Università di Trento

Web Architectures

Assignment 2

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

October 16, 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, fir as long as the HTTP server is running. The users are supposed to be stored in a text file. Additionally, a authentication system has to 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 which 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 send 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 user, 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* exist, 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.
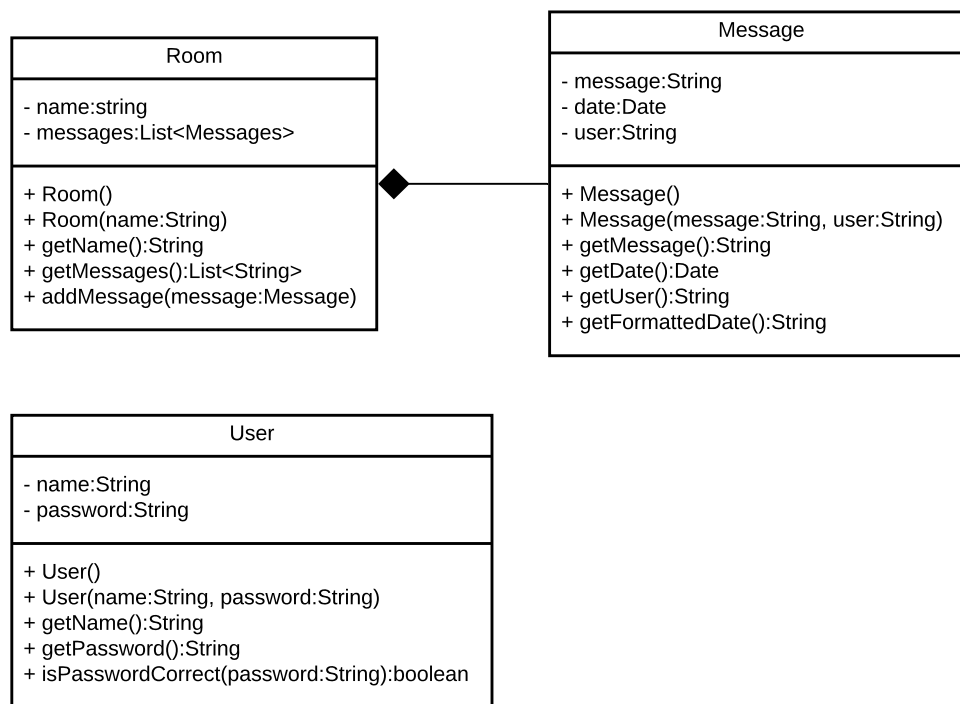

Figure 1: Used models for the chat system

Figure 1 shows all models used in the implementation of the chat system. A `Room` models 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 send 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 send. 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`[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 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

3.3.1 Authentication

3.3.2 Login

3.3.3 User

3.3.4 Room

3.3.5 Admin

3.4 Filters

3.4.1 Authentication

3.4.2 Admin

---

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

4 Conclusion

A Part 1

A.1 Implementation

A.1.1 StringReverser

```java
package it.unitn.disi.webarch.assignment1;

public class StringReverser {

    public static void main(String[] args) {
        if (args.length >= 1) {
            String text = args[0];

            if (text != null) {
                String reversedString = reverseString(text);
                System.out.println(reversedString);
            }
        } else {
            System.out.println("A string to reverse is required.");
            System.exit(0);
        }
    }

    /**
     * This method reverses the given string.
     *
     * @param text
     * @return Reversed text
     */
    private static String reverseString(String text) {
        StringBuilder stringBuilder = new StringBuilder(text);
        stringBuilder.reverse();
        return stringBuilder.toString();
    }

}
```

Listing 1: StringReverser class implementation