UNIVERSITY OF TRENTO

ASSIGNMENT 2

Vinci Nicolò 220229

nicolo.vinci@studenti.unitn.it

Web Architectures 2021/2022

03 October 2021

Contents

1 Introduction

A messaging web app need to be developed. The web app is based on rooms where a user can enter and send message. After that, any user who enters in the same room can see the message sent before. A user can also decide to create a room. There must be a login page where a user must authenticate in order to access to the available rooms. There is also an admin account who can check the actual users and add new user. However, only the admin is allowed to perform the previous actions. Rooms and messages are not persistent, it means that when the web app shuts down, everything is lost. Instead, credentials of authenticated users need to be persistent in order to check the provided credentials by the login page. To sum up, a room is a channel where any user can write messages and see messages written by other users.

2 Description

The web app has been developed following the Model View Controller (MVC) pattern:

- Model: JavaBean
- View: JavaServer Page (JSP)
- Controller: Java Servlet

Two JavaBean(s) have been designed:

- User: it is stored in *Session* and contains the username.
- Room: it is stored in *ServletContext* and contains *HashMap<UUID, Room>*. The *Room* object has a title attribute for the room name and an *ArrayList<Message>*. The *Message* object has a writtenBy attribute for the sender message, the text attribute for the content message and the date attribute to know when the message is sent.

Then, different JSPs have been developed:

- adminPage.jsp: it shows the list of users and allows to create a new one.
- banner.jsp: it shows a logout button and the actual username of the user. It is included in all the other JSPs except for the login.jsp and inputError.jsp.
- inputError.jsp: it shows an error paragraph. It is included in login.jsp and adminPage.jsp to check if any text input is not empty.
- login.jsp: it shows a form where the user can insert credentials.
- roomPage.jsp: it shows the actual messages in the room in chronological order (newest first). It allows to send a new message through the message input and the *Send* button. The user can reload manually the page thanks to the *Reload* button. The page refreshes itself automatically every 15 seconds. The user can go back to the *userPage.jsp* through a link.
- userPage.jsp: it shows the actual rooms if there are. It allows to create a new room inserting the room name and clicking on the button *Create*.

After that, there are some Java Servlets to satisfy the various http requests:

- AdminServlet: it handles the *post* request from *adminPage.jsp* and creates a new user.
- Login: it handles the *post* request from *login.jsp* and authenticates the user. It can authenticate the user as admin or plain user. It handles the *get* request from the *index.html*.

- Logout: it handles the *get* request from *banner.jsp* and redirects the user to *login.jsp*.
- RoomServlet: it handles the *post* request from *roomPage.jsp* to create a new Message object. It also handles the *get* request from the clickable link in the *userPage.jsp* or from the automatic *roomPage.jsp* refresh.
- UserServlet: it handles the *post* request from *userPage.jsp* to create a new object Room.

A Filter has been developed:

- AuthFilter: it is applied to any url that matches the pattern */user/**. It checks if the session is non null. It checks also if the user is authenticated inspecting the attribute *authenticated* stored in the session.

There is also Listener:

- FileListener: when the *ServletContext* is created, it reads the *authentication.txt* file through the method *contextInitialized*. So, every credentials stored in the file are mapped into a Java *HashMap<String, String>* and stored in the *ServletContext*. When the web app is stopped, the *ServletContext* is destroyed. Hence, the method *contextDestroyed* is called and the file *authentication.txt* is updated with every credentials stored in the previous HashMap.

A Java enumeration has been developed in the *utils* package:

- State: it is useful to have three different authentication states. It is used in *login* Servlet to check the input credentials. A user can be ADMIN, PLAIN_USER or UNAUTHENTICATED.

In the *resources* has been placed a file:

- authentication.txt: it contains the actual user credentials. Each line has *username-password* of the user. It is updated when the web app is stopped by the *FileListener*.

At the beginning, the web app shows a static page:

- index.html: it is the welcome static page. The user can go to the *login.jsp* from here.

A summary MVC model schema applied for the web app is reported at figure 1.

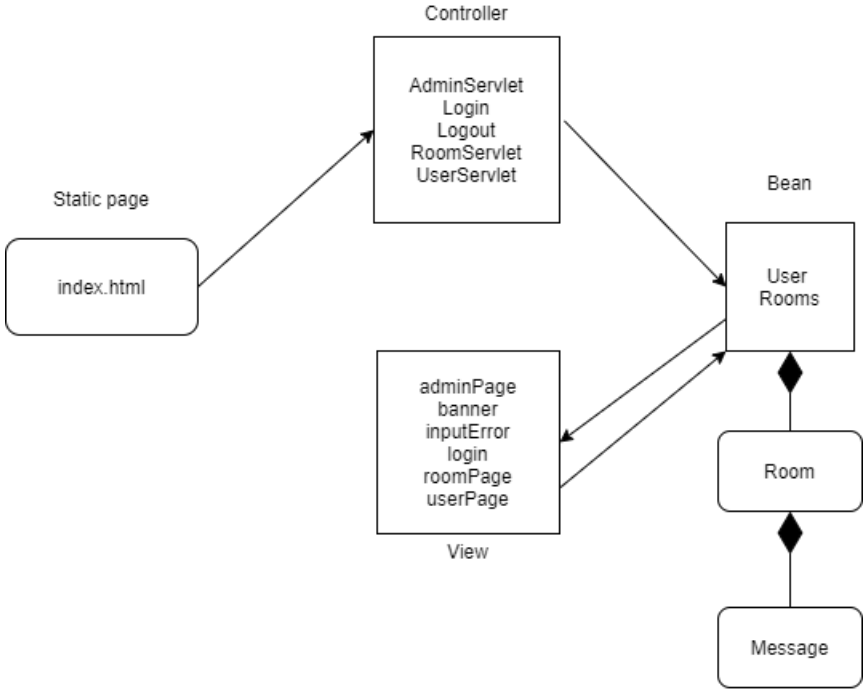

Figure 1: MVC schema

3 App

The web app functioning is described in this section.

3.1 Deployment

First of all, the web app needs to be deployed in the Tomcat server. The folder generated in Intellij has been copied in the Tomcat *webapps* folder.

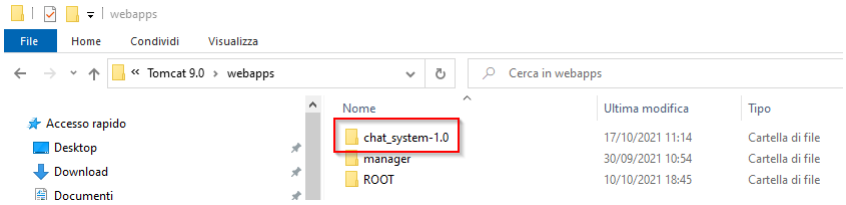

Figure 2: Webapps Tomcat folder

Now, the web app is visible in the Tomcat manager page. It can be stopped, reloaded and undeployed from the manager page.

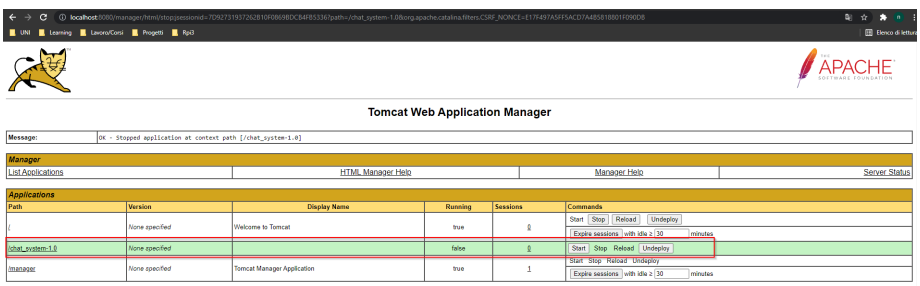

Figure 3: Tomcat manager page

The web app can be started clicking on the link shown in the figure 3.

3.2 Start web app and Login

When the web app starts, it shows the static page *index.html*.

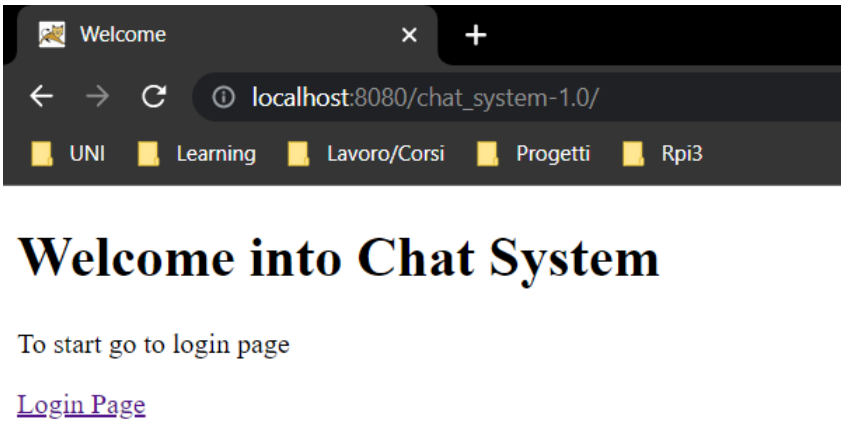

Figure 4: Static index page

The method *contextInitialized* in the *FileListener* is also called. Here the *authentication.txt* file is read and every credentials are stored in a *HashMap<String, String>* called *credentials* in the *ServletContext*. The path for the *authentication.txt* file has been defined in the web.xml as *contex-param*.

```
<context-param>
    <param-name>PathToAuthentication</param-name>
    <param-value>WEB-INF/classes/authentication.txt</param-value>
</context-param>
```

Figure 5: *authentication.txt* path in web.xml

When the user clicks to the link *Login Page*, a *get* request is performed calling the *Login* Servlet. The request is handled by the *doGet* method. An *error* attribute is set to false in the request. After that, the request is forwarded to the *login.jsp*.

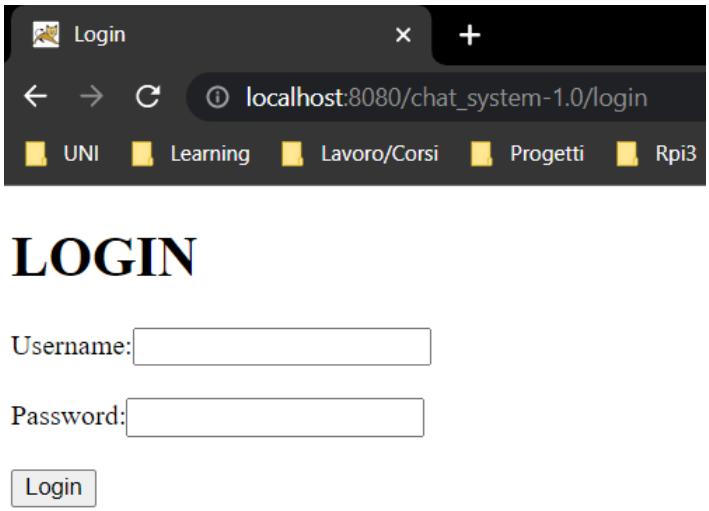

Figure 6: login.jsp

The *login.jsp* shows a form to insert credentials and get authenticated. It also includes the *inputError.jsp* that shows a paragraph error if the request attribute *error* is set to true. The *inputError.jsp* is shown if the user provides invalid credentials or if he leaves blank the username or password field. In the figure 7 is reported one of the two cases described before.

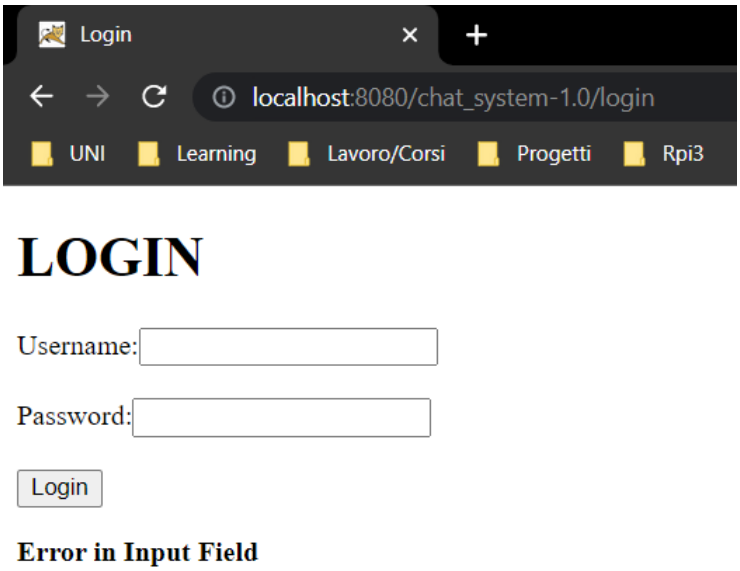

Figure 7: errorInput.jsp

The user can authenticate inserting username and password and clicking the *Login* button. A *post* request is handled by the *Login* Servlet in the *doPost* method when the button is clicked. An session is created at the beginning of the method. The session will be expired after 30 minutes. Credentials are checked retrieving the *credentials* HashMap stored in the *ServletContext*. Then, the user can be ADMIN, PLAIN_USER or UNAUTHENTICATED. If the user is PLAIN_USER, he will be forwarded to the *userPage.jsp* setting the *authenticated* attribute in session to true. Moreover, a user bean is initialized in the session scope setting the username and a rooms bean is initialized in the servlet context scope.

Figure 8: userPage.jsp

If the user is ADMIN, he will be forwarded to the *adminPage.jsp* setting the *authenticated* attribute in session to true. The request attribute *error* is also set to false for the *inputError.jsp*. Moreover, a user bean is initialized in the session scope setting the username and a rooms bean is initialized in the servlet context scope.

Figure 9: adminPage.jsp

The password for the admin account is stored in the web.xml file as *init-param* for the *Login* Servlet.

```
<servlet>
    <servlet-name>Login</servlet-name>
    <servlet-class>com.example.chat_system.controller.Login</servlet-class>
    <init-param>
        <param-name>AdminPassword</param-name>
        <param-value>password</param-value>
    </init-param>
</servlet>
<servlet-mapping>
    <servlet-name>Login</servlet-name>
    <url-pattern>/login</url-pattern>
</servlet-mapping>
```

Figure 10: Password Admin in web.xml

If the user is UNAUTHENTICATED, the session attribute *authenticated* is set to true and the request attribute *error* is set to false. Then, the request is forwarded to the *login.jsp*.

3.3 Plain user

The *userPage.jsp* is shown to the PLAIN_USER. The user can create a new room inserting the room name and clicking on the *Create* button. When button is clicked, a *post* request is performed to the *UserServlet* in the *doPost* method. The Servlet retrieves the rooms bean and add to the *HashMap<UUID, Room>* a new Room object. A Room object is identified by the unique UUID object used as key in the HashMap.

Figure 11: Create new Room

Then, the user can enter in the room clicking the name. When it happens, a *get* request is performed to the *RoomServlet* passing as parameter in the request the room UUID. The *doGet* method of the *RoomServlet* retrieves the room object and the room title from the rooms bean. It also stores in the session the room UUID and the title, because it will be useful for the *userPage.jsp* retrieves them without passing them in the request attribute every time. At the end, the request is forwarded to the *userPage.jsp*

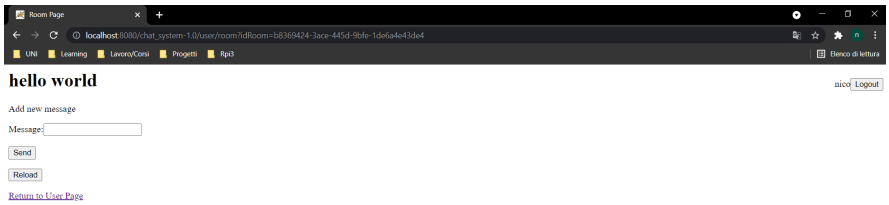

Figure 12: Room

In the *userPage.jsp*, a user can add a new message inserting the text in the input and clicking the *Send button*. When the button is clicked, a *post* request is performed to the *RoomServlet*. It is handled in the *doPost* method. It retrieves the room beans and the right room exploiting the UUID stored in the session parameter called *idRoomInSession*. Then, it creates a new message object and add it in the *ArrayList<Message>* contained by the room object. At the end, the request is forwarded to the *roomPage.jsp* again. It will be able to show messages following a reversed chronological order calling the *getOrderedMessage* method.

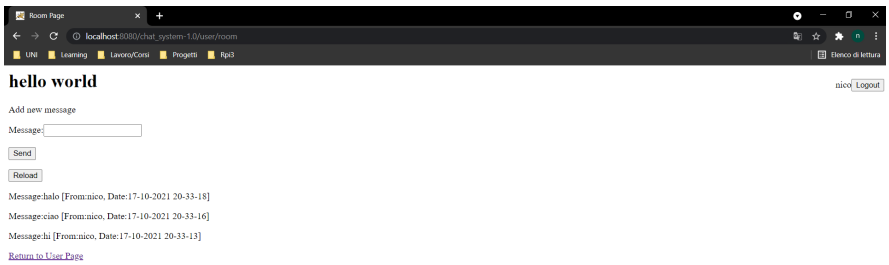

Figure 13: Ordered messages

The user can reload manually the *roomPage.jsp* with the *Reload* button and he can also returns to the *userPage.jsp* with the link *Return to User Page*. However, the *roomPage.jsp* refreshes itself every 15 seconds thanks to the meta tag *refresh*.

```jsp
<%@ page import="java.util.ArrayList" %>
<%@ page import="com.example.chat_system.model.Message" %>
<%@ page import="java.util.UUID" %>
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>Room Page</title>
    <meta http-equiv="refresh" content="15" >
</head>
<body>
<jsp:include page="banner.jsp"></jsp:include>
<h1><%=request.getSession(false).getAttribute("titleRoomInSession")%>
</h1>
<jsp:useBean id="rooms" class="com.example.chat_system.model.Rooms" scope="application"/>
<div>
    <p>Add new message</p>
    <form action="room" method="post">
        Message:<input type="text" name="message"/><br/><br/>
        <input type="submit" value="Send"/>
    </form>
</div>
<div>
    <form action="room" method="get">
        <input type="submit" value="Reload"/>
    </form>
</div>
```

Figure 14: Meta tag refresh

Now, another user can open two different browsers and test the web app. The user is able to see the previous messages as soon as he enters in the *hello world* room.

Figure 15: *vinci* user from Mozilla

Then, the *vinci* user sends a message *hello from vinci* and the user *nico* is able to see it thanks to the automatic refresh.

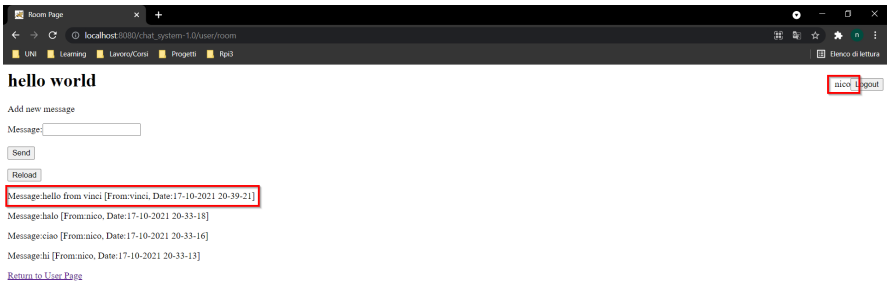

Figure 16: *nico* user is able to see the message from *vinci*

3.4 Admin user

The ADMIN user in the *adminPage.jsp* is able to see the list of users with their credentials. Moreover, he can create a new user filling the form and clicking the *Create* button. When it is clicked, a *post* request is performed to the *AdminServlet*. It handles the request in the *doPost* method. Firstly, it retrieves the HashMap *credentials* stored in the servlet context. Secondly, it checks if the username or password parameter are empty. If it so, it sets the request attribute *error* to true in order to show the *inputError.jsp* and forwards the request to the *adminPage.jsp*. Otherwise, it adds the new user in the *credentials* HashMap and forwards the request to the *adminPage.jsp*.

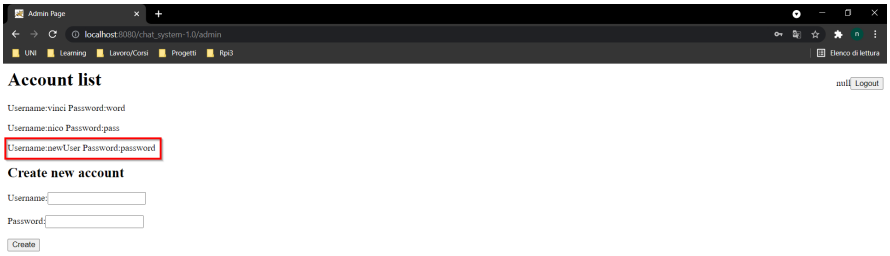

Figure 17: Adding new user

3.5 Filter

The *AuthFilter* has been developed for access control. It checks if the session is not null or if the attribute *authenticated* in the session is set to true. If one of two conditions is not passed, the unauthorized error will be returned. Otherwise, the user can go ahead. The filter is defined in the web.xml file and it is applied to any path that matches */user/**.

```
<filter>
    <filter-name>AuthFilter</filter-name>
    <filter-class>com.example.chat_system.filter.AuthFilter</filter-class>
</filter>

<filter-mapping>
    <filter-name>AuthFilter</filter-name>
    <url-pattern>/user/*</url-pattern>
</filter-mapping>
```

Figure 18: Filter in web.xml

A session can be null when it expires. For example, if the session is expires and a user wants go to */user/room*, the filter will return the error 401 as shown in the figure 19.

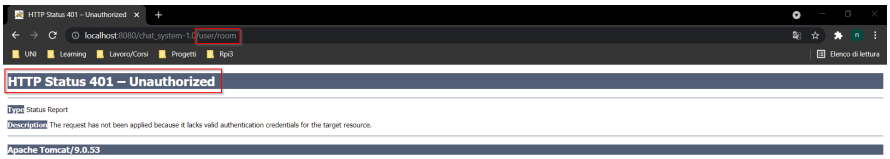

Figure 19: Error 401

3.6 Logout

Every JSP includes the *banner.jsp*, except for the *login.jsp*. It shows the username of the actual logged user and a *logout* button. When the user clicks it, a *get* request to the *Login* Servlet is performed. The request is handled by the *doGet* method. It checks if the session objects is not null and invalidate the session. Then, it forwards the request to the *Login* Servlet.

3.7 End web app

The web app can be stopped from the Tomcat manager panel.

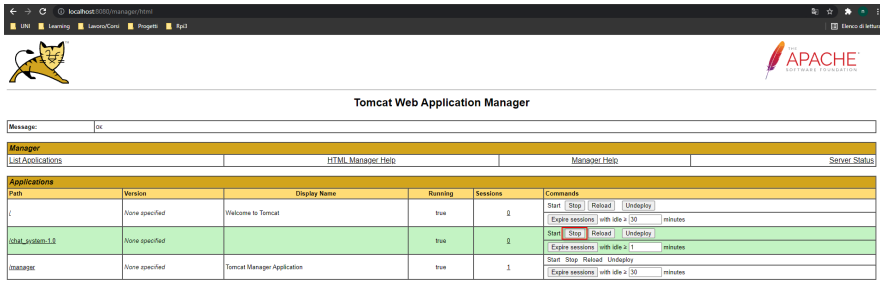

Figure 20: Stop web app through Tomcat manager panel

When the web app is stopped, the method *contextDestroyed* of the *FileListener* is called. The method retrieves the *credentials* HashMap and stores every user in the *authentication.txt* file overwriting it. The ADMIN user added a new user with username *newUser* and password *password* in the figure 17. Indeed, the new user is correctly stored in the *authentication.txt*.

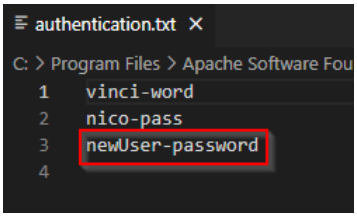

Figure 21: New *authentication.txt* file