



Università di Trento

Web Architectures

Assignment 3

Author:

Marcel Pascal Stolin
marcelpascal.stolin@studenti.unitn.it

October 25, 2021

1 Introduction

The task of the third assignment is to create a web version of the Memory game. In this game, different user can play a game and see the highscore of all users who have played before.

The applications is composed of two systems, the backend and frontend. The backend is responsible to authenticate the user with a username, save the highscore of each user, and respond the number of points for each guess made by the user. Additionally, it should be possible to set a *development mode*, where the grid of the memory is deterministic. Otherwise, in *production mode*, the grid is generated randomly.

The frontend side consists of the game (User Interface),v where the user can play the actual game. This frontend has to be implemented using JavaScript and it should communicate with the backend using Ajax.

2 Conceptual Design

The conceptual design is based on the problem statement introduced in Section 1.

2.1 Backend

The backend is responsible to provide authentication to the user. Additionally, after the user is authenticated, the user should be able to see a ranking of all users. Furthermore, the grid of the memory game is supposed to be generated on the server side. Therefore, the following paths have to be created:

- Authentication /authentication
- Ranking /ranking
- Grid /grid

2.1.1 Authentication

Whenever a user visits the application for the first time, the user has to provide a username. If not, the user should not be able to access the game, and will be redirected to the authentication page until the user enters a name. After the user has entered a name, the user will be redirected to the ranking page.

2.1.2 Ranking

On the ranking page, the user can see a list of all user who played a game before. Additionally to the username, the list shows the number of points the user has achieved. Furthermore, on the ranking page, the user can click the button Play game to start the memory game.

2.1.3 Memory Grid

When the user start the game, the game needs to know how to arrange the cards in a grid. Therefore, the backend has to generate a 2D array of the grid. The grid consists of 16 cards in a 4x4 grid. Each card exists 2 times, therefore there are 8 different cards on the grid in total.

In addition, it should be possible to decide if the backend operates in a *development* or *production* mode. If development mode is activated, the grid is generated in a deterministic way. Otherwise, in production mode, the grid is generated in a random way. Figure 1 shows an example of the different grid versions for each mode.

Additionally, the game will make a **POST** request to the grid, to receive the value of a position in the grid.

1	1	2	2		5	2	3	8
3	3	4	4		2	7	7	4
5	5	6	6		6	4	6	1
7	7	8	8		1	5	8	3
Development Mode					Production Mode			

Figure 1: Example of the grid for *development* and *production* mode

2.2 Frontend

The frontend part of the application consists of the Memory game. It show 16 cards in a grid (introduced in Section 2.1.3). The user can click on a card a the card will flip. After that, the user can click on a second card and the card will flip as well. After a card has been clicked, it will become unclickable as long as the user makes finishes the guess. If both cards match the guess was successful and the points will be added to the score of the user. The added points are two times the guessed card value (e.g. if both cards have the value 4, 8 points will be added to the user score). Otherwise, if the guess was incorrect, one point will be subtracted from the points of the user. To get the value of a card, the frontend has to send a request to the backend given the index of the card in the grid. Then, the backend returns the card value. The user can make 8 guesses (4 attempts to find pairs) in total. After that, the game finishes. Then, a *Game Over* label appears, and after 1 second, the user will be redirected to the ranking page.

3 Implementation

This section explains the implementation based on the conceptual design introduced in Section 2. As mentioned in Section 1, the MVC architecture is used to implement this application. Figure 2 presents the project folder structure of the MVC implementation. The models are introduced in Section 3.1, the views, and controllers in Section 3.3. Additionally, this implementation uses custom object stores (introduced in Section 3.2) and filters (introduced in Section 3.5).

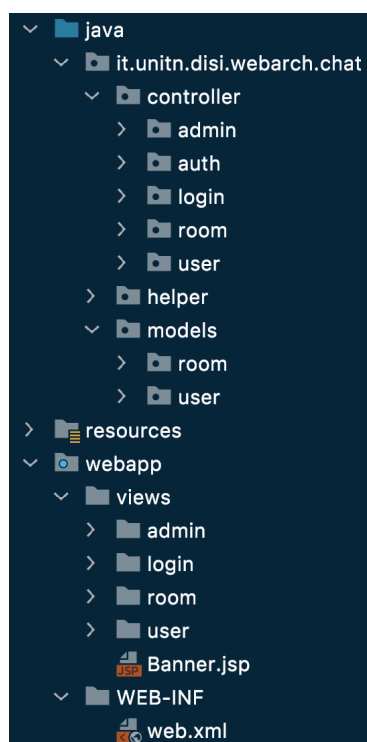


Figure 2: Project folder structure

3.1 Models

As being mentioned, this project is implemented using the MVC pattern. Therefore, each entity of the chat system is represented by a model. Each model is implemented using the Java Bean specification. This enables to reuse the models in JSP files. Figure 3 shows all models used in the implementation of the chat system. A **Room** model exists, which represents a room where users can chat with each other. Therefore, multiple messages are saved in a **Room**. A **Message** model represents a message sent by any user in a specific room. Each **Message** is identified by its message-text, the name of the user, and the date when it was sent. Each user is represented by a **User** model. This model is identified by the username and the password.

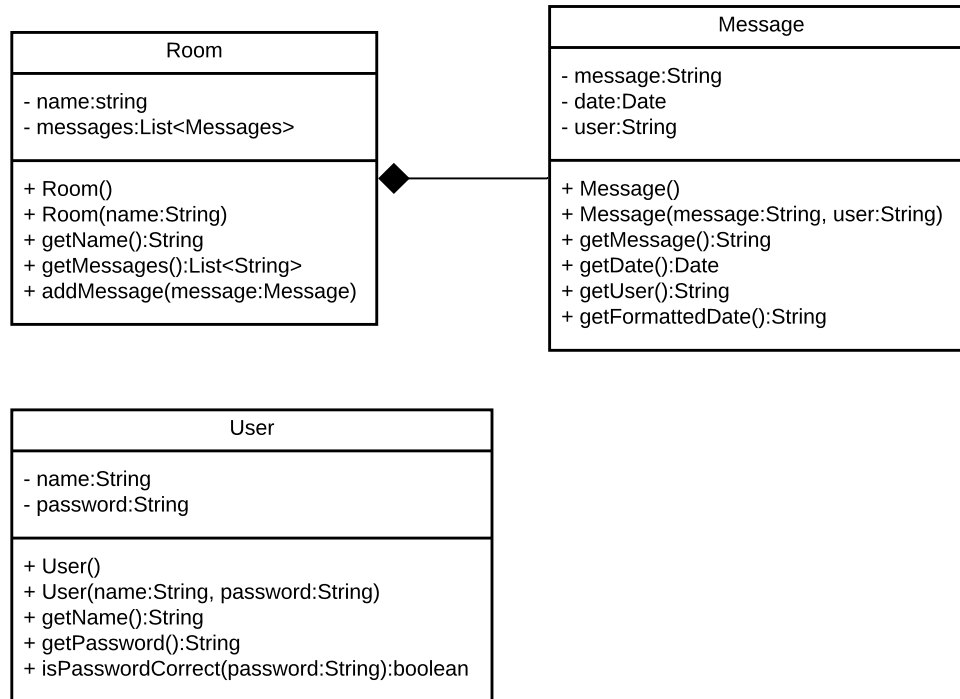


Figure 3: Used models for the chat system

3.2 Object Stores

As is mentioned in Section ??, a custom store is needed to save models introduced in Section 3.1.

Therefore, the following stores are being implemented:

- **UserStore**, saves **User** models
- **RoomStore**, saves **Room** models

Because a user and room can only exist once, a **Set**¹ data structure is used to save both **Users** and **Rooms**. Furthermore, a store needs to provide the functionality to add an object to the store, get all objects from the store, or a specific one, and check if an object has already been added to the store.

Therefore, an abstract class called **ObjectsStore** is implemented, which implements the previously mentioned functionalities. The **ObjectStore** uses generics to implement the properties and methods. The **UserStore** and **RoomStore** implement the **ObjectStore** class and set the model (**User** and **Room**, respectively) for the generic type. The lifetime of an object saved to a store is equal to the lifetime of the running webserver. As long as the webserver is running, the objects are saved to memory. If the webserver gets shut down, all rooms and messages are being removed from the memory. Except for the **UserStore**, which reads all **User**

¹Set (Java Platform SE 8) - <https://docs.oracle.com/javase/8/docs/api/java/util/Set.html>

models from a text file, and writes newly created Users to the same file. Therefore, when the server restarts, all previously added `User` models are available again.

To make the same set of Users, and Rooms available for each single servlet instance, the singleton pattern is used for the stores. This ensures, that all servlets work with the same set of `Users` and `Rooms`. An example of the singleton pattern usage is shown in Listing 1.

```
1 Set<Room> rooms = RoomStore.getInstance().getAll();
```

Listing 1: Example usage of the singleton pattern

3.3 Servlets

Each route introduced in Section ??, is implemented using a servlet. The following servlets are needed for this architecture:

- `AuthLoginServlet`
- `AuthLogoutServlet`
- `LoginServlet`
- `UserPageServlet`
- `RoomServlet`
- `RoomCreateServlet`
- `AdminServlet`

Most of the mentioned servlets own a specific view (a JSP file) which is saved in `Chat/src/main/webapp/views`. Listing 2 shows, how a servlet forwards a request to a specific view.

```
1 this.getServletContext()  
2 .getRequestDispatcher("/views/room/RoomCreate.jsp")  
3 .forward(request, response);
```

Listing 2: Forward a request to a view

3.3.1 Authentication

For authentication, the `AuthLoginServlet` and the `AuthLogoutServlet` are created. The `AuthLoginServlet` requires a POST request and validates the request accordingly. If the given username and password are valid credentials (the user with the given name exists, and the password is correct), the `AuthLoginServlet` creates a new `HTTPSession`, sets the active user, and the attribute `is_authenticated` to `true`. This property is important for the `AuthFilter` introduced in Section 3.5.1. After that, the `AuthLoginServlet` redirects the user to the *User-Page*. The users are received using the `getUser` method of the `UserStore`. However, the admin user is not stored in the `UserStore`. If the username is equal to

admin, the `AuthLoginServlet` checks if the password is equal to the init property `AdminPassword`, which is set in the `web.xml` (equals to `root`). If the POST request is invalid (no username and password set), the `AuthLoginServlet` sends a 400 - Bad Request HTTP error. Otherwise, if the request is valid, but the credentials are invalid (wrong username or wrong password), the `AuthLoginServlet` redirects back to the *Login-Page*.

To log out, the `AuthLogoutServlet` exists. If a GET request is made from an active user, it invalidates the `HTTPSession` and redirects the user to the *Login-Page*.

3.3.2 Login

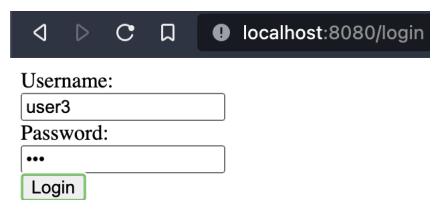


Figure 4: The login HTML form

Figure 4 shows the *Login-Page*. It consists of a single HTML form that sends a POST request to the `AuthLoginServlet` (introduced before in Section 3.3.1) consisting of the username and the corresponding password.

3.3.3 User

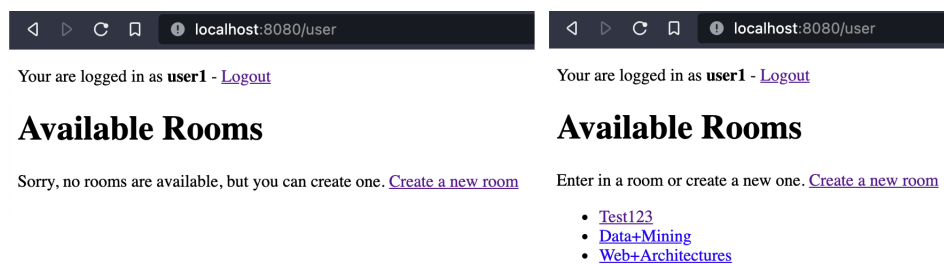


Figure 5: The *User-Page* with the empty message and available rooms

Figure 5 shows the *User-Page*. It is shown that the *User-Page* lists all available rooms, otherwise a message if no rooms exist. The view reads all available rooms from the `RoomStore` (introduced in Section 3.2) and uses a `for` loop to list the rooms as clickable links in a `ul` element, which is shown in Listing 3.


```

1 <%@ page import="java.util.Set" %>
2 <%@ page import="it.unitn.disi.webarch.chat.helper.RoomStore" %>
3 <%@ page import="it.unitn.disi.webarch.chat.models.room.Room" %>
4 <%
5     Set<Room> rooms = RoomStore.getInstance().getAll();
6 %>
7 ...
8 <body>
9 <ul>
10 <% for(Room room: rooms) { %>
11 <li><a href="<% request.getContextPath(); %>/room/<%= room.
    getName() %>"><%= room.getName() %></a></li>
12 <% } %>
13 </ul>
14 </body>

```

Listing 3: List all available rooms

3.3.4 Create Room

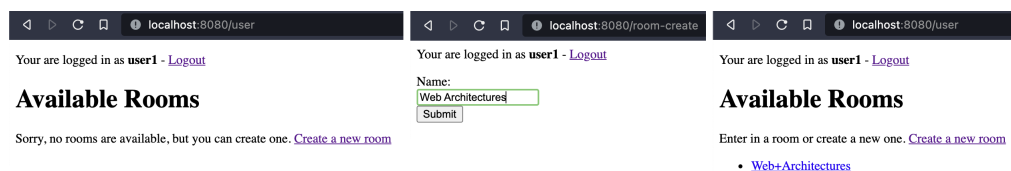


Figure 6: Procedure of creating a new room

The `RoomCreateServlet` includes a view that presents an HTML form to the user. The user can set a room name and submit the form as a POST request to itself. After a POST request has been received, the `RoomCreateServlet` creates a new instance of the `Room` model and adds the `Room` to the `RoomStore`. After a room has been created successfully, the user will be redirected to the *User-Page*. This procedure is illustrated in Figure 6, where the room *Web Architectures* is being created. Additionally, the `RoomCreateServlet` encodes the name of a room using `URLEncoder.encode(ROOM_NAME, "UTF-8")`. This is necessary because the user is allowed to use special characters like whitespaces in the room name. Figure 6 also shows, how the room name has been encoded. The example shows that the name *Web Architectures* has been encoded to *Web+Architectures*.

3.3.5 Room

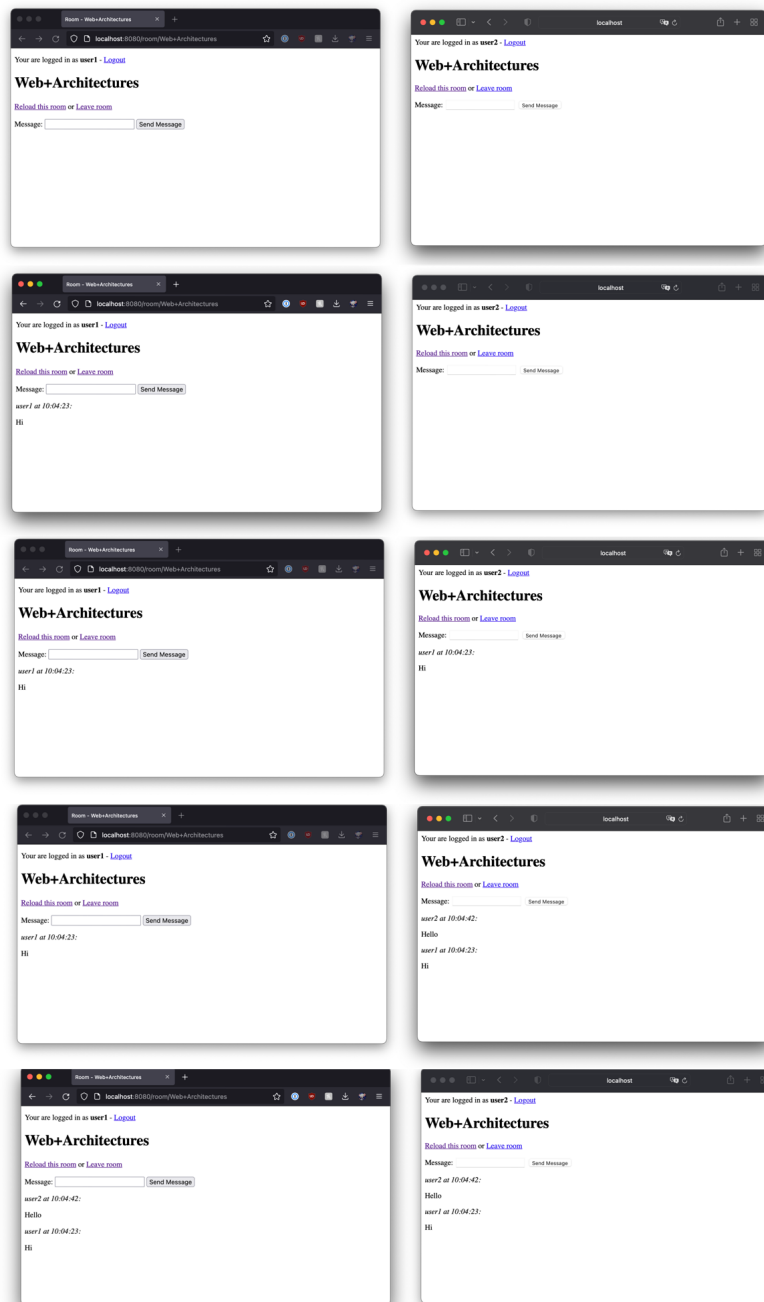


Figure 7: Chat interaction between two different users in a room

In a *Room-Page*, multiple users can chat with each other, which is shown in Figure 7. The `RoomServlet` owns a view called `Room.jsp`, which shows an HTML form, and lists all messages of the active Room.

The active room is set as a Java Bean in the request. Therefore, the JSP view can access the model of the active room via `jsp:getProperty`. Then, all messages which belong to the active room can be received via the `getMessages()` method of a `Room` model and can be listed in a `ul` element using a `for` loop. Listing 4 shows the implementation, how messages are listed in the room view.

```

1 <%@ page import="java.util.List" %>
2 <%@ page import="it.unitn.disi.webarch.chat.models.room.Message" %
  >
3 <jsp:useBean id="activeRoom" class="it.unitn.disi.webarch.chat.
  models.room.Room" scope="request" />
4 ...
5 <body>
6 <%
7 List<Message> messages = activeRoom.getMessage();
8 for(Message message: messages) {
9 %>
10 <div>
11 <p><em><%= message.getUser() %> at <%= message.
  getFormattedDate() %>:</em></p>
12 <p><%= message.getMessage() %></p>
13 </div>
14 <% } %>
15 </body>

```

Listing 4: List messages for a specific room

The HTML form sends a POST request to itself. The request consists of an attribute called `message`, which is the message text of the user. After a POST request has been made, the `RoomServlet` constructs a new `Message` model, using the message text received from the POST request, and the username of the active user which is saved in the HTTP session. After that, the newly created `Message` model is added to the active `Room` model via the `addMessage` method. Finally, the page gets reloaded using the `doGet` method of the servlet. If the request is not valid (no room requested), the `RoomServlet` responds with a 400 - Bad Request HTTP error. Otherwise, if the requested room does not exist, the `RoomServlet` responds with a 404 - Not Found HTTP error.

To update newly created messages, the view reloads itself every 15 seconds using `<meta http-equiv="refresh" content="15">`.

3.3.6 Admin

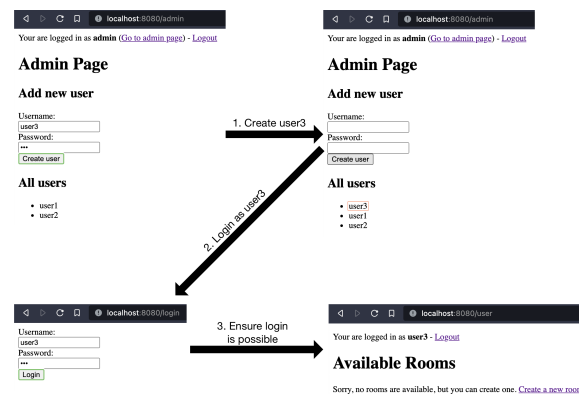


Figure 8: Creation-process of *user3*

Figure 8 illustrates the successful creation process of a user. The admin can fill out the HTML form of the *Admin-Page*. After clicking submit, it will send the data (username and password) via a POST request to itself. The *AdminServlet*, validates the POST request attributes and checks if the user already exists. If the user does not exist, the user will be added via the *addUser* method of the *UserStore*. As mentioned in Section 3.2, the *UserStore* will write the user to the *users.txt* file. To check if the given credentials are valid, the *AdminServlet* checks if the given username and password are not null. Otherwise, the *AdminServlet* will respond with a 400 - Bad Request HTTP error. Additionally, the *AdminServlet* checks if the length of the username and password is bigger than 0 and if the username does not equal *admin*. If one condition is not true, the *AdminServlet* executes the *doGet* method to reload itself.

3.4 Banner

As mentioned in Section ??, a *Banner* is included on each page. The *Banner* only consists of a JSP file. It shows the name of the active user and a link to the logout route. Additionally, if the active user is the *admin* user, the *Banner* presents a link to the *Admin-Page*. Figure 9 shows the different *Banner* for a normal user, and the *admin* user for the *User-Page*.

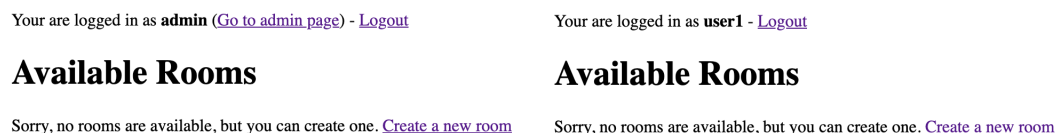


Figure 9: Different Banner for different users

The Banner gets included in a view using `<jsp:include page="../../Banner.jsp"/>`. To get the active user, it reads the user as a Java Bean from the current session.

3.5 Filters

Because users have to authenticate to access rooms and write messages, filters are needed to check if a user is authenticated. Additionally, as mentioned in Section ??, only the *admin* user is allowed to access the *Admin-Page*. Therefore, an additional filter is needed which ensures this behavior. To conclude these requirements, the following filters are implemented:

- `AuthFilter`
- `AdminFilter`

3.5.1 Authentication

As mentioned before, the chat system is only available for authenticated users. Therefore, a filter called `AuthFilter` checks if a request to a route, described in Section ??, has been made by an authenticated user. The `AuthFilter` checks if the session attribute `is_authenticated` is set to `true`. This attribute is set by the `AuthLoginServlet` after a successful login, described in Section 3.3.1. Otherwise, if a session does not exist or the `is_authenticated` attribute is not set to `true`, the `AuthFilter` redirects the client to the *Login-Page*. Figure 10 shows the login HTML form after an unauthorized request has been made to the *User-Page*.

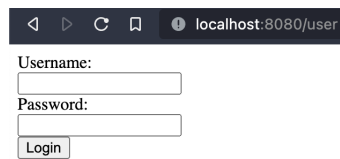
A screenshot of a web browser window. The address bar shows 'localhost:8080/user'. The page content includes a 'Username:' label followed by a text input field, a 'Password:' label followed by a text input field, and a 'Login' button below them.

Figure 10: Redirect to the *Login-Page*

3.5.2 Admin

The `AdminFilter` checks if a request has been made to the *Admin-Page*. If the request was made by an authenticated user, and the name of the user is equal to *admin*, the filter redirects the user to the *Admin-Page*. Otherwise, as illustrated in Figure 11, the `AdminFilter` sends a 403 - Forbidden HTTP error response.

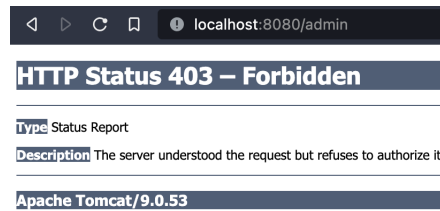


Figure 11: 403 - Forbidden HTTP error response