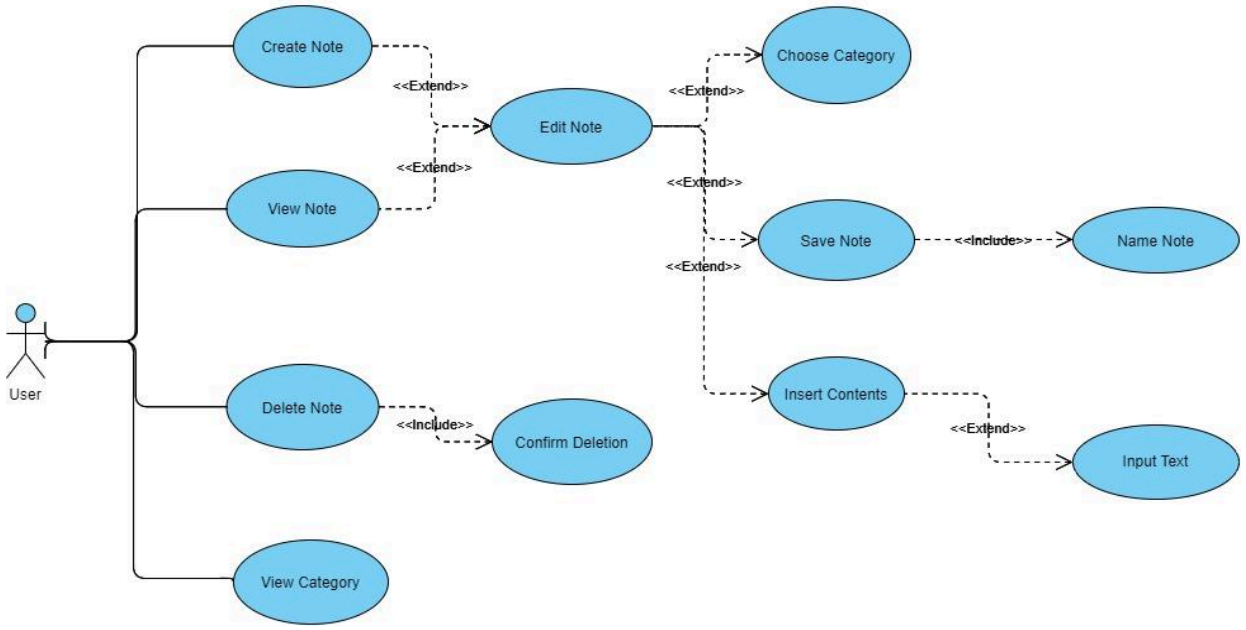
COLLABORATIVE REAL-TIME NOTE-TAKING WEB APPLICATION

Nicole Nkala (25022318@sun.ac.za)
Zirna Lala (25863304@sun.ac.za)
Katlego Ngobeni (25883674@sun.ac.za)
Tasheel Govender(25002112@sun.ac.za)
Keuran Kisten (23251646@sun.ac.za)
Robert Shone (25132687@sun.ac.za)

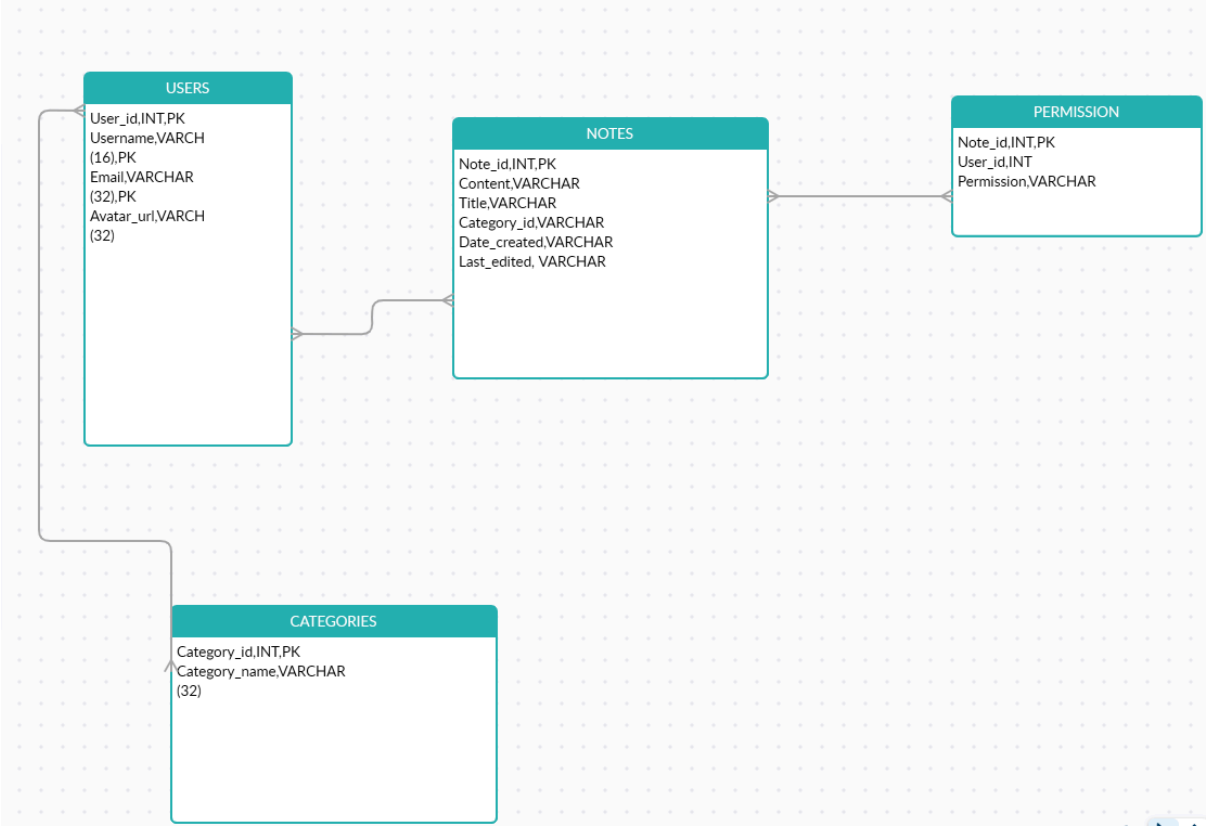
Introduction

This project involves developing a collaborative note-taking web application using a React.js front-end and Node.js for the back-end, connected to a PostgreSQL database. The application allows multiple users to collaborate in real-time on shared notes, offering markdown support and ensuring secure user authentication. The objective is to provide an intuitive, responsive user interface styled with Tailwind CSS and real-time collaboration using WebSockets.

Use case diagrams



Data modelling



Operating environment and dependencies

1. Operating Environment

1.1 Development Environment

- **Operating System**: Linux is used by all team members for development.
- **Node.js**: Version v20.16.0 is required for running the server-side code.
- **Package Manager**: The project uses pnpm to manage dependencies.

1.2 Database Environment

- **PostgreSQL**: Version v14.13 is used for the relational database.
- **Supabase**: A managed service for PostgreSQL is utilised.
- **Dotenv**: This library is used to manage environment variables, including database credentials.

1.3 Frontend Environment

- **React.js**: Version v18.3.1 is used for building the user interface.
- **Tailwind CSS**: Version v3.4.13 is used to ensure consistent styling across the app.

- **Markdown Renderer**: The react-marked library (v9.0.1) is used to render markdown content.

1.4 Hosting Environment

- **Server**: A Node.js server handles API requests and WebSocket connections via Socket.IO.
- **GraphQL**: Used to manage API communication between the frontend and backend.

2. Major Dependencies

2.1 Frontend Dependencies

- **React**: Core library for building the user interface.
- **Tailwind CSS**: A utility-first CSS framework for maintaining consistent and flexible styling.
- **ReactMarked**: A library used for parsing and rendering Markdown content within the app.
- **React Router**: Used for managing navigation and routing within the application.
- **Vite**: A fast, lightweight build tool and development server that is used to bundle and serve the frontend application, optimising the development experience with fast hot module replacement (HMR).
- **Apollo Client**: A fully-featured GraphQL client used to fetch, cache, and manage application data via GraphQL queries and mutations. It integrates well with React for managing state and data fetching.
- **emailjs.com**: A service used for sending emails directly from the frontend without needing a server, used for sending emails to reset the password.
- **Socket.IO Client**: A client-side library for establishing WebSocket connections to a server, enabling real-time communication between the frontend and backend.
- **Chakra UI**: A simple and accessible component library for React, providing a range of pre-built, customizable UI components..

2.2 Backend Dependencies

- **Socket.IO**: Used for real-time communication via WebSockets.
- **jsonwebtoken**: For generating and verifying JSON Web Tokens (JWT) used in authentication.
- **bcryptjs**: For securely hashing passwords before storing them in the database.
- **dotenv**: For managing sensitive environment variables, including database credentials and secret keys.
- **CORS**: Middleware that enables Cross-Origin Resource Sharing, allowing the server to specify which origins are permitted to access resources, useful for handling frontend-backend communication across different domains.
- **Express**: A minimalist web framework for Node.js, used to create the server and manage routing and middleware.

- **GraphQL**: A query language for APIs that allows clients to request the exact data they need, facilitates efficient communication between the frontend and backend.
- **Apollo Server Express**: An Express-compatible middleware for setting up a GraphQL server, used to handle GraphQL queries and mutations from the frontend.
- **Node.js**: The JavaScript runtime used on the server-side for running backend code, handling HTTP requests, and interacting with the database or other APIs.

2.3 Development Dependencies

- **ESLint**: A tool for linting JavaScript code to maintain consistent code quality.
- **Prettier**: A code formatter that automatically formats code to ensure a consistent style across the project, making it easier to maintain and read.
- **Autoprefixer**: A plugin that automatically adds vendor prefixes to CSS properties, ensuring that the app's styles are compatible with different browsers and their versions.
- **pnpm**: A fast and efficient package manager for managing project dependencies, offering a more optimised approach to installing and linking packages.

Authentication

3.1 Difference Between Authentication (AuthN) and Authorization (AuthZ)

3.1.1 Authentication (AuthN)

AuthN is the process of verifying the identity of a user or entity to ensure that they are who they claim to be. Usually, this entails comparing stored data in the system with identification details like a username and password. When a user inputs their email address and password, for example, the system checks these details before providing access. Tokens created following a successful login are also validated for follow-up requests, guaranteeing the user's authentication for the duration of the session.

3.1.2 Authorization (AuthZ)

AuthZ focuses on determining what actions an authenticated user is allowed to perform. The system determines what resources or data a user can access by looking at their role or permissions after their identification has been authenticated. For instance, a normal user may be able to see and modify their own notes and dashboard, but only the person who generated the note can remove it. Based on their position within the system, authorization guarantees that users have a particular amount of access.

3.2 Authentication Workflow

3.2.1 JWT (JSON Web Tokens)

- The system uses **JWT** for stateless authentication. Upon login, a JWT is generated and signed with a secret key.
- **Access Tokens**: Encoded user information (e.g., username, email) is sent to the client and used for verifying identity in subsequent requests.
- **Token Expiration**: Access tokens expire after a set period (e.g., 1 hour), after which the user must reauthenticate or use a refresh token.

3.2.2 Password Hashing

- User passwords are securely hashed with **bcryptjs** before being stored in the database.
- During login, the system hashes the provided password and compares it to the stored hash to authenticate the user.

3.3 Libraries and Tools for Authentication

3.3.1 jsonwebtoken

- The **jsonwebtoken** library is responsible for providing the tokens used to authenticate the user and the user's actions.

3.3.2 bcryptjs

- **bcryptjs** is used for securely hashing passwords before storing them in the database. It implements strong cryptographic hashing to protect user passwords.

3.4 Authentication Steps

3.4.1 User Registration

- Upon registering, the user's password is hashed and stored securely.

3.4.2 User Login

- When a user logs in, the system verifies the provided credentials.
- If successful, a JWT access token is issued and sent to the client.
- This token is saved in local storage.

3.4.3 Token Verification

- For each request to a protected route, the server verifies the JWT in the Authorization header.
- If valid, the user's identity is confirmed, and they can access the requested resource.

3.4.4 Session Management

- JWT tokens are stored client-side (e.g., cookies or localStorage), making the system stateless and scalable.

3.4.5 Logout

- The user logs out by removing the token from client storage.

High level description of design patterns for client and API

Client

The client-side architecture implements several key design patterns to promote maintainability, scalability, and user-friendly interfaces. The component-based architecture is central to the structure, where individual pages like the Login page or the Register page and reusable components like input fields and buttons are modularised, making it easier to manage and scale the application. State management is handled using React's useState hook, which ensures dynamic updates to the user interface as data changes (e.g., user input, avatar uploads). The use of API's like Supabase is abstracted through asynchronous functions, promoting separation of concerns and enhancing reusability.On top of this, responsive UI design is achieved through CSS and Tailwind CSS frameworks, ensuring the application is mobile-friendly and accessible across devices.

API

The API follows a GraphQL-based architecture, which allows flexible querying of data. It uses resolver functions to handle specific data-fetching or mutation operations, promoting separation of concerns and clear delegation of responsibilities. The Apollo Client allows the front-end to interact with these API queries and mutations, ensuring smooth, efficient data exchange. The API also uses asynchronous processing to handle requests efficiently, ensuring non-blocking execution for better performance. Schema-driven design is another key pattern where GraphQL schemas define the structure of data and operations, ensuring strong typing and validation across API endpoints. The use of RESTful principles for other API endpoints and a modular approach for routes, queries, and mutations enhances scalability and reusability.

Contributions

Tasheel → Database, CRUD points, backend
Robert → Sockets, backend, API

Keuran → Integration, backend, Socket
Nicole → Report,Frontend ui design,Login and register functionality
Zirna → Report, frontend ui design, email verifcation
Katlego → Frontend Ui Design, Report, avatar Functionality