COURSEWORK 1: DISTRIBUTED APPLICATION

by Kristian Spiropali

CS6002 Distributed and Internet Systems
Dr. Herbert Maosa
London Metropolitan University
London, United Kingdom

TABLE OF CONTENTS

INTRODUCTION

The aim of this project is to demonstrate my abilities in creating a web-application that will cater to the needs of online and live chatting. By following business logic methods and the lecture slides in weblearn, the web-application must attribute all of the coursework's standard requirements plus personal ones, all of which include but not limited to:

- Having adequate performance, at least not visible to the human eye
- Use top of the end hashing and ecryption in order to safely store data
- Provide sessions for the users of the system in order to track their activity
- Provide an easy and universal means of invoking the backend system API
- All of the above while having little to none bugs where possible

The tools and frameworks which will be used are the following:

- For the backend server, Java programming language will be used along with a very popular framework which is springboot
- For the frontend technology, custom made html pages along with their respective css for styling will be used
- For the functionality of the frontend technology, javascript will be utilized coupled with other libraries which will be discussed in order to accomplish most request calls
- Last but not least, as a preferred database I used java's H2 for all the data management.

The keyword of this project is **websockets**. I had a very good time playing around with multiple sockets in order to provide necessary functionality to the system. All in all, based on my understanding so far, websocket's functionality depends on the following:
- It is the most suitable for two way communication between server/client or client-client
- They are much more faster than the standard HTTP requests
- Very low overhead, as opposed to HTTP requests

There are also problems such as:
- Older non HTML5 compliant browsers will not be able to use it
- It can be considered more complex that just sending HTTP requests, since there are more intermediate steps to do
- Websockets provide persistent and connection-full states, so depending on the web application it might be a disadvantage in comparison to HTTP requests

Since there are a lot of classes involved and not every class needs to be displayed, I will omit some of them. The logic is the same for all classes.

DESIGN AND LOGIC

The abstract or top-level business logic I that accomplished to adopt throughout the entirety of the backend server can be depicted as follows:

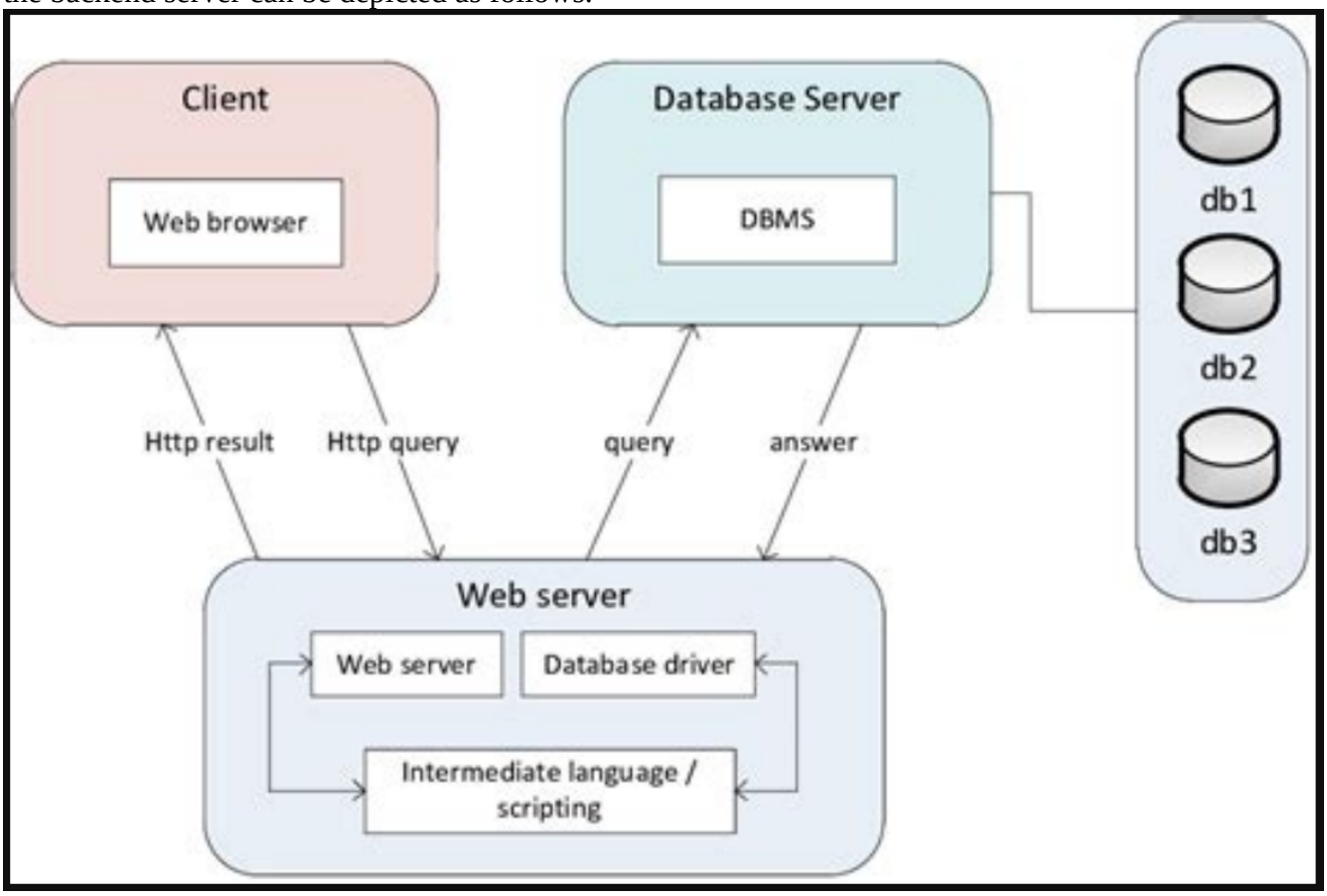

FIGURE 1; 3 TIERED BUSINESS LOGIC ARCHITECTURE

In my case, the above abstract picture can help one imagine that:

- Client → All of the javascript, css and html which will be used to interact with the server

- Web server, which also breaks down into multiple classes:
    - Models
    - Services or otherwise APIs*
    - Controllers or otherwise endpoints mappers
    - Repository control objects
    - System Loggers

- Database Server which as mentioned above will be H2, it also contains a lot more classes which I will not be covering but more importantly has the necessary functionality for me to use it.

In order to better explain the structure of my project, I will start with the class diagrams of each package in the backend server. I firstly start with the **user** package:
First class of the **user** package is the Model one, which is used to shape our data in and out of the database.

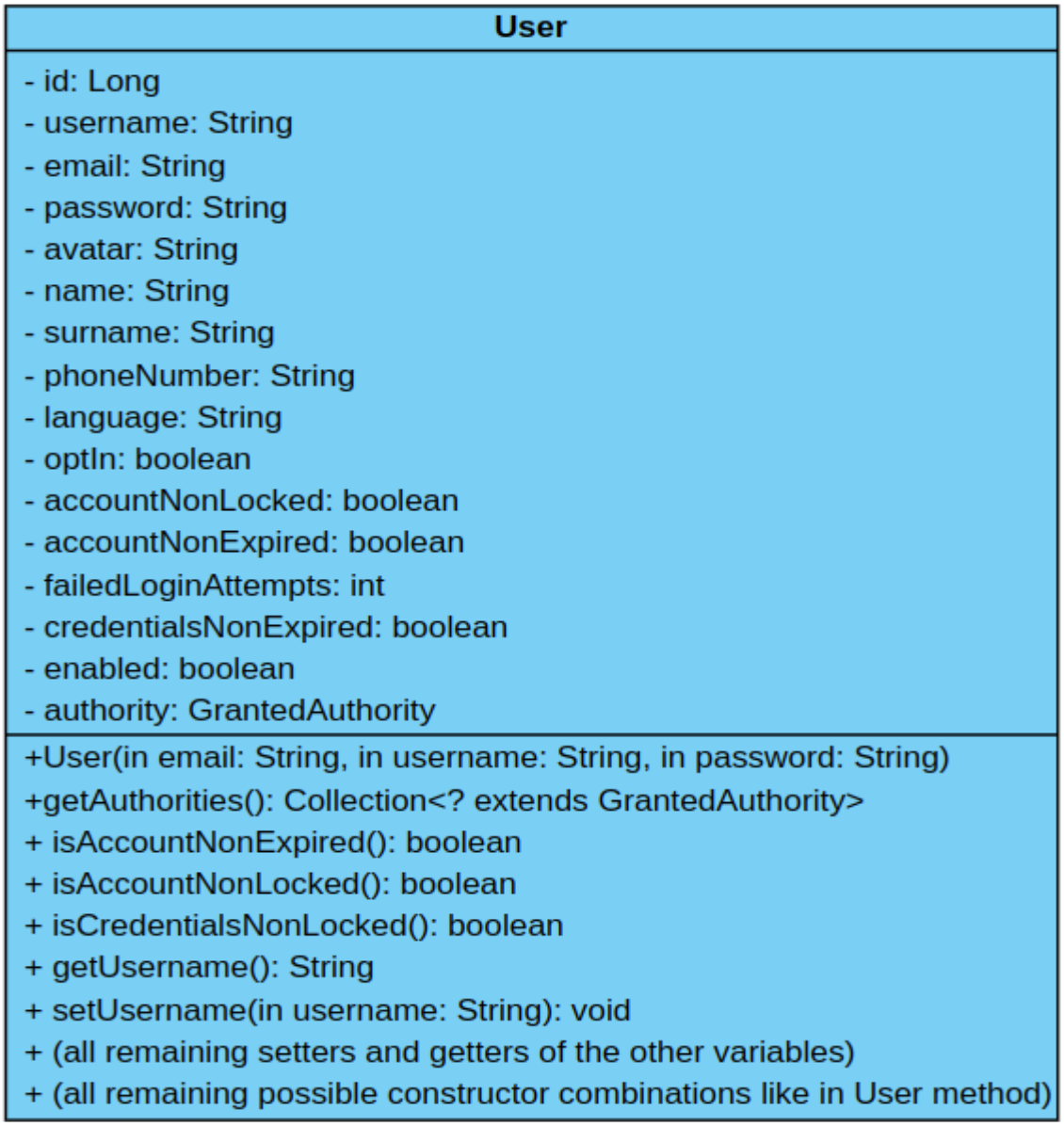

FIGURE 2; ClASS DIAGRAM OF USER MODEL

As shown in the picture, the rest of the getters and setters(mutators) of the 15 other variables are omitted(about 30 methods). There are also more class constructors, but those include all of the other class variable combinations that we can imagine.

Later on in the implementation part, I have used some life-saving libraries that automatically build all of the getters, setters, constructors and toString() methods without me having to write 300 lines of code. The library is called Project Lombok.

Next one is the class responsible for directly interacting with our database repository. It is as follows:

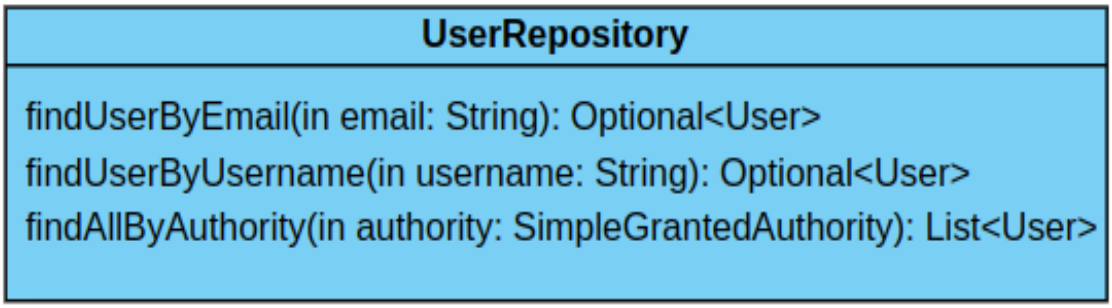

FIGURE 3; CLASS DIAGRAM OF USER REPOSITORY MANAGER

This interface class inherits the methods from another superclass(JpaRepository<?>) which I will not go into depth, but it has all the necessary tools for me to make custom made queries, map them into java functions and use them freely afterwards.

The next one will be the Service or otherwise API class. This is an abstract interface class which defines how one may use the API call functions in order to communicate with the User database repository class. It is as follows:

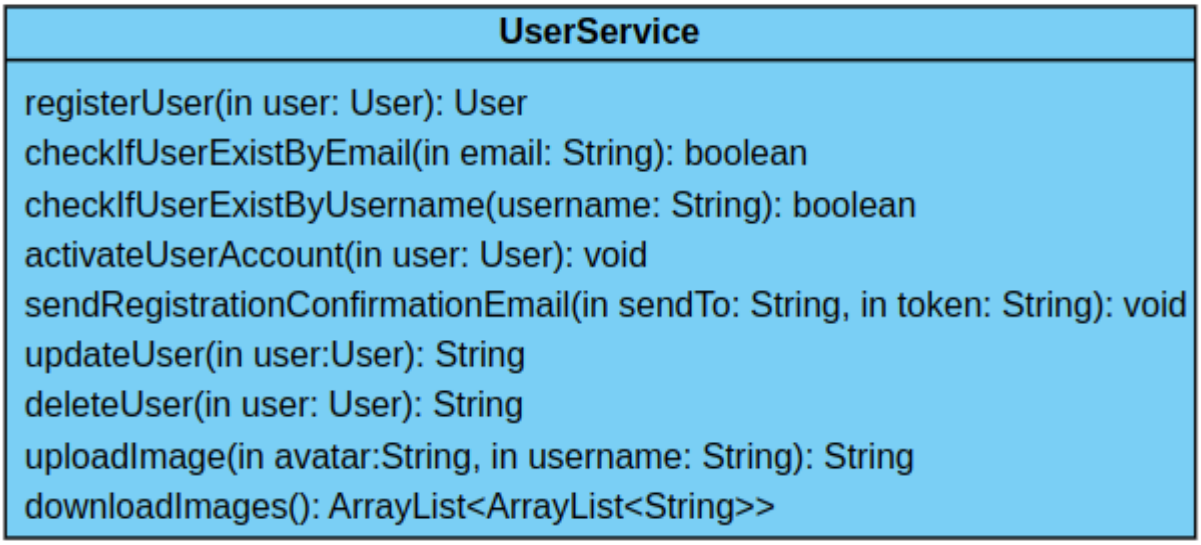

FIGURE 4; CLASS DIAGRAM OF USER SERVICE API

As we will later see this class does not have the actual implementation, but rather the functions that one class should have AND override in order to implement the UserService.class

The following class will just be the one implementing the UserService, hence the name UserServiceImpl.

The UserServiceImpl class is as follows:

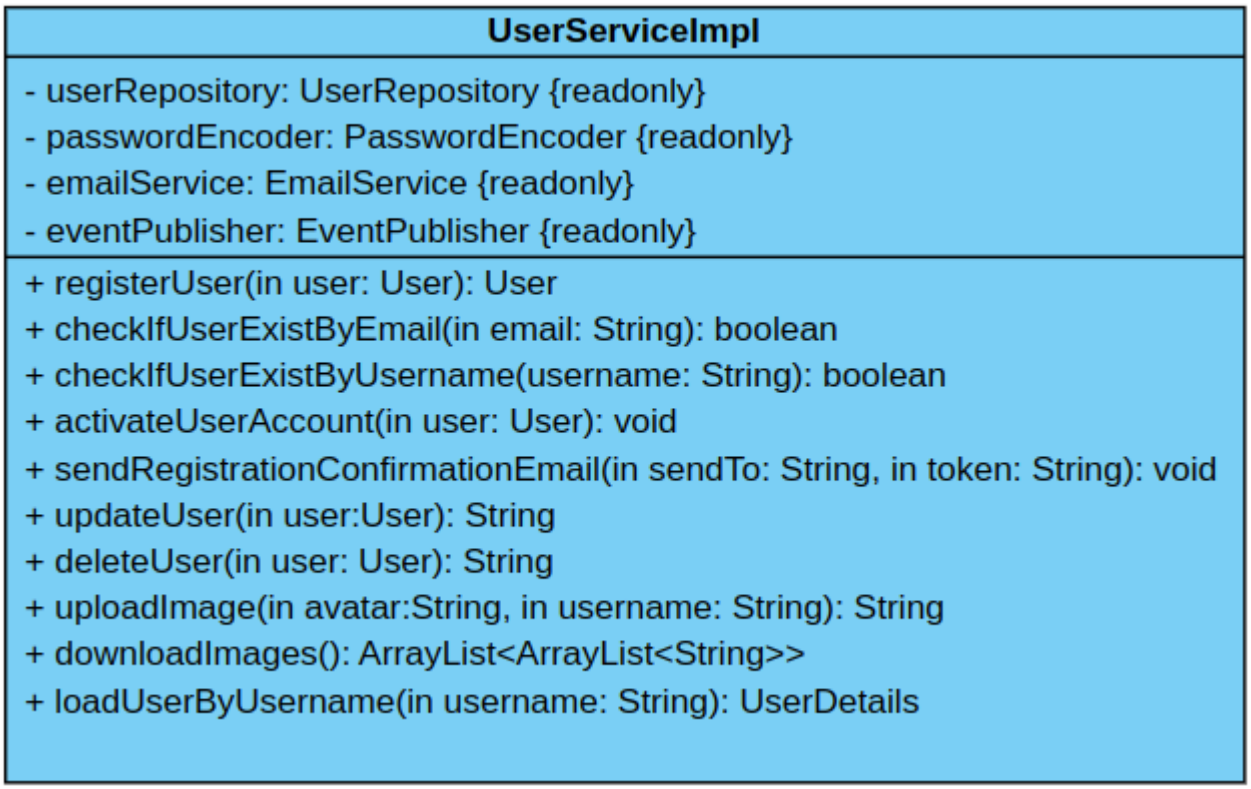

FIGURE 5; CLASS DIAGRAM OF USER SERVICE IMPLEMENTATION

A couple of things to note, first of all this class implements our UserService class plus one more which is the UserDetailService class which is used for authentication and will not be demonstrated as it is a library class. That is why the UserServiceImpl class has 9 methods which it implements from the UserService class plus 1 more which implements from the UserDetailService class.

The variables of this class are used to statically connect other object classes so we can later manipulate their methods, such as UserRepository class we can use the findUserByMail() very easily with such approach.

The next class is just a simple configuration class which only sets up a couple of users for me to test. The UserConfig is as follows:

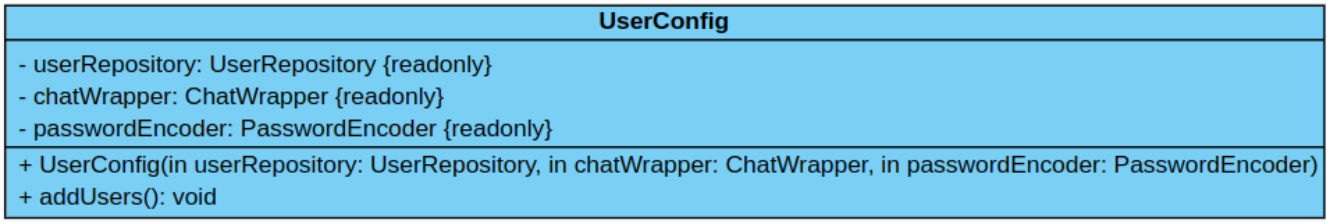

FIGURE 6; CLASS DIAGRAM OF USER CONFIGURATION

The final class in our user package is the nonetheless important that the other ones, UserController. This class maps our java functions from the UserService to appropriate urls. Its diagram is as follows:

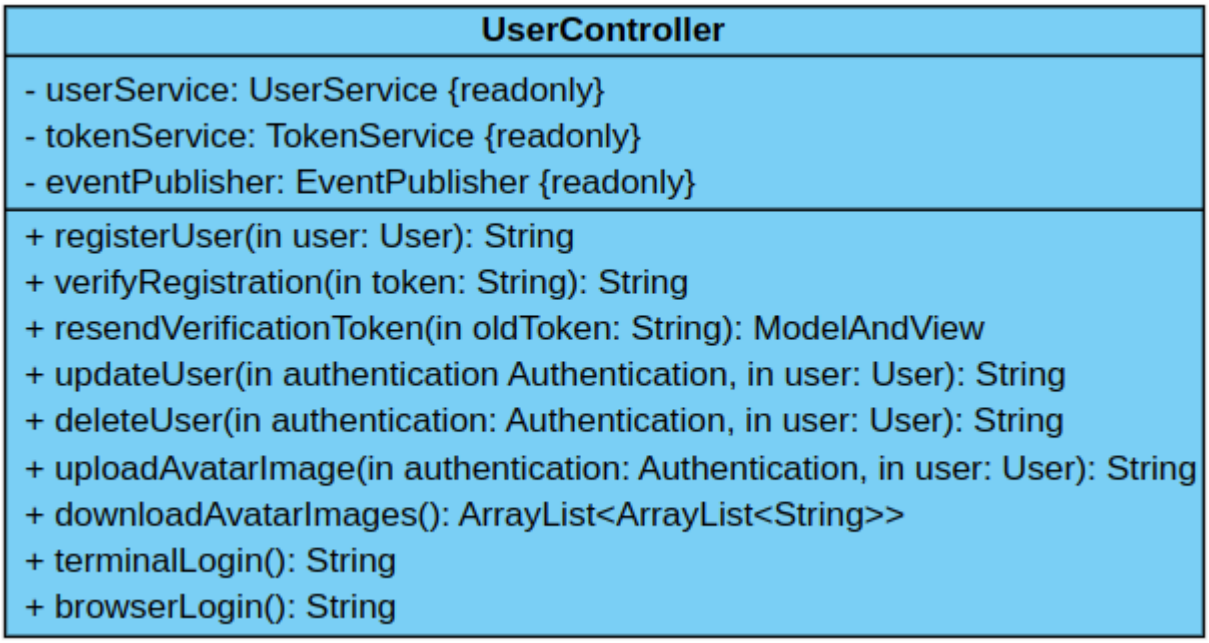

FIGURE 7; CLASS DIAGRAM OF USER CONTROLLER ENDPOINTS

As per usual, all these classes are a direct result of following the 3 tiered business logic. So, the abstract generic package diagram of all the classes is as follows:

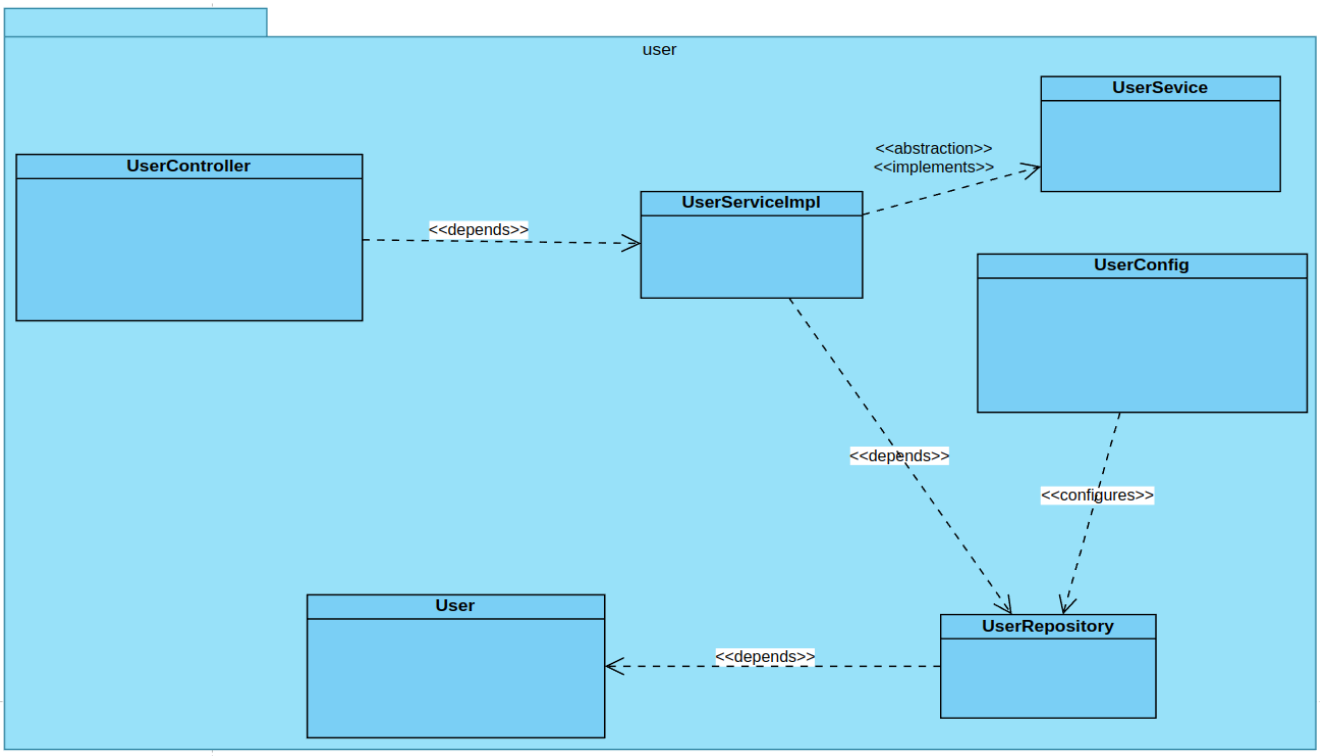

FIGURE 8; USER PACKAGE DIAGRAM

With the same logic I also implemented a **token** based verification package, which generates tokens for a newly registered user so as to activate the user's account. I will make a brief explanation and a package diagram which will contain all the class diagrams of the token package. The diagram is as follows:

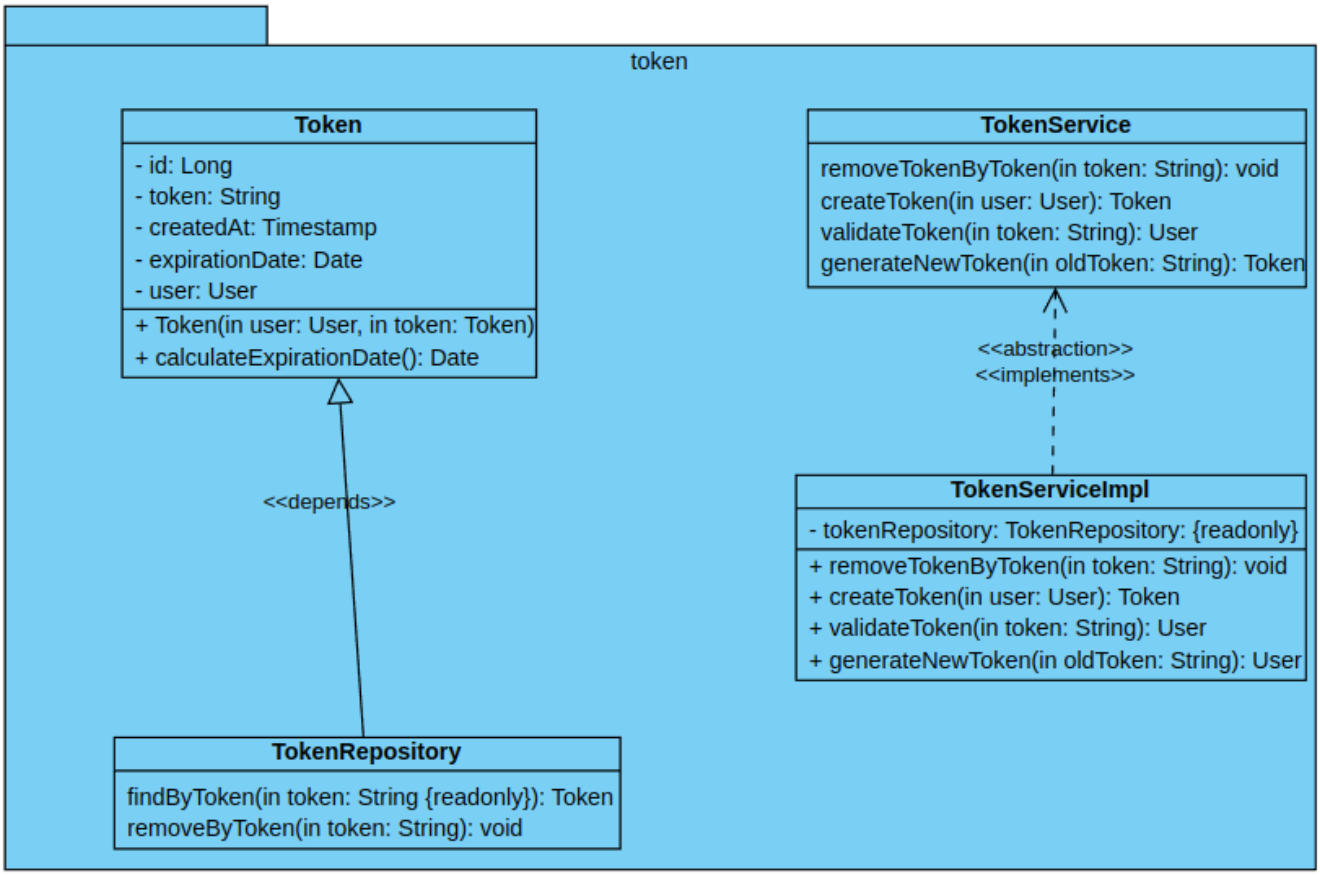

FIGURE 9; TOKEN PACKAGE DIAGRAM WITH CLASS DIAGRAMS INSIDE

A couple of notes to take are:
- There is no need for a controller here, since the package token is a dependency as a whole to the user package, it will be demonstrated later.

- The Token class diagram is missing the getters() and setters() methods. As discussed above like in the User class, I am using a library to auto generate all getters, setters, toString() method and all possible constructors with the 5 variables as inputs.

- As we can see in the Token class, it contains a reference variable to the User(-user: User). That is because I wanted to link each and every newly registered but not activated user to one and only one token so in order to achieve that foreign user_id key in the token table, user needs to be referenced in as a constructor class variable.

Without repeating myself, I will also demonstrate how I handle custom made Events with the help of the Event package. The relevant diagrams are shown below:

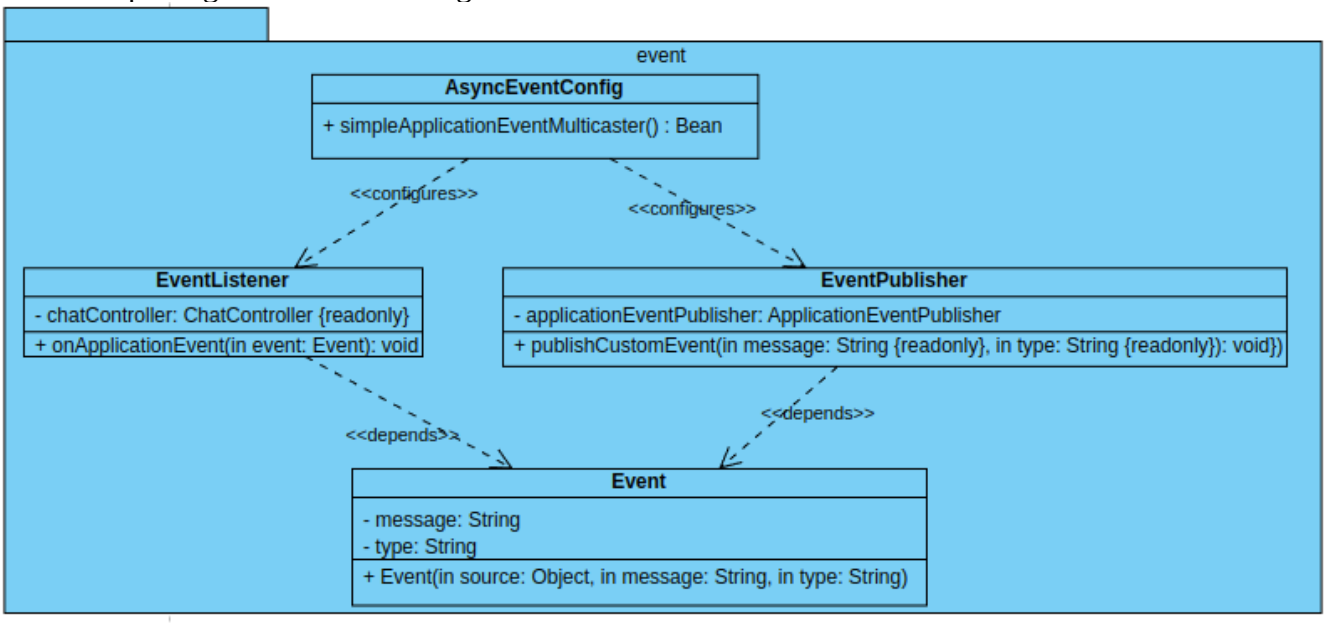

FIGURE 10; EVENT PACKAGE DIAGRAM WITH CLASS DIAGRAMS INSIDE

Again, a couple of notes to take are:

- This package might be considered useless since the only times I am using it for now is when I would like to notify my chat package, which will be discussed later, that a new user registered. This can also be very easily implemented by importing the chat class and passing the message directly. It is goes against the 3 tiered business architecture though, rewiring 2 completely independent packages together, hardcoding the services of one to another. That is why a middle-man is used instead, which is the Event.
- The Event class is pretty straight forward, defining just a message and a type. The EventPublisher is also self explanatory, it is used to publish an event with the EventListener catching that event and passing it through whatever method is calling it. For the purpose of asynchronously publishing and listening to event, a special type of configuration class is used, the AsyncEventConfig.
- The AsyncEventConfig has only a special @Bean type of method, which can help us encapsulate the Event properties in just a class! Properties such as the Event buffer size, asynchronous or no async and etc. After that, springboot automatically collects the @Bean and injects it to the Event package on runtime.

The next package is the **email**. The package diagram along with its respective class diagram is as follows:

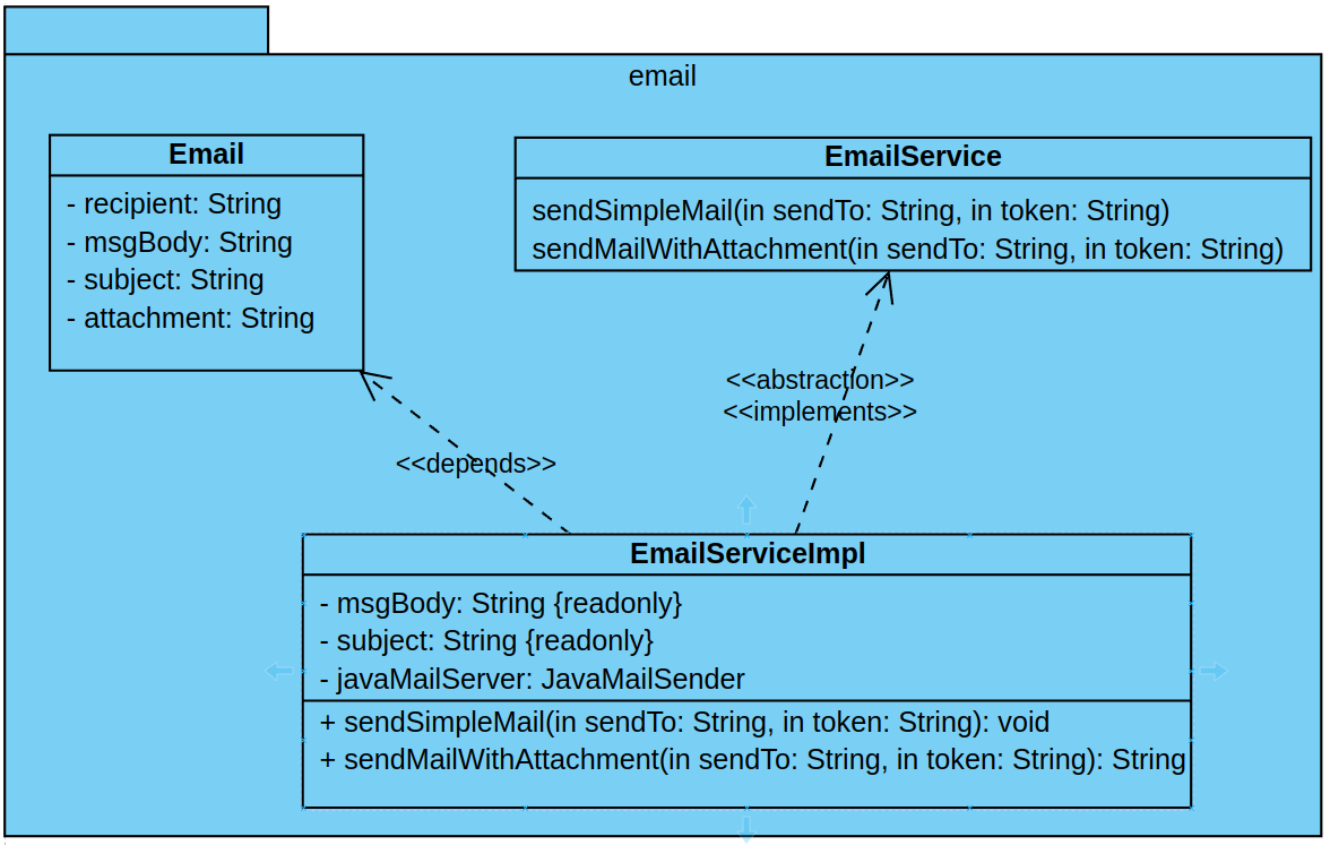

FIGURE 11; EMAIL PACKAGE DIAGRAM WITH CLASS DIAGRAMS INSIDE

A couple of notes:

- The email package is solely used to send emails with or without attachments. Emails will contain the verification link in which newly registered users need to click in order to activate their accounts and use them. If the link is not clicked, the user will not be able to log in, something which will be demonstrated in the testing chapter.

- The email package actually uses a third party library called JavaMailSender which can be found here.

- We can notice that there is no EmailRepository. If I wanted to store such emails in the database, I could very easily do so with the EmailRepository class implementation, but there is not much point here, at least for the scope of this project.

The next class is **security**, which is vital for anything related to Security, Session management, authentication, authorization and generally controlling what type of user can do specific actions. While as of writing, the WebSecurityConfigurerAdapter class is near its end of life and will soon be deprecated, I nonetheless used it because it fullfills my needs. The package diagram is as follows:

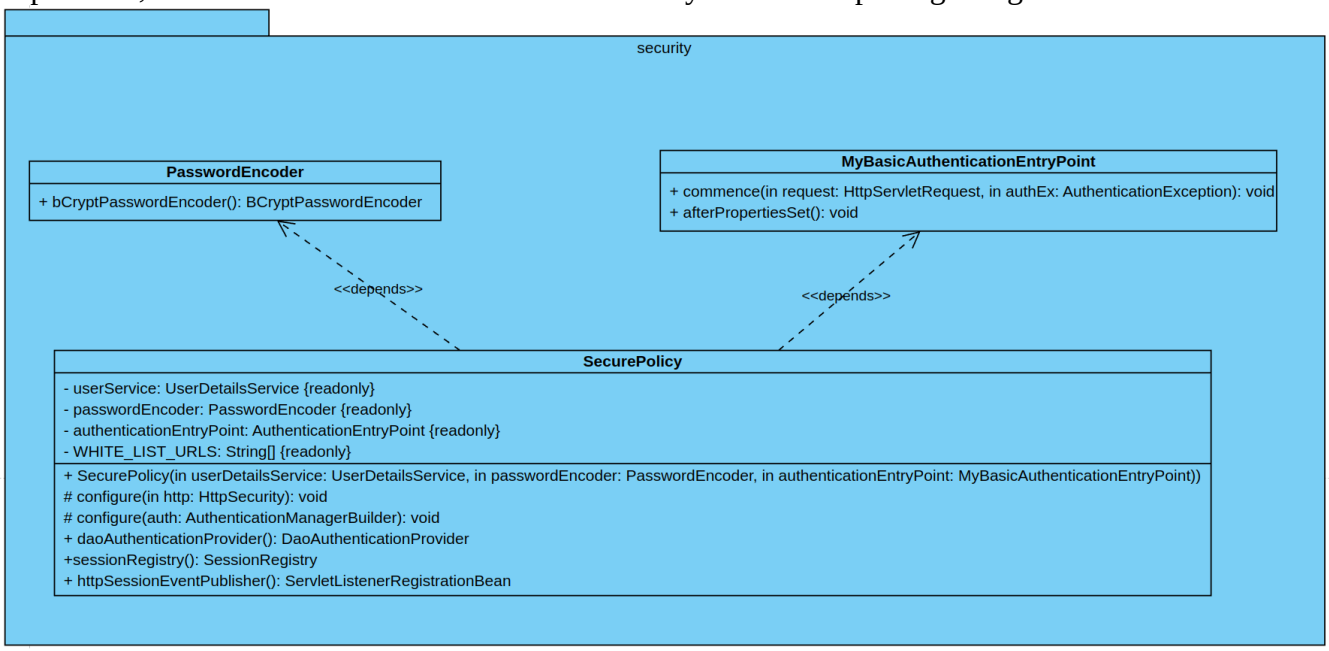

FIGURE 12; SECURITY PACKAGE DIAGRAM WITH CLASS DIAGRAM INSIDE

A couple of notes to keep in mind:
- This package is solely responsible for managing all of the above. By providing it appropriate beans such as the special class PasswordEncoder, I can configure how complex should the hashing be.
- Because I mentioned hashing above, I am hashing the passwords of the users along with their role in the database. Even if attackers can infiltrate the database, they will not find the actual password since hashing is a one way procedure, unlike encrypting.
-The MyBasicAuthenticationEntryPoint is used instead of Springboot's default one in order to stop the annoying http authentication popup from asking the user correct username and password, when previous authentication has failed. Demonstration is as follows:

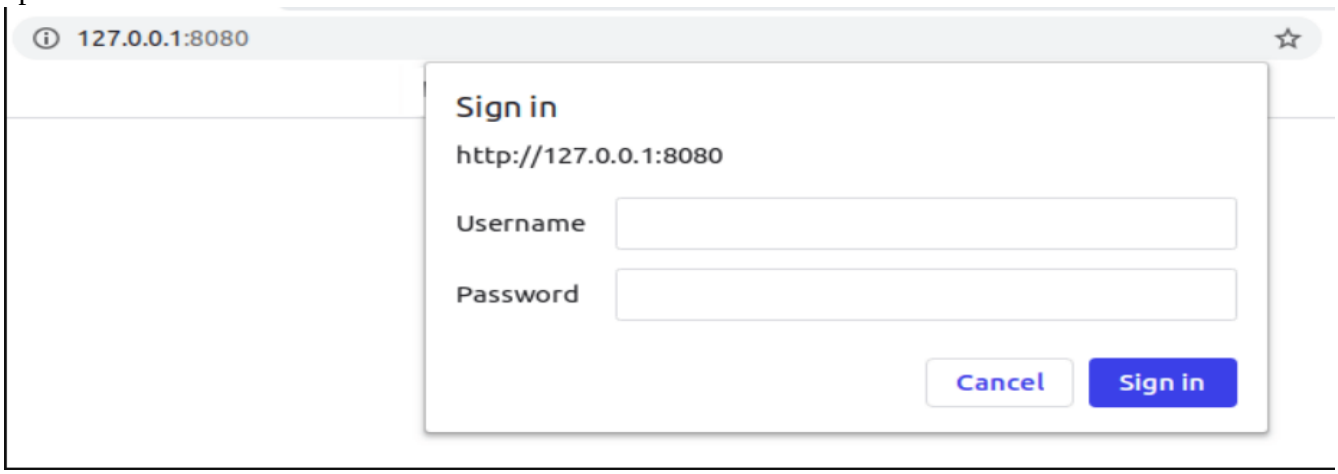

FIGURE 13; DEFAULT BROWSER HTTP AUTHENTICATION POPUP

Because I like to save the good for last, I specifically omitted the **chat** package for the end, even if it is the first one in the alphabetic hierarchy. This is where all the websocket magic happens, the reason why almost instantaneous communication can happen between active users in the webpage.

I will first start with the its Model class, Message:

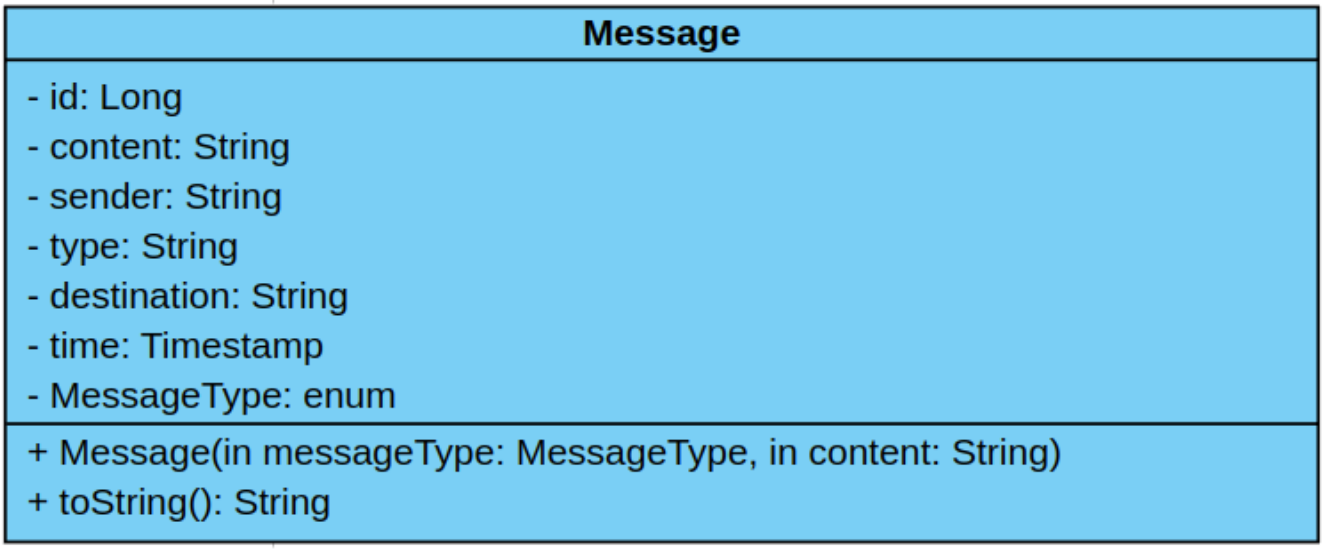

FIGURE 14; MESSAGE CLASS DIAGRAM

A couple of notes to keep in mind:

- The message has an enumeration class variable called MessageType. This is the variable that enables me to differentiate between messages in and out of websockets when I wrote the frontend javascript part of the project.

The next class will be MessageRepository, which is solely used to store messages in the database. Diagram is as follows:

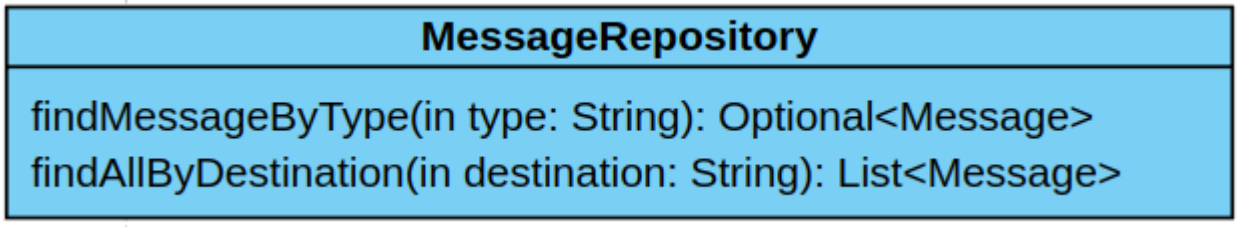

FIGURE 15; MESSAGE REPOSITORY CLASS DIAGRAM

Notes on this class:
- The Optional<?> as discussed before, is used to tell the compiled that its ok if we get nothing(null) from the database, when searching for messages by type in this case. Otherwise errors would be thrown around and I would have to write more classes than I have currently in order to handle them.

The next class diagram is the ChatWrapper. While it is similar to a Service class, I needed to differentiate it from one because it also handles a variable which is registeredUsers and this variable is accessed from 2 other classes in the chat package. The diagram is as follows:

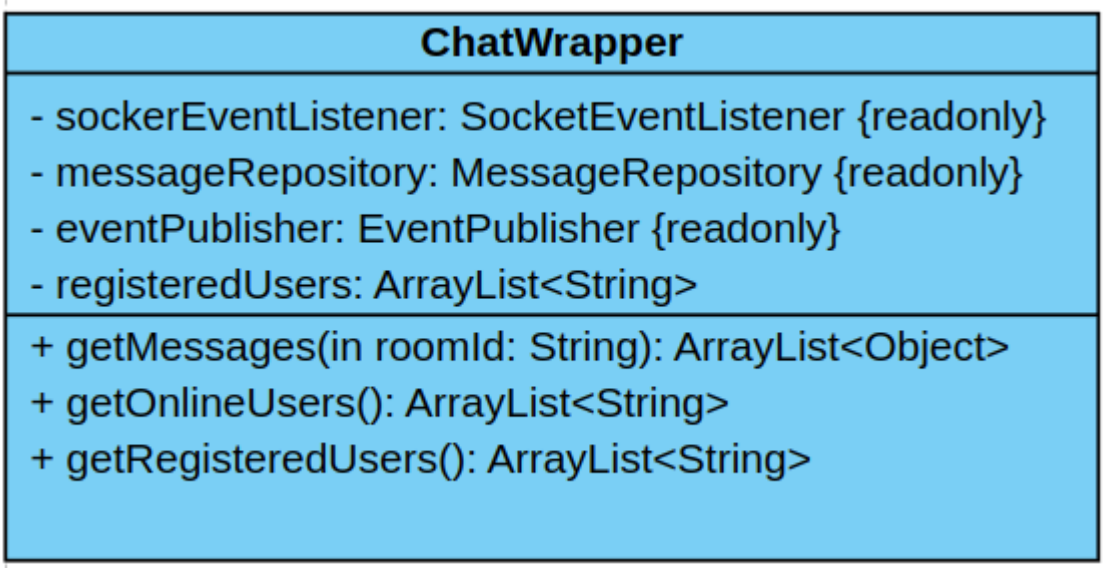

FIGURE 15; CHATWRAPPER CLASS DIAGRAM

A couple of notes:

- This class is used to update the registered user list when a user logs in via the browser. It also is used to update the current active user list, by changing a dot from white to green and vise versa for the non active users.
- It does the above functions by the EventPublisher package, that is the reason why the eventPublisher is a class variable.

Next is the ChatController class. It acts as the controller of the chat but instead of mapping http requests to functions, it is mapping ws:// websocket events, based on the type of the message as explained above. The diagram is as follows:

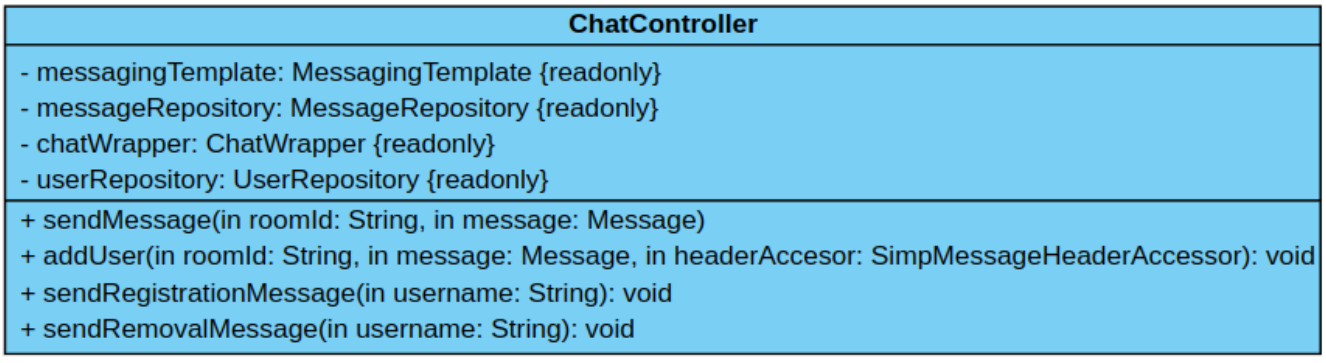

FIGURE 16; CHATCONTROLLER CLASS DIAGRAM

The next class is the actual socket address configuration, SocketConfig. The diagram is as follows:

FIGURE 17; SOCKETCONFIG CLASS DIAGRAM

No special notes on this class, all it does is setup the address, port is taken from the default one which is :8080 in this case. I had problems while transporting big messages, audio messages and image messages because the socket buffer size was too small. The configureWebSocketTransport does exactly that purpose.

Last class of the package is the Event listener and logger, SocketEventListener. Diagram is as follows:

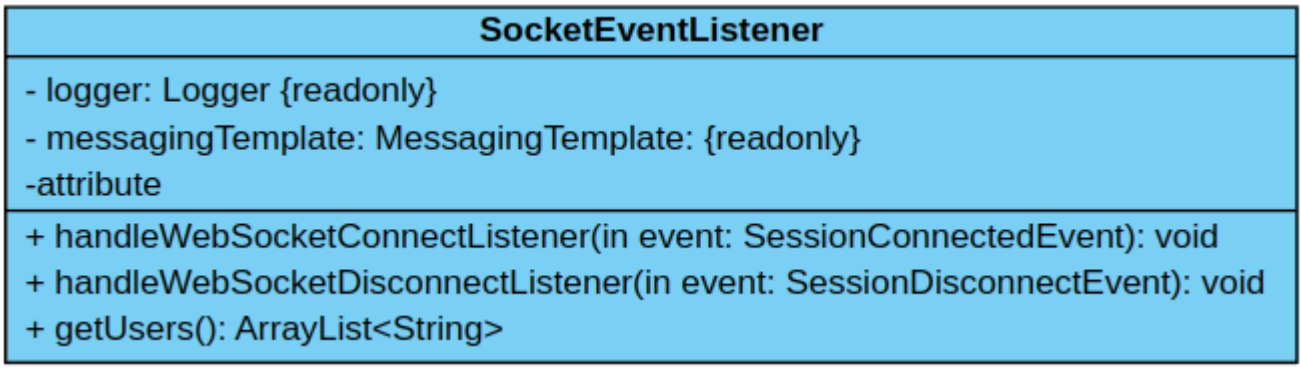

FIGURE 18; SOCKETEVENTLISTENER CLASS DIAGRAM

The final package diagram based on the above class diagram displayed above is as follows:

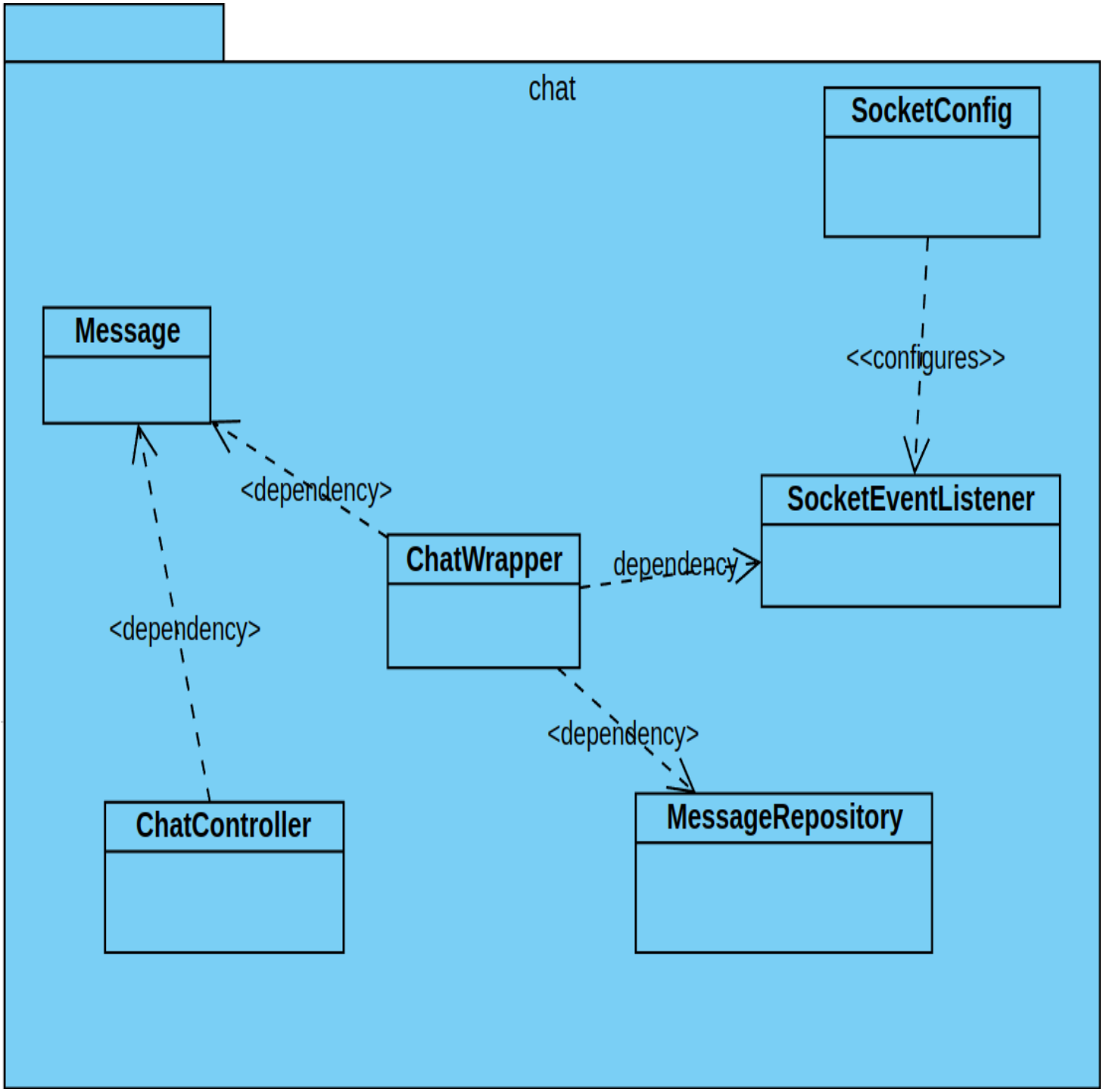

FIGURE 19; PACKAGE DIAGRAM OF THE CHAT PACKAGE WITH CLASS DIAGRAMS

There is one more package which is just used to map the different html pages into endpoints as well as map the default console manager of the h2 database in the http://localhost:8080/h2-console, as long as the user is logged in(authenticated). The diagram is as follows:

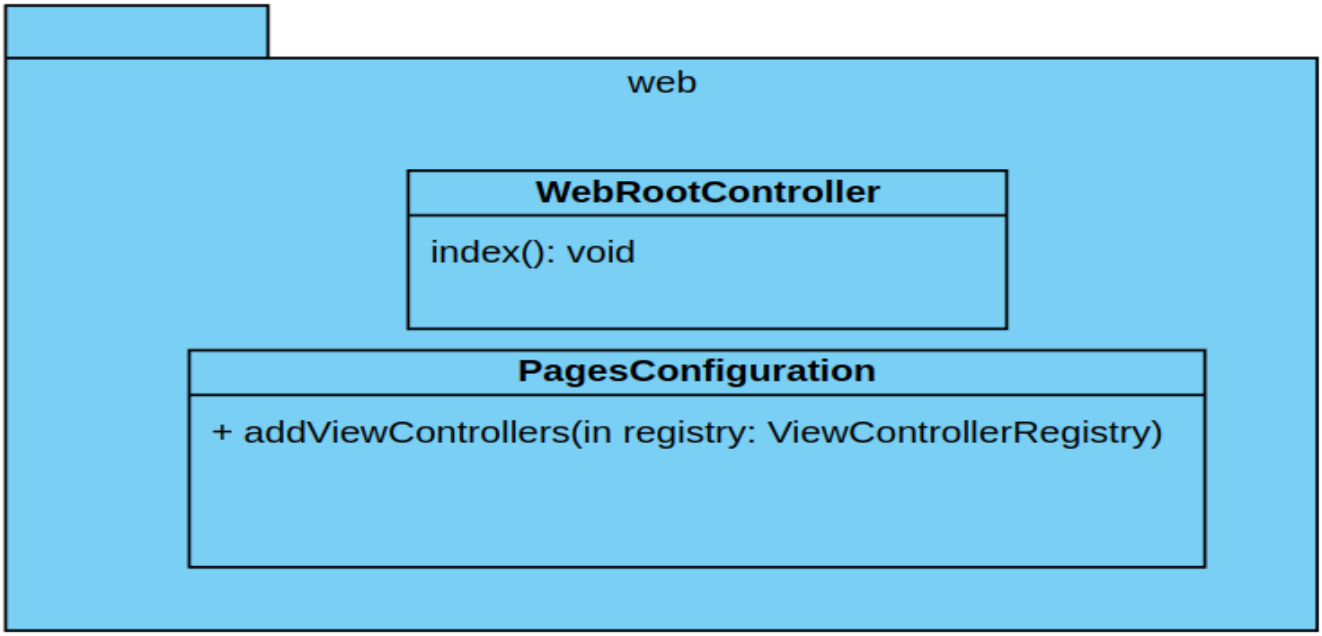

FIGURE 20; PACKAGE DIAGRAM OF THE WEB PACKAGE

Since we constructed all of the pieces of the puzzle, I can now demonstrate a more abstract diagram of all the packages in the backend:

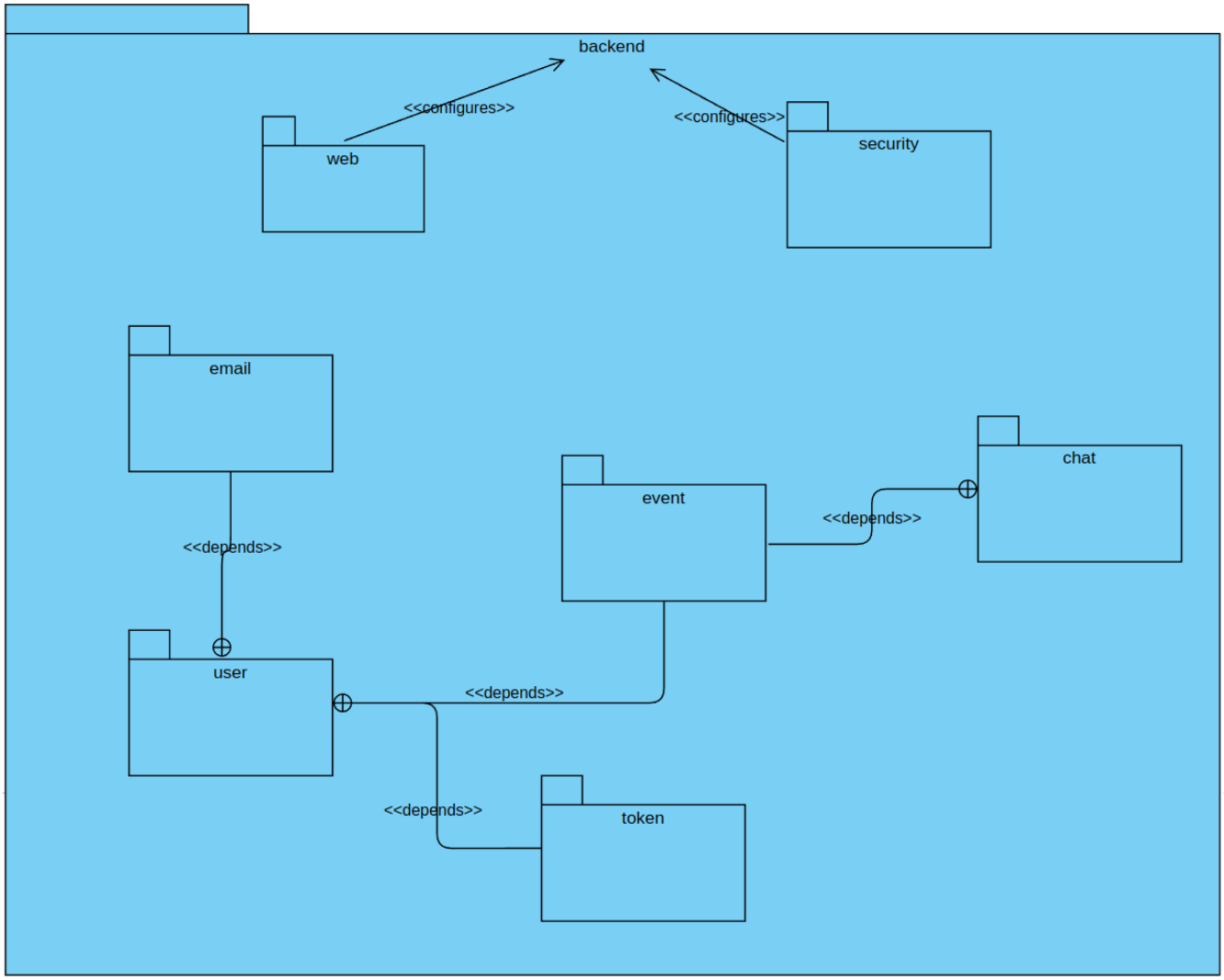

FIGURE 21; PACKAGE DIAGRAM OF BACKEND SERVER

DATABASE

The dabase used, as mentioned above, is H2. I did not use any complex relational tables as I wanted to keep things as simple as possible, for my own sake. The database's diagram is as follows:

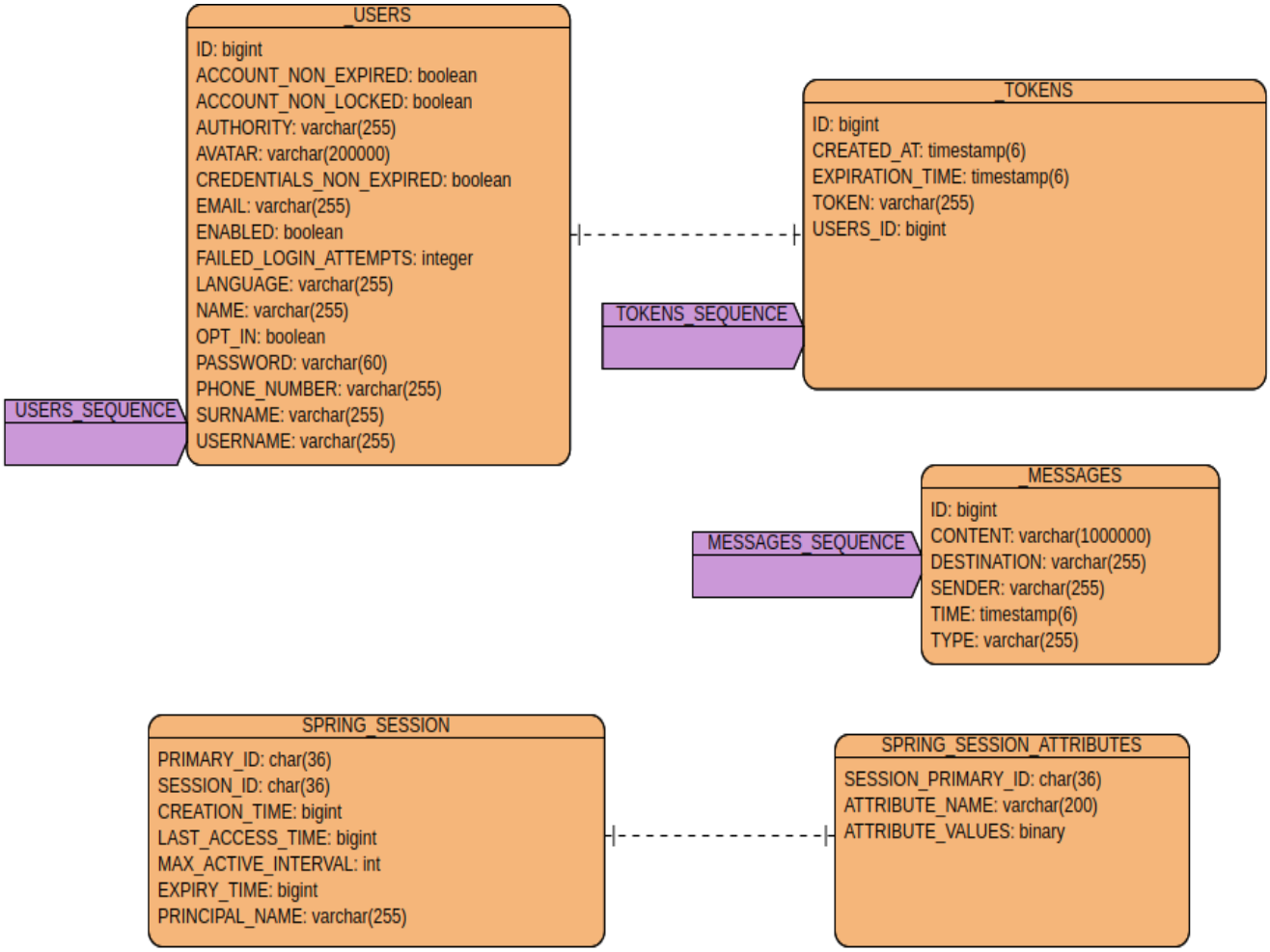

FIGURE 22; ENTITY RELATIONSHIP DIAGRAM

As we can see, every user can have one session. Every session can have an attribute only. I did not implement strict session checking, while I could restrict the user from logging in 2 or more times, it did not ruin the flow of the program.

We also can see one to one relationship between USERS and TOKENS. That is, as mentioned before, one user can only have 1 account activation token.

Lastly, I did not link the MESSAGES table with the USERS table, since it would complicate things. Instead I used sender and destination variables in order to get and sort out messages fast, without including more objects in the table.

FRONTEND

For the frontend I have used strictly AJAX calls in order to asynchronously, where possible and viable, get data from the server. Requests include:
- Authenticate and validate a user in order to log them in
- Create a new user account
- Verify and complete a user's registration
- Get all messages/filter messages by x variable from the database
- get all registered usernames from the database
- get all logged in users from the database(SPRING_SESSION → PRINCIPAL_NAME)
- calls to validate Cross-origin forgery requests when accessing the server from outside the webpages

A generic sitemap would be like shown below:

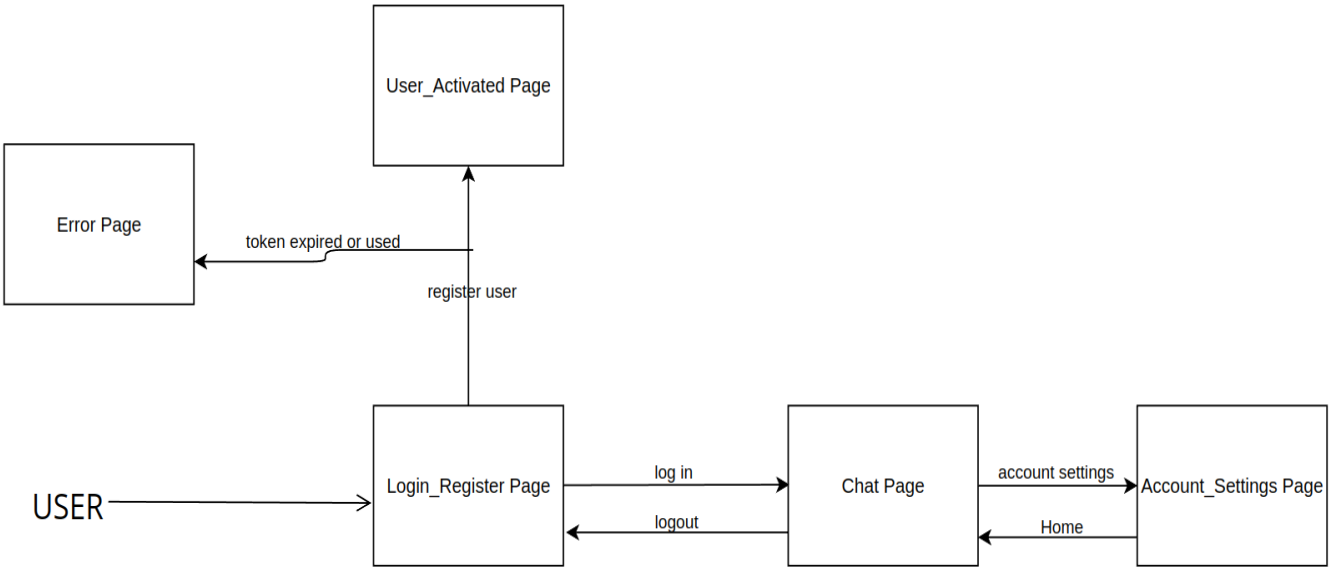

FIGURE 23; SITEMAP OF THE FRONTEND

A couple of notes to keep in mind:

- The Error page will occur when an incorrect url is manually entered in the url bar.
- The above will not happen when the user is not logged in yet, only when a session is established.
- The basic url mapping are:
    - http://localhost:8080/ for the Login_Register page
    - http://localhost:8080/chat for the Chat Page
    - http://localhost:8080/settings for the Account_Settings page
    - http://localhost:8080/user/verifyRegistration?x for the User_Activation page, the x is the token
    - http://localhost:8080/error is the actual error page, if and invalid url is entered and the user is logged in, user will get redirected there. If user is not logged in, any url apart from the Login_Register Page will just redirect the user to the Login_Register Page again until authenticated and authorized.
    - http://localhost:8080/h2-console Needs to be logged in as a user, these details need be entered:
        - Driver Class: org.h2.Driver
        - JDBC URL: jdbc:h2:~/h2
        - User Name: sa
        - password: password

The overall file structure and hierarchy of the project is as follows:
Backend server:

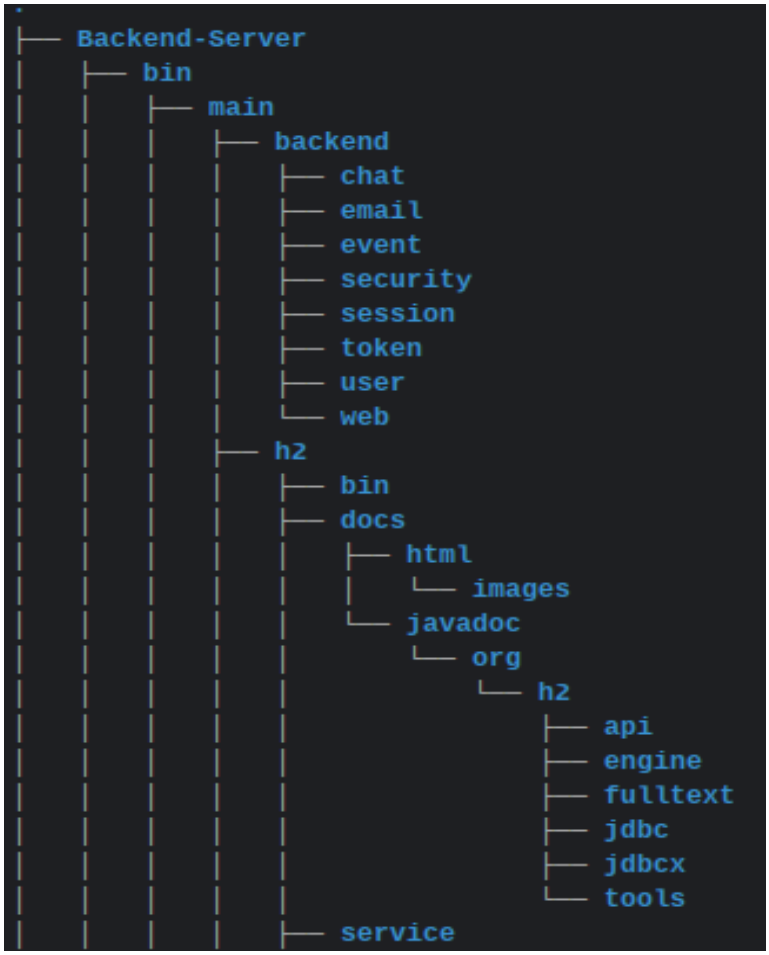

FIGURE 24; STRUCTURE OF THE BACKEND SERVER

Frontend files:

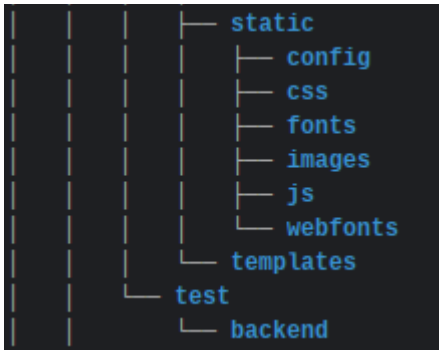

FIGURE 25; STRUCTURE OF THE FRONTEND FILES

H2 database:

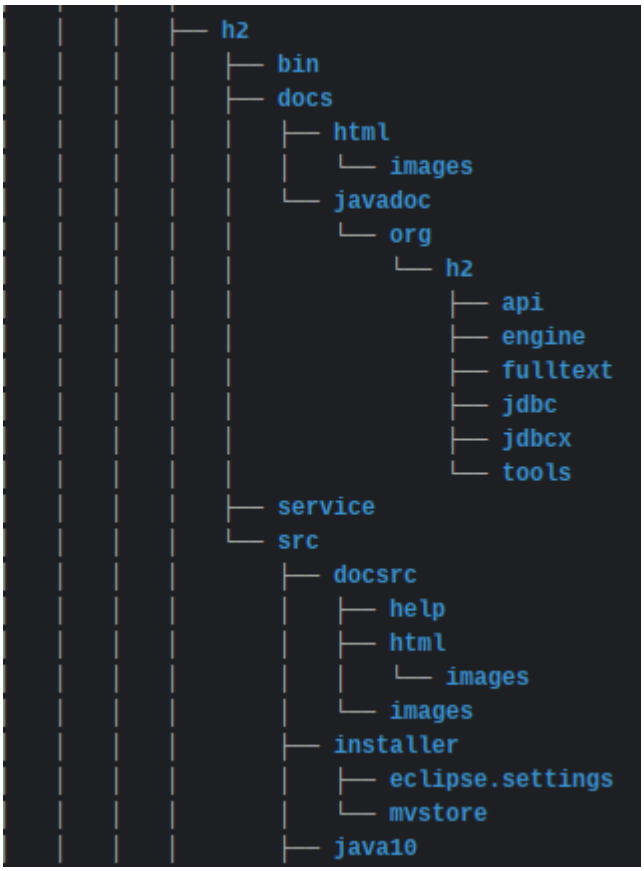

FIGURE 26; STRUCTURE OF THE H2 DATABASE

Take note that there are a lot of files that are omitted. Files such as gradle, files starting with .
files such as bin/ and build which are for binaries, as well as gradle configuration files which configure what libraries the project needs.

So to sum up:
- Backend files are located in:
DISTRIBUTED_INTERNET_SYSTEMS_CW/Backend-Server/src/main/java/backend/

-Frontend files are located in: DISTRIBUTED_INTERNET_SYSTEMS_CW
/Backend-Server/src/main/resources/

- Database files are stored in: DISTRIBUTED_INTERNET_SYSTEMS_CW
/Backend-Server/src/main/resources/h2

For the javascript part there is nothing of note to display, it is just simple ajax calls and some basic document.getElementById() manipulations in order to update html parts.

User experience and design has been taken into consideration thus I have adopted a centered, compact and dark themed look. All validation has been taken into account and will be demonstrated later on in the Testing section.

DEPLOYMENT

In order to deploy this project, a couple of tools are needed:
- JDK17 source/binary files or similar, I personally have compiled the jar files with openjdk17 and temurin jdk17.
- Intellij or
- Eclipse

That is it pretty much, since I have integrated the database so it will be activated autonomously on runtime.

To run the project in Eclipse:

- Open Eclipse
- Go to File → Import → Pick Gradle → Choose Existing Gradle Project
- Choose the DISTRIBUTED_INTERNET_SYSTEMS_CW/ folder
- Click Next on the next page, without overriding any default options that are there
- Wait for the import to finish and click Finish
- Some popups may come up for annotation processing or/and download gradle caches, accept them all

If you would like to explore the classes, in the Package Explorer top right, open up the Backend-Server folder:
- Explore the src/main/java for the backend server java classes
- Explore the src/main/resources for the h2 database file and frontend files
- Explore the Project and External Dependencies to check up on all the libraries that I have used

In order to run the project, on the Top Left side there should be a gradle tab. IF there is no gradle tab make sure that in Eclipse plugins there is gradle installed and restart. Open up the tab and pick Distributed_Internet_System_CW folder.

From there, there are a lot of options to do. The ones that we need are:
- Distributed_Internet_Systems_CW → Backend-Server → application → bootRun, to run the project
- Distributed_Internet_Systems_CW → Backend-Server → application → bootJar, to compile a jar file
- Distributed_Internet_Systems_CW → Backend-Server → application → clean, to clean compiled, jar
- Distributed_Internet_Systems_CW → Backend-Server → documentation → javadoc, for documentation on libraries

I have also attached a jar compiled with opendj17 on the:
Distributed_Internet_Systems_CW/compiled_jar_stable/Backend-Server.jar

To run it, open up a terminal and change directory to the following;
→ Distributed_Internet_Systems_CW/compiled_jar_stable/
→ and just run java -jar Backend-Server.jar

TESTING

I will demonstrate how I have hardened the security, structure, validation of different variables used throught the whole server and more:

Starting with registration in the registration page:

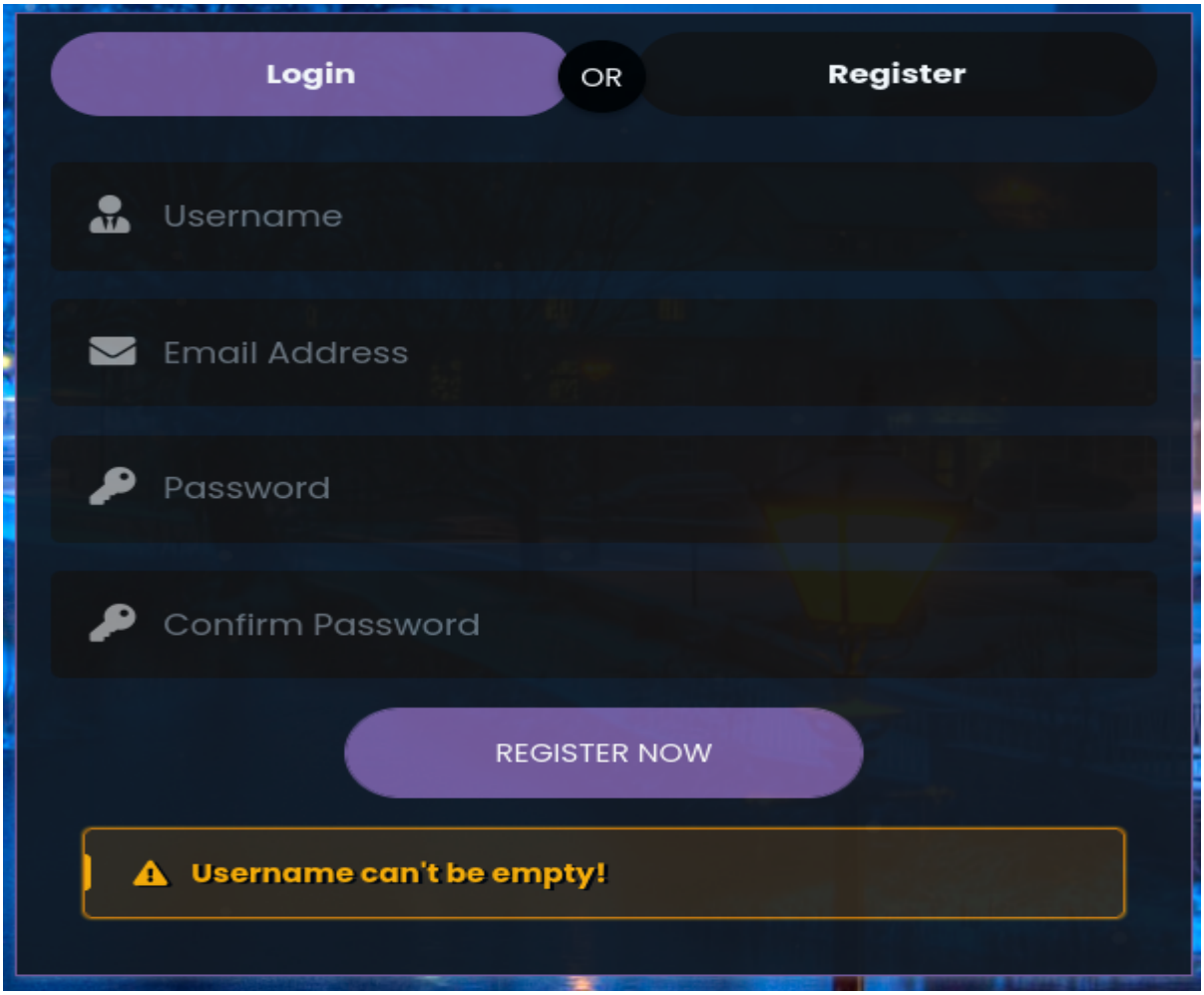

FIGURE 26; EMPTY USERNAME FIELD

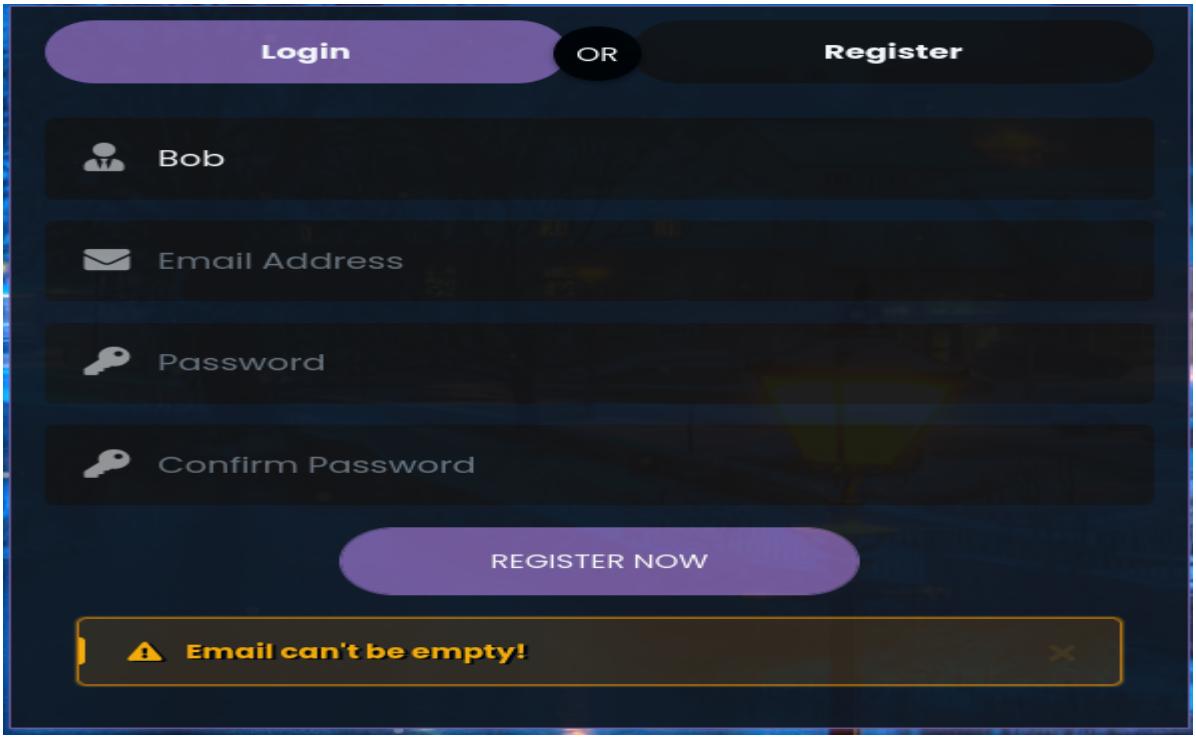

FIGURE 27; EMPTY EMAIL FIELD

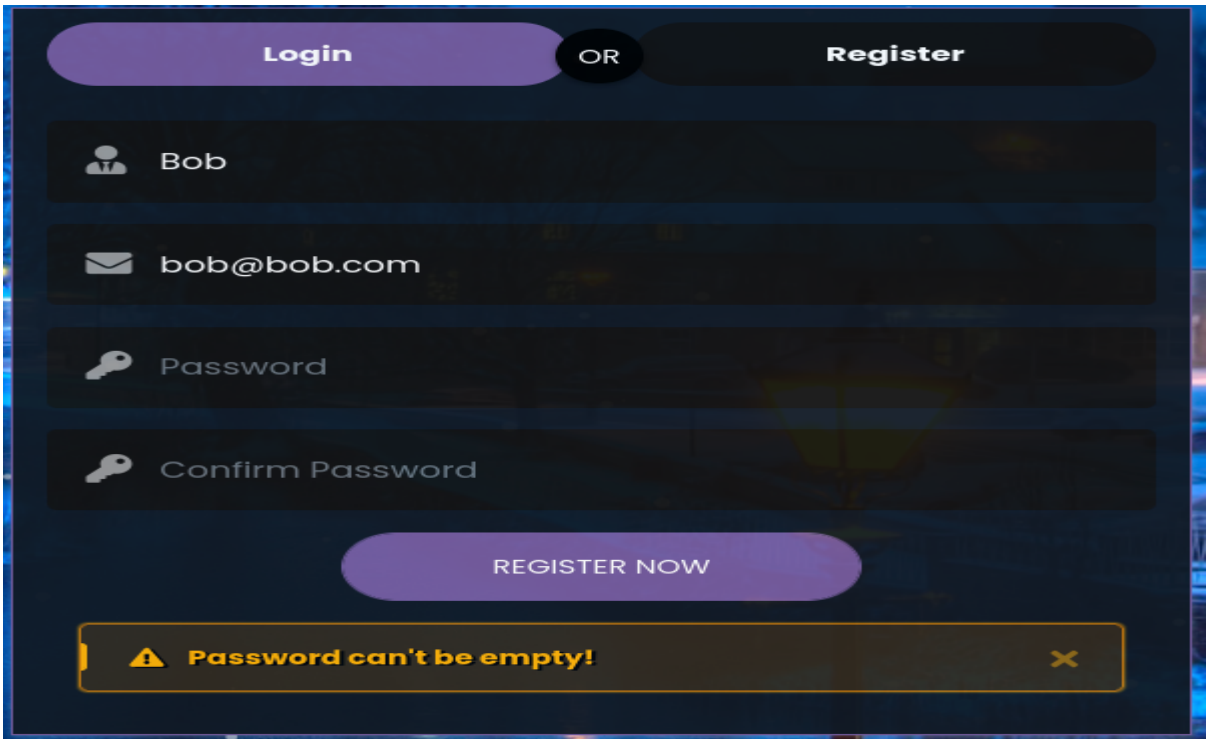

FIGURE 28; EMPTY PASSWORD FIELD

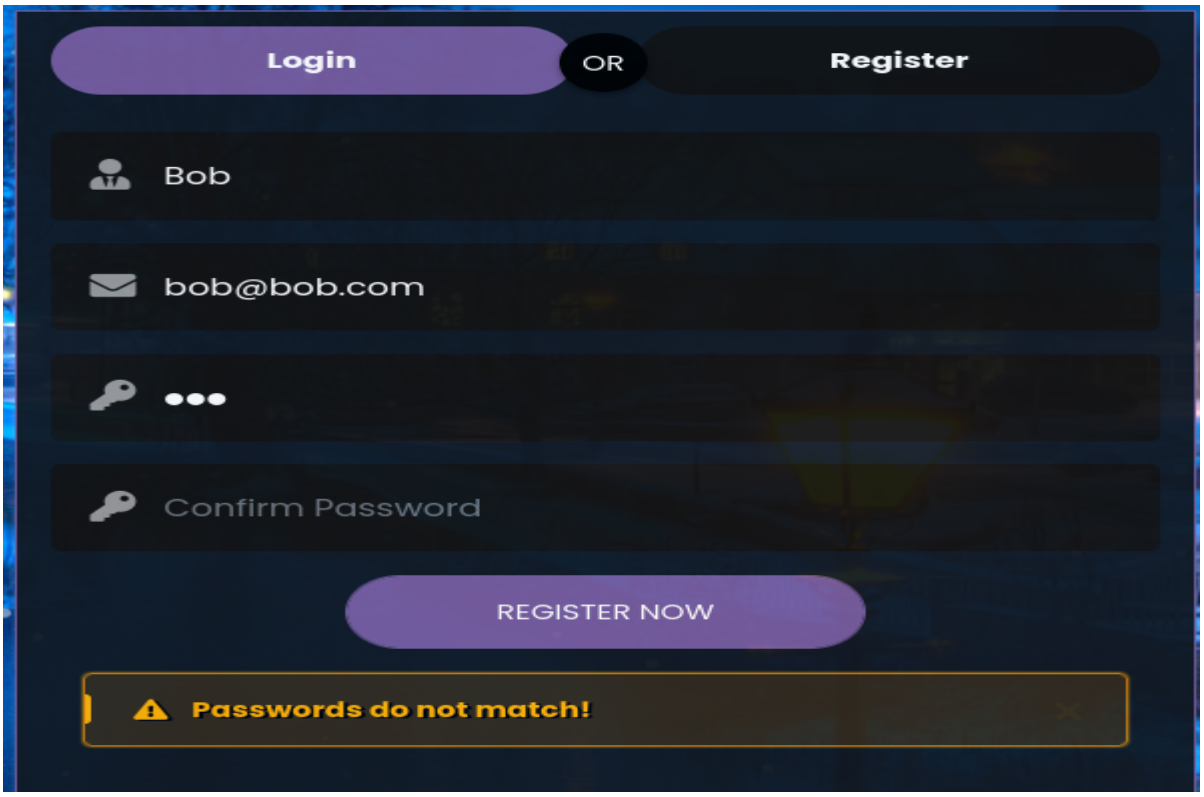

FIGURE 29; EMPTY PASSWORD CONFIRM FIELD

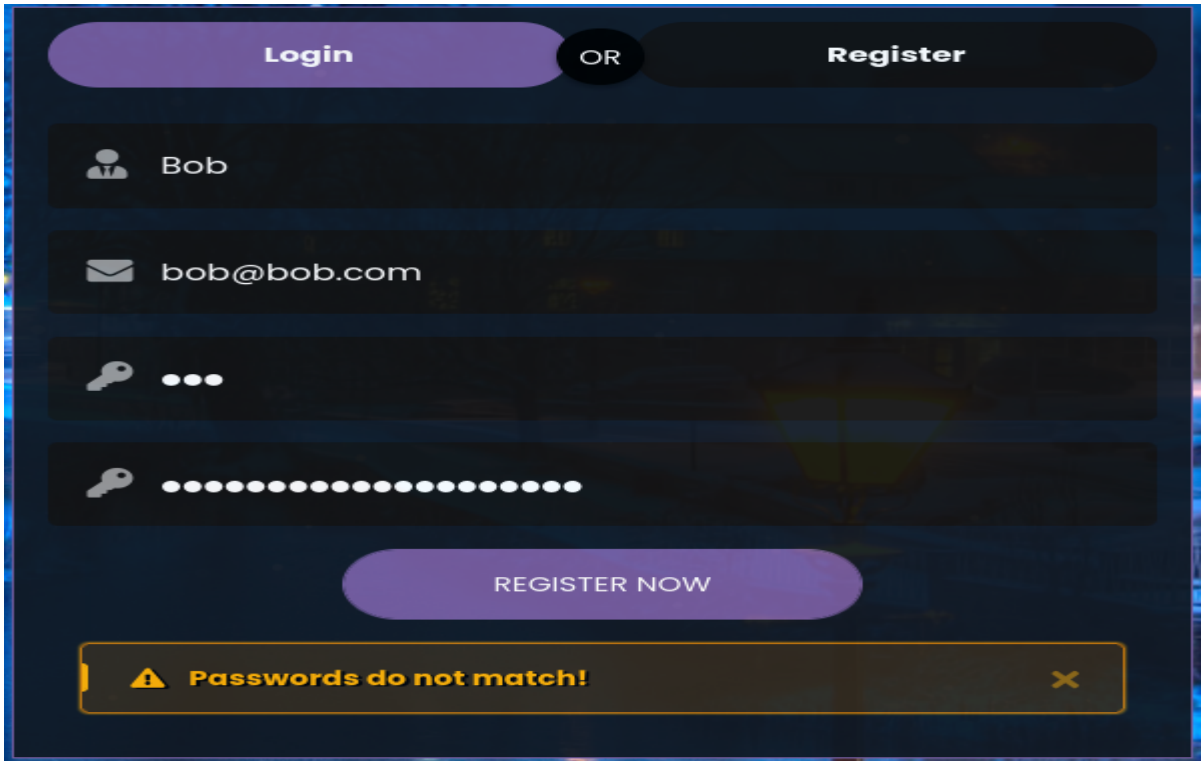

FIGURE 30; PASSWORDS NO MATCH

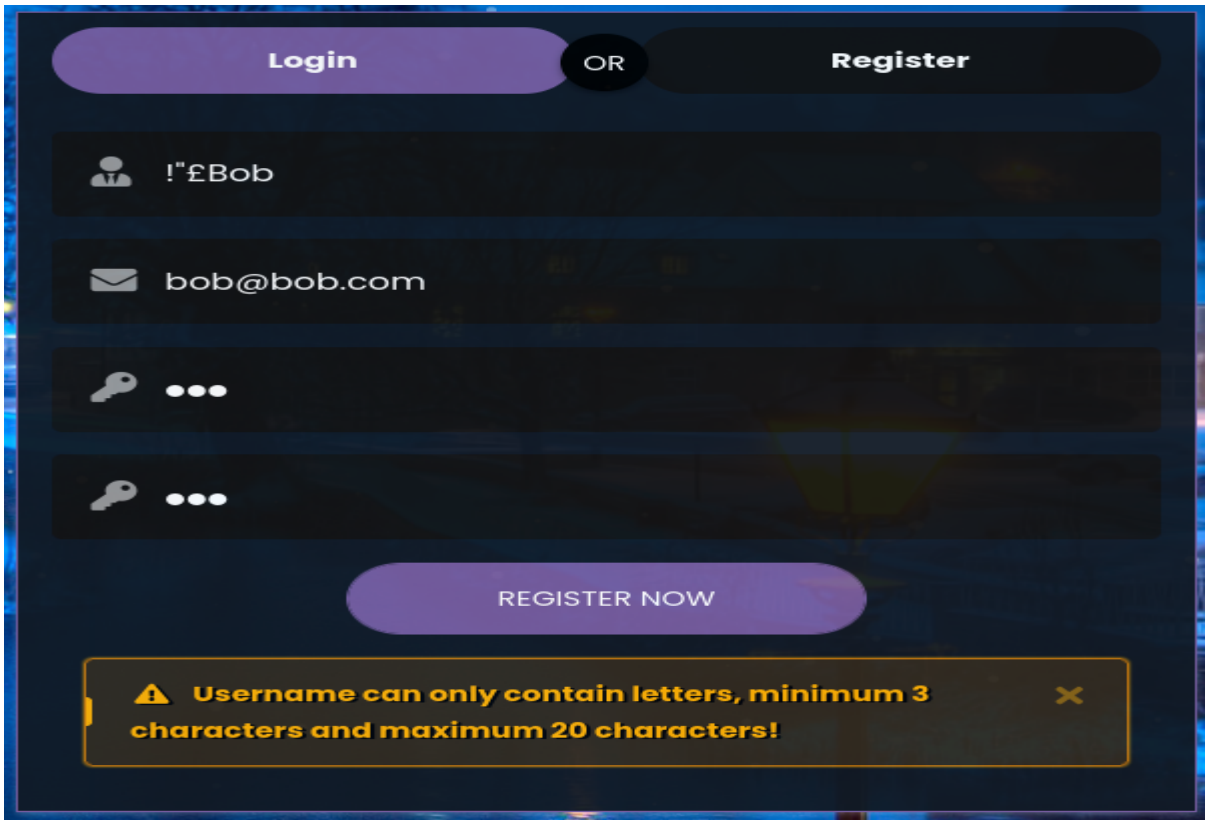

FIGURE 31; INVALID USERNAME

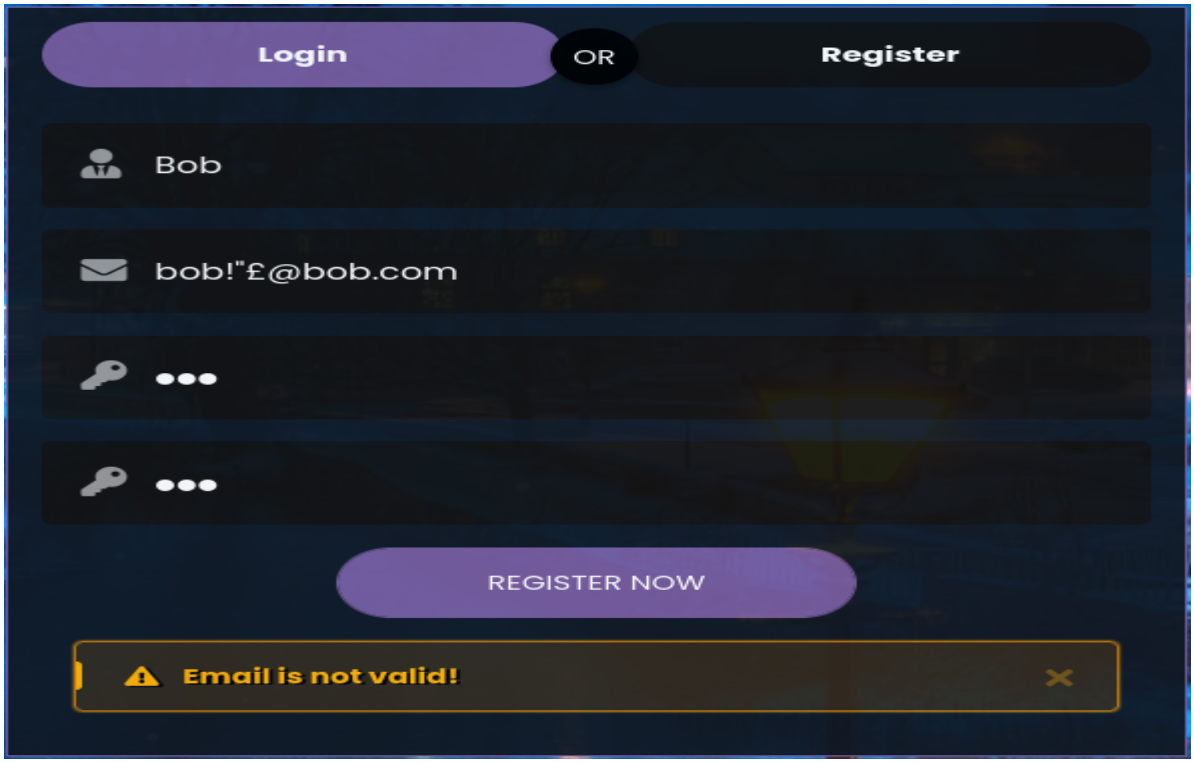

FIGURE 32; INVALID EMAIL

Since I have added already 3 users by default: Bob, Sam, John we get the following validation errors:

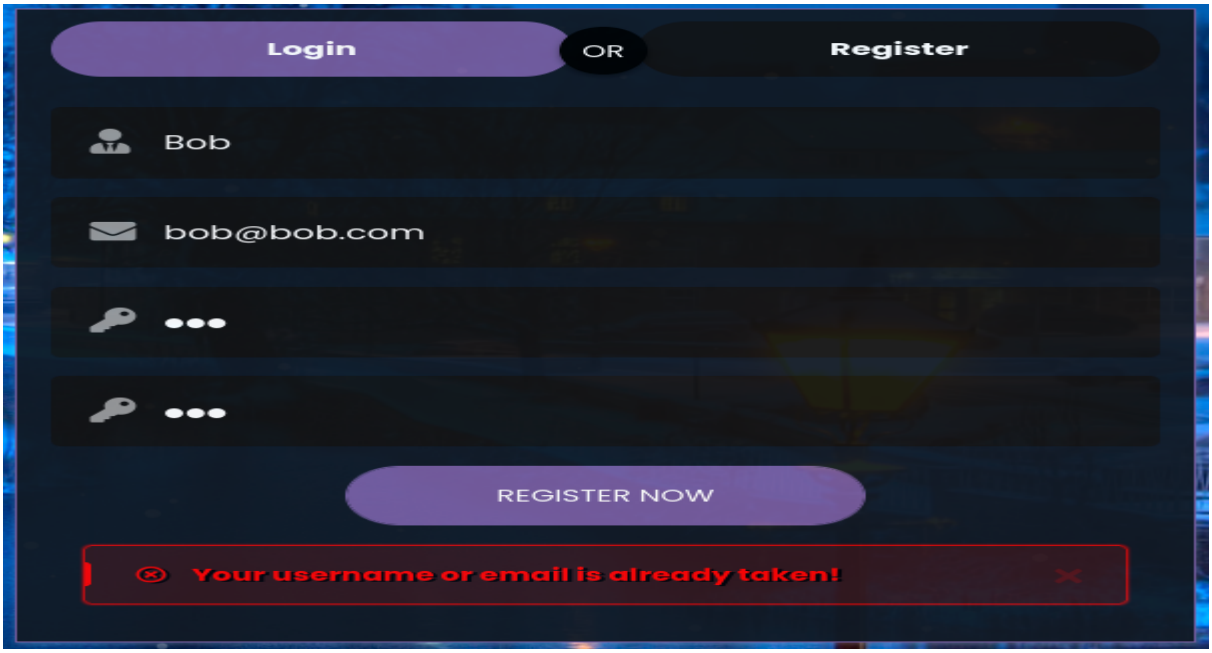

FIGURE 33; USER ENTERING EXISTING USERNAME

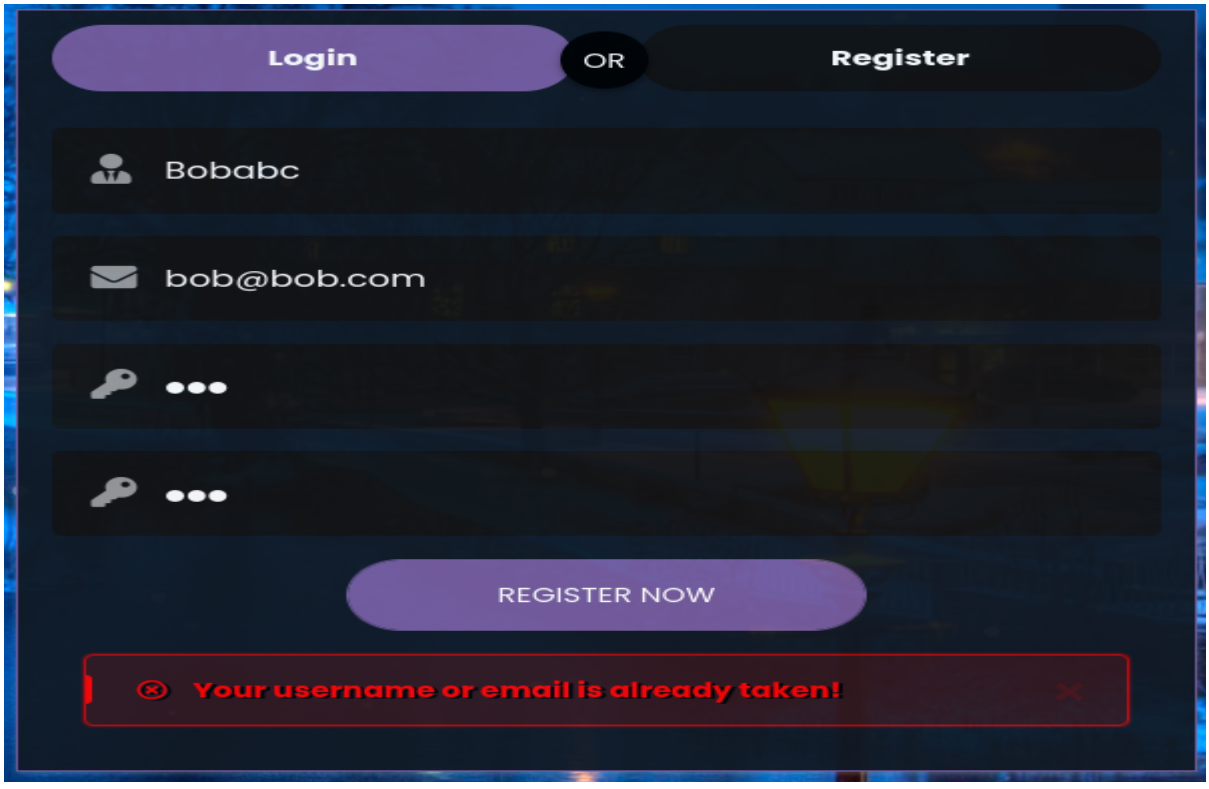

FIGURE 34; USER ENTERING EXISTING EMAIL

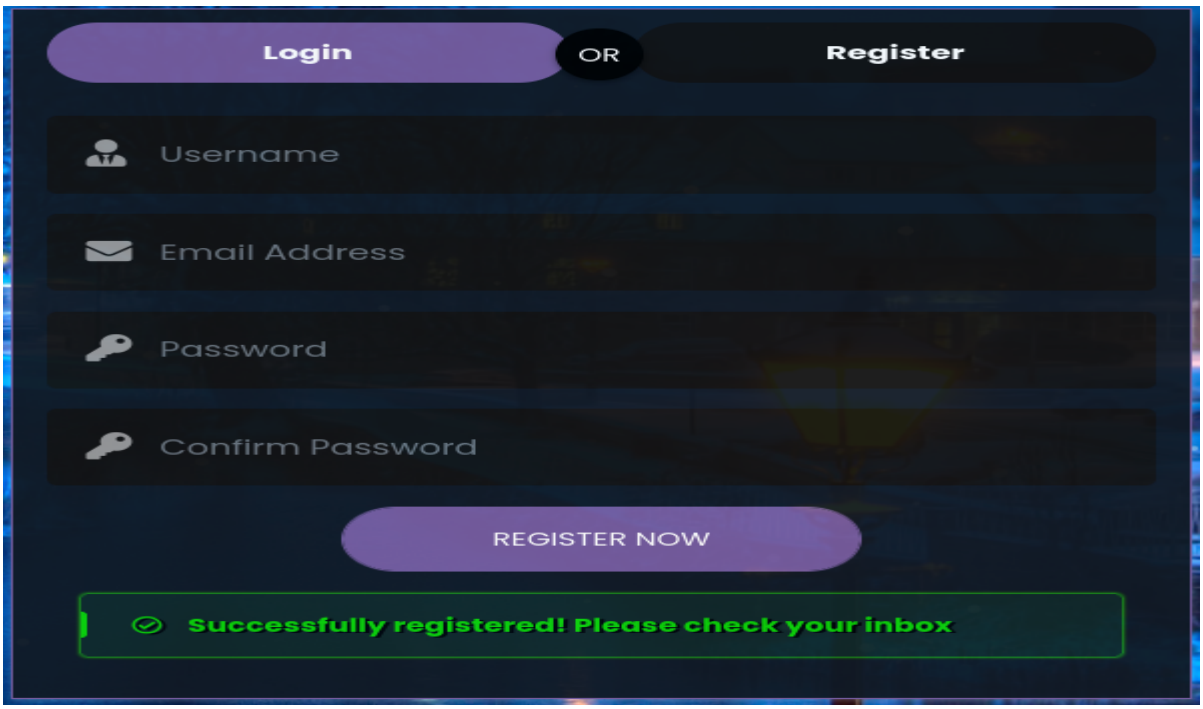

FIGURE 35; SUCCESSFULL REGISTRATION

If user does not verify their account and goes to login:

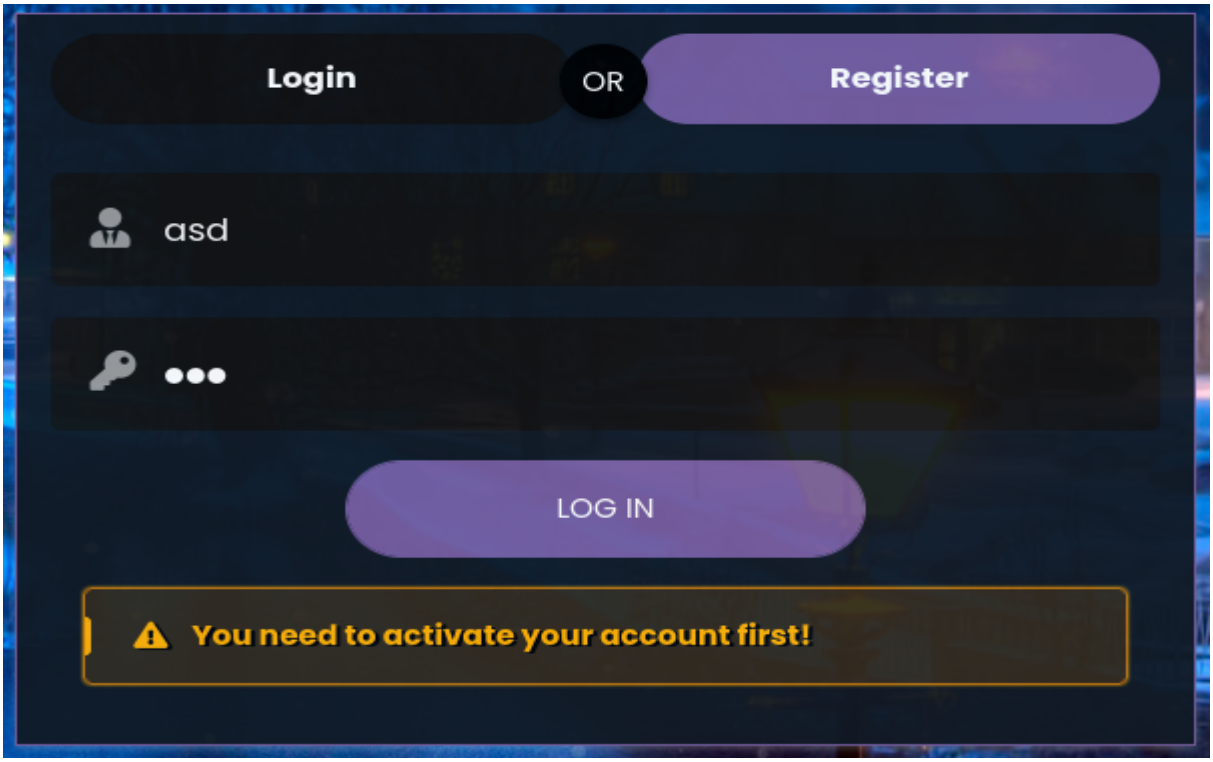

FIGURE 36; USER NOT ACTIVATED LOGIN ERROR

From the logs of the server we can see a url with a token after it:
Your registration is: http://localhost:8080/user/verifyRegistration?token=342b1592-fc65-4073-abb1-44b65b6e3816

If the user clicks on the link:

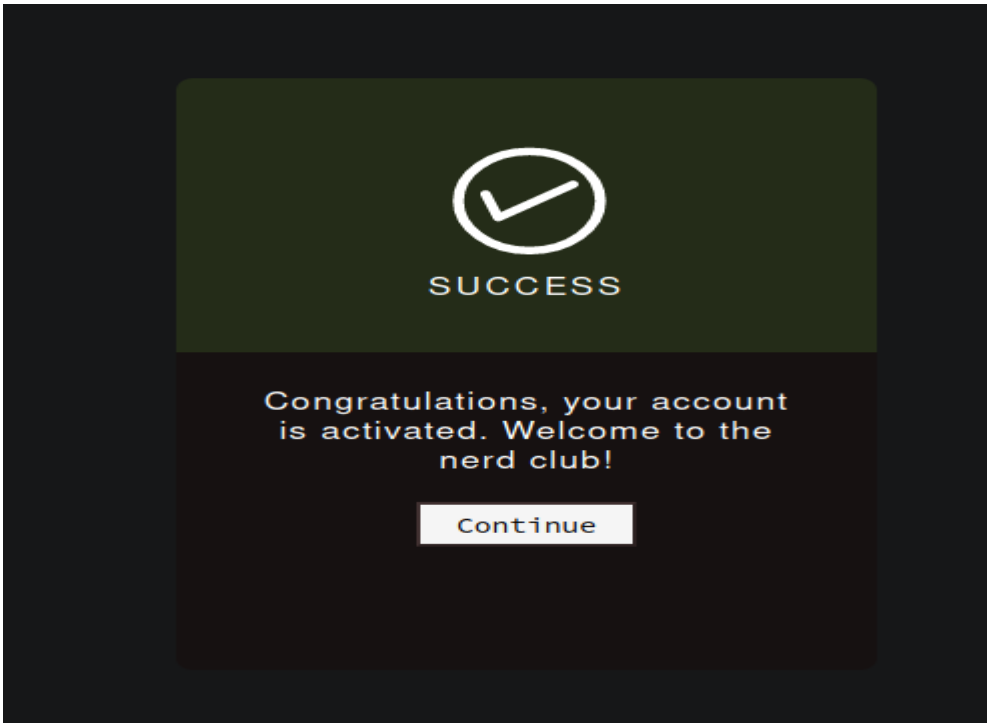

FIGURE 37; SUCCESSFULLY ACTIVATING ACCOUNT

But if the user clicks or refreshes the page again, after activating their account:

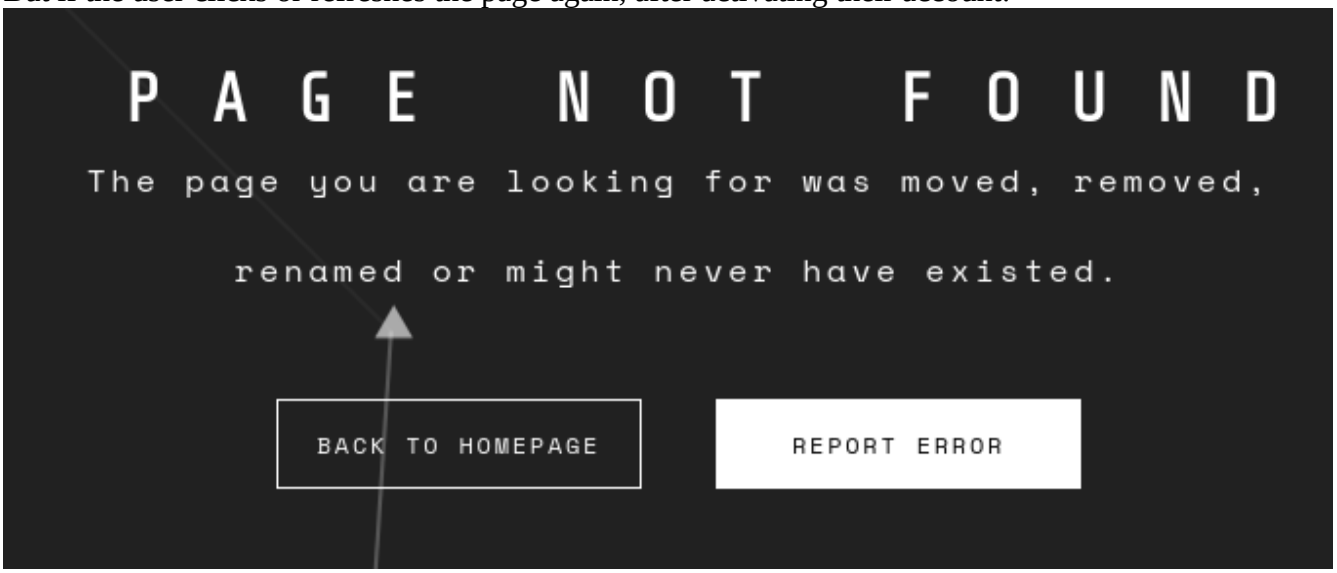

FIGURE 38; USER REFRESHING OR RECLICKING VERIFICATION LINK

The same thing will happen if the link expires, which is about 10 minutes of real clock time.
Logging in the webpage after being activated will directly throw us to the home page:

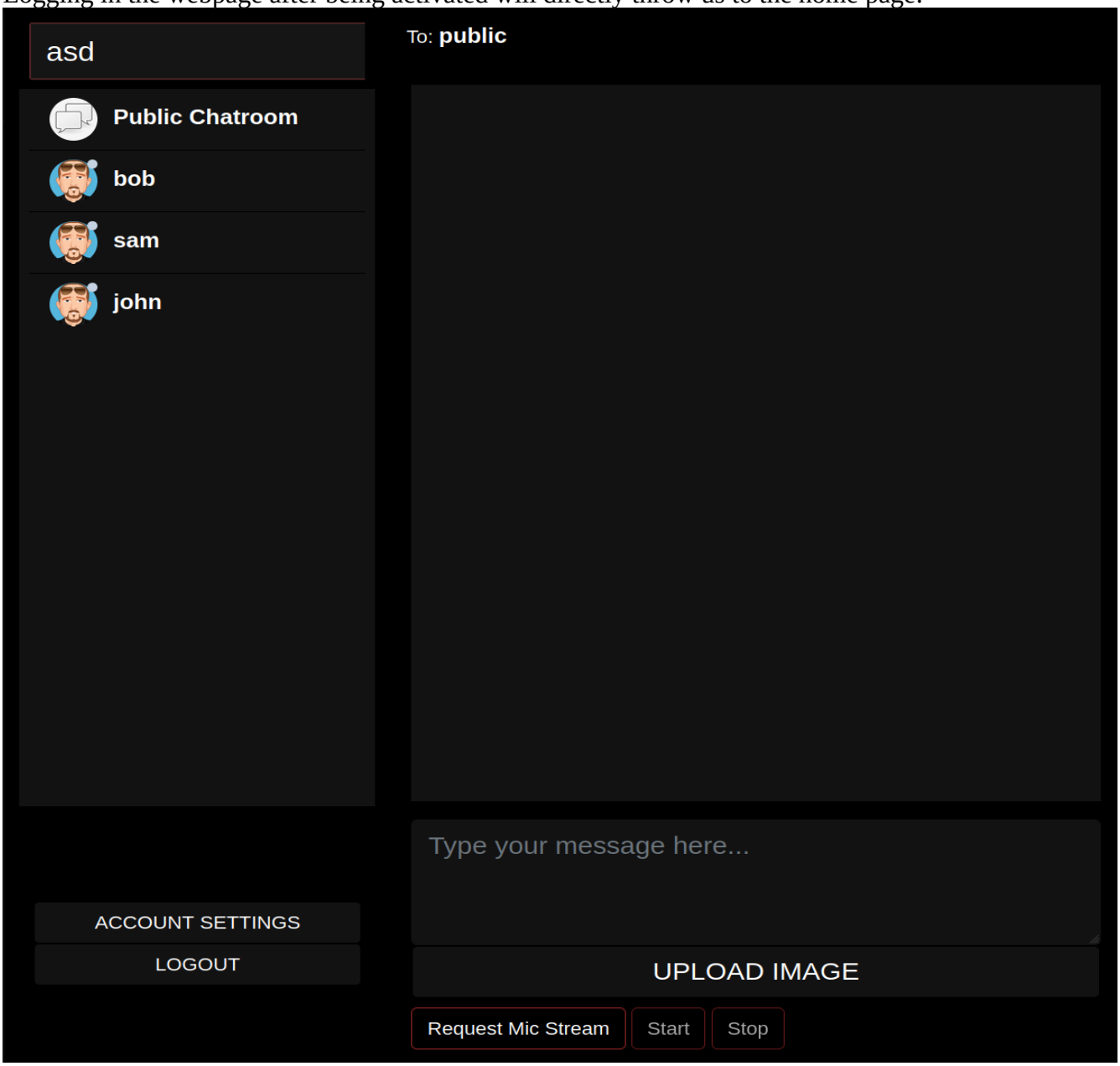

FIGURE 39; CHAT HOME PAGE

As we can see, we have the registered users top left along with their status which is the dot. We have Account Settings and Logout buttons bottom left as well as the chat box, the text input, Upload image button for images and Request Mic Stream to Request for the microphone permission after which the Start button will get activated.
On pressing the start button recording will start and the stop button will go enabled. Upon Stopping the recording, the recording will instantly be uploaded and the start button will go active again with the stop button disabling itself.

Demonstration:

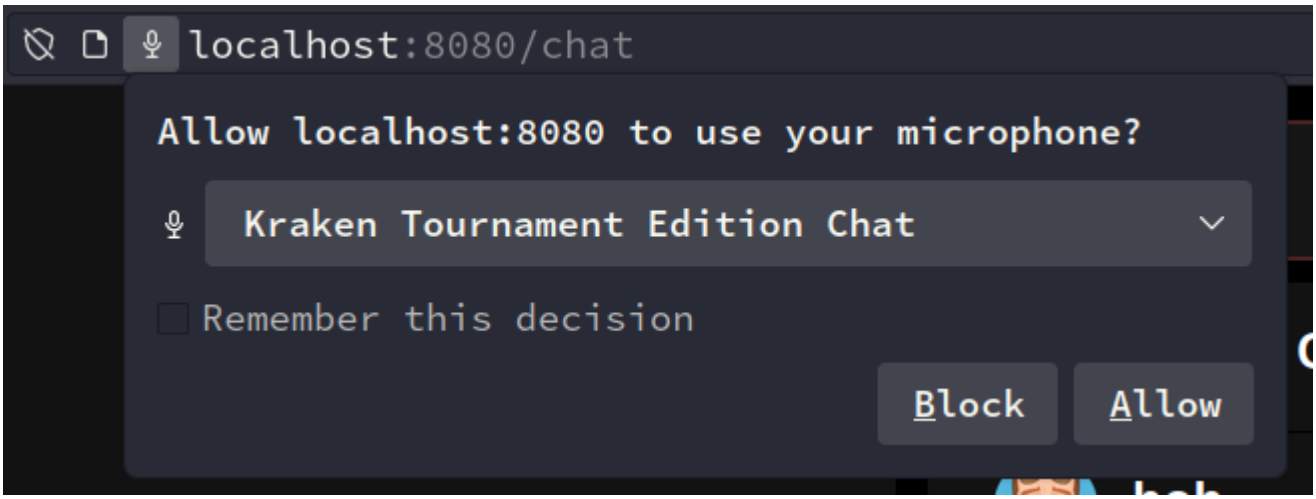

FIGURE 40; REQUEST MIC STREAM BUTTON

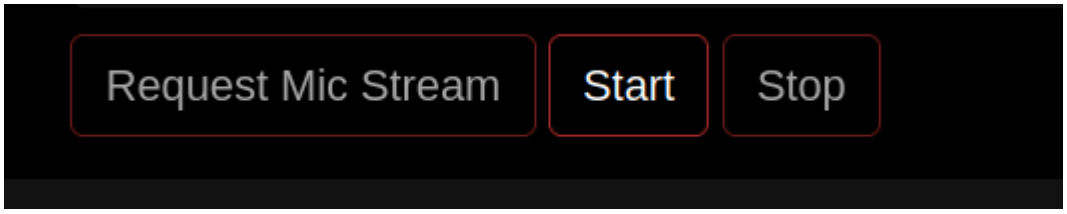

FIGURE 41; ALLOWING MIC STREAM INPUT

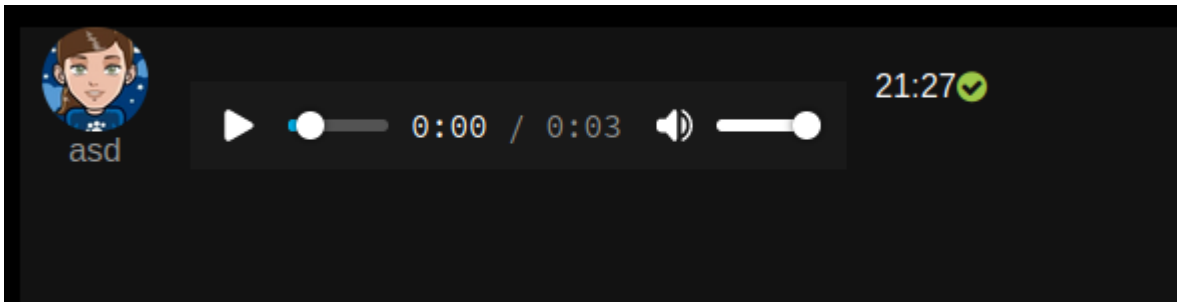

FIGURE 42; RECORDING SOUND MESSAGE AND PRESSING STOP BUTTON

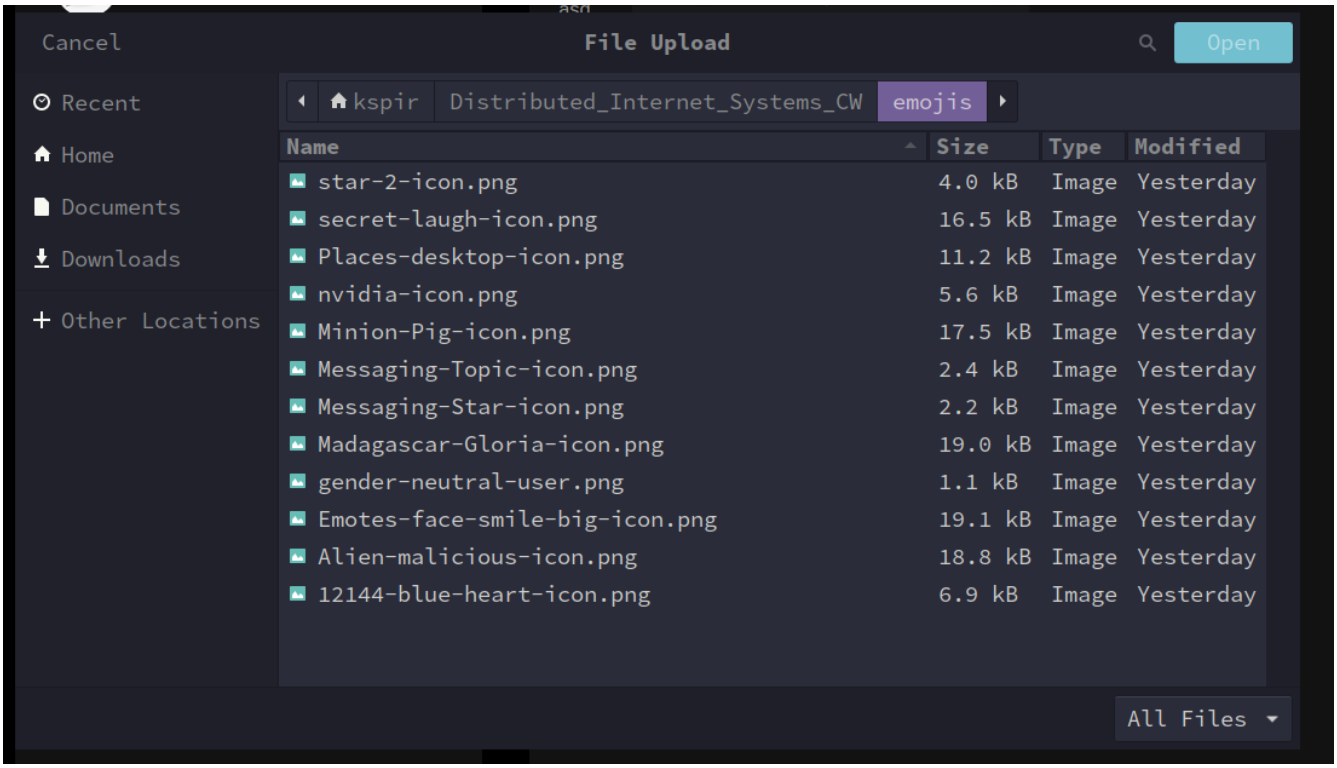

FIGURE 43; PRESSING UPLOAD IMAGE BUTTON

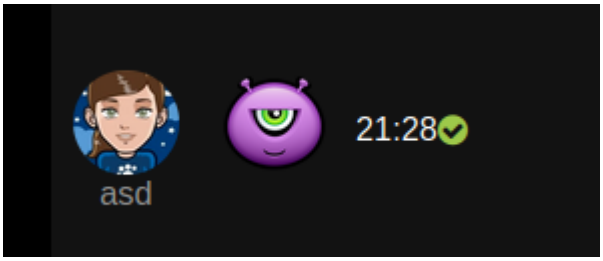

FIGURE 44; UPLOADING IMAGE

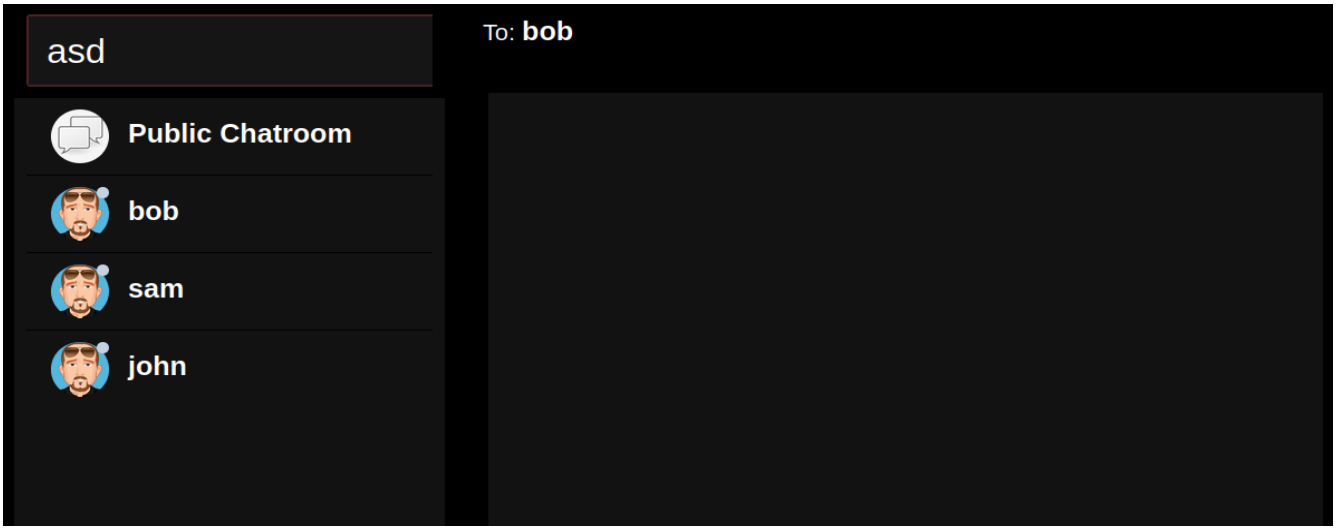

FIGURE 45; SWITCHING CHAT ROOMS

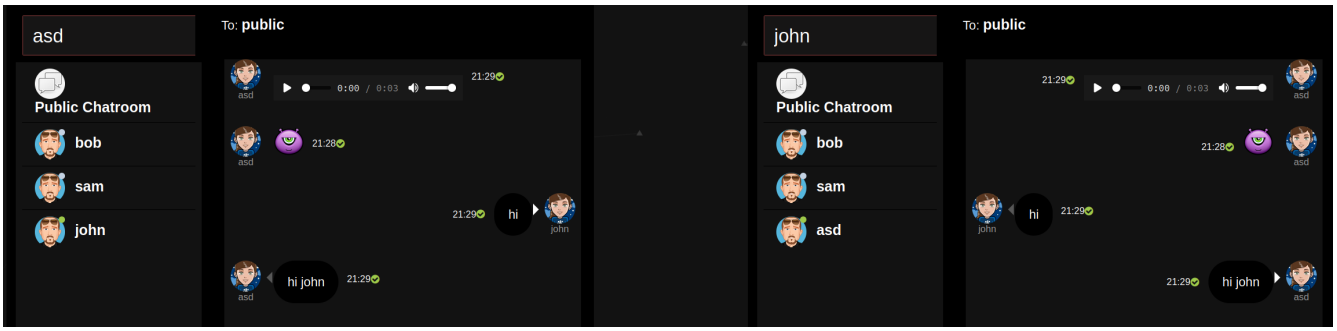

FIGURE 46; SENDING MESSAGES BETWEEN USERS

Notice the green cirle in john and asd usernames. Each one is active for one another.

Public and Private chat room works as expected:

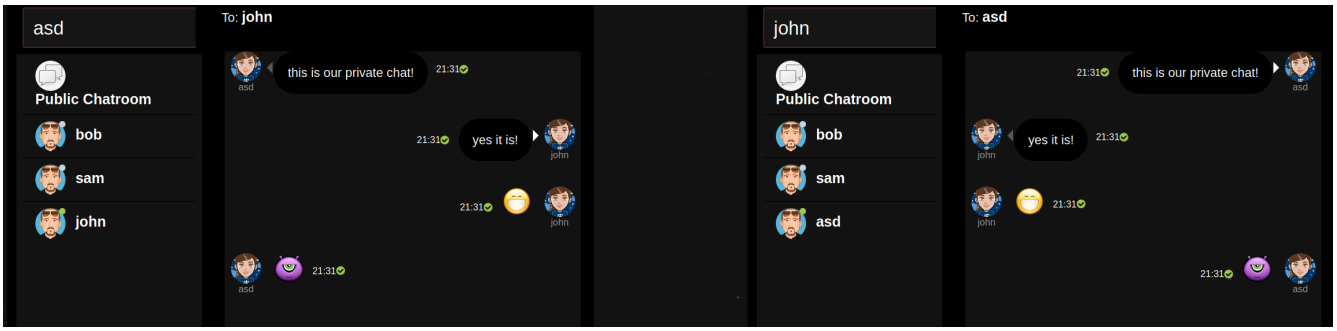

FIGURE 47; PRIVATE CHAT ROOM BETWEEN USERS

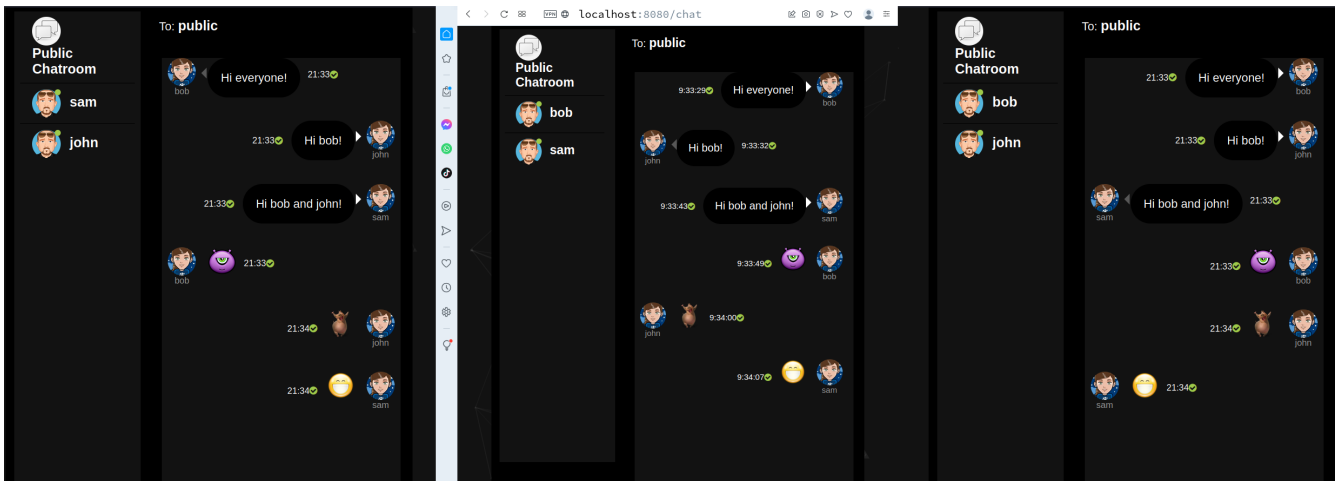

FIGURE 48; PUBLIC ROOM CHAT BETWEEN 3 USERS

Account settings page:

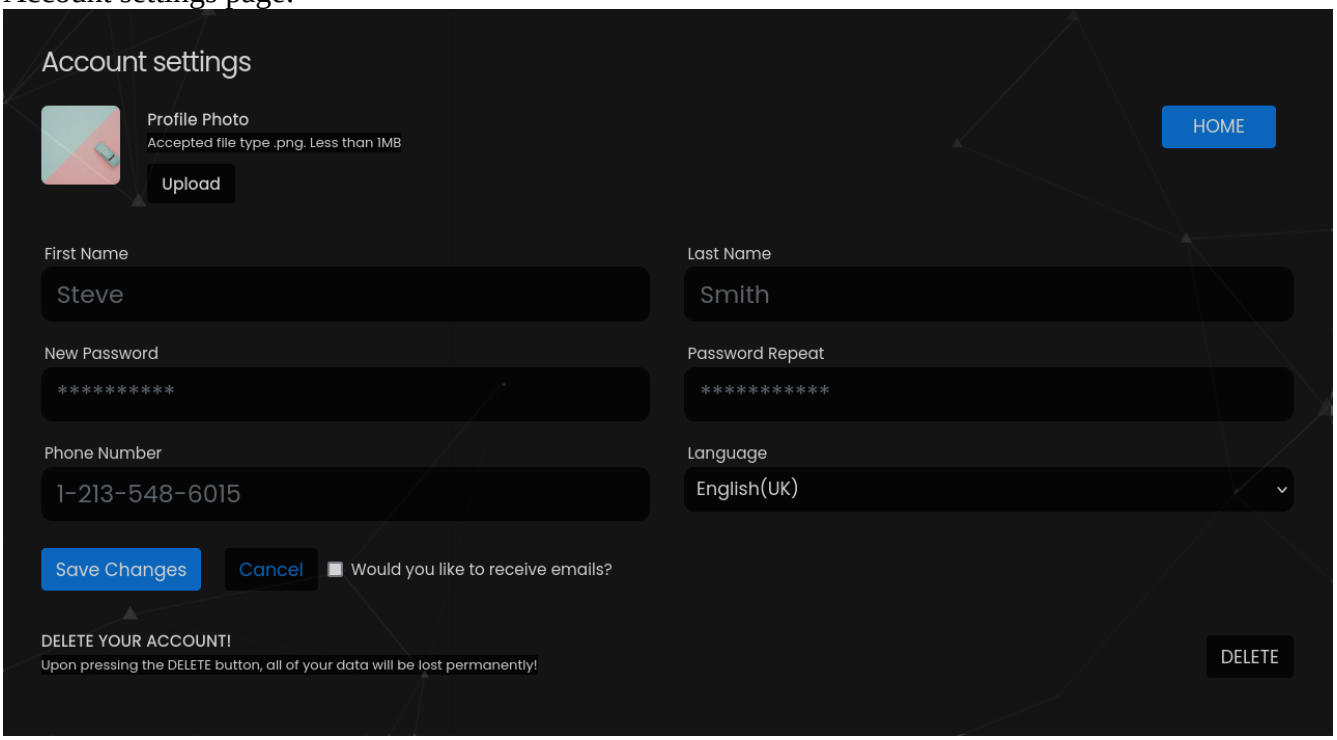

FIGURE 49; ACCOUNT SETTINGS PAGE

I have added no symbols, no numbers or special characters limitations in the First and lastname:

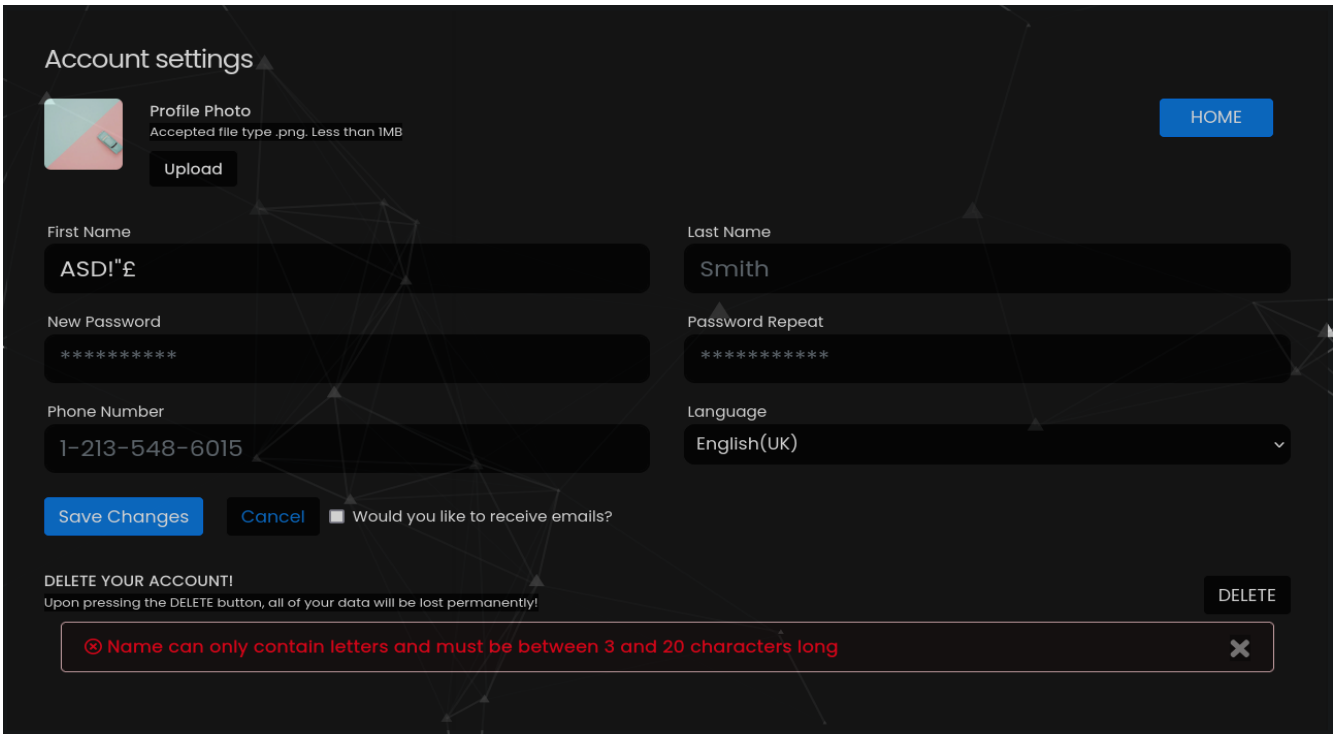

FIGURE 50; INVALID FIRST NAME

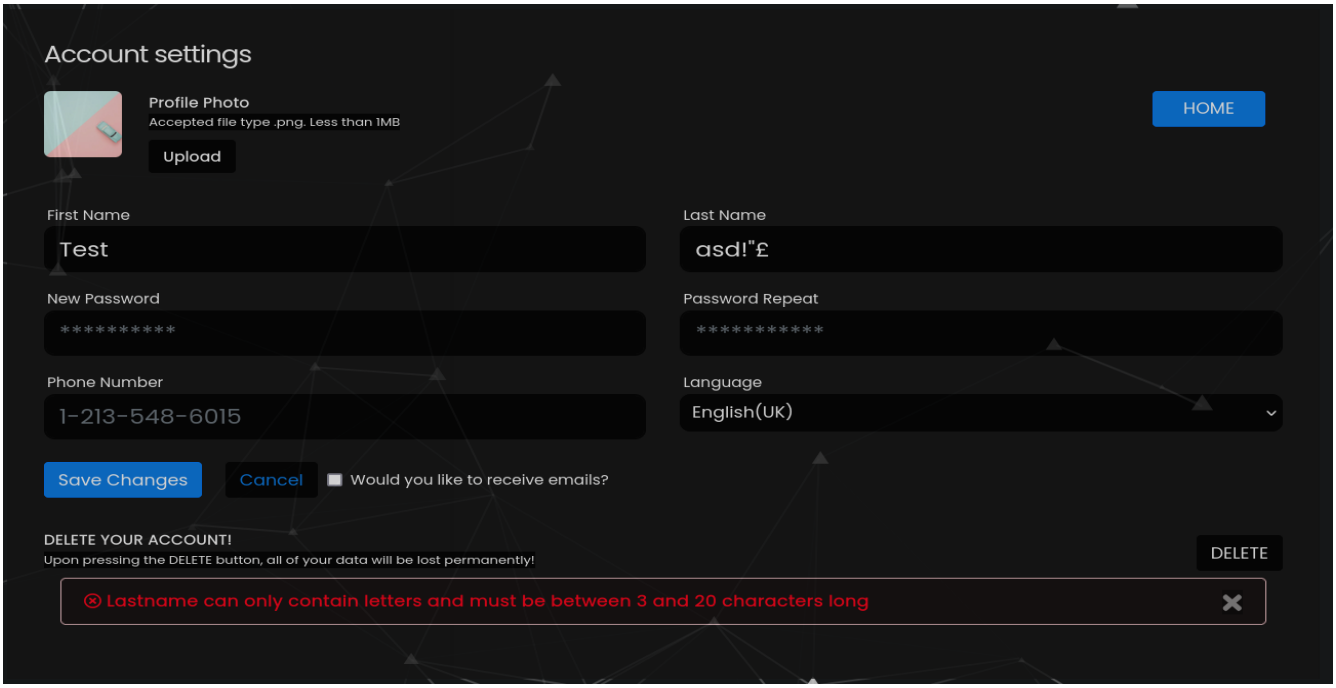

FIGURE 51; INVALID SURNAME

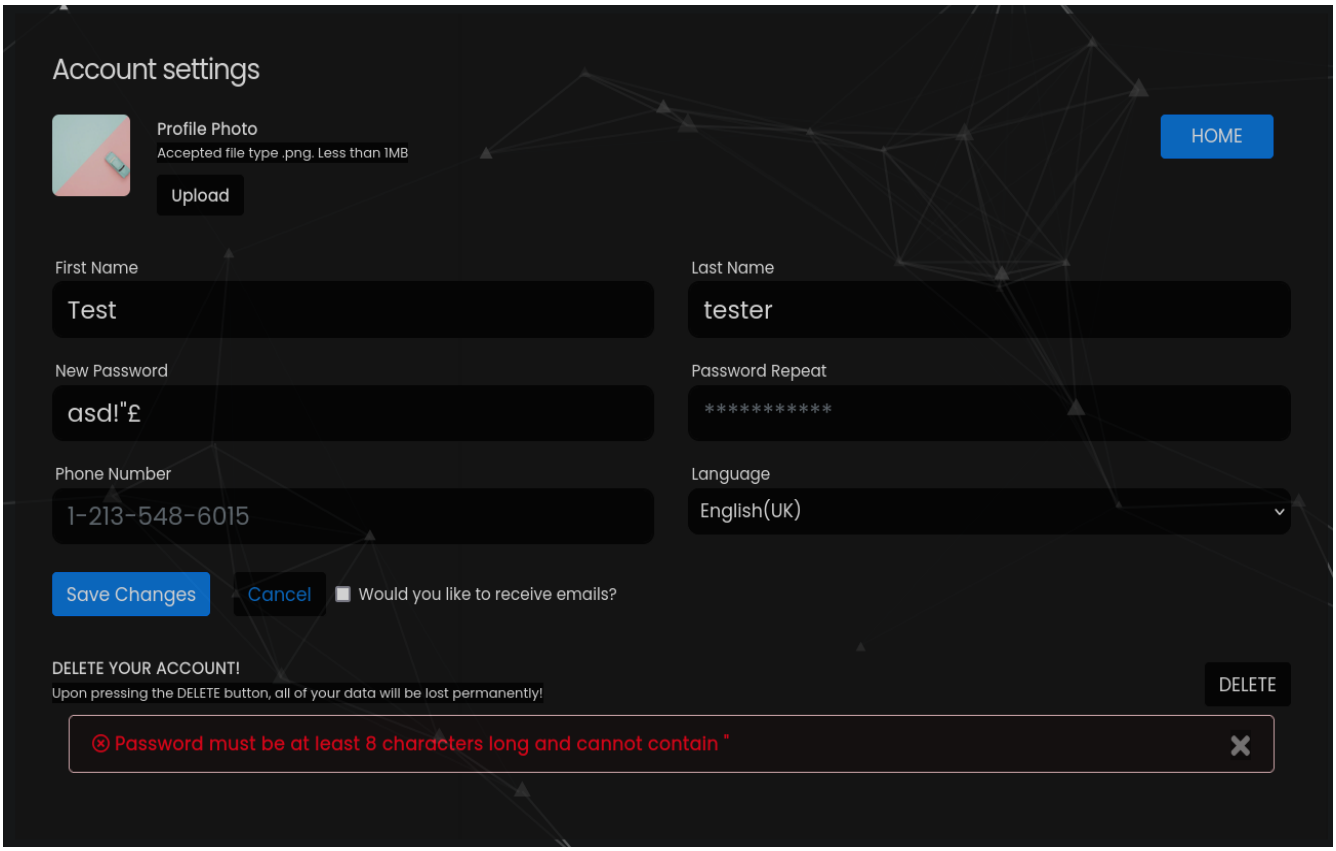

FIGURE 52; INVALID PASSWORD

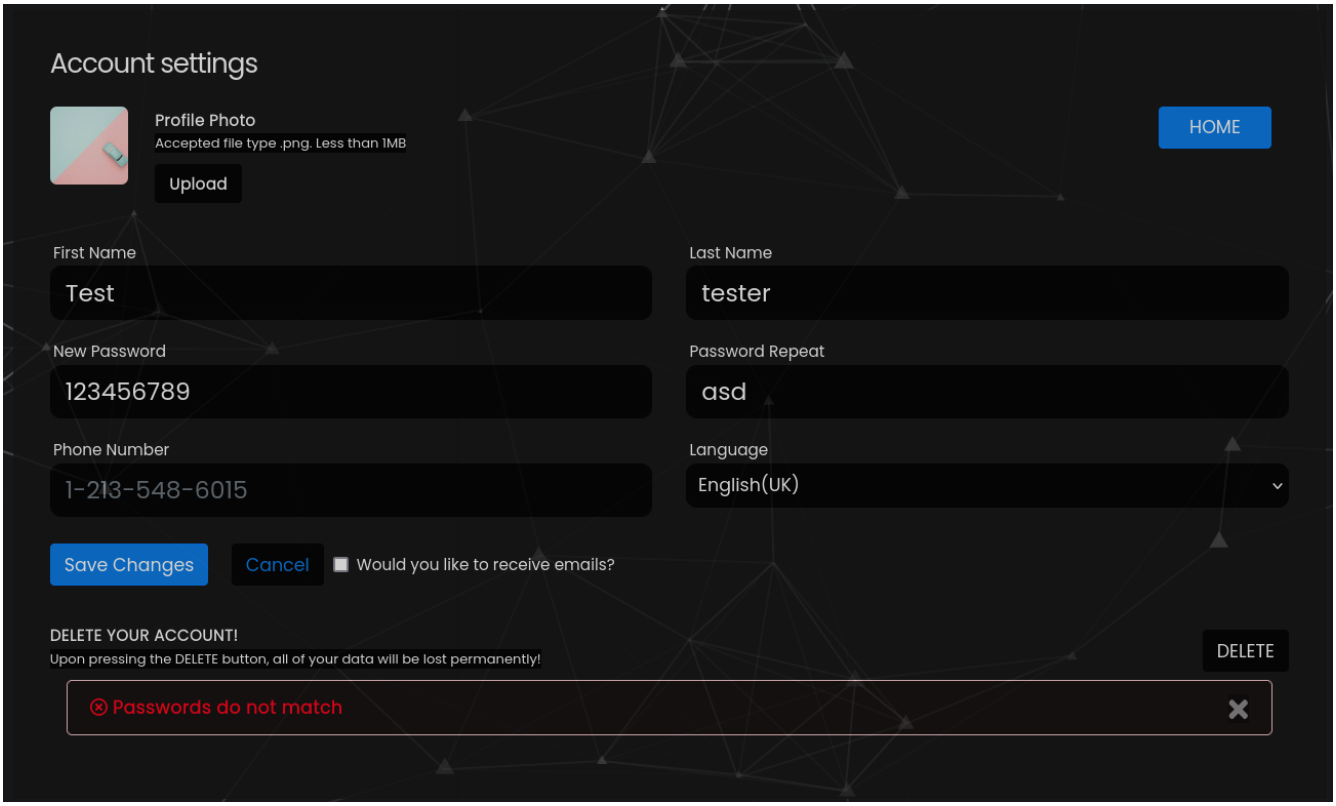

FIGURE 53; PASSWORDS NO MATCH

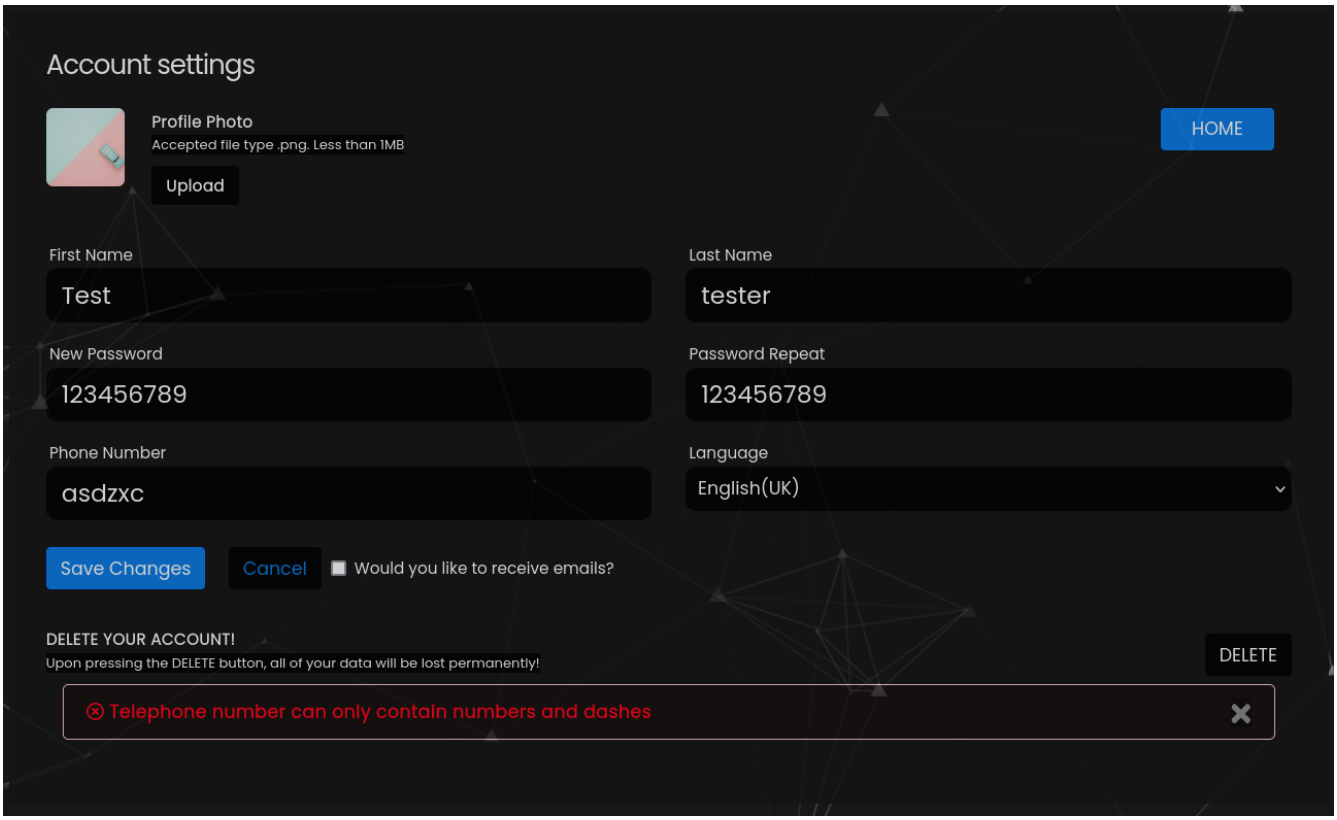

FIGURE 54; TELEPHONE NUMBER INVALID

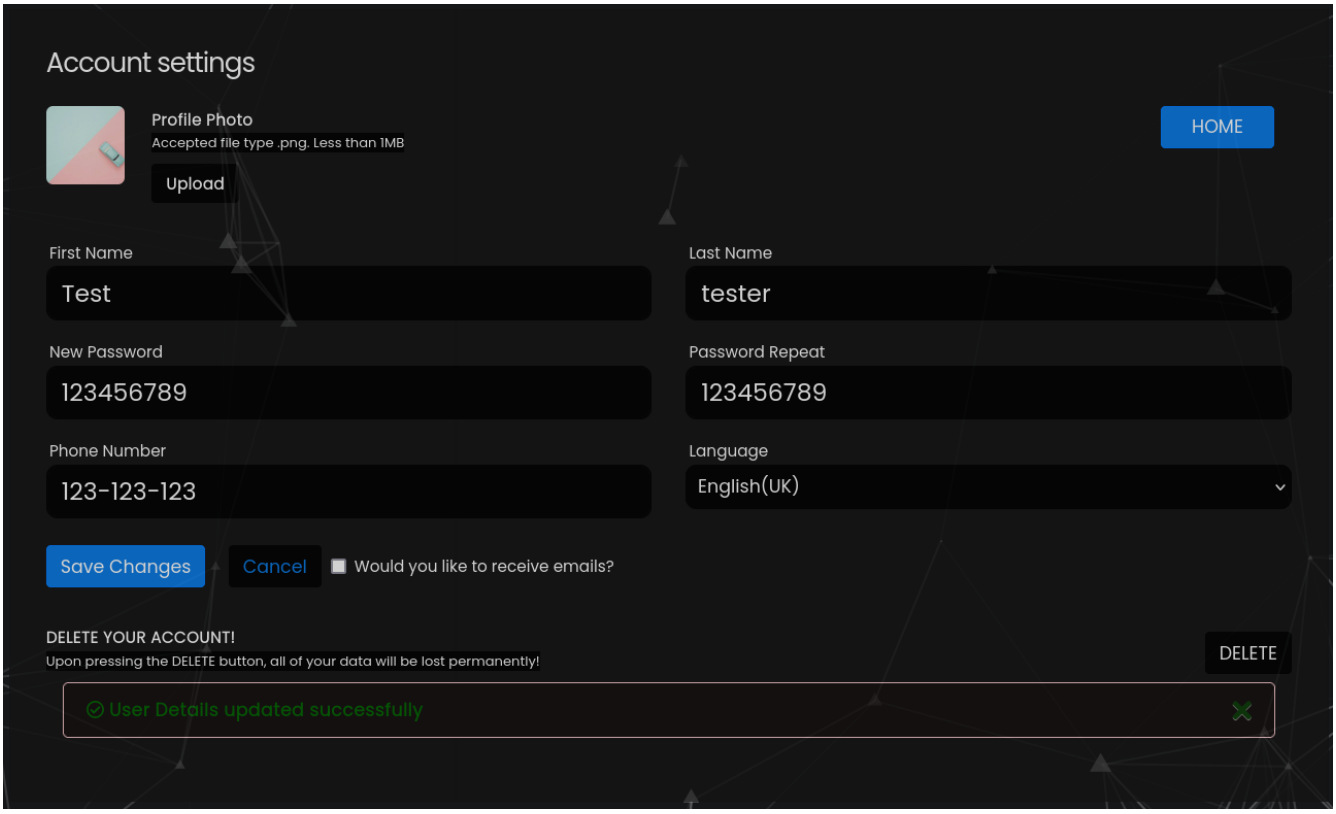

FIGURE 55; USER DETAILS UPDATED SUCCESSFULLY

In addition, I also added user avatars, so users can change their avatar anytime they like!

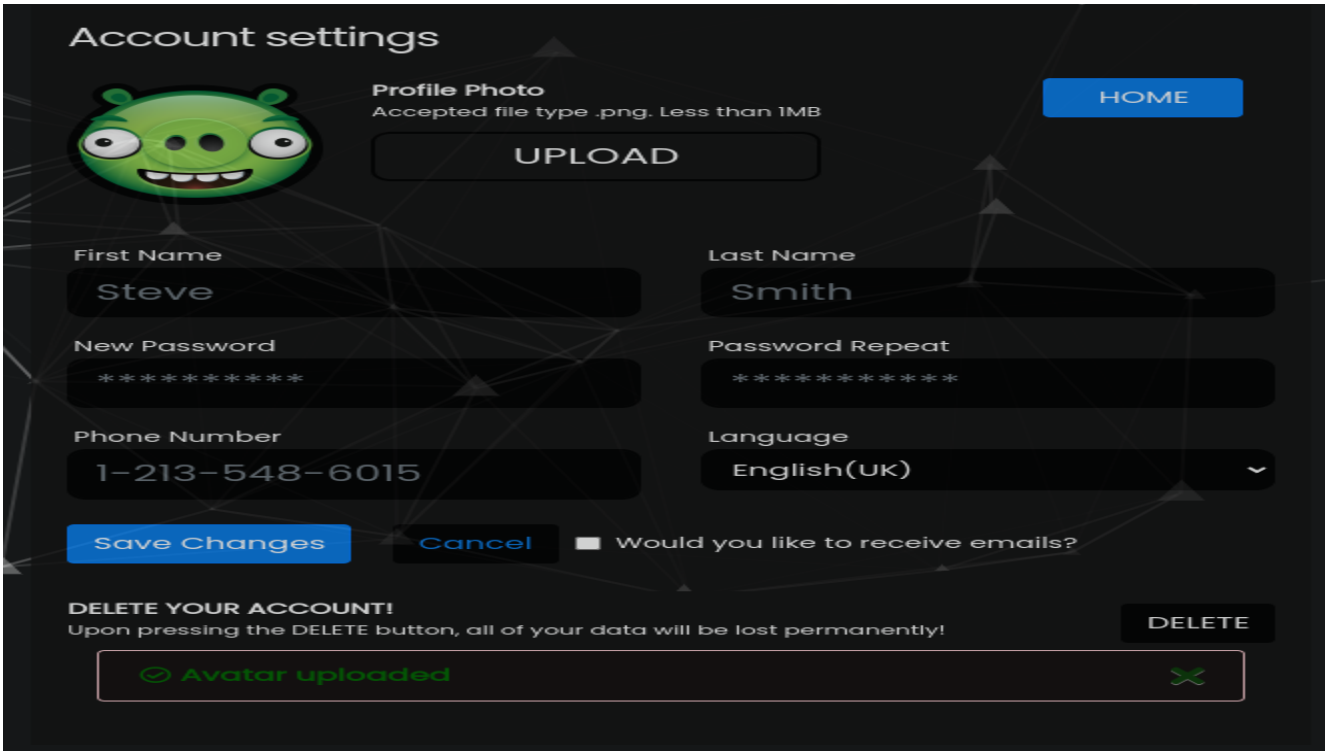

FIGURE 56; USER UPDATING THEIR AVATAR

So other users that are currently online get notified about this change:

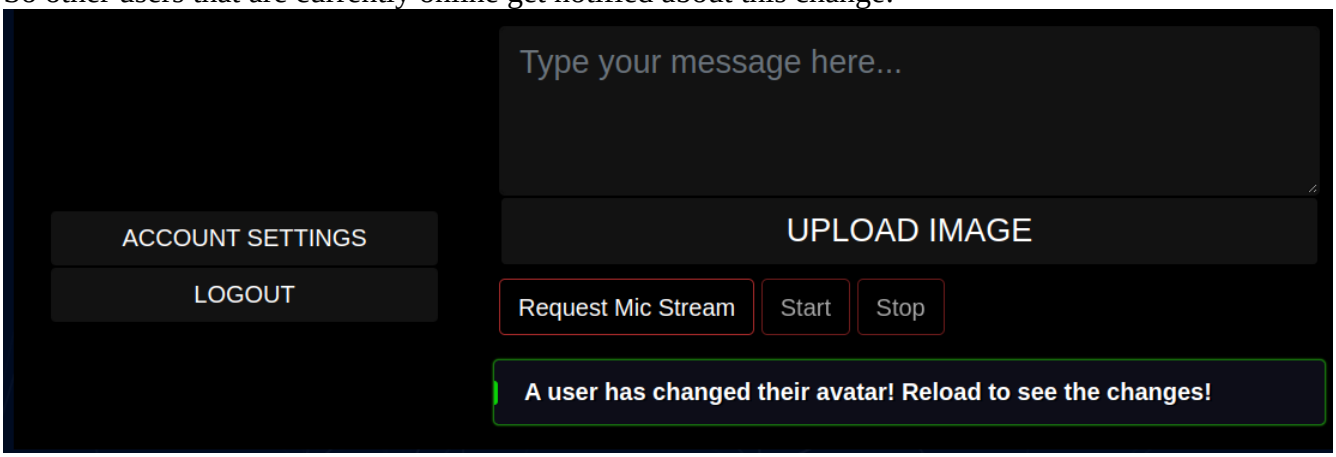

FIGURE 57; ACTIVE USERS NOTIFIED ABOUT CHANGED AVATAR

So the active users can reload their browser and:

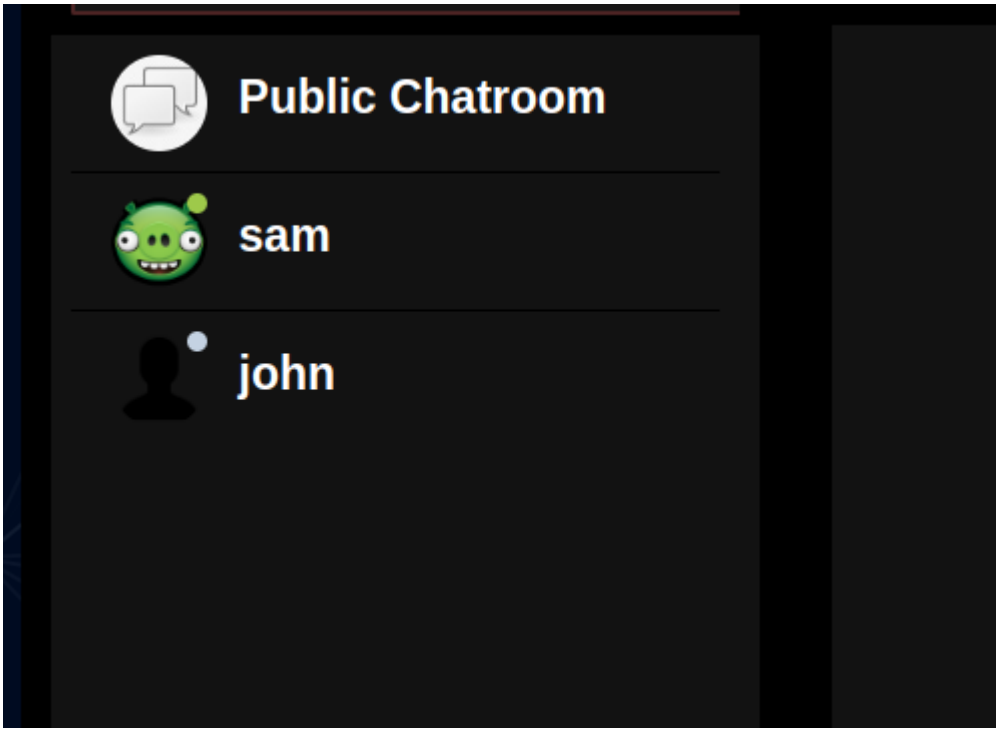

FIGURE 58; ACTIVE USERS RELOADING NEW AVATAR CHANGE

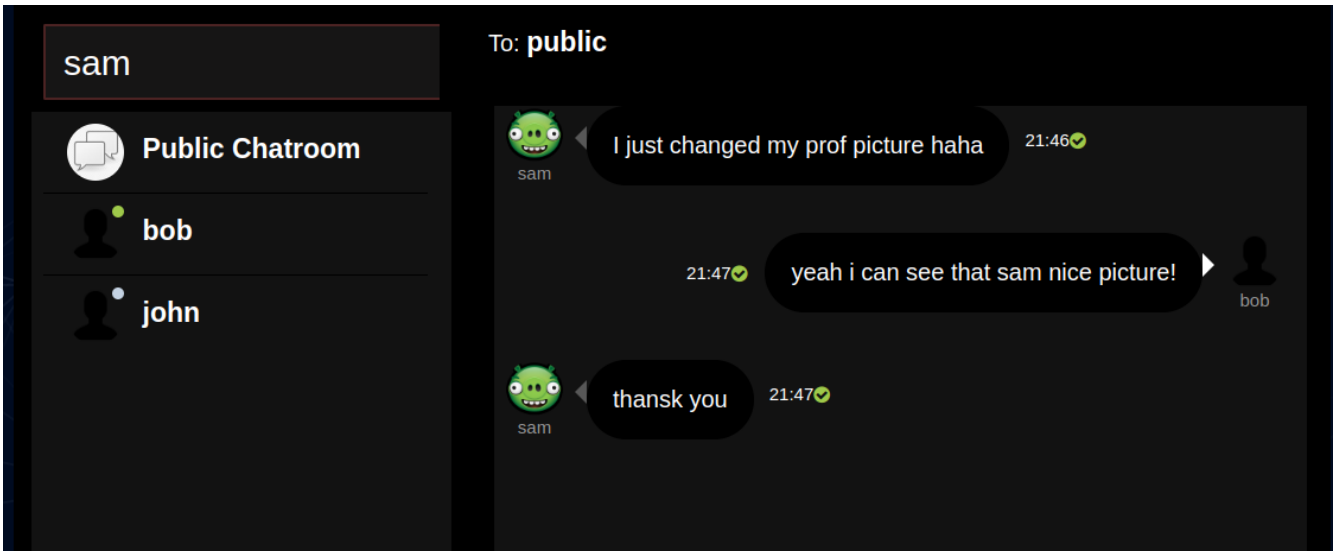

FIGURE 59; UPDATED AVATAR USER PERSPECTIVE

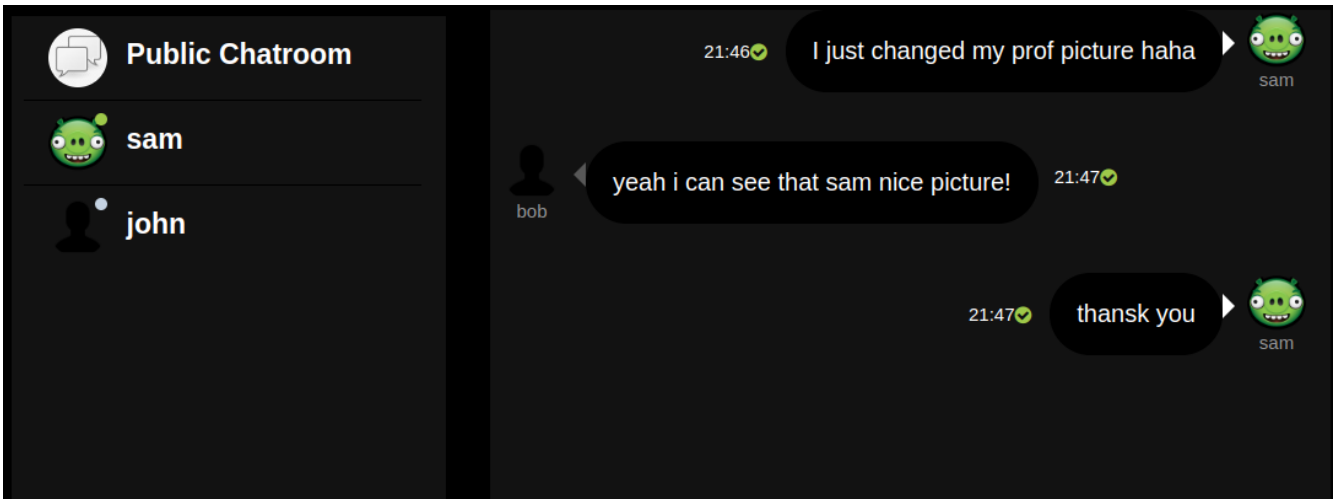

FIGURE 60; OTHER ACTIVE USERS PERSPECTIVE

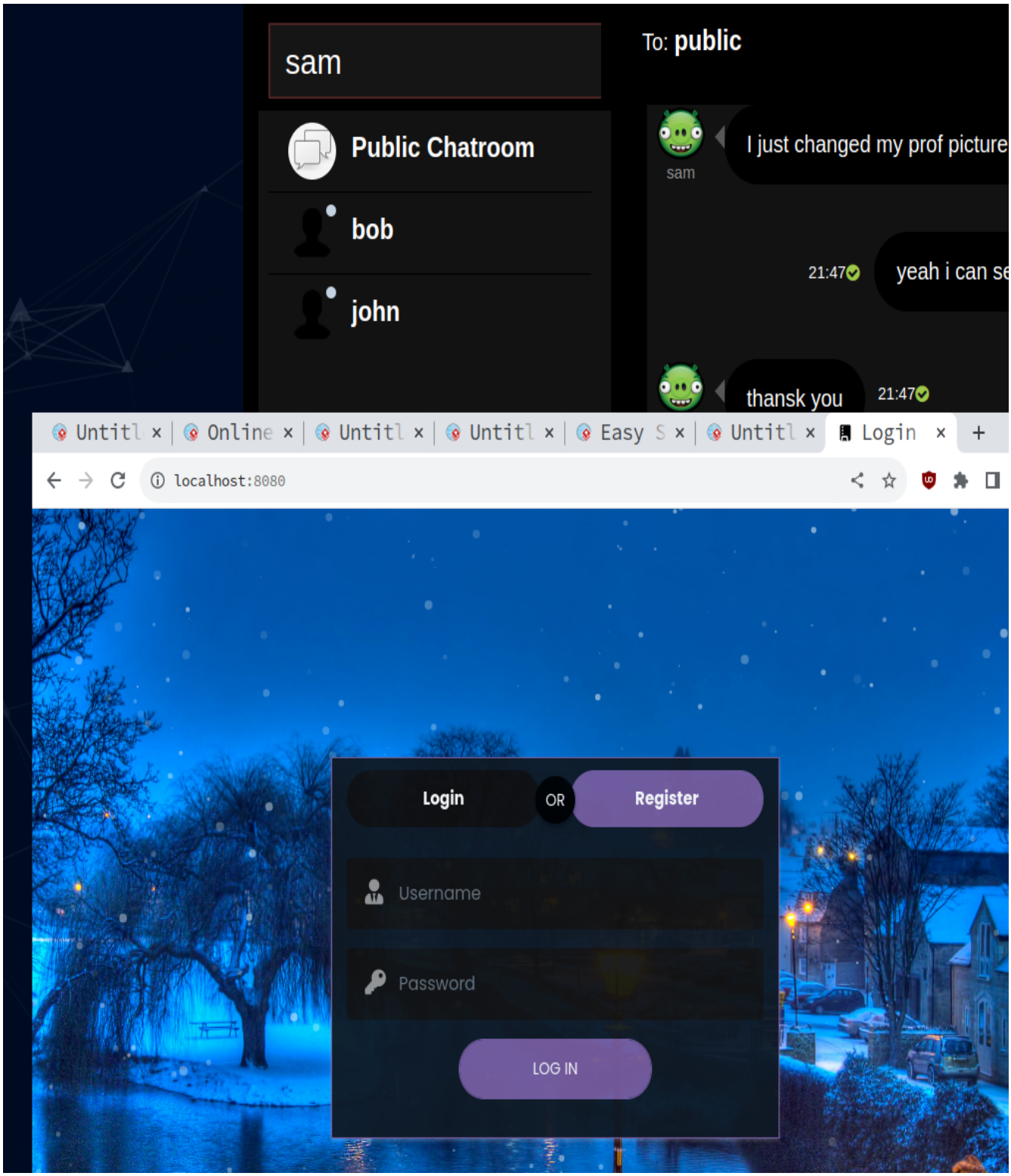

FIGURE 61; USER LOGGING OUT UPDATING USER STATUS TO OFFLINE

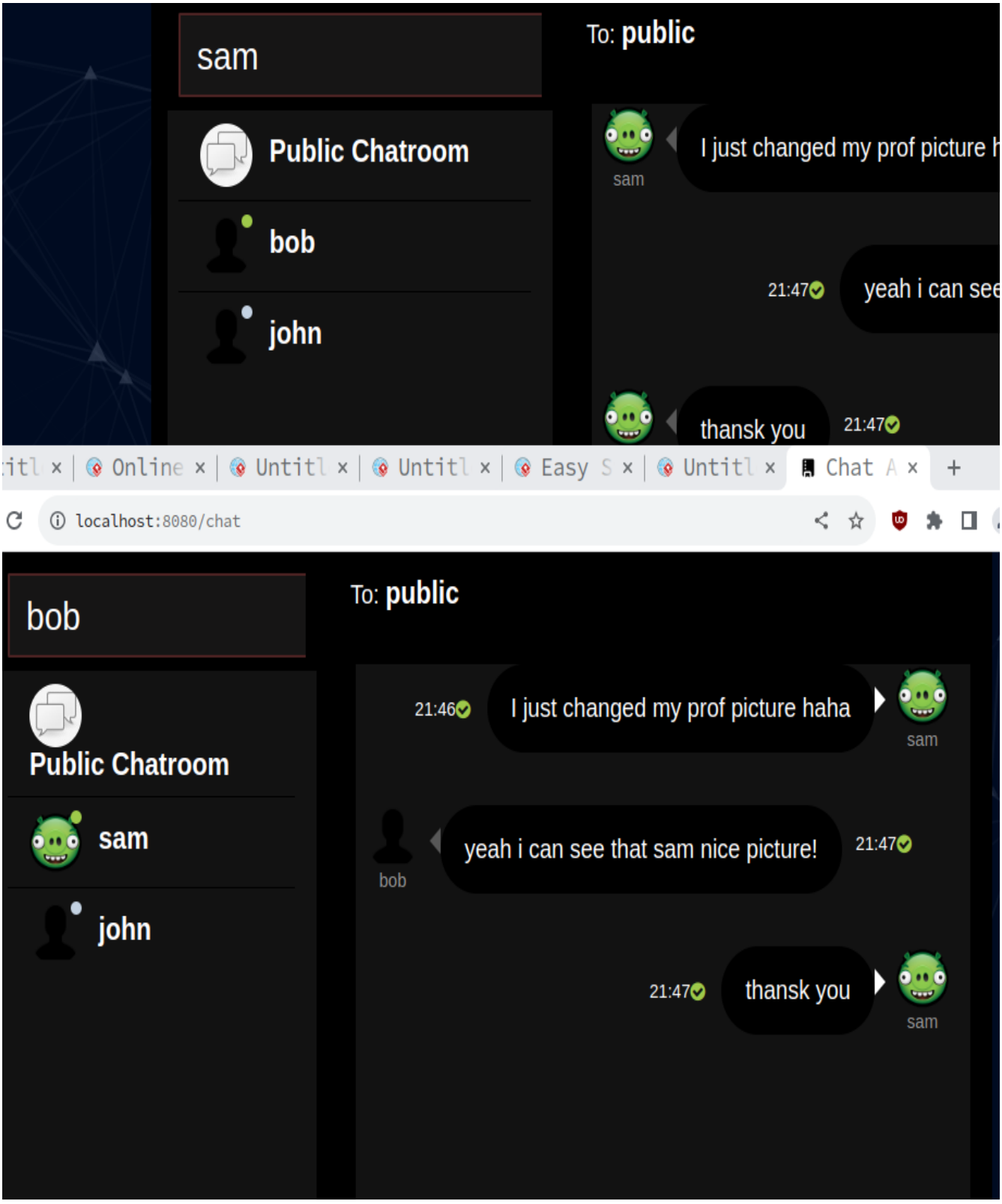

FIGURE 62; USER RE-LOGGING IN UPDATED STATUS

Finally, I have added functionality to delete a user from the Account Settings page:

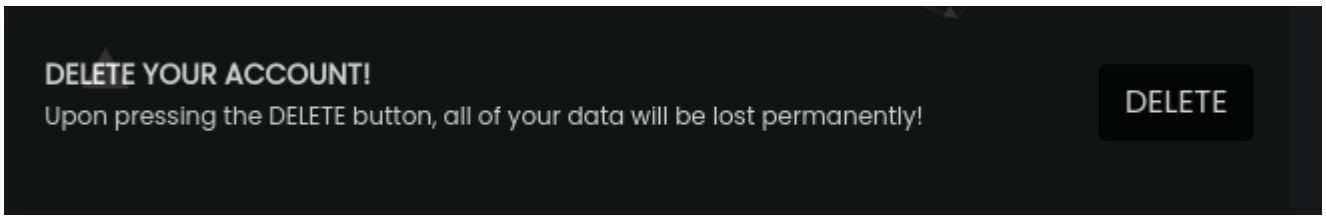

FIGURE 63; DELETING A USER ACCOUNT

CONCLUSION

Overall, I feel that this module has been eye opening to me. I started out with not even being able to understand what a socket is, much less a websocket. Although hard, once properly utilised there are infinite possibilities.

I really like the explanation of the sockets lecture in week 3 and 4 and while not being a mandatory part of the coursework, nonetheless, added websocket implementation for faster data exchange.

Weeks 6 and 7 were especially helpful since I would run into synchronization problems with every little feature I would use. Since I was using async functions in addition to AJAX, it was very easy to end up with null variables. Worst case scenario was a bug that bombed the server with 100requests/second. Springboot could still handle it though.

Although throught the whole project I never used xml, but json, I did want to use raw binary data. It was impossible though so I had to do some base64 conversion in order for the backend to understand data such as messages sent from the frontend.

This project was solely done by depending on lecture and lab material as well as documentation for the different libraries I have used. Under no circumstances did I ever receive external help or “ready code” without me referencing the appropriate sources.

KEYWORDS & CODE TAGS

AJAX = Asynchronous Javascript And Xml
@Column = specifies that the variable is part of a table
@Table = specifies that the class is a table in a database
@Getter = produces all of the class getter() methods for all variables
@Setter = produces all of the class setter() methods for all variables
@AllArgsConstructor = produces all of the class constructors with all possible variables as inputs
@NoArgsConstructor = produces an empty constructor
@ToString = produces the toString() method which prints the object's data in a string
@Controller = specifies that the class is a url to method mapper
@Service = specifies that the class acts as middleware service
@Configuration = specifies that this class is a config class and should be firstly compiled and loaded
@Repository = specifies that this class is a repository man-in-the-middle handler
@Bean = specifies that the method/class is a special bean type
@Component = specifies that the class will override Springboot's default component config
@EnableWebSecurity = uses the WebConfigurerAdapter library for security, session ,validation and etc
@Entity = specifies that the class is an entity in a table inside of a database
@CreationTimeStamp = specifies that the variable is a DateTime variable and is initiated on creation
@EnableWebSocketMessageBroker = enables the ability to configure springboot's websocket config
@Query = creates customized queries and maps them to a method inside of a repository class
@Id= specifies that the variable will be a primary_key in the table in the database
@SequenceGenerator = specifies the sequence at which the id will be generated
@PostMapping = sets the type of the HTTPRequest of a method to POST
@GetMapping = sets the type of the HTTPRequest of a method to GET
@PutMapping = sets the type of the HTTPRequest of a method to PUT
@DeleteMapping = sets the type of the HTTPRequest of a method to DELETE
@RestController = sets the class to act as a restfull controller
@RequestMapping = requests for the current class mapping to start from a specific value
transient = is a variable type which will disallow initiating that variable in the database
Optional<?> = specifies that the function may or may not return a result(or return null)
ArrayList<ArrayList<String>> = 2d arraylist
@SpringbootApplication = enables the Springboot Framework support for the server

REFERENCES

Building an Application with Spring Boot [WWW Document], n.d. URL https://spring.io/guides/gs/spring-boot/ (accessed 12.5.22).

Full details and actions for Distributed systems: concepts and design [WWW Document], n.d. URL https://www.vlebooks.com/Product/Index/438073?page=0 (accessed 12.5.22).

Full details and actions for Introduction to security. [WWW Document], n.d. URL https://www.vlebooks.com/Product/Index/1310136?page=0 (accessed 12.5.22).

Krutz, R.L., Vines, R.D., Brunette, G., 2010. Cloud Security: A Comprehensive Guide to Secure Cloud Computing. John Wiley & Sons, Incorporated, Hoboken, UNITED STATES.

Spring Boot [WWW Document], n.d. URL https://spring.io/projects/spring-boot (accessed 12.5.22).