

Java Web Applications: servlets and jsp

Rui Couto António Nestor Ribeiro

April 12, 2021

Contents

| | | |
|----------|-------------------------------------|-----------|
| 1 | Introduction | 1 |
| 2 | GMS Facade | 2 |
| 3 | Setup | 3 |
| 3.1 | Application Server Setup | 3 |
| 3.2 | Project Setup | 3 |
| 3.3 | Setting up the data | 4 |
| 4 | Implement the MVC pattern | 6 |
| 4.1 | Controller | 6 |
| 4.2 | View | 7 |
| 5 | Request, Response, Session | 8 |
| 5.1 | Accessing data & Sessions | 9 |
| 6 | Listing games | 10 |
| 6.1 | Resulting JSP | 10 |
| 7 | Registering a user | 11 |
| 8 | Development | 12 |

1 Introduction

Building on top of the last tutorial, this one describes the process of creating a Java Web application.

We have already implemented the business and the data layers. The data layer, accordingly to the first tutorial, was generated using visual paradigm, while the business was implemented on top of that.

In this tutorial we present how to implement the presentation layer, resorting to Java Web technologies.

The presentation layer will respect the MVC pattern. The model corresponds to the Facade developed in the previous tutorial. It contains a set of features required for the interface implementation.

In this example, the controller is implemented resorting to a servlet. A servlet is a special Java class that can handle http requests (made by the browser, for instance). The servlet will process the request by specifying the required interactions with the model and providing the presentation.

In this tutorial the presentation is represented by a JavaServer Page (JSP). A JSP is a server-side webpage that contains both html and Java source code, and is compiled in order to deliver HTML to the user. It can receive parameters from the servlets and processes them in order to create dynamic pages.

When a request is made to a Java Web application, the Application Server will search for a resource to handle it (being it a servlet, a static page, a file, etc). The servlet will handle the requests and will have access to all the data submitted by the client. This includes both the explicit data (i.e. forms), or implicit data (e.g. user-agent).

2 GMS Facade

According to the previous tutorial, the following list of features should exist in the GMS Facade.

1. Register a user;
2. Register a game;
3. Register a platform;
4. List user games;
5. List all games;
6. Search a game;
7. Delete a game.

In this tutorial in order to demonstrate the concepts the *Register a user* and *List all games* use cases will be addressed.

3 Setup

A setup phase is required prior to development, namely regarding the development environment.

3.1 Application Server Setup

Several Application Servers exist to run Java web applications, being the most well known **GlassFish**, **TomCat**, **Websphere** and **Wildfly**. This tutorial will present the process of deploying an application for the **Wildfly**. Any Java web application developed resorting only to the standard APIs is interchangeable with different application servers.

The first step is to download the application server ¹ from the official website <http://wildfly.org/downloads/>.

After extracting the downloaded file, the application server is ready to run. It can be run via command-line and its administration can be done via a web interface. However, in this tutorial, the application server will be integrated in the IDE instead. This way, the IDE manages the application server (i.e. start, stop), deployment of applications, etc.

The process of adding this application server in NetBeans² is straightforward.

1. Click in `Window > Services`.
2. Right click in `Servers > Add server...`.
3. Select `WildFly Application Server`, then `next`.
4. In `Server Location` select the application server folder.
5. Click `Next` and `Finish`. If a warning message appears, accept it.

At this stage, the application server setup in the IDE is complete. Now it is possible to create a new web application.

3.2 Project Setup

The setup of the new project requires to a) import the previously generated Java source code, and, b) select the recently created application server setup.

1. Create a `New project...`.

¹This tutorial was made using version 16.0.0.Final.

²The process should be similar for other IDEs.


2. Select `Java Web`, `Web Application`.
3. In the next step, give the application a name (e.g. `GMSWeb`) and proceed.
4. In `Server`, select `WildFly Application Server`, and finish the process.

The application has now been created. Next, the generate Java source code needs to be imported.

1. Open the folder where the source code has been generated.
2. Select and copy the folders inside `src`.
3. In NetBeans, paste the folder in the project `Source packages`.

Several errors should appear in the imported files. This is due to the missing ORM library.

1. Copy the file `orm.jar` inside the generated sources `lib` to your project folder.
2. Right click the project (in NetBeans), and select `Properties`, `Libraries`, `Add JAR/Folder`.
3. Select the `orm.jar`, and click `ok`.

At this stage, the base project is set up and ready to run. Click in `run` (either right click in the project, and `run`, or click in the  icon), wait for the application server to start (might take a while), and the browser should open automatically with an empty page. If you see a page similar to the one in Figure 1, the setup is complete

3.3 Setting up the data

Prior to start the development we need to create some data in the database. For instance, using the following test data:

Platforms

- Mega Drive, 1988, A 16-bit home video game console, SEGA.
- PC, 1977, A personal computer (PC) is a multi-purpose electronic computer whose size, capabilities, and price make it feasible for individual use., N/A.
- Playstation 4, 2013, The PlayStation 4 (abbreviated as PS4) is a home video game console developed by Sony Computer Entertainment., SONY.

Figure 1: Index.html provided by WildFly.



Games

- Sonic the Hedgehog, 1991, 60, Sonic the Hedgehog is a platform video game developed by Sonic Team and published by Sega for the Sega Genesis console. (Mega Drive)
- Dune, 1993, 65, Dune: The Battle for Arrakis is a real-time strategy Dune video game developed by Westwood Studios. (Mega Drive)
- Half Life 3, 2018, 60, Half-Life 3 is a first-person shooter video game developed and published by Valve Corporation. (PC)
- Age of Empires, 1997, 70, Age of Empires is a series of personal computer games developed by Ensemble Studios (defunct in 2009) and published by Microsoft Studios. (PC)
- GTA V, 2013, 70, Grand Theft Auto V is an action-adventure video game developed by Rockstar North and published by Rockstar Games. (PS4)

Also, at least a user is required for the application. The user password should not be plain text, but a hash instead (e.g. SHA-256)³.

Tasks

1. Populate the database with the provided information.
2. Create a new user, with an hashed password.

³See the following link for more information on hashing in java <https://docs.oracle.com/javase/7/docs/api/java/security/MessageDigest.html>

4 Implement the MVC pattern

Having the data created, next follows the process of creating the supporting files for the MVC pattern. It is a good practice to start by creating a new package to place the web related Java classes. So, create a package called (for instance), `web`, inside `Source Packages`.

As stated before, the *controller* is implemented by a servlet. In that sense, right click in the new package, and select `new...`, `Servlet`. Set the servlet name to `Index`, and then `Finish`. At the moment, the application server is responding to requests on `/Index`: `http://localhost:8080/GMSWeb/Index`.

The *view* is implemented by a JavaServer page file. So, right click `WEB-INF`, and select `new`, `JSP`. Create the file `index.jsp`. The content of this page is not directly visible by the browser, since the page is placed in `WEB-INF`. The page content needs to be rendered by the a Servlet instead. A `JSP` consists in a file which contains both HTML and Java code. This code is processed in the server side, and only the rendered HTML is sent to the client.

Although all the MVC components are gathered, they are still not unified. Prior to that, configurations are required.

Tasks

1. Access `index.html` in the browser. Locate it in the project and see its content.
2. Access the servlet in the browser. Locate it in the project and see its content.
3. Modify the servlet, to see the changes reflected in the browser.

4.1 Controller

When a web application loads, it will look for an `index` file (either `html` or `jsp`). However, it is possible to configure it to load another file instead. The existing `index.html` is providing static content, therefore not required for this tutorial.

In order for the application to load the Controller (instead of a view), we need to create a configuration file. So, right click the project, and select `new`, `Standard Deployment Descriptor (web.xml)`. This file allows to configure several parameters of the application. We are specially interested in the file to load when the application starts. The specification is as follows (inside `web-app`).

```
1 <welcome-file-list>
2   <welcome-file>Index</welcome-file>
3 </welcome-file-list>
```

This way, when our application starts, the content will be generated by the servlet. We can now delete the static `index.html` page. The url `http://localhost:8080/GMSWeb/` should now be handled by the servlet.

4.2 View

Currently, the Servlet is generating the content by explicitly doing it through programming code sentences. We want, however, this content to be generated rendered by the `JSP`. The content of the servlet `processRequest` method can be replaced by the following one.

```
1 Object obj = "hello";
2 request.setAttribute("obj", obj);
3 request.getRequestDispatcher("/WEB-INF/index.jsp").forward(request, response);
```

Opening the url `http://localhost:8080/GMSWeb/`, should now present the `JSP` content.

The previous code shows an example of sending a parameter from the controller (servlet), to the view (jsp). Now, in the `JSP` we can access the parameter through *Expression Language*⁴ or *scriptlets*⁵, in several ways:

`EL`

```
1 The parameter value is: ${obj}
```

`scriptlet`

```
1 The parameter value is
2 <%
3     String param = request.getAttribute("obj").toString().toUpperCase();
4     out.println(param);
5 %>
```

`scriptlet (inline)`

```
1 The parameter value is: <%= request.getAttribute("obj") %>
```

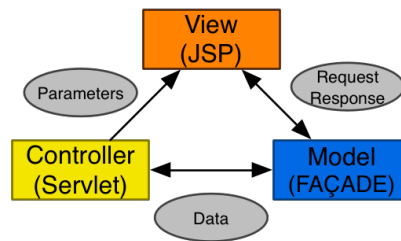
Briefly, through EL, it is possible to access the parameters provided by the servlet, with the syntax `${<parameter>}`. Scriptlet supports writing Java code inside the tags `<% <code> %>`. With this syntax it is possible to print html with `out.println`. Finally, Scriptlet inline allows to directly print a value `<%= <value %>` into html.

⁴More information on EL <http://docs.oracle.com/javaee/6/tutorial/doc/gjddd.html>

⁵<http://docs.oracle.com/javaee/5/tutorial/doc/bnaou.html>

At this stage, the MVC pattern is implemented, as presented in Figure 2. The Servlet (controller) receives the browser requests, and interacts with the GMS Facade (model), in order to retrieve data. Through parameters, that data can be send to the JSP (view), which renders it.

Figure 2: MVC in Java web.



Tasks

1. Create a Game object in the servlet.
2. Pass it as an attribute to the JSP.
3. Display in the JSP the game information.

5 Request, Response, Session

Three variables are relevant when dealing with the information sent from the Servlet to (and provided by) the JSPs.

Request contains the data provided by the user. This includes, on the one hand, the browser and connection information. Are example of fields, the user IP, the browser user-agent, etc. On the other hand, it provides the explicit data. This includes url query string values, data from forms, etc. With this variable it is possible to access the data provided by the user, for instance, to perform login or create a new entity instantiation.

Response variable is accessible both to the servlet and to the JSP. This means that data injected in the Servlet can be accessed in the JSP. Furthermore, the response includes data such as the kind of response, to be interpreted by the browser.

Session contains information which is kept, in the server side, between requests. This allows the application to identify the client, and create resources specifically for that client. Sessions are useful to keep information about the user, such as the login state. Sessions expire when the user is inactive (see `web.xml`).

The session is stored inside the request variable, and provides the following methods (similar to a `Map`) to store and retrieve data:

- `getAttribute(key)` given a *key*, returns the associated value if it exists, or `null` otherwise;
- `setAttribute(key,value)` puts the *key/value* pair in the session;

Tasks

1. Get the user-agent in the servlet (see `request.getHeader` method).
2. Send that information to the JSP, and display it.

5.1 Accessing data & Sessions

The data layer of the application is built on top of the Hibernate framework. Each time a request is made through a DAO, a new persistence session is created to handle the request. This is very resource consuming, and will eventually exhaust the database connection pool.

Web applications attribute a session to each user (as already seen), which is kept between resources. So, in order to avoid connection issues, the Hibernate persistence session should be bind to the web application session. This way, when a user makes a request, the persistence session is reused, until it eventually expires.

A persistence session is created with:

```
1 PersistentSession session = GamesLibraryPersistentManager.instance().getSession();
```

The application session is retrieved with:

```
1 HttpSession httpSession = request.getSession();
```

When invoking the DAO methods, the methods which take as an argument the session should be used, as for instance:

```
1 List<Game> games = GMS.listGames(session);
```

Tasks

1. Store a persistence session for each user, resorting to the session variable. Start by getting the application session, and check the existence of a persistence session. If it does not exist, create a new one, and put it in the session. Otherwise, simply retrieve the existing session.
2. Confirm that the connection is being reused in each request (e.g. through a `System.out.println`).

6 Listing games

At this stage, all the requirements to create a page to list all the existing games are met. So, in the servlet we can retrieve the list of all existing games. The list should be communicated to the JSP, which will render it (through a scriptlet). An example of listing the games is as follows:

```
1 <ul>
2 <% List<Game> games = (List) request.getAttribute("games");
3     for(Game g : games) { %>
4         <li> <%= g.getName() %> </li>
5     <% } %>
6 </ul>
```

Notice that it is possible to start and end the scriptlet tags and refer the variables outside them (c.f. line 4).

Tasks

1. Modify the listing example to show all the game details in a table.

6.1 Resulting JSP

The resulting JSP to list the games is as follows.

```
1 <%--
2     Document : index
3     Author : ruicouto
4 --%>
5
6 <%@page import="java.util.List"%>
7 <%@page import="pt.uminho.di.aa.Game"%>
```

```

8 <%@page contentType="text/html" pageEncoding="UTF-8"%>
9 <!DOCTYPE html>
10 <html>
11     <head>
12         <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
13         <title>JSP Page</title>
14     </head>
15     <body>
16         <h1>Games</h1>
17
18         <ul>
19             <% List<Game> games = (List) request.getAttribute("games");
20                 for(Game g : games) { %>
21                 <li><%= g.getName() %></li>
22             <% } %>
23         </ul>
24     </body>
25 </html>

```

7 Registering a user

In order to create a new entry it is required to send data from the view to the controller. As presented before, this is done through the request variable. We start by creating a way for the user to input data. This is done through an HTML form, as shown below.

```

1 <form method="POST">
2     <input type="text" name="name"/>
3     <input type="text" name="email"/>
4     <input type="password" name="password"/>
5     <input type="submit" value="Register"/>
6 </form>

```

This will present a form with three fields, and a button to submit data for the server. When the Register button is pressed, the servlet is invoked again, and in the request we can access the fields.

```

1 String email = request.getParameter("email");
2 String name = request.getParameter("name");
3 String password = request.getParameter("password");
4
5 try {
6     GMS.register(name, email, password);

```

```
7 } catch (Exception e) {  
8     e.printStackTrace();  
9 }
```

Tasks

1. Implement the register feature. This feature should be handled by another Servlet.
2. Implement the login feature. You can use the web session to keep track of the logged user. Also, the register should only be available for unregistered users.

8 Development

A set of features is proposed to implement, as follows.

1. Implement the "my games" page, in order to show the user games. Validate (in the controller) that only logged users can view this page.
2. Implement the feature to add a game to the personal library.
3. Implement the search feature.