Unit 4. Final exercise

Actors database

Service and Process Programming

Arturo Bernal
Nacho Iborra

IES San Vicente

Index of contents

1. Introduction

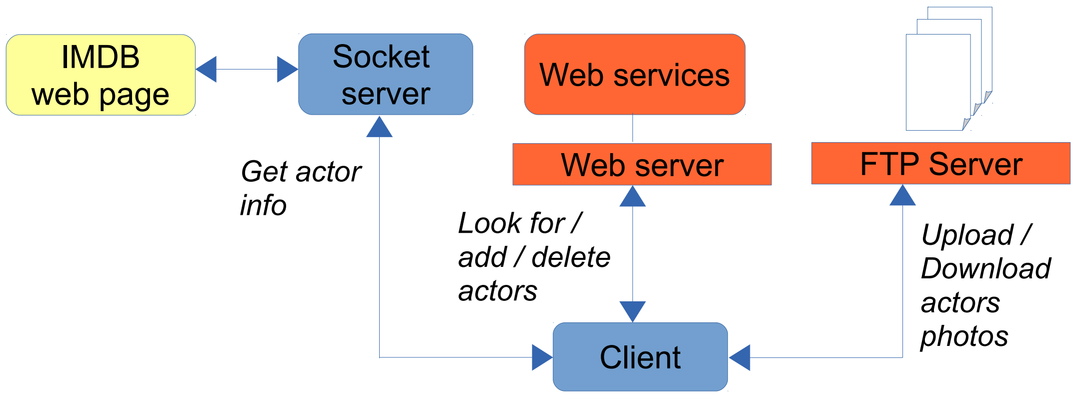
In this final exercise, we are going to implement a client-server application that stores some information about actors and actresses. We will firstly see the main structure of the application, and all the elements involved, and then we will explain how to implement each one, and the main features that must be covered.

1.1. Application structure

The application will have these elements:

- A client JavaFX application from which we will connect to multiple elements: a socket server, a web service and an FTP server. We will see later what to do with each connection.
- A socket server that will attend to client connections to look for information about a given actor. It will connect to a remote web to retrieve this information.
- A set of web services that will receive actors data to store them in a server, and queries to retrieve information about the stored actors.
- An FTP server to upload/download the photos of the actors

The diagram of communications is as follows:



You will need to implement all the elements coloured in blue in the diagram above, and you will also need to install and start the servers and services coloured in red in a local or remote machine, to test the application. You will be provided with all the web services, and the instructions needed to communicate with each remote web service and get information from it.

Let's see how each element must work, and how to test the application.

2. The socket server

The socket server will be a TCP socket server that will attend to client connections. For each connection established, the server will receive a text to be searched in a remote web (IMDB movie and actors database). This text will be typically the name of an actor (or part of it). Then, it will connect to that web and retrieve the information about this actor (or actress), if it exists. Finally, it will send the information to the client, and the communication will be closed afterwards.

2.1. How to connect and parse IMDB's web page

We are going to use the IMDB web to get the information of the actors. In order to connect to it and get the information, we need to follow some instructions.

2.1.1. Connect to URL with the given search string

Firstly, we have to transform the search string received from the client to a URL format (spaces, accents, etc. are not allowed). To do this use this method:

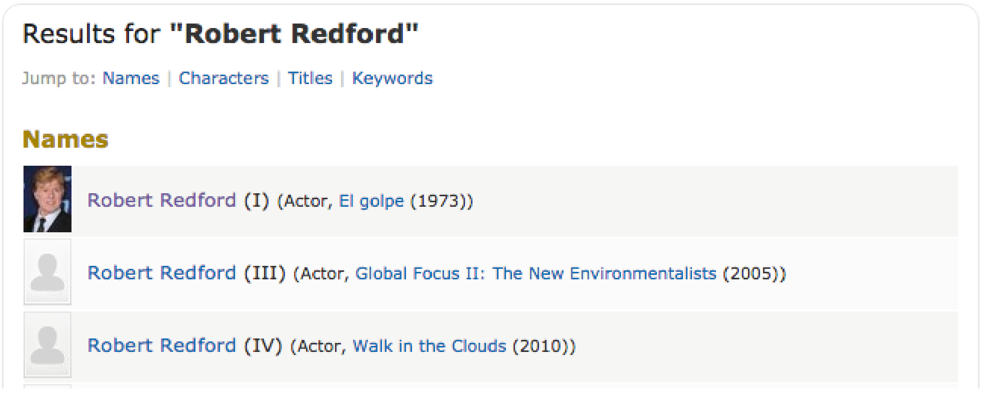
```
URLEncoder.encode(search, "UTF-8");
```

Then, to find out if there are any results for a search string, get the HTML from this link:

"http://www.imdb.com/find?q=" + search

2.1.2. Parsing the HTML

If you get the complete HTML response as a String, you'll have to search for several HTML elements in order to get the necessary information. **Right click** on the element you want to find and open the **inspector** in your browser. For example, when searching for actors, you'll need the link for the first name (if any). This is the HTML code to get an actor link:


Results for "Robert Redford"

Jump to: Names | Characters | Titles | Keywords

Names

Robert Redford (I) (Actor, El golpe (1973))

Robert Redford (III) (Actor, Global Focus II: The New Environmentalists (2005))

Robert Redford (IV) (Actor, Walk in the Clouds (2010))

Then, search for any element close to the link (use a class or an id better). For example:

```
int index = resp.indexOf("class=\"findResult");
```

Once you are close, search for the beginning of the link to the actor's page (using the previous index as a starting point):

```
index = resp.indexOf("href=", index);
```

Then, find the end position for href (+7 is to start searching 7 letters after the 'h' of "href="):

```
int index2 = resp.indexOf("\"", index+7);
```

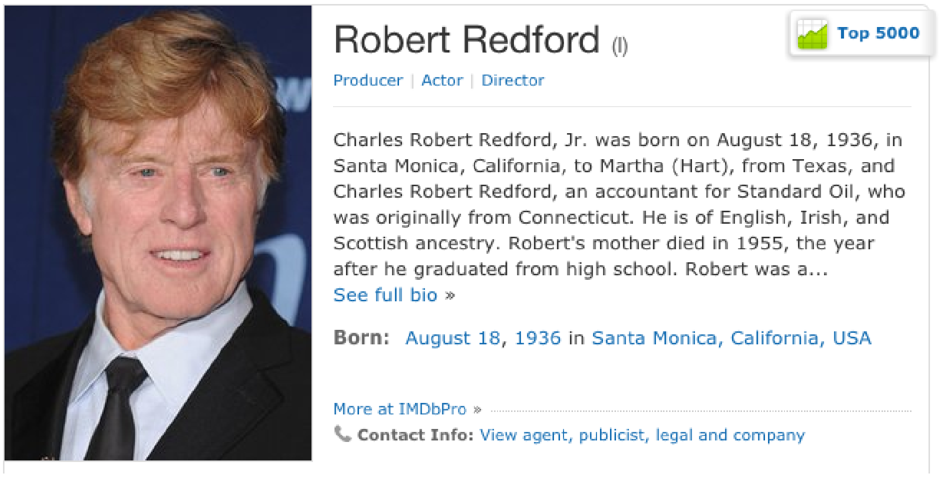
Finally, you can get the link for the actor result:

```
String link = resp.substring(index + 6, index2);
```

Get the HTML from **"http://www.imdb.com" + link** and you can start to get the actor information.

2.1.3. Information to get from the actor's page

Once you get to the actor's page by following previous steps, you will receive actor's page contents:



Robert Redford (I)

Producer | Actor | Director

Charles Robert Redford, Jr. was born on August 18, 1936, in Santa Monica, California, to Martha (Hart), from Texas, and Charles Robert Redford, an accountant for Standard Oil, who was originally from Connecticut. He is of English, Irish, and Scottish ancestry. Robert's mother died in 1955, the year after he graduated from high school. Robert was a...
See full bio »

Born: August 18, 1936 in Santa Monica, California, USA

More at IMDbPro »

Contact Info: View agent, publicist, legal and company

From this web page, you must get the following information:

- Image link (search for id “name-poster”).
- Actor's name
- Short description
- Birth date (only date, not including birth place)

2.2. How to send the information to the client

Once the socket server gathers the information from IMDB web page, it must send it to the client. Create a serializable Actor class (you should use the same class for this and for the web services JSON parsing) with the information of the actor and send an object of this class to the client. Sending other kind of text-structured data to the client (string separated by “;” for example) will have a penalty of -0,25 points on your final mark. Keep in mind that:

- If the search query returns no results, the server must notify the client that no results have been found. This can be accomplished by sending an empty object (empty name field for example).
- If the search query returns one or more results, the server must return only the first actor found to the client.

- Look at the web services (specially GET) to know the necessary fields for the class. You can create a Java JAR project or class library to easily share that class between the server and the client.

3. Web services

There are 8 web services available (use all with the prefix **http://<IP SERVER>/finalex4**):

- /**login** (POST) → Will be used to log in into the server, so that we can keep track of our favourite actors. The fields that must contain this JSON object should be:

  ```
  {
       "login":"user's login",
       "password":"user's password"
  }
  ```

  It will return the same JSON data than for inserting or deleting actors, and an additional *token* field with the *token* string that must be stored in the ServiceUtils class in order to call the rest of web services once the user is authenticated. If *ok* field is not *true*, then *token* field will be empty.

  ```
  {
       "ok":true,
       "error":"",
       "token":"token_string"
  }
  ```

  There are 2 users created in the database, **bob** and **alice**, with password **1234**.

- **/actor** (GET) → Will return an array of all actors with this format:

  ```
  [
     {
       "id" : "1",
       "name" : "Robert Redford",
       "birthDate" : "August 18, 1936",
       "imgFile" : "http://SERVER/finalex4/img/r_redford.jpg",
       "description" : "Charles Robert Redford, Jr. was born
  on..."
     },
     { … }
  ]
  ```

- **/actor/{name}** (GET) → The same format as before but returning only actors that contain the search string sent as part of their name.

- **/actor** (POST) → Will receive by POST the information of an actor object and insert it in the database. The minimum fields that must contain this JSON object should be:

  ```
  {
       "name":"Actor's name",
       "imgFile":"Name of the actor's image file (uploaded by
  FTP)",
       "birthDate":"Actor's birth date (string)",
       "description":"Short description about the author"
  }
  ```

  This service will return a JSON object with the following structure: a boolean to check if the operation was successful, and an error message (empty if everything went OK)

```
{
     “ok”:false,
     “error”:”There was an error inserting the actor”
}
```

- **/actor/{id} (DELETE)** → Will receive the id of the actor to delete. It will return a JSON object with the same structure than previous web service for inserting actors.
- /**favourite** (GET) → Will return the list of favourite actors for current user. It will return the same JSON data format than for listing actors (*actor GET* web service)
- /**favourite/{id}** (POST) → Will be used to add a new favourite for current user. **{id}** is the actor’s id that will be added to favourites (you don’t need to send any additional data).

  It will return the same JSON data than for inserting or deleting actors **.**
- /**favourite/{id}** (DELETE) → Will be used to delete an existing favourite from current user's list. **{id}** is the actor’s id that will be removed from favourites.

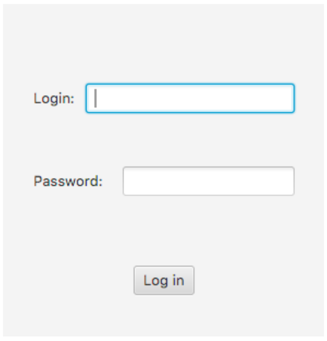
  It will return the same JSON data than for inserting or deleting actors.

4. The client JavaFX application

The client part of this application is quite complex, since it has to handle many connections to different types of servers and services.

4.1. The login view and the login process

At the beginning, the main application must show a view with a login form, like this one:

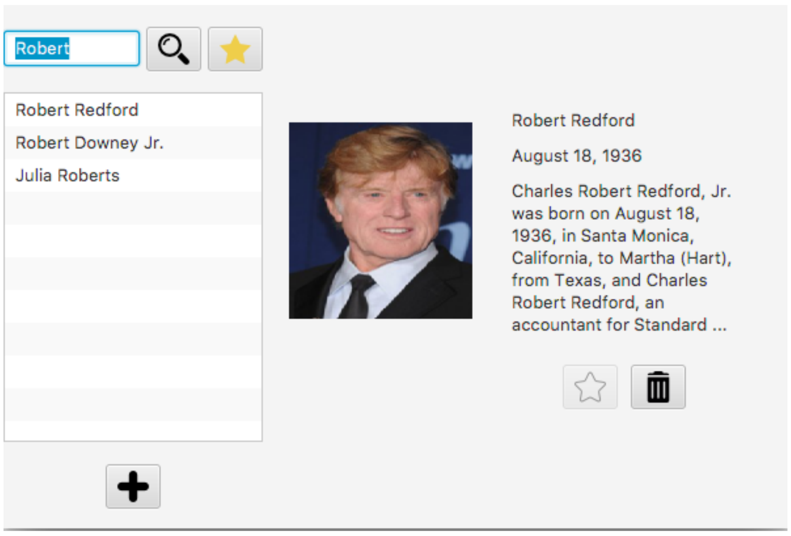


When we type the login and password and click on the *Log in* button, we will call a remote web service (*login* web service, see section 3 before for more details). It will return JSON data where we can see if log in process was OK. If so, we must store the login token to that we can use it from our *ServiceUtils* class to send it in every web service call. If login was not successful, we must show an alert or label message.

> To see how to store the token and use it from *ServiceUtils* class, see Annex IV of this unit and the *ServiceUtils* example code (specially *ServiceUtils setToken* static method).

4.2. The main view

Once we log in, we must switch to the main application view (see Annex II of Unit 2 to remember how to manage multiple views). The general appearance of the main application must be more or less like this:



Robert Redford

August 18, 1936

Charles Robert Redford, Jr. was born on August 18, 1936, in Santa Monica, California, to Martha (Hart), from Texas, and Charles Robert Redford, an accountant for Standard ...

4.2.1. Adding icons to buttons

If you want to specify a given image or icon for a button, you need to:

- Have the image available in the project folder
- Add these lines (in the *initialize* method of the corresponding controller, for instance):

  ```
  Image imgDelete = new Image("file:delete.png");
  btnDelete.setGraphic(new ImageView(imgDelete));
  ```

  In this example, we are loading an image called *delete.png.* from root project folder.

You can search for icon images in many web pages, such as this one. Anyway, you will be provided with the set of icons used in our implementation, in case you want to use the same ones. Once you get your desired icon images, place them in the project's root folder in order to locate them as in previous example.

4.2.2. Scaling images

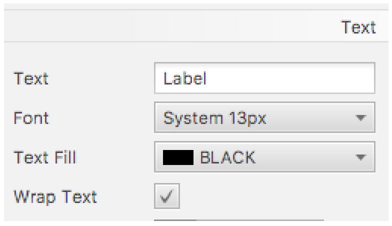
You may need to scale the actor's image when putting it in the *ImageView* control of your application, so that this control does not change its size to adapt to the corresponding image. To do this, you must create the *Image* object with some additional parameters. For instance:

```
Image img = new Image("r_redford.jpg", 140, 150, false, true);
```

In this example, we are specifying that the image must be scaled to a width of 140px and a height of 150px (these dimensions may vary depending on the size of your *ImageView* control), without preserving ratio (*false*, otherwise either width or height will not be the same than you specified), and with smooth scale (*true*).

4.2.3. Wrapping text into a label

If you pay attention to the main application appearance, we are using three labels at the right of the actor's image to display his information: name, birth date and description. For this last field, we need to wrap multiple lines into the label, and to do this, we just check the *Wrap Text* property in Scene Builder for that label.



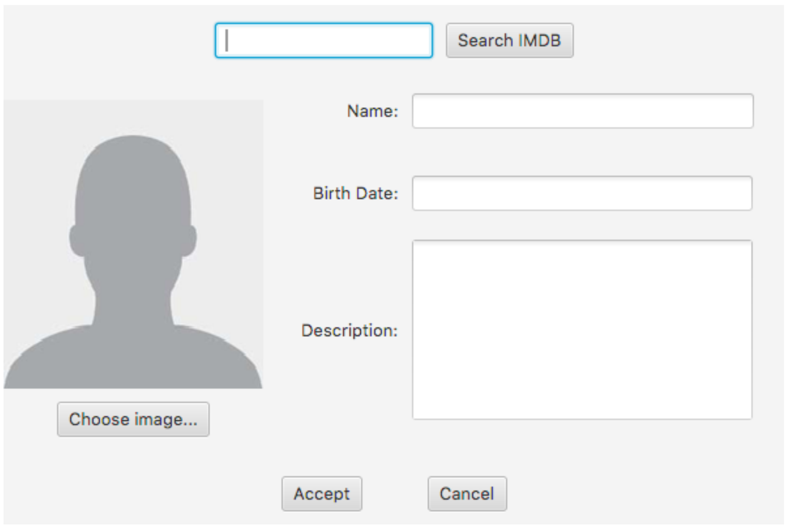
Text

Text	Label
Font	System 13px
Text Fill	■ BLACK
Wrap Text	✓

4.3. Adding actors

If we want to add an actor/actress to our collection, we must click on the *Add* button:

Then, a new window (modal dialog) will be opened, with this appearance:



From this dialog, we can add actors directly by typing its information in the text fields (name, birth date and description) and choosing an image.

We can also search the actor in the IMDB web repository, by typing some search text in the text field above, and clicking on the *Search IMDB* button. Then, we will connect to the socket server explained in section 2, send the search text to it, and retrieve the information. Finally, all the information about the actor (the text fields and the image view) will be automatically filled with the server response. If no results were found, an alert or a label with a message must be shown with the error.

Once we have the form filled, we can add the actor by clicking on the *Accept* button. Then, we will call the appropriate web service from the ones explained in section 3. Otherwise, we can click on the *Cancel* button or close the dialog to come back to the main window (without adding the actor).

4.3.1. Connecting to an FTP server to upload the movie image

If the add operation is successful, then we will need to upload the movie image to the server (the web service will only send the image name). So, after adding the movie info, the client will automatically connect to an FTP server and upload the image loaded in the ImageView.

Image's file name could be for example the same as the movie's name, followed by the appropriate extension.

This is how you can store an image to a file using the link received from the server:

```
BufferedImage buffImg = ImageIO.read(new URL(imgUrl));
ImageIO.write(buffImg, "jpg", imgFile);
```

Once we connect to the server, we must upload the image to the subfolder *public/img* inside the application *finalex4* of your web server (you will need to download this application from the resources of this final exercise, in the Virtual Classroom). Make sure that this folder has write permissions for everyone in order to upload the images.

4.4. Looking for actors in our database

If we want to look for a given actor/actress in our database, we will type the actor's name (at least, partially) in the text field above, and click on the *Search* button 🔍. When we do this, the client will connect to a web service and send a GET operation to it, to retrieve all the actors that match with the search pattern. The web service will return a JSON array of actors matching the pattern, and the client will process this information, store it in an appropriate collection and show the results in the left list view. If we click on any actor in the list, its information will be displayed on the right.

4.4.1. Deleting actors

Once we are navigating through a given actors list returned from the web service, we can choose any of these actors and click on the *Delete* button to delete it from our repository. When we click on this button, we will connect to the local web service to delete the information about this actor, and to the local FTP server to delete its associated image.

4.5. Favourites

Apart from adding, deleting and listing actors, our client application must be able to handle a list of favourite actors. To do this, we are going to use the *favourite* web services (GET, POST and DELETE) explained in section 3.

4.5.1. Listing favourites

If we want to see our favourites list, we click on the *Search favourites* button . Then, the program must call the *favourite* web service with a *GET* command, gather the information provided by the server and if everything is OK, then load the actors in the left list view. If no favourites are found, then the list will remain empty, and an alert message must be shown.

4.5.2. Adding favourites

If we want to add a new favourite actor, we must:

- Look for it in our database so that he appears in the left list (type the text to search and click on the search button, as explained in subsection 4.3)
- Select his/her name from the left list
- Click on the *Add favourite* button . As soon as we click on this button, the application will call the *favourite* web service with a POST command, and send the appropriate data to that service to include that actor in the favourites list of current user.

4.5.3. Deleting favourites

If we want to delete a favourite, we must:

- List the favourites by clicking on the search favourites button .
- Select the favourite that we want to remove from the left list.
- Click on the *add favourite* button . In this case, this button will call the *favourite* web service with a DELETE command to remove the favourite from the list.

5. Implementation and marks

Apart from the elements explained in this document, you can add any additional classes and methods to the basic structure. For instance, you can implement an *Actor* class with all the information about an actor (name, birth date, description and image file name), make it serializable, and use it to send information between the server socket and the client.

5.1. Your final mark

The mark for this exercise will be calculated as follows:

- Socket server (**2 points**)
  - Connecting to the remote web (IMDB) and retrieving information from it according to a given search text: **1 point**
  - Accepting client connections and sending actors information to them once it is retrieved from the remote IMDB web: **1 point**
    - Sending an String instead of a serialized object will have a penalty of **-0.25p**
- Client JavaFX application (**7 points**)
  - Login process, calling the appropriate login service and storing the token properly (**0,5 points**)
  - Connecting to server socket and retrieving actors information when clicking on the *Search IMDB button* from the *Add actor* dialog. It includes connecting to server, receive information and show it in the *Add actor* modal dialog**: 1 point**
  - Connecting to local web service and retrieving search results when clicking on the *Search* button. It includes connecting to service and getting the results (transforming JSON into Java objects). Then, show results in the list view and show actor information when selecting any actor from the list: **1,5 points**
  - Adding actors to the repository. It includes using the local web service to add the actor when clicking the *Accept* button from the modal dialog, and connecting to the FTP server to upload the image: **1,5 points**
  - Deleting actors from the repository (using the local web service to delete the selected actor when clicking the *Delete* button, and deleting the image through an FTP connection): **1 point**
  - Favourite management: adding/listing/deleting favourites from the server by accessing the appropriate web service: **1,5 points**
- Error handling and documentation (**1 point**). It includes things like:
  - Showing appropriate error messages when connection to servers and services fails: **0,25 points**
  - Hiding or disabling controls when no operation can be done with them. For instance, we can't delete an actor if we don't select any from the list view. We can't add an actor if we don't search in the remote web or fill the dialog data: **0,25 points**

    - Code is properly documented and Javadoc is generated: **0,5 points**

5.1.1. Optional add-ons

You can add this elements to the project and have some extra points for your mark:

- Add the ability of editing the actor information without deleting and/or adding the actor again (**1 point**)

  Use this web service if you need to edit an actor:

  /actor/{id} (PUT) → Edits an actor information. **{id}** is the actor’s id. You’ll have to send the same information as in the actor’s POST service, but you can omit or leave empty the “imgFile” field when you don’t want to change the image.

  This service returns the same as the POST service.