteMusicalGenre(@PathParam("id") Long id) {` `entityManager.remove(getMusicalGenre(id));` `}` will remove from the dataset the genre that correspond to the id specified on the path. Since the behaviour of the relationship between the two entities is “cascade”, also all the songs related with the genre in question will be removed from the dataset.

SongResource.java:

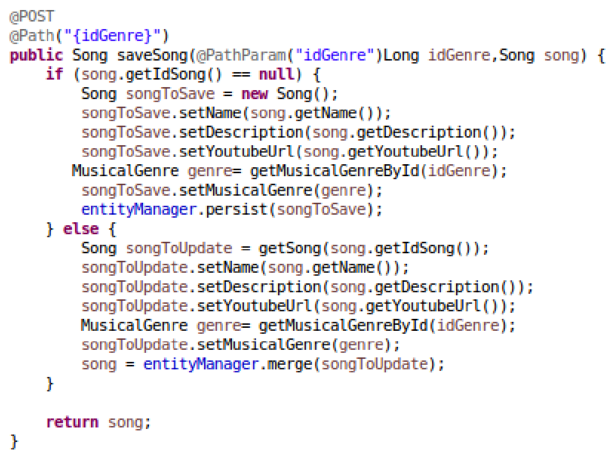


```java
package resources;

import javax.ejb.Stateless;

@Stateless
@ApplicationPath("/resources")
@Path("songs")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public class SongResource extends Application {
    @PersistenceContext
    private EntityManager entityManager;
```

*T*he class has been defined as the resource class for the MusiclaGenre entity and expose the RESTful service on the path */resources/songs*

The CRUD operation implemented on this resource are:

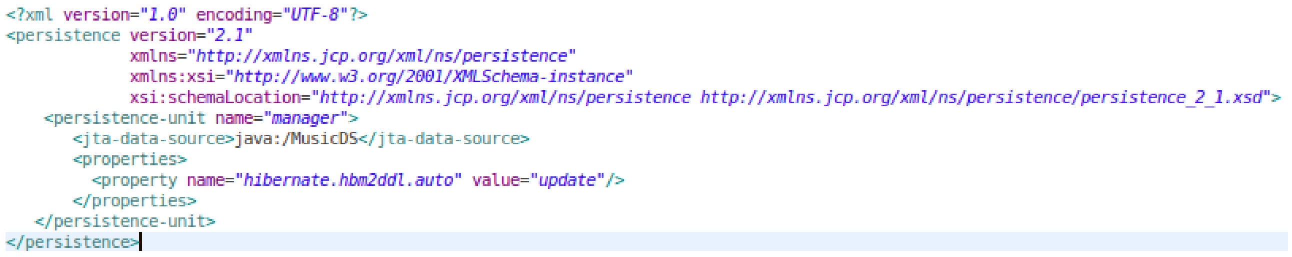
@GET listSongs	will return the list of the songs present in the dataset;
@GET @Path("{id}") getSong	will return the song that corresponds to the id specified in the path (i.e. *resources/song/id*);

@POSTsaveSong(MusicalGenre genre)
@Path("{idGenre}")
	
	`@POST`
	`@Path("{idGenre}")`
	`public Song saveSong(@PathParam("idGenre")Long idGenre,Song song) {`
	`    if (song.getIdSong() == null) {`
	`        Song songToSave = new Song();`
	`        songToSave.setName(song.getName());`
	`        songToSave.setDescription(song.getDescription());`
	`        songToSave.setYoutubeUrl(song.getYoutubeUrl());`
	`       MusicalGenre genre= getMusicalGenreById(idGenre);`
	`        songToSave.setMusicalGenre(genre);`
	`        entityManager.persist(songToSave);`
	`    } else {`
	`        Song songToUpdate = getSong(song.getIdSong());`
	`        songToUpdate.setName(song.getName());`
	`        songToUpdate.setDescription(song.getDescription());`
	`        songToUpdate.setYoutubeUrl(song.getYoutubeUrl());`
	`        MusicalGenre genre= getMusicalGenreById(idGenre);`
	`        songToUpdate.setMusicalGenre(genre);`
	`        song = entityManager.merge(songToUpdate);`
	`    }`
	
	`    return song;`
	`}`
	
	will create(persist) or update(merge) the song passed in the request body, the song will refer to the genre specified in the path;

@DELETE @Path("{id}")	will remove from the dataset the song that correspond to the id specified on the path.

2.1.3 Settings:

The entities are managed by the EntityManager and are defined in a single persistence unit described in the persistence.xml file.



```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
             xmlns="http://xmlns.jcp.org/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
    <persistence-unit name="manager">
       <jta-data-source>java:/MusicDS</jta-data-source>
       <properties>
         <property name="hibernate.hbm2ddl.auto" value="update"/>
       </properties>
   </persistence-unit>
</persistence>
```

Furthermore the project has been implemented as a Maven project in order to deploy it using maven instructions by command line. As every Maven project the pom.xml contains all the information, configuration, details and dependencies used in the application in order to deploy the project, and, since the pom.xml used in the http://www.radcortez.com/java-ee-7-with-angular-js-part-1/ application is really exhaustive and properly done, it has been decided to keep the pom file unchanged.

2.1 Web application:

The web application is the one in charge to make the http requests to the service and print out the results through a user interface.
For the implementations of the application have been used the following technologies:

- **html:** used for the structure of the web application
- **css and bootstrap** : used in order to have a more detailed and beautiful graphic design;
- **JavaScript and AngularJS**: used in order to manage the events and the dynamism of the web page.

2.1.1 Index.html:

The web application is based on a single html page: index.html This is also the default page for the web app shown on the address <web application-address>/musicManagement-angular/
The page is html well formed and the main parts are:

- **header**: it contains the imports used in order to load all the resource needed for the application: angular script-library, bootstrap library, stylesheets and the script that contains the angular/javascript instructions.
  Differently from the project guide the web application doesn't contain any internal library, all the external resource used are included using cdn;

- **body**: the body has been split into 4 sections: the feedback messages sections, the selects section (used to list all the genres and relative songs present in the db), the genre form and the song form, each with its own Angular controller defined in the music.js file. The html contains angular annotations connected with the angular controllers used in order to create a dynamism in the web page.

Since the functionalities of the sections present in the application are different but their lifecycle is

basically the same I'm going to explain in details the GenresCtrl controller in order to better understand how the connection between the html and the framework works.
However the web application make use of all the CRUD operations implemented in the Java EE project.

GenresCtrl section:

This section is in charge to fill two html <select> using the Musical genres and relative songs present in the database.

```
<div  ng-controller="GenresCtrl">
<label>Genre:</label><br/><select  class="form-control" ng-options="genre as genre.name for genre in genres"
    ng-model="selected" ng-change="jsFunction()"></select><br/>

<label>Song:</label><br/><select  class="form-control" ng-options="song as song.name for song in songs.songs"
     ng-model="selectedSong" ng-change="jsFunctionSong()"></select><br/>
</div>
```

The lifecycle of this section could be summarized in the following steps:

1. First of all, at the initialization of the page, the controller makes a http get request on the path resources/genres in order to get all the genres present in the DB;

```
App.controller('GenresCtrl', function ($scope, $rootScope,$http) {
       $http.get('resources/genres').success(function(data) {
           $rootScope.$broadcast('clearForm');
           $rootScope.$broadcast('clearGenreForm');
           $rootScope.$broadcast('disableButtonAlert');
           $rootScope.genres = data;
           if ( $scope.selected==null){
               $rootScope.$broadcast('disableSave');
           }
       }).error(function(){
          $rootScope.$broadcast('error');

       });
```

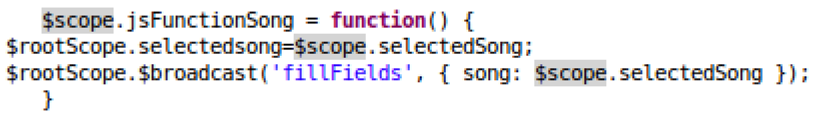
2. Fill the “genre<select>” by using the angular instruction ng-option that will parse the server response data;

3. When a user change the <select> option (ng-change) of the “genre<select>”, it will execute the javascript function jsFunction().
   JsFunction(), defined in the javascript file within the controller, will make a http get request on the path /resources/genres/<id-genres-selected> looking for the songs related to the genre chosen;

```
$scope.jsFunction = function() {
    $http.get('resources/genres/'+$scope.selected.idGenre).success(function(data) {
        $rootScope.songs = data;
        $rootScope.$broadcast('setGenre', { idGenre: $scope.selected.idGenre });
        $rootScope.$broadcast('fillGenreFields', { genre: $scope.selected });
        $rootScope.$broadcast('ableSave');
        $rootScope.$broadcast('clearForm');

   }).error(function(){
       $rootScope.$broadcast('error');

   });

}
```

4. Fill the “song<select>” by using the angular instruction ng-option that will parse the server response data;

5. On the change event of the “song<select>” it will be called the function jsFunctionSong() that will make a http get request on the path /resources/songs/<id-song-selected> looking for the song information contained in the database.



```
   $scope.jsFunctionSong = function() {
$rootScope.selectedsong=$scope.selectedSong;
$rootScope.$broadcast('fillFields', { song: $scope.selectedSong });
   }
```

The controller contains the implementation of other two function: refreshSongList() and refreshGenereList used to refresh the two select list when field is added, deleted or updated.
The controller contains also other function calls that have not been mentioned in the list above. These functions are utilized in order to manage some graphic and utility topics such as, alert message, clear the form when the page is reloaded or enable and disable some buttons in particular situations in order to avoid user errors.

As we can see, the controller will bind the javascript variable required to the html allowing in this way the dynamism of the web page.

3.Deployment:

For this project has been decided to use the same DBMS and JDBC datasource settings used in the previous assignemnts so, the first step is to start the DBMS and to configure the JDBC Datasource in the Wildfly console.
For the development of this project I chose to use derby(version. 10,12,1,1).

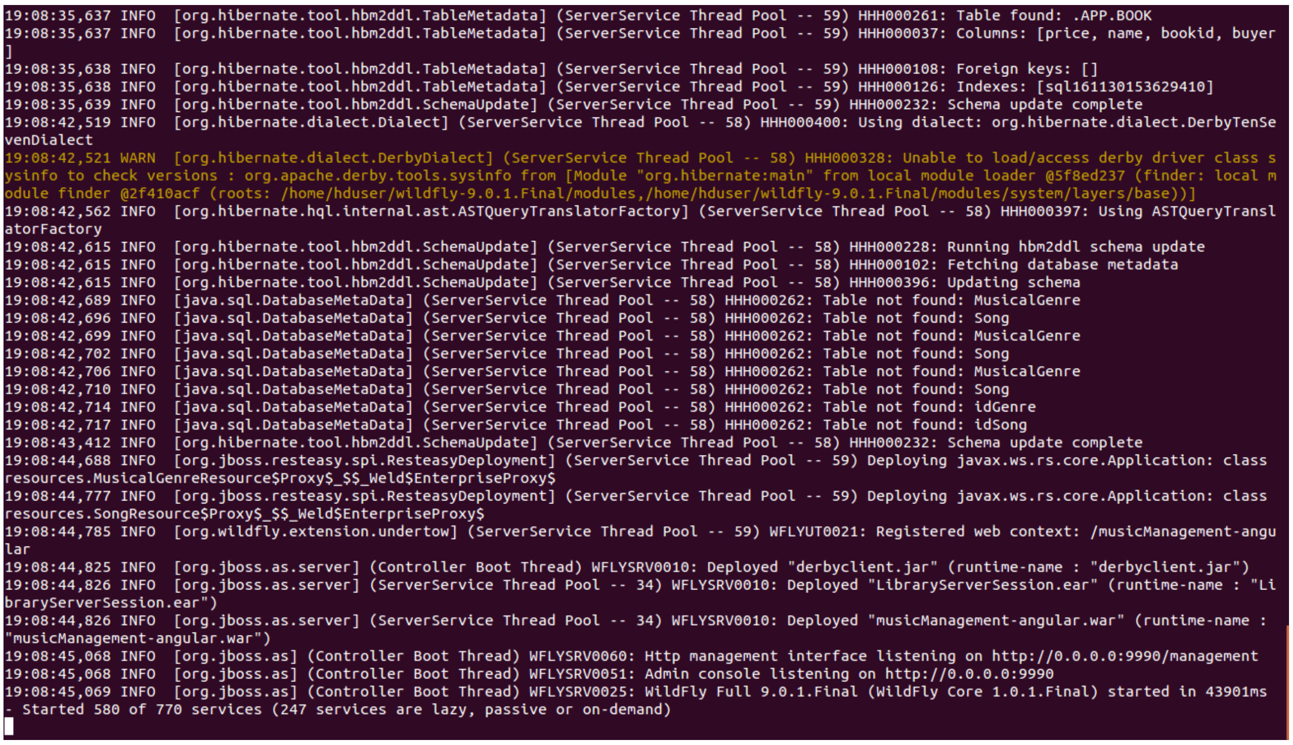
Derby will run on the derby default port 1527 but it is also possible to change it if necessary.
In order to start Derby go in the derby located folder and type the command:

```
java -jar lib/derbyrun.jar server start &
```

In order to configure the JDBC Datasource in Wildfly, open the administrator console and upload the derby JDBC driver (derbyclient.jar) on the deployment tab. After that, add a datasource calling it “MusicDS” adding, as driver, the one just uploaded. If the connection test succeed the Datasource configuration is completed.

The next step is to deploy the musicManagement-angular.war file on Wildfly.
In order to do that, copy the war in $JBOSS_HOME/standalone/deployments folder and than run Wildfly by the command:

```
$JBOSS_HOME/bin/standalone.sh
```



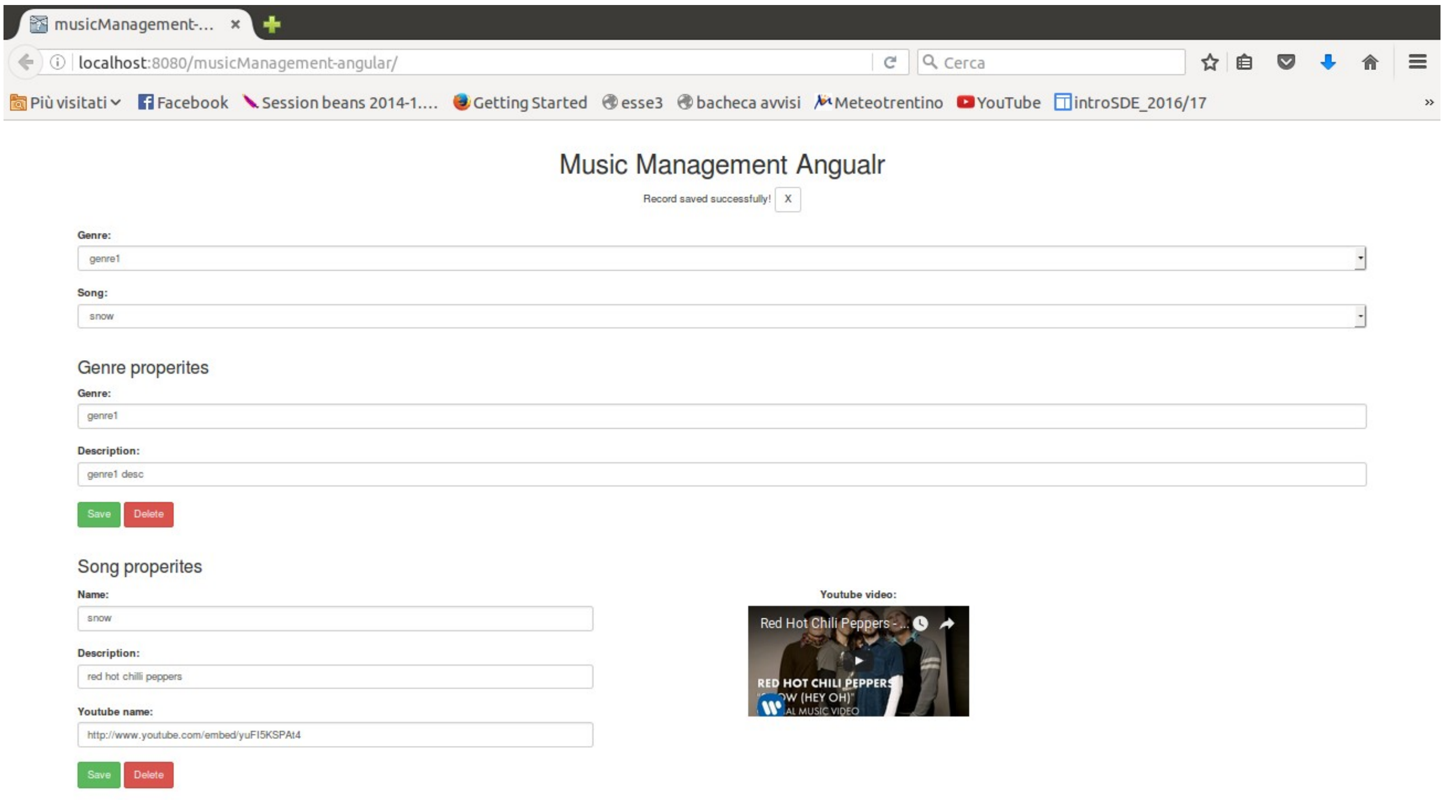
```
19:08:35,637 INFO  [org.hibernate.tool.hbm2ddl.TableMetadata] (ServerService Thread Pool -- 59) HHH000261: Table found: .APP.BOOK
19:08:35,637 INFO  [org.hibernate.tool.hbm2ddl.TableMetadata] (ServerService Thread Pool -- 59) HHH000037: Columns: [price, name, bookid, buyer
]
19:08:35,638 INFO  [org.hibernate.tool.hbm2ddl.TableMetadata] (ServerService Thread Pool -- 59) HHH000108: Foreign keys: []
19:08:35,638 INFO  [org.hibernate.tool.hbm2ddl.TableMetadata] (ServerService Thread Pool -- 59) HHH000126: Indexes: [sql161130153629410]
19:08:35,639 INFO  [org.hibernate.tool.hbm2ddl.SchemaUpdate] (ServerService Thread Pool -- 59) HHH000232: Schema update complete
19:08:42,519 INFO  [org.hibernate.dialect.Dialect] (ServerService Thread Pool -- 58) HHH000400: Using dialect: org.hibernate.dialect.DerbyTenSe
venDialect
19:08:42,521 WARN  [org.hibernate.dialect.DerbyDialect] (ServerService Thread Pool -- 58) HHH000328: Unable to load/access derby driver class s
ysinfo to check versions : org.apache.derby.tools.sysinfo from [Module "org.hibernate:main" from local module loader @5f8ed237 (finder: local m
odule finder @2f410acf (roots: /home/hduser/wildfly-9.0.1.Final/modules,/home/hduser/wildfly-9.0.1.Final/modules/system/layers/base))]
19:08:42,562 INFO  [org.hibernate.hql.internal.ast.ASTQueryTranslatorFactory] (ServerService Thread Pool -- 58) HHH000397: Using ASTQueryTransl
atorFactory
19:08:42,615 INFO  [org.hibernate.tool.hbm2ddl.SchemaUpdate] (ServerService Thread Pool -- 58) HHH000228: Running hbm2ddl schema update
19:08:42,615 INFO  [org.hibernate.tool.hbm2ddl.SchemaUpdate] (ServerService Thread Pool -- 58) HHH000102: Fetching database metadata
19:08:42,615 INFO  [org.hibernate.tool.hbm2ddl.SchemaUpdate] (ServerService Thread Pool -- 58) HHH000396: Updating schema
19:08:42,689 INFO  [java.sql.DatabaseMetaData] (ServerService Thread Pool -- 58) HHH000262: Table not found: MusicalGenre
19:08:42,696 INFO  [java.sql.DatabaseMetaData] (ServerService Thread Pool -- 58) HHH000262: Table not found: Song
19:08:42,699 INFO  [java.sql.DatabaseMetaData] (ServerService Thread Pool -- 58) HHH000262: Table not found: MusicalGenre
19:08:42,702 INFO  [java.sql.DatabaseMetaData] (ServerService Thread Pool -- 58) HHH000262: Table not found: Song
19:08:42,706 INFO  [java.sql.DatabaseMetaData] (ServerService Thread Pool -- 58) HHH000262: Table not found: MusicalGenre
19:08:42,710 INFO  [java.sql.DatabaseMetaData] (ServerService Thread Pool -- 58) HHH000262: Table not found: Song
19:08:42,714 INFO  [java.sql.DatabaseMetaData] (ServerService Thread Pool -- 58) HHH000262: Table not found: idGenre
19:08:42,717 INFO  [java.sql.DatabaseMetaData] (ServerService Thread Pool -- 58) HHH000262: Table not found: idSong
19:08:43,412 INFO  [org.hibernate.tool.hbm2ddl.SchemaUpdate] (ServerService Thread Pool -- 58) HHH000232: Schema update complete
19:08:44,688 INFO  [org.jboss.resteasy.spi.ResteasyDeployment] (ServerService Thread Pool -- 59) Deploying javax.ws.rs.core.Application: class
resources.MusicalGenreResource$Proxy$_$$_Weld$EnterpriseProxy$
19:08:44,777 INFO  [org.jboss.resteasy.spi.ResteasyDeployment] (ServerService Thread Pool -- 59) Deploying javax.ws.rs.core.Application: class
resources.SongResource$Proxy$_$$_Weld$EnterpriseProxy$
19:08:44,785 INFO  [org.wildfly.extension.undertow] (ServerService Thread Pool -- 59) WFLYUT0021: Registered web context: /musicManagement-angu
lar
19:08:44,825 INFO  [org.jboss.as.server] (Controller Boot Thread) WFLYSRV0010: Deployed "derbyclient.jar" (runtime-name : "derbyclient.jar")
19:08:44,826 INFO  [org.jboss.as.server] (ServerService Thread Pool -- 34) WFLYSRV0010: Deployed "LibraryServerSession.ear" (runtime-name : "Li
braryServerSession.ear")
19:08:44,826 INFO  [org.jboss.as.server] (ServerService Thread Pool -- 34) WFLYSRV0010: Deployed "musicManagement-angular.war" (runtime-name :
"musicManagement-angular.war")
19:08:45,068 INFO  [org.jboss.as] (Controller Boot Thread) WFLYSRV0060: Http management interface listening on http://0.0.0.0:9990/management
19:08:45,068 INFO  [org.jboss.as] (Controller Boot Thread) WFLYSRV0051: Admin console listening on http://0.0.0.0:9990
19:08:45,069 INFO  [org.jboss.as] (Controller Boot Thread) WFLYSRV0025: WildFly Full 9.0.1.Final (WildFly Core 1.0.1.Final) started in 43901ms
- Started 580 of 770 services (247 services are lazy, passive or on-demand)
```

If the deployment doesn't report any error, the deployments is completed and the application is ready on the following address:

http:/localhost:<Wildfly-port>/musicManagement-angular/

Since the pom.xml has not been changed from the project guide, in order to create the war file and deploy it at the same time without using an IDE, follow the README instructions present at this link:

https://github.com/radcortez/javaee7-angular



Music Management Angualr

Record saved successfully! X

Genre: genre1

Song: snow

Genre properites

Genre: genre1

Description: genre1 desc

Save Delete

Song properites

Name: snow

Description: red hot chilli peppers

Youtube name: http://www.youtube.com/embed/yuFI5KSPAt4

Save Delete

Youtube video:

4.Note:

Using the bidirectional one to many relationship, I noticed that hibernate has a strange behaviour on this realtions, generating cyrcular dependings and returning as resource an infinite JSON cycle.
In order to avoid that I had to break the bi-directional relationship setting the returnment state = null at the method public MusicalGenre getMusicalGenre() situated in the entity Song.