



*Università di Trento*

*Web Architectures*

---

## Assignment 2

---

*Author:*

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

October 17, 2021

# 1 Introduction

The task of this assignment is to implement a chat system using Java Servlets. Different users can login at the same time, create different rooms, and chat in a room with each other. In addition, an admin user can login as well, and create new user from the *admin-page*.

## 1.1 Problem Statement

For this assignment, it is necessary to implement a custom storage system, because using a Database is not part of the assignment. Therefore, a custom storage system has to be implemented, which is able to store users, and rooms, for as long as the HTTP server is running. The users are supposed to be stored in a text file. Additionally, an authentication system has to be implemented. It should restrict unauthorized users to use the chat. If a request of an unauthorized user has been made, the user should be forwarded to the login page first. However, there exists a special user called *admin*, which is allowed to request the *admin-page* where user can be created.

## 1.2 Domain Description

Given the above mentioned problem statement, the following steps have to be implemented:

1. Create a temporal object storage
2. Implement an authentication system
3. Develop controller and views for all needed routes

## 2 Conceptual Design

The conceptual design which is introduced below, is based on the problem statement introduced in Section 1.1.

### 2.1 Data Storage

As mentioned in the problem statement, the user has the ability to create multiple rooms. When the user opens a room, the user can send messages inside the selected room. A room should only exist as long as the server is running. Therefore, a store system has to be implemented to store rooms and messages temporarily.

FIG XY shows the design of a *RoomStore*. The *RoomStore* has the ability to save rooms and messages exists within the room.

Additionally, the *admin* user has the ability to create new users. Users are saved consistently in a text file and should be available after the server has restarted. Given this, a *UserStore* has to be implemented, which is able to read and write user object from a text file.

### 2.2 Routes

The following pages are available in the chat system:

- *Login-page*, route: `http://localhost/login`
- *User-page*, route: `http://localhost/user`
- *Room-page*, route: `http://localhost/room/ROOM_NAME`
- *Create-room-page*, route: `http://localhost/room-create`
- *Admin-page*, route: `http://localhost/admin`

It is important to mention, that each page, except the *Login-page*, includes a *Banner* where the user can see a logout link and the username. If the active user is the *admin* user, there will also be an *Admin-page* link available.

#### 2.2.1 Login-Page

At the *Login-page*, the user sees an HTML-form where the user can insert a username, and a password. If the credentials are correct, the user will be forwarded to the *User-page* after the form has been submitted.

#### 2.2.2 User-Page

After the user has logged in successfully, the user will be forwarded to the *User-page*. There, the user sees all available rooms which are clickable, a link to the *Create-room-page*, and the above mentioned *Banner*.

### 2.2.3 Room-Page

When the user clicks on the name of a room, the user opens the *Room-page* where authenticated users can chat with each other.

### 2.2.4 Create-Room-Page

At the *Create-room-page*, an authenticated user can create a new room through an HTML-form. After the form has been sent successfully, the user will be redirected back to the *User-page*, where the user sees the newly created room.

### 2.2.5 Admin-Page

If the active user is the *admin* user, the user can access the *Admin-page*. There, the *admin* user sees the name of all available users, and create new users through an HTML-form which will be saved to the text file mentioned in Section 2.1.

## 2.3 Authentication

User needs to authenticate first to be able to access the above mentioned pages of the chat system, except the *Login-page*. Therefore, a filter has to be implemented which is responsible to check the authentication status of the user, every time a request has been made. The authentication status of the user is saved as an attribute in the session. The session will be created, after a user has been logged-in successfully.

As mentioned in Section 2.2, a special page called *Admin-page* exists, which can only be accessed by the *admin* user. Therefore, an additional filter is needed to check if a request has been made to the *Admin-page* route, and check if the user is authenticated and if the user is the *admin* user. Otherwise, the server responds with an HTTP error.

### 3 Implementation

This section explains the implementation based on the conceptual design introduced in Section 2. As mentioned in PROBLM, the MVC architecture is used to implement this application. 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 3.4).

#### 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 models in JSP files.

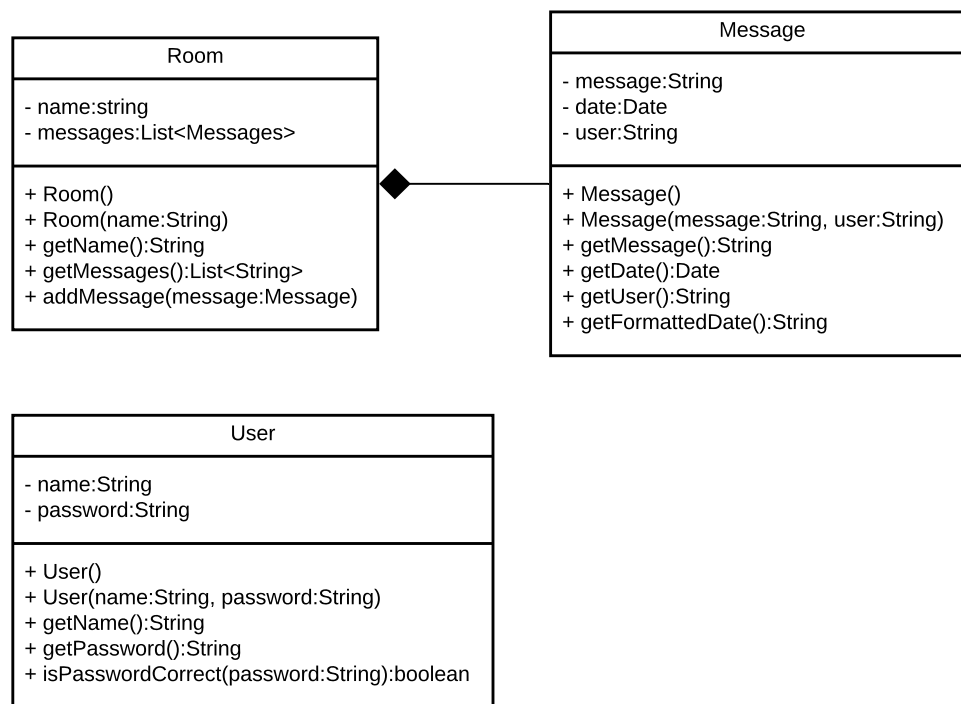


Figure 1: Used models for the chat system

Figure 1 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.

## 3.2 Object Stores

As being mentioned in SEC DESIGN STORAGE, a custom store is needed to save the in Section 3.1 introduced models.

Therefore, the following stores are being implemented:

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

Because a user and room can only exists once, a **Set**<sup>1</sup> 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 to 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 this **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, an the objects are saved to memory. Except for the **UserStore**, which reads all **User** models from a text file, and write **Users** to a text file. Therefore, when the server restarts, all previously added **User** modes are available again.

## 3.3 Servlets

Fig XY shows the architecture of the in SEC DESIGN introduced routes. To implement this architecture, the following Servlets are implemented:

- **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**.

---

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

### 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 exist, 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 3.4.1. After that, the `AuthLoginServlet` redirects the user to the *User-Page*. 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 logout, the `AuthLogoutServlet` exist. If a `GET` request is made from an active user, it invalidates the `HTTPSession` and redirects to the *Login-Page*.

### 3.3.2 Login

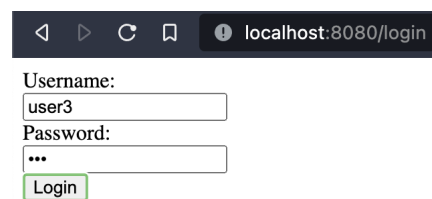


Figure 2: The login HTML form

Figure 2 shows the *Login-Page*. It consists of a single HTML form which 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

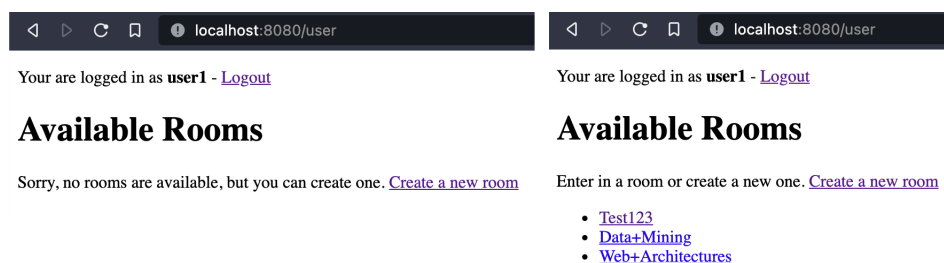


Figure 3: Creation-process of *user3*

Figure 3 shows the *User-Page*. It is shown that the *User-Page* lists all available rooms, otherwise a message if no rooms exists. The view reads all available rooms from the `RoomStore` (introduced in SEC ROOMSTORE) and uses a `for` loop to list the rooms as clickable links in a `ul` element, which is shown in Listing 1.

```

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 1: List all available rooms

### 3.3.4 Room

In a *Room-Page*, multiple users can chat with each other, which is shown in Figure 4. 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 belongs 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 2 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.getMessages();
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 2: 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` responses with a `400 - Bad Request` HTTP error. Otherwise, if the requested room does not exists, the `RoomServlet` responses 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">`.

### 3.3.5 Admin

Figure 5 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 the `POST` request attributes and checks if the user already exist. If the user does not exist, the user will be add via the `addUser` method of the `UserStore`. As mentioned in SEC USERSTORE, 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 response 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 Filters

### 3.4.1 Authentication

### 3.4.2 Admin

## 4 Conclusion

# A Part 1

## A.1 Implementation

### A.1.1 StringReverser

```
1 package it.unitn.disi.webarch.assignment1;
2
3 public class StringReverser {
4
5     public static void main(String[] args) {
6         if (args.length >= 1) {
7             String text = args[0];
8
9             if (text != null) {
10                 String reversedString = reverseString(text);
11                 System.out.println(reversedString);
12             }
13         } else {
14             System.out.println("A string to reverse is required.")
15         };
16         System.exit(0);
17     }
18
19     /**
20      * This method reverses the given string.
21      *
22      * @param text
23      * @return Reversed text
24      */
25     private static String reverseString(String text) {
26         StringBuilder stringBuilder = new StringBuilder(text);
27         stringBuilder.reverse();
28         return stringBuilder.toString();
29     }
30
31 }
```

Listing 3: StringReverser class implementation

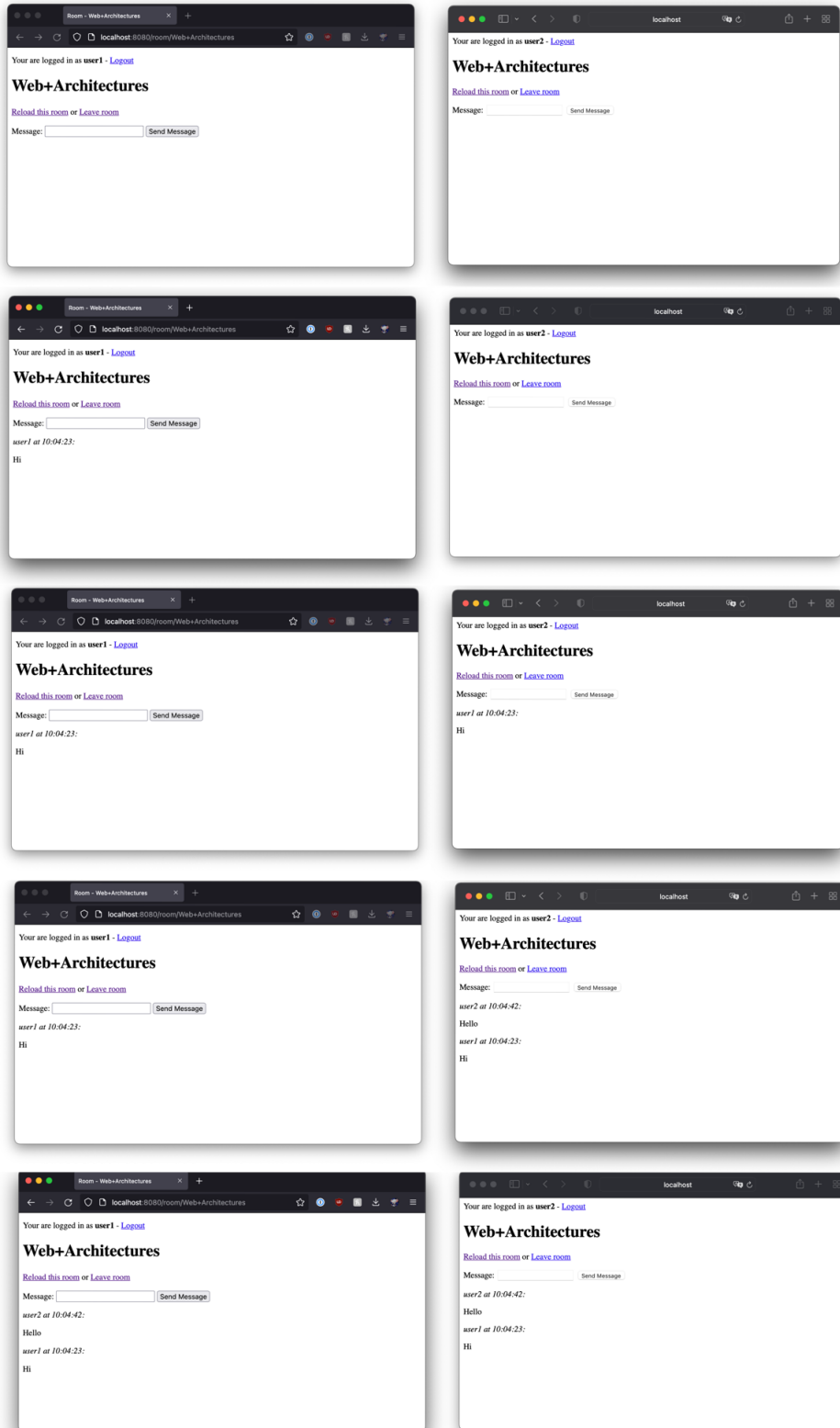


Figure 4: Creation-process of *user3*

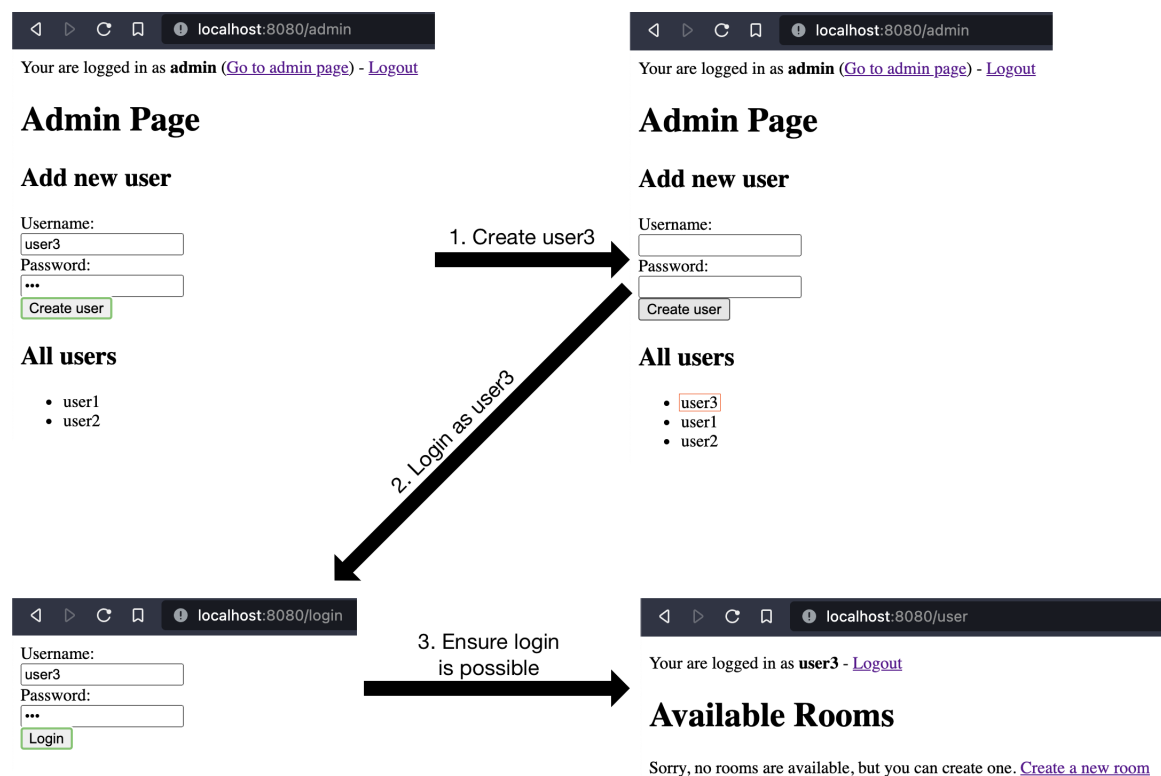


Figure 5: Creation-process of *user3*