ADVANCED WEB APPLICATIONS PROJECT WORK DOCUMENTATION

CT30A3204

25.02. 2024

Karoliina Aaltonen, 0612213

Table of Contents

1 List of features 2
1.1 Basic features 2
1.2 Additional features 14
1.3 Table of features and attempted points 22
2 Program description 23
3 Running the program 24
4 Dependencies 24

1 List of features

1.1 Basic features

Feature	Status
Users can login	Functional
Users can register	Functional
Users can like other users	Functional
Users who like each other can send messages	Functional
Responsive design	Functional
Utilization of database (MongoDB)	Functional
Backend with Node.js	Functional
Authorization (JWT)	Functional

The default route of the program "/" leads to the login view:

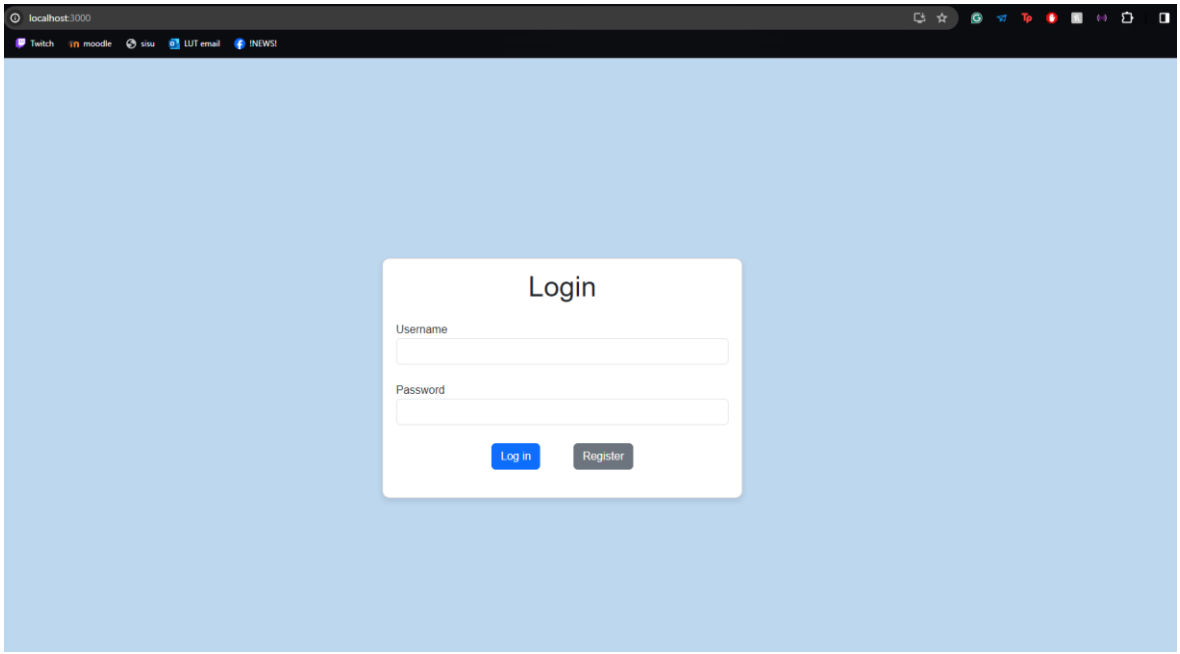

Image 1: Log in view.

If the user tries to log in with a non-existing user or the password is wrong, an error message is shown when “Log in” button is pressed:

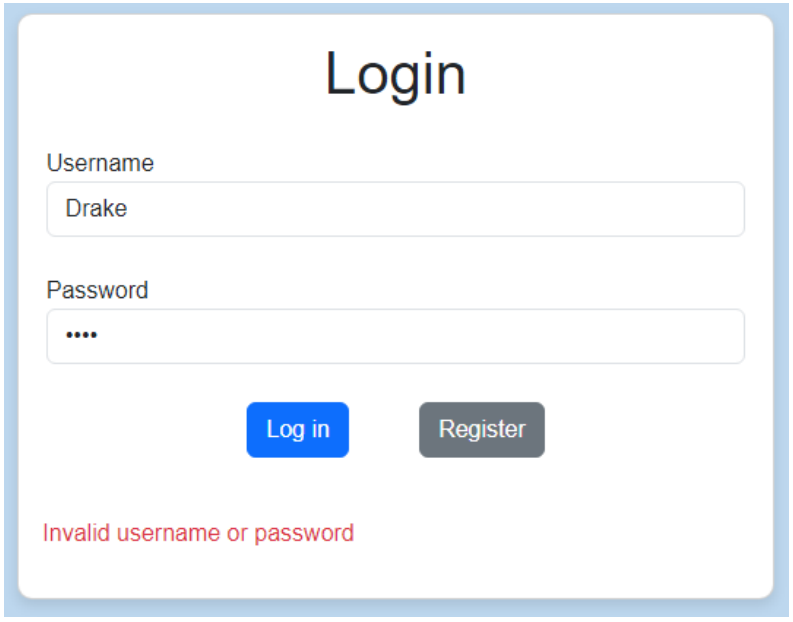

Image 2: Invalid log in information.

By clicking “Register” button, the user is redirected to route /register where they can create their user:

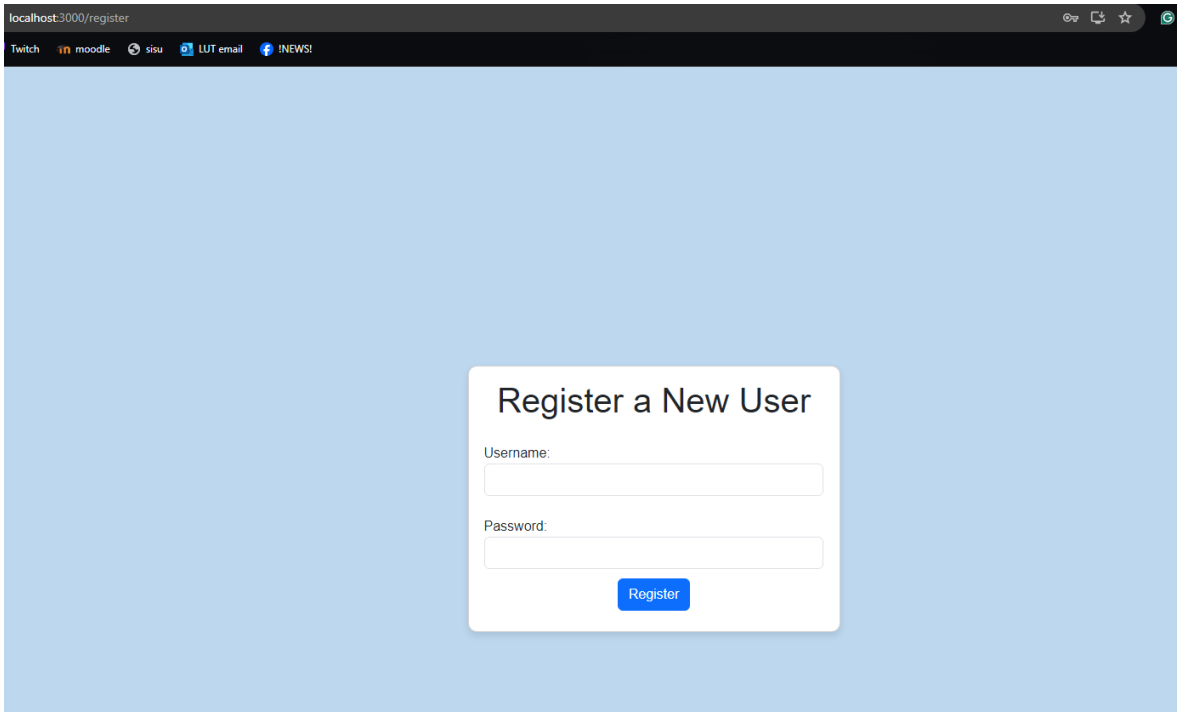

Image 3: Registration view.

Each username must be unique, I already have a user called Shrek, so if I try to make another one, I get an error:

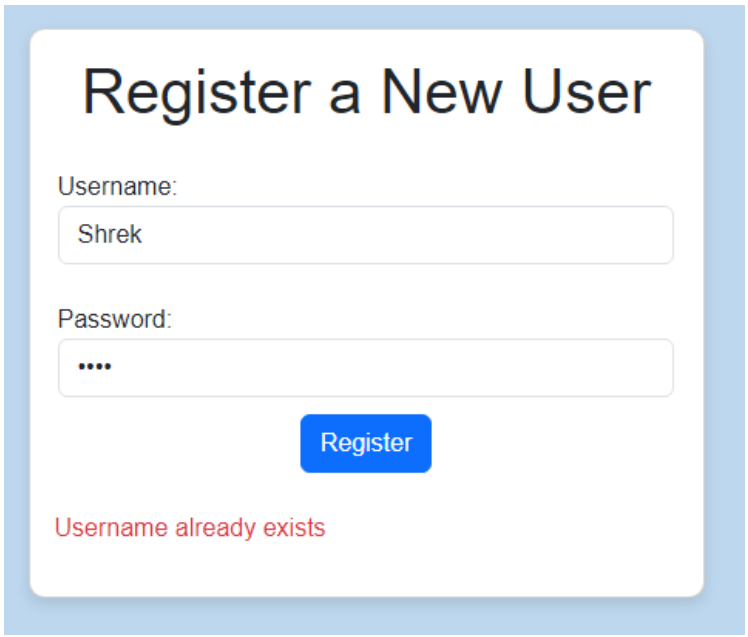

Image 4: Username already exists.

If the username is accepted, the user is directed back to the root to the login page and they may now log in.

With successful login, the user is redirected to /main route which acts as the main page of the app. The header displays the name of the logged in user and by clicking the text the app navigates to /main. The header also contains links “Edit Your Info”, “Display Chats”, and “Log Out”. The main view fetches random users from the program’s user database keeping in mind who is logged in so that no one is suggested their own profile. Underneath the displayed profile there are buttons “Like” and “Dislike”:

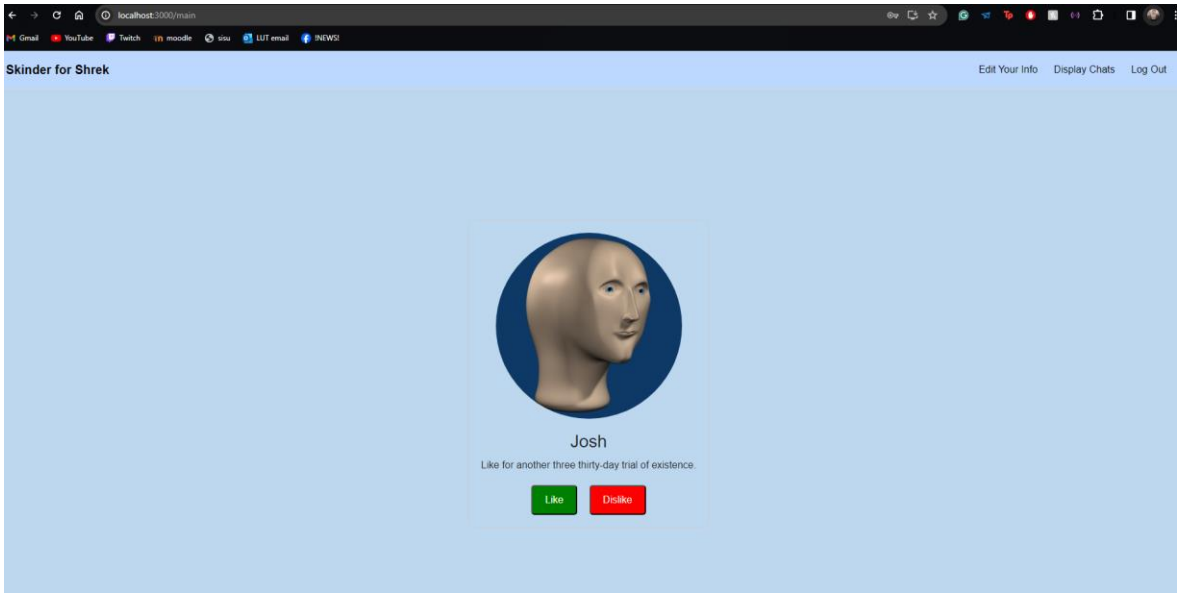

Image 5: Main view.

If I press “Like” on Josh’s profile (I am logged in as Shrek), another suggestion is shown to me:

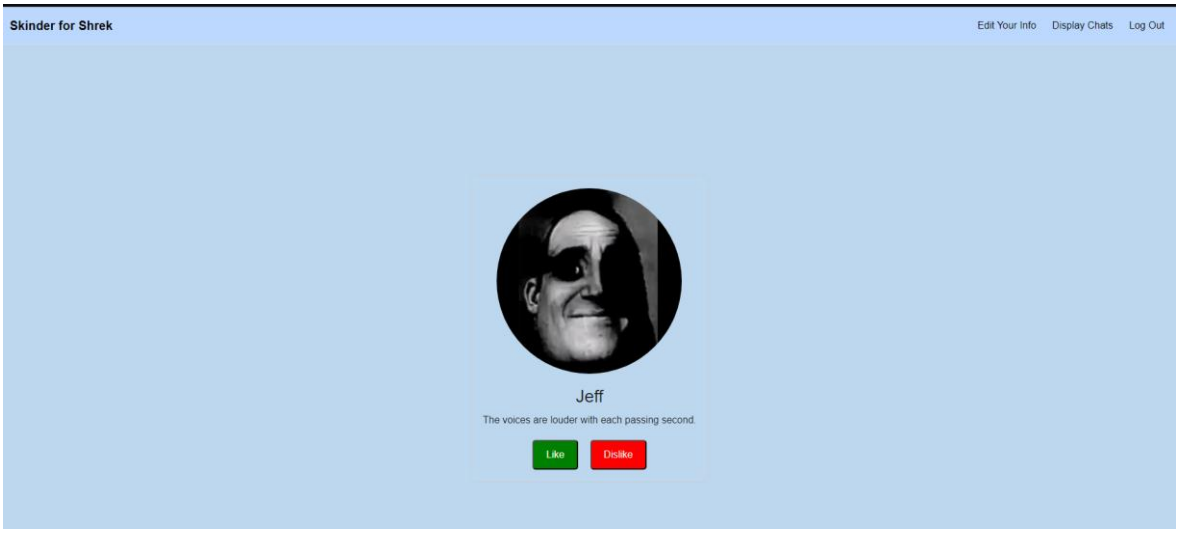

Image 6: Main view after pressing "Like".

If I log out from Shrek's profile and log in as Josh and press like on Shrek when he is recommended to me, I get notified that we got a match:

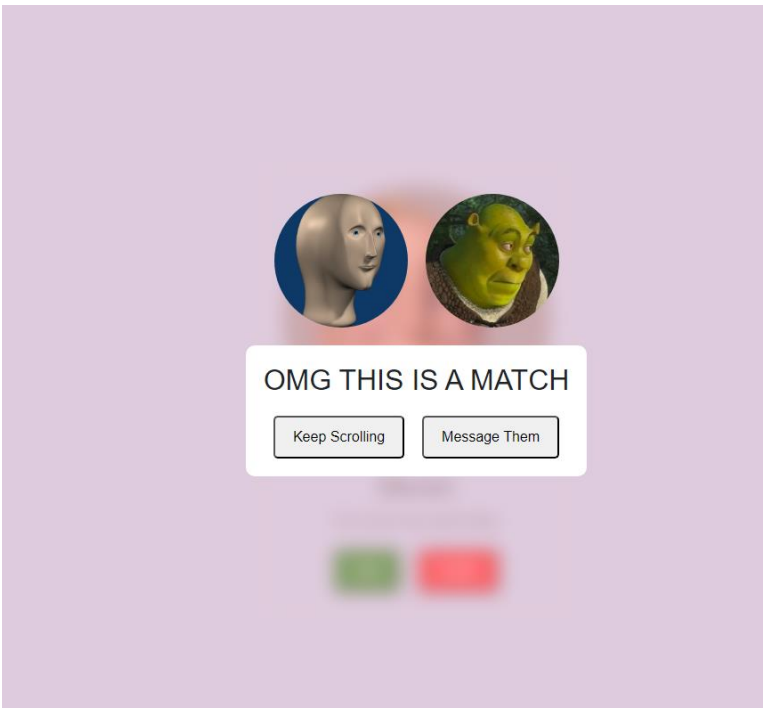

Image 7: Match.

I can either keep scrolling by pressing the "Keep Scrolling" button, or then send a message to my match by clicking "Message Them" button which redirects to route /chats and I can see my conversations. In the below screen shot (image 8) I already have a pre-existing conversation with Jeff, but we are focusing on Shrek now:

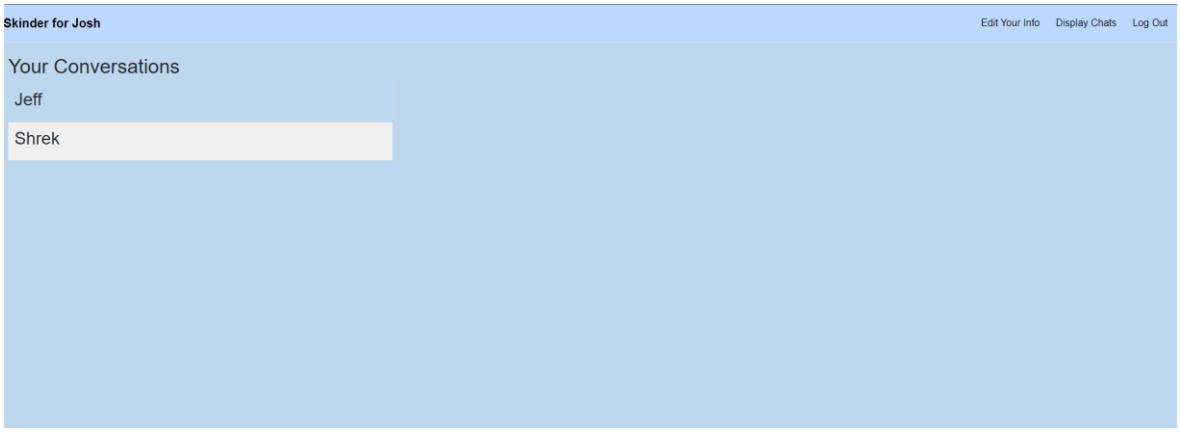

Image 8: Chat view.

Clicking the name opens and closes the chat view between the logged in user (Josh) and their match. The /chat route can be reached by clicking the header link “display chats” as well.

If the conversation is empty, text “*crickets*” is shown:

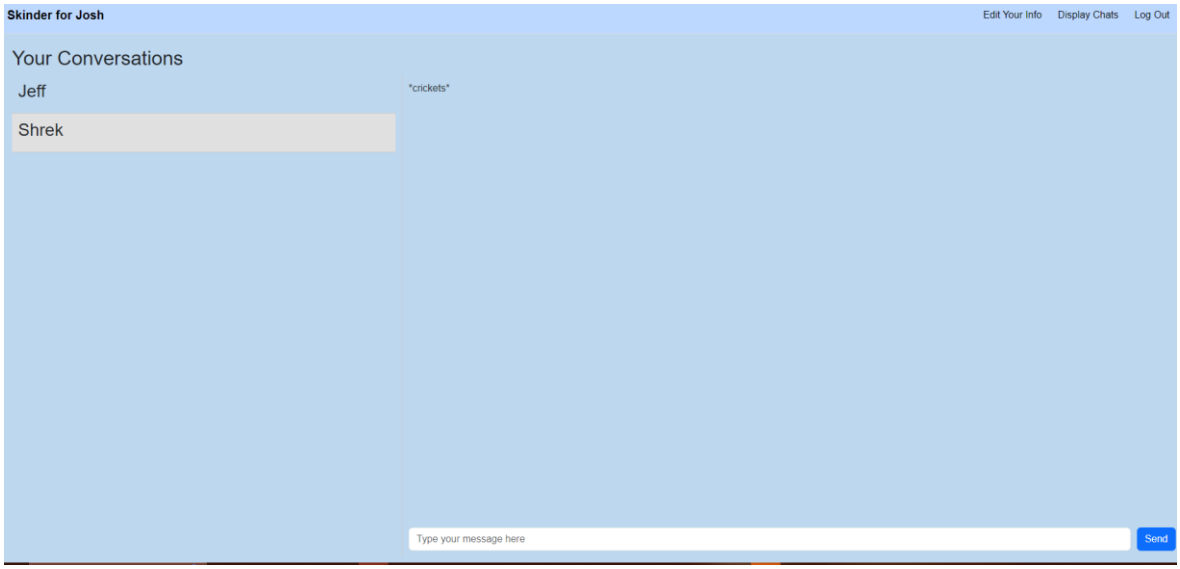

Image 9: Empty conversation.

Sent messages appear in the chat view instantly after sending them and the styling of the messages makes it easier to see whose message what is. Below is Josh’s view of sending messages to Shrek:

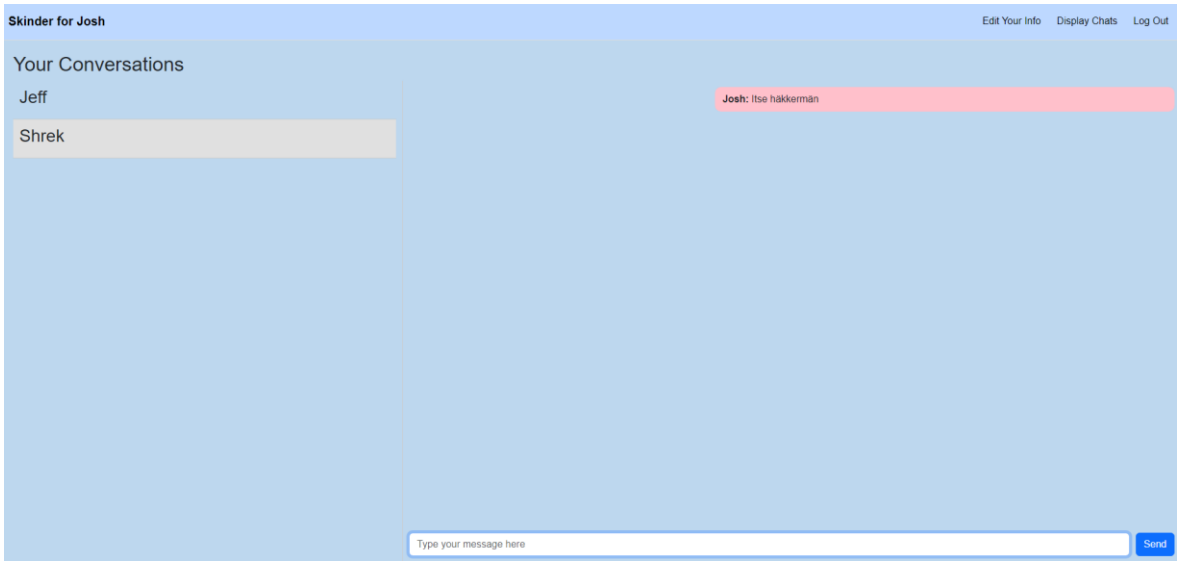

Image 10: Sending a message.

And when logging out from Josh's account and into Shrek's account, the message is visible to him as well:

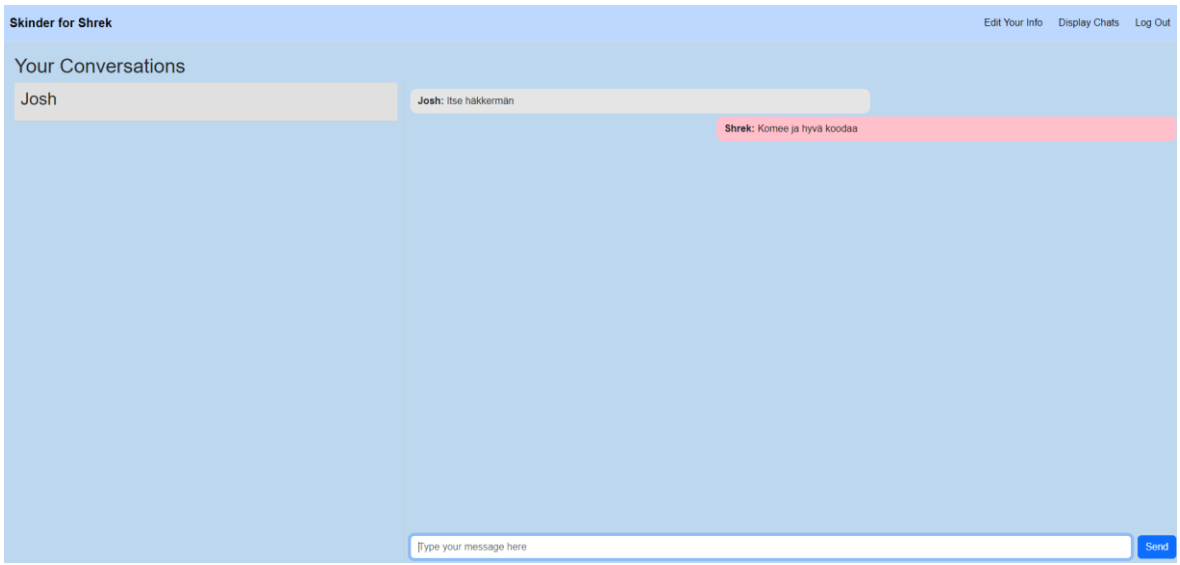

Image 11: Receiving a message.

Clicking the header link "Edit Your Info" redirects the user to /editinfo where the current profile is displayed as well as fields for updating user information and image:

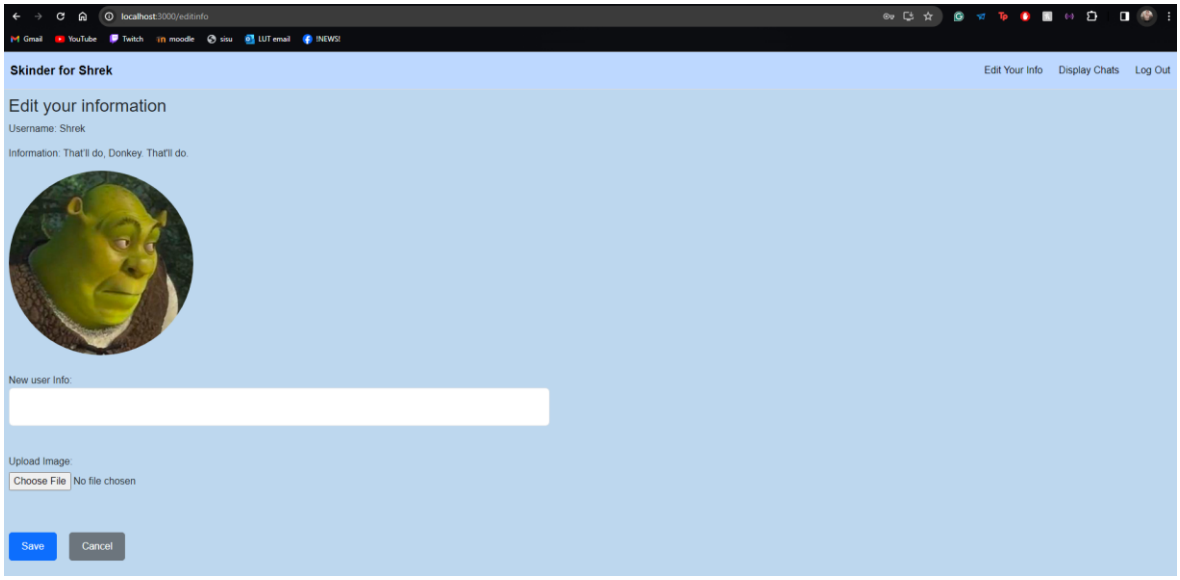

Image 12: View for displaying profile and editing information.

Before saving the changes, the selected image is displayed in a block:

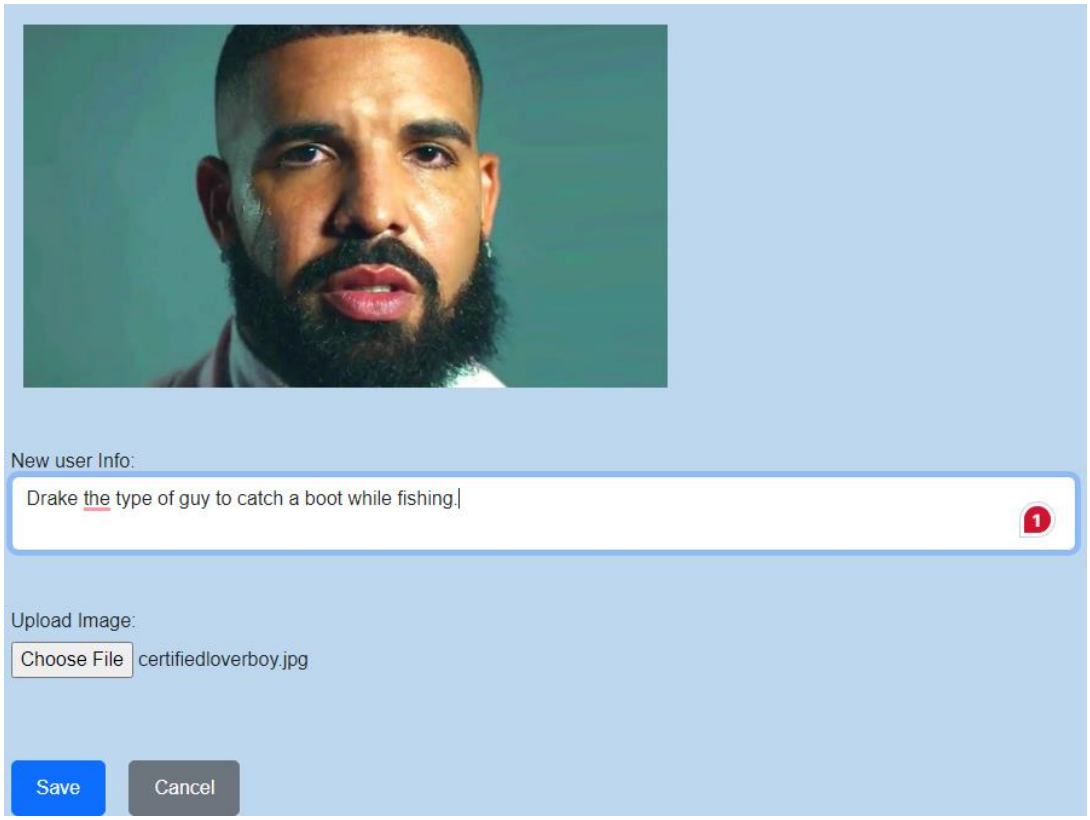

Image 13: Selected changes.

Updated user information and image are set and refreshed when the “Save” button is pressed and “Cancel” button naturally interrupts the changes:

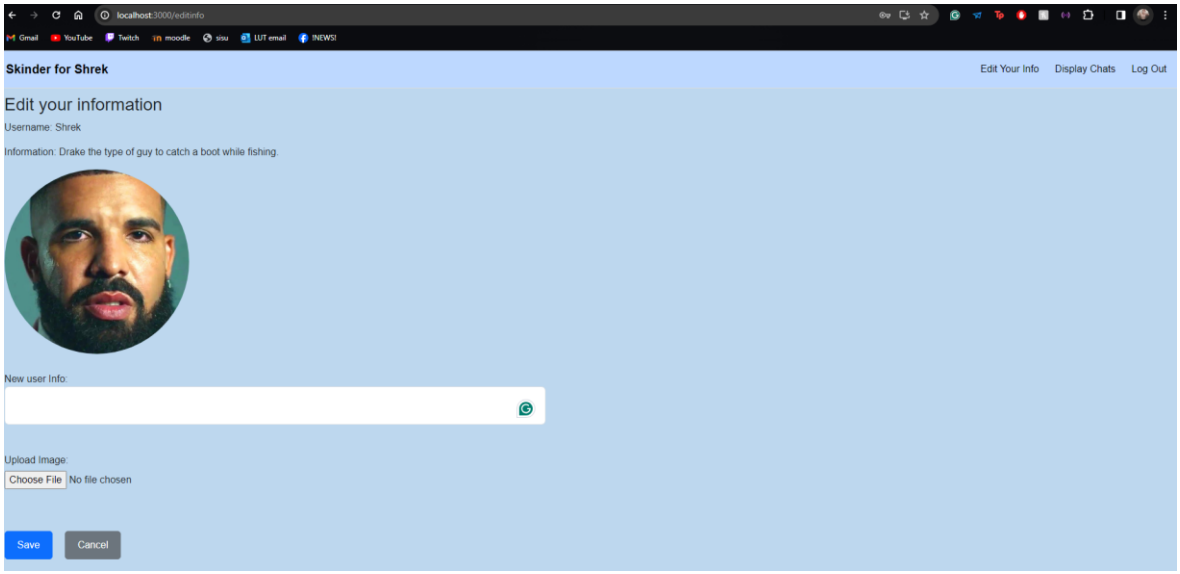

Image 14: Edit info page after pressing “Save” button.

If the user tries to connect to a route that does not exist, an error page is shown:

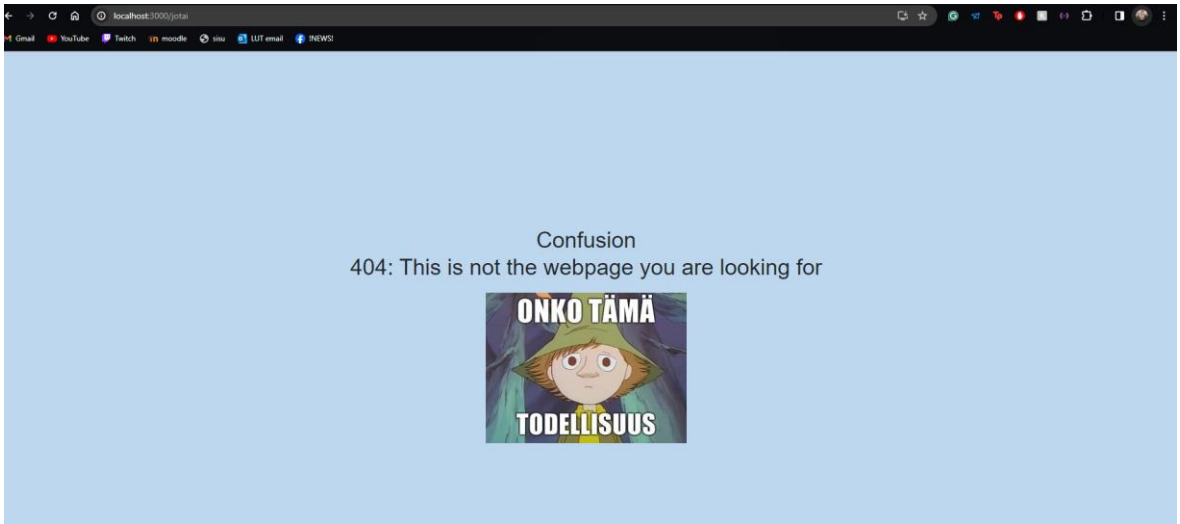

Image 15: connecting to non-existing route.

The front-end code is located in client/src and the routing logic is in server/routes/index.js. Uploaded images are stored in the server to server/routes/uploads and user information, chatlogs, and liked users are saved in databases with corresponding names (defined in in server/routes/index.js):

```
// MongoDB schema for users
const usersSchema = new mongoose.Schema({
  username: { type: String, unique: true },
  password: { type: String },
  profileImage: { type: String },
  userInfo: { type: String }
});
// MongoDB schema for user likes (matches)
const likeSchema = new mongoose.Schema({
  user: { type: mongoose.Schema.Types.ObjectId, ref: 'User' },
  likedUser: { type: mongoose.Schema.Types.ObjectId, ref: 'User' }
});
// MongoDB schema for chat logs
const chatlogSchema = new mongoose.Schema({
  participants: [{ type: mongoose.Schema.Types.ObjectId, ref: 'User' }],
  messages: [{
    sender: { type: mongoose.Schema.Types.ObjectId, ref: 'User' },
    content: String,
    timestamp: { type: Date, default: Date.now }
  }]
});

const User = mongoose.model('User', usersSchema);
const Match = mongoose.model('Match', likeSchema);
const Chatlog = mongoose.model('Chatlog', chatlogSchema);
```

Image 16: Databases.

The program uses JWT authentication, and I am decoding the tokens in the front-end before sending the data to backend for further handling (I know it would make more sense to simply decode on the server side, but I am too far to change it now):

```
// Inside MainPage component
import React, { useEffect, useState, useCallback } from 'react';
import Header from './Header';
import { jwtDecode } from "jwt-decode";
import './main.css'; // Import CSS file for styling

function MainPage() {
  const [randomUser, setRandomUser] = useState(null);
  const [showMatchText, setShowMatchText] = useState(false);
  const [matchUsers, setMatchUsers] = useState({});

  // Function to extract username from JWT token
  const usernameFromToken = () => {
    const authToken = localStorage.getItem('authToken');
    if (!authToken) {
      console.error('Authentication token not found');
      return null;
    }
    const decodedToken = jwtDecode(authToken);
    return decodedToken.username;
  };
```

Image 17: JWT token.

The back-end uses express() and Node.js as requested. As for why I have chosen to use MongoDB, express, and JTW tokens is that I find them the simplest way to do what I want since I have used them earlier.

have included the databases of users, conversations, and matches I have created already to my submission (all their passwords are 1234 although it cannot be seen from the database since the passwords are encrypted). Images 18, 19, and 20 display the contents of the databases.

Image 18: users database (MongoDB Compass).

Image 19: matches database (MongoDB Compass).

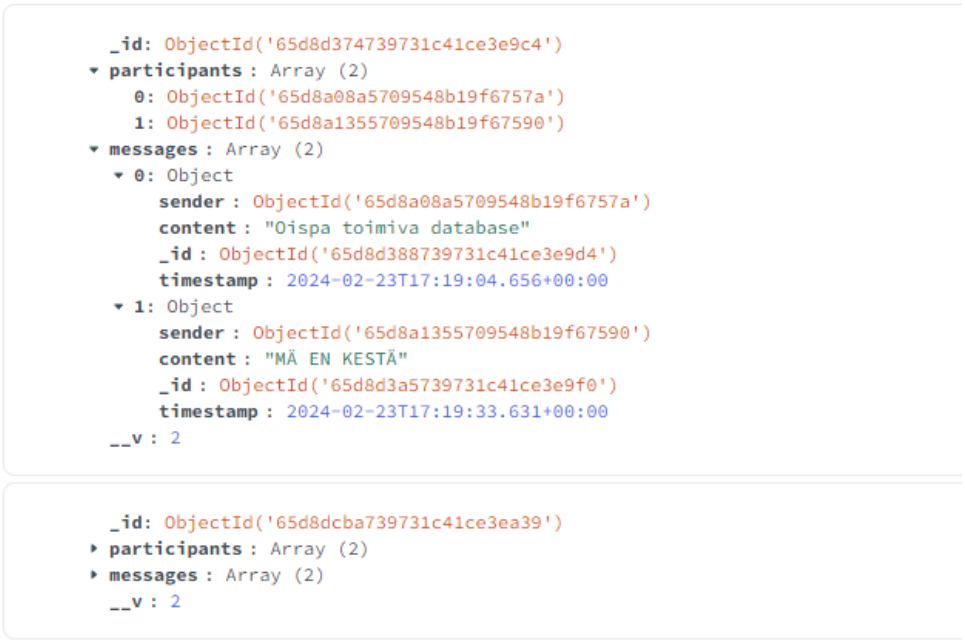

Image 20: chatlogs database (MongoDB Compass).

I have also used @media queries to scale the app for different sized devices. If the device has a small screen, padding is added to the elements and header is shrunk into a hamburger menu using bootstrap:

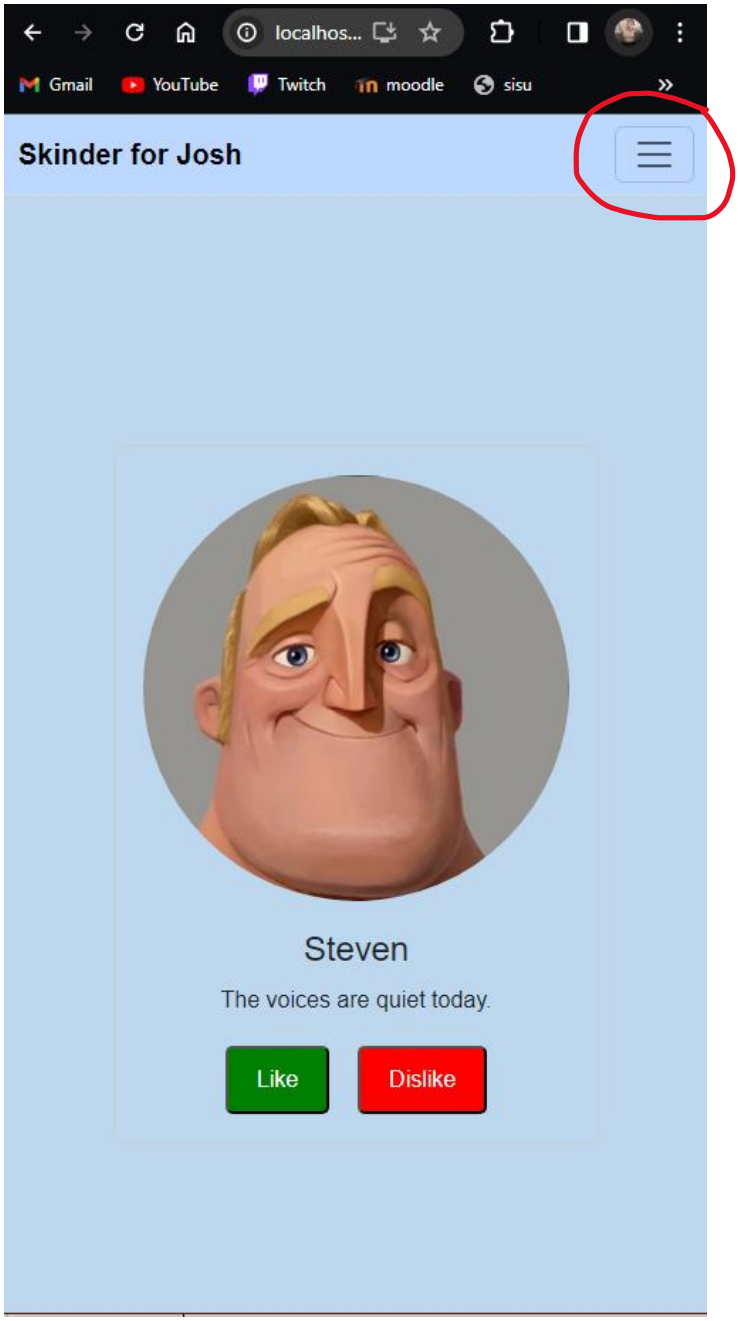

Image 21: Scaling for different sized devices.

Media queries and other CSS is in client/src/main.css where the styling of the entire project is done aside from bootstrap elements.

1.2 Additional features

In the basic features chapter I showcased the additional features of the program prompting the user to send a message to their match immediately when the match happens as well as my suggested password encryption feature.

The client side is using React framework and I am using i18n.js to translate the app in Swedish, English, and Finnish (Image 22).

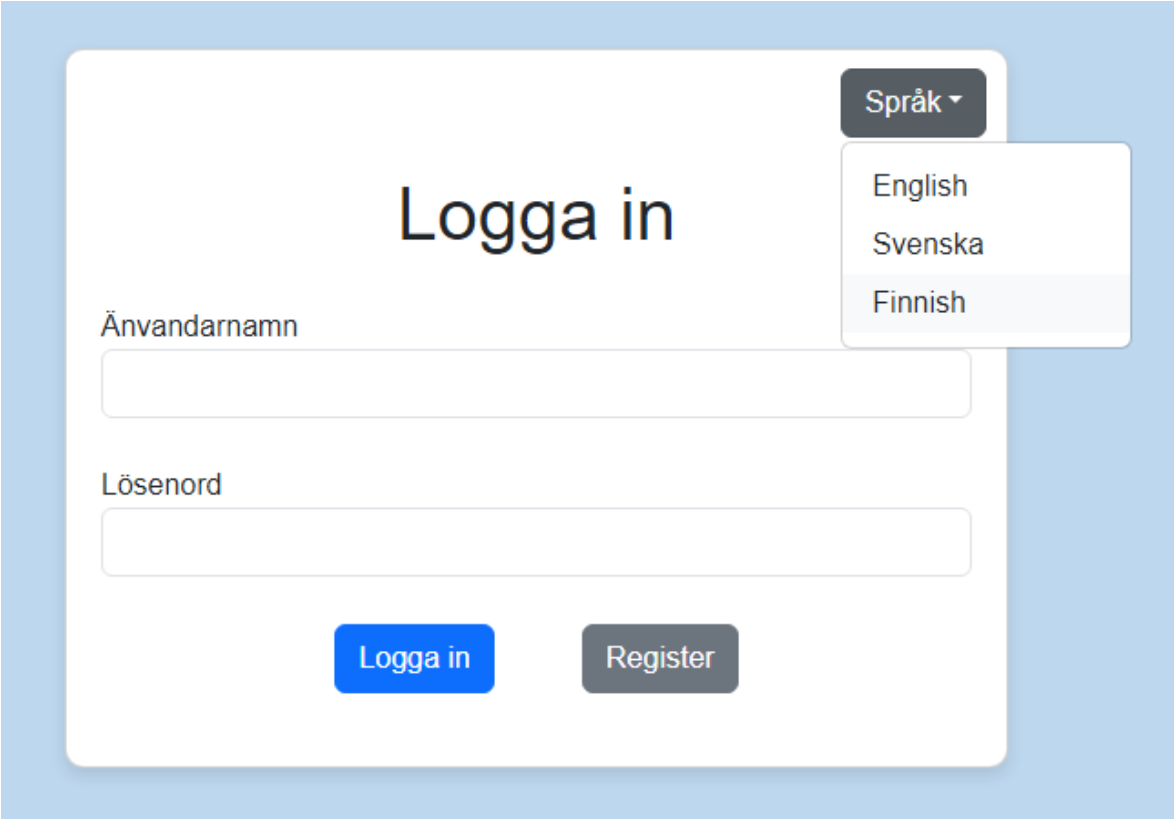

Image 22: Multiple language support.

English is the default language and the language definition JSON files are located in client/src/lang.

If the user who logs in is ”Admin”, they get redirected to /admin route after login. The admin view shows a list of every user profile as well as their information from all databases (matches, user info, and chatlogs). If the admin deletes an user they are prompted for a confirmation and then the user and all their data is deleted. Screenshots below showcase the situation before deleting user ”Josh” (Image 23) and after (Image 24).

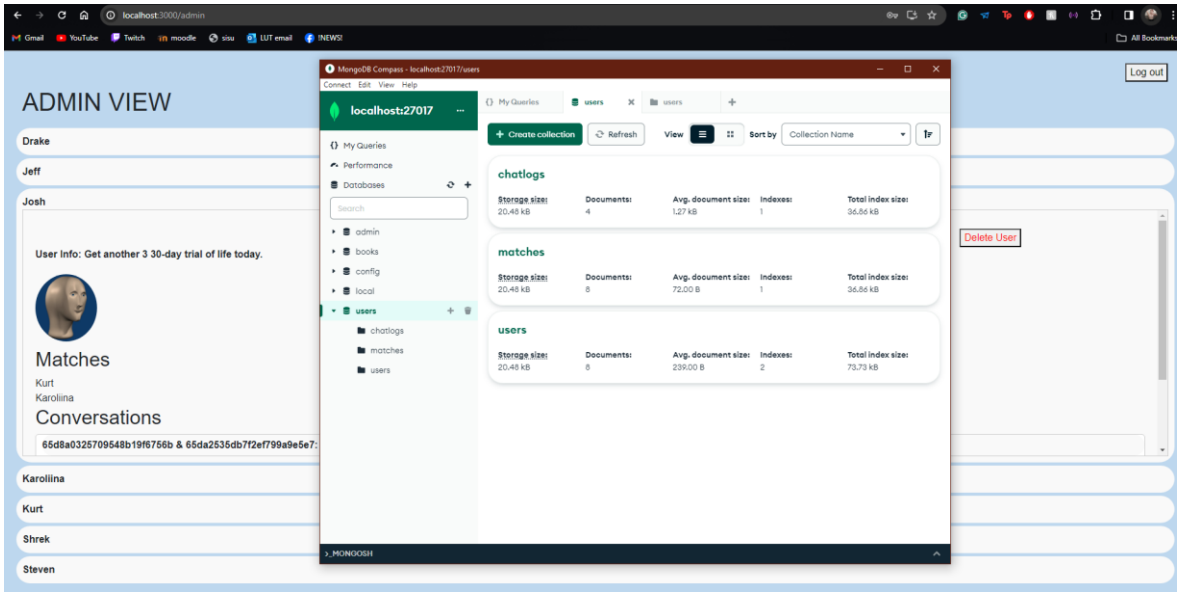

Image 23: Admin view and MongoDB before deleting user ”Josh”.

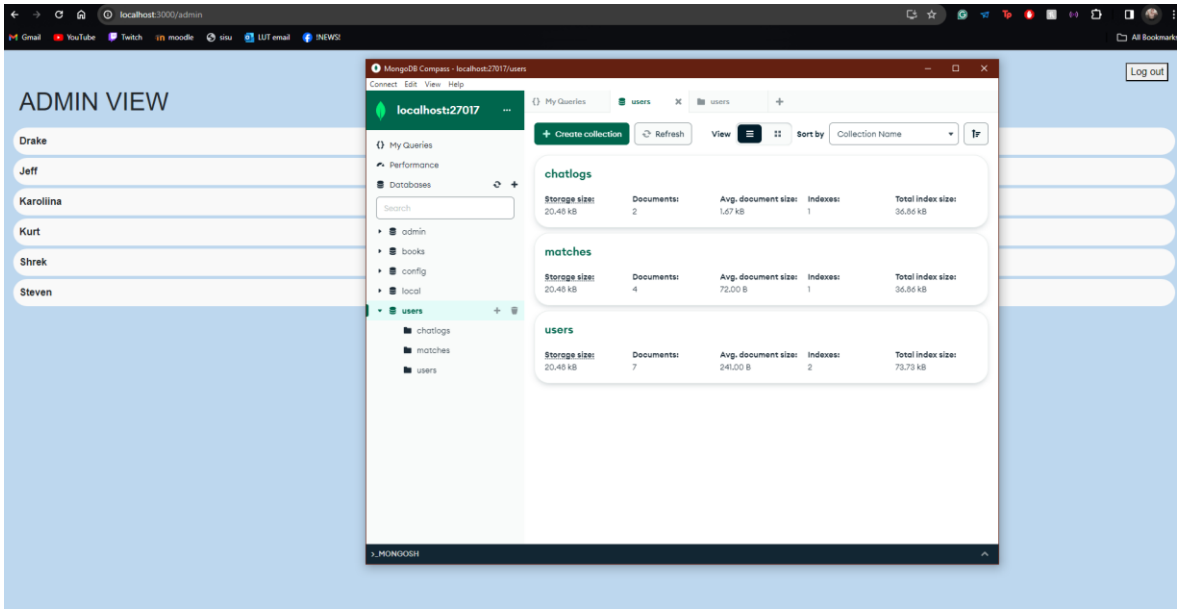

Image 24: Admin view and MongoDB after deleting user ”Josh”.

The admin view is defined in client/src/AdminPage.js and routes are found from server/routes/index.js.

The admin can also edit user information (except for the username because that would be trolling that the users cannot log in anymore because someone has changed their username). By clicking ”Edit” in /admin view (Image 25) the admin is redirected to /admin/edit-user/:username where they can change the user’s image and info text (Image 26).

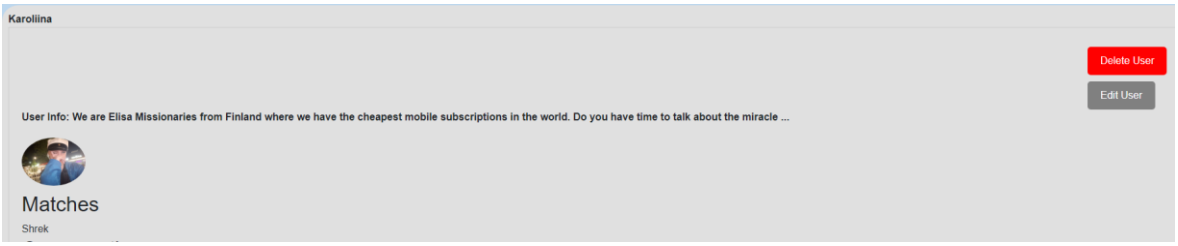

Image 25: Edit User button in /admin view.

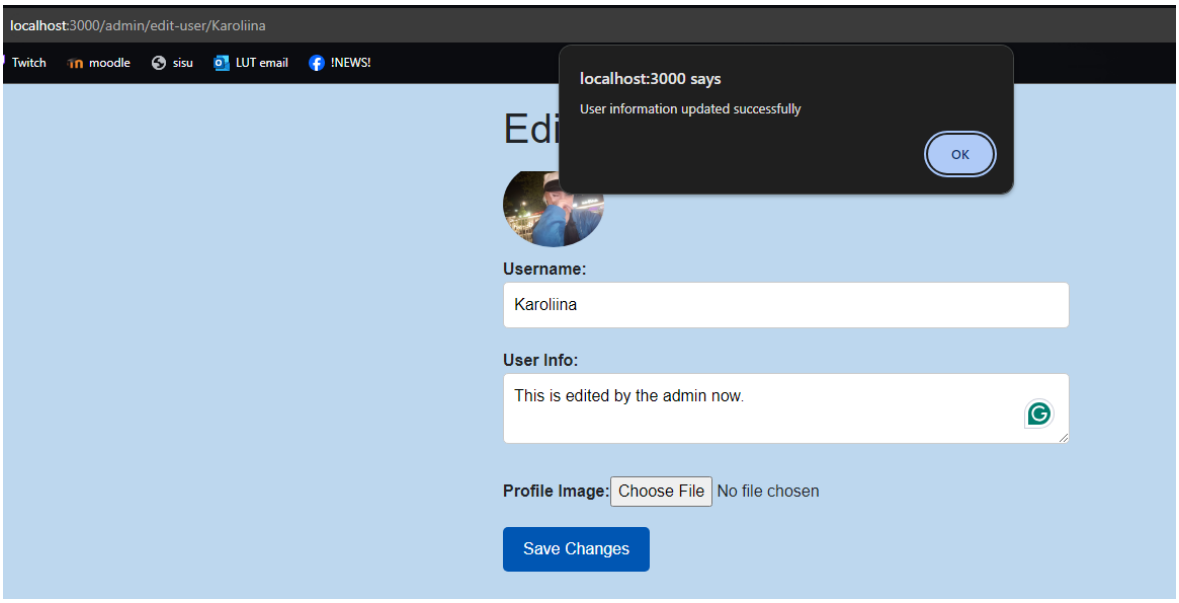

Image 26: /admin/edit-user/:username.

Underneath the save changes button is a return button that takes back to the /admin view where the edited information is visible (Image 27).

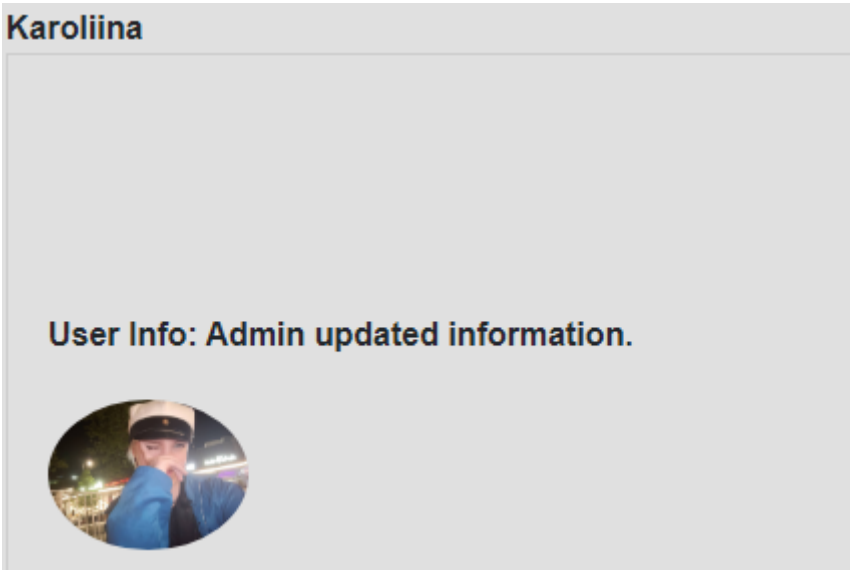

Image 27: Admin can edit user information.

If a non-admin user tries to connect to /admin route, they are redirected to 404 error page by using JWT token authentication.

In the chat view (/chats) the match's image is displayed next to their name (Image 27).

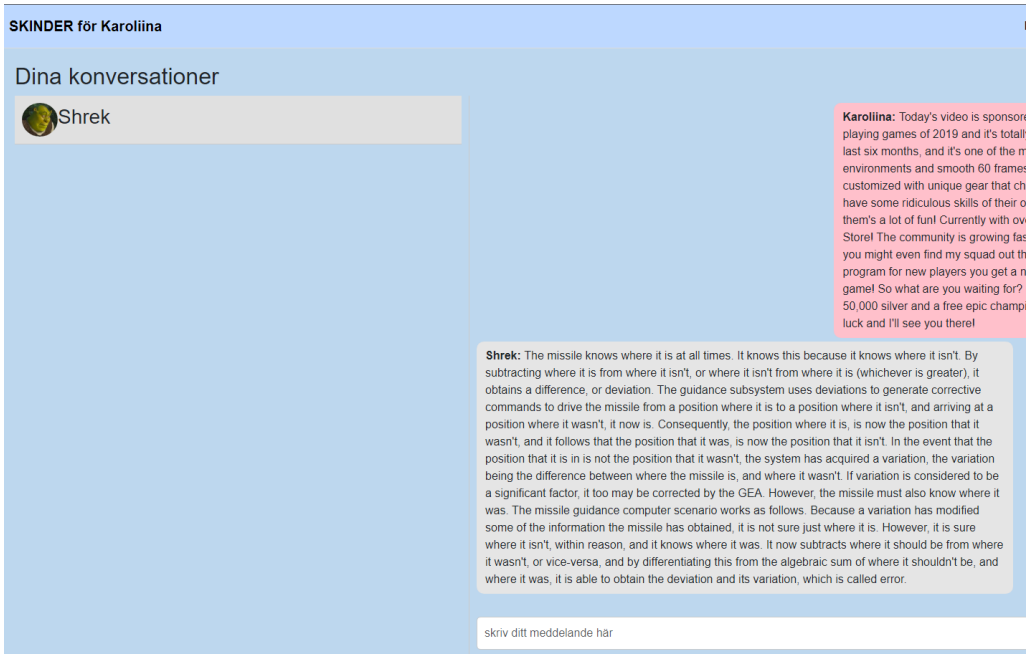

Image 27: profile image displayed in the chat view.

The messages also display the timestamp when they were last edited (sent because in my app there is no editing your messages after sending):

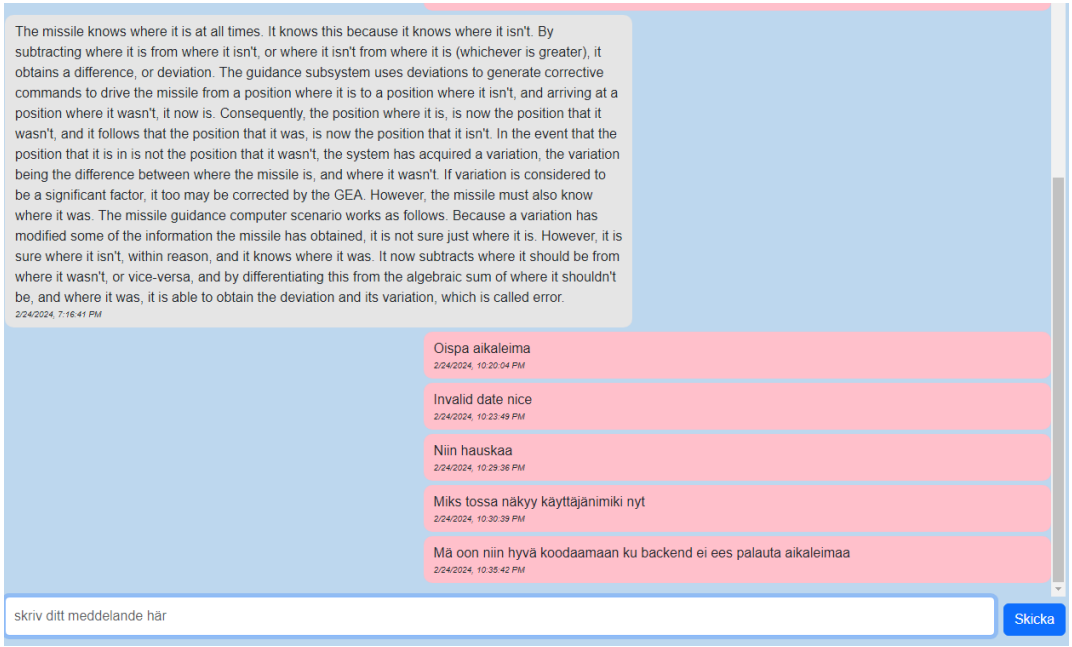

Image 28: message timestamps.

When the suggested user’s name or profile picture is clicked in the main page of the app (the ”swiping” view), the user’s profile is displayed at /profile/:username as shown in Image 29.

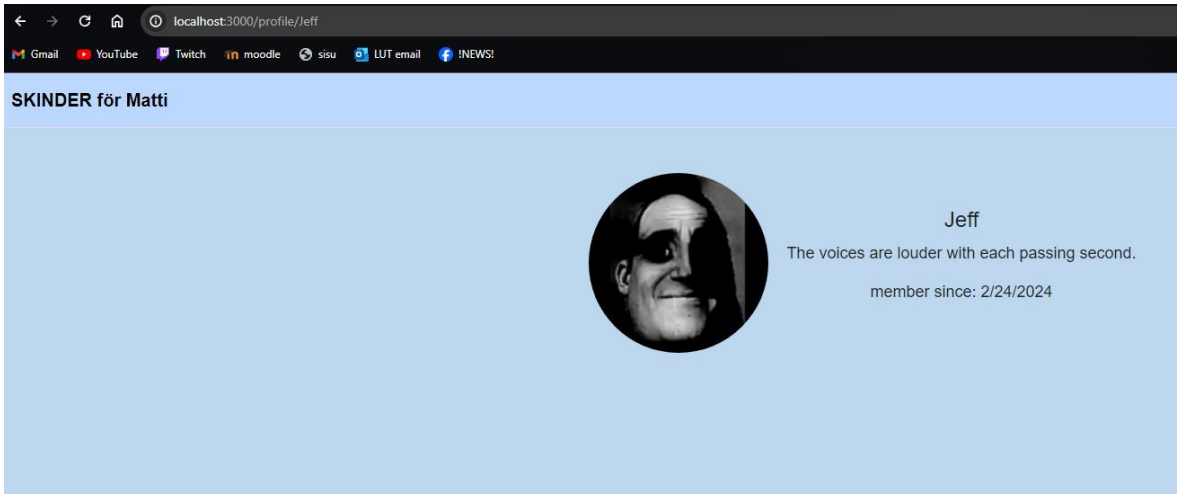

Image 29: user profile and registration date

In the /chats view the user can search the conversations for messages with specific words. The search works in real time and uses highlighting as shown in Image 30.

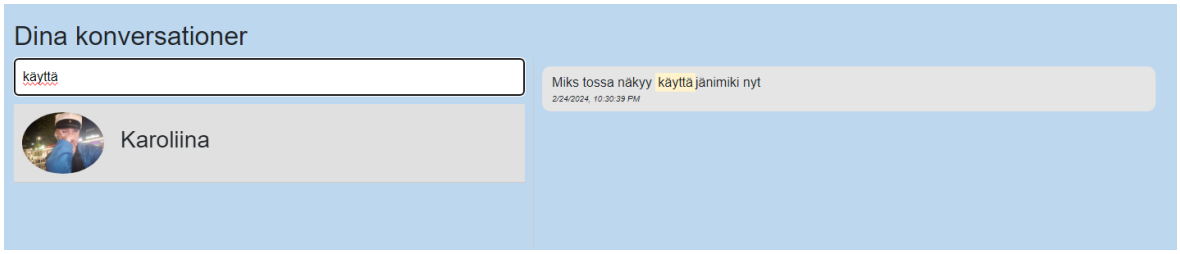

Image 30: messages filtered by keywords.

If the user has more than ten matches, there is a pager as demonstrated in Image 31 and Image 32.

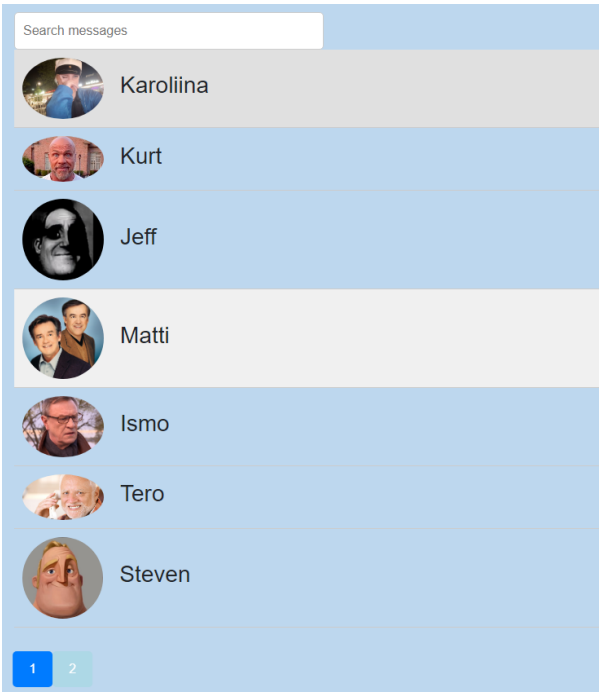

Image 31: ten matches.

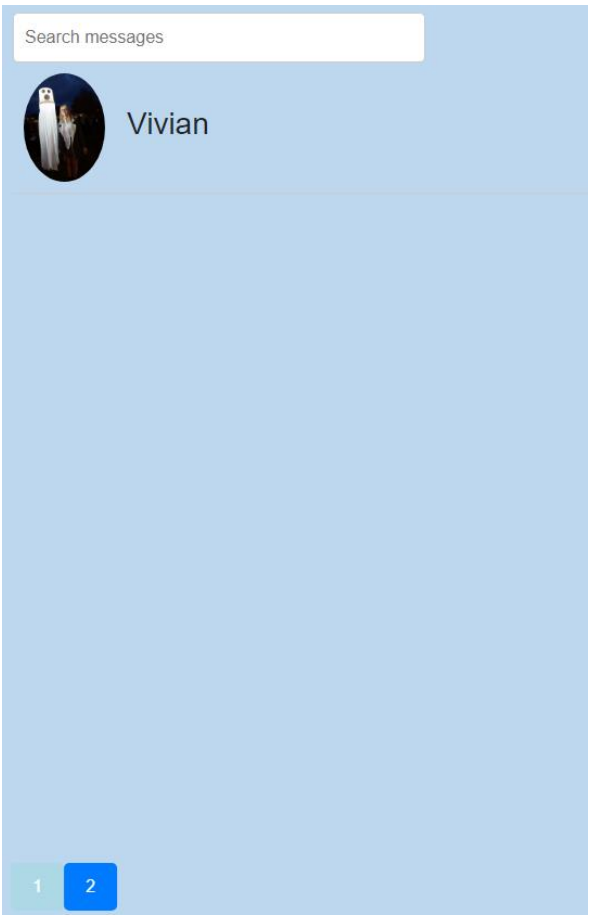

Image 32: 11th match on second page.

In the above demonstration the currently active pager page is highlighted with a darker shade of blue.

If the user has liked every profile in the app they get this view:

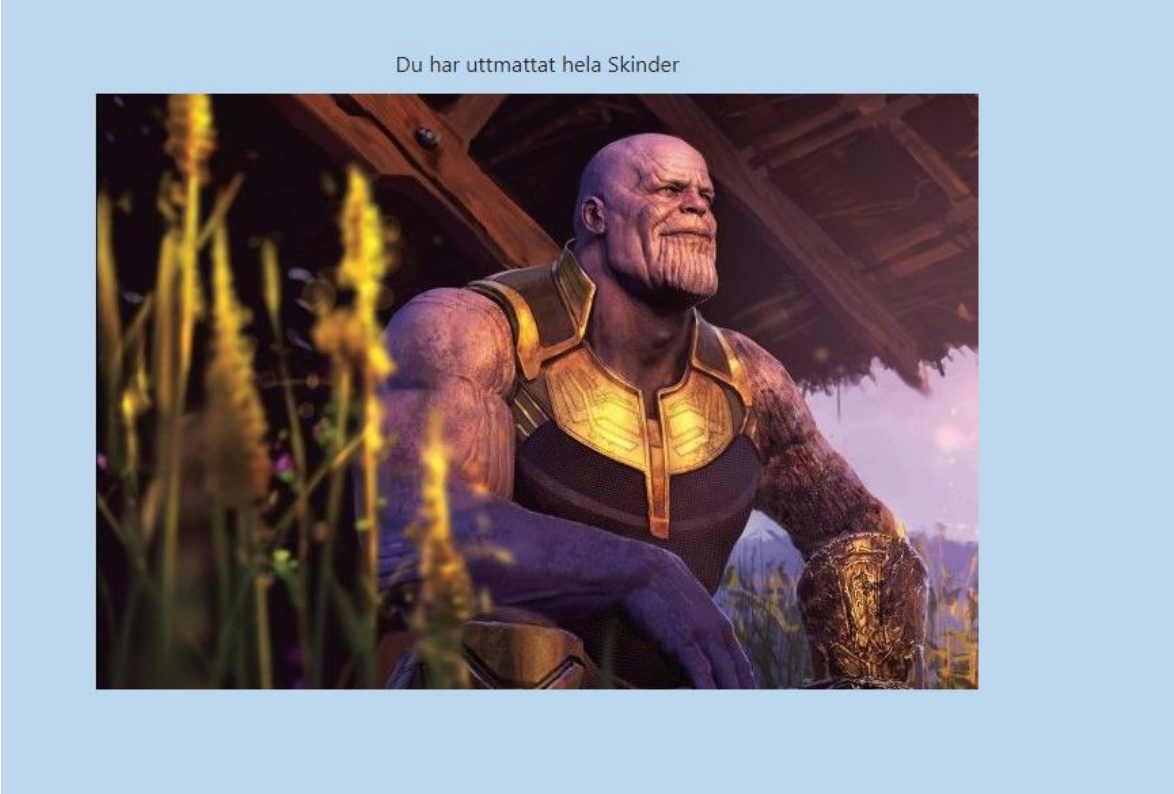

Image 33: No more unliked profiles

Unliked and disliked profiles keep being suggested but liked profiles and matches, as well as the user's own profile and admin profile are not suggested.

1.3 Table of features and attempted points

According to my calculations I have 49 points worth of functionalities in my program but I am requesting points for my password encryption so the total points mount to 51 as specified in the below table:

Feature	**Max points**
Basic features (as stated in the previous chapter) with well written documentation	25
Utilization of a frontside framework, such as **React**, but you can also use Angular, Vue or some other	5
Use of a pager when there is more than 10 chats available	2
Admin account with rights to edit all the users and comments and delete content/users (if a user is removed, all its chats should be removed too)	3
Provide a search that can filter out only those messages that have the searched keyword	3
If match is being found the UI gives option to start chat immediately	2
User profiles can have images which are shown on the main page and in the chat	3
User can click username and see user profile page where name, register date, (user picture) and user bio is listed	2
Last edited timestamp is stored and shown within chat	2
Translation of the whole UI in two or more languages	2
SUGGESTION: Password is properly encrypted before storing into the database because that is good practice when dealing with user information	2
TOTAL	51

2 Program description

The flow chart below (figure 1) describes the program functionality and what routes the program uses. Items with blue tint mean Header links.

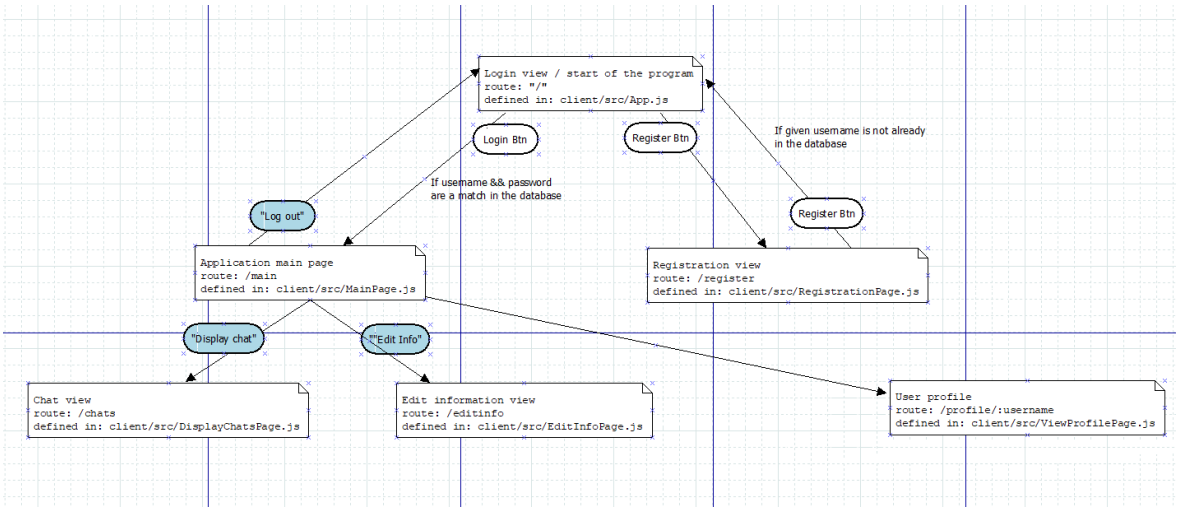

Figure 1: program navigation for user

The administrator uses profile with username “Admin” (password 1234 for testing). The administrator functionality is described in figure 2 below.

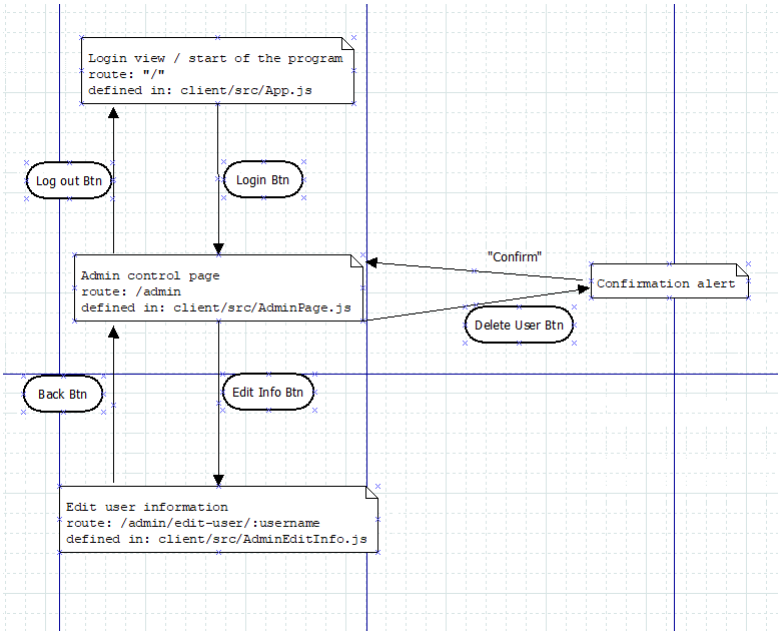

Figure 2: program navigation for administrator.

Only user with JWT authentication token for administrator can access /admin routes.

3 Running the program

The program can be run from project root directory with cd server ; npm start and then opening another terminal from root directory and running cd client ; npm start (or each command separately by opening a terminal in root directory and running cd server and then npm start and opening another terminal in root directory and running cd client and then npm start).

The code can be found from my GitHub at: https://github.com/KaroliinaAaltonen/AWP-Project .

4 Dependencies

This chapter is just a disclaimer about what dependencies I have installed.

The client side of the program uses the following dependencies:

- react-dom: DOM-specific methods for interacting with the browser's DOM (Document Object Model). It's necessary for rendering React components into the DOM.
- bootstrap: CSS framework that provides pre-styled UI components and utilities to help with building responsive web pages.
- React-tinder-card: for the main view Tinder features.
- jsonwebtoken: JWTs are a compact and self-contained way for securely transmitting information as a JSON object.
- react-i18next: UI in multiple languages.
- i18next-browser-languagedetector: Detects browser language.

The server side of the program uses following dependencies:

- Express: Web application framework for Node.js. Set of features and utilities for building web servers and handling HTTP requests and responses.

- jsonwebtoken.
- express-formidable: I could not handle FormData() objects without this.
- fs: file system module because I am dealing with images.
- bcryptjs: encrypting passwords.
- mongoose: I am working with MongoDB.