Computer Science 343
October 2023

Project 2 Report

Victor Bakker(24789682), Matthew Beattie (25113712), Peter-Jack Harrison (25160435), Alex Petika (24966851), Saurabh Roychoudhury(24809055), Jonathan van Druten (25092936)

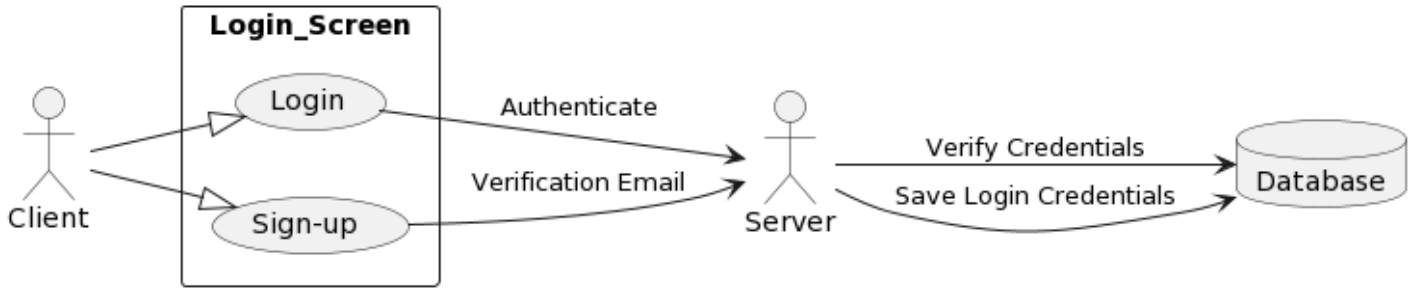
Introduction:

Mark Notes is our innovative solution to a collaborative note-taking platform with live markdown rendering.

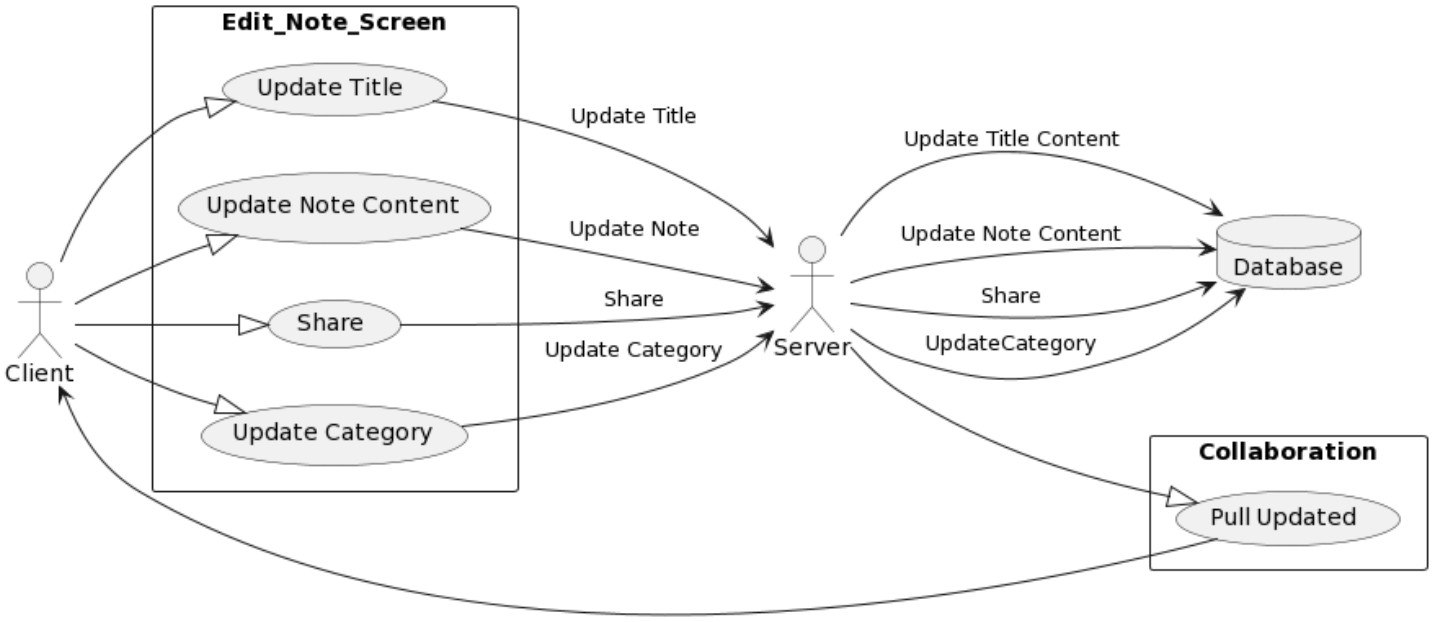
This report presents a comprehensive insight into the architecture in place, and emphasizes the integration between the front-end and back-end. It will also cover the database schema, which underpins the foundation of the application.
The foundation is a PostgreSQL database managed by pgAdmin. The core of this application is a Node.js backend powered by Express, which is connected to a React-Vite frontend framework. In terms of the aesthetics of the application, Tailwind CSS was utilised. A description of the dependencies used and the rationale behind them is provided under Operating Environment.
To offer more insight, Use Case Diagrams are incorporated to better illustrate a User Journey, the API calls involved and the authentication process.
Additionally, included are the team's contributions and an overview of the document used for task tracking, with a more in-depth workflow handled by an Atlassian Trello board.

Use Case Diagrams:

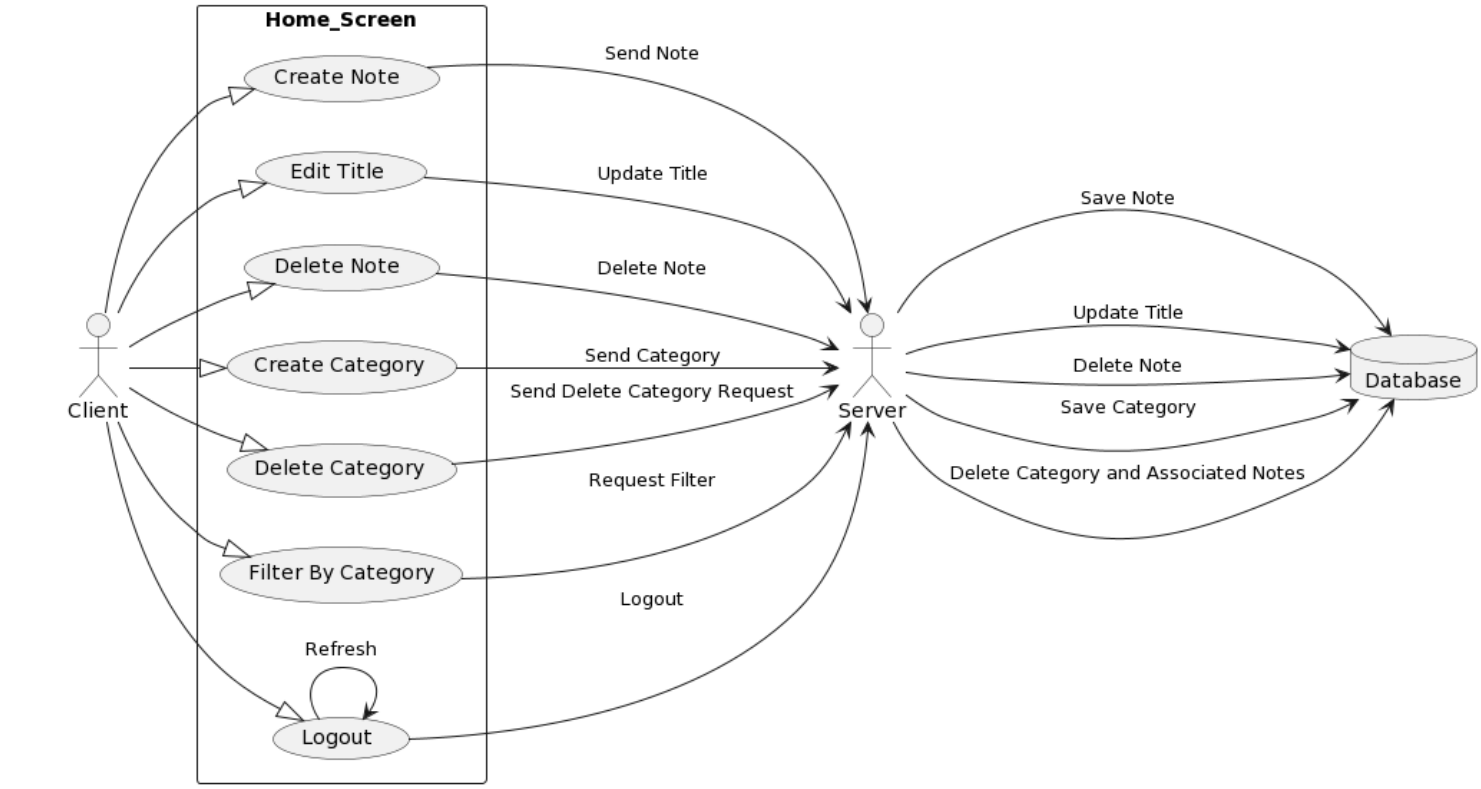
(1) Login Screen:

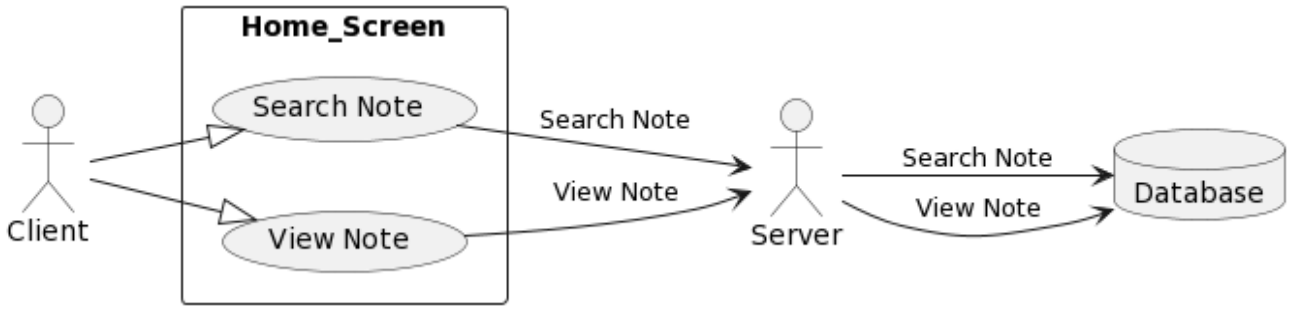


(2) Edit Note:

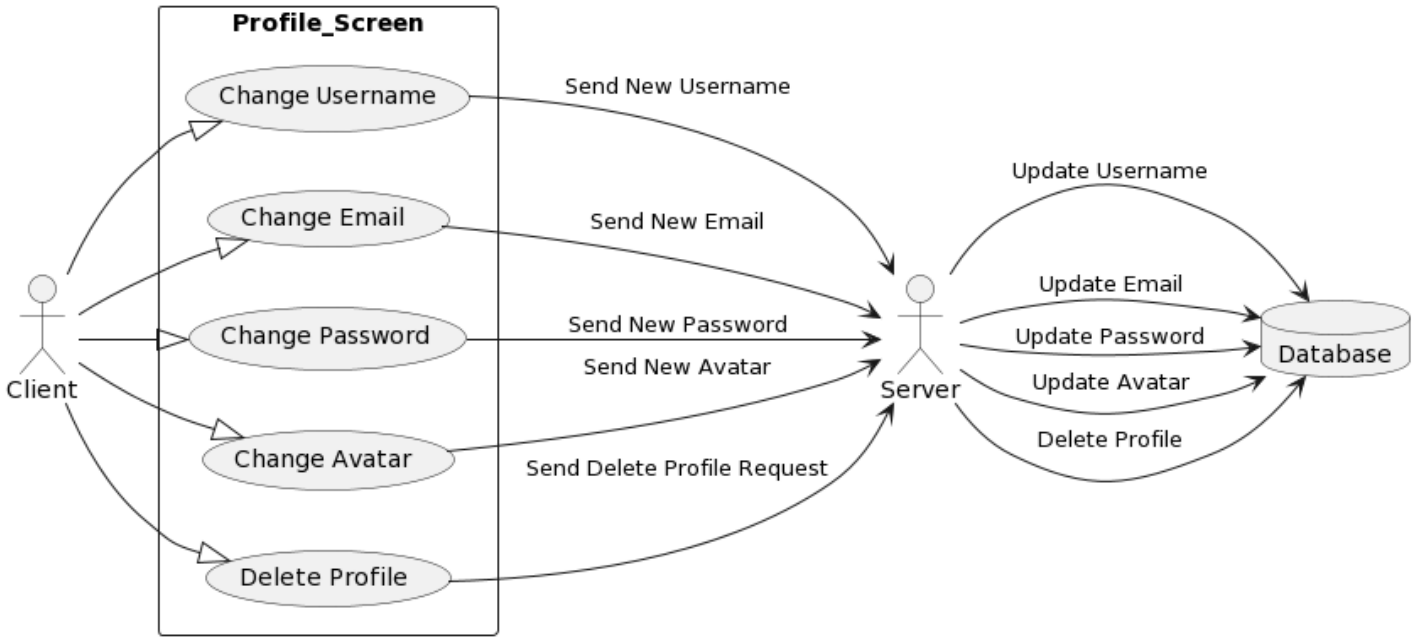


(3) Home:





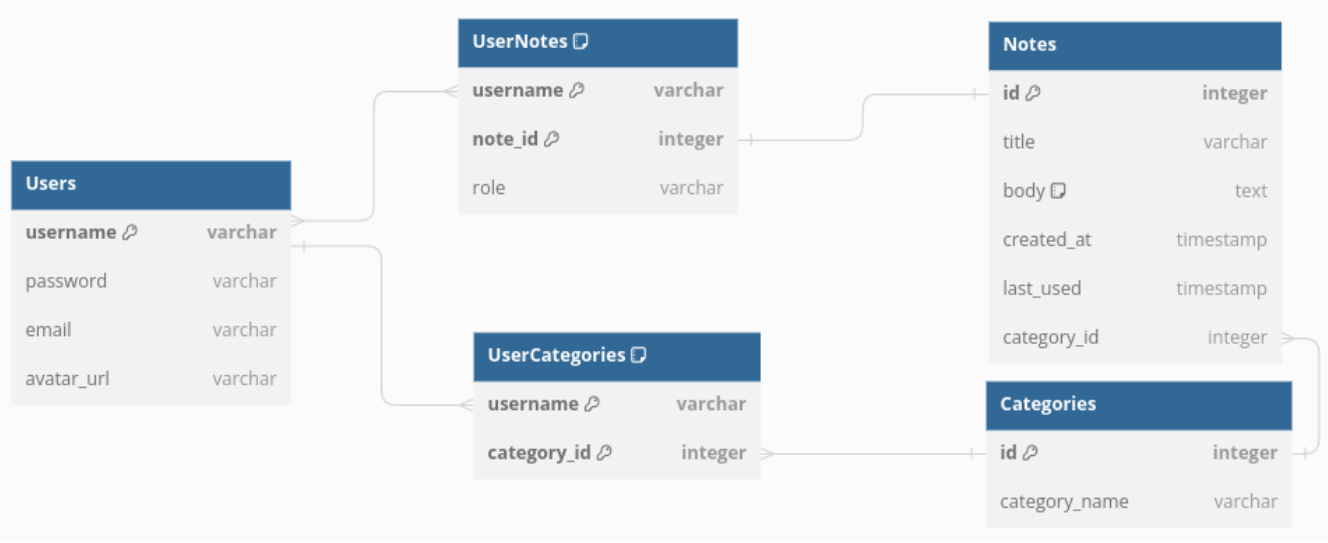
(4) Profile:



Data Modelling and Description:

The database is maintained as a PostgreSQL database. At the highest level the database consists of five tables: Categories, Notes, UserCategories, UserNotes and Users.

Entity Relationship Diagram



At a more granular level the data is organized as follows.

Users and Categories:
‘Users’ is in conjunction with **‘Categories’** via the association table **‘UserCategories’**. This relationship allows a user to be associated with one or more categories.

Users and Notes:
The **‘Users’** table is indirectly related to the **’Notes’** table though the association table **‘UserNotes’**. Containing both *primary keys* as its *super key.* The relationships between **‘Users’** and **‘Notes’** are characterized by the role the user has over the note. This allows users to be either invited to collaborate on a note, or be the user that created the note.

Categories and Notes:
‘Categories’ has a one-to-many relationship with the **‘Notes’** table. Established via the *foreign key* ‘category_id’ in the **‘Notes’** table. The relationship allows notes to be categorized under specific categories.

In conclusion the database is in 3rd normal form because:
(1NF) All the attributes are atomic. **(2NF)** All non-key attributes are indeed dependent on the primary key. **(3NF)** No transitive dependencies exist.

Operating Environment:

For the frontend React was used with the Vite buildtool to build our project.

List of all frontend packages and what they do:

- **marked:** This package allows the application to, in real time, display text input from a user as Markdown formatted text.
- **react-router-dom:** This package allows the website to have multiple pages/routes that have different endpoint urls.
- **socket.io-client:** This package serves as the client side websocket interface that can communicate with the websocket system on the backend
- **minidenticons:** One of the requirements of the system was to not store images on the database. We did not want to have to store lots of images on our server and we wanted the user to have an abundance of choice. This package can turn text input into a small semi-unique avatar. This means that the system can just store a string/text on the database and then the frontend can take that text/string and display its corresponding avatar for the user.
- **quill and react-quill:** Quill is a rich text editor package that gives you more functionality than a regular html <textarea/>. It allows for [Tab] presses as an example. One of the more relevant features is that it allows the system to send 'deltas' (individual changes) to websocket as opposed to the entire text. This resulted in a more efficient websocket that is easier to configure for real-time collaboration.
- **jwt-decode:** Lets the system decode the information stored in json web tokens that are sent from the backend to the frontend.
- **tailwindcss:** A powerful css framework for styling our site.

The backend has been created in Nodejs. This means that the server has been written in javascript.

List of all backend packages and what they do:

- **express:** Framework for creating REST APIs.
- **socket.io:** Package used for creating server-side websocket code.
- **http:** In order to have the websocket and the express server listen on the same port the http package was used.
- **nodemailer:** Allows the server to send emails to users.
- **pg:** Package which allows communication between the server and the postgresql database.
- **jsonwebtoken:** Used for creating and signing json web tokens.
- **bcryptjs:** Used for hashing and salting passwords before they are stored on the database.
- **dotenv:** Allows access to .env file which stores secret keys and other data needed/used by the server.

How components were hosted:

- Both the front-end and back-end components were deployed on Heroku's Cloud Application Platform.
- These components were hosted as separate entities to ensure modularity and scalability.
- To guide Heroku on executing the applications, Procfiles were placed in the root directory of each hosted application.

- In order to deploy the code, the code must be merged from master into deployment branch and then manually redeploy to the heroku branch and then redeploy with heroku

Client Design:

Structure

The client-side is composed of several main pages (HomPage.js, ProfilePage.js, EditPage.js, LoginPage.js etc stored in a pages folder) that each import lower level components (stored in a components folder) such as the modals for creating and deleting notes and categories, as well as icons from the assets folder, in order to modularize the code. Connections to the server-side are handled by methods (such as handleFetchNotes) in all the main pages, which are sent through child components to be triggered by specific user interactions, like clicking the Create Note button.

Routing

The client-side routing is managed by the 'react-router-dom' library. This library assists in the creation of Single Page Applications in React, in which the navigation between different parts of the application does not cause a full reload, but only manages a few changes in the client.
The 'useNavigate' hook provides functionality in allowing the changing of routes based on the user's actions.

Markdown

The client uses a Markdown component (HomepageMarkdownComponent and MarkdownComponent) to handle the display of the Markdown content. This integrates with the rendering library 'Marked'. The component processes and converts the raw string text into HTML formatted Markdown text, which is displayed to users and enables rich-text formatting.

Fetching note and category data

Notes and categories are fetched from API endpoints. The client makes HTTP requests and once the data is fetched, it is set into the state, using React's 'useState'. The list of notes and their corresponding details are propagated down to child components. Data is displayed up-to-date as handleRefresh methods ensure that any changes to a note or category will result in an update of the notes and categories lists.

Searching for a note by its title

This function is handled client-side in HomePage.js only, by filtering the list of notes based on the searched title. This function searches for the typed substring in all note titles, and returns only notes within the selected category filter that match the substring.

User Avatars

User avatars are generated from a random seed, but users can also select an avatar from a predefined group, or generate a new one randomly. Only the seed is stored in the database.

API Design:

The backend was constructed in Nodejs using the ‘Express’ framework for the REST api endpoints and ‘Socket.io’ for the websocket portion of the server. For authentication json web tokens were used to give users authorized access to the Express endpoints and the websocket. In addition there is an email system which can send links to the user to reset passwords or verify email addresses.

File Structure:

There are 3 main files in the api project. There is a server.js file which contains all the express endpoints. This file is run to run the server. There is a database.js file which connects to the database and contains all of the functions which perform access to the database. Lastly there is a file called email.js. This file creates the nodemailer which can send emails to users. All of the functions which send emails to users are in this file.

There is also a folder called middleware which contains two authentication functions. One of the authentication functions is a middleware authentication fo the express endpoints and the other is a middleware authentication function for the websocket.

API endpoints:

The API endpoints are rather extensive and well documented in server.js. And so only the high level description of the API will be written here, for details on use please see the documentation.

This RESTFUL API allows for all the CRUD operations needed for this website to function. These include the necessary POST, PUT, GET and DELETE function for users to

- Create, share, rename, edit and categorize their notes
- Create their account, update their email, password, avatar url and their username
- Allow for real-time collaboration using websockets

Each of the endpoints are also secured using JWT authentication tokens, this is reason for the presence of the authenticate Token function in all of the endpoint definitions in server.js.

Some general practices made throughout the API.

- Username of user making request always placed in query parameter user
- All information required to obtain desired resource (such as note id or category id) were passed as query parameters
- All types are done on the server side to ensure data integrity
- All data to be sent to the server is to be placed in the body of the request
- All passwords and login information is passed through the body
- If any error does occur a json will be returned where the error will be placed in the “Error” property

File ownership and how it is implemented on server:

The file ownership for MarkedNotes is implemented such that a user can have one of three roles

- owner - this user can edit the title and category of the note
- collaborator - this user may only make changes to the body property of the note
- Viewer - this user may only view the note and has no writing rights

Changing ownership is a simple process. Only the previous owner may make another user an owner and once the new owner is set the old one is made a collaborator by default. This can also be changed by the new owner if they so wish.

What happens if a user deletes the note?
If the owner of the note deletes the note, then it is deleted for everyone and no one will be able to access the note anymore. However the same is not true for collaborators and viewers. These users only delete the note for themselves and not for any of the other viewers or collaborators of the note (including the owner of course)

What happens if the owner of the note deletes their account?
If the owner of the note deletes their account, then all the notes they owned and shared will no longer be available to anyone, however all notes shared with them, will remain accessible to all other viewers and collaborators of that shared note

Data return Format and passing format
All data is returned from the server as a json object. And all data that is to be placed in the body of a request is expected in json format as well.

Development Endpoints:
Extra endpoints were specifically included to ease development. When developing an API it is very useful to have extra endpoints which allow you to, for example, see all instances in a table, as you want to be able to ensure that you are manipulating the data correctly. Further, we created a Postman collection so that the front-end developers can easily refer to any API call.

Authentication:
To authenticate users on all endpoints a middleware was used. All requests to the server that need authentication ought to have a json web token (jwt) in the requests cookies. The middleware function checks the jwt to see if it is valid based on a secret key stored on the server environment. If the token is invalid then the middleware prevents access to the api endpoints.

A user may access a valid jwt through the login endpoint. If they wish to use ‘remember me’ functionality then they will receive a cookie with no expiration date, otherwise their token will expire within 1 hour.

Web-Socket:
Splitting users into groups:

For the websocket implementation ensuring that users can edit different notes on different socket groups is essential. To this different socket groups were split based on the note id a user is editing. Since each note id is unique this means that we can ensure that every note will be edited through its own socket group.

On a user's first connection they will provide the note id of the note they wish to edit. The socket will then place the user in the appropriate group, fetch the relevant note from the database, and send it through to the user.

How changes to notes are shared:
Once a change is made it is sent to the socket. The socket will then emit the change to all the users within the relevant socket group. The client side will then check if the source of the socket emission comes from a different user to implement the change. This ensures that a user's changes do not happen twice.

Saving notes:
Notes are saved directly through the socket. Every 1.5 the client sends the full note to the socket. The socket will then store the note on the database.

Email System

An email verification and a reset password system was created using the nodemailer package. Our own email address marknotes.system@gmail.com was created for this.

Email verification:
The steps for the email verification are as follows. The user must register their account with a unique username, unique email address, and a password. The server receives the registration data a valides that the unique fields are in fact unique. Then an email is sent to the unique email address with an activation link. The activation link contains a jwt. The activation link will take the user to a page which will send a request to the server containing the jwt stored in the activation link. When the server has validated the jwt then the user will be added to the database. The jwt will expire in 1hr if the user does not click the activation link.

Reset Password:
If a user has forgotten their password they must send their email to the server. The server will then verify that this email exists in the database. An email will be sent with a ‘reset password link’ that contains a jwt. Once a user clicks the ‘reset password link’ they will be directed to a page where they can enter their password. They must do this within 10 minutes or else the jwt will expire. Once the user has entered their new password the jwt will be sent to the server along with the password. If the jwt is valid then the new password will be hashed, salted, and stored on the database.

Summary of Contributions:

Victor Bakker:

Front-end:

- Profile page

Backend

- Database schema
- Homepage delete api

Matthew Beattie:

Back-end:

- Delete operation contribution

Front-end:

- Homepage

Peter-Jack Harrison:

Back-end:

- PUT and DELETE endpoints
- Various Database.js functions

Front-end:

- Minor changes to home page

Alex Petika:

Back-end:

- GET and CREATES endpoints
- Various database.js functions
- Hosting

Front-end:

- Public-facing Hosting

Saurabh Roychoudhury:

Front-end:

- Login and Register pages
- Edit Note page
- Websocket implementation

Backend:

- JWT implementation and authentication
- Login and Register routes
- Verification Emails and password reset
- Websocket implementation

Jonathan van Druten:

Front-end:

- UI design: homepage and modal components
- Connecting API to client in homepage