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. In this chat system, users can create different rooms where multiple users can send messages to each other.

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, rooms, and messages. It is important to mention, that the life-time of rooms and messages are are bound to the webserver. This means, if the webserver stops running, all rooms and messages are being removed from the memory. However, users should be read from a text file. A special user called admin can write new users to the text file. Additionally, users can authenticate using a username and a password. Otherwise, unauthorized requests are forbidden. Another requirement, is to implement this chat system using the Model-View-Controller (MVC) pattern.

2 Conceptual Design

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

2.1 Data Storage

As mentioned in Section 1, a custom storage system has to be implemented. 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, the custom storage system should be able to store rooms, and messages associated to a specific 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 can read and write user objects from/to a text file. If the webserver restarts, all newly created users need to be available again.

2.2 Routes

The following pages need to be 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

Users need to authenticate first to be able to access the above mentioned pages of the chat system, except the *Login-Page*. Therefore, a filter is 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 Section 1, the MVC architecture is used to implement this application. Figure 1 present 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).
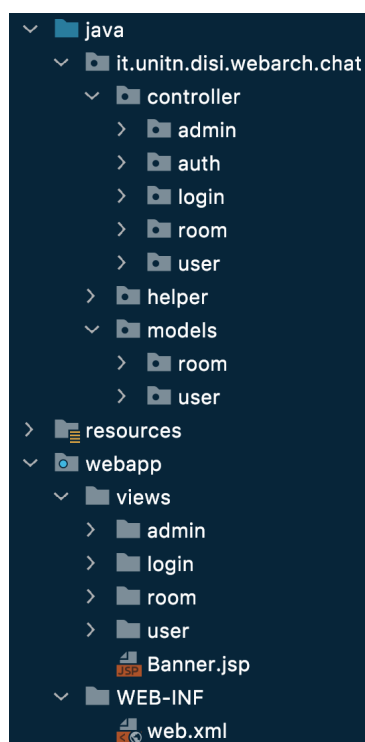

Figure 1: Project folder structure

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 models in `JSP` files. Figure 2 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.
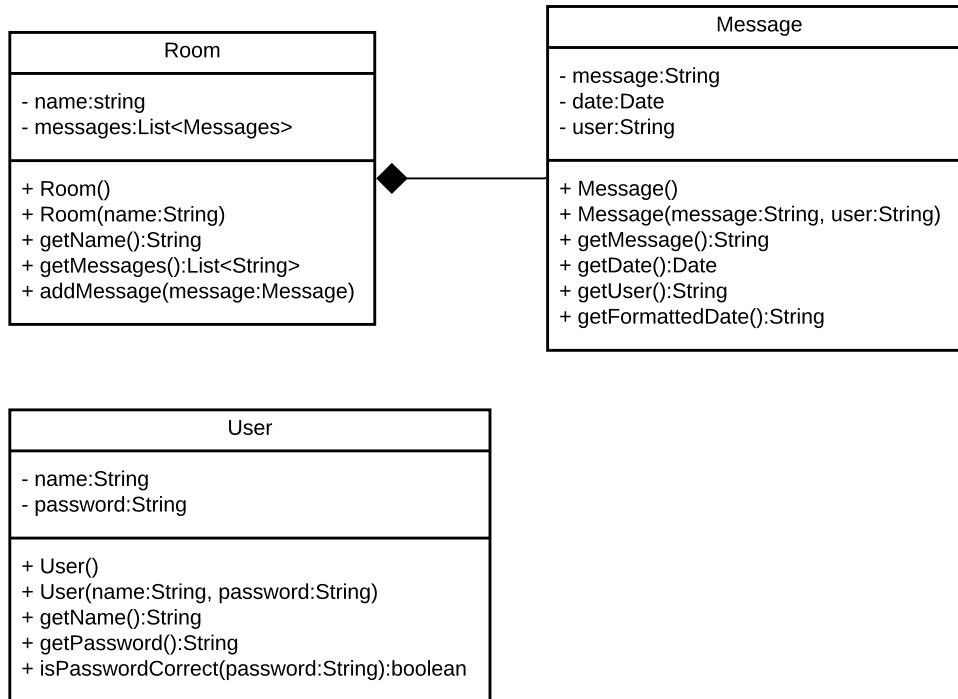
Room

- name:string
- messages:List<Messages>

+ Room()
+ Room(name:String)
+ getName():String
+ getMessages():List<String>
+ addMessage(message:Message)

Message

- message:String
- date:Date
- user:String

+ Message()
+ Message(message:String, user:String)
+ getMessage():String
+ getDate():Date
+ getUser():String
+ getFormattedDate():String

User

- name:String
- password:String

+ User()
+ User(name:String, password:String)
+ getName():String
+ getPassword():String
+ isPasswordCorrect(password:String):boolean

Figure 2: Used models for the chat system

3.2   Object Stores

As being mentioned in Section 2.1, 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 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 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 shout down, all rooms and messages are being removed from the memory. Except for the `UserStore`, which reads all `User`

---

[1]Set (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 usage is shown in Listing 1.

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

Listing 1: Example usage of the singleton pattern

3.3 Servlets

Each route introduced in Section 2.2, 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.

```
this.getServletContext()
  .getRequestDispatcher("/views/room/RoomCreate.jsp")
  .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 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 3.5.1. 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 the user to the *Login-Page*.
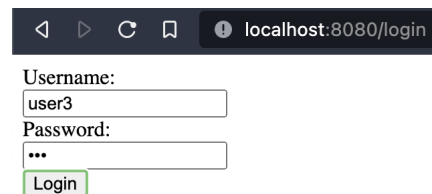
3.3.2 Login

Figure 3: The login HTML form

Figure 3 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.
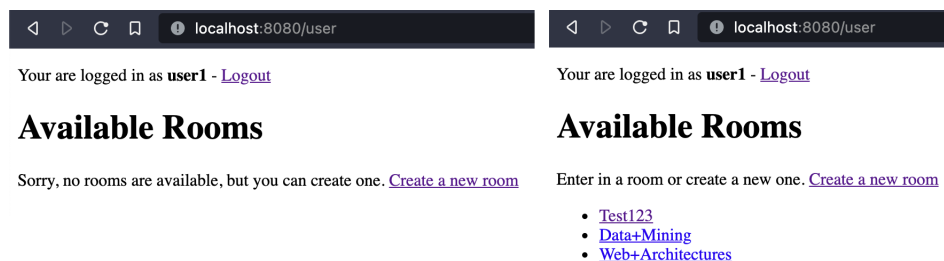
3.3.3 User

Figure 4: The *User-Page* with the empty message and available rooms

Figure 4 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 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>
```
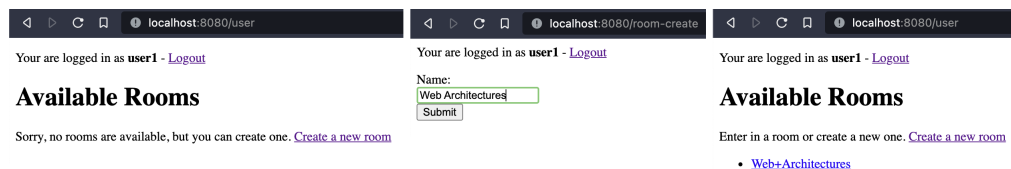
Listing 3: List all available rooms

3.3.4 Create Room

Figure 5: Procedure of creating a new room

The `RoomCreateServlet` includes a view which 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 5, 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 5 also shows, how the room name has been encoded. The example shows that the name *Web Architectures* has been encoded to *Web+Architectures*.
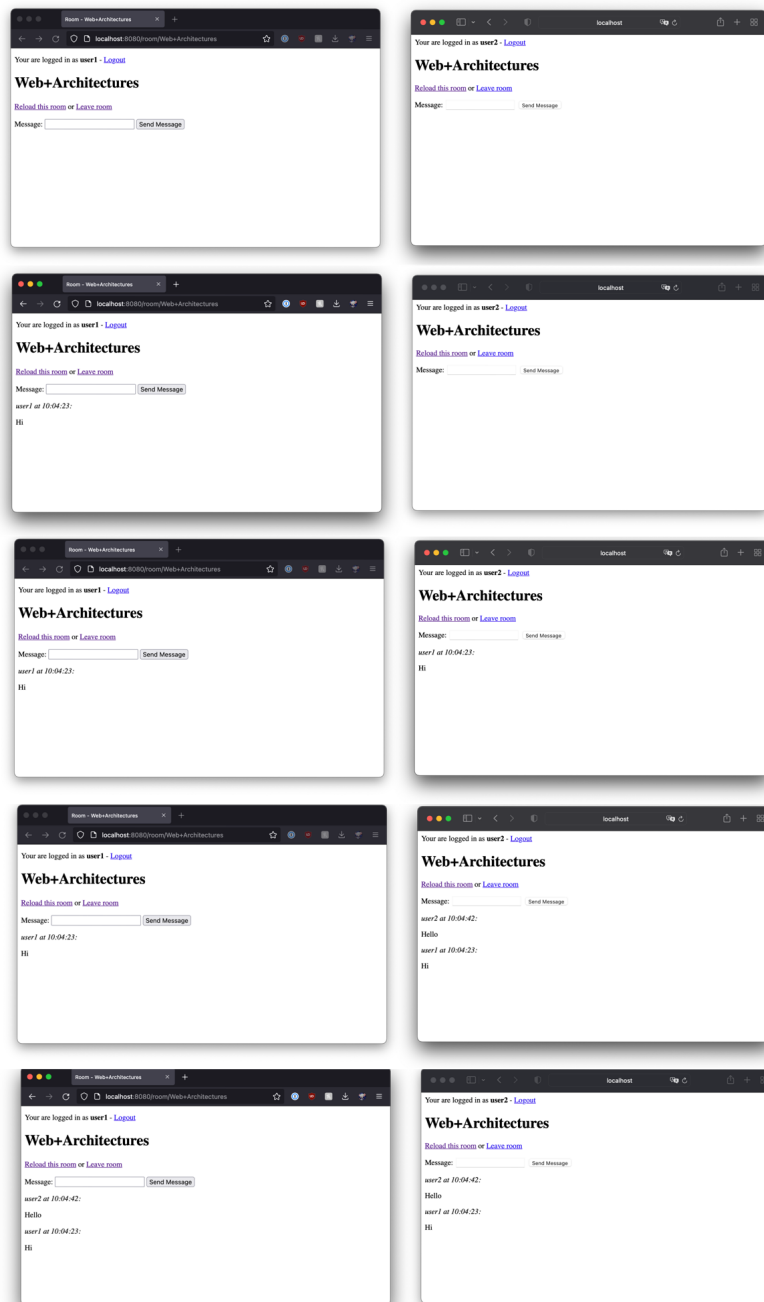
3.3.5   Room

Figure 6: 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 6. 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 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.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 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 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">.
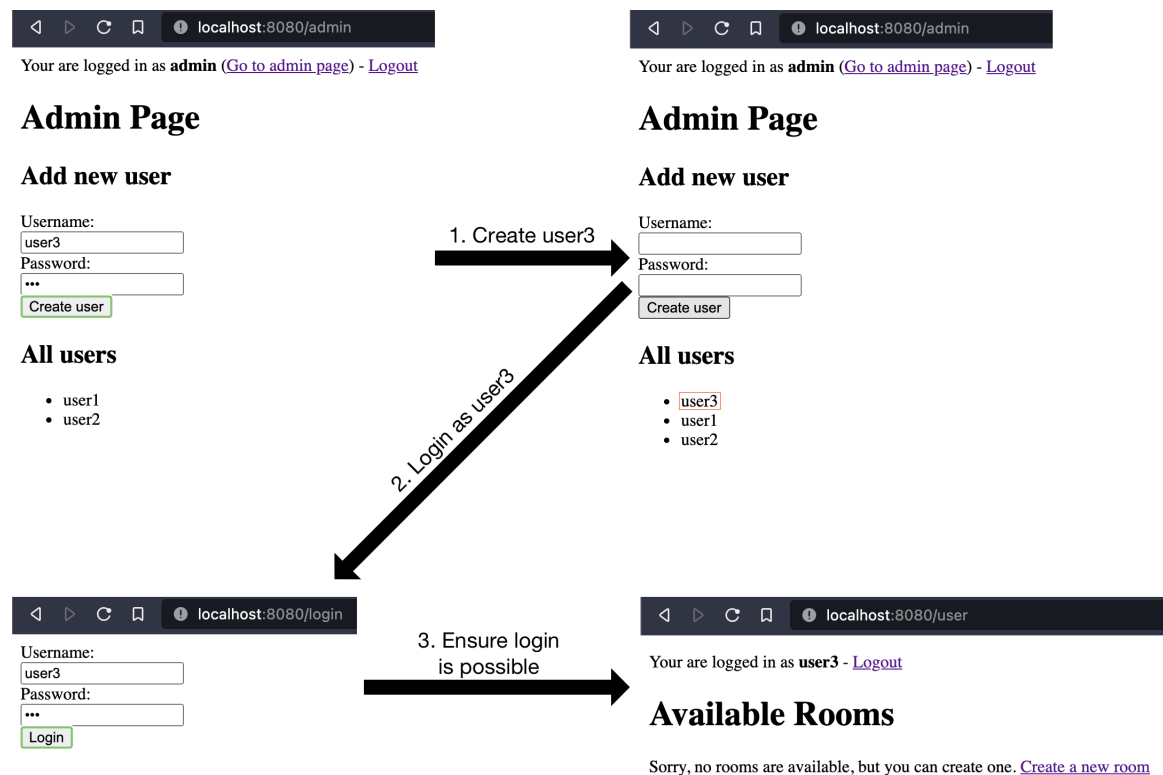
3.3.6 Admin

Figure 7: Creation-process of *user3*

Figure 7 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 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 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 Banner

As mentioned in Section 2.2, a *Banner* is included in 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 link to the *Admin-Page*. Figure 8 shows the different *Banner* for a normal user, and the *admin* user for the *User-Page*.

Your are logged in as **user1** - Logout

Available Rooms

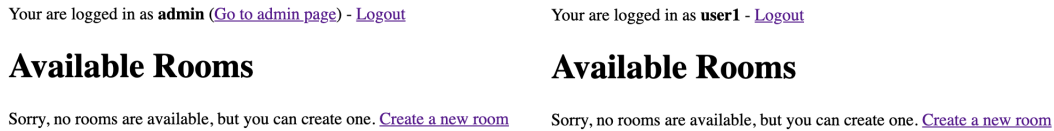Sorry, no rooms are available, but you can create one. Create a new room

Figure 8: Different Banner for different users

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 mention in Section 2.2.5, only the *admin* user is allowed to access the *Admin-Page*. Therefore, an additional filter is needed which ensures this behaviour. To conclude this 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 2.2, 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 9 shows the login HTML form after an unauthorized request has been made to the *User-Page*.

localhost:8080/user

Username:

Password:

Login

Figure 9: 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 10, the `AdminFilter` sends a `403 - Forbidden` HTTP error response.
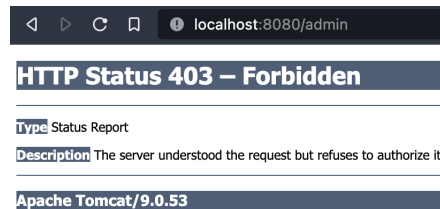

Figure 10: `403 - Forbidden` HTTP error response