NoteBody

Collaborative Markdown Note-Taking Web App

Chris Pretorius: 24694266
Dirk Bester: 25137018
Marcell Muller: 23603097
Jano Beukes: 23551399
Wian van Wyk: 24840653
Schalk Visagie: 25349589

Contributions:

- Chris: *Back-end Design, *Database Design
- Schalk: *Front-end, *Avatars, *Email validation
- Wian: *Front-end, *Note Filtering,
- Jano: *Note Deletion, *Report
- Marcell: *Back-end, *Database

Table of Contents:

Introduction: 3
Use case Diagram: 4
Data Modelling: 6
Operating Environment: 7
Design Patterns Used in the Note-Taking Web App 9

Table of Figures:

Figure 1: Use Case Diagram 4
Figure 2: ER Diagram 6

Introduction:

In today's digital age, the ability to seamlessly capture, organize, and share information is paramount. With remote collaboration becoming a mainstay of both educational and professional environments, the demand for tools that facilitate real-time collaboration on shared content has significantly increased. Addressing this need, this project presents a collaborative markdown note-taking web application, designed to offer users an intuitive platform for creating, editing, and sharing notes in real-time.

At its core, the application merges the simplicity of markdown—a popular lightweight markup language known for its human readability—with the dynamism of real-time collaboration. By leveraging the markdown rendering capabilities of Marked, users can view their content transformations instantaneously, making the note-taking process both efficient and interactive.

Moreover, the application is built using some of the most sought-after web technologies. React.js, renowned for its component-based architecture, powers the user interface, while Tailwind CSS ensures a responsive and modern design. On the backend, Node.js paired with Express offers a robust API, and PostgreSQL ensures data integrity and security. The real-time collaboration feature, which stands as the cornerstone of this application, is enabled using WebSocket—a protocol that provides full-duplex communication channels over a single TCP connection.

Additionally, understanding the importance of security and user-centric features, the application incorporates features such as user authentication, profile customization, and advanced search and filter capabilities for notes.

This report delves into the architecture, design, and functionalities of the collaborative markdown note-taking web app, providing insights into its data models, operating environment, and design patterns. Through this exploration, we aim to underscore the application's potential to revolutionize the way users interact with notes in a collaborative setting.

Use case Diagram:

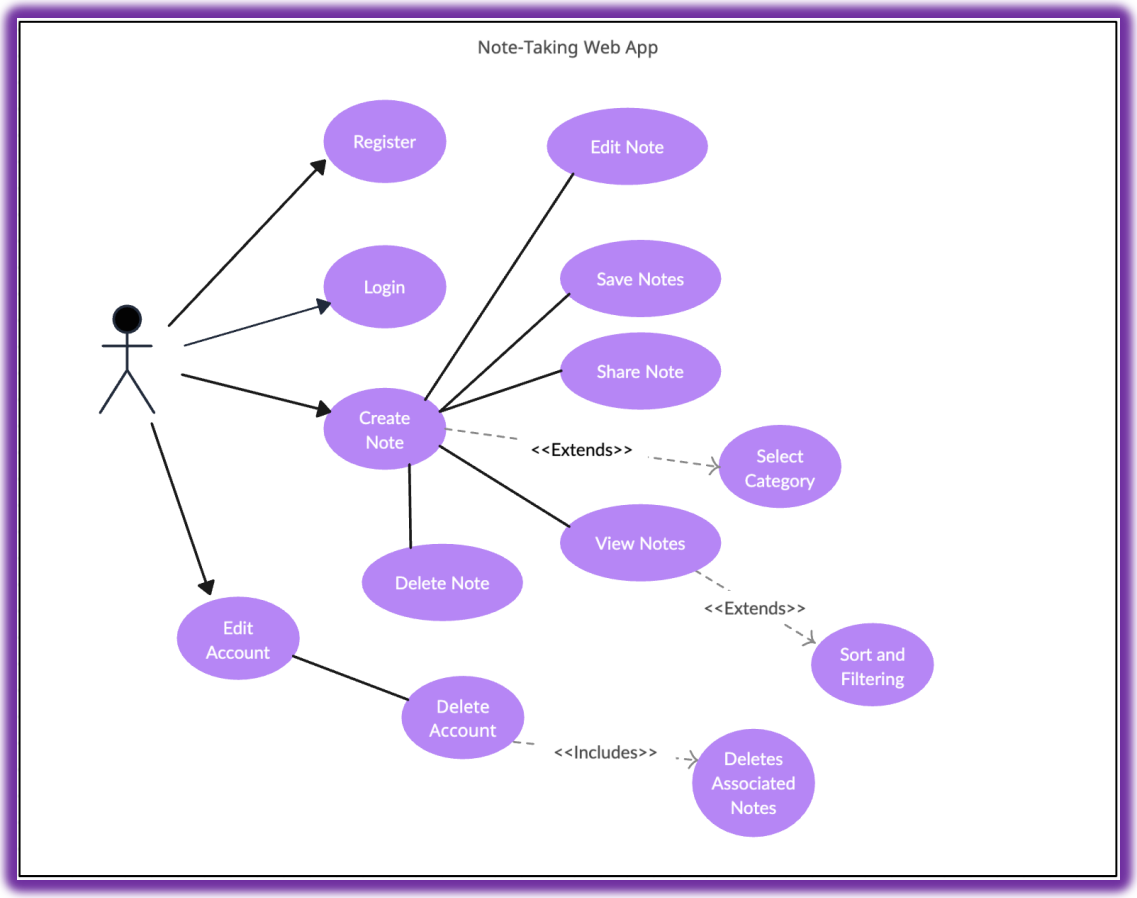

Figure 1: Use Case Diagram

1. Register:
    - Allows a new user to create an account by providing necessary details such as a unique username, email, and password. Once registration is successful, the user becomes a part of the system database.
2. Login:
    - Enables existing users to access the application by providing their registered email/username and password.

3. Create Note:
    - After logging in, a user can create a new note using markdown. The note is stored in the database and can be accessed anytime for viewing or editing.

4. Edit Note:
    - Allows the user to modify an existing note. Any changes made are saved in real-time, ensuring that the note always displays the most recent version.
5. View Note:
    - Users can view any of their created or shared notes. The content is rendered using markdown formatting.
6. Share Note:
    - A user can share a note with another registered user, granting them access to view or edit the note.
7. Search Notes:
    - Users can search for specific notes based on the title, allowing for quick and easy access to desired content.
8. Edit Profile:
    - Allows users to modify their profile details, such as username, email, and avatar. Changes are updated in the system database.

*Extended cases:

9. Delete Account with Associated Notes:
    - When a user opts to delete their account, not only is their profile information removed from the system database, but all notes associated with that account are also deleted. This ensures that no orphaned notes remain in the system after a user's departure. This use case includes the removal of both the user account and its associated notes to ensure data integrity.

10. Sort and Filter Notes (Extended):
    - While a user is viewing their list of notes, they have the option to enhance their viewing experience by sorting the notes based on the most recent edits or by filtering them according to specific categories. This use case extends from the basic "View Notes" functionality, offering users a more tailored approach to accessing their content.

11. Select Category (Extended from Create Note):
    - During the process of creating a note, users have the capability to categorize their content for better organization and future referencing. By assigning a category, users can group similar notes together, making it easier to locate related content. This use case is an extension of the "Create Note" functionality, ensuring every note created has the potential for categorization.

Data Modelling:

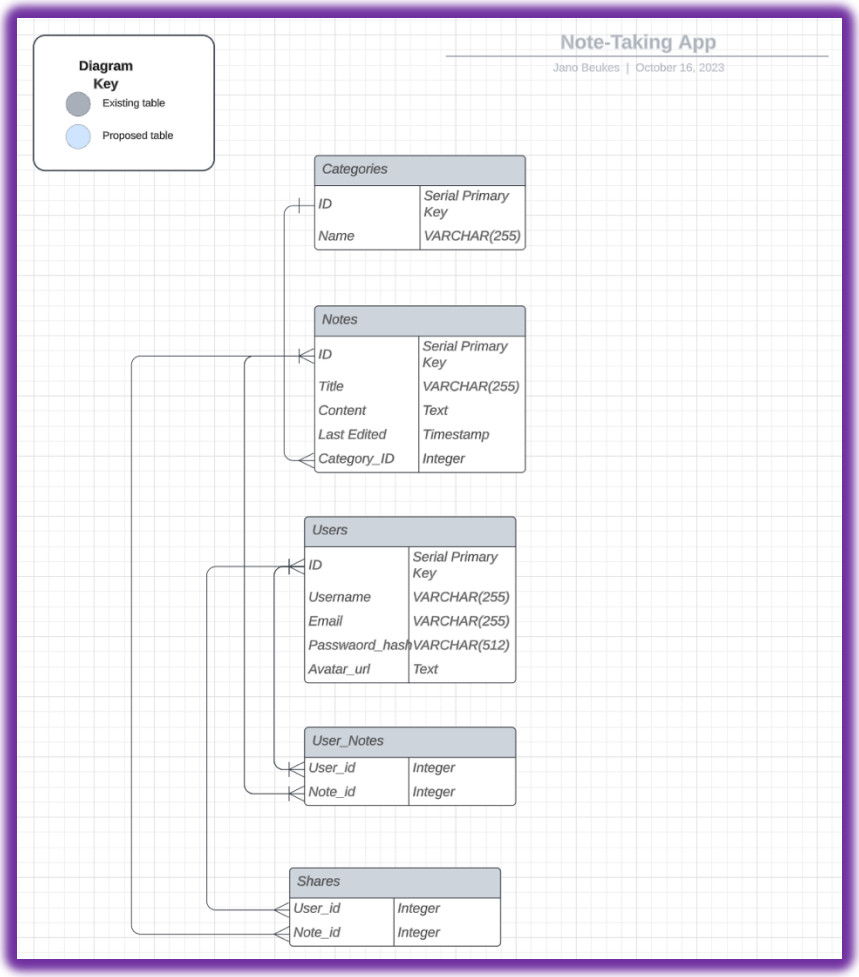

Figure 2: ER Diagram

The Entity-Relationship (ER) diagram for our markdown note-taking application elucidates the structure and connections within our database. Central entities include Categories, Notes, and Users, with Categories having a one-to-many relationship with Notes indicating a category can encompass numerous notes. Users form the core, having many-to-many relationships with Notes through intermediary tables User_Notes and Shares, signifying a user can both create and share multiple notes.

Additionally, the diagram highlights essential attributes for each entity: for instance, Notes possesses attributes like title and content while Users maintains data such as username and email. The relationships, marked with descriptors like "owns" and "shares", convey the diverse interactions possible between notes and users, embodying the collaborative spirit of the application.

In the design of our application's database, we adhered to the principles of normalization up to the 3NF (Third Normal Form). This ensures that every non-key attribute is functionally dependent on the primary key, and there is no redundancy. By doing this, we ensure efficient data retrieval, minimize the database size, and ensure that our system is scalable and performs optimally.

Primary Key Constraints: Tables such as Categories, Notes, and Users have primary keys (id attributes) that provide a unique identifier for each record, ensuring that there are no duplicate entries in these tables.

Foreign Key Constraints: Our many-to-many and one-to-many relationships, like those between Users and Notes or Categories and Notes, respectively, are maintained using foreign keys. This ensures referential integrity, meaning that there can't be a note referencing a non-existent category or user.

In our Users table, the password_hash column doesn't store the actual password but its hashed counterpart. When users attempt to log in, the entered password is hashed on-the-fly and compared to the stored hash in the database. This mechanism not only secures user data but also reinforces data integrity by ensuring that sensitive information, like passwords, remains uncompromised and intact.

Operating Environment:

The collaborative markdown note-taking web application was developed and tested in a diverse software environment, ensuring optimal functionality and responsiveness across various platforms.

Software Environment:

- Operating Systems: The application was rigorously tested on both Linux and Mac OS, verifying its cross-platform capabilities.
- Web Server: We employed Express for Node.js as our web server. Express, being a minimal and flexible Node.js web application framework, provided us with a robust set of features for web and mobile applications. The advantages of using Express include its swift performance, simplicity, versatility, and the ability to design single-page, multi-page, and hybrid web applications seamlessly.

- Database System: The system's backend relies solely on PostgreSQL, a powerful, open-source object-relational database system known for its proven architecture and extensibility.
- Programming Languages & Frameworks: The core development involved the use of Node.js, React.js, and Express. The application's aesthetics were crafted using HTML, CSS, and the utility-first Tailwind CSS framework, which enabled rapid design and customization.
- Browsers Supported: Although the application's primary testing occurred on Google Chrome, leveraging its developer tools for efficient debugging, it also showcased commendable compatibility with Firefox.

Hardware Environment:

While the application is designed to be lightweight, a recommended hardware specification for running the server would be a machine with at least 4GB RAM and a dual-core processor. This ensures smooth operation, especially when managing multiple user interactions simultaneously.

Network Environment:

A stable internet connection is essential for the seamless functionality of the application, especially given its real-time collaboration features. While the exact bandwidth requirements can vary depending on the scale of usage, a standard broadband connection should suffice for most operational needs.

External Dependencies:

- Third-Party Services: Our application incorporated "Marked" for live markdown rendering, enabling real-time visualization of user input in the markdown format.
- Version Control: Development practices leveraged GitLab for version control. The platform facilitated effective tracking of code changes and collaborative contributions. Additionally, to streamline task management and team coordination, we utilized Notion. This ensured a transparent division of tasks, tracking progress, and maintaining a synchronized approach to development.

Deployment & Scaling:

The application was primarily deployed locally on individual machines during the development phase. This allowed for intensive testing in controlled environments. Code integrations and enhancements were handled through GitLab by creating distinct branches, such as ‘frontend’, ‘backend’, and ‘database’, which were subsequently merged post thorough review.

The environment thus described encapsulates the diverse tools, platforms, and methodologies employed throughout the development of our collaborative note-taking application. It underscores our commitment to delivering a resilient, user-friendly, and efficient tool suitable for the dynamic needs of contemporary digital collaboration.

Design Patterns Used in the Note-Taking Web App

Client Side:

1. **Component-Based Architecture**: The client-side application is built using React, which follows a component-based architecture. Each feature or module is encapsulated into a separate component, allowing for modular and reusable code.
2. **Single Page Application (SPA)**: Using the `react-router-dom` package, the application is structured as an SPA, allowing for seamless navigation between different views without reloading the entire page.
3. **State Management**: State is managed locally within components using React's `useState` hook. For example, `MarkdownEditor` uses state to manage the markdown text, socket connections, and other functionalities.
4. **Effects & Lifecycle**: The application utilizes React's `useEffect` hook to handle side effects and component lifecycle events. This is evident in the `MarkdownEditor` where socket connections are established and cleaned up using effects.
5. **Real-Time Collaboration**: The `MarkdownEditor` uses WebSockets (via the `socket.io-client` package) to facilitate real-time collaboration. When a note is edited, changes are broadcasted to other users in real-time, allowing for simultaneous editing of notes.
6. **Form Management**: The `Register` component showcases a pattern for form management, where form input values are stored in a state and updated through event handlers. Additionally, there's form validation in place, particularly for password matching.
7. **Routing**: The `App` component sets up client-side routing, defining routes for different application views like login, registration, homepage, and user management. The use of `<Navigate to="/login" />` ensures users are redirected to the login page if they access the root path.
8. **Debouncing**: The `MarkdownEditor` introduces a debouncing pattern where the `saveCurrentNoteContent` function is debounced to prevent rapid, unnecessary API calls when the user is typing.
9. **Async Operations & Error Handling**: Asynchronous operations, such as HTTP requests to the backend, are handled using async/await. Error scenarios, like failed HTTP requests or WebSocket errors, are captured and handled, often with feedback to the user via alerts or console errors.
10. **Styling & Presentation**: Each component is accompanied by a corresponding CSS file (e.g., `MarkdownEditor.css`, `Register.css`) for styling. This ensures a clear separation of concerns between functionality and presentation.
11. **LocalStorage**: The application uses the browser's `localStorage` to store the user's token after successful registration, demonstrating a pattern for persisting data on the client side.

In summary, this client-side application adopts a modern React-based architecture with clear separation of concerns, modularity, real-time collaboration, and effective state and lifecycle management. It also emphasizes user feedback and error handling, ensuring a robust user experience.

API:

Modularization and Separation of Concerns:

- The API is architected in a modular manner, segregating routes, controllers, and utilities into distinct files. This approach ensures that routing logic remains separate from business logic, streamlining code manageability and scalability.

Middleware Pattern:

- The backend utilizes middlewares like jsonParser and verifyToken for request pre-processing. These middlewares ensure consistent validation, parsing, or other preparatory steps before the core controller functions take over.

RESTful Design:

- Adhering to REST principles, the API uses standard HTTP verbs such as GET, POST, PUT, and DELETE to represent operations. This design offers intuitive endpoints that map clearly to actions on resources like notes and users.

Singleton Pattern:

- Controllers and routers are instantiated once and then exported for use throughout the application. This structure guarantees a single, efficient instance of each module.

Dependency Injection:

- The NoteRoutes.js file manifests dependency injection by passing the io object (related to Socket.io for real-time functionalities) into the route. This decouples the route from the creation or configuration of the io object, enhancing flexibility.

Strategy Pattern:

- The token validation process in tokenRoutes.js employs a strategy pattern. The outcome of the isTokenValid function dictates the response strategy – be it success, expiration-based failure, or other failures.

Protection & Authentication:

- A bifurcation exists between authenticated routes and public ones. Middleware such as verifyToken is employed to safeguard certain routes, ensuring they are accessible solely by authenticated users.

Feedback and Error Handling:

- Comprehensive feedback mechanisms are built into the API. Through appropriate HTTP status codes and messages, especially during token validation, the API provides explicit guidance to the client on subsequent steps based on server responses.

Scalability Considerations:

- Due to the clear demarcation of routes, controllers, and other components, the API is well-poised for scalability. Incorporating additional features or routes can be achieved with minimal disruption to existing operations.

Entity-Relationship and Data Integrity:

- The database schema respects the many-to-many relationship between users and notes and the one-to-many relationship between categories and notes. This design guarantees data consistency and clear relational delineations. The normalization to 3NF (or BCNF) further ensures that the database is free from update, deletion, and insertion anomalies.

CRUD Operations and Database Interactions:

- The API offers endpoints that facilitate CRUD (Create, Read, Update, Delete) operations, aligned with RESTful design principles. Direct interactions with the PostgreSQL database occur through these endpoints, enabling actions such as creating notes, managing user accounts, and updating categories.

In essence, the backend API manifests an amalgamation of classic and contemporary design patterns befitting web applications. With an emphasis on modularity, clarity, and separation of responsibilities, it offers a solid foundation for ease of extension, robustness, and maintenance.