Table of Contents

1 Analysis

1.1 Identifying the Problem

1.1.1. Background:

Anton is an A-level student who does not have much time on his hands. With tasks and life goals snowballing around him, he needs to keep track of his progress on everything all at once. His current methods, which are paper-based or reliant on scattered digital tools, are inefficient and make it difficult to:

- Centralise and organise his tasks, goals, and habits in one place.
- Visualise his progress towards short-term and long-term objectives.
- Stay motivated by tracking his habits and achievements.
- Access his information across different devices (e.g., laptop, smartphone).

1.1.2. Identification

Anton's requirements for a notetaking app include:

- An all-in-one solution for both short-term and long-term tasks and goals.
- Easy organisation and accessibility of notes.
- Features to motivate and track progress, such as habit tracking.
- A user-friendly, cross-platform experience.

1.2 Identifying the End Users

The primary end users of this notetaking app include:

- **Students**: Students like Anton who juggle multiple tasks, assignments, and projects can use the app to organise their workload, set goals, maintain habits, and track progress.
- **Professionals**: Busy professionals can benefit from managing their tasks, projects, and goals to improve productivity.
- **Individuals Seeking Personal Organisation**: People aiming to enhance their daily routines and achieve personal goals can utilise features like habit tracking and progress visualisation.
- **Individuals with Specific Needs**: Those with conditions like ADHD or difficulties with time management can use the app's features (e.g., summarisation, advanced lookup, and sorting) to stay productive.

1.3 Research

Research Methodology:

To thoroughly understand the problem and design a suitable solution, I:

- Explored existing notetaking apps (e.g., Notion, Obsidian, Evernote, Google Keep) to identify their strengths and weaknesses.
- Read notable books on notetaking systems, such as *Building a Second Brain* by Tiago Forte and *The Bullet Journal Method* by Ryder Carroll.
- Conducted interviews with potential users, including Anton and other students, to gather insights into their preferences and pain points.
- Analysed feedback from online communities and forums discussing productivity tools.
- Created prototypes and tested them with users to refine the app's features.

My app was greatly inspired by books such as Building a Second Brain by Tiago Forte and The Bullet Journal Method by Ryder Caroll. However, such systems as one were too complex to me and I could not make myself use my physical Bullet journal for long enough due to the objective superiority of note taking apps, that were always in my pocket or my computer. As a result, I took some ideas from these two books and implemented them in my app.

1.3.1. Key Insights from Research

1.3.1.1 Books

1.3.1.1.1 Building a Second Brain

1.3.1.1.1.1 The CODE Method

One of the concepts in this book is the four-step CODE method, which is an acronym for Capture, Organise, Distill, Express.

- The first step in the Building a Second brain method is seeing the recurring themes and determining which knowledge does one want to interconnect. This information, most importantly, must be kept in one place. This is why my app must have different types of notes for various use cases.
- According to the author, an approach with folders may look good, but is not practical. Therefore, in the Organise part of the CODE method, a flexible organisation method is more preferable. As a result, using a tagging system would be more flexible and would allow a note to be shared across multiple categories.

Finally, adding tags such as “Active” or “Inactive” would help organise projects by their urgency.

- Distilling notes into bite-sized summaries would significantly speed up the process of obtaining and retrieving valuable knowledge. One possible solution for summarising notes would be to use AI language models, such as ChatGPT or Gemini to quickly summarise the notes based on a predetermined and optimised prompt. The prompt could automatically:
    - Highlight key terms
    - Retain links to websites
    - Insert placeholders if the initial note is incomplete

  By using the following summarization feature, the notes can be readable for the user even in a long time

1.3.1.1.1.2 The PARA Method

Using tags for the Organise step will aid with organising data from most actionable to the least actionable using the Projects->Areas->Resources->Archive. Additionally, to separate forgotten notes and automatically clean them up, a separate archive section should be introduced, which will also delete unused notes after some period of time.

1.3.1.1.2 The Bullet Journal Method

Even though the author is against the digital approach and bases their book on an analogue notebook, it is impossible to deny that digital is the future. Some concepts in this book are very specific to the Bullet Journal method and would be complicated to understand for users that haven’t read the book prior to opening the app. However, some concepts, such as Collections and Migration can be implemented, keeping the app usable for everyone. Just like in the previous example, Collections can be solved by sorting notes by types and using tags to group them together. Meanwhile, migrating tasks to the future or present would require a changeable due date on a note.

1.3.1.2 In-depth analysis of existing software

1.3.1.2.1 Notion

Notion is a cross-platform tool developed by Notion Labs, Inc. It allows the user to create their own note templates and organise their data in databases, tables. Advanced users can create completely custom solutions that can be considered their own websites.

Advantages:

- Notion is customizable

- Has AI tools to summarise notes and generate text
- Has support for organising entries in tables by tags
- Is cross platform

Disadvantages:

- Notion is very complicated to set up, so base users cannot use most of these features fully. Customization is very time consuming, so most users rely on paid templates to get any advanced functionality.
- From my experience, and experience of other Notion users I interviewed, the cross-platform experience is mediocre. The app is not optimised for touchscreens.
- The AI features are a paid subscription. Some users may underutilize the features and overpay for services they never use.

1.3.1.2.2 Google Keep:

Advantages:

- Simple and intuitive to use, making it accessible to users of all technical levels.
- Well-integrated with Google's ecosystem, allowing seamless synchronisation across devices.
- Offers dedicated apps for Android, iOS, and WearOS, ensuring availability on most platforms.

Disadvantages:

- Limited to basic note types, which restricts functionality for advanced users.
- Lacks features for tracking statistics, such as task completion or habit trends.
- Minimal options for customisation or organisation, making it less suitable for complex workflows.

1.3.1.2.3 Evernote:

Advantages:

- Offers robust organisational tools, including notebooks, tags, and advanced search capabilities.
- Supports multimedia notes, allowing users to attach images, audio, and files.

- Cross-platform support with apps for desktop, mobile, and web.
- Allows integration with third-party apps and services, enhancing functionality.

Disadvantages:

- Free version has significant limitations, such as device restrictions and reduced storage capacity.
- Interface can feel cluttered and overwhelming for new users.
- Features, like AI tools and offline access, are locked behind a premium subscription.

1.3.1.2.4 Apple Notes:

Advantages:

- Seamlessly integrated into Apple's ecosystem, providing excellent synchronisation across iPhone, iPad, and Mac devices.
- Simple and straightforward interface, ideal for quick note-taking.
- Supports rich text formatting, drawing tools, and file attachments.
- Includes collaborative features, allowing multiple users to edit notes in real time.

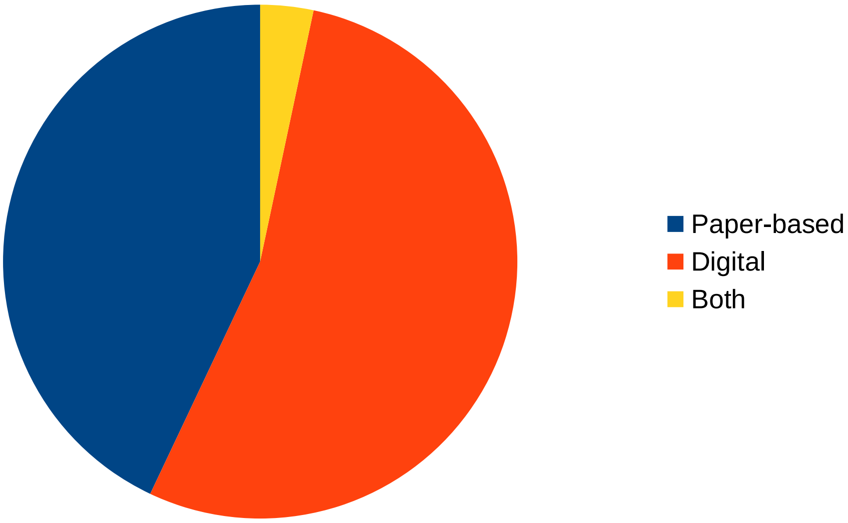
Disadvantages:

- Limited to Apple devices, restricting cross-platform usability.
- Organisation features are basic compared to competitors like Notion or Evernote.
- Lacks advanced capabilities, such as tagging or detailed analytics for tracking progress.

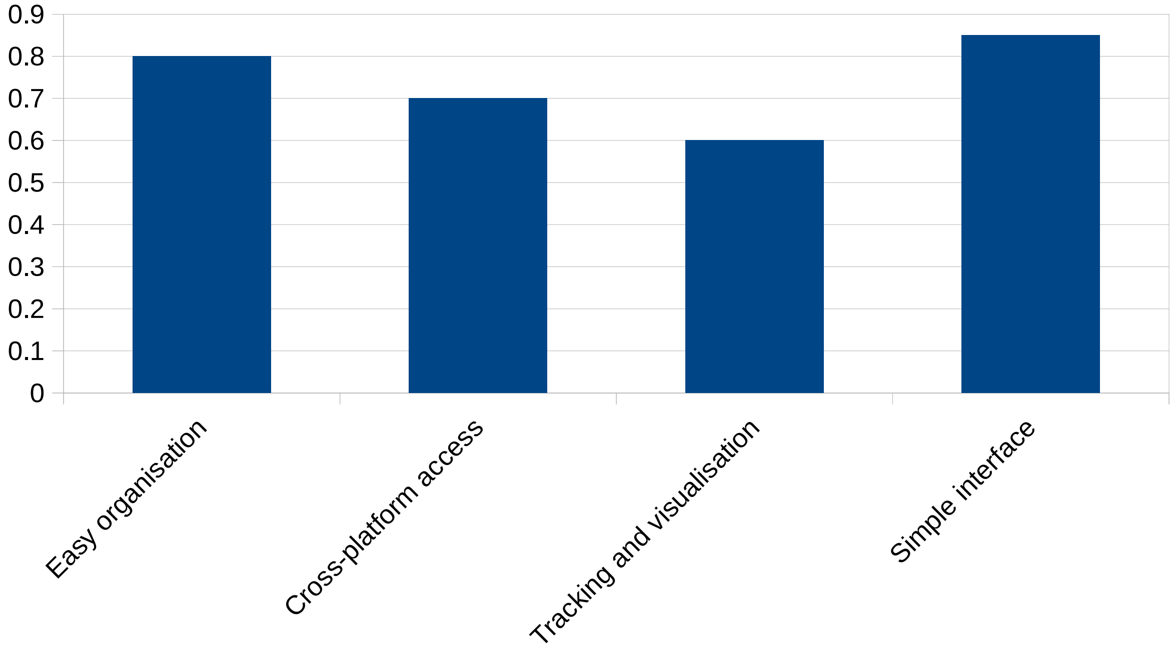
1.3.2. Survey results

To gather direct input from end users, I conducted a survey with around 50 respondents among my friends, their parents and my classmates. I tried to include people of various age groups to be able to account for the needs of as many users as possible. Below are the results of some conducted surveys:

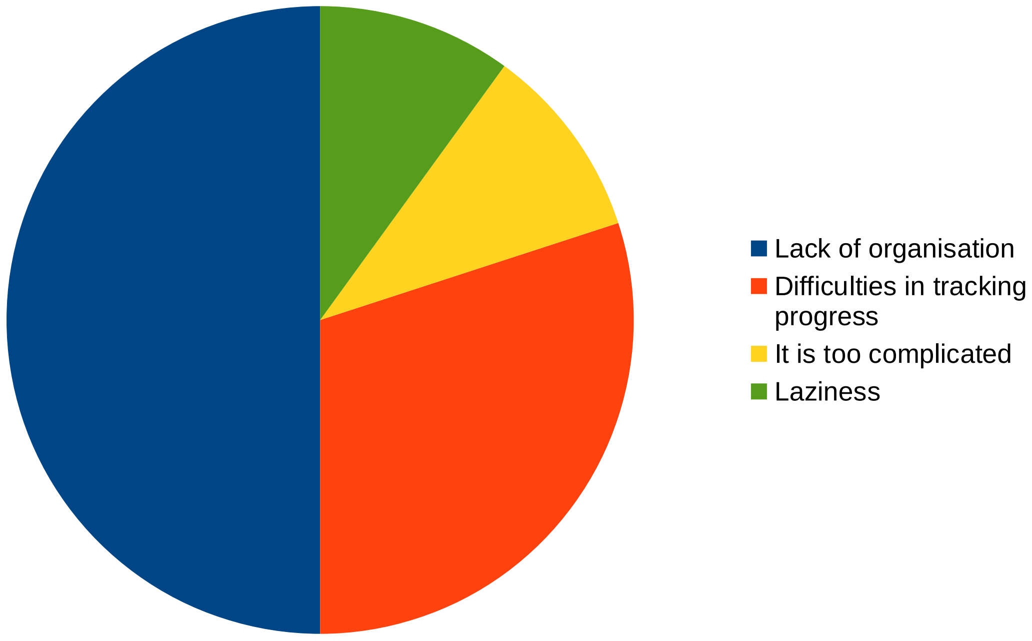
1.3.2.1 How do you currently organise your tasks and notes?



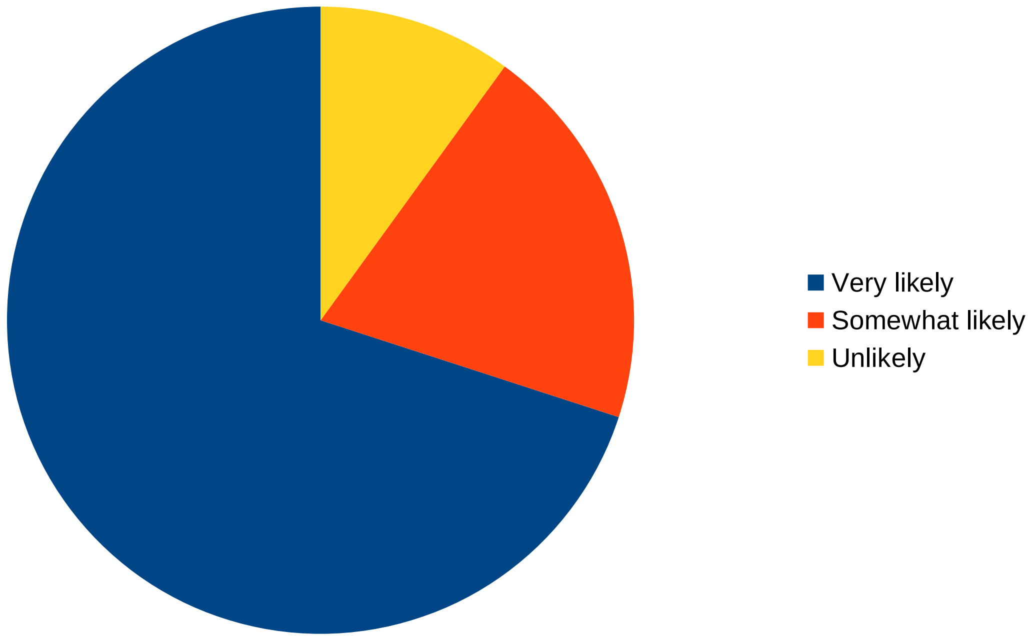
1.3.2.2 What features do you value most in a notetaking app? (Multiple choices allowed)



1.3.2.3 What are the biggest challenges with your current notetaking method?



1.3.2.4 How likely are you to use an app that combines notetaking and statistics?



1.3.2.5 Conclusion

This small survey reveals key insights into users current task and note organization habits, and favourite features in notetaking tools. The data suggests a strong preference for digital solutions, with 50% of respondents utilizing digital tools, compared to 40% relying on paper-based methods and only 10% employing a hybrid approach. The findings of the survey confirmed my ideas regarding the importance of usability and efficiency for modern-day users. The most valued feature among respondents was a simple and intuitive interface (85%), indicating a search for user-friendly tools that minimize friction and maximize productivity. Finally, a combined 90% of respondents expressed at least some level of likelihood (70% very likely, 20% somewhat likely) to use an app that combines notetaking and progress tracking in a simple way. This response suggests a significant market opportunity for such a tool.

To sum up, the survey results point to a clear demand for efficient, user-friendly, and digitally accessible notetaking solutions. Users prioritize easy organization, cross-platform accessibility, and intuitive interfaces.

1.4 Interview with Anton Kogun

Having concluded the survey on a wider potential customer base, I decided to probe deeper into the wants of potential users, so I conducted a short interview with the A-level student, Anton Kogun, from 1.1. Below, will be the results of this interview in a question-answer format:

What's the biggest problem you face with managing your schoolwork digitally?

Definitely the organization. I have notes scattered everywhere – some in Word docs, some in Google Docs, others in random note-taking apps. Plus, I use different task management apps, so I end up forgetting deadlines and losing track of what I've actually done.

If you had a magic wand and could create the perfect app for yourself, what's the one feature it absolutely must have?

A way to bring it all together. Notes, assignments, to-do lists, even my habit tracker – all in one place. And it needs to be easy to find what I need, fast.

And what about accessing your notes and assignments? Do you use multiple devices?

All the time! I use my laptop at home, my tablet at school, and sometimes even my phone. It needs to work seamlessly across all of them, without any ssues or syncing problems.

What would be your preferred way of accessing the application?

On PC, it would probably be a website. Web apps are fast and I can use my browser plugins to customize them to my liking. It is also to switche between tabs than multiple screens if I need to jor something down quickly. On the other hand, on my phone I would rather have an app, since some websites are not optimized for the smaller phone screens.

Imagine you've got hundreds of notes and tasks in this app. How would you quickly find the one you need?

Search is crucial, of course. But it needs to be smart. Searching by keywords is good, but what if I can't remember the exact words? Maybe something that understands the context of my notes. That would be amazing.

1.5 Modelling the problem

Here's how each objective addresses specific issues and integrates research findings:

1. **Provide a tagging system to categorise notes flexibly:** Helps users with easy organization (80% in the survey) and aligns with the "Building a Second Brain" emphasis on tags over folders. This also addresses Anton's need for centralization.

2. **Support multiple note types (e.g., text notes, checklists, habit trackers):** Addresses the need for an all-in-one solution for tasks, goals, and habits (Anton's requirement) and directly responds to the 90% survey interest in combined habit tracking and notetaking.
3. **Record and display metadata (e.g., creation dates, due dates):** Supports task management and progress tracking, addressing the challenges identified in the survey (40% difficulty tracking progress) and the Bullet Journal concept of migration (managing tasks over time).
4. **Allow summarisation of notes using AI to highlight key information:** Helps users perform the "Distill" step of the CODE method from "Building a Second Brain" and helps with quick lookup and long-term readability of notes.
5. **Enable quick and advanced note lookup based on context and keywords:** Addresses the need for easy accessibility of notes (Anton's requirement) and improves upon the limitations of basic search in some existing apps.
6. **Ensure cross-platform compatibility with automatic interface adjustments:** From the surveys conducted, (70% value cross-platform accessibility, 35% cite poor cross-device support as a challenge) and Anton's need for access across devices.
7. **Display user statistics, such as note completion rates and habit trends:** Directly addresses the need for progress visualization (Anton's requirement) and the survey results (40% difficulty tracking progress, 60% desire habit tracking/goal visualization).
8. **Allow users to customise the app’s appearance and preferences:** While not a primary pain point, this improves user experience and lets users
9. **Include an archive feature to manage unused or outdated notes:** Directly addresses the "Archive" component of the PARA method from "Building a Second Brain" .
10. **Implement security measures to protect user data:** Users will have an expectation of privacy while using the app to store their personal data

Keeping these 10 main objectives in mind, I formulated a list of tasks that I will need to complete in order to successfully implement this application.

1.6 Coding objectives

1.6.1. Database

1.6.1.1 Configuration and Security

- Create algorithms to generate unique IDs.
- Configure usage access by limiting permissions of clients that will be used with the databases.

- Ensure that the database is encrypted and adheres to modern security standards.
- Have a hashing algorithm to further encrypt user passwords.
- Implement input sanitization to prevent attacks, such as SQL injection.
- Use HTTPS for all communication.
- Upon login, provide users with an identity token to keep them logged in.
- Create functions to search notes by their vector representation instead of text.

1.6.1.2 Create tables for:

- Users
- Notes
- Tags
- Habits
- Statistics
- Goals
- Tasks

1.6.2. Design the backend functionality:

- Have functionality to retrieve, create, update and delete various note types and tags, using filters and pagination where needed.
- Be able to authentify and register new users.
- Link and unlink tags to and from notes.
- Complete/uncomplete tasks.
- Retrieve user statistics for several note types.
- Be able to save user preferences.
- Communicate with other APIs, such as one from Google or OpenAI to be able to pass submitted notes for AI summarization and return the result to the user.
- Implement an archiving function.
- Create algorithms to vectorize notes for context-based searches

1.6.3. User Interface

- Design the overall layout of the app.

- Have a screen to view all notes for a selected note type with pagination and filter controls, along with the titles and tags of the notes displayed
- Have a rich text editor for creating and editing notes (support for checklists and habit trackers).
- Design a navigation bar, searching screens and alerts.
- Have an account page where the user can edit their details.
- Design the statistics page and unique widgets with information for each note type.
- Ensure that the designs will adhere to accessibility standards.
- Make the application cross-platform compatible by choosing a technology that supports such development natively, such as React.
- Have error handling in place to prevent the app from crashing and communicate any errors to the user.

1.6.4. Validation

1.6.4.1 Frontend Validation

- Form validation on the UI to provide immediate feedback to users if data was incorrectly entered.
- Validate passwords and emails to avoid unnecessary requests.

1.6.4.2 Backend Validation

- Input validation for all API endpoints.
- Authentication and authorization checks to ensure users can only access their own data.
- Ensure that new entries have unique IDs before inserting them to the database.
- Impose a limit on the character length of submitted notes.

1.7 Proposed solution

My solution this feature-rich note-taking application leverages a robust and scalable technology stack. For the user interface, I will utilize ReactJS, a powerful JavaScript library for building dynamic and responsive single-page applications. To speed up design, I will use a component library called shadcn, which is open source, modern, adheres to acceccibility guidelines such as WCAG and is highly customizable. This choice ensures a smooth and intuitive user experience across various platforms. The backend will be constructed using Python and Flask, a lightweight and flexible microframework that allows for rapid development and efficient API creation. This combination provides a strong foundation for handling complex logic, data processing, and communication with the database. Data persistence and management will be handled by PostgreSQL, a powerful and reliable relational database system known for its data integrity, scalability, and support for complex queries. This technology stack offers a balanced approach, combining a modern frontend framework with a robust backend and database, enabling us to deliver a performant, scalable, and maintainable application that effectively addresses all the specified requirements above.

1.8 A

2 Documented Design

2.1 User data flow

2.1.1. Determining frontend-backend communication

2.1.1.1 Optimistic updating

Optimistic updating is a design pattern where a user interface is immediately updated based on the user's actions, assuming that the backend operation will succeed. This gives a more responsive user experience by avoiding unnecessary delays, especially when it comes to saving or creating notes. However, it will need careful error handling to prevent inconsistencies between the frontend and backend in case of errors.

2.1.1.2 Pessimistic updating

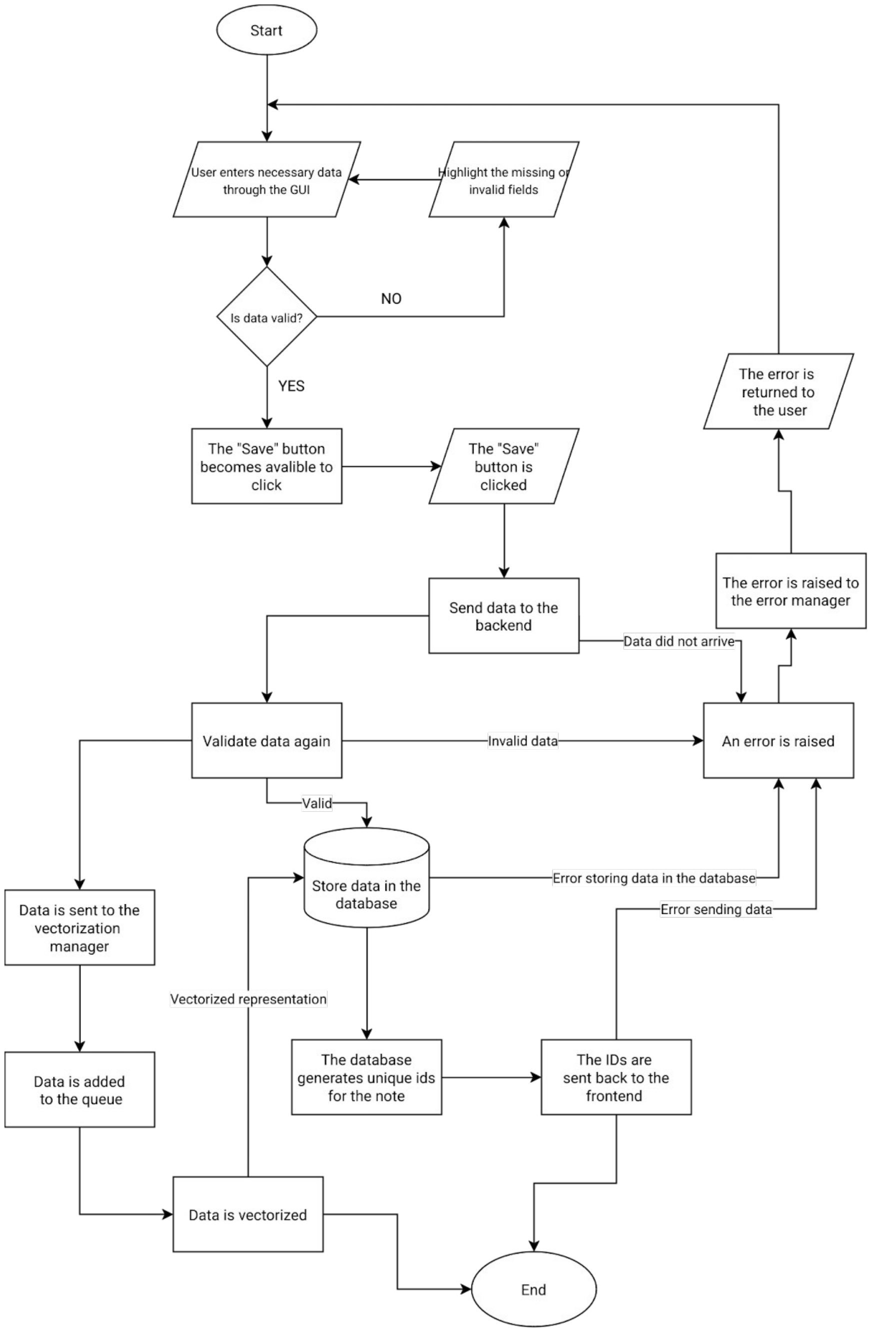
Pessimistic updating is a design pattern where the user interface is not updated until the backend operation has been successfully completed. This approach ensures data consistency but can lead to perceived delays, especially in scenarios with slow network connections or complex backend processing, which, for tasks such as vectorization, may take significant time and can be left for later.

2.1.1.3 Why I chose optimistic updating

With optimistic updating, changes made by the user are immediately reflected in the app's interface, providing a seamless and responsive experience. This is very beneficial for notetaking, where the user does not want to be concerned with the processing of their notes and just needs to write them down as quickly as possible. Pessimistic updating, on the other hand, requires the backend to process and validate the changes before they are displayed, leading to delays, amplified by poor connectivity. However, most of the benefits of pessimistic updating such as data validation can be addressed on the frontend too. For example, input text can be firstly validated on the frontend, displaying a message to the user if some values are missing or of wrong datatype. Using a type safe language such as TypeScript would ensure that the data types passed within the frontend are also correct. Finally, data can be additionally validated on the backend, which would allow for more complicated validation algorithms, as there is no longer a time constraint.

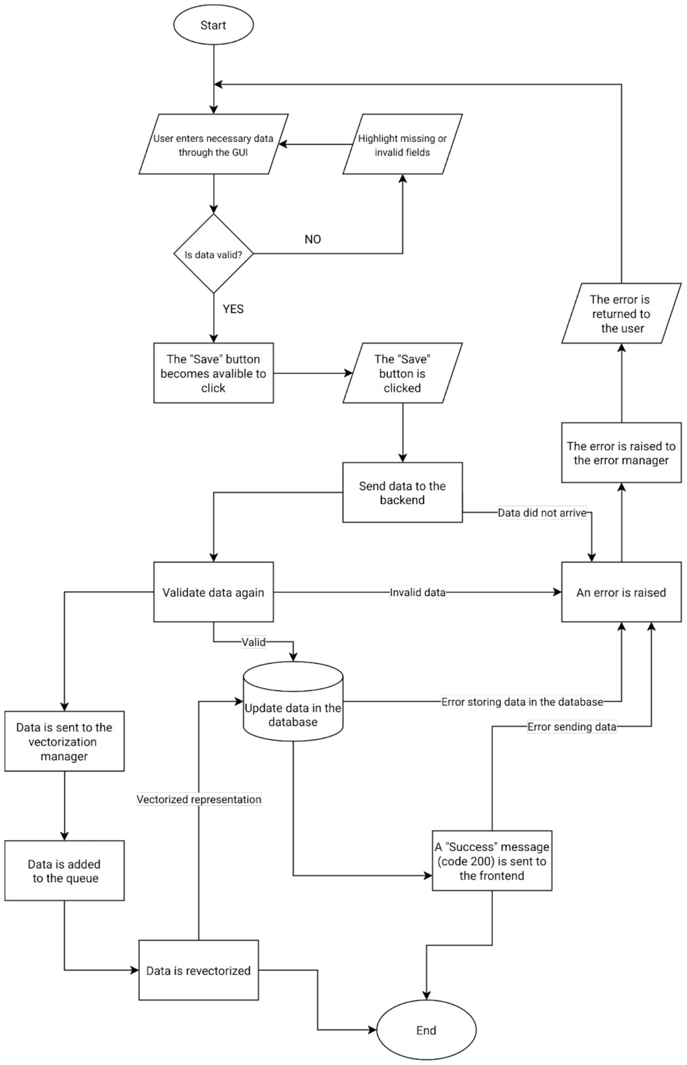
2.1.2. Creating data flow diagrams

2.1.2.1 Note creation


Start
User enters necessary data through the GUI
Highlight the missing or invalid fields
Is data valid?
NO
YES
The "Save" button becomes avalible to click
The "Save" button is clicked
Send data to the backend
Data did not arrive
The error is returned to the user
The error is raised to the error manager
Validate data again
Invalid data
An error is raised
Valid
Store data in the database
Error storing data in the database
Error sending data
Data is sent to the vectorization manager
Vectorized representation
Data is added to the queue
The database generates unique ids for the note
The IDs are sent back to the frontend
Data is vectorized
End

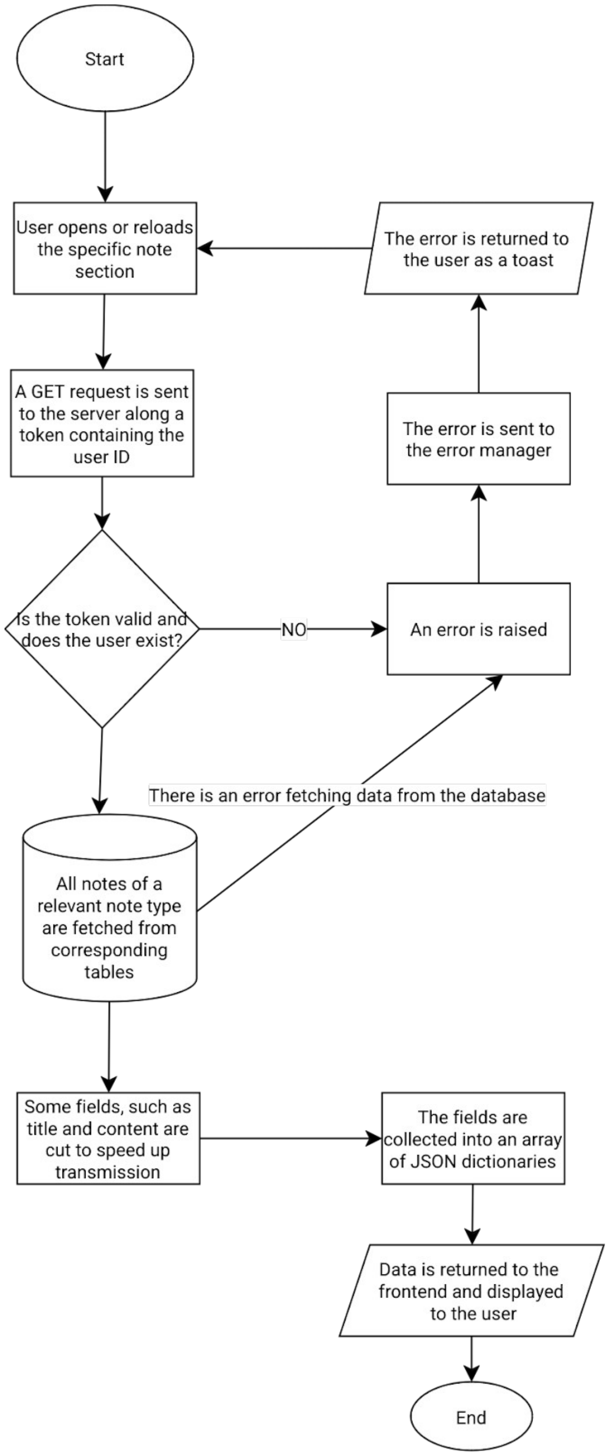
When the user clicks on “Create” button on the home page for a note type, a screen should be opened with the necessary fields, such as title, content, etc. Once the user enters all the details for a note to be considered valid by the validation script and all the validation errors have been resolved, the “Save” button is enabled allowing the user to click it and send the data to the backend. If the data gets to the backend successfully, it is again validated using and stored in the database. Additionally, meaningful parts of the note, such as its content, title and information from any additional fields are added to the queue to be vectorized. At the same time, the unique IDs, generated by the database, are returned to the frontend and, along the data entered by the user, are stored in a corresponding array for fast retrieval. The gui changes necessary elements from “Create” to “Edit” to mirror the success of the process and a success toast is displayed and the user can continue editing the note using the process mentioned below. Finally, when the server has the resources, data is vectorized and stored in the Notes table along the non-vectorized parts of the note.

2.1.2.2 Editing a note


Start
User enters necessary data through the GUI
Highlight missing or invalid fields
Is data valid?
NO
YES
The "Save" button becomes avalible to click
The "Save" button is clicked
Send data to the backend
Data did not arrive
The error is returned to the user
The error is raised to the error manager
Validate data again
Invalid data
An error is raised
Valid
Update data in the database
Error storing data in the database
Error sending data
Data is sent to the vectorization manager
Vectorized representation
Data is added to the queue
A "Success" message (code 200) is sent to the frontend
Data is revectorized
End

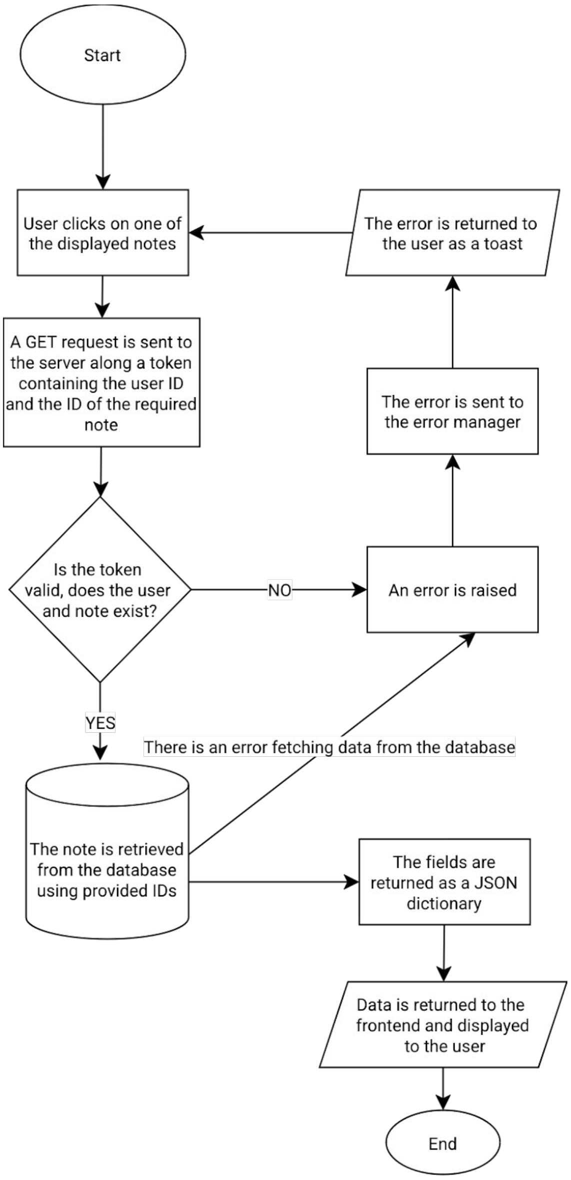
Generally, the note editing data flow is similar to creation, apart from first few details. To begin with, the “Save” button will not be enabled if no additional data was entered into the note. This is done to prevent unnecessary edit requests to the backend. In addition, if the database was updated correctly, instead of the IDs, a simple https code is sent to the frontend to indicate the success of the program. No additional toasts are shown to the user and the process can be repeated indefinitely as long as the saved note is valid.

2.1.2.3 Fetch all notes of a specific type


Start
User opens or reloads the specific note section
The error is returned to the user as a toast
A GET request is sent to the server along a token containing the user ID
The error is sent to the error manager
Is the token valid and does the user exist?
NO
An error is raised
There is an error fetching data from the database
All notes of a relevant note type are fetched from corresponding tables
Some fields, such as title and content are cut to speed up transmission
The fields are collected into an array of JSON dictionaries
Data is returned to the frontend and displayed to the user
End

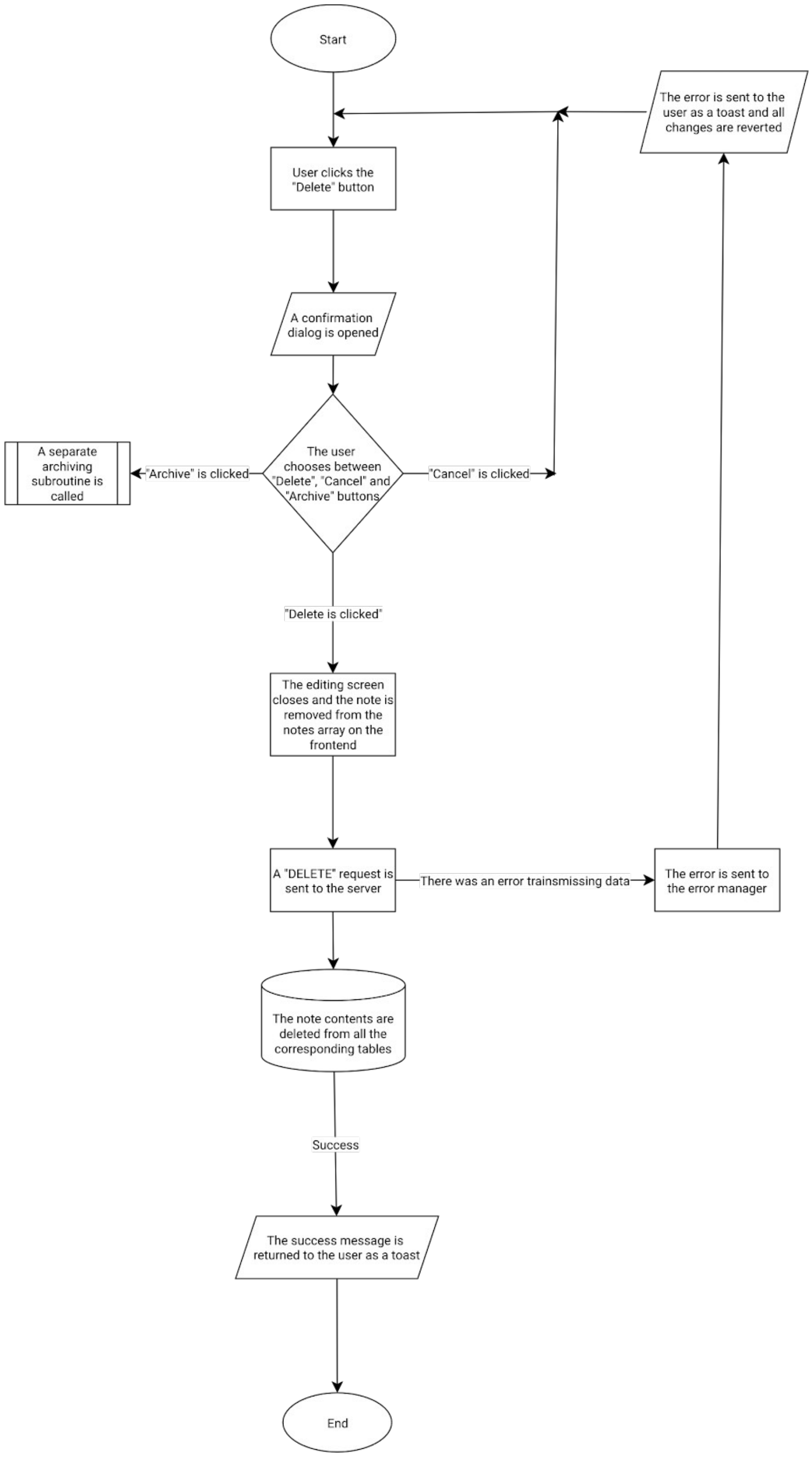
Every time a user loads a page of a specific note type, such as Tasks or Goals, a GET request is sent to the server alongside a JSON Web Token containing encoded user Id and information about the session. If the token is valid and the user exists in the database, all notes from a specific note type are fetched from the main Notes table and connected tables, such as Tasks and Subtasks for a Tasks note type. If the retrieval is successful. Since this page would only be used to find the correct note and then click on it to open a note editing page, there is no need to fetch the whole note, it makes more sense to fetch the first n characters from each field that would fit on the user's screen and then fetch the rest once the note is clicked. Therefore, before sending the data back to the frontend, some fields' contents are cut. Finally, data is returned as an array of JSON dictionaries and notes are displayed to a user.

2.1.2.4 Fetch one note


Start
User clicks on one of the displayed notes
The error is returned to the user as a toast
A GET request is sent to the server along a token containing the user ID and the ID of the required note
The error is sent to the error manager
Is the token valid, does the user and note exist?
NO
An error is raised
YES
There is an error fetching data from the database
The note is retrieved from the database using provided IDs
The fields are returned as a JSON dictionary
Data is returned to the frontend and displayed to the user
End

When the user chooses the note to edit, when they click, the GUI changes into an editing mode and a GET request is sent to the backend containing the ID of the clicked note along the JWT mentioned above. The main difference between fetching all notes and one note is that the fields are not shortened and only one note is returned. When the request succeeds, the user can continue to edit the note.

2.1.2.5 Delete a note


Start
The error is sent to the user as a toast and all changes are reverted
User clicks the "Delete" button
A confirmation dialog is opened
A separate archiving subroutine is called
"Archive" is clicked
The user chooses between "Delete", "Cancel" and "Archive" buttons
"Cancel" is clicked
"Delete is clicked"
The editing screen closes and the note is removed from the notes array on the frontend
A "DELETE" request is sent to the server
There was an error trainsmissing data
The error is sent to the error manager
The note contents are deleted from all the corresponding tables
Success
The success message is returned to the user as a toast
End

When a user attempts to delete a note, they should be first greeted by a dialogue which allows them to either cancel the deletion, continue with it or put the note into an archive instead. If the user chooses to proceed with the deletion, the note is removed from the corresponding notes array on the frontend and the DELETE request with a token, containing a user id and the ID of the note to be deleted. If the token is valid and the user is the owner of the note, its data is deleted from all corresponding tables. Finally, the success or error message is returned to the user as a toast.

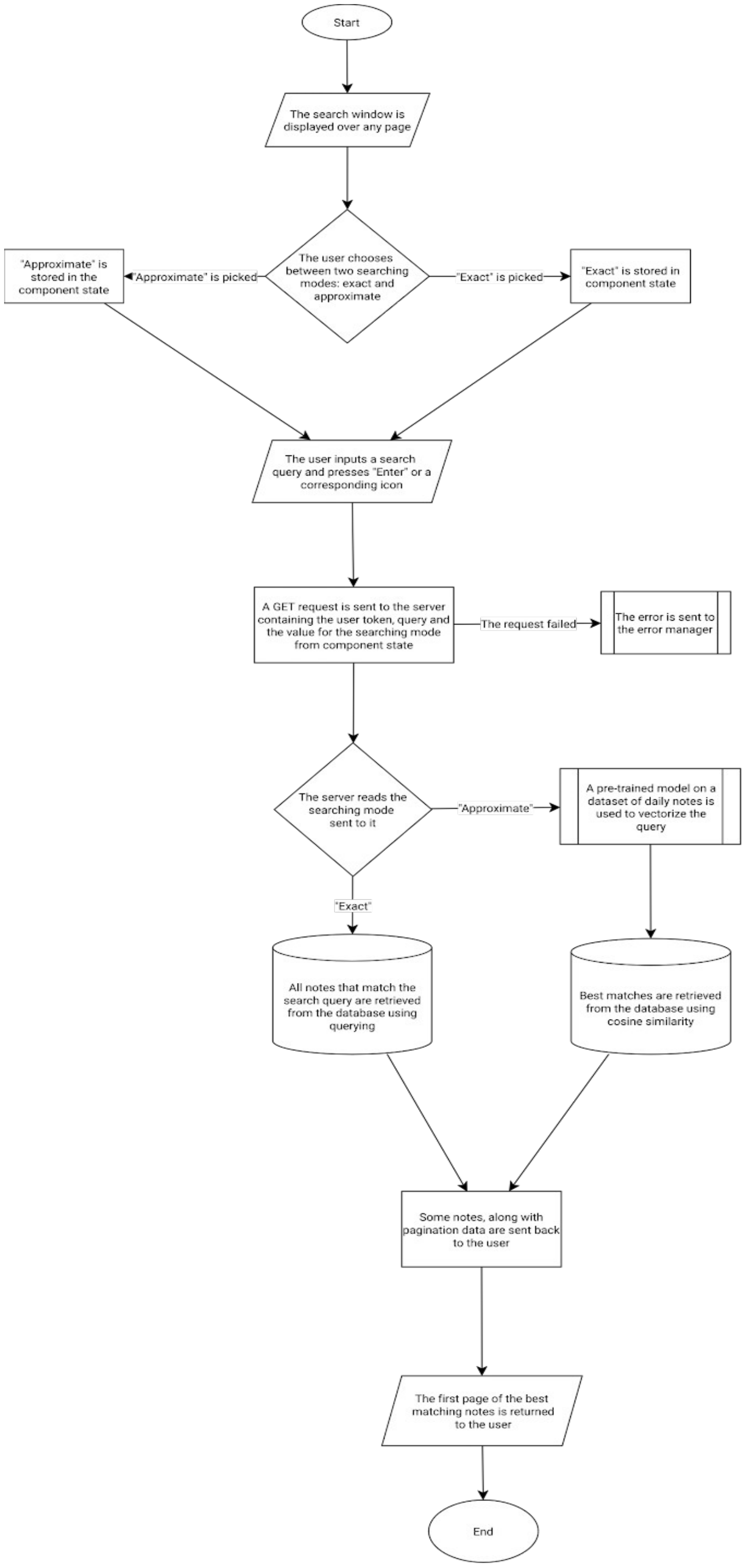
2.1.2.6 Searching for a note

To search for a note, the user will have to click a search icon that will always be present in the header of the page. Upon clicking, a popup will appear over the page with the search bar alongside a toggle. The user is provided with an option to choose between “Approximate” and “Exact” searching modes, the latter of which is toggled by default and stored in the component state. Then, the user enters their query and presses the enter key or a corresponding button. The query, user token and the search mode are sent to the server. If the token is valid, the server reads the value of the search mode and processes the query accordingly:

a) “Exact” search mode. The query will be used to look up title or content fields in the database using SQL querying

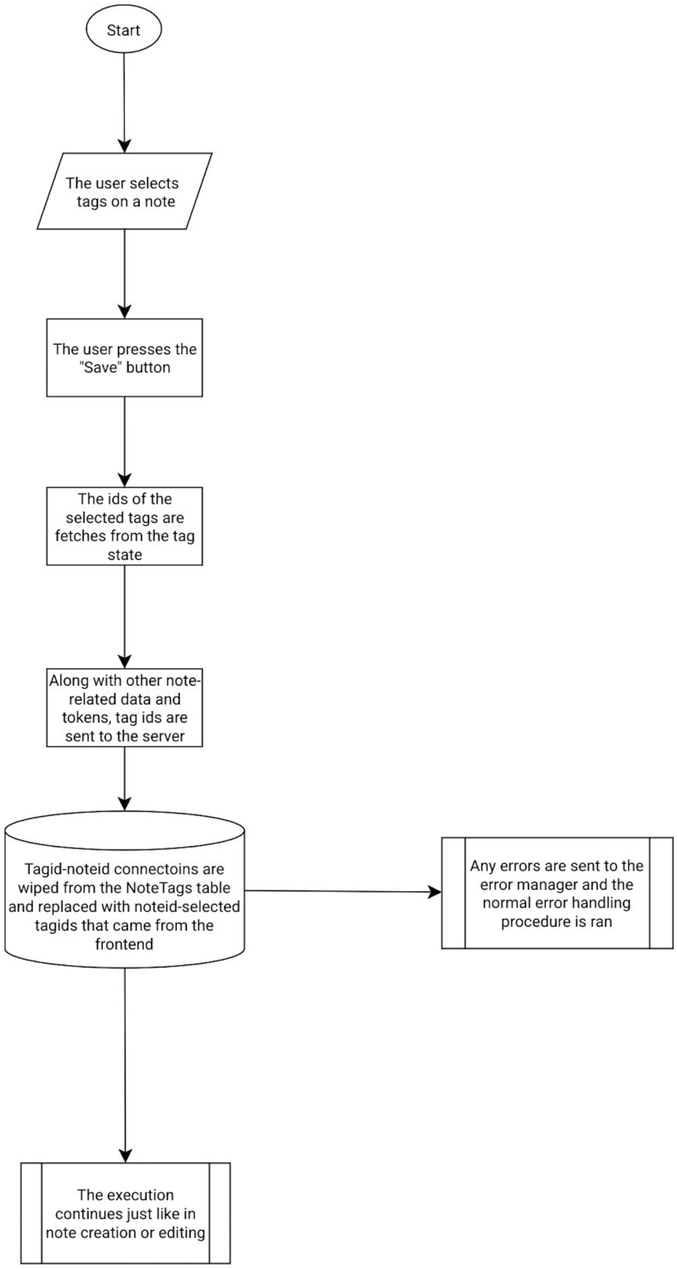
b) “Approximate” search mode. The query will be vectorized with a pretrained model and its vector representation will be compared to the stored vector representations of notes in the database using a function such as cosine similarity.

Once the notes have been retrieved, they are split into pages of 5 or 10 notes and, along with its pagination data, are sent to the user to be displayed. The user can use a selector on the window to switch pages of notes.


Start
The search window is displayed over any page
The user chooses between two searching modes: exact and approximate
"Approximate" is picked
"Approximate" is stored in the component state
"Exact" is picked
"Exact" is stored in component state
The user inputs a search query and presses "Enter" or a corresponding icon
A GET request is sent to the server containing the user token, query and the value for the searching mode from component state
The request failed
The error is sent to the error manager
The server reads the searching mode sent to it
"Approximate"
A pre-trained model on a dataset of daily notes is used to vectorize the query
"Exact"
All notes that match the search query are retrieved from the database using querying
Best matches are retrieved from the database using cosine similarity
Some notes, along with pagination data are sent back to the user
The first page of the best matching notes is returned to the user
End

2.1.2.7 Tag workflow

In general, the workflow associated with tags is similar to notes. Tags can be created, edited, viewed and deleted. Tags come with an additional field, which has their colour to be distinguishable not only by name. More differences start when notes need to be linked to tags. One way to do so is to have a state which stores the selected tags when a note is opened or created. Then, the ids are passed to the backend whenever the note is saved and then linked to their note id in a separate table, which will be stored in the database.


Start
The user selects tags on a note
The user presses the "Save" button
The ids of the selected tags are fetches from the tag state
Along with other note-related data and tokens, tag ids are sent to the server
Tagid-noteid connectoins are wiped from the NoteTags table and replaced with noteid-selected tagids that came from the frontend
Any errors are sent to the error manager and the normal error handling procedure is ran
The execution continues just like in note creation or editing

This procedure is executed alongside any note creation or editing procedure whenever the selected tag ids differ from the fetched ones.

2.1.2.8 Registering and logging users in

One way to ensure that users are correctly logged in, we can store a JWT (Json web token) in the browsers cookies. The token will be passed alongside any user request to the server, would include the user identity and expire after a set time period. This will cause the user to log in again, which would protect their personal information from being accesses by third parties.

The user data will be stored in the Users table. This may include their username, email and encrypted password to prevent it from being compromised in case of a data breach. Additionally, each user field in the table may store their preferences, such as dark or light themes and any additional information.

2.2 Backend

2.2.1. Designing the database

2.2.1.1 Detecting patterns in data representation

In order for my database to accommodate multiple note types without wasting space, we can use multiple tables to split the data from one note into multiple parts. Since any note can have a title, some basic content, an owner and tags, information, concerning the note, can be put into a table called Notes. Then, any other task types will pass the id of the note further into the database into note type specific tables, which will contain additional fields for a note.

Since the same tag can be used for many notes, it is unnecessary to store all the tags for each note in the Notes table. As a result, a Tags table needs to be created and connected to the Notes table with a one to many relationship.

2.2.1.2 Choosing the database

There are many different databases available with their own pros and cons. However, after trying multiple databases, such as MySQL and MongoDB, which had their own pros and cons, I finally landed on PostgreSQL, which satisfied all of my requirements for the notetaking app due to its robust features and capabilities:

2.2.1.3 Efficiency

It efficiently handles complex relationships between tables, such as one-to-many, many-to-many, and hierarchical structures, which will be useful in my app. Moreover, Postgres offers numerous built-in functions for data manipulation, aggregation, and analysis, simplifying tasks like searching, filtering, and sorting notes.

2.2.1.4 Data Types and Functions:

Postgres supports a wide range of data types such as UUID for primary keys, which will accommodate various note content formats and use cases.

2.2.1.5 Why use UUID?

UUIDs (Universally Unique Identifiers) are highly beneficial for generating unique IDs. UUID ensures that no two IDs will ever be the same, even across different systems or networks. UUIDs are also generated randomly, making it extremely difficult to predict or guess their values, which can improve security. Finally, they can be generated efficiently, making them suitable for high-performance applications.

2.2.1.6 Performance:

The database is designed to scale efficiently, handling large volumes of data and supporting high-performance queries, which will speed up note loading and retrieval.

2.2.1.7 Security:

Postgres supports user authentication, role based access control and encryption, which will encrypt sensitive user information in the notes and limit the extent to which the server can access the database to stop potentially malicious requests.

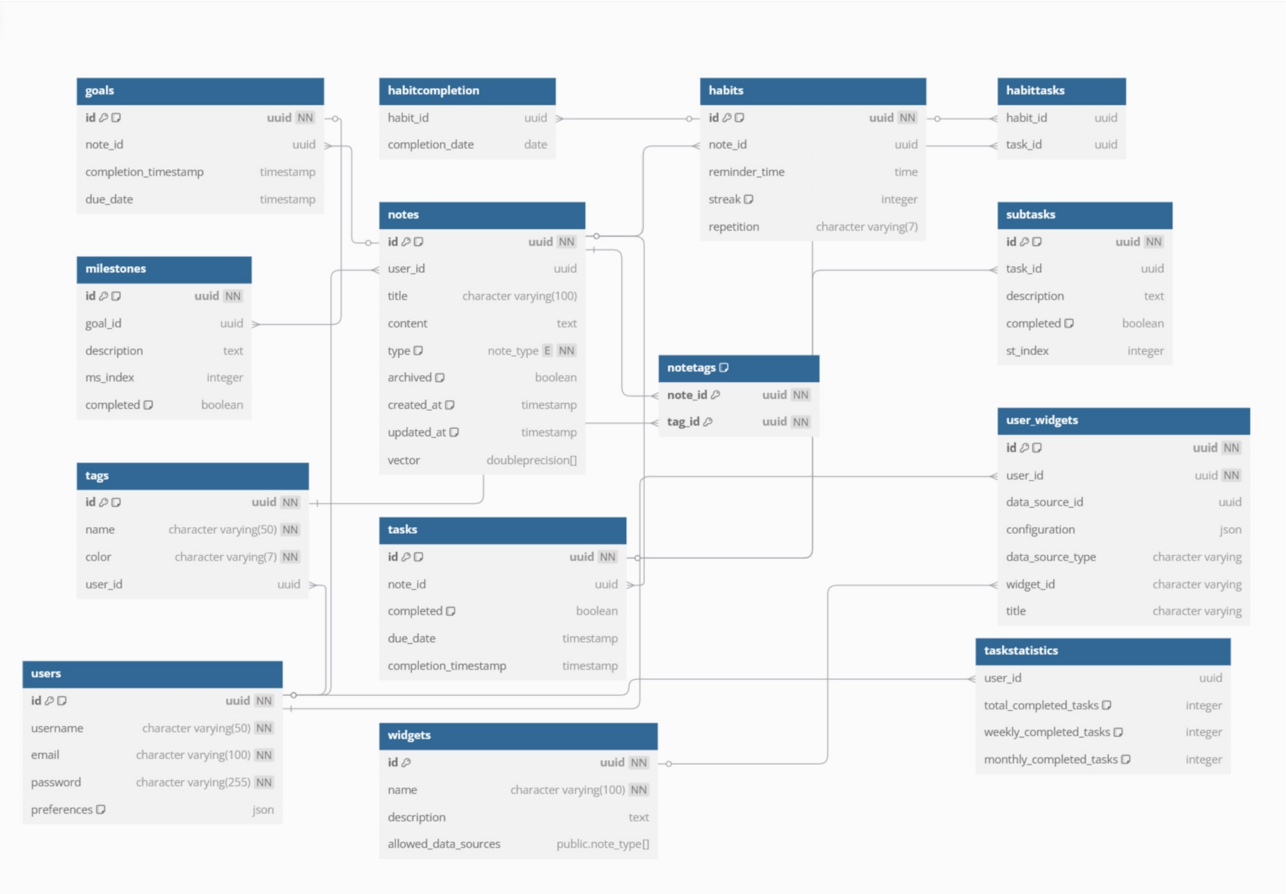
2.2.2. Database relationships

It is important to note that tables, such as events, eventtasks, and goalhistory were later removed from the design during the creation of the program as they were found redundant. For example, goalhistory would simply copy the completion timestamp of a goal, which is unnecessary repetition. In addition, several tables to store widgets were added

After picking the appropriate database, I started planning the tables that could be possibly useful for storing note and user data.

2.2.2.1 User data

Firstly, the user must be able to create their account. In order to keep records of users, each account must have a unique id, which will be generated using the UUIDv4 module of the UUID library in the database whenever a user is created. Each user must have a unique username, an email and an encrypted password. Additionally, each user table may also store their preferences and other optional data which may not need its own column in the preferences field.



```
CREATE TABLE users (
    id uuid DEFAULT uuid_generate_v4() NOT NULL,
    username character varying(50) NOT NULL,
    email character varying(100) NOT NULL,
    password character varying(255) NOT NULL,
    preferences json DEFAULT '{}'::json
);
```

2.2.2.2 Notes

The main table for each note, no matter its note type, is the Notes table. It contains the fields universal for all note types, such as the title, content, type and the vector representation of the note. Finally, each note is connected to a user using the user_id foreign key and has a few other fields, such as creation and update timestamps and whether a note is archived or not. Additionally, note type tables, such as habits, milestones, events and goals reference the note_id of a corresponding note in the notes table to store additional data specific for this note type, such as a due date, location or whether a task is completed or not. Finally, some note types may have their own little notes, such as subtasks in tasks or milestones in goals. These tables are usually similar to their parent note, as they may also have a completion boolean and a description, but they reference the corresponding Task or Goal id.

Main notes table:

```
CREATE TABLE notes (
   id uuid DEFAULT uuid_generate_v4() NOT NULL,
   user_id uuid,
   title character varying(100),
   content text,
   type note_type NOT NULL,
   archived boolean DEFAULT false,
   created_at timestamp without time zone DEFAULT CURRENT_TIMESTAMP,
   updated_at timestamp without time zone DEFAULT CURRENT_TIMESTAMP,
   vector double precision[]
);
```

Tables for the “Task” note type:

```
CREATE TABLE tasks (
 id uuid DEFAULT uuid_generate_v4() NOT NULL,
 note_id uuid,
 completed boolean DEFAULT false,
 due_date timestamp without time zone,
 completion_timestamp timestamp without time zone
);
```

```
CREATE TABLE subtasks (
   id uuid DEFAULT uuid_generate_v4() NOT NULL,
   task_id uuid,
   description text,
   completed boolean DEFAULT false,
   st_index integer
);
```

Tables for the “Habit” note type:

```
CREATE TABLE habits (
 id uuid DEFAULT uuid_generate_v4() NOT NULL,
 note_id uuid,
 reminder_time time without time zone,
 streak integer DEFAULT 0,
 repetition character varying(7)
);
```

Ability to link multiple tasks to a single habit

```
CREATE TABLE habittasks (
    habit_id uuid,
    task_id uuid
);
```

Tables for the “Goal” note type:

```
CREATE TABLE goals (
 id uuid DEFAULT uuid_generate_v4() NOT NULL,
 note_id uuid,
 completion_timestamp timestamp without time zone,
 due_date timestamp without time zone
);
```

A table for subgoals(checkpoints)

```
CREATE TABLE milestones (
 id uuid DEFAULT uuid_generate_v4() NOT NULL,
 goal_id uuid,
 description text,
 ms_index integer,
 completed boolean DEFAULT false
);
```

2.2.2.3 Tagging

Tags are independent from notes, so a corresponding table containing data about tags, such as their name and color, will be connected to notes using a separate table, called NoteTags, where note and tag ids are linked together.

```
CREATE TABLE tags (
id uuid DEFAULT uuid_generate_v4() NOT NULL,
name character varying(50) NOT NULL,
color character varying(7) NOT NULL,user_id uuid);

CREATE TABLE notetags (
note_id uuid NOT NULL,tag_id
uuid NOT NULL
);
```

2.2.2.4 Statistics

Finally, some tables containing data, such as statistics on goal, task and habit completion, are linked to each user and a corresponding note to provide the users with a history of their progress, even if the initial note is deleted.

```
CREATE TABLE taskstatistics (
    user_id uuid,
    total_completed_tasks integer DEFAULT 0,
    weekly_completed_tasks integer DEFAULT 0,
    monthly_completed_tasks integer DEFAULT 0
);

CREATE TABLE goalhistory (
 user_id uuid,
 goal_id uuid,
 completion_timestamp timestamp without time zone
);

CREATE TABLE habitcompletion (
 habit_id uuid,
 completion_date date
);
```

2.2.2.5 Widgets

The last two tables are the widgets and user_widgets tables. The widgets table is populated by the administrator using INSERT statements to create predetermined widget types with allowed data sources. When a user creates a widget, their widget goes into user_widgets and uses the widget table as a stencil to assemble a correct data structure that will be displayed as a widget

2.2.2.6 Conclusion

This database design prioritizes normalization by separating data into distinct tables, minimizing redundancy and improving data integrity. For instance, user data is stored separately from notes, and note types have their own tables, negating data duplication. This approach enhances data consistency and simplifies updates. Furthermore, the use of foreign keys establishes clear relationships between tables, enabling efficient data retrieval and querying. By optimizing table structures and utilizing appropriate indexing, this design aims to ensure efficient data storage and retrieval, providing a robust foundation for the application.

2.2.3. Backend file structure

The structure I came up with follows common Python project conventions and uses a combination of modules for separation of concerns and a utils directory for reusable utility functions:

- models
    - w2v.model
- modules
    - __init__.py
    - goals.py
    - tasks.py
    - universal.py
    - etc
- utils
    - __init__.py
    - word2vec.py
    - utils.py
    - etc

- app.py
- config.py
- validation.py

In this system, models will be used to store pre-trained models for NLP (natural language processing) tasks such as note vectorization for context-based search. The main logic will be happening in the modules folder, where the __init__.py file signifies that this directory is a Python package. Goals, Tasks and other files will handle functionality related to similarly named note types. These files contain all the CRUD (create, read,update,delete) functionality for each note type respectively and will interact with the SQL database and process request sent by the frontend. Universal.py will contain 'universal' endpoints such as site-wide search that are not significant enough to be put in a separate file.

On the other hand, more complicated algorithms, such as text vectorization do deserve a separate file, which will be placed in the utils folder of the root directory. Any remaining functions without an endpoint will be put in utils.py.

Additionally, app.py will host the Flask application and be the place where all the routes are defined and the Flask application is initialised. Config.py is excluded from source control as in contains confidential configuration settings, such as database connection credentials and other enviroment variables. Finally, validation.py will contain validation forms for user submitted data to sanitize the inputs in notes or to add an additional layer of type safety and security before inserting data into the database.

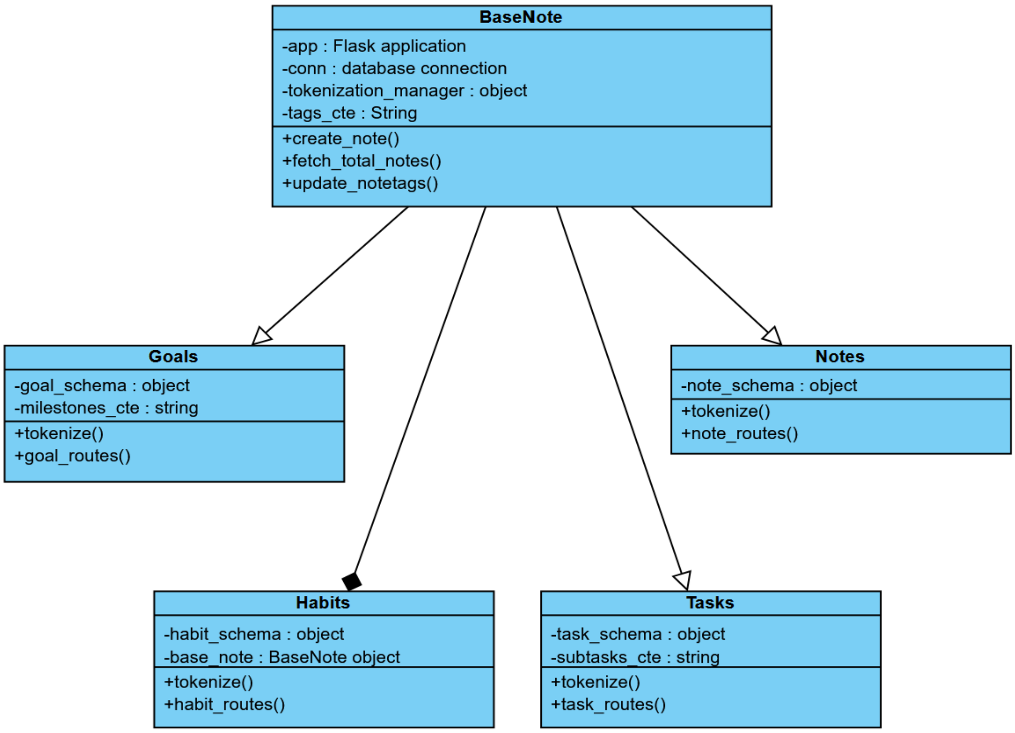
2.2.4. OOP

OOP (object oriented programming) is a programming paradigm that organizes software design around objects (instances of classes), rather than functions. By using OOP, we can structure the code in a modular and organized manner. This improves code reusability, maintainability, and flexibility.

2.2.4.1 Constructing note processing code

In the context of my note-taking app, we can apply OOP by creating classes to represent key entities like Notes, Users, and Tags. These classes would encapsulate data and methods that operate on that data.

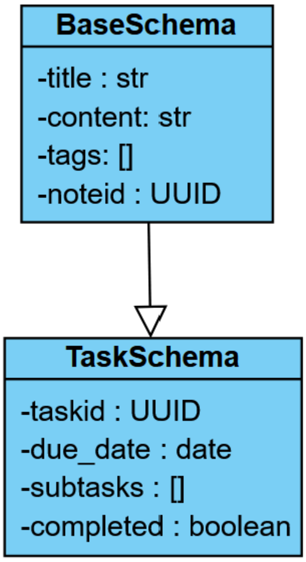
However, I noticed that most notes share similar base characteristics, such as tags, content and title and decided to use classes to mix and match actual api components to create an object that will be able to work with a specific note type. By doing that, I will encapsulate all logic for a note type in one place and create multiple classes for any note type by inheriting some properties, such as tag functionality or data cleanup for vectorization from the base class.



With the implementation above each note api inherits or composes all the necessary connections and managers, along with commonly used methods. In each case, a tokenization function is created that adjusts for the note type’s unique attributes (e.g. apart from note title and content on a task note type, the contents of the subtasks may also be read). All the endpoints that contain the logic are in the routes() function of each class.

2.2.4.2 Simplifying validation schemas

Object-Oriented Programming can also be uset so implify validation schemas by creating a base schema class. This class acts like a blueprint, defining common fields such as title, content, tags and their validation rules . Other schema classes inherit from the base class, just like in the processing part of the backend, reusing the common field definitions and validations. This reduces code duplication and makes it easier to maintain the validation rules. If a schema needs additional fields, it can simply add them to its own class definition. Based on the designed database and encapsulating any repeating fields or datastructures, here is an example of how such approach may be used to create a validation schema for an incoming request to validate and store a new Task note:



It is important to note that in this implementation the datatypes on attributes are the valid datatypes that are checked for in incoming data. In the Technical Solution, this will be expanded on further and the actual datatypes of the properties may turn out different due to the logic of the chosen validation library.

2.3 Frontend

2.3.1. User data on the frontend

Now that we've established how the data will be stored and processed on the backend, it's time to delve deeper into how will it be collected and displayed to users.

Frontend applications often employ a state management system to efficiently handle and update data displayed to the user. This may include information about the currently selected note, a list of notes, loading status, and any validation errors.

The data flow involves these steps:

a) **Fetching Data:** When the application starts or when the user reloads the page to view notes, the application fetches an initial set of note previews from a backend API based on the user's pagination settings and the size of the display. Previews will have the note title and contents cut to speed up loading times and avoid loading long notes. This data is then stored in the application's state.

b) **User Interaction:** As the user interacts with the application (e.g., creating, editing, or deleting notes), the application updates its internal state optimistically to reflect these changes. For example, when a user starts creating a new note, a new note object is created and added to the state and a request is sent to the backend to process and store the submitted data or reflect any changes, following one of the flowcharts in **2.1.2**.

c) **Data Display:** The application uses the updated state to render the user interface. For example, the list of notes is dynamically updated as new notes are created or existing ones are modified, all while the request may not have even reached the backend. Any changes are reverted if the request fails and an error message is displayed

This general data flow ensures that the user interface always reflects the latest state of the application's data, providing a seamless and responsive user experience.

2.3.2. Frontend file structure

Since the frontend of the application will be built with the React framework using JavaScript and TypeScript, the file structure will be predetermined by web application generation tools, such as Vite, unlike on the backend. Vite and similar tools provide a faster development experience for modern web projects. Running a simple command would create the basic components of a web app, such as index.html which is the default file that web servers look for on a website, along with installing NodeJS modules, creating a Git repository and providing the developer with a few sample components to start development. By some convention and from my experience, I arrived to the following file structure of the web app:

- node_modules
- components
- public
- src
    - api
    - types
        - taskTypes.ts
    - tasksApi.ts
    - Pages
        - Tasks
            - Components
            - Tasks.tsx
            - useTasks.ts
        - Account

        - Dashboard
    - index.css
    - routes.tsx
    - main.tsx
    - etc
- index.html
- .gitignore
- other configuration files

In the root directory of my app, I will have configuration files, such as package-lock.json, which will have the NodeJs libraries used in the project written down or .gitignore, which cofigures the folders that will be tracked by my source control scripts. One such folder is node_modules, which contains the actual NodeJs libraries and their files. These files are to be configured once and rarely touched during the development process.

The first folder necessarily involved with development of the app would be components, which would contain any reusable components such as buttons, cards or layout components. Public conventionally contains fonts, any images used in the website, documents or icons, so the user's web browser would know in advance about the media it would have to load.

The most important folder of the project is src, containing files such as index.css, which would store the color codes for the main theme of the website, along with the router in routes.tsx to predetermine the routes the users would be able to follow on the website. In addition, main.tsx will act as the 'brain' of the application, where the most important tools such as contexts and providers will be imported and the core of the React app is set up. This will also be the file which renders the website on the webpage. The Pages folder will store the page components of the website, such as the Accounts page along with any components uniquely used in this page. Any logic involving interaction with accounts will be stored in a hook named useAccounts, which may involve api calls or changes to the state of the application. This hook can be called in other componetns or pages to retrieve some information from the Accounts state.

Finally, the api folder will contain files involved with moving data between frontend and backend. It will contain interfaces and schemas for type safety and validation, as well as CRUD (create, read, update, delete) methods on predetermined backend endpoints.

2.3.3.

2.3.4.

2.4 User Interface

2.4.1. The 10 principles of good design

Design is more than aesthetics—it's a philosophy that governs how users interact with and perceive a product. Dieter Rams' 10 principles of good design, though originally conceived for physical products, are still used in the web design industry. Here are the 10 principles that will be used during the design of the user interface of the app:

- **Good design is innovative**
- **Good design makes a product useful**
- **Good design is aesthetic**
- **Good design makes a product understandable**
- **Good design is unobtrusive**
- **Good design is honest**
- **Good design is long-lasting**
- **Good design is thorough down to the last detail**
- **Good design is environmentally friendly**
- **Good design is as little design as possible**

Even though not all of these principles can apply to digital designs or this specific scenario, they will still be referred to throughout the design process.

2.4.2. Design as an iterative process

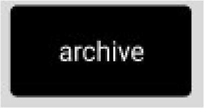
Design should be an iterative process because it allows for continuous refinement and improvement, ensuring that the final product meets user needs effectively. This approach helps developers to not get tunnel visioned and obsess over their own design, but rather take advice from the end users of the product, keeping the final design as little as possible. Furthermore, iterative design is flexible, adapting to changing requirements or technologies, and ensures that the end product remains long-lasting, understandable and thorough.

2.4.3. Creating wireframes

Wireframes are an excellent first step in designing a website because they provide a clear, simplified visual blueprint of the layout and structure before investing time in detailed design or coding. By focusing on functionality and user flow, wireframes allow designers to map out where key elements like navigation, content, and receive user feedback, allowing the design of the website to quickly change without editing the code and being distracted by colors or typography. Wireframes save time and resources by laying a solid foundation for further development.

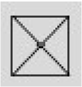
Before beginning with the wireframes, let’s establish a simple code which will be used to denote various elements on the webpage, in accordance with industry standards:

Button



A button is denoted by a black box with or without any text inside

Icon



Icons or images are crossed wireframe boxes

Legend

The left of each wireframe is designated to the legend of the wireframe, which contains descriptions of primary components of the wireframe.

On the other hand, the right side of the screen may be used for additional notes, such as explaining the usage of buttons which were too small to have text inside.

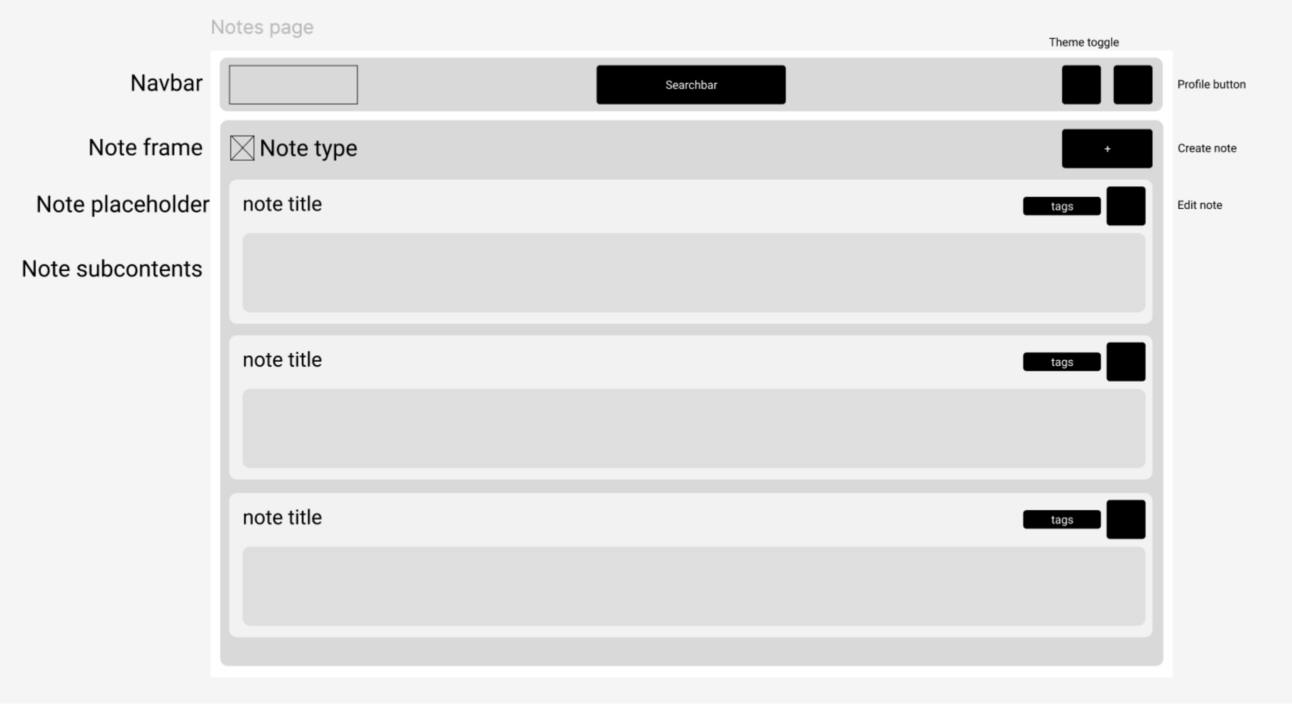
Cards

Each different card or element which is not a button or an icon is colored in a different shade of gray to be easier to distinguish from other elements of the design.

Having considered the objectives in Analysis, I came to the following pages and elements that need to be sketched out first:

2.4.3.1 A generic note viewing and editing page, which supports organization by tagging, additional form fields and timestamps.

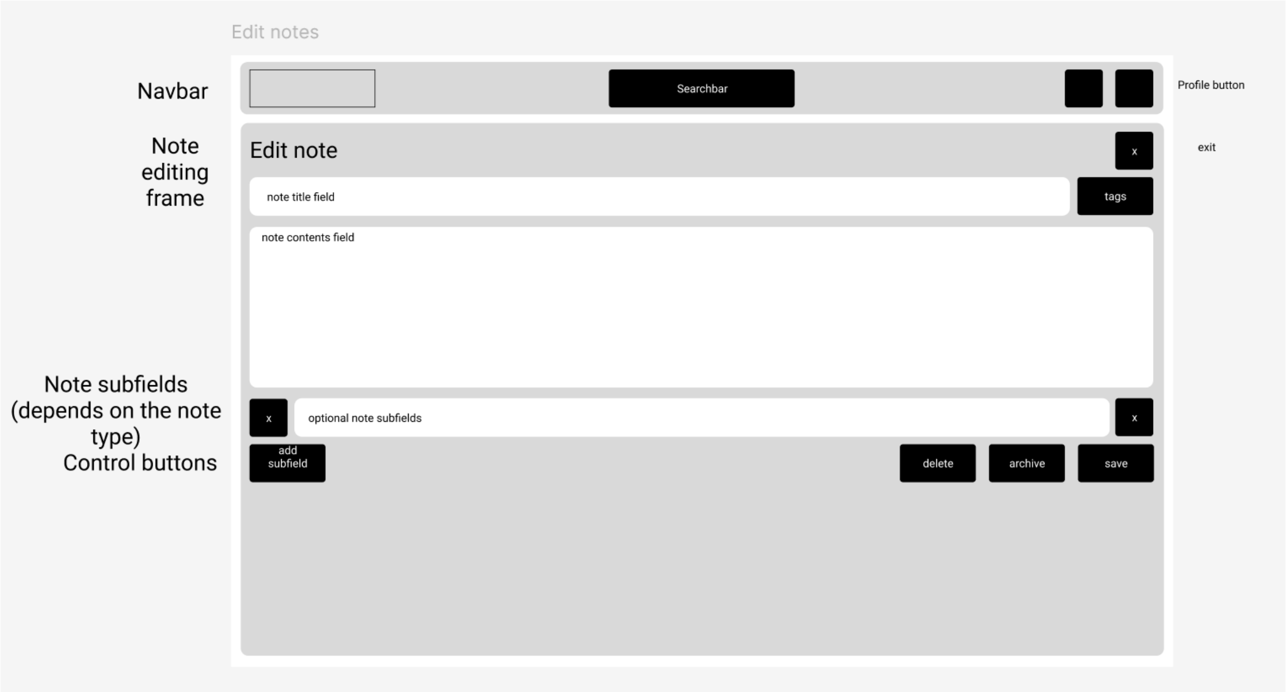
2.4.3.1.1 Viewing all notes



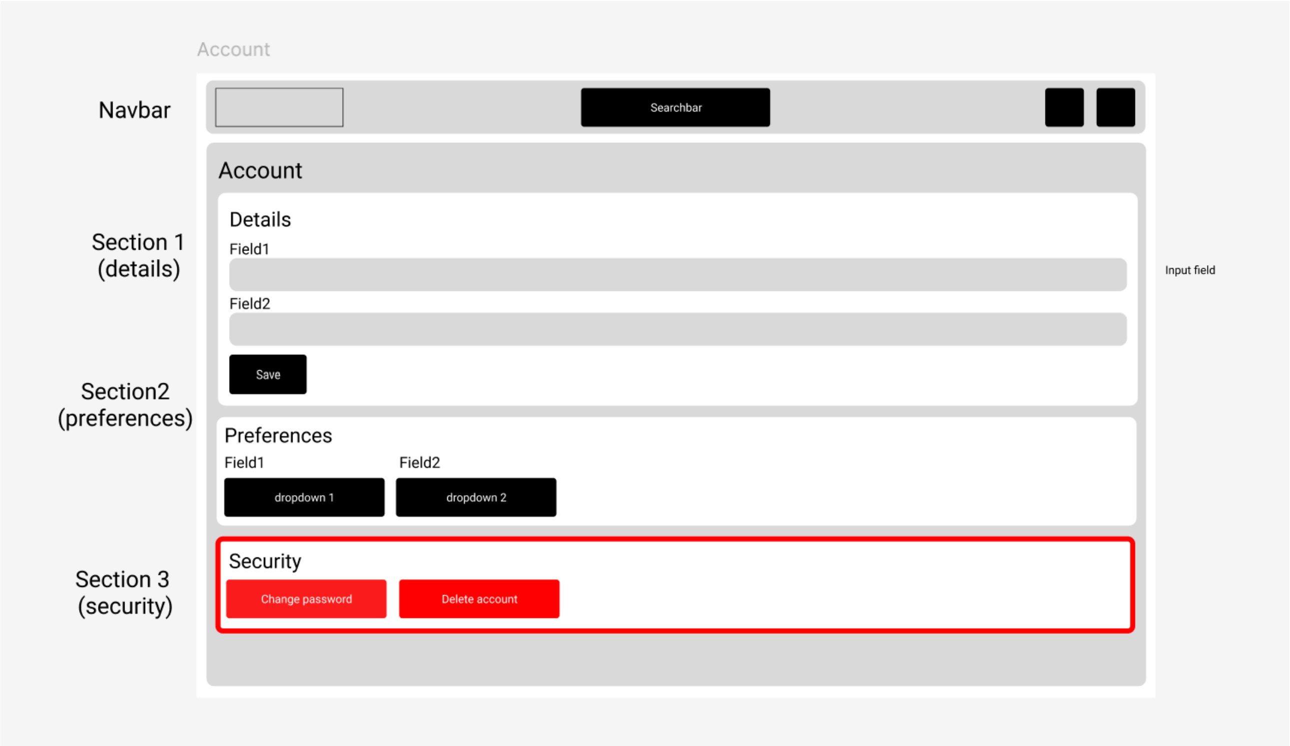
This wireframe represents the screen which will be containing all of user's notes of a specific type, along with pagination controls, ability to create new notes, view their previews (such as a cropped title and content, if they exceed a certain character limit) and open each note in a separate screen for editing or viewing.

2.4.3.1.2 Editing a note

In the note editing page, the user will be able to alter the contents of each note, as long as organize tags, delete or archive it

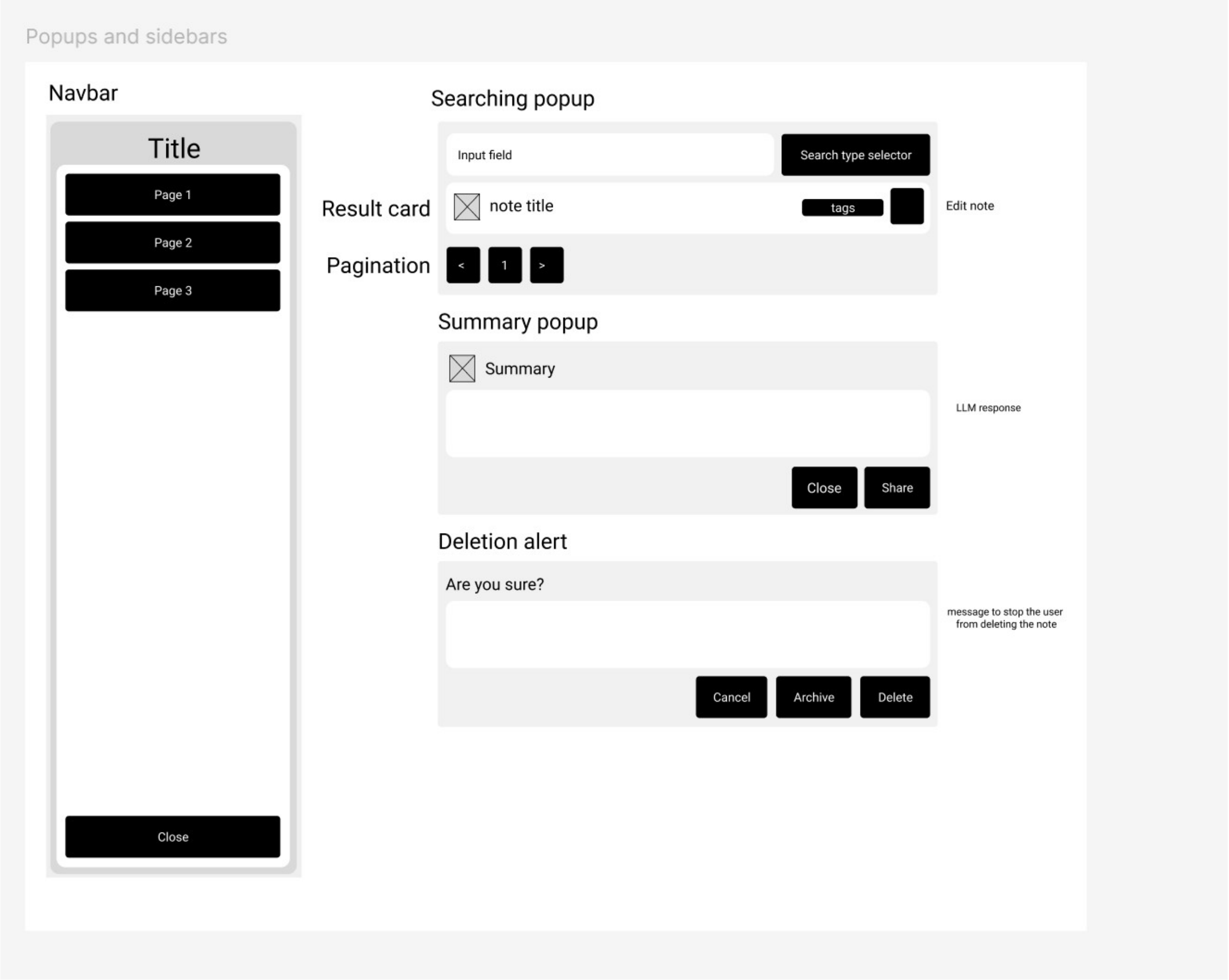


2.4.3.2 An account page where the user can edit their details



The account page will be divided into multiple sections, such as a form section (Section 1), where the user can edit text-based details such as their display name or email, Section 2 with preferences, such as font sizes or theme and finally a security section, which is outlined in red to warn users that it contains buttons such as changing the password.

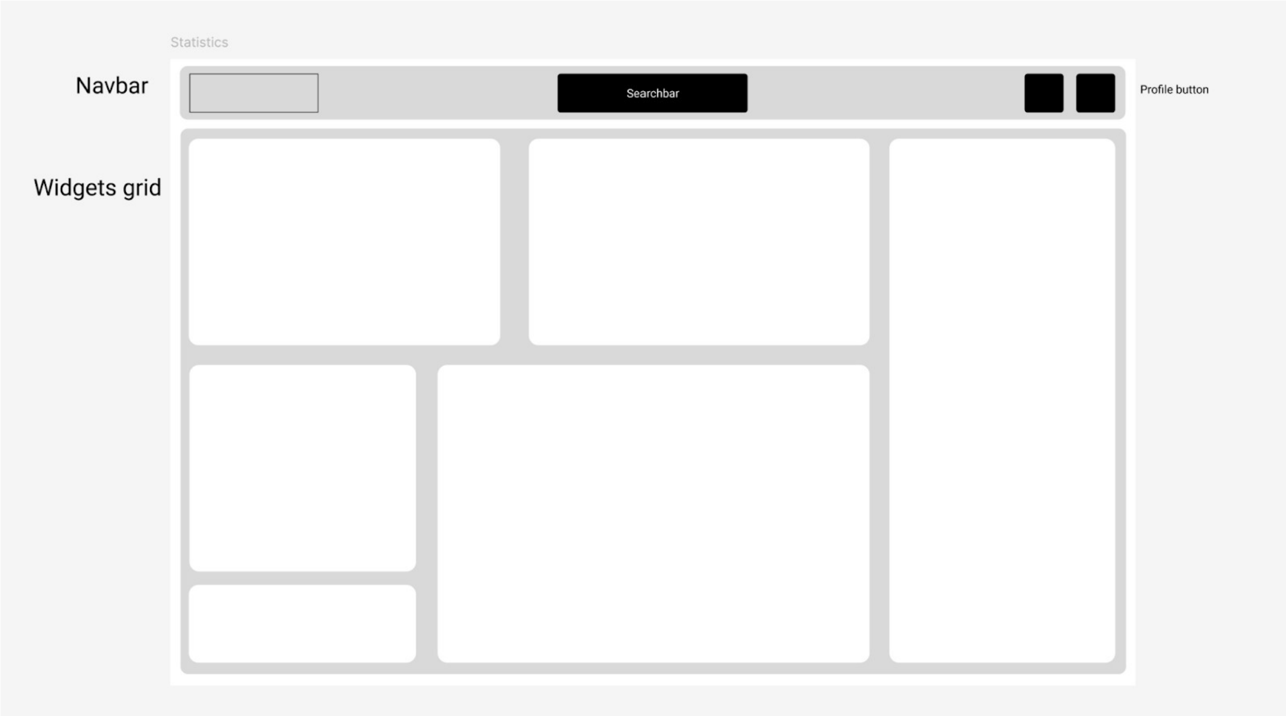
2.4.3.3 A sidebar and other popups, such as alerts or search widgets



This wireframe contains reusable components, such as alerts, a searching popup with an ability to search notes, select types and edit them and a navbar, which can be used by the user at any time to navigate through the website.

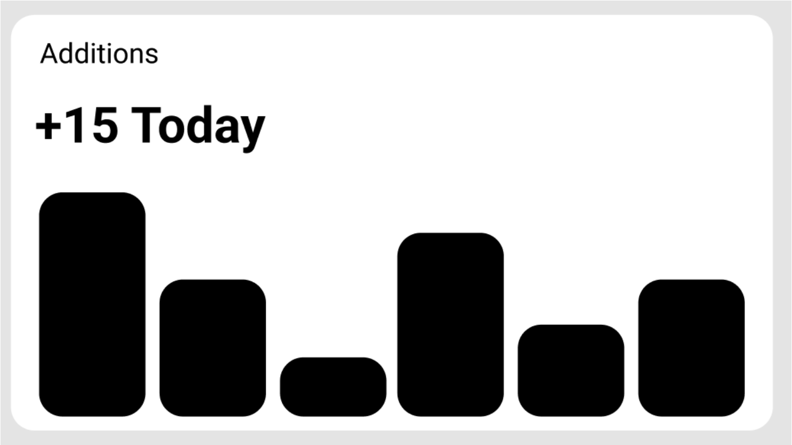
2.4.3.4 A note statistics page

The requirement for accountability and statistics can be further expanded into a dashboard, which will contain widgets with statistics and other information. The general look for such a page should be as follows:



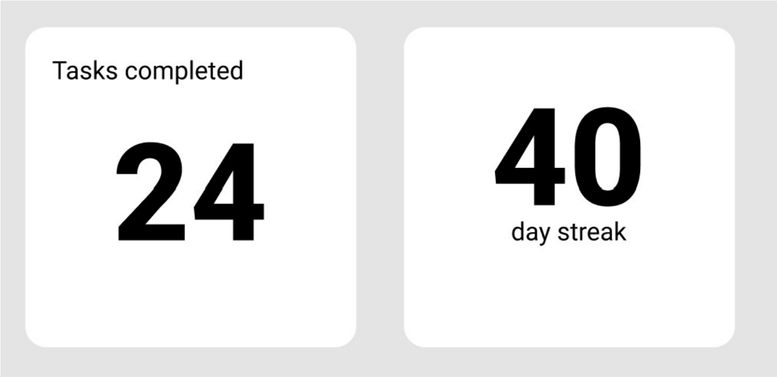
The page can be a grid with customizable widgets of various sizes. There are plenty of widget options, but most of them can be simplified to several types:

2.4.3.4.1 Charts



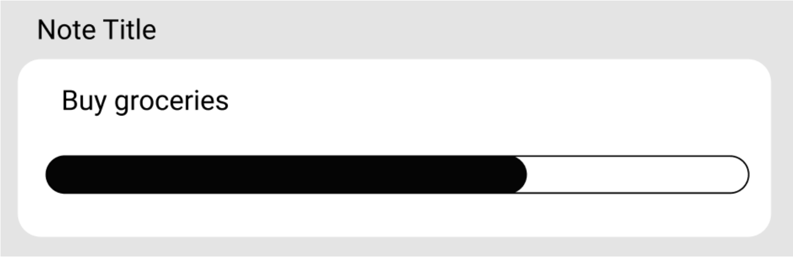
Charts can take the form of bar or graph charts and may be used to represent data such as task completion, activity, streaks etc. Depending on the size of the widget, several key features such as the scale may be dynamically added as the user scales the widget.

2.4.3.4.2 Numbers

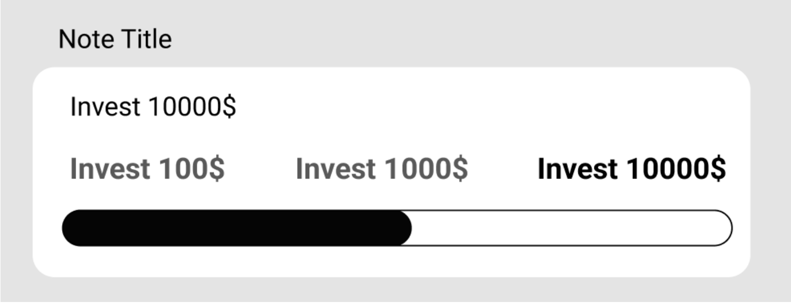


The smallest possible gridcell may be used to store a simple numerical value, such as the numbers provided above. In the examples, the widgets represent the total of tasks completed for a user and the length of their streak of using the app or a specific habit

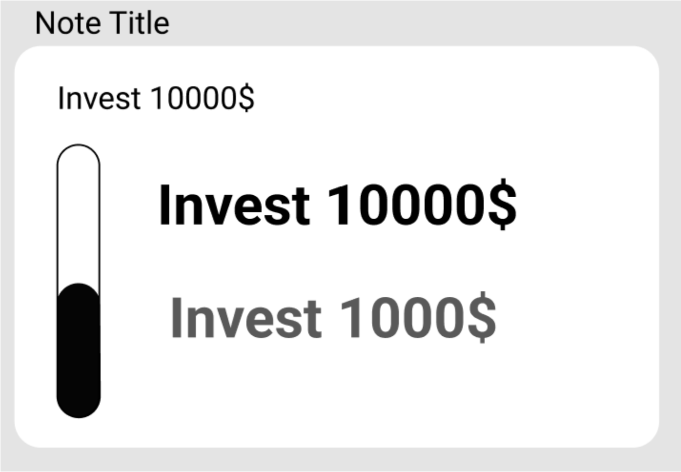
2.4.3.4.3 Progress Bars



Such a simple thing as a progress bar may motivate users to continue with their commitments and provide gratification when they see their progress move after successfully completing a task.



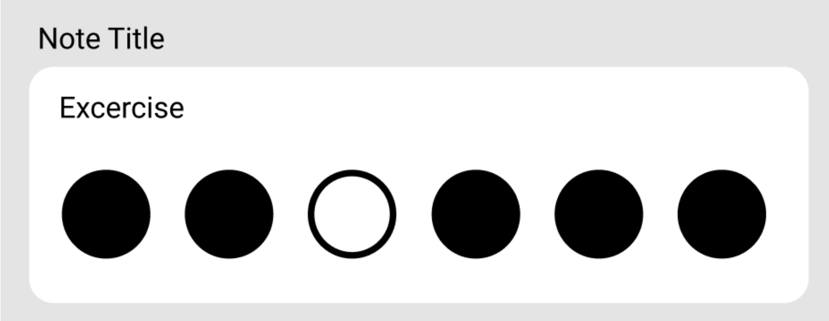
This idea may be expanded with the addition of milestones for goals, which may further motivate the user. Additionally, this example maybe used to highlight the ease of adapting these components for different sizes by stacking the milestones vertically:



It is also important to remember that these widgets are drafts/wireframes and will look differently after being implemented in code.

2.4.3.4.4 Streaks

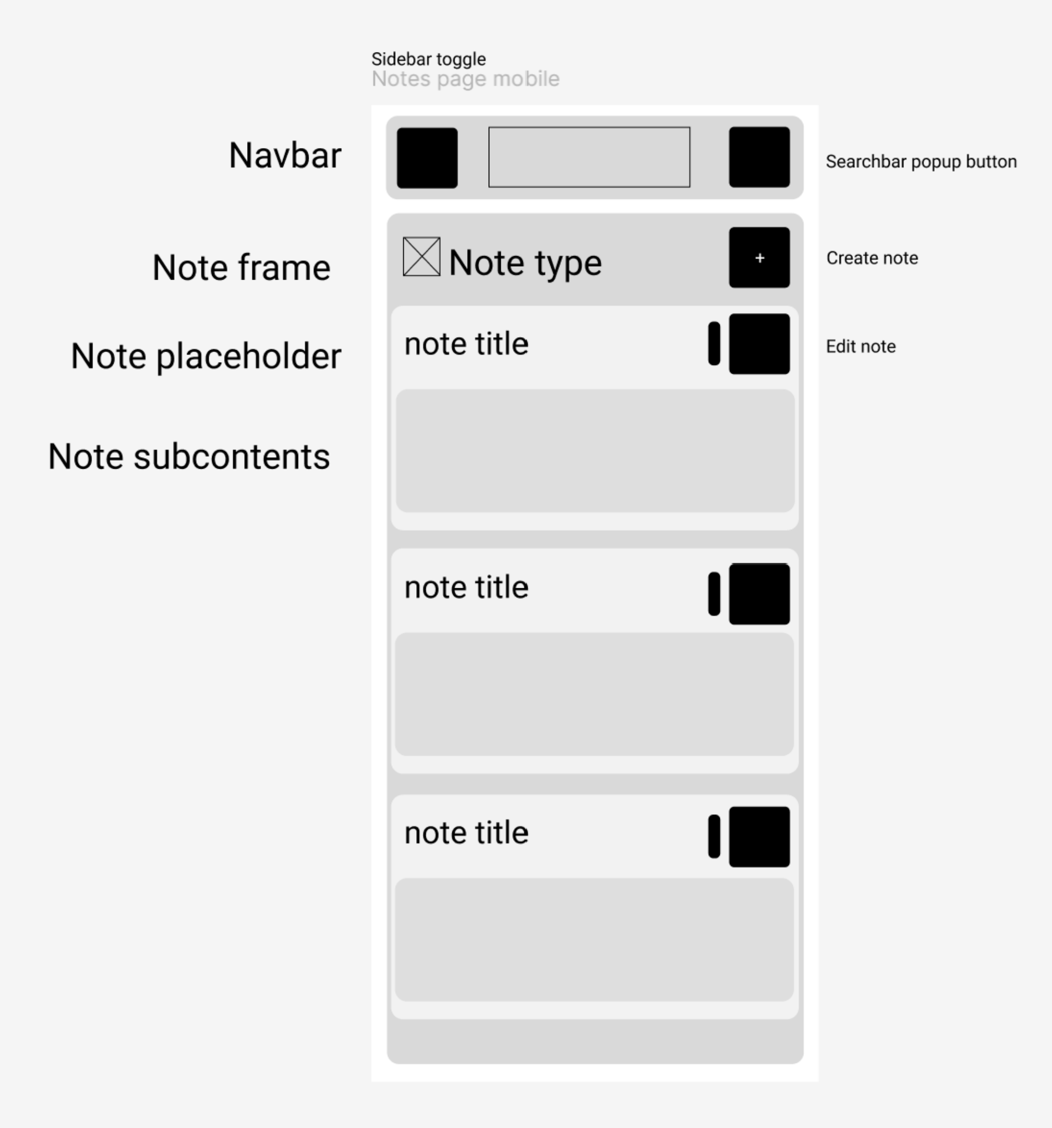
The following type can only be used to very specific notes that require daily commitment, allowing the user to visualise their progress in a simple way.

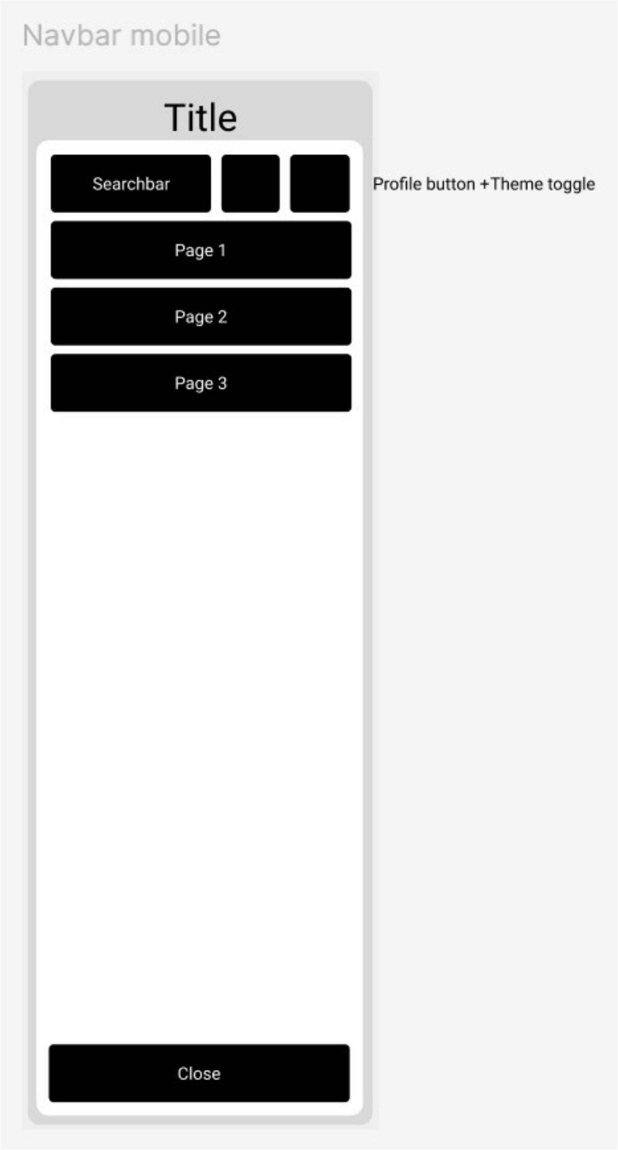


2.4.3.4.5 Mobile design

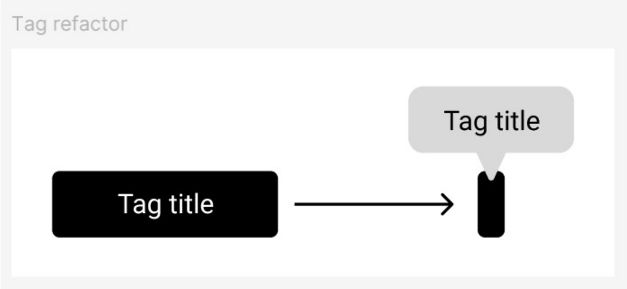
In addition, there should be design options for devices of different display sizes, to ensure that the products is long-lasting and thorough. Even though the app should be able to auto-adapt to various display sizes, some components must be moved around or replaced to accommodate for smaller screens.

Several elements of the interface have to be rearranged in order to keep everything readable. The biggest change was done to the navbar, where the theme toggle and account buttons have been moved to the sidebar to make more space for the logo on the narrower navbar.


Sidebar toggle
Notes page mobile
Navbar
Searchbar popup button
Note frame
Note type
+
Create note
Note placeholder
note title
Edit note
Note subcontents
note title
note title

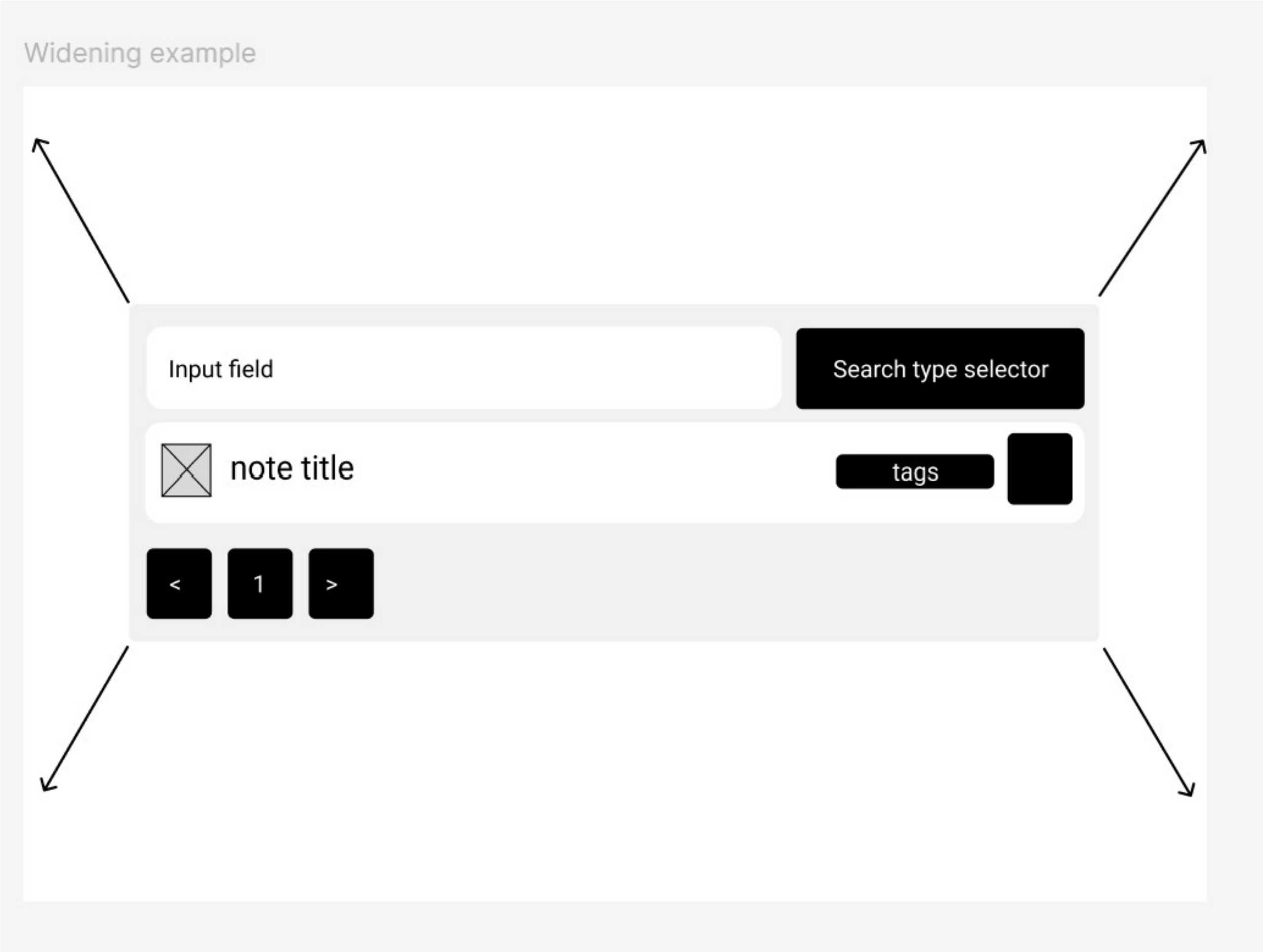


A similar refactoring happened to the note cards themselves. For example, there are less characters displayed per line as the screen shrinks to make the note cards grow in height instead of width. Moreover, additional whitespace can be found by shrinking the tag labels and just leaving small colored badges, with the actual tag titles being accessible only on click or hover:



Other components such as widgets, editing screens or other pages without moving components or popups can be dynamically resized by the software or rearranged using adjustable grids for various display sizes

Finally, some components, that were previously used as popups on wider screens, such as search popups or alerts now can take the full width of the device



2.4.4. Choosing the color and fonts

2.4.4.1 Core color palette

Foreground, background, primary, and other colors are essential components of any visual design, serving crucial roles in creating a functional, visually appealing, and accessible user experience.

Foreground typically represents the text or content that users need to read or interact with. It's crucial for it to have a high contrast with the background color, making the text easily legible and reducing eye strain. Meanwhile, background sets the overall tone and provides a visual foundation for the design. The most prominent color in the design is called the primary color, often used for buttons, calls to action, and brand elements. It helps guide the user's attention to important areas of the interface.

There are a few optional colors that may be included in the color palette of a design. These colors are used to add visual interest and distinguish different elements within the design. They might include secondary colors for less prominent elements, accent colors for highlighting specific features, and neutral colors for borders or subtle background elements.

Since the users will have an ability to edit the colors of the tags that they create, I opted for a simple black and white color theme for light mode, and dark blue-light gray for the dark mode, to avoid the colors of the design from interfering with user's note tag colors. In addition to that, the colors were chosen based on the following guidelines:

- **Contrast:** The color choices ensure contrast between foreground and background elements for readability and accessibility by all users.
- **Accessibility:** The color palette adheres to accessibility guidelines, aiming to meet WCAG contrast ratios to ensure sufficient color differentiation for users with visual impairments.
- **Aesthetics:** The colors were selected to create a visually appealing and cohesive aesthetic. The colors allow for a clean and relaxed look, without any attention-grabbing hues that would interfere with user's focus.
- **User Experience:** The inclusion of a dark mode option provides users with greater flexibility to customize the interface to their preferences or in dark environments, which would improve their overall experience.

The following colors will be provided using the HSL color model (Hue, Saturation, Lightness), which would allow me to intuitively tweak the color palette, ensuring harmony and flexubility during the design process.

Background:

- Light Mode: 0 0% 100% (white)
- Dark Mode: 222.2 84% 4.9% (dark gray)

Foreground:

- Light Mode: 222.2 84% 4.9% (dark gray)
- Dark Mode: 210 40% 98% (light gray)

Primary: 222.2 47.4% 11.2% (a dark blue/purple hue)

Secondary:

- Light Mode: 210 40% 96.1% (light gray)
- Dark Mode: 217.2 32.6% 17.5% (darker gray)

Muted:

- Light Mode: 210 40% 96.1% (light gray)
- Dark Mode: 217.2 32.6% 17.5% (darker gray)

Accent:

- Light Mode: 210 40% 96.1% (light gray)
- Dark Mode: 217.2 32.6% 17.5% (darker gray)

Destructive: (warning)

- Light Mode: 0 84.2% 60.2% (red)
- Dark Mode: 0 62.8% 50.6% (red)

2.4.4.2 Typeface: Open Sans

2.4.4.2.1 Introduction

Sans-serif fonts, which is the category that Open Sans is part of, is characterized by their clean lines and lack of decorative flourishes (serifs), are commonly used in web development for several key reasons:

To begin with, sans fonts easier to read quickly due to their simplicity and reduces eye strain, crucial for comfortable digital experiences. Simplicity is often associated with technology, innovation, and minimalist design, which would align well with the preferences of the target user base (students and professionals that may not have time to decypher a complex font). Finally, these fonts can be used effectively for headlines, body text, and interface elements, ensuring consistency throughout the design while adhering to modern web accessibility standards.

2.4.4.2.2 Character set support

Font's character set support is a crucial consideration for a project where users are allowed to enter their own information in their language. The chosen font must encompass a broad range of characters, including:

- Support for all Latin characters, including accented characters and diacritics
- Support for common special characters such as punctuation, symbols, and mathematical symbols.
- Comprehensive Unicode support to ensure accurate rendering across different platforms and operating systems and allowing users to write non-latin characters.

Luckily, Open Sans supports 586 languages at the time of writing and alphabets, such as Greek, Cyrillic, Latin, Hebrew with many math operators and punctuation signs. As a result, it is safe to say that most is not all of the characters that users write will be supported.

2.4.4.2.3 Licensing

Open Sans is an open-source font, licensed under the OFL (Open Font License), so I can use use it for free for both personal and commercial projects, completely eliminating any licensing costs.

2.4.4.2.4 Performance

One more consideration when picking a font for a project may be performance, as complex fonts may take a longer time to render, resulting in a worse user experience. On most modern computers this change in performance may be negligible, but this decision may be impactful on mobile phones or old computers.

2.4.4.2.5 A

3 Technical solution

3.1 Backend

3.1.1. app.py

backend/app.py: This file is the main entry point for the backend application, built using the Flask framework in Python. It sets up the core application instance, configures middleware, initializes database connections, loads models, and defines the application's routes. The file starts by importing necessary modules and classes.

```
app = Flask(__name__)
app.config.from_object(Config)
```

This creates the Flask application instance and loads configuration settings from the `Config` class.

```
CORS(app, resources={r"/api/*": {"origins": ["http://localhost:5173"]}} ,
supports_credentials=True)
```

This configures CORS to allow requests from `http://localhost:5173`, which is the default port for the React frontend during development. The `supports_credentials=True` allows sending cookies and authorization headers.

```
jwt = JWTManager(app)

 conn = psycopg2.connect(
            host=Config.host,
            database=Config.database,
            user=Config.user,
            password=Config.password,
            port=Config.port
          )
```

This initializes the JWT manager for handling authentication and establishes a connection to the PostgreSQL database using credentials from the configuration file

```
model = load_or_train_model()
tokenization_manager = TokenizationTaskManager(Config,model)
recents_manager = RecentNotesManager()
```

This section loads the vectorization model and initializes the `TokenizationTaskManager` and `RecentNotesManager`, modules that will encapsulate logic, concerning note vectorization and recent notes manaers.

```
notes = NoteApi(app, conn, tokenization_manager, recents_manager)
tasks = TaskApi(app, conn, tokenization_manager, recents_manager)
habits = HabitApi(app, conn, tokenization_manager, recents_manager)
goals = GoalApi(app,  conn, tokenization_manager, recents_manager)

tag_routes(app, conn, tokenization_manager)
user_routes(app, conn)
archive_routes(app, conn, tokenization_manager)
universal_routes(app, conn, model, recents_manager)
```

API endpoints for different resources (notes, tasks, habits, goals, tags, users, archive, and universal routes) are initialised. It passes the Flask app instance, the database connection, and other managers to the route handlers.

```
@app.errorhandler(Exception)
def handle_exception(error):
    return jsonify({'message': 'An error occurred', 'details': str(error)}), 500

@app.errorhandler(429)
def ratelimit_handler(e):
     return jsonify({'message': 'An error occurred', 'details' : "Rate limit exceeded"}), 429
```

These define global error handlers for catching exceptions and rate limit errors, providing consistent JSON error responses.

```
if __name__ == '__main__':
     app.run(debug=True, port=5000)
```

This starts the Flask development server when the script is executed directly. The `debug=True` option enables debugging features, and the development server runs on port 5000.

3.1.1. modules/universal.py

```
from datetime import datetime
import os
import sys

import psycopg2
from flask import g, jsonify, request
from flask_jwt_extended import jwt_required

sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), os.path.pardir)))
```

```
from flask import g, jsonify, request
from flask_jwt_extended import jwt_required

from formsValidation import GeminiSummarySchema
from utils.utils import process_universal_notes, token_required
from utils.word2vec import combine_strings_to_vector
```

This section imports necessary libraries. `datetime` handles dates and times. `os` and `sys` are for operating system interactions (file paths, etc.). `psycopg2` is the PostgreSQL database adapter. `flask`, `jsonify`, and `request` are from the Flask framework for building web APIs. `flask_jwt_extended` manages JWT authentication. The `sys.path` modification adds a parent directory to the Python path. `process_universal_notes` and `token_required` are custom utility functions, that will be talked about later

3.1.1.1 BaseNote class

```
class BaseNote:
    def __init__(self, app, conn, tokenization_manager, recents_manager):
        self.app = app
        self.conn = conn
        self.tokenization_manager = tokenization_manager
        self.recents_manager = recents_manager

        self.tags_cte = """TagsCTE AS (
                    SELECT
                        nt.note_id,
                        json_agg(json_build_object(
                            'tagid', tg.id,
                            'name', tg.name,
                            'color', tg.color
                        )) AS tags
                    FROM NoteTags nt
                    JOIN Tags tg ON nt.tag_id = tg.id
                    GROUP BY nt.note_id
                )"""

    def create_note(self, cur, userId, title, content, noteType, tags):
        cur.execute( # Insert into Notes table
            "INSERT INTO Notes (user_id, title, content, type) VALUES (%s, %s, %s,
%s) RETURNING id",
            (userId, title, content, noteType)
        )
        noteId = str(cur.fetchone()[0])

        self.update_notetags(cur, noteId,tags, withDelete=False)

        return noteId
    def fetch_total_notes(self, cur,note_type, userId, page,offset,per_page):
        cur.execute("""
                        SELECT COUNT(DISTINCT n.id) AS total
                        FROM Notes n
                        WHERE n.user_id = %s AND n.type = %s AND n.archived = FALSE
                    """, (userId,note_type,))
        total = cur.fetchone()['total']
        nextPage = page + 1 if (offset + per_page) < total else None
        return total, nextPage

    def update_notetags(self, cur, note_id, tags, withDelete=True):

        if withDelete:
            cur.execute("DELETE FROM NoteTags WHERE note_id = %s", (note_id,))
        if tags:
            tag_tuples = [(note_id, str(tagId)) for tagId in tags]
            cur.executemany(
                """
                INSERT INTO NoteTags (note_id, tag_id)
                VALUES (%s, %s)
                """,
                tag_tuples
            )
```

The `BaseNote` class provides core functionality for managing notes, that will be incorporated into other note types in the application. The `__init__` method initializes the class with the Flask app, a database connection (`conn`), a `tokenization_manager`, and a `recents_manager`. `tags_cte` defines a Common Table Expression (CTE) for efficiently retrieving note tags from the database, since the tag functionality is the same for all note types. `create_note` inserts a new note into the `Notes` table and then calls `update_notetags`. `fetch_total_notes` retrieves the total number of notes for a given user and note type, used for pagination. `update_notetags` updates the `NoteTags` table, associating tags with a note. It can either delete existing tags for the note before adding new ones or just add new tags.

3.1.1.2 Universal note functionality

```
def universal_routes(app, conn, model, recents_manager,genai):
    @app.route('/api/search', methods=['GET'])
    @jwt_required()
    @token_required
    def search():
      # ...


    @app.route('/api/recents', methods=['GET'])
    @jwt_required()
    @token_required
    def get_recents():
      # ...


    @app.route('/api/calendar', methods=['GET'])
    @jwt_required()
    @token_required
    def get_calendar_notes():
       # ...


    @app.route('/api/summarize', methods=['POST'])
    @jwt_required()
    @token_required
    def summarize():
        # ...
```

The `universal_routes` function sets up the API endpoints using Flask. It takes the Flask app instance, the database connection, a word2vec model (for note vectorization), a `recents_manager`, and a `genai` object (part of Google's AI libraries). It defines four routes: `/api/search`, `/api/recents`, `/api/calendar`, and `/api/summarize`. Each route is decorated with `@app.route`, specifying the URL and HTTP method. `@jwt_required()` and `@token_required` ensure that the user is authenticated before accessing these routes. Here is how each function works:

3.1.1.3 Searching

```
    @app.route('/api/search', methods=['GET'])
    @jwt_required()
    @token_required
    def search():
        try:
            user_id = g.userId
            cur = conn.cursor(cursor_factory=psycopg2.extras.DictCursor)
            search_query = request.args.get('searchQuery')
            search_mode = request.args.get('searchMode')
            page = int(request.args.get('pageParam', 1))
            per_page = int(request.args.get('perPage', 5))
            offset = (page - 1) * per_page

            if search_mode == "approximate":
                query_vector = combine_strings_to_vector(search_query.split(),
model, False)

                cur.execute("""
                    SELECT n.id AS note_id, n.title, n.content, n.type,
                    t.id AS tagid,
                    t.name,
                    t.color,
                    cosine_similarity(n.vector, %s) as similarity
                    FROM Notes n
                    LEFT JOIN NoteTags nt ON n.id = nt.note_id
                    LEFT JOIN Tags t ON nt.tag_id = t.id
                    WHERE n.user_id = %s AND n.archived = FALSE AND n.vector IS NOT
NULL
                    ORDER BY similarity DESC
                    LIMIT %s OFFSET %s;
                """, (query_vector, user_id, per_page, offset))

            else:
                cur.execute("""
                    SELECT n.id AS note_id, n.title, n.content, n.type,
                        t.id AS tagid,
                        t.name,
                        t.color
                    FROM Notes n
                        LEFT JOIN NoteTags nt ON n.id = nt.note_id
                        LEFT JOIN Tags t ON nt.tag_id = t.id
                    WHERE n.user_id = %s AND n.archived = FALSE
                        AND (n.title ILIKE %s OR n.content ILIKE %s)
                    LIMIT %s OFFSET %s;
                """, (user_id, f"%{search_query}%", f"%{search_query}%", per_page,
offset))

            rows = cur.fetchall()

            # Get the total number of results for pagination
            if search_mode == "exact":
                cur.execute("""
                    SELECT COUNT(*) FROM Notes n
                    WHERE n.user_id = %s AND n.archived = FALSE AND n.vector IS NOT
NULL;
                """, (user_id,))
            else:
                cur.execute("""
                    SELECT COUNT(*) FROM Notes n
            WHERE n.user_id = %s AND n.archived = FALSE
                        AND (n.title ILIKE %s OR n.content ILIKE %s);
                """, (user_id, f"%{search_query}%", f"%{search_query}%"))

            total = cur.fetchone()[0]
            next_page = page + 1 if offset + per_page < total else None

            if rows:
                notes=process_universal_notes(rows,cur)

            return jsonify({'message': 'Notes retrieved successfully', 'data':
notes,
                            'pagination': {
                                    'total': total,
                                    'page': page,
                                    'perPage': per_page,
                                    'nextPage': next_page
                                }
                            }), 200
        except Exception as e:
            conn.rollback()
            print(f"An error occurred: {e}")
            return jsonify({'message': 'An error occurred', 'error': str(e)}), 500
        finally:
            cur.close()
```

The /api/search route handles searching for notes. It retrieves the user ID from the JWT (g.userId), gets the search query and mode (approximate or regular) from the request arguments, and sets up pagination. If the search mode is "approximate," it uses a cosine similarity search against pre-calculated note vectors (more about it later). Otherwise, it performs a standard case-insensitive search using ILIKE on the note title and content. It fetches the results, sorts them, calculates the total number of results for pagination, and calls process_universal_notes to format the data. Finally, it returns the notes and pagination information as a JSON response.

3.1.1.4 Recents

```
@app.route('/api/recents', methods=['GET'])
@jwt_required()
@token_required
def get_recents():
    try:
        userId=g.userId
        noteIds = recents_manager.get_recent_notes_for_user(userId)
        cur = conn.cursor(cursor_factory=psycopg2.extras.DictCursor)

        notes = []
        for note_id in noteIds:
            cur.execute("""
                SELECT n.id AS note_id, n.title, n.content, n.type,
                    t.id AS tagid,
                    t.name,
                    t.color
                FROM Notes n
                    LEFT JOIN NoteTags nt ON n.id = nt.note_id
                    LEFT JOIN Tags t ON nt.tag_id = t.id
                WHERE n.user_id = %s AND n.id = %s;
            """, (userId, note_id))

            row = cur.fetchone()
            if row:
                note = process_universal_notes([row], cur)[0]
                notes.append(note)

        return jsonify({'message': 'Recent notes retrieved successfully',
'data': notes}), 200
    except Exception as e:
        conn.rollback()
        raise
```

The /api/recents route retrieves recently accessed notes. It gets the user ID, uses the recents_manager to fetch a list of recent note IDs, and then retrieves the full note details from the database for each ID. It works as an endpoint to retrieve recently accessed notes by the user from the recents_manager(more about it later), then format the notes and return them as a JSON response.

3.1.1.5 Calendar functionality

```
    @app.route('/api/calendar', methods=['GET'])
    @jwt_required()
    @token_required
    def get_calendar_notes():
        try:
            userId = g.userId
            start_date = request.args.get('startDate')
            end_date = request.args.get('endDate')

            # Convert start and end dates to datetime objects
            start_date = datetime.fromisoformat(str(start_date))
            end_date = datetime.fromisoformat(str(end_date))

            with conn.cursor(cursor_factory=psycopg2.extras.DictCursor) as cur:
                # Fetch notes from goals table
                cur.execute("""
                    SELECT n.id AS note_id, n.title, n.content, n.type, g.due_date
                    FROM Notes n
                    JOIN Goals g ON n.id = g.note_id
                    WHERE n.user_id = %s AND g.due_date BETWEEN %s AND %s
                """, (userId, start_date, end_date))
                goal_notes = cur.fetchall()

                # Fetch notes from tasks table
                cur.execute("""
                    SELECT n.id AS note_id, n.title, n.type, n.content, t.due_date
                    FROM Notes n
                    JOIN Tasks t ON n.id = t.note_id
                    WHERE n.user_id = %s AND t.due_date BETWEEN %s AND %s
                """, (userId, start_date, end_date))
                task_notes = cur.fetchall()

                # Combine results
                notes = []
                for row in goal_notes + task_notes:
                    note = {
                        'noteid': row['note_id'],
                        'title': row['title'][:50] + '...' if len(row['title']) >
50 else row['title'],
                        'content': row['content'],
                        'due_date': row['due_date'],
                        'type': row['type'],
                        'tags' : []
                    }
                    #Select tags
                    cur.execute("""
                        SELECT
                            json_agg(json_build_object(
                                'tagid', tg.id,
                                'name', tg.name,
                                'color', tg.color
                            )) AS tags
                        FROM NoteTags nt
                        JOIN Tags tg ON nt.tag_id = tg.id
                        WHERE nt.note_id = %s
                        GROUP BY nt.note_id
                    """, (row['note_id'],))
                    note['tags'] = cur.fetchone()
                    notes.append(note)

                return jsonify({'message': 'Notes retrieved successfully', 'data':
notes}), 200

        except Exception as e:
            conn.rollback()
            print(f"An error occurred: {e}")
            return jsonify({'message': 'An error occurred', 'error': str(e)}), 500
```

The /api/calendar route retrieves notes associated with goals and tasks that fall within a specified date range. It gets the start and end dates from the request arguments, converts them to datetime objects, and then queries the Goals and Tasks tables to retrieve the relevant notes. It combines the results, truncates long titles, and fetches tags for each note. The notes with their due dates are then returned as a JSON response, and will be displayed on the calenar component on the website.

3.1.1.6 Summarization

~~~
@app.route('/api/summarize', methods=['POST'])
@jwt_required()
@token_required
def summarize():
        try:

            gemini_schema = GeminiSummarySchema()
            data = gemini_schema.load(request.json)
            title  = data['title']
            selection =  data['selection']

            model = genai.GenerativeModel("gemini-2.0-flash")
            response = model.generate_content("""
                Please summarize the following extract from a note titled
              {title}, delimeted by three backticks:

                ```{selection}```

                Please include:

                    Key points: What are the most important things mentioned in
                     the extract?

                    Context: How does this extract relate to the note as a
                     whole?
~~~

```
                        Purpose: What is the purpose of this particular extract
                         within the note?

                    Keep the summary concise, clear and below 1000 characters.
                    """.format(title=title, selection=selection),
                    # stream=True,
                    generation_config = genai.types.GenerationConfig(
                        temperature=0.2,
                        top_p=0.95,
                        max_output_tokens=256,
                    ),

                    )
                return jsonify({'message': 'Summary generated successfully',
'data': response}), 200
            except Exception as e:
                raise
```

The Summarization function is used to summarize the contents of the note a user selects on the website. It first checks whether the data is of valid type and length through the Schema, then extracts the relevant information. After that, a model is loaded from Google's genai library. This model is a Large Language Model (LLM) called Gemini. The model is provided with a specific prompt, that was engineered to avoid prompt injection by telling the model that any text inside the backticks must be summarized. Moreover, extra instructions are given to ensure that summaries are generated with consistent formatting and the model knows maximum length of data it has to generate. Finally, the model is configured to match the task at hand with the following properties:

- **temperature=0.2**: Temperature controls the "randomness" of the generated text. A lower temperature (closer to 0) makes the output more deterministic and predictable, sticking closely to the most likely next words. It tends to produce more focused and conservative text.
- **top_p=0.95**: This is the nucleus sampling parameter. Nucleus sampling considers the most likely words whose cumulative probability exceeds top_p. In this case, it considers the words whose probabilities add up to 95%. A value of 0.95 means the model will consider a wider range of possible next tokens, but still constrain the selection to the most probable ones.
- **max_output_tokens=256**: This sets a hard limit on the number of tokens (roughly words or sub-words) that the model can generate. This prevents the model from running indefinitely and helps to control the length of the output. 256 tokens is a moderate length and likely corresponds to a few paragraphs of text.

When the response is generated, data returns to the user in form of a JSON object

3.1.2. modules/notes.py

Generally, most endpoints for different note types use similar logic, as each note has a create, update, edit and delete functionality. This is why the BaseNote class, and its predetermined functions to create a note in the Notes table, edit tags, etc. exists. Before we dive deeper into more complex note processing logic, I would like to first present the ‘simplest’ module, which is notes. By definition, this note type has no extra fields except for the usual title, contents and tags, to be used as a way to quickly jot down information and use advanced searching and summarization techniques that were briefly mentioned above.

```
from datetime import datetime
import os
import sys
from flask import Blueprint, g, jsonify, request
from flask_jwt_extended import jwt_required

from modules.universal import BaseNote
from utils.userDeleteGraph import delete_notes_with_backoff,
delete_user_data_with_backoff
sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__),
os.path.pardir)))
from formsValidation import BaseSchema
from utils.utils import token_required
import psycopg2

class NoteApi(BaseNote):
    def __init__(self, app, conn, tokenization_manager, recents_manager):
        super().__init__(app, conn, tokenization_manager, recents_manager)
        self.note_schema = BaseSchema()
        self.note_routes()

    def tokenize(self,noteId,title,content):
        text = [title, content]
        priority = sum(len(string) for string in text)
        self.tokenization_manager.add_note(
            text=text,
            priority=priority,
            note_id=noteId
            )
```

The NoteApi class inherits the methods and properties of BaseNote and initializes them. The Tokenize function will be used for text vectorization and contains the fields that exist in this note type and will be relevant for the tokenization function

To sanitize and validate all the inputs, I am using a validation library called Marshmallow. With this library, I create schemas that validate the type, length and other properties of the input data. All the schemas are stored in a separate file named formsValidation.py. I will not explicitly mention this file, just provide relevant validation schemas along each module. For example, for notes, the following validation schemais used, which is then inherited into other schemas, that build on the BaseNote class:

```
from marshmallow import Schema, fields, ValidationError
```

```
# Base Schema for common fields
class BaseSchema(Schema):
    title = fields.Str(required=True, validate=lambda s: 100 >= len(s) > 0)
    content = fields.Str(required=False, validate=lambda s: len(s) <= 1000)
    tags = fields.List(fields.UUID(), required=False)
    noteid = fields.UUID(required=False)
```

Additionally, note_routes method defines the following API endpoints. By convention, all modules that require a token are decorated by @token_required function, and each endpoint features error handling algorithms that display the error to the user, roll back any database transactions and safely exit the function:

3.1.2.1 Create Note

I/api/notes/create **(POST):** Creates a new note. It validates the request data using self.note_schema.load(), retrieves the user ID from the JWT, creates the note in the database using the create_note method (inherited from BaseNote), adds the note text to the tokenization manager, commits the transaction, and returns the new note's ID It uses the functions from BaseNote, as there are no extra properties in the Notes type:

```
        @self.app.route('/api/notes/create', methods=['POST'])
        @jwt_required()
        @token_required
        def create_note():
            try:
                userId = str(g.userId)
                data = self.note_schema.load(request.get_json())
                title = data['title']
                tags = data['tags']
                content = data['content']
                with self.conn.cursor(cursor_factory=psycopg2.extras.DictCursor) as
cur:
                    noteId = self.create_note(cur, userId, title, content, 'note',
tags)

                self.tokenize(noteId,title,content)
                self.conn.commit()

                return jsonify({
                    'message': 'Note created successfully',
                    'data': {
                        'noteid': noteId,
                    }
                }), 200
            except Exception as error:
                self.conn.rollback()
                print('Error during transaction', error)
                raise
            finally:
                cur.close()
```

3.1.2.2 Update Note

/api/notes/update **(PUT):** Updates an existing note. It validates the request data, retrieves the note ID, title, content, and tags, updates the note in the database, updates the note's tags, updates the tokenization manager, commits the transaction, and returns a success message.

```
        @self.app.route('/api/notes/update', methods=['PUT'])
        @jwt_required()
        @token_required
        def update_note():
            try:
                userId = str(g.userId)  # Convert userId to string
                note = self.note_schema.load(request.get_json())

                note_id = str(note['noteid'])
                title = note['title']
                content = note['content']
                tags = note.get('tags', [])

                with self.conn.cursor(cursor_factory=psycopg2.extras.DictCursor) as
cur:

                    cur.execute("""
                        UPDATE Notes
                        SET title = %s, content = %s
                        WHERE id = %s AND user_id = %s
                    """, (title, content, note_id, userId))

                    self.update_notetags(cur, note_id, tags)

                    self.tokenize(note_id, title, content)
                    self.conn.commit()

                    return jsonify({'message': 'Note updated successfully', 'data':
None}), 200
            except Exception as e:
                self.conn.rollback()
                print(f"An error occurred: {e}")
                raise
            finally:
                cur.close()
```

3.1.2.3 Delete Note

/api/notes/delete **(PUT):** Deletes a note. It retrieves the note ID from the request, and uses delete_notes_with_backoff(more about it later) to attempt deletion with retries, handles NoteTags table before Notes table. It also removes the note from the tokenization manager, if it is currently awaiting tokenization. It returns a success message or an error message if deletion fails after retries.

```
        @self.app.route('/api/notes/delete', methods=['PUT'])
        @jwt_required()
        @token_required
        def delete_note():
            try:
                userId = g.userId
                data = request.get_json()
                note_id = data['noteId']
                stack = [12,2] #NoteTags, Notes
                stack.reverse()
                if delete_notes_with_backoff(self.conn, note_id, stack):
                    self.tokenization_manager.delete_note_by_id(note_id)
                    return jsonify({'message': 'Note deleted successfully'}), 200
                else:
                    return jsonify({'message': 'Failed to delete note data after
multiple retries'}), 500
            except Exception as e:
                self.conn.rollback()
                raise
```

3.1.2.4 Get a specific note

/api/note **(GET):** Retrieves a single note by ID. It retrieves the user ID and note ID from the request, fetches the full note details along with its tags using self.tags_cte, and adds the note to the recents_manager (for tracking recently viewed notes). It returns the note details or a 404 error if the note is not found. It also uses the self.tags_cte to efficiently retrieve tags.

```
        @self.app.route('/api/note', methods=['GET'])
        @jwt_required()
        @token_required
        def get_note():
            try:
                userId = g.userId
                noteid = request.args.get('noteid')

                with self.conn.cursor(cursor_factory=psycopg2.extras.DictCursor) as
cur:

                    cur.execute(f"""
                        WITH {self.tags_cte}
                        SELECT
                            n.id AS note_id,
                            n.title AS note_title,
                            n.content AS note_content,
                            n.created_at AS note_created_at,
                            n.type AS note_type,
                            COALESCE(tags_cte.tags, '[]') AS tags
                        FROM Notes n
                        LEFT JOIN TagsCTE tags_cte ON n.id = tags_cte.note_id
                        WHERE n.user_id = %s AND n.id = %s AND n.type = 'note' AND
n.archived = FALSE
                    """, (userId, noteid))

                    row = cur.fetchone()
                    if not row:
                        return jsonify({'message': "Note not found"}), 404

                    note = {
                        'noteid': row['note_id'],
                        'title': row['note_title'],
                        'content': row['note_content'],
                        'tags': row['tags']
                    }
                    self.recents_manager.add_note_for_user(userId, noteid)
                    return jsonify({"note": note, 'message': "Note fetched
successfully"}), 200

            except Exception as e:
                self.conn.rollback()
                print(f"An error occurred: {e}")
                return jsonify({'message': 'An error occurred', 'error': str(e)}),
500

            finally:
                cur.close()
```

3.1.2.5 Get Note previews

/api/notes/previews **(GET):** Retrieves previews of notes, paginated. It retrieves the user ID, gets the page number and items per page from the request arguments, calculates the offset, retrieves the total number of notes for pagination, and then fetches note previews (truncated title and content) along with their tags. This is implemented to reduce the amount of data being transmitted when many notes are loaded, which will speed up loading time. It returns the previews and pagination information as a JSON response. The query uses a Tag CTE, self.tags_cte, defined in the BaseNote class.

```
        @self.app.route('/api/notes/previews', methods=['GET'])
        @jwt_required()
        @token_required
        def get_note_previews():
            try:
                userId = g.userId
                # Pagination
                page = int(request.args.get('pageParam', 1))  # Default to page 1
                per_page = int(request.args.get('per_page', 5))  # Default to 10
items per page
                offset = (page - 1) * per_page

                with self.conn.cursor(cursor_factory=psycopg2.extras.DictCursor) as
cur:
                    # Fetch the total count of notes for pagination metadata
                    total,nextPage = self.fetch_total_notes(cur, 'note', userId,
page, offset, per_page)

                    cur.execute(f"""
                        WITH {self.tags_cte}
                        SELECT
                            n.id AS note_id,
                            n.title AS title,
                            n.content AS content,
                            n.type AS type,
                            COALESCE(tags_cte.tags, '[]') AS tags
                        FROM Notes n
                        LEFT JOIN TagsCTE tags_cte ON n.id = tags_cte.note_id
                        WHERE n.user_id = %s AND n.type = 'note' AND n.archived =
FALSE

                        ORDER BY n.updated_at DESC
                        LIMIT %s OFFSET %s
                    """, (userId, per_page, offset))

                    rows = cur.fetchall()
                    notes = []
                    for row in rows:
                        note = {
                            'noteid': row['note_id'],
                            'title': row['title'][:100] + '...' if
len(row['title']) > 100 else row['title'],
                            'content': row['content'][:200] + '...' if
len(row['content']) > 200 else row['content'],
                            'tags': row['tags']
                        }
                        notes.append(note)

                    return jsonify({"notes": notes,
                                    'pagination': {
                                        'total': total,
                                        'page': page,
                                        'perPage': per_page,
                                        'nextPage': nextPage
                                    }}), 200

            except Exception as e:
                self.conn.rollback()
                print(f"An error occurred: {e}")
                return jsonify({'message': 'An error occurred', 'error': str(e)}),
500

            finally:
        cur.close()
```

3.1.3. modules/tasks.py

The tasks module combines the most techniques from other modules, building on the base provided by the notes.py files. It contains cross-table parametrized SQL, uses Common Table Expressions to simplify SQL statements and uses schemas to validate and sanitize the inputs.

3.1.3.1 Initialization

```
class TaskApi(BaseNote):
    def __init__(self, app, conn, tokenization_manager, recents_manager):
        super().__init__(app, conn, tokenization_manager, recents_manager)
        self.task_schema = TaskSchema()
        self.task_routes()

        self.subtasks_cte = """SubtasksCTE AS (
                    SELECT
                        st.task_id,
                        json_agg(json_build_object(
                            'subtaskid', st.id,
                            'description', st.description,
                            'completed', st.completed,
                            'index', st.st_index
                        )) AS subtasks
                    FROM Subtasks st
                    GROUP BY st.task_id
                )"""

    def tokenize(self,noteId,title,content,subtasks):
        text = [title, content] + [subtask['description'] for subtask in subtasks]
if subtasks else [title, content]
        priority = sum(len(string) for string in text)
        self.tokenization_manager.add_note(
            text=text,
            priority=priority,
            note_id=noteId
      )
```

The taskAPI class inherits the BaseNote class mentioned previously, but some additions are made to accommodate for the new task-related tables, such as subtasks

3.1.3.2 Task creation

```
    @self.app.route('/api/tasks/create', methods=['POST'])
        @jwt_required()
        @token_required
        def create_task():
            try:
                userId = str(g.userId)  # Convert UUID to string if userId is a
UUID

                data = self.task_schema.load(request.get_json())

                title = data['title']
                tags = data['tags']
                content = data['content']
                subtasks = data['subtasks']
                due_date = data.get('due_date')

                with self.conn.cursor(cursor_factory=psycopg2.extras.DictCursor) as
cur:
                    noteId = self.create_note(cur, userId, title, content, 'task',
tags)

                    cur.execute( #Insert into Tasks table
                        """
                        INSERT INTO Tasks (note_id, completed, due_date)
                        VALUES (%s, %s, %s) RETURNING id
                        """,
                        (noteId, False, due_date)
                    )
                    taskId = cur.fetchone()[0]
                    new_subtasks = None
                    if subtasks:
                        subtask_tuples = [
                            (taskId, subtask['description'], False,
subtask['index'])
                            for subtask in subtasks
                        ]
                        cur.executemany(
                            """
                            INSERT INTO Subtasks (task_id, description, completed,
st_index)
                            VALUES (%s, %s, %s, %s)
                            RETURNING id, task_id, description, completed, st_index
                            """,
                            subtask_tuples
                        )
                        cur.execute(
                            """
                            SELECT id, task_id, description, completed, st_index
                            FROM Subtasks
                            WHERE task_id = %s
                            """,
                            (taskId,)
                        )
                        new_subtasks = cur.fetchall()


                self.tokenize(noteId,title,content,subtasks)
                self.conn.commit()
                return jsonify({
                    'message': 'Task created successfully',
                    'data': {
                        'noteid': noteId,
                        'taskid': taskId,
                        'subtasks': new_subtasks
                    }
                }), 200
            except Exception as error:
                self.conn.rollback()
                print('Error during transaction', error)
                raise
            finally:
                cur.close()
```

The task creation goes as follows:

Firstly, the ID of the user that sent this request is taken and stored, along with the data that had been passed in the request. This data is validated via the following schemas, that inherit the BaseNote schema:

```
# Subtask Schema
class SubtaskSchema(Schema):
    subtaskid = fields.UUID(required=False)
    description = fields.Str(required=True, validate=lambda s: 500 >= len(s) > 0)
    completed = fields.Bool(required=True)
    index = fields.Int(required=True)
```

```
# Task Schema
class TaskSchema(BaseSchema):
    taskid = fields.UUID(required=False)
    due_date = fields.Str(required=False, allow_none=True)
    subtasks = fields.List(fields.Nested(SubtaskSchema), required=False,
allow_none=True)
    completed = fields.Bool(required=False)
```

Secondly, the note is created in the Notes table of the database, with the autogenerated by the database UUID being returned. Then, this ID is passed as a foreign key into the Tasks table through an SQL statement and the subsequent TaskID as a foreign key into the Subtasks table using nested loops for simplicity, if there were any provided in the POST request.
Finally, any user-entered data is sent for tokenization and the database-generated IDs are returned in a JSON request to be used to correctly display the new notes on the frontend.

3.1.3.3 Updating tasks

```
       @self.app.route('/api/tasks/update', methods=['PUT'])
        @jwt_required()
        @token_required
        def update_task():
            try:
                userId = str(g.userId)  # Convert userId to string
                task = self.task_schema.load(request.get_json())

                note_id = str(task['noteid'])  # Convert note_id to string
                task_id = str(task['taskid'])  # Convert task_id to string
                title = task['title']
                content = task['content']
                due_date = task.get('due_date')
                subtasks = task.get('subtasks', [])
                tags = task.get('tags', [])

                with self.conn.cursor(cursor_factory=psycopg2.extras.DictCursor )
as cur:
                    if not verify_task_ownership(userId, task_id, cur):
                        return jsonify({'message': 'You do not have permission to
update this task'}), 403

                    cur.execute("""
                        UPDATE Notes
                        SET title = %s, content = %s
                        WHERE id = %s AND user_id = %s
                    """, (title, content, note_id,  userId))
                    cur.execute("""
                        UPDATE Tasks
                        SET due_date = %s
                        WHERE id = %s
                    """, (due_date, task_id))

                    cur.execute("DELETE FROM Subtasks WHERE task_id = %s",
(task_id,))
                    for subtask in task.get('subtasks', []):
                        cur.execute("""
                            INSERT INTO Subtasks (task_id, description, completed,
st_index)
                            VALUES (%s, %s, %s, %s)
                        """, (task_id, subtask['description'],
subtask['completed'], subtask['index']))

                    self.update_notetags(cur, note_id, tags)

                    self.tokenize(note_id, title, content, subtasks)
                    self.conn.commit()

                    return jsonify({'message': 'Task updated successfully', 'data':
None}), 200
            except Exception as e:
                self.conn.rollback()
                print(f"An error occurred: {e}")
                raise
            finally:
                cur.close()
```

The data is received in a similar method: the user ID is retrieved and the data is validated using a schema mentioned above. However, when updating tasks, a special function runs to verify that the task and subtask being altered indeed belong to the user that tries to alter them, to avoid any mishaps or attacks:

```
def verify_subtask_ownership(user_id, subtask_id, cur):
    cur.execute("""
        SELECT st.id
        FROM Subtasks st
        JOIN Tasks t ON st.task_id = t.id
        JOIN Notes n ON t.note_id = n.id
        WHERE st.id = %s AND n.user_id = %s
    """, (subtask_id, user_id))
    return cur.fetchone() is not None

def verify_task_ownership(user_id, task_id, cur):

    query = """
    SELECT Notes.user_id
    FROM Tasks
    JOIN Notes ON Tasks.note_id = Notes.id
    WHERE Tasks.id = %s
    """
    cur.execute(query, (task_id,))
    result = cur.fetchone()

    if (len(result)<1) or (result['user_id'] != user_id):
        print("False")
        return False
    return True
```

The function works by checking if the task or subtask exist in the database and if the stored user id is the same as the provided user id. Similar functions exist for every note type. If the ownership is verified, SQL queries run to update the corresponding Note, Task and Subtask records for a specific note. Finally, corresponding tags are updated, user content is sent for tokenization and a success code is returned.

3.1.3.4 Deleting a task

```
        @self.app.route('/api/tasks/delete', methods=['PUT'])
        @jwt_required()
        @token_required
        def delete_task():
            try:
                userId = g.userId
                data = request.get_json()
                taskId = data['taskId']
                noteId = data['noteId']
                with self.conn.cursor(cursor_factory=psycopg2.extras.DictCursor) as 
cur:
                    if verify_task_ownership(userId, taskId, cur) == False:
                        return jsonify({'message': 'You do not have permission to 
update this task'}), 403

                    stack =[12,11,10,2] #NoteTags, Subtasks, Tasks, Notes
                    stack.reverse()
                    if delete_notes_with_backoff(self.conn, noteId, stack):
                        self.tokenization_manager.delete_note_by_id(noteId)
                        return jsonify({'message': 'Task deleted successfully'}),
200
                    else:
                        return jsonify({'message': 'Failed to delete task data
after multiple retries'}), 500
            except Exception as e:
                self.conn.rollback()
                raise
```

The task deletion method is similar for all the notes. However, this time a few extra tables are added to the execution stack, and user's ownership is checked among more tables. The function ends with a success or error codes with messages returned to the frontend.

3.1.3.5 Toggling task fields

```
        @self.app.route('/api/tasks/toggle', methods=['PUT'])
        @jwt_required()
        @token_required
        def toggle_task_fields():
            try:
                userId = g.userId
                data = request.get_json()
                taskId = data.get('taskid')
                subtaskId = data.get('subtaskid')

                with self.conn.cursor(cursor_factory=psycopg2.extras.DictCursor) as
cur:
                    if not verify_task_ownership(userId, taskId, cur):
                        return jsonify({'message': 'You do not have permission to
update this task'}), 403

                    if not subtaskId:
                        toggle_task_sql = """
                            UPDATE Tasks
                            SET completed = NOT completed,
                                completion_timestamp = CASE
                                    WHEN completed THEN NULL  -- If the task is
being uncompleted, set timestamp to NULL
                                    ELSE CURRENT_TIMESTAMP    -- If the task is
being completed, set timestamp to current date and time
                                END
                            WHERE id = %s
                        """
                        cur.execute(toggle_task_sql, (taskId,))

                        cur.execute("""
                            SELECT completed
                            FROM Tasks
                            WHERE id = %s
                        """, (taskId,))
                        task_completed = cur.fetchone()['completed']
                        try:
                            cur.execute("""
                                SELECT total_completed_tasks
                                FROM taskstatistics
                                WHERE user_id = %s
                            """, (userId,))
                            total_completed_tasks = cur.fetchone()
['total_completed_tasks']
                            cur.execute("""
                                UPDATE taskstatistics
                                SET total_completed_tasks = %s
                                WHERE user_id = %s
                            """, (total_completed_tasks + 1 if task_completed else
(total_completed_tasks - 1 if total_completed_tasks-1>0 else 0), userId))
                        except:
                            cur.execute("""
                                INSERT INTO taskstatistics (user_id,
total_completed_tasks)
                                VALUES (%s, %s)
                            """, (userId, 1))

                    if subtaskId and verify_subtask_ownership(userId, subtaskId,
cur):
                        toggle_subtask_sql = """
                            UPDATE Subtasks
                            SET completed = NOT completed
                            WHERE id = %s AND task_id = %s
                        """
                        cur.execute(toggle_subtask_sql, (subtaskId, taskId))

                    self.conn.commit()
                    return jsonify({'message': 'Field toggled successfully',
"data": None}), 200

            except Exception as e:
                self.conn.rollback()
                print(f"An error occurred: {e}")
                return jsonify({'message': 'An error occurred', 'error': str(e)}),
500

            finally:
                cur.close()
```

This section is new and specific for the Tasks note type. Tasks are made in a way that they have an extra property named completion in their tables, which shows whether the task has been completed or not. Each subtask also has the same property, which allows to track progress on a specific task. Task completion is reversible, so toggle_task_fields() can work for both completing tasks and doing the opposite. It works by first receiving two properties: taskID and subtaskID. TaskID is always received and is used to find the task that the user is trying to mark as done. If no subtaskID is received, the application assumes that the user is attempting to complete the whole task. Regardless for that, the task and/or subtask ownership is first checked by functions mentioned at 3.1.3.3. If the user has the relevant rights, a CASE SQL query is used to set the value of the completion property in the relevant task or subtask record to the opposite boolean value.

Additionally, if a task was completed, several extra queries run to update the task statistics. For example, the number of total completed tasks is fetched from the TaskStatistics table, which is then either incremented or decremented based on the action that was instigated by the user.

Lastly, as always, any changes in the database are commited and sufficient error or success codes are sent back to the user.

3.1.3.6 Task Previews

```
@self.app.route('/api/tasks/previews', methods=['GET'])
@jwt_required()
@token_required
def get_task_previews():
    try:
        userId = g.userId
        # Pagination
        page = int(request.args.get('pageParam', 1))  # Default to page 1
        per_page = int(request.args.get('per_page', 5))  # Default to 5 items per
page
        offset = (page - 1) * per_page
        with self.conn.cursor(cursor_factory=psycopg2.extras.DictCursor) as cur:
            # Fetch the total count of tasks for pagination metadata
            total,nextPage = self.fetch_total_notes(cur, 'task', userId, page,
offset, per_page)
            cur.execute(f"""
                WITH {self.subtasks_cte},
                {self.tags_cte}
                SELECT
                    n.id AS note_id,
                    n.title AS title,
                    n.content AS content,
                    t.id AS task_id,
                    t.completed AS task_completed,
                    t.due_date AS task_due_date,
                    COALESCE(stc.subtasks, '[]') AS subtasks,
                    COALESCE(tgc.tags, '[]') AS tags
                FROM Notes n
                LEFT JOIN Tasks t ON n.id = t.note_id
                LEFT JOIN SubtasksCTE stc ON t.id = stc.task_id
                LEFT JOIN TagsCTE tgc ON n.id = tgc.note_id
                WHERE n.user_id = %s AND n.type = 'task' AND n.archived = FALSE
                ORDER BY n.updated_at DESC
                LIMIT %s OFFSET %s
            """, (userId, per_page
                , offset
                ))
            rows = cur.fetchall()
            tasks = []
            for row in rows:
                task = {
                    'noteid': row['note_id'],
                    'taskid': row['task_id'],
                    'title': row['title'][:100] + '...' if len(row['title']) > 100
else row['title'],
                    'content': row['content'][:200] + '...' if len(row['content'])
> 200 else row['content'],
                    'completed': row['task_completed'],
                    'due_date': row['task_due_date'],
                    'subtasks': row['subtasks'],
                    'tags': row['tags']
                }
                tasks.append(task)
            return jsonify({"tasks": tasks,
                            'pagination': {
                                'total': total,
                                'page': page,
                                'perPage': per_page,
                                'nextPage': nextPage
                            }}), 200
    except Exception as e:
        self.conn.rollback()
        print(f"An error occurred: {e}")
        return jsonify({'message': 'An error occurred', 'error': str(e)}), 500
    finally:
        cur.close()
```

As stated previously in the Notes module, previews are useful to reduce the size of the notes being loaded by removing unnecessary detail, which will anyways be invisible to the user on the page where all the notes are shown.

Since each user may have many notes, pagination is used. Pagination involves sending a small chunk of the actually avalible data to again reduce loading times and avoid fetching hundreds of notes, if a single page may only have enough space to house 10 previews. To implement pagination, several extra parameters are sent to the backend. In my implementation, it is the current page and the number of tasks per page. These are arguments determined by the frontend’s settings and pickers, but default to 1 and 5 to prevent errors if for some reason such arguments are not received. Then, an inherited method fetch_total_notes is used with a ‘task’ argument to calculate the total amount of notes. With these values stored, SQL schemas are executed to fetch data from Notes, Tasks, Subtasks and Tags tables. This information is grouped together and sectioned based on the LIMIT property, with the OFFSET being calculated from the current page number and the amount of data per page. OFFSET basically moves the window of fixed length (5 by default) across the data in the database to retrieve information that should be on the specified page. Finally, long text strings are truncated and the data is sent back to the user.

3.1.3.7 Retrieving a single task

```
@self.app.route('/api/task', methods=['GET'])
@jwt_required()
@token_required
def get_task():
    try:
        userId = g.userId
        noteid = request.args.get('noteid')

        with self.conn.cursor(cursor_factory=psycopg2.extras.DictCursor) as cur:
            cur.execute(f"""
                WITH {self.subtasks_cte},
                {self.tags_cte}
                SELECT
                n.id AS note_id,
                n.title AS note_title,
                n.content AS note_content,
                n.created_at AS task_created_at,
                t.id AS task_id,
                t.completed AS task_completed,
                t.due_date AS task_due_date,
                COALESCE(stc.subtasks, '[]') AS subtasks,
                COALESCE(tgc.tags, '[]') AS tags
            FROM Notes n
            LEFT JOIN Tasks t ON n.id = t.note_id
            LEFT JOIN SubtasksCTE stc ON t.id = stc.task_id
            LEFT JOIN TagsCTE tgc ON n.id = tgc.note_id
            WHERE n.user_id = %s AND n.id = %s AND n.type = 'task' AND n.archived =
FALSE
        """, (userId, noteid))

            row = cur.fetchone()
            if not row:
                return jsonify({'message': "Task not found"}), 404

            task = {
                'noteid': row['note_id'],
                'taskid': row['task_id'],
                'title': row['note_title'],
                'content': row['note_content'],
                'completed': row['task_completed'],
                'due_date': row['task_due_date'],
                'subtasks': row['subtasks'],
                'tags': row['tags']
            }

            self.recents_manager.add_note_for_user(userId, noteid)
            return jsonify({"task": task, 'message': "Task fetched successfully"}),
200

    except Exception as e:
        self.conn.rollback()
        print(f"An error occurred: {e}")
        return jsonify({'message': 'An error occurred', 'error': str(e)}), 500

    finally:
        cur.close()
```

This function has little difference from the function shown in 3.1.2.4, apart from the tables. There are more tables associated with the Tasks type, so more tables and Common Table Expressions are used. But in the end, data is sent to the user in JSON format.

3.1.4. modules/archive.py

Now I will pivot away from modules that are directly linked to a specific note type and continue talking about more general code. The Archive module helps complete the requirement of organization, as notes can be removed from their subsequent note page and placed into a storage without being deleted directly. This can be executed by using the archived property of the Notes table, which allows the application to filter out any notes that are not archived to display in corresponding sections. Meanwhile, all archived notes will be displayed on a separate page with options to permanently delete or restore the notes.

3.1.4.1 Archive and Restore

Due to this one property, the implementation is note type-agnostic, so there is no need to have a lot of cases concerning different note type-specific tables. This is because the Notes table contains the main information for any user-created note, regardless for its note type. The archiving function is called when a correct method calls the /archive API endpoint. The function just accesses the archived property of the Notes table and sets it to TRUE, subsequently removing the note from display anywhere else. Then, the connection commits and a success or error code is sent. A complete opposite SQL operation is performed when the note needs to be unarchived.

```
    @app.route('/api/notes/archive', methods=['PUT'])
    @jwt_required()
    @token_required
    def archive():
        try:
            userId = g.userId
            cur = conn.cursor()
            data = request.get_json()
            note_id = data.get('noteId')

            cur.execute(
                "UPDATE Notes SET archived = TRUE WHERE id = %s AND user_id = %s",
                (note_id, userId)
            )
            conn.commit()
            return jsonify({'message': 'Note archived successfully'}), 200
        except Exception as e:
            conn.rollback()
            raise
        finally:
            cur.close()
@app.route('/api/notes/restore', methods=['PUT'])
@jwt_required()
@token_required
    def restore():
        try:
            userId = g.userId
            cur = conn.cursor()
            data = request.get_json()
            note_id = data.get('noteId')
            cur.execute(
                "UPDATE Notes SET archived = FALSE WHERE id = %s AND user_id = %s",
                (note_id, userId)
            )
            conn.commit()
            return jsonify({'message': 'Note restored successfully'}), 200
        except Exception as e:
            conn.rollback()
            raise
        finally:
            cur.close()
```

3.1.4.2 GetAll

The code, however, gets more complicated whenever the archived notes need to be retrieved. Since any note type can be archived, the function must work regardless for the note type.

```
@app.route('/api/notes/archive', methods=['GET'])
@jwt_required()
@token_required
def getAll():
    try:
        userId = g.userId
        page = request.args.get('pageParam', 1, type=int)
        per_page = request.args.get('per_page', 10, type=int)
        offset = (page - 1) * per_page

        cur = conn.cursor(cursor_factory=psycopg2.extras.DictCursor)

        cur.execute("""
                SELECT COUNT(DISTINCT n.id) AS total
                FROM Notes n
                WHERE n.user_id = %s AND n.archived = TRUE
            """, (userId,))
        total = cur.fetchone()['total']

        cur.execute(
            """
            SELECT n.id AS note_id, n.title, n.content, n.type,
            t.id AS tagid,
            t.name,
            t.color
            FROM Notes n
            LEFT JOIN NoteTags nt ON n.id = nt.note_id
            LEFT JOIN Tags t ON nt.tag_id = t.id
            WHERE n.user_id = %s AND n.archived = TRUE
            LIMIT %s OFFSET %s
```

```
                """,
                (userId, per_page + 1, offset)  # Fetch one more note than the
limit
            )
            rows = cur.fetchall()
            if rows:
                notes = process_universal_notes(rows,cur)

            # Determine if there is a next page
            if len(notes) > per_page:
                next_page = page + 1
                notes = notes[:per_page]  # Return only the number of notes
requested
            else:
                next_page = None

            return jsonify({'message': 'Archived notes retrieved successfully',
'data': notes,
                            'pagination': {
                                    'total': total,
                                    'page': page,
                                    'perPage': per_page,
                                    'nextPage': next_page
                                }}), 200
        except Exception as e:
            conn.rollback()
            raise
        finally:
            cur.close()
```

Firstly, the user ID and the request arguments are retrieved in a type-safe manner. Then, other pagination parameters, such as the offset, are calculated. Then, the database cursor is created and the total number of archived notes is fetched for pagination purposes. After that, note information, such as its type, title, content and their tags are fetched from the Note and Tags tables. This more simplified implementation is used to avoid returning data of different structures for notes of different type. Since the user cannot open the note in the Archive section of the application, only data necessary to display the note and make a decision to restore or delete it will be returned. Finally, a utility function is used to process such universally applicable data is used (more about it later), and pagination values, along with the requested data composed into a JSON array is returned to the user, if there are no errors on the way.

3.1.5. modules/tags.py

What differentiates tags from other notes is that apart from the CRUD functionality, there also should be a way to link the tags to other notes. For this, a link table NoteTags is used, which links tag IDs to corresponding note IDs. However, before we get to this functionality, let’s quickly go over creating, updating, retrieving and deleting tags.

3.1.5.1 CRUD tags

```
#TAG MODULE
from datetime import datetime
from flask import Blueprint, g, jsonify, request
from flask_jwt_extended import jwt_required
import psycopg2.extras

import os
import sys
sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__),
os.path.pardir)))
from utils.utils import merge_sort, token_required, verify_tag_ownership
from formsValidation import TagSchema
```

Firstly, all modules from libraries like datetime, flask and psycopg2 are imported. Additionaly, any custom functions and schemas are imported from other modules.

```
def tag_routes(app, conn, model):

    @app.route('/api/tags/create', methods=['POST'])
    @jwt_required()
    @token_required
    def create_tag():
        cur = None
        try:
            userId = g.userId
            tag_schema = TagSchema()
            data = tag_schema.load(request.get_json()) #Deserialize the incoming
data
            tag_name = data['name']
            tag_color = data['color']

            cur = conn.cursor() # Create a cursor object
            cur.execute(
                "INSERT INTO Tags (name, color, user_id) VALUES (%s, %s, %s)
RETURNING id",
                (tag_name, tag_color, userId)
            )
            tag_id = cur.fetchone()[0]

            conn.commit() # Commit the transaction
            return jsonify({'message': 'Tag created successfully', 'data': {'id':
tag_id}}), 200
        except Exception as error:
            if conn:
                conn.rollback()
            print('Error during transaction', error)
            return jsonify({'message': 'Failed to create tag'}), 500
        finally:
            if cur:
                cur.close()
```

Tag creation is similar to creating any other notes: data is retrieved from the request, broken down, validated. Then, a cursor object is created which connects to the database and performs an INSERT operation to insert the tag data, such as its name, HEX color code and the ID of the user it belonds to. Finally, connection is committed and the generated ID of the tag is returned with a success or error code.

```
    @app.route('/api/tags/', methods=['GET'])
    @jwt_required()
    @token_required
    def getAll_tags():
        try:
            userId = g.userId # Get the user id from the g object
            with conn:
                with conn.cursor(cursor_factory=psycopg2.extras.DictCursor) as cur:
# Create a cursor object
                    cur.execute(
                        "SELECT id AS tagid, name, color FROM Tags WHERE user_id =
%s", # Query to fetch all tags
                        (userId,)
                    )
                    rows = cur.fetchall()
                    sorted_tags = merge_sort(rows, key=lambda tag: tag['name'])
#Mergesort to sort tags by name
                    tags = [{'tagid':tag['tagid'], 'name': tag['name'], 'color':
tag['color']} for tag in sorted_tags]

            return jsonify({'message': 'Tags fetched successfully', 'data': tags}),
200
        except Exception as e:
            return jsonify({'message': 'Failed to fetch tags'}), 500
```

This function is used on the Tags page to show the user all the tags they own, regardless of the notes they are connected to. Here, I used a mergesort (will be talked about in the utilities module) to quickly sort through the tags that were returned by the query. Finally, inline iteration is used to create an array of dictionaries of tags, which are returned along an appropriate code.

```
    @app.route('/api/tags/<int:note_id>', methods=['GET'])
    @jwt_required()
    @token_required
    def getTags():
        try:
            userId = g. userId

            cur = conn.cursor(cursorclass=psycopg2.extras.DictCursor) # Create a
cursor object

            note_id = request.args.get('note_id')
            cur.execute(
                "SELECT t.id AS t.tagid, t.name, t.color FROM Tags t JOIN NoteTags
nt ON t.id = nt.tag_id WHERE nt.note_id = %s AND t.user_id = %s",
                (note_id, userId)
            )
            tags = cur.fetchall()
            conn.commit() # Commit the transaction
            return jsonify({'message': 'Tags fetched successfully', 'data': tags}),
200
        except Exception as e:
            conn.rollback()
            raise
        finally:
            cur.close()
```

The getTags() function returns an array of tags that correspond to a specific note. The cursor joins Tags and NoteTags to only return tags that have the note id that was sent as arguments of the request. The retrieved tags are formed into an array and returned to the frontend.

```
    @app.route('/api/tags/edit', methods=['PUT'])
    @jwt_required()
    @token_required
    def editTag():
        cur = None
        try:
            tag_schema = TagSchema()
            userId = g.userId
            data = tag_schema.load(request.get_json()) # Deserialize the incoming
data
            tag_id = str(data['tagid'])
            name = data['name']
            color = data['color']
            cur = conn.cursor(cursor_factory=psycopg2.extras.DictCursor)# Create a
cursor object
            if verify_tag_ownership(userId, tag_id, cur) is False:  # Check if the
user has permission to update the tag
                return jsonify({'message': 'You do not have permission to update
this tag'}), 403

            cur.execute(
                "UPDATE Tags SET name = %s, color = %s WHERE id = %s", (name,
color, tag_id),
            )
            conn.commit()
            return jsonify({'message': 'Tag updated successfully', 'data': None}),
200
        except Exception as e:
            if cur is not None:
                conn.rollback()
            return jsonify({'message': 'An error occurred', 'error': str(e)}), 500
        finally:
            if cur is not None:
                cur.close()
```

To edit the tag, apart from the deserialization, creation of cursor objects and other techniques like error handling, a function is used to validate that the tag indeed belongs to the user.

```
    @app.route('/api/tags/delete', methods=['PUT'])
    @jwt_required()
    @token_required
    def deleteTag():
        try:
            userId = g.userId

            cur = conn.cursor()
            data = request.get_json() # Deserialize the incoming data
            tag_id = data['tagid']

            if not verify_tag_ownership(userId, tag_id, cur):
                return jsonify({'message': 'You do not have permission to update 
this tag'}), 403

            # Perform deletion operations
            cur.execute("DELETE FROM Tags WHERE id = %s AND user_id = %s", (tag_id, 
userId))
            cur.execute("DELETE FROM NoteTags WHERE tag_id = %s", (tag_id,))
            conn.commit()

            return jsonify({'message': 'Tag deleted successfully', 'data': None}), 
200
        except Exception as e:
            conn.rollback()
            raise
        finally:
            cur.close()
```

Finally, deleting tags ensures that the tag belongs to a user and that all the note-tag relationships are also deleted from the NoteTags table.

3.1.5.2 Linking Tags

```
    @app.route('/api/tags/add', methods=['POST'])
    @jwt_required()
    @token_required
    def addTag():
        try:
            userId = g. userId
            cur = conn.cursor()
            data = request.get_json() # Deserialize the incoming data
            note_id = data['note_id']
            tag_id = data['tagid']
            if verify_tag_ownership(userId, tag_id, cur) is False: # Check if the
user has permission to update the tag
                return jsonify({'message': 'You do not have permission to update
this tag'}), 403

            cur.execute (
                "INSERT INTO NoteTags (note_id, tag_id) VALUES (%s, %s)", (note_id,
tag_id),
            )
            conn.commit() # Commit the transaction
            cur.close()
            return jsonify({'message': 'Tag added successfully', 'data': None}),
200
        except Exception as e:
            conn.rollback()
            cur.close()
            raise
```

Linking tags does not even require editing the Tags table, as everything is done in the NoteTags table before verifying that the tags belong to the user trying to link them. No data needs to be sent to the user.

3.1.6. Widget-related modules

3.1.6.1 models/widget.py

```
from abc import ABC, abstractmethod
from dataclasses import dataclass
from typing import Dict


@dataclass
class WidgetData:
    id: int
    widget_id: str
    title: str
    data_source_type: str
    data_source_id: str
    configuration: Dict
    source_data: Dict = None

class Widget(ABC):
    @abstractmethod
    def get_source_data(self, cur, user_id):
        pass

class NumberWidget(Widget):
    def get_source_data(self, cur, user_id):
        cur.execute('SELECT streak FROM habits WHERE note_id = %s',
(self.data_source_id,))
        streak = cur.fetchone()
        return {'streak': streak[0] if streak else 0}

class ChartWidget(Widget):
    def get_source_data(self, cur, user_id):
        source_mapping = {
            'task': ('tasks', 'completion_timestamp'),
            'habit': ('habitcompletion', 'completion_date'),
            'goal': ('goals', 'completion_timestamp')
        }

        if self.data_source_type not in source_mapping:
            return {}

        table, date_column = source_mapping[self.data_source_type]

        cur.execute("""
            SELECT DATE_TRUNC('month', t.{}) as month, COUNT(*) as count
            FROM {} t
            JOIN notes n ON t.note_id = n.id
            WHERE n.user_id = %s
            AND t.{} >= NOW() - INTERVAL '6 months'
            GROUP BY month
            ORDER BY month DESC
        """.format(date_column, table, date_column), (user_id,))

        return {'monthly_data': cur.fetchall()}

class ProgressWidget(Widget):
    def get_source_data(self, cur, user_id):
        # Get total and completed milestones for a goal
        cur.execute("""
            SELECT
                COUNT(*) as total_milestones,
                SUM(CASE WHEN completed = TRUE THEN 1 ELSE 0 END) as
completed_milestones
            FROM milestones m
            JOIN goals g ON m.goal_id = g.id
            WHERE g.note_id = %s
        """, (self.data_source_id,))

        result = cur.fetchone()
        if not result:
            return {'total_milestones': 0, 'completed_milestones': 0}

        return {
            'total_milestones': result[0],
            'completed_milestones': result[1] or 0
        }

class HabitWeekWidget(Widget):
    def get_source_data(self, cur, user_id):
        # Get last 7 days of habit completions
        cur.execute("""
            SELECT completion_date::date
            FROM habitcompletion
            WHERE habit_id = %s
            AND completion_date >= NOW() - INTERVAL '7 days'
            ORDER BY completion_date DESC
        """, (self.data_source_id,))

        completions = {row[0] for row in cur.fetchall()}

        # Create boolean array for last 7 days
        today = datetime.now().date()
        weekly_completions = [
            (today - timedelta(days=i)) in completions
            for i in range(7)
        ]

        return {'weekly_completions': weekly_completions}

def create_widget(widget_type):
    """Factory method to create appropriate widget instance"""
    widget_types = {
        'Number': NumberWidget,
        'Chart': ChartWidget,
        'Progress': ProgressWidget,
        'HabitWeek': HabitWeekWidget
    }
    return widget_types.get(widget_type, Widget)()
```

Since there are several predetermined widget types we need to first define define the data structures and abstract base class for widgets to make the subsequent creation of other widget types easier. The base class for a widget will look as follows:

WidgetData:

- id: A UUID type ID for a widget
- widget_id: The type of the widget (“task”, “chart”, etc.)
- title: Title of the widget.
- data_source_type: Type of data source for the widget ("task", "habit", "goal").
- data_source_id: Identifier of the specific data source (such as the ID of a specific task).
- configuration: Dictionary containing widget-specific configuration options like its location.

- `source_data: Dictionary containing the actual data fetched from the data source. Initially None, it's populated by the get_source_data method.`

The base Widget class has the following methods:

- get_source_data(): This method is made to be overridden by actual widget classes. It will be responsible for fetching widget source data from the database

Next, come the predetermined widget types:

- **NumberWidget:** Widget class for displaying a single number, such as a habit streak. Here, get_source_data() is overwritten with appropriate SQL queries for this widget type
- **ChartWidget:** This widget fetches data for a chart based on the data_source_type (task, habit, or goal). It queries the appropriate table (tasks, habitcompletion, or goals) and aggregates the data by month for the last six months. It joins with the notes table to filter by user id. The source_mapping dictionary helps map the data_source_type to the correct table and date column names.
- The **ProgressWidget** class, is made to provide data for a progress display, which would visualise completion of a goal. Its get_source method runs SQL query to retrieve the total number of milestones associated with a specific goal and the count of completed milestones. The query uses a CASE statement within the SUM aggregation to count only those milestones where the completed column is set to true. The widget returns a dictionary of completed and total milestones for further procesing.
- Finally, **HabitWeekWidget** class provides data to display on a streak widget. Its get_source_data method queries the habitcompletion table to find all completion dates for a specific habit within the last seven days. Then, it creates an array of booleans based on whether a habit has been completed on each day, using timedelta and datetime functions from the datetime library to work with dates.

3.1.6.2 repositories/widget_repo.py

```
import json
from typing import List, Optional, Dict
import psycopg2.extras
from models.widget import WidgetData, create_widget

class WidgetRepository:
    def __init__(self, conn):
        self.conn = conn
        self._cache = {}

    def _invalidate_cache(self, user_id):
        self._cache.pop(f"user_widgets_{user_id}", None)
```

The widget repository is used to split data access logic from the Flask Api routes of my application for simplicity purposes. It imports WidgetData from modules/widget.py to create widget objects that will be displayed on the user's dashboard. This repository works as follows:

3.1.6.2.1 WidgetRepository initialisation cache

Firstly, WidgetRepository takes a database connection object and stores it in self.conn. It also creates an in-memory cache called self._cache as a protected attribute to store retrieved widgets for each user to avoid many database queries. The cache works as a hashmap with the key being the user ID.
The _invalidate_cache method is used to clear the cache for a given user. It takes the user_id as an argument and removes the corresponding entry from the self._cache dictionary.

3.1.6.2.2 Getting user widgets

```
def get_user_widgets(self, user_id):
        cache_key = f"user_widgets_{user_id}"
        if cache_key in self._cache:
            return self._cache[cache_key]

        with self.conn.cursor(cursor_factory=psycopg2.extras.DictCursor) as cur:
            cur.execute("""
                SELECT w.*,
                       COALESCE(jsonb_agg(s.*) FILTER (WHERE s.id IS NOT NULL),
'[]') as sources
                FROM user_widgets w
                LEFT JOIN widget_sources s ON w.data_source_type = s.type
                WHERE w.user_id = %s
                GROUP BY w.id
            """, (user_id,))
            widgets = [WidgetData(**row) for row in cur.fetchall()]
            self._cache[cache_key] = widgets
            return widgets

def get_widget_by_id(self, widget_id, user_id):
        with self.conn.cursor(cursor_factory=psycopg2.extras.DictCursor) as cur:
            cur.execute("""
                SELECT * FROM user_widgets
                WHERE id = %s AND user_id = %s
            """, (widget_id, user_id))
            widget = cur.fetchone()
            return WidgetData(**widget) if widget else None
```

The get user widgets method retrieves the widgets that belong to a certain user. It first checks if the widgets for the given user_id are already in the cache. If they are, data is returned straight away, which improves performance

If the data is not cached, an SQL query fetches the widgets. The query results are then used to create a list of WidgetData objects using list comprehension: [WidgetData(**row) for row in cur.fetchall()]. The **row unpacks the dictionary row as arguments to the WidgetData class constructor. Retrieved widgets are cached and returned.

The get_widget_by_id function has similar functionality, but it just retrieves a single widget.

3.1.6.2.3 Other CRUD functions

```
 def create_widget(self, user_id, widget_data):
        with self.conn.cursor(cursor_factory=psycopg2.extras.DictCursor) as cur:
            cur.execute("""
                INSERT INTO user_widgets
                (user_id, widget_id, title, data_source_type, data_source_id,
configuration)
                VALUES (%s, %s, %s, %s, %s, %s)
                RETURNING *
            """, (
                user_id,
                widget_data['widget_id'],
                widget_data['configuration']['title'],
                widget_data['data_source_type'],
                widget_data.get('data_source_id'),
                json.dumps(widget_data['configuration'])
            ))
            widget = WidgetData(**cur.fetchone())
            self.conn.commit()
            self._invalidate_cache(user_id)
            return widget

    def update_widget(self, widget_id, user_id, widget_data) :
        with self.conn.cursor(cursor_factory=psycopg2.extras.DictCursor) as cur:
            cur.execute("""
                UPDATE user_widgets
                SET widget_id = %s, data_source_type = %s,
                    data_source_id = %s, configuration = %s
                WHERE id = %s AND user_id = %s
                RETURNING *
            """, (
                widget_data['widget_id'],
                widget_data['data_source_type'],
                widget_data['data_source_id'],
                json.dumps(widget_data['configuration']),
                widget_id,
                user_id
            ))
            widget = cur.fetchone()
            if widget:
                self.conn.commit()
                self._invalidate_cache(user_id)
                return WidgetData(**widget)
            return None

    def delete_widget(self, widget_id, user_id):
        with self.conn.cursor() as cur:
            cur.execute("""
                DELETE FROM user_widgets
                WHERE id = %s AND user_id = %s
                RETURNING id
            """, (widget_id, user_id))
            deleted = cur.fetchone() is not None
            if deleted:
                self.conn.commit()
                self._invalidate_cache(user_id)
            return deleted
```

Other CRUD widget functions include creation, deletion and update. Generally, there is little difference from note-related methods, except for caching, where appropriate functions to add, remove or update a widget in cache are ran if the database update is successful.

3.1.6.2.4 Widget data sources

```
    def get_widget_data_sources(self, widget_type, user_id):
        with self.conn.cursor(cursor_factory=psycopg2.extras.DictCursor) as cur:
            query_mapping = {
                'Chart': self._get_chart_sources_query(),
                'Progress': self._get_progress_sources_query(),
                'Number': self._get_number_sources_query(),
                'HabitWeek': self._get_habit_week_sources_query()
            }

            query = query_mapping.get(widget_type)
            if not query:
                return []

            cur.execute(query, (user_id,))
            return [row['source'] for row in cur.fetchall()]

    def _get_chart_sources_query(self):
        return """
            SELECT json_build_object(
                'id', type,
                'title', type || ' (' || count || ' items)',
                'type', type
            ) as source
            FROM (
                SELECT 'task' as type, COUNT(*) as count FROM Notes
                WHERE user_id = %s AND type = 'task'
                UNION ALL
                SELECT 'habit', COUNT(*) FROM Notes
                WHERE user_id = %s AND type = 'habit'
                UNION ALL
                SELECT 'goal', COUNT(*) FROM Notes
                WHERE user_id = %s AND type = 'goal'
            ) as counts
        """
```

_get_chart_sources_query() and similar _get_*sources functions are helper methods that define the SQL queries for retrieving data sources for different widget types. They may range from complex SELECT statements across multiple tables to specific notes. Then, these queries are ran to build an array of widget data sources.

3.1.6.3 modules/widgets.py

```
import json
import os
import sys

import psycopg2
from flask import g, jsonify, request
from flask_jwt_extended import jwt_required

sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__),
os.path.pardir)))
from repositories.widget_repository import WidgetRepository
from utils.utils import token_required


def widget_routes(app, conn):
    widget_repo = WidgetRepository(conn)


    @app.route('/api/user_widgets', methods=['GET'])
    @jwt_required()
    @token_required
    def get_user_widgets():
        try:
            widgets = widget_repo.get_user_widgets(g.userId)
            return jsonify([widget.to_dict() for widget in widgets])
        except Exception as e:
            return jsonify({'error': str(e)}), 500
    @app.route('/api/user_widgets/create', methods=['POST'])
    @jwt_required()
    @token_required
    def create_user_widget():
        try:
            widget = widget_repo.create_widget(g.userId, request.get_json())
            return jsonify(widget.to_dict())
        except Exception as e:
            return jsonify({'error': str(e)}), 500

    @app.route('/api/user_widgets/<int:widget_id>', methods=['PUT'])
    @jwt_required()
    @token_required
    def update_user_widget(widget_id):
        try:
            widget = widget_repo.update_widget(widget_id, g.userId,
request.get_json())
            if not widget:
                return jsonify({'error': 'Widget not found'}), 404
            return jsonify(widget.to_dict())
        except Exception as e:
            return jsonify({'error': str(e)}), 500

    @app.route('/api/user_widgets/<int:widget_id>', methods=['DELETE'])
    @jwt_required()
    @token_required
    def delete_user_widget(widget_id):
        try:
            if widget_repo.delete_widget(widget_id, g.userId):
                return jsonify({'success': True})
            return jsonify({'error': 'Widget not found'}), 404
        except Exception as e:
            return jsonify({'error': str(e)}), 500

    @app.route('/api/widgets/datasources/<widget_type>', methods=['GET'])
    @jwt_required()
    @token_required
    def get_widget_data_sources(widget_type):
        try:
            sources = widget_repo.get_widget_data_sources(widget_type, g.userId)
            return jsonify({'sources': sources})
        except Exception as e:
            return jsonify({'error': str(e)}), 500
```

This module is not as extensive as other note modules, as all logic here was moved to other files. What the widgets.py file does is receive the API request, run any necessary validations and call the widget repository for information to be returned. Each function also does error handling and returns appropriate errors if they do occur.

3.1.7. modules/users.py

Since my application expects multiple users, I need to provide them with several features, such as login, registration, being able to update their details and preferences, as well as their password, and lastly having a choice to completely wipe their account from the database.

3.1.7.1 Registration

```
 @app.route('/api/register', methods=['POST'])
    def register():
        try:
            user_schema = UserSchema()
            result = user_schema.load(request.json)

            username = result['username']
            email = result['email']
            password = result['password']

            # Hash the password
            hashed_password = bcrypt.hashpw(password.encode('utf-8'), 
bcrypt.gensalt())

            cur = conn.cursor()
            cur.execute("SELECT * FROM users WHERE username = %s OR email = %s", 
(username, email))
            existing_user = cur.fetchone()

            if existing_user:
                return jsonify({'message': 'User with this username or email 
already exists'}), 400

            preferences = json.dumps({"theme": "light", "model": "default"})
            cur.execute("INSERT INTO users (username, email, password, preferences) 
VALUES (%s, %s, %s, %s) RETURNING id",
                        (username, email, hashed_password, preferences))
            userId = cur.fetchone()[0]

            try:
                access_token = create_access_token(identity=userId)
            except Exception as e:
                conn.rollback()
                raise

            conn.commit()
            cur.close()

            response = jsonify({'message': 'User created successfully', 'userId': 
userId})
            response.set_cookie('token', access_token)
            return response
        except ValidationError as err:
            return jsonify(err.messages), 400
        except Exception as error:
            conn.rollback()
            raise
        finally:
            cur.close()
```

It is important to notice that no @token_required decorator function is used since the user does not yet have a token. When the request is sent to the /api/register endpoint, it is validated with the following schema:

```
class UserSchema(Schema):
    username = fields.Str(required=True, validate=lambda s: len(s) > 4)
    email = fields.Email(required=True)
    password = fields.Str(required=True, validate=lambda s: len(s) >= 6)
```

The validation ensures that the requred fields are present on registration and that the password is long enough. Then, data is unpacked from the request and the password is encrypted with the hashpw function of the bcrypt library. Before creating the user, a SELECT query is first ran to ensure that there are no users with the same name or email. If there are, a corresponding error is raised and the function returns an error message.

However, if there is no user, the user details and the encrypted password are stored in the users table of the database, with default preferences being added to the stored data. To ensure that the user stays logged in, a Json WebToken is generated by an in-built Flask function, which stores the userId and will be passed along any other requests to let the code know the ID of the user trying to access the API. Finally, a success message is returned along with a cookie, which stores the JWT generated.

3.1.7.2 Login

```
    @app.route('/api/login', methods=['POST'])
    def login():
        try:
            login_schema = LoginSchema()
            data = login_schema.load(request.json)
            username = data['username']
            password = data['password']

            cur = conn.cursor(cursor_factory=psycopg2.extras.DictCursor)
            cur.execute("SELECT * FROM users WHERE username = %s", (username,))
            user = cur.fetchone()

            if (user
                and bcrypt.checkpw(password.encode('utf-8'),
user['password'].encode('utf-8'))
                ):

                userId = str(user['id'])
                access_token = create_access_token(identity=userId,
expires_delta=timedelta(days=5))

                response = jsonify({'message': 'Login successful', 'preferences':
user['preferences']})
                response.set_cookie('token', access_token)
                return response, 200
            else:
                return jsonify({'message': 'User not found'}), 401
        except ValidationError as err:
            return jsonify(err.messages), 400
        except Exception as error:
            conn.rollback()
            raise
        finally:
            cur.close()
```

The way that login works is quite similar: username and password are unpacked and validated. Then, the returned password is encoded with a bcrypt.hashpw function and the resulting hash is compared to the one stored in the database. If the hashes match, the password is assumed to be correct and the ID of the user with this username is fetched with a SELECT SQL statement. A JWT is generated, with an expiry date set for five days, so the user will stay logged in for this time period. Finally, the token is returned as a cookie along a success message.

3.1.7.3 Updating user data

Users should be able to change their username, email and password, as well as their preferences, such as the theme that automatically loads on the website upon login. To retrieve user data and change their username and email, the following code is used:

```
@app.route('/api/user', methods=['GET'])
@jwt_required()
@token_required
def get_user():
        try:
            userId = g.userId

            with conn.cursor(cursor_factory=psycopg2.extras.DictCursor) as cur:
                cur.execute(
                    """SELECT username, email, preferences FROM users WHERE id =
%s""", (userId,)
                )
                data = cur.fetchone()
                user = {
                    'username': data['username'],
                    'email': data['email'],
                    'preferences': data['preferences']
                }
                return jsonify({'data': user, 'message': 'User data retrieved
successfully'}), 200
        except Exception as e:
            raise

@app.route('/api/user', methods=['PUT'])
@jwt_required()
@token_required
def update_user():
        try:
            userId = g.userId
            data = request.get_json()

            with conn.cursor() as cur:
                if 'username' in data and 'email' in data:
                    cur.execute(
                        """UPDATE users SET username = %s, email = %s WHERE id =
%s""",
                        (data['username'], data['email'], userId)
                    )
                if 'preferences' in data:
                    cur.execute(
                        """UPDATE users SET preferences = %s WHERE id = %s""",
                        (json.dumps(data['preferences']), userId)
                    )
                conn.commit()
                return jsonify({'message': 'User data updated successfully'}), 200
      except Exception as e:
            raise
```

These functions use simple SELECT and UPDATE SQL statements alongside error handling techniques and do not require any elaboration. More complex cases are reviewed in **3.1.2 and 3.1.3.**

3.1.7.4 Update user password

```
    @app.route('/api/user/password', methods=['PUT'])
    @jwt_required()
    @token_required
    def update_password():
        try:
            userId = g.userId
            data = request.get_json()
            password = data['password']
            new_password = data['newpassword']

            with conn.cursor() as cur:
                cur.execute(
                    '''SELECT password FROM users WHERE id = %s''',
                    (userId,)
                )
                user = cur.fetchone()
                if not user:
                    return jsonify({'message': 'Invalid user credentials'}), 401
                elif bcrypt.checkpw(password.encode('utf-8'), user[0].encode('utf-
8')):
                    if password == new_password:
                        return jsonify({'message': 'New password cannot be the same
as the old password'}), 400

                    hashed_newpassword = bcrypt.hashpw(new_password.encode('utf-
8'), bcrypt.gensalt()).decode('utf-8')
                    cur.execute(
                        """UPDATE users SET password = %s WHERE id = %s""",
                        (hashed_newpassword, userId)
                    )
                    conn.commit()
                    return jsonify({'message': 'Password updated successfully'}),
200
                else:
                    return jsonify({'message': 'Invalid user credentials'}), 401
        except Exception as e:
            raise
```

To update a user password, the user must provide their old and new passwords. These two passwords are unpacked from the request parameters and compared. Firstly, the hashes of the provided old password and the actual password that was freshly fetched from the database are compared to ensure that it is the user changing the password is doing so with consent of the user that created the account. Additionally, the two passwords are the same, an error is raised to notify the user of such mistake. If there are no errors, the new password is encrypted and stored instead of the old hash in the database and a success code is returned from the API call.

3.1.7.5 Delete user

```
  @app.route('/api/user/delete', methods=['DELETE'])
    @jwt_required()
    @token_required
    def delete_user():
        try:
            userId = g.userId
            data = request.get_json()
            password = data['password']

            with conn.cursor() as cur:
                cur.execute(
                    '''SELECT password FROM users WHERE id = %s''',
                    (userId,)
                )
                user = cur.fetchone()
                if not user:
                    return jsonify({'message': 'Invalid user credentials'}), 401
                elif bcrypt.checkpw(password.encode('utf-8'), user[0].encode('utf-
8')):
                    delete_user(conn, userId) #Attempt deleting user from the db
                    return jsonify({'message': 'User deleted successfully'}), 200
#Return success message even if deletion fails (hehehe)
                else:
                    return jsonify({'message': 'Invalid user credentials'}), 401
        except Exception as e:
            conn.rollback()
            raise
```

This function just works as an API endpoint , with the main logic being performed by the delete_with_backoff function, that will be explained in-depth in the utils file. What is done in this specific function is the comparison of the two password hashes and any error handling.

3.1.8. modules/ honorable mentions

3.1.9. utils/utils.py

This file hosts any utility functions that are not large enough to deserve a separate file for their implementation but are widely used throughout the program, so they cannot be defined in any of the module files. Most of the code, however, can be categorized in the following categories:

3.1.9.1 Security

```
def generateToken(userId):
    try:
        exp = datetime.datetime.utcnow() + datetime.timedelta(days=7) # expires in
1 week
        payload = {
            'identity': str(userId),
            'exp': exp
        }
        token = jwt.encode(payload, Config.JWT_SECRET_KEY, algorithm='HS256')
        return token
    except Exception as e:
        print('Error generating token', e)
        return None
```

The first utility function encapsulates logic that involves generating the JsonWebToken storing the user ID for identification. The expiration date is set to be one week from the time of creation of the token and will be stored as a cookie in the browser, allowing the user to automatically log in if they open the webpage. Then, the user will have to log in again.

```
# Decorator for authentication
def token_required(f):
    @wraps(f)
    def decorated_function(*args, **kwargs):
        token = request.cookies.get('token') #Get token from cookies
        if not token:
            return jsonify({'message': 'Token is missing!'}), 403
        try:
            decoded = jwt.decode(token, Config.JWT_SECRET_KEY,
algorithms=['HS256'])
            g.userId = decoded.get('sub')  # Store decoded token data in Flask's g
object

        except jwt.ExpiredSignatureError:
            return jsonify({'message': 'Token has expired'}), 403
        except jwt.InvalidTokenError:
            return jsonify({'message': 'Invalid token'}), 403
        return f(*args, **kwargs)
    return decorated_function
```

token_required is a decorator function. A decorator function wraps the function and dynamically alters its code, without me having to manually change it everywhere. Since the token decoding function is the same for all protected endpoints (endpoints where a JWT is required), this implementation removes redundant repetition and makes the code easier to maintain

3.1.9.2 Note ownership

```
def verify_subtask_ownership(user_id, subtask_id, cur):
    cur.execute("""
        SELECT st.id
        FROM Subtasks st
        JOIN Tasks t ON st.task_id = t.id
        JOIN Notes n ON t.note_id = n.id
        WHERE st.id = %s AND n.user_id = %s
    """, (subtask_id, user_id))
    return cur.fetchone() is not None

def verify_task_ownership(user_id, task_id, cur):

    query = """
    SELECT Notes.user_id
    FROM Tasks
    JOIN Notes ON Tasks.note_id = Notes.id
    WHERE Tasks.id = %s
    """
    cur.execute(query, (task_id,))
    result = cur.fetchone()

    if (len(result)<1) or (result['user_id'] != user_id):
        print("False")
        return False
    return True

def verify_habit_ownership(user_id, habit_id, cur):

    query = """
    SELECT Notes.user_id
    FROM Habits
    JOIN Notes ON Habits.note_id = Notes.id
    WHERE Habits.id = %s
    """
    cur.execute(query, (habit_id,))
    result = cur.fetchone()

    if (len(result)<1) or (str(result['user_id']) != str(user_id)):
        print("False")
        return False
    return True

def verify_goal_ownership(user_id, goal_id, cur):
    query = """
    SELECT Notes.user_id
    FROM Goals
    JOIN Notes ON Goals.note_id = Notes.id
    WHERE Goals.id = %s
    """
    cur.execute(query, (goal_id,))
    result = cur.fetchone()

    if (len(result)<1) or (str(result['user_id']) != str(user_id)):
        return False
    return True
```

```
def verify_milestone_ownership(user_id, milestone_id, cur):
    query = """
        SELECT Notes.user_id
        FROM Milestones
        JOIN Goals ON Milestones.goal_id = Goals.id
        JOIN Notes ON Goals.note_id = Notes.id
        WHERE Milestones.id = %s
    """
    cur.execute(query, (milestone_id,))
    result = cur.fetchone()
    print(result)

    if (len(result)<1) or (result['user_id'] != user_id):
        return False
    return True

def verify_tag_ownership(user_id, tag_id, cur):
    query = """
    SELECT user_id
    FROM Tags
    WHERE id = %s
    """
    cur.execute(query, (tag_id,))
    result = cur.fetchone()

    if (len(result)<1) or (result['user_id'] != user_id):
        return False
    return True
```

These functions are used on top of token verification to ensure that the user is editing a note or a tag that belongs to them. Each function runs a SELECT query to return the values of the user_id field of a record or one that is in its parent table and compares it with the provided user ID. If there are no such records or the IDs don't match, the function returns false, which results in the user getting blocked from editing this specific note and the transaction aborting.

3.1.9.3 Universal utilities

This is the last subcategory of the functions in the file. These functions can be applied to notes universally, but do not have a specific endpoint and are generally used as part of other subroutines.

```
def merge_sort(arr, key=lambda x: x):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left = merge_sort(arr[:mid], key)
    right = merge_sort(arr[mid:], key)

    return merge(left, right, key)

def merge(left, right, key):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if key(left[i]) <= key(right[j]):
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    result.extend(left[i:])
    result.extend(right[j:])
    return result
```

The above two functions are the implementation of mergesort using recursion: a sorting algorithm that works by splitting a list in to multiple sublists recursively until the list is sorted. Then, the results are added together. Mergesort is efficient as it has a time complexity of O(n log n). Mergesorts also perform fewer comparisons on average than quicksorts, so it may be more efficient in sorting large database queries, containing a lot of data in each item of the list.

```
def process_universal_notes(rows,cur):
    notes_dict = {}
    for row in rows:
        note_id = row['note_id']
        if note_id not in notes_dict:
            notes_dict[note_id] = {
                'noteid': note_id,
                'title': row['title'][:50] + '...' if len(row['title']) > 50 else 
row['title'],
                'content': row['content'][:100] + '...' if len(row['content']) > 
100 else row['content'],
                'type': row['type'],
                'tags': []
            }
        if row['tagid']:
            notes_dict[note_id]['tags'].append({
                'tagid': row['tagid'],
                'name': row['name'],
                'color': row['color']
            })
    # Fetch the secondary ID based on the type
    for note in notes_dict.values():
        if note['type'] == 'task':
            cur.execute("SELECT id AS taskid FROM Tasks WHERE note_id = %s", 
(note['noteid'],))
            secondary_id = cur.fetchone()
            note['secondaryid'] = secondary_id['taskid'] if secondary_id else None
        elif note['type'] == 'habit':
            cur.execute("SELECT id AS habitid FROM Habits WHERE note_id = %s", 
(note['noteid'],))
            secondary_id = cur.fetchone()
            note['secondaryid'] = secondary_id['habitid'] if secondary_id else None
        elif note['type'] == 'goal':
            cur.execute("SELECT id AS goalid FROM Goals WHERE note_id = %s", 
(note['noteid'],))
            secondary_id = cur.fetchone()
            note['secondaryid'] = secondary_id['goalid'] if secondary_id else None
        # Add any other types here
    notes = list(notes_dict.values())
    return notes
```

Some endpoints, such as search and the Archive module must return multiple different note types, that use different tables and datastructures along with all necessary IDs, so the user can select a note they want to edit and make another api call, this time to the correct module endpoint. This function first fetches the note title, type, content and other fields from the Notes table, where any valid note must have a record, regardless for its type, truncates and tuns an IF statement, which triggers other queries based on the note type. Finally, the results are converted into a list and returned.

```
def remove_overdue_archived_notes(conn):
    cur = conn.cursor(cursor_factory=psycopg2.extras.DictCursor)
    deleted_notes = 0
    cur.execute("""
        SELECT id, type FROM Notes
        WHERE archived = TRUE AND updated_at < NOW() - INTERVAL '1 month'
    """)
    notes_to_remove = cur.fetchall()
    for note in notes_to_remove:
        if note['type'] == 'task':
            cur.execute("""
                DELETE FROM Subtasks WHERE task_id IN (
                    SELECT id FROM Tasks WHERE note_id = %s
                )"""), (note['id'],)
            cur.execute("""
                DELETE FROM Tasks WHERE note_id = %s
            """, (note['id'],))
        elif note['type'] == 'habit':
            cur.execute("""
                DELETE FROM HabitsTasks WHERE habit_id IN (
                        SELECT id FROM Habits WHERE note_id = %s
                )""",(note['id'],))
            cur.execute("""
                DELETE FROM Habits WHERE note_id = %s
            """, (note['id'],))
        elif note['type'] == 'goal':
            cur.execute("""
                DELETE FROM Milestones WHERE goal_id IN (
                    SELECT id FROM Goals WHERE note_id = %s
                )
            """, (note['id'],))
            cur.execute("""
                DELETE FROM Goals WHERE note_id = %s
            """, (note['id'],))
        cur.execute(""" DELETE FROM NoteTags WHERE note_id = %s """, (note['id'],))
        cur.execute("""
            DELETE FROM Notes WHERE id = %s
        """, (note['id'],))
        deleted_notes += 1
    print(f"Number of removed notes: {deleted_notes}")
```

The Archive module allows users to archive notes instead of deleting them for a time period of up to one month. This function, whenever ran, will fetch all notes that are marked as archived and compare the timestamp of the last update of that specific record with the current data, all within a single SQL statement. If the interval is greater than one month, an iterative deletion algorithm runs to delete notes and other related records from all possible tables and the number of removed notes is displayed in the server console.

3.1.9.4 utils/word2vec.py

This file contains the logic required to convert a text-based note into a vector representation, which then can be used to search notes not based on keywords, but on context. These vector representation are in fact just multidimensional matrices. For example, the user may search for “barbecue” and get “chicken marinade recipe”, since the vector representations of the two strings may be similar. Of course, the actual representation of a vectorized note is much more complex. For example, in my project, notes are represented as matrices of hundreds of different values of double-precision floating point datatype.

The way to convert a string input, which contains user-entered keywords, such as the note title, content and values of any extra fields, such as milestones and subtasks, a text vectorization model is used. I decided to implement a Word2Vec model, that was trained on a dataset of journal entries. This dataset includes some extra data, such as respondent emotions, which were cut off when the program was being made.

3.1.9.5 Training the model

```
import ast
import os
import pickle
import pandas as pd
from gensim.models import Word2Vec
from nltk.tokenize import sent_tokenize, word_tokenize
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
import numpy as np
import nltk

MODEL_PATH = os.path.join('data', 'w2v.model')
PROCESSED_DATA_PATH = os.path.join('data', 'processed_data.txt')

def load_or_train_model():
    if os.path.exists(MODEL_PATH):
        print("Loading existing model...")
        model = Word2Vec.load(MODEL_PATH)
    else:
        print("Training new model...")

        if not os.path.exists(PROCESSED_DATA_PATH):
            df = pd.read_csv(os.path.join('backend/data', 'data.csv'),
encoding="utf-8")
            text_data = df['text'].dropna().tolist()
            data = process_text_data(text_data)
            with open (PROCESSED_DATA_PATH, 'w') as f:
                f.write(str(data))
                f.close()
        else:
            with open(PROCESSED_DATA_PATH, 'r') as f:
                data = f.read()
                data = ast.literal_eval(data)
                f.close()

        print("Data processed.")
        # Train the Word2Vec model
        model = Word2Vec(data, vector_size=100, window=5, sg=1)  # Skip Gram

        # Save the model for future use
        model.save(MODEL_PATH)

    return model
```

The first function in the file is load_or_train_model(). It handles loading and training of the Word2Vec model. It checks if the model exists, and if it does not, it processes the dataset, trains a new model and saves it.

It first checks if the model file (w2v.model) exists at the specified MODEL_PATH.

- If the file exists, it loads the pre-trained Word2Vec model.This avoids retraining the model every time the backend runs, saving time and resources.
- If the file doesn't exist, it proceeds to train a new model.
- If the processed dataset file exists, it reads the data from the file, using ast.literal_eval() to safely convert the string representation of the list of sentences back into a Python list.
- If the processed data file does not exist, it reads the raw data from a CSV file ('data.csv'), drops any rows without text, converts the text column to a list, and then calls the process_text_data() function. The processed data is then saved to the processed_data.txt file using f.write(str(data)) using built-in python file operations.
- After obtaining the processed data, it trains the Word2Vec model from the Gensim library using Word2Vec(data, vector_size=100, window=5, sg=1), where:
    - data: The list of sentences (each sentence is a list of words) used to train the model.
    - vector_size=100: The dimensionality of the word vectors.
    - window=5: The maximum distance between the current and predicted word 1 within a sentence.

3.1.9.6 Text tokenization

```
def process_text_data(text_data):

    lemmatizer = WordNetLemmatizer()
    stop_words = set(stopwords.words('english'))
    print('Processing text data...')
    data = []
    for text in text_data:
        for sent in sent_tokenize(text):
            cleared_sent = []
            tokens = word_tokenize(sent.lower())
            for token in tokens:
                if token not in stop_words and token.isalnum():
                    lemmatized_token = lemmatizer.lemmatize(token)
                    cleared_sent.append(lemmatized_token)
            if len(cleared_sent) > 1:
                data.append(cleared_sent)

    print(f'Processed {len(data)} sentences.')
    return data
```

The function’s job is to take raw text and clean it up, making it ready for the Word2Vec model.

Tokenization

During the tokenization step, I used nested for loops to break the text down into smaller pieces by breaking down paragraphs in single words.

- sent_tokenize(text) splits the text into sentences.
- word_tokenize(sent.lower()) splits each sentence into words and converts everything to lowercase. This is important because "Hello" and "hello" should be treated the same. These functions are part of the NLTK (Natural Language Toolkit) library that provides such tokenization tools.

Stop Words Removal:

- Some words appear a lot in user-generated text, but don't have much meaning for a model. These words include "the", "a", "is", "are" and are called stop words. They are removed so the model can focus on more important words
- stop_words = set(stopwords.words('english')) gets a set of these common words, and the code checks if each token is in this set.

Lemmatization:

- Words can have different forms like "run," "running," "ran". Lemmatization reduces words to their base or dictionary form, called the lemma. As a result, "running" would become "run." This helps the model understand that these different forms are related without having to “remember” all the forms of words.
- This task is performed on the split-up paragraphs with lemmatizer.lemmatize(token).

Final clean-up:

- token.isalnum() removes punctuation and other symbols.
- if len(cleared_sent) > 1: this makes sure that only sentences with more than one word are kept, so that there is some context for the word2vec model.

3.1.9.7 Processing tokens with the model

```
def combine_strings_to_vector(strings, model, preprocess):
    if preprocess:
        # Preprocess the text data
        processed_data = process_text_data(strings)
    else:
        processed_data = strings

    all_word_vectors = []

    for sentence in processed_data:
        # Ensure sentence is tokenized into words
        words = sentence.split() if isinstance(sentence, str) else sentence
        word_vectors = [model.wv[word] for word in words if word in model.wv]
        all_word_vectors.extend(word_vectors)

    if len(all_word_vectors) == 0:
        # Return a zero vector if no valid words found
        return np.zeros(model.vector_size).tolist()

    # Return the average vector as a list
    return np.mean(all_word_vectors, axis=0).tolist()
```

Now that the model is trained and the input data is processed to be used most efficiently, its time to actually use the model to convert whatever is left of user inputs into a note into vectors:

Firstly, depending on the preprocess flag, input data is preprocessed using the function mentioned above. An empty list storing all the word vectors is also initialised.

Then, every tokenized sentence is looped through and the model.wv[word] retrieves the vector for the given word from the Word2Vec model, with an inline IF statement, which ensures that the word is indeed in the model dictionary. All the word vectors are added to the all_word_vectors array.

Finally, the mean of all vectors is returned as a list, utilising the NumPy mean module.

3.1.9.8 Looking notes up with cosine similarity

```
DECLARE
 dot_product double precision := 0;
 norm_a double precision := 0;
 norm_b double precision := 0;
 i int;
BEGIN
 IF vec1 IS NULL OR vec2 IS NULL THEN
 RETURN 0;
 END IF;

 FOR i IN array_lower(vec1, 1)..array_upper(vec1, 1) LOOP
 dot_product := dot_product + (vec1[i] * vec2[i]);
 norm_a := norm_a + (vec1[i] * vec1[i]);
 norm_b := norm_b + (vec2[i] * vec2[i]);
 END LOOP;
 IF norm_a = 0 OR norm_b = 0 THEN
 RETURN 0;
```

```
    END IF;
    RETURN dot_product / (sqrt(norm_a) * sqrt(norm_b));
END;
```

This SQL code implements the cosine similarity function, a common metric used to determine how similar two vectors are.
dot_product double precision := 0 This variable will store the dot product of the two input vectors (`vec1` and `vec2`). The dot product is a measure of how much the two vectors point in the same direction.
The next tow variables will store the squared Euclidean norm (magnitude) of the vectors, initialised to zero.

```
IF vec1 IS NULL OR vec2 IS NULL THEN
 RETURN 0;
END IF;
```

This checks if either of the input vectors is `NULL`. If either vector is `NULL`, it means there's no valid vector representation, and the function returns 0, indicating no similarity.

```
FOR i IN array_lower(vec1, 1)..array_upper(vec1, 1) LOOP
 dot_product := dot_product + (vec1[i] * vec2[i]);
 norm_a := norm_a + (vec1[i] * vec1[i]);
 norm_b := norm_b + (vec2[i] * vec2[i]);
END LOOP;
IF norm_a = 0 OR norm_b = 0 THEN
    RETURN 0;
END IF;
```

This FOR loop iterates through each element of the input vectors, it calculates the product of the corresponding elements of vec1 and vec2 and adds it to the dot_product. Then, the square of each element is calculated and added to the Euclidean norm. Finally, null checks run to ensure that we are not operating with matrices of zeros.

```
RETURN dot_product / (sqrt(norm_a) * sqrt(norm_b));
```

Finally, cosine similarity is calculated. The result, which is a value between -1 and 1, is returned. A value closer to 1 indicates that the vectors are very similar, a value closer to -1 indicates they are not alike.

This function as a database function, along ID generation functions and is invoked in the SQL queries for the search endpoint.

3.1.10. utils/priorityqueue.py

Since note processing is a performance-intensive task and the web server may be running with limited resources, the vectorization tasks must be scheduled in a way that maximizes performance, while ensuring that notes are processed quickly. For this reason, a priority queue is implemented, where priority is calculated by the amount of characters to be processed.

For example, here is the tokenize function in goals.py, a file, similar to tasks.py and notes.py mentioned previously, which contains goal processing endpoints:

```
    def tokenize(self,noteId,title,content,milestones):
        text = [title, content] + [milestone['description'] for milestone in
milestones] if milestones else [title, content]
        priority = sum(len(string) for string in text)
        self.tokenization_manager.add_note(
            text=text,
            priority=priority,
            note_id=noteId
            )
```

As can be seen from the subroutine, the text array is iterated over and the length of the strings in the array is added and passed into the vectorization manager.

3.1.10.1 What is Heap?

In Python, a "heap" is a specialized tree-based data structure where parent nodes are ordered in relation to their children. The heapq module provides an implementation of the heap queue algorithm, which will be used to create a "priority queue." Here is why this specific datastructure will be used over an array:

- heapq efficiently maintains this priority order, making it ideal for tasks like scheduling or finding the smallest/largest elements.
- Priority queues, implemented with heaps, prioritize elements based on their values.
- Instead of strict LIFO or FIFO, the element with the highest (or lowest) priority is retrieved first, which simplifies queueing the items

3.1.10.2 Implementation

```
import threading
import heapq
import time
from dataclasses import dataclass, field
from typing import Any, Callable, List, Tuple
from utils.word2vec import load_or_train_model, combine_strings_to_vector
import psycopg2

 # Initialize the TokenizationTaskManager
def __init__(self, db_config, model):

 self.db_config = db_config
 self.model = model #Word2Vec model for text vectorization.

 self.task_queue: List[TokenizationTask] = [] # Initialize an empty list to serve
as a priority queue for tokenization tasks.
 self.task_counter = 0
 self.lock = threading.Lock()
 self.new_task_event = threading.Event()
 self.running = True
 self.worker_thread = threading.Thread(target=self.process_tasks, daemon=True) #
Create and start a worker thread that will process tasks from the queue.
 self.worker_thread.start()
```

This is the constructor for the TokenizationTaskManager class. It initialises all the necessary variables. A threading.Event is used for inter-thread communication, allowing the worker thread to wait efficiently for new tasks to be added. Furthermore, it initializes and starts a worker thread (self.worker_thread) from the threading library, which is responsible for asynchronously processing the tokenization tasks in the background. This is part of the multithreading functionality of the manager, which means that notes can be processed concurrently, speeding up the processing.

```
def connect_db(self):
 # Establishes a connection to the database
 conn = psycopg2.connect(
 host=self.db_config.host,
 database=self.db_config.database,
 user=self.db_config.user,
 password=self.db_config.password,
 port=self.db_config.port
 )
 return conn
```

This function is responsible for creating a separate connection to the database for the manager, so that other modules will not interfere with their requests.

```
def delete_note_by_id(self, note_id: str):
 # Removes all tasks associated with a specific note ID from the task queue
 with self.lock:
      self.task_queue = [note for note in self.task_queue if note.note_id !=
note_id]
      heapq.heapify(self.task_queue)
```

The delete_note_by_id function removes any pending tokenization tasks associated with a given note_id from the task queue. This could be used if a user deletes a note that has not yet been processed, so it needs to be removed from the queue to speed up processing of relevnt notes. The function begins by acquiring a lock (self.lock) to ensure thread-safe modification of the task_queue, as it might be accessed by the worker thread . It then uses a list comprehension to create a new task_queue that includes only the TokenizationTask objects whose note_id does not match the provided note_id. After removing the relevant tasks, it calls heapq.heapify(self.task_queue) to reorganize the list back into a min-heap.

```
def add_note(self, text: List[str], note_id:str, priority: int = 10, callback:
Callable[[Any], None] = None):
        # Adds a new note tokenization task to the task queue.

        with self.lock:
            # Remove any existing tasks with the same note_id to avoid duplicates
            self.task_queue = [note for note in self.task_queue if note.note_id !=
note_id]
```

```
            heapq.heapify(self.task_queue)

            note = TokenizationTask(priority, note_id, text, callback ,
self.task_counter)
            # Push the new task onto the priority queue
            heapq.heappush(self.task_queue, note)
            print(self.task_queue)
            self.task_counter += 1
            self.new_task_event.set() # Signal the worker thread that a new task is
available
```

This function adds a new note tokenization task to the processing queue. It takes the note's text content (`text`) and priority, provided by a relevant note module.

It first checks if any tasks with the same note_id already exist in the queue and removes them, so only the latest version of a note is processed. After potentially removing duplicates and re-heapifying, it creates a new object with the provided details.

This object is then pushed to the heap (that works as a priority queue) (self.task_queue) using heapq.heappush, which maintains the heap order based on task priority. The function also increments a task_counter to uniquely identify tasks and sets the new_task_event to signal to the worker thread that a new task has been added and is ready for processing.

```
def process_tasks(self):
        # Worker thread function to continuously process tokenization tasks from
the priority queue.
        conn = self.connect_db() # Establish a dedicated database connection for
this thread.
        model = self.model
        while self.running:
            self.new_task_event.wait()

            while True:
                with self.lock:
                    if not self.task_queue: # Check if the task queue is empty.
                        self.new_task_event.clear()
                        break
                    note = heapq.heappop(self.task_queue) # Pop the highest
priority task from the queue.

                # Process the note (tokenize and handle callback).
                try:
                    vector = self.tokenize_text(note.text, model)
                    if note.callback:
                        note.callback(vector, note, conn,)
                    else:
                        self.default_callback(vector, note, conn,) # Execute the
default callback.
                except Exception as e:
                    print(f"Error processing note {note.note_id}: {e}")
            time.sleep(0.1)
        conn.close()
```

This function runs a dedicated worker thread to process notes. It continiously monitors the heap queue, processed the notes with the highest priority and returns them through their specified callback.

It first establishes a new database connection, then enters a while self.running loop, which runs while the manager runs. Inside this loop, it waits for a new task signal using self.new_task_event.wait(). As a result, the thread only wakes up when there is data in the queue, which conserves processing power. Once it wakes up, it enters another loop to process all tasks in the queue. Inside this inner loop, it pops the highest priority task using heapq.heappop. Then, it calls the tokenization function to tokenize the retrieved text. It then checks for a task-specific callback. If it exists, it's executed; otherwise, the self.default_callback is called. After processing all available tasks, the thread pauses using time.sleep(0.1) before checking for new items in the queue.

```
    def tokenize_text(self, text: List[str], model):
        # Utilize the model to tokenize text
        if not isinstance(text, list) or not all(isinstance(item, str) for item in 
text):
            raise ValueError("text must be a list of strings")
        vector = combine_strings_to_vector(text, model, True)
        return vector
```

The tokenize_text() mentioned previously uses the combine_strings_to_vector function from word2vec.py, along a few validations to ensure that the data sent for tokenization is of appropriate datatype.

```
    def default_callback(self, vector, note, conn, ):
        # Default callback if no other callback is provided
        try :
            cur = conn.cursor()
            cur.execute("UPDATE notes SET vector = %s WHERE id = %s", (vector, 
note.note_id))
            conn.commit()
        except Exception as e:
            print(f"Error updating note {note.note_id}: {e}")
            conn.rollback()

    def stop(self):
        self.running = False
        self.new_task_event.set()
        self.worker_thread.join()
```

The last two functions are the default callback, which involves updating the corresponding recrods in the Notes table with the new vectorized representation of the note.

Lastly, the stop function shuts down the Tokenization manager.

3.1.11. utils/recentsList.py

Users may want to see history of the recently accessed notes in order to quickly come back and amend the note, if something was incorrectly written. F

3.1.11.1 Implementing a doubly linked list

```
class ListNode:
    def __init__(self, note_id):
        self.note_id = note_id
        self.prev = None
        self.next = None
```

This is the constructor for the ListNode class. It initializes a new node with a given note_id. It also sets the previous and next pointers to None, indicating that the node is initially isolated. It basically creates the building block for the doubly linked list, storing a note identifier and setting up the links needed to connect it to other nodes.

```
class RecentlyAccessedNotes:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0
        self.max_size = 5
```

This is the constructor for the RecentlyAccessedNotes class. It initializes an empty doubly linked list to store recently accessed note ids. It sets the head and tail pointers to None, indicating an empty list, initializes the size to 0, and sets the max_size to 5, limiting the number of stored notes. This sets up the data structure that will manage the recent notes.

```
    def add(self, note_id):
        new_node = ListNode(note_id)
        if not self.head:
            self.head = self.tail = new_node
        else:
            new_node.next = self.head
            self.head.prev = new_node
            self.head = new_node

        if self.size == self.max_size:
            self.remove_last()
        else:
            self.size += 1
```

The function adds a new note_id to the beginning of the recently accessed notes list. It creates a new ListNode and inserts it at the front of the list. If the list was previously empty, the new node becomes both the head and tail. Otherwise, it updates the head and the prev pointer of the old head. If the list reaches its max_size, it calls remove_last() to remove the least recently accessed note. This function maintains the "recently accessed" order.

```
    def remove_last(self):
        if self.tail:
            if self.tail.prev:
                self.tail = self.tail.prev
                self.tail.next = None
            else:
                self.head = self.tail = None
            self.size -= 1
```

remove_last removes the last note in the list, which is pointed at by self.tail. If the list is not empty, it updates the tail pointer to the previous node and sets the new tail's next pointer to None. If the list had only one element, both head and tail become None. It also decrements the size of the list. This function helps maintain the size constraint of the recent notes list, so users are not suggested notes that were accessed months ago.

```
   def remove(self, note_id):
        current = self.head
        while current:
            if current.note_id == note_id:
                if current.prev:
                    current.prev.next = current.next
                if current.next:
                    current.next.prev = current.prev
                if current == self.head:
                    self.head = current.next
                if current == self.tail:
                    self.tail = current.prev
                self.size -= 1
                break
            current = current.next
```

Finally, the above function iterates through the list, searching for the node with the matching note_id. If found, it updates the prev and next pointers of the surrounding nodes to bypass the node being removed. It also handles edge cases where the node to be removed is the head or tail. The list's size is decremented. This ensures that duplicate notes are removed before adding a note back to the front of the list, thus updating its position.

3.1.11.2 Implementing the recent notes manager

```
class RecentNotesManager:
    def __init__(self, capacity=100):
        self.capacity = capacity
        self.user_notes = [None] * capacity

    def _hash(self, uuid_str):
        # hashing algo: sum of ASCII values of the UUID string modulo capacity
        return sum(ord(c) for c in str(uuid_str)) % self.capacity
```

```
            return notes
        return []
```

This class manages a hashmap, which contains key-value pairs, where the key is the user ID, with a simple hashing algorithm applied onto it, and the values are the doubly linked lists of recently accessed note IDs.

Firstly, the __init__() method initializes the manager with a fixed-size hashmap, implemented using a list. The capacity parameter, defaulting to 100, determines the size of this list, which acts as the hash table and matches my maximum estimations of the number of concurrent users of the application at the start.

_hash() is a private method responsible for generating a hash value from a given user id of UUID datatype. It takes the UUID, and then calculates the sum of the ASCII values of each character in that string. This sum is then taken modulo the capacity of the hash table. This modulo operation ensures that the resulting hash value falls within the valid index range of the user_notes list.This is a simple and quick hashing algorithm, which ensures that the index generated falls within the capacity of the table.

```
    def add_note_for_user(self, user_id, note_id):
        index = self._hash(user_id)
        if self.user_notes[index] is None:
            self.user_notes[index] = {}
        if user_id not in self.user_notes[index]:
            self.user_notes[index][user_id] = RecentlyAccessedNotes()
        user_notes = self.user_notes[index][user_id]
        user_notes.remove(note_id)
        user_notes.add(note_id)
```

This function adds a note_id to the recently accessed notes list for a given user_id by calculating the hash index for the user_id using the _hash() method. If the corresponding slot in the user_notes list is None, it initializes an empty dictionary at that index. Then, if the user_id is not already a key in the dictionary at that index, a new RecentlyAccessedNotes doubly linked list object is created and associated with the user_id. This approach handles collisions by using a nested dictionary to store different user's notes that happen to hash to the same index, if such collisions occur

```
    def get_recent_notes_for_user(self, user_id):
        index = self._hash(user_id)
        if self.user_notes[index] and user_id in self.user_notes[index]:
            notes = []
            current = self.user_notes[index][user_id].head
            while current:
                notes.append(current.note_id)
                current = current.next
```

This final function retrieves the list of recently accessed note IDs for a given user_id. It first calculates the hash index for the user_id using the _hash method. It then checks if the corresponding slot in the user_notes list is not None, and if the user_id exists as a key in the dictionary at that index. If both conditions are met, it retrieves the corresponding RecentlyAccessedNotes object . It then iterates through the linked list of notes, starting from the head, and appends each note_id to a notes list. Finally, it returns the list of note IDs. If the user's notes are not found, it returns an empty list. The list will later be used by the modules to fetch note data to display on the dashboard page.

3.1.12. utils/userDeleteGraph.py

3.1.12.1 Setup

Since I am using a relational database with many tables, I need to have a way to delete related tables in a safe manner, so foreign key constraints are not broken.

In order to achieve this task, I first need to give the algorithm an understanding of how the tables are related. For this, using the database diagram from **2.2.2**, I first construct an adjacency matrix.

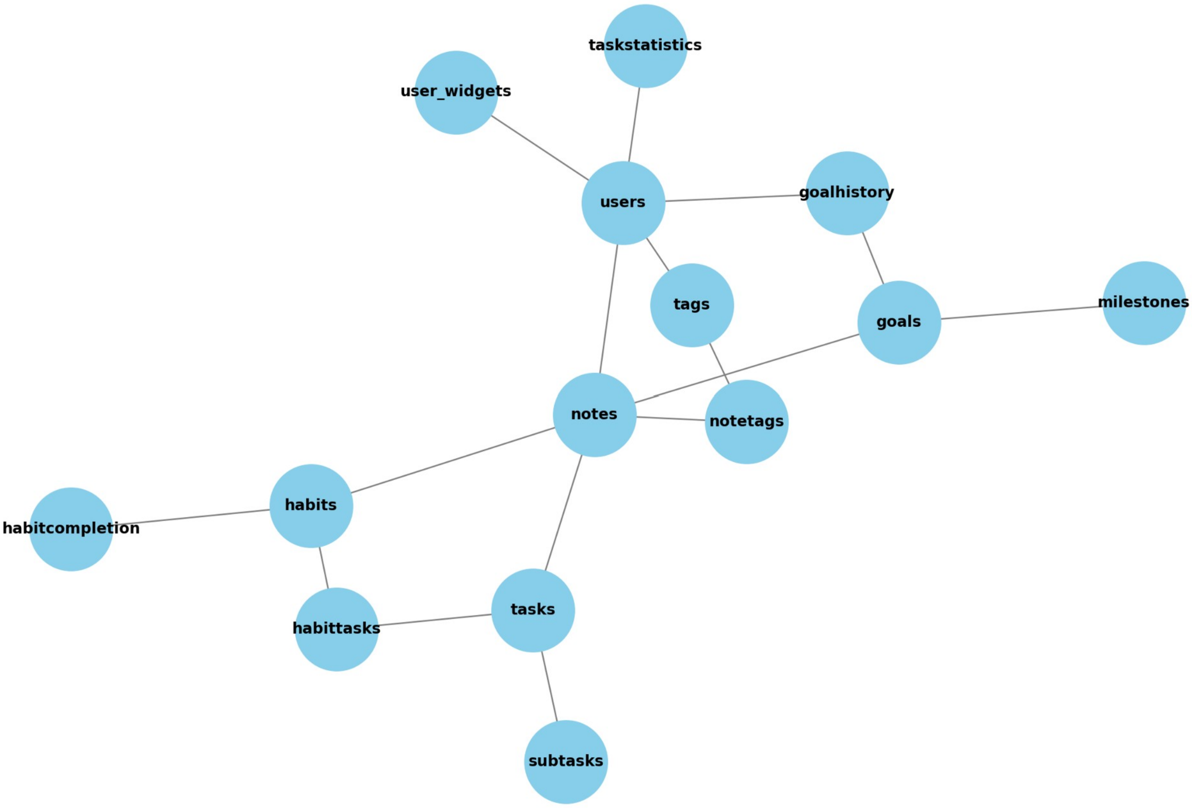
```
from datetime import time
import psycopg2
import os
import sys
from psycopg2 import extras
sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__),
os.path.pardir)))
from config import Config
import psycopg2
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt

# Adjacency matrix
adj_matrix = np.array([
    [0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1],  # users
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],  # taskstatistics
    [1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0],  # notes
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0],  # tags
    [0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0],  # goals
    [1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],  # goalhistory
    [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],  # milestones
    [0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0],  # habits
    [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],  # habitcompletion
    [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0],  # habittasks
    [0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0],  # tasks
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0],  # subtasks
    [0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],  # notetags
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],  # user_widgets
])
```

In the code, the matrix is represented as a 14x14 NumPy array, where each “1” represents a connection between two tables. Using this matrix, I can use a library like networkx to plot the relationship”

```
def draw_graph(adj_matrix):
    G = nx.from_numpy_array(adj_matrix)
    # Relabel nodes
    G = nx.relabel_nodes(G, node_labels)
    # Draw the graph
    plt.figure(figsize=(12, 8))
    pos = nx.spring_layout(G)  # positions for all nodes
    nx.draw(G, pos, with_labels=True, node_size=3000, node_color="skyblue",
font_size=10, font_weight="bold", edge_color="gray")
    plt.title("Graph Visualization from Adjacency Matrix")
    plt.show()
draw_graph(adj_matrix)
```

If this code is ran, the following diagram is created:



As can be see, tables like “notes” or “tasks” cannot have records deleted in them, without ensuring that they are first deleted from their children tables, such as subtasks. Basically, I need an algorithm that will traverse to the deepest points of the graph and remove the tables starting from them. By definition, this is what depth traversal algorithm does (dfs). In my case, this algorithm should work both for user deletion tasks, as well as note deletion tasks. For that, I need to define SQL queries to delete records from each table, as well as label each array in the adjacency matrix for debugging purposes:

```
node_labels = {
    0: "users",
    1: "taskstatistics",
    2: "notes",
    3: "tags",
    4: "goals",
    5: "goalhistory",
    6: "milestones",
    7: "habits",
    8: "habitcompletion",
    9: "habittasks",
    10: "tasks",
    11: "subtasks",
    12: "notetags",
    13: "user_widgets"
}

# Define SQL query templates for deletion
DELETE_QUERIES = {
    0: "DELETE FROM users WHERE id = %s",
    1: "DELETE FROM taskstatistics WHERE user_id = %s",
    2: "DELETE FROM notes WHERE user_id = %s",
    3: "DELETE FROM tags WHERE user_id = %s",
    4: "DELETE FROM goals WHERE note_id IN (SELECT id FROM notes WHERE user_id =
%s)",
    5: "DELETE FROM goalhistory WHERE goal_id IN (SELECT id FROM goals WHERE
note_id IN (SELECT id FROM notes WHERE user_id = %s))",
    6: "DELETE FROM milestones WHERE goal_id IN (SELECT id FROM goals WHERE note_id
IN (SELECT id FROM notes WHERE user_id = %s))",
    7: "DELETE FROM habits WHERE note_id IN (SELECT id FROM notes WHERE user_id =
%s)",
    8: "DELETE FROM habitcompletion WHERE habit_id IN (SELECT id FROM habits WHERE
note_id IN (SELECT id FROM notes WHERE user_id = %s))",
    9: "DELETE FROM habittasks WHERE habit_id IN (SELECT id FROM habits WHERE
note_id IN (SELECT id FROM notes WHERE user_id = %s))",
    10: "DELETE FROM tasks WHERE note_id IN (SELECT id FROM notes WHERE user_id =
%s)",
    11: "DELETE FROM subtasks WHERE task_id IN (SELECT id FROM tasks WHERE note_id
IN (SELECT id FROM notes WHERE user_id = %s))",
    12: "DELETE FROM notetags WHERE note_id IN (SELECT id FROM notes WHERE user_id
= %s)",
    13: "DELETE FROM user_widgets WHERE user_id = %s"
}

# Query templates for note-specific deletion
NOTE_DELETE_QUERIES = {
    2: "DELETE FROM notes WHERE id = %s",
    4: "DELETE FROM goals WHERE note_id = %s",
    5: "DELETE FROM goalhistory WHERE goal_id IN (SELECT id FROM goals WHERE
note_id = %s)",
    6: "DELETE FROM milestones WHERE goal_id IN (SELECT id FROM goals WHERE note_id
= %s)",
    7: "DELETE FROM habits WHERE note_id = %s",
    8: "DELETE FROM habitcompletion WHERE habit_id IN (SELECT id FROM habits WHERE
note_id = %s)",
    9: "DELETE FROM habittasks WHERE habit_id IN (SELECT id FROM habits WHERE
note_id = %s)",
    10: "DELETE FROM tasks WHERE note_id = %s",
    11: "DELETE FROM subtasks WHERE task_id IN (SELECT id FROM tasks WHERE note_id
= %s)",
    12: "DELETE FROM notetags WHERE note_id = %s"
}
```

3.1.12.2 DFS

```
def delete_with_dfs(conn, userId):
    try:
        adjacency_list = {}
        for i in range(len(adj_matrix)):
            adjacency_list[i] = []
            for j in range(len(adj_matrix[i])):
                if adj_matrix[i][j] == 1:
                    adjacency_list[i].append(j)
        print(adjacency_list)
        root_node = 0  # users

        stack = []
        visited = set()

        def dfs(node):  # Depth First Search
            if node not in visited:
                visited.add(node)
                for neighbor in adjacency_list[node]:
                    dfs(neighbor)
                stack.append(node)
        dfs(root_node)
        stack.reverse()

        if (delete_user_data_with_backoff(conn, userId, stack)):
            print("User data deleted successfully.")
            return True
        else:
            return False
    except Exception as e:
        print(f"Error deleting notes: {e}")
        raise
```

This code implements a depth-first search (DFS) algorithm to determine the order in which user-related data should be deleted from a database.

Firstly, the code constructs an adjacency list representation of a graph from an adjacency matrix mentioned previously. The code then initializes a DFS traversal starting from a root_node (the users table).

Dfs function recursively explores the graph. It marks each visited node and then recursively calls itself on each unvisited neighbor. After exploring all neighbors of a node, it appends the node to a stack. This post-order traversal, when the stack is reversed, provides the correct order for deletion.

Finally, the code calls delete_user_data_with_backoff function with the database connection, the user ID, and the correctly ordered stack.

3.1.12.3 Deleting data with backoff

The next two functions are used in user delete tasks and note delete tasks, with the latter being referenced throughout the modules. There, dfs is not used, and the stack was predetermined by me to avoid running the algorithm again and again for such a frequent event. Where the above function is used, is delete_user_data:

```
def delete_user_data_with_backoff(conn, userId, stack, max_retries=3,
backoff_time=1):
    retry_queue = []
    retries = 0

    while stack or retry_queue:
        if not stack and retry_queue:
            stack = retry_queue
            retry_queue = []
            retries += 1
            if retries > max_retries:
                print("Max retries reached. Some entries could not be deleted.")
                return False
            time.sleep(backoff_time)  # Backoff before retrying

        node = stack.pop()
        try:
            with conn.cursor(cursor_factory=psycopg2.extras.DictCursor) as cur:
                if node in DELETE_QUERIES:
                    cur.execute(DELETE_QUERIES[node], (userId,))
        except Exception as e:
            print(f"Error deleting node {node} ({node_labels.get(node,
'unknown')}): {e}. Retrying later.")
            retry_queue.append(node)
            continue
    conn.commit()
    return True
```

This function takes a database connection , a user ID, and a stack of nodes (the deletion order) as input. The function uses a retry_queue to store nodes that failed to delete initially. It iterates through the stack or retry_queue, popping nodes and attempting to delete them using corresponding SQL queries stored in the DELETE_QUERIES dictionary. If a deletion fails, the node is added to the retry_queue, and the process retries after a backoff period. The max_retries parameter limits the number of retry attempts. Finally, the function commits the changes to the database and returns True if all deletions are successful, or False if the maximum number of retries is reached.

```
def delete_notes_with_backoff(conn, noteId, stack, max_retries=3, backoff_time=1):
    retry_queue = []
    retries = 0

    while stack or retry_queue:
        if not stack and retry_queue:
            stack = retry_queue
            retry_queue = []
            retries += 1
            if retries > max_retries:
                print("Max retries reached. Some entries could not be deleted.")
                return False
            time.sleep(backoff_time)  # Backoff before retrying

        node = stack.pop()
        try:
            with conn.cursor(cursor_factory=psycopg2.extras.DictCursor) as cur:
                if node in NOTE_DELETE_QUERIES:
                    cur.execute(NOTE_DELETE_QUERIES[node], (noteId,))
        except Exception as e:
            print(f"Error deleting node {node} ({node_labels.get(node, 
'unknown')}): {e}. Retrying later.")
            retry_queue.append(node)
            continue
    conn.commit()
    return True
```

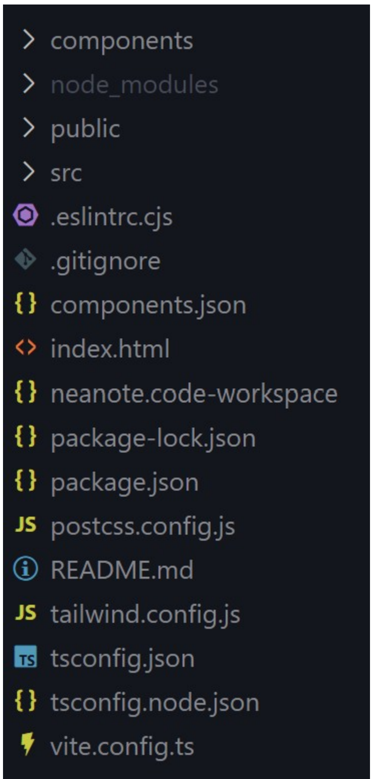
The delete_notes_with_backoff function is almost the same, but it targets note-related data instead of user data. It takes a note ID and a stack of nodes as input, instead of user ID. The SQL queries for note deletion are stored in the NOTE_DELETE_QUERIES dictionary. The function follows the same retry and backoff logic.

3.1.13. Conclusion

This makes up for the backend of my program. We have looked at the APIs that operate with user requests, as well as the utilities helping speed up or simplify processing of notes. However, some files were still left undiscussed, such as goals.py and habits.py, as well as the rest of formsValidation.py. However, I believe that the techniques used there have already been outlined in tasks.py, and their operation will be further visualised when we get to those modules on the frontend.

3.2 Frontend

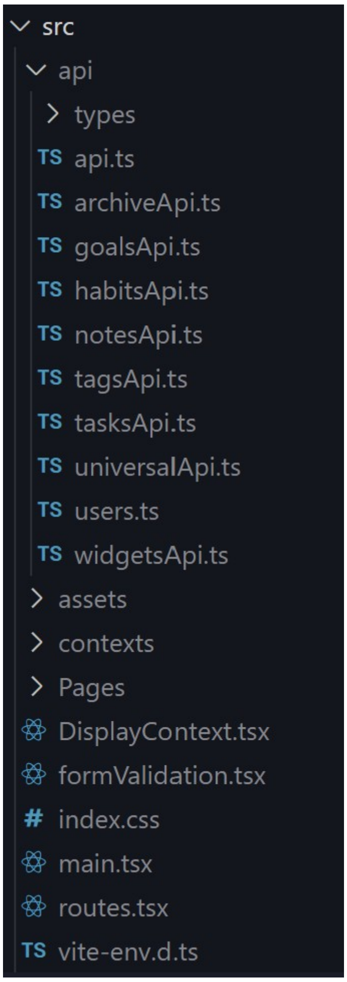
Due to the nature of React, there are many TypeScript, HTML, TSX files containing various code, such as the building elements of components, that are then combined into webpages. In addition to that, each page in my application has several other files containing the logic, predetermined types and backend communication functionality. All in all, I can count up to 100 separate files that the web app is built on. Here is how they are structured in the project's folders:



```
> components
> node_modules
> public
> src
.eslintrc.cjs
.gitignore
components.json
index.html
neanote.code-workspace
package-lock.json
package.json
postcss.config.js
README.md
tailwind.config.js
tsconfig.json
tsconfig.node.json
vite.config.ts
```

The files in the root directory of the frontend are the configuration files for the GIT repository, JavaScript, CSS and and other library configs. In addition, it contains NodeJS package configuration files that can be easily used to install the libraries required to run the web app. I will first skim through the most important files here

The ./components folder contains pre-built react components, such as buttons and labels and their combinations that are widely reused in the application, where some components will be additionally showcased. Generally, most of the files here are UI elements, that may or may not invoke some logic, which is generally stored in other files for separation of concerns. It will not be directly addressed, however, many building bricks of webpages, down to the layout, widget designs,navbars and other menus are here and will be referenced alongside their parent page.



```
v src
  v api
    > types
    TS api.ts
    TS archiveApi.ts
    TS goalsApi.ts
    TS habitsApi.ts
    TS notesApi.ts
    TS tagsApi.ts
    TS tasksApi.ts
    TS universalApi.ts
    TS users.ts
    TS widgetsApi.ts
  > assets
  > contexts
  > Pages
  DisplayContext.tsx
  formValidation.tsx
  # index.css
  main.tsx
  routes.tsx
  TS vite-env.d.ts
```

The final and most important folder is ./src, which, by convention, stores the main React app folder, as well as any React logic. Here, you may find the APIs that are used to communicate with the backend, in addition to data contexts and finally pages, which combine the components, apis, take data from contexts and have some logic on their own to retrieve data and display it to the user.

3.2.1. Setup and configuration

3.2.1.1 ./index.html

The first file a browser reads when the website is loaded is index.html, which contains the title of the website, its logo, any <meta> tags that describe the contents of the website for SEO (search engine optimization), along with information that helps properly display the website using Open Graph protocol so the website can be easily embedded into any social media post. Moreover, this file allows me to mount the actual React application into the body of the webpage:

```html
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <link rel="icon" type="image/svg+xml" href="/logo.svg" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />

    <!-- Additional meta tags -->
    <meta name="description" content="Neanote - Your ultimate note-taking app for organizing and managing your tasks and ideas." />
    <meta name="keywords" content="neanote, note-taking, task management, organization, productivity" />
    <meta name="author" content="Roman Sasinovich" />

    <!-- Open Graph meta tags for social media sharing -->
    <meta property="og:title" content="Neanote - Note-taking and Task Management" />
    <meta property="og:description" content="Neanote - Your ultimate note-taking app for organizing and managing your tasks and ideas." />
    <meta property="og:image" content="/logo.svg" />
    <meta property="og:url" content="https://www.neanote.com" />
    <meta property="og:type" content="website" />
    <title>neanote</title>
  </head>
  <body>
    <div id="root"></div>
    <script type="module" src="/src/main.tsx"></script>
  </body>
</html>
```

3.2.1.2 ./package.json

This file contains the name and version of any library used so NPM (Node Package Manager) can quickly retrieve the correct project-specific version from its repository if these files need to be reinstalled.

```
{
  "name": "neanote",
  "private": true,
  "version": "0.0.0",
  "type": "module",
  "scripts": {
    "dev": "vite",
    "build": "tsc && vite build",
    "lint": "eslint . --ext ts,tsx --report-unused-disable-directives --max-
warnings 0",
    "preview": "vite preview"
  },
  "dependencies": {
    "@dnd-kit/core": "^6.1.0",
    "@dnd-kit/sortable": "^8.0.0",
    "@hookform/resolvers": "^3.6.0",
    "@radix-ui/react-accordion": "^1.1.2",
    "@radix-ui/react-alert-dialog": "^1.0.5",
    "@radix-ui/react-aspect-ratio": "^1.0.3",
    "@radix-ui/react-avatar": "^1.0.4",
    "@radix-ui/react-checkbox": "^1.0.4",
    "@radix-ui/react-collapsible": "^1.0.3",
    "@radix-ui/react-context-menu": "^2.1.5",
    "@radix-ui/react-dialog": "^1.0.5",
    "@radix-ui/react-dropdown-menu": "^2.0.6",
    "@radix-ui/react-hover-card": "^1.0.7",
    "@radix-ui/react-label": "^2.0.2",
    "@radix-ui/react-menubar": "^1.0.4",
    "@radix-ui/react-navigation-menu": "^1.1.4",
    "@radix-ui/react-popover": "^1.0.7",
    "@radix-ui/react-progress": "^1.0.3",
    "@radix-ui/react-radio-group": "^1.1.3",
    "@radix-ui/react-scroll-area": "^1.0.5",
    "@radix-ui/react-select": "^2.0.0",
    "@radix-ui/react-separator": "^1.0.3",
    "@radix-ui/react-slider": "^1.1.2",
    "@radix-ui/react-slot": "^1.0.2",
    "@radix-ui/react-switch": "^1.0.3",
    "@radix-ui/react-tabs": "^1.0.4",
    "@radix-ui/react-toast": "^1.1.5",
    "@radix-ui/react-toggle": "^1.0.3",
    "@radix-ui/react-toggle-group": "^1.0.4",
    "@radix-ui/react-tooltip": "^1.0.7",
    "@tanstack/react-query": "^5.51.3",
    "axios": "^1.7.2",
    "class-variance-authority": "^0.7.0",
    "clsx": "^2.1.1",
    "cmdk": "^1.0.0",
    "crypto": "^1.0.1",
    "date-fns": "^3.6.0",
    "dnd-kit": "^0.0.2",
    "embla-carousel-react": "^8.1.3",
    "immer": "^10.1.1",
    "input-otp": "^1.2.4",
    "js-cookie": "^3.0.5",
    "jwt-decode": "^4.0.0",
    "lucide-react": "^0.390.0",
    "next-themes": "^0.3.0",
    "react": "^18.2.0",
    "react-colorful": "^5.6.1",
    "react-day-picker": "^8.10.1",
    "react-dom": "^18.2.0",
    "react-hook-form": "^7.51.5",
    "react-icons": "^5.2.1",
    "react-resizable-panels": "^2.0.19",
    "react-router-dom": "^6.23.1",
    "react-toastify": "^10.0.5",
    "recharts": "^2.15.0",
    "sonner": "^1.5.0",
    "tailwind-merge": "^2.3.0",
    "tailwindcss-animate": "^1.0.7",
    "uuid": "^10.0.0",
    "vaul": "^0.9.1",
    "zod": "^3.23.8",
    "zustand": "^4.5.2"
  },
  "devDependencies": {
    "@types/node": "^20.14.2",
    "@types/react": "^18.2.66",
    "@types/react-dom": "^18.2.22",
    "@typescript-eslint/eslint-plugin": "^7.2.0",
    "@typescript-eslint/parser": "^7.2.0",
    "@vitejs/plugin-react-swc": "^3.5.0",
    "autoprefixer": "^10.4.19",
    "eslint": "^8.57.0",
    "eslint-plugin-react-hooks": "^4.6.0",
    "eslint-plugin-react-refresh": "^0.4.6",
    "postcss": "^8.4.38",
    "tailwindcss": "^3.4.4",
    "typescript": "^5.2.2",
    "vite": "^5.2.0"
  }
}
```

3.2.1.2.1 Libraries used

As can be seen in this file, there are several key libraries being used, that I would like to outline:

- dnd-kit: This is a library that makes adding drag-and-drop functionality easier

- shadcn: Despite not being a node package, UI components from this library are primarily used throughought the app, since they are customizable, easy to use and have several features such as theming prebuilt, which allows me to create and apply new application themes if I need such. Read more
- radix-ui: This is another component library that shadcn is primarily built on. It is not directly used in the application.
- Axios is a promise-based HTTP Client. It allows me to make http requests from nodejs and helps with JSON data and error handling.
- Crypto is a cryptography module, providing functionality for encryption, as well as TypeScript types for UUID datatype.
- Zustand and Immer are the two libraries that provide the basic necessities for complex state management in React and allow me to work with immutable state conveniently. Most of the frontend logic is written using Zustand states, instead of the built-in React states, letting me write more complex functionality more easily.
- React-icons is a massive open-source icon library that provided icons to many components in the app.
- React-router: This library adds routing functionality to my React app
- zod is a TypeScript schema validation library and is used to validate and sanitize user input before it is sent to the backend as an additional layer of security.
- Miscellaneous: Other libraries like vaul, sonner, etc are often dependencies of bigger libraries such as radix-ui and do not need to be mentioned additionally.

3.2.2. ./src

3.2.2.1 ./src/main.tsx

This file is the backbone of the whole program. It is used to mount the React app to the webpage, while also allowing me to wrap the app with context providers for key functionality:

```
import * as React from "react";
import * as ReactDOM from "react-dom/client";
import { ToastContainer } from "react-toastify";
import 'react-toastify/dist/ReactToastify.css';
import { ThemeProvider } from '../components/providers/theme-provider.tsx';
import './index.css';
import AppRouter from "./routes.tsx";
import { ScreenSizeProvider } from "./DisplayContext.tsx";

const App = () => (
  <ThemeProvider defaultTheme="light" storageKey="vite-ui-theme">
   <ScreenSizeProvider>
```

```
    <AppRouter/>
    <ToastContainer/>
   </ScreenSizeProvider>
  </ThemeProvider>
);

//mount the app
ReactDOM.createRoot(document.getElementById('root')!).render(
 <React.StrictMode>
   <App />
 </React.StrictMode>,
)
```

Now let’s go over each provider and explain their purpose.

3.2.2.1.1 Providers

In React, a context provider is like a shared storage space for data that you want to make accessible to multiple components in your application, without having to pass that data as props through every level of the component tree. It allows to manage global application state without engaging in prop drilling (i.e. passing same information through many and many components)

3.2.2.1.2 ./components/providers/ThemeProvider.tsx

```
import React from 'react'
import { createContext, useContext, useEffect, useState } from "react"
type Theme = "dark" | "light" | "system"

type ThemeProviderProps = {
 children: React.ReactNode
 defaultTheme?: Theme
 storageKey?: string
}

type ThemeProviderState = {
 theme: Theme
 setTheme: (theme: Theme) => void
}

const initialState: ThemeProviderState = {
 theme: "system",
 setTheme: () => null,
}

const ThemeProviderContext = createContext<ThemeProviderState>(initialState)

export function ThemeProvider({
 children,
 defaultTheme = "light",
 storageKey = "vite-ui-theme",
 ...props
}: ThemeProviderProps) {
 const [theme, setTheme] = useState<Theme>(
  () => (localStorage.getItem(storageKey) as Theme) || defaultTheme
 )

 useEffect(() => {
  const root = window.document.documentElement

  root.classList.remove("light", "dark")

  if (theme === "system") {
   const systemTheme = window.matchMedia("(prefers-color-scheme: dark)")
    .matches
    ? "dark"
    : "light"

   root.classList.add(systemTheme)
   return
  }

  root.classList.add(theme)
 }, [theme])

 const value = {
  theme,
  setTheme: (theme: Theme) => {
   localStorage.setItem(storageKey, theme)
   setTheme(theme)
  },
 }

 return (
  <ThemeProviderContext.Provider {...props} value={value}>
   {children}
  </ThemeProviderContext.Provider>
 )
}

export const useTheme = () => {
 const context = useContext(ThemeProviderContext)

 if (context === undefined)
  throw new Error("useTheme must be used within a ThemeProvider")

 return context
}
```

This code defines a React component called ThemeProvider that allows you to manage and apply a theme (dark, light, or system-defined) to your application. It uses the useState hook to store the current theme, and the useEffect hook to update the document's root element's class based on the selected theme. It also provides a useTheme hook that allows other components to easily access and utilize the current theme and its setter function. The ThemeProvider component also stores the selected theme in local storage using the storageKey prop, so the user does not need to change their theme every time they reopen the page.

3.2.2.1.3 ./components/providers/DisplayContexxt.tsx

This provider adds event listeners which dynamically determine the current screen size (small, medium, or large) based on the current window width. The current screen size is stored using a useState hook, while the useEffect hook handles window rezise events, allowing me to calculate whether certain elements of the UI will be compressed or not at the current screen width.

```
import React, { createContext, useContext, useState, useEffect, ReactNode } from
'react';

interface ScreenSizeContextProps { //interface to ensure type safety
  screenSize: 'small' | 'medium' | 'large';
  isDateCollapsed: boolean;
  isTagCompressed: boolean;
}

const ScreenSizeContext = createContext<ScreenSizeContextProps |
undefined>(undefined);

const ScreenSizeProvider = ({ children }: { children: ReactNode }) => {
  const [screenSize, setScreenSize] = useState<'small' | 'medium' |
'large'>('large');

  useEffect(() => {
    const handleResize = () => {
      if (window.innerWidth < 650) {
        setScreenSize('small');
      } else if (window.innerWidth >= 650 && window.innerWidth < 1024) {
        setScreenSize('medium');
      } else {
        setScreenSize('large');
      }
    };

    // Set initial size
    handleResize();

    window.addEventListener('resize', handleResize);
    return () => window.removeEventListener('resize', handleResize);
  }, []);

  const isDateCollapsed = screenSize === 'small';
  const isTagCompressed = screenSize !== 'large';

  return (
    <ScreenSizeContext.Provider value={{ screenSize, isDateCollapsed,
isTagCompressed }}>
      {children}
    </ScreenSizeContext.Provider>
  );
};

const useScreenSize = () => {
  const context = useContext(ScreenSizeContext);
  if (context === undefined) {
    throw new Error('useScreenSize must be used within a ScreenSizeProvider');
  }
  return context;
};

export { ScreenSizeProvider, useScreenSize };
```

3.2.2.2 ./src/routes.tsx

The two components passed directly in the App component are AppRouter and ToastContainer. The latter is just a part of the react-toastify library and is used to show small alerts(toasts) to provide feedback to user actions. Meanwhile, AppRouter utilizes the react-router-dom library to render different components for different frontend endpoints.

```
import React from 'react';
import Layout from "../components/Layout/Layout";

import {
 BrowserRouter as Router, Routes, Route,
 createBrowserRouter
} from "react-router-dom";
import Dashboard from "./Pages/Dashboard/Dashboard.tsx";
import Login from "./Pages/Login/Login.tsx";
import Register from "./Pages/Register/Register.tsx";
import withTokenCheck from '../components/providers/token-check.tsx';
import Landing from './Pages/Landing/Landing.tsx';
import Notes from './Pages/Tasks copy/Notes.tsx';
import Tasks from './Pages/Tasks/Tasks.tsx';
import Tags from './Pages/Tags/Tags.tsx';
import EditTasks from './Pages/Tasks/EditTasks.tsx';
import Habits from './Pages/Habits/Habits.tsx';
import EditHabits from './Pages/Habits/EditHabits.tsx';
import Goals from './Pages/Goals/Goals.tsx';
import EditGoals from './Pages/Goals/EditGoals.tsx';
import CreateGoal from './Pages/Goals/CreateGoal.tsx';
import CreateTask from './Pages/Tasks/CreateTask.tsx';
import CreateHabits from './Pages/Habits/CreateHabits.tsx';
import Archive from './Pages/Archive/Archive.tsx';
import Account from './Pages/Account/Account.tsx';
import CreateNote from './Pages/Tasks copy/CreateNote.tsx';
import EditNotes from './Pages/Tasks copy/EditNotes.tsx';
import Calendar from './Pages/Calendar/Calendar.tsx';

const CheckedLayout = withTokenCheck(Layout);

const AppRouter = () => (

 <Router>
  <Routes>
   <Route path="/" element={<CheckedLayout />}>
    <Route index element={<Dashboard />} />
    <Route path="account" element={<Account/>} />
    <Route path="calendar" element={<Calendar/>} />

    <Route path="notes" element={<Notes/>} />
    <Route path="notes/edit" element={<EditNotes/>} />
    <Route path="notes/create" element={<CreateNote/>} />

    <Route path="tasks" element={<Tasks/>} />
    <Route path="tasks/edit" element={<EditTasks/>} />
    <Route path="tasks/create" element={<CreateTask/>} />

    <Route path="habits" element={<Habits/>} />
    <Route path="habits/edit" element={<EditHabits/>} />
    <Route path="habits/create" element={<CreateHabits/>} />

    <Route path="goals" element={<Goals/>} />
    <Route path="goals/edit" element={<EditGoals/>} />
    <Route path="goals/create" element={<CreateGoal/>} />

    <Route path="archive" element={<Archive/>} />
    <Route path="tags" element={<Tags/>} />
   </Route>
   <Route path="login" element={<Login />} />
   <Route path="register" element={<Register />} />
   <Route path="get-started" element={<Landing/>} />
  </Routes>
 </Router>
);
export default AppRouter
```

In this file, all the endpoints of the website are defined, along with the React components that are rendered on them. However, before the user can access most of the pages, they must be authenticated.

3.2.2.2.1 ./components/providers/token-check.tsx

As can be seen in the previous file, most routes are encapsulated in another <Route/> element, which renders CheckedLayout before any actual webpage. This is because any webpage rendered on the website is actually rendered inside the CheckedLayout component, which checks if the user has a JWT authentication token stored in cookies of their web browser. If the user does not have such a token, they are instantly rerouted to the get-started page.

```
import Cookies from 'js-cookie';

import React from 'react';
import { useNavigate } from 'react-router-dom';

const withTokenCheck = (WrappedComponent) => {
 return (props) => {
  const navigate = useNavigate();

  React.useEffect(() => {
   if (!Cookies.get('token')) {
    navigate('/get-started');
   }
  }, [navigate]);

  return <WrappedComponent {...props} />;
 };
};

export default withTokenCheck;
```

3.2.2.3 ./src/formValidation.tsx

Furthermore, the directory contains a validation file where all the validation rules for notes, emails and passwords. As stated previously, a validation library named “zod” is used here.

Firstly, some basic validation rules are defined that can be reused later. Another benefit of predefining such rules is that by changing a single line of code, I can change the behaviour of specific fields in the whole programme

```
import { zodResolver } from "@hookform/resolvers/zod";
import { z } from "zod";

// Reusable validation rules
const usernameSchema = z.string().min(4, {
  message: "Username must be at least 4 characters.",
});
```

```
const passwordSchema = z.string().min(6, {
  message: "Password must be at least 6 characters.",
}).regex(
  /^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[@$!%*?&])[A-Za-z\d@$!%*?&]{8,}$/,
  "Password must contain at least one uppercase letter, one lowercase letter, one
digit, and one special character."
);

const emailSchema = z.string().regex(/^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]
{2,}$/ , {
  message: "Invalid email address.",
});

const uuidSchema = z.string().uuid();

const titleSchema = z.string().min(1, "Title is required").max(100, "Title cannot
exceed 100 characters");

const contentSchema = z.string().max(1000, "Content cannot exceed 1000
characters");

const descriptionSchema = z.string().min(1, "Description is required");

const completedSchema = z.boolean();
```

These reusable constants include zod schemas for: username, password, email, any kind of ID, and note fields like title, content, description,etc. Primarily, the schemas define the minimum number of characters required for a field to be considered valid, the datatype of the input. Additionally, basic error messages are predefined. However, some schemas like the email and password require a more customized approach. As a result, regular expressions are used to validate user inputs. Regex is supported by the zod library, allowing to seamlessly integrate it into the application.
Here is an example of how the emailSchema regular expression works:

1. ^[a-zA-Z0-9._%+-]+: Matches one or more alphanumeric characters, dots, underscores, percent signs, plus signs, or hyphens at the beginning of the string .

2. @: Matches the "@" symbol.

3. [a-zA-Z0-9.-]+: Matches one or more alphanumeric characters, dots, or hyphens (the domain part of the email).

4. \.: Matches a dot.

5. [a-zA-Z]{2,}$: Matches two or more alphabetic characters at the end of the string (the top-level domain).

6. ^ and $: ensure the entire string matches the pattern, not just a substring.

Now that we have the building blocks defined, they can be combined with other use case-specific inline schemas to create validation rule blocks, that contain multiple rules for various fields:

```
/ Main schemas
export const registerFormSchema = z.object({
  username: usernameSchema,
  password: passwordSchema,
  email: emailSchema,
});

export const loginFormSchema = z.object({
  username: usernameSchema,
  password: passwordSchema,
});

export const SubtaskSchema = z.object({
  subtask_id: z.number(),
  description: descriptionSchema,
  completed: completedSchema,
});

export const TaskSchema = z.object({
  taskid: uuidSchema,
  noteid: uuidSchema,
  title: titleSchema,
  content: contentSchema,
  subtasks: z.array(
    z.object({
      subtaskid: uuidSchema,
      description: descriptionSchema.max(500, "Subtask description cannot exceed
500 characters"),
      completed: completedSchema,
      index: z.number(),
      isNew: z.boolean().optional(),
    })
  ),
  due_date: z.date().optional(),
  completed: completedSchema,
});

export const NoteSchema = z.object({
  noteid: uuidSchema,
  title: titleSchema,
  tags: z.array(uuidSchema), // tag ids
  content: contentSchema,
});

export const UserSettingsSchema = z.object({
  username: usernameSchema.optional(),
  email: emailSchema.optional(),
});

export const PasswordSchema = z.object({
  password: passwordSchema,
  newpassword: passwordSchema,
});

export const GoalSchema = z.object({
  goalid: uuidSchema,
  noteid: uuidSchema,
  title: titleSchema,
  content: contentSchema,
  milestones: z.array(
    z.object({
      milestoneid: uuidSchema,
      description: descriptionSchema.max(500, "Milestone description cannot exceed
500 characters"),
      completed: completedSchema,
      index: z.number(),
      isNew: z.boolean().optional(),
    })
  ),
  due_date: z.date().optional(),
});

export const HabitSchema = z.object({
  habitid: uuidSchema,
  noteid: uuidSchema,
  title: titleSchema,
  content: contentSchema,
  reminder: z.object({
    reminder_time: z.string().min(4, "Reminder time is required"),
    repetition: z.string().min(1, "Repetition is required"),
  }),
  streak: z.number(),
  completed_today: completedSchema,
});
```

As can be seen from this code, each frontend schema matches the backend validation rules, so the user may be able to get real-time feedback if some field they entered does not satisfy the regex pattern or other rules.

3.2.3. ./src/api

The connecting point between the frontend and the backend can be considered to lay in this folder. It contains files and methods that take data passed to them, construct it and send it to an appropriate backend endpoints with a corresponding REST method.

The frontend API uses the axios library to simplify and speed up frontend-backend interaction. The frontend API manager is constructed in the following file:

3.2.3.1 ./src/api/api.ts

```
import axios from 'axios';
import Cookies from 'js-cookie';

let a = axios.create({
      baseURL: "http://localhost:5000",
      withCredentials: true,
      headers: {
        'Authorization': `Bearer ${Cookies.get('token')}`
    }
});

a.interceptors.request.use((request) => {
      //if there is no internet connection
      if (!navigator.onLine) {
            throw new Error('No internet connection');
      }

      //if user doesnt have a token
      if ((request.url !== '/api/login' && request.url !== '/api/register') && !
Cookies.get('token')) {
            window.location.href = '/get-started';
            throw new Error('No token');
      }

       //Add the JWT token to the headers
       request.headers['Authorization'] = `Bearer ${Cookies.get('token')}`;

      return request;
});


export default a
```

let a = axios.create({...});: Creates a customized version of the axios instance. We're setting some default settings for all requests made using this instance.

BaseURL sets the base URL for all requests. So, if we ask for /api/users, axios will actually request http://localhost:5000/api/users.

withCredentials: true: This tells axios to include cookies in the requests. This is important for authentication, as it allows the server to recognize the user's identity.

a.interceptors.request.use((request) => {...});: This sets up an interceptor, which is a function that gets called before every request is sent. It allows us to modify or check the request before it goes out. It first detects whether the user is online, then, another if statement runs, which checks if the user is trying to access a protected resource (any resource other than login or register) without a valid JWT token. If so, it redirects them to the /get-started page and throws an error.

Finally, the axios instance is exported to be used in subsequent Apis. For convenience, most APIs will be omitted, such as habitsApi, tasksApi, archiveApi, notesApi, etc., because these files do not contain much logic and the code is repetetive. Rather, they are used as stencils to collect necessary data and send it to a correct endpoint with appropriate parameters, and catch and return a response or error message, related to the note type or function at hand. The following functions are not standalone, but they are invoked by other functions that will be discussed later.

3.2.3.2 ./src/api/users.ts

```
import { showToast } from "../../components/Toast";
import a from "./api";
import { UserGetResponse, UserLoginResponse } from "./types/userTypes";
```

```
const users = {

    login: async (body) => {
        try {
            let response = await a.post<UserLoginResponse>(`/api/login`, body);

            if (response.status === 200) {
                showToast('s', 'Login successful');
            } else {
                showToast('e', `There was an error logging in: $
{response.data.message}`)
            }

            return response.data;
        } catch (error) {
            showToast('e', error);
            return false;
        }
    },
    register: async (body) => {
        try {
            let response = await a.post(`/api/register`, body);

            if (response.status === 200) {
                showToast('s', 'User has been registered created successfully');
            } else {
                showToast('e', 'There was an error registering the user')
            }

            return response.data;
    } catch (error) {
        showToast('e', error);
        return false;
    }
},
    getUser: async () => {
        try {
            let response = await a.get<UserGetResponse>(`/api/user`);

            if (response.status === 200) {
                return response.data.data;
            }
            else {
                showToast('e', `There was an error getting the user: $
{response.data.message}`)
                return false;
            }
        } catch (error) {
            showToast('e', error);
            return false;
        }
    },

    updateUserDetails: async (body) => {
        try {
            let response = await a.put(`/api/user`, body);
            if (response.status === 200) {
                showToast('s', 'User data has been updated successfully');
            } else {
                showToast('e', `There was an error updating the user: $
{response.data.message}`)
            }

            return response.data;
        } catch (error) {
            showToast('e', error);
            return false;
        }
    },
    changePassword: async (password, newpassword) => {
        try {
            let response = await a.put(`/api/user/password`,{ password, newpassword
});
            let success = false

            if (response.status === 200) {
                showToast('s', 'Password has been updated successfully');
                success = true
            } else {
                showToast('e', `There was an error updating the password: $
{response.data.message}`)
            }

            return {data:response.data, success:success};
        } catch (error) {
            showToast('e', error);
            return false;
        }
    },

    deleteUser: async (password) => {
        try {
            let response = await a.delete(`/api/user/delete`, {data:{password}});

            if (response.status === 200) {
                showToast('s', 'User has been deleted successfully');
                return {success:true}
            } else {
                showToast('e', `There was an error deleting the user: $
{response.data.message}`)
                return {success:false}
            }
        } catch (error) {
            showToast('e', error);
            return false;
        }
    }
}


export default users
```

This file contains user-related API interactions . It firstly defines a users object that serves as a module, grouping logically related functions. Each function within this object handles a specific user action, such as login, registration, or profile updates, by making HTTP requests to the backend. The use of async/await simplifies the handling of asynchronous operations, making the code more readable and maintainable.

Error handling is implemented using try...catch blocks, and any errors are displayed to the user in form of a toast, is provided through the showToast function, that was previously discussed. The file also demonstrates the use of generics in TypeScript (a.post<UserLoginResponse>), which allows for type safety when working with API responses. Essentially, each function takes the provided data and sends it through an appropriate REST method (put, update, delete, etc.) to the backend endpoint where any other operations are performed on the data.

3.2.3.3 ./src/api/tagsApi.ts

```
import { UUID } from "crypto";
import { showToast } from "../../components/Toast";
import a from "./api";
import { TagCreateResponse, TagResponse } from "./types/tagTypes";

const tagsApi = {
    create : async (name: string, color:string) => {
        try {
            let response = await a.post<TagCreateResponse>(`/api/tags/create`, {
                name,
                color
            })
            let success = false
            if (response.status === 200) {
                showToast('s', 'Tag has been created successfully');
                success = true;
            } else {
                showToast('e', 'There was an error creating the tag')
            }

            return {success: success, tagid: response.data.data.id};
        } catch (error) {
            showToast('e', error);
            return false;
        }
    },
    add : async (noteid: UUID, tagid: UUID) => {
        try {
            let response = await a.post(`/api/tags/add`, {
                noteid,
                tagid
            });

            if (response.status === 200) {
                return response.data; }
        } catch (error) {
            showToast('e', error);
            return false;
        }
    },

    getTags : async (noteId: UUID) => {
        try {
            let response = await a.get<TagResponse>(`/api/tags/${noteId}`);
            return response.data;
        } catch (error) {
            showToast('e', error);
            return false;
        }
    },

    getAll : async () => {
        try {
            let response = await a.get<TagResponse>(`/api/tags/`);
            return response.data;
        } catch (error) {
            showToast('e', error);
            return false;
        }
    },

    delete : async (tagid: UUID) => {
        try {
            let response = await a.put(`/api/tags/delete`, {tagid})
            let success = false
            if (response.status === 200) {
                showToast('s', 'Tag has been deleted successfully');
                success = true;
            } else {
                showToast('e', 'There was an error deleting the tag')
            }
            return {success: success};
        } catch (error) {
            showToast('e', error);
            return false;
        }
    },

    edit : async (tagid: UUID, name: string, color: string) => {
        try {
            let response = await a.put(`/api/tags/edit`, {
                tagid,
                name,
                color
            });

            if (response.status === 200) {
                showToast('s', 'Tag has been updated successfully');
            } else {
                showToast('e', 'There was an error updating the tag')
            }

            return response.data;
        } catch (error) {
            showToast('e', error);
            return false;
        }
    }


}
export default tagsApi
```

The above api essentially differs by having different tag-related functions, along with replacing the endpoints and error handling messages to tag related ones.

3.2.3.4 ./src/api/goalsApi.ts

Now this API corresponds to the goals note type, which is one of the four note types in my application.

```
import axios from "axios";
import { showToast } from "../../components/Toast";
import a from "./api";
import {  GoalCreateResponse, GoalResponse, GoalsPreview } from
"./types/goalTypes";
import { UUID } from "crypto";

const goalsApi = {
    create: async (title, selectedTagIds, content, due_date, milestones) => {
        try {
            let response = await a.post<GoalCreateResponse>(`/api/goals/create`, {
                title,
                tags: selectedTagIds,
                content,
                due_date,
                milestones: milestones.map((milestone) => {
                    const { description, completed, index } = milestone;
                    return { description, completed, index };
                }),
            });

            if (response.status === 200) {
                return { success: true, data: response.data.data };
            } else {
                return { success: false, message: 'An error occurred while creating
the goal' };
            }
        } catch (error) {
            const errorMessage = axios.isAxiosError(error)
                ? error.response?.data?.message || 'An error occurred while
creating the goal'
                : 'An unexpected error occurred';
            return { success: false, message: errorMessage };
        }
    },

    getGoalPreviews: async (page: number) => {
        try {
            const response = await a.get<GoalsPreview>(`/api/goals/previews?page=$
{page}`);
```

```
            return { success: true, data: response.data.goals, nextPage: 
response.data.pagination.nextPage, page: response.data.pagination.page }; //total, 
perPage
        } catch (error) {
            const errorMessage = axios.isAxiosError(error)
                ? error.response?.data?.message || 'An error occurred while 
fetching goal previews'
                : 'An unexpected error occurred';
            return { success: false, message: errorMessage };
        }
    },

    getGoal: async (noteId: string) => {
        try {
            const response = await a.get<GoalResponse>("/api/goal", { params: 
{ noteId } });
            return { success: true, data: response.data.goal };
        } catch (error) {
            const errorMessage = axios.isAxiosError(error)
                ? error.response?.data?.message || 'An error occurred while 
fetching the goal previews'
                : 'An unexpected error occurred';
            return { success: false, message: errorMessage };
        }
    },

    completeMilestone: async (goalid: UUID, milestoneid: UUID) => {
        try {
            const response = await a.put(`/api/goals/milestone/complete`, { goalid, 
milestoneid });
            return response.status === 200
                ? { success: true }
                : { success: false, message: 'An error occurred while completing 
the milestone' };
        } catch (error) {
            const errorMessage = axios.isAxiosError(error)
                ? error.response?.data?.message || 'An error occurred while 
completing the milestone'
                : 'An unexpected error occurred';
            return { success: false, message: errorMessage };
        }
    },

    update: async (goalUpdates: {}) => {
        try {
            const response = await a.put(`/api/goals/update`, goalUpdates);

            if (response.status === 200) {
                return { success: true };
            } else {
                return { success: false, message: 'There was an error updating the 
goal' };
            }
        } catch (error) {
            const errorMessage = axios.isAxiosError(error)
                ? error.response?.data?.message || 'An error occurred while 
updating goals'
```

```
                : 'An unexpected error occurred';
            return { success: false, message: errorMessage };
        }
    },

    delete: async (goalid: UUID, noteid: UUID) => {
        try {
            const response = await a.delete(`/api/goals/delete`, { params:
{ goalid, noteid } });

            if (response.status === 200) {
                return { success: true };
            } else {
                return { success: false, message: 'There was an error deleting the
goal' };
            }
        } catch (error) {
            const errorMessage = axios.isAxiosError(error)
                ? error.response?.data?.message || 'Failed to delete goal'
                : 'An unexpected error occurred';
            return { success: false, message: errorMessage };
        }
    }
}

export default goalsApi;
```

It may look more complicated than previously mentioned APIs, but it just has improved error handling by allowing a backend error to be returned, apart from a predefined one, and unpacks or replaces some query parameters to appropriate ones, such as in the create function:

```
            let response = await a.post<GoalCreateResponse>(`/api/goals/create`, {
                title,
                tags: selectedTagIds,
                content,
                due_date,
                milestones: milestones.map((milestone) => {
                    const { description, completed, index } = milestone;
                    return { description, completed, index };
                }),
```

For example, the milestones are converted into dictionaties using loops and passed into the request, instead of being objects.

3.2.3.5 ./src/api/universalApi.ts

```
import { showToast } from "../../components/Toast";
import a from "./api";
import { UniversalType } from "./types/ArchiveTypes";

export interface SearchResponse {
    data: UniversalType[];
    pagination : {
      nextPage: number | null;
      perPage: number;
      total: number;
      page: number;
    };
    message: string;
}

interface RecentsResponse {
    data: UniversalType[];
    message: string;
}

interface SummarizeResponse {
    data:string
    message: string;
}

export const universalApi = {
    search : async (searchQuery:string, searchMode: string, pageParam:number) => {
        try {
            let response = await a.get<SearchResponse>(`/api/search`, {params:
{searchQuery, searchMode, pageParam}});
            return { data: response.data.data, pagination:{nextPage:
response.data.pagination.nextPage, page: response.data.pagination.page} };
        } catch (error) {
            showToast('e', error);
            return null;
        }
    },

    getRecents : async () => {
        try {
            let response = await a.get<RecentsResponse>('/api/recents');
            let success = false
            if (response.status === 200)
                success = true
            return{data: response.data, success: success};
        } catch (error) {
            return false;
        }
    },

    getDue : async(startDate,endDate) => {
        try {
            let response = await a.get('/api/calendar', {params:{startDate,
endDate}});
            let success = false
            if (response.status === 200)
                success = true
            return{data: response.data, success: success};
        } catch (error) {
            return false;
    }},

    summarize : async (text:string) => {
        try {
            let response = await a.post<SummarizeResponse>('/api/summarize',
{text});
```

```
            let success = false
            if (response.status === 200)
                success = true
            return{data: response.data, success: success};
        } catch (error) {
            return false;
        }

    }


}
```

Finally, the universalApi handles requests and responses for universal endpoints, such as search, summary and recents. Since there are not that many functions here, this file also defines the interfaces that will be used for type checking the response, ensuring that the data will be handled correctly.

3.2.4. ./src/api/types

3.2.4.1 What is an interface?

In TypeScript, an interface is essentially a contract that defines the structure of an object. It specifies the names and types of properties and methods that an object must have. It works as a blueprint, ensuring that any object passed through that interface possesses the required characteristics. Interfaces are primarily used for type checking, helping to catch errors early and improve code clarity and maintainability. Each note type has many different interfaces associated with it, which are used from the APIs to the logic parts of the programme. To simplify updating interfaces, they are consolidated into the ./src/types directory, and then imported into any files that require them.

3.2.4.2