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.

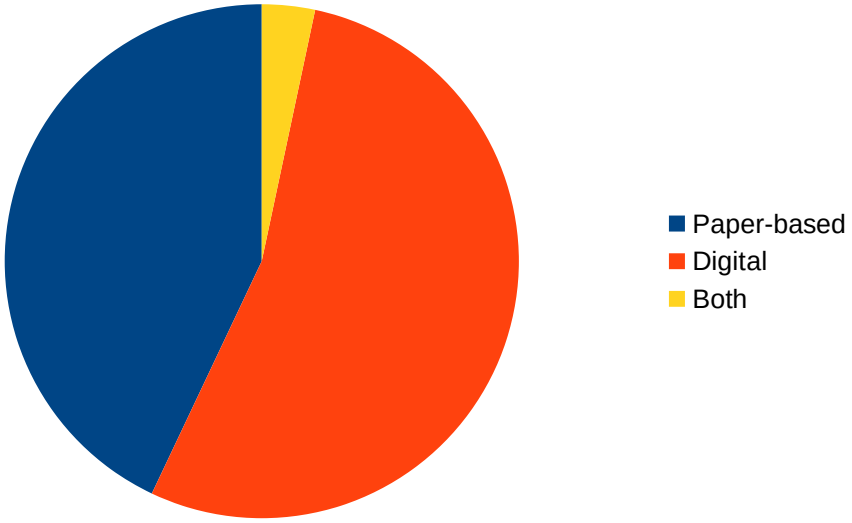
Disadvantages:

- 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.

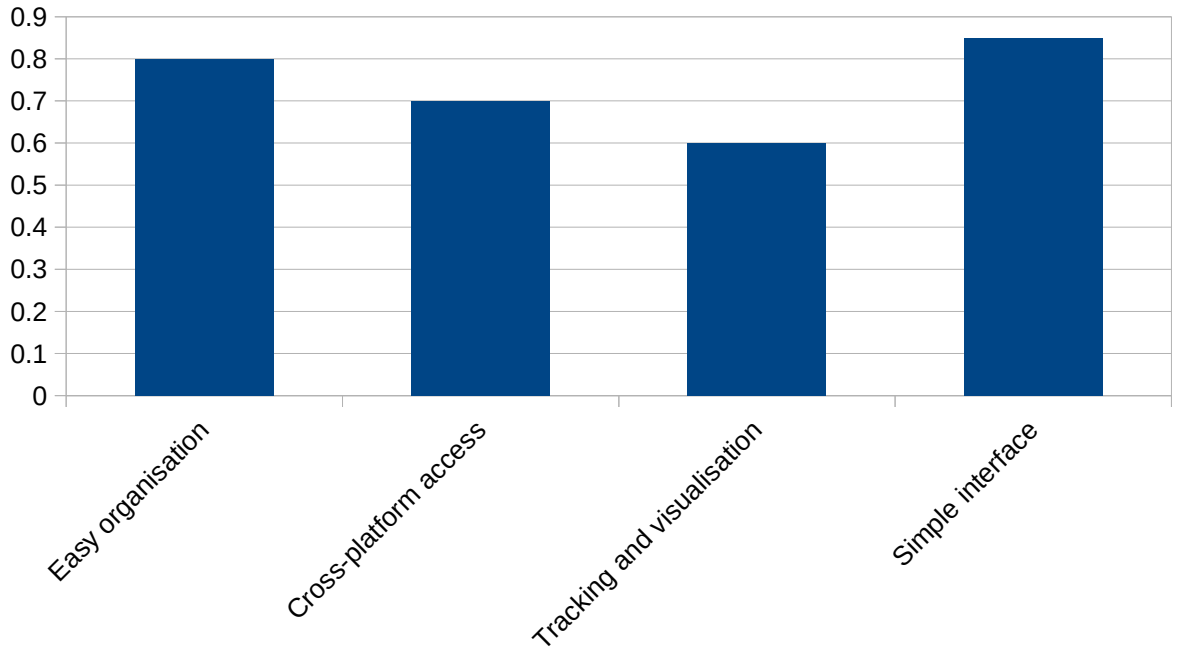
1.3.2. Survey results

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:

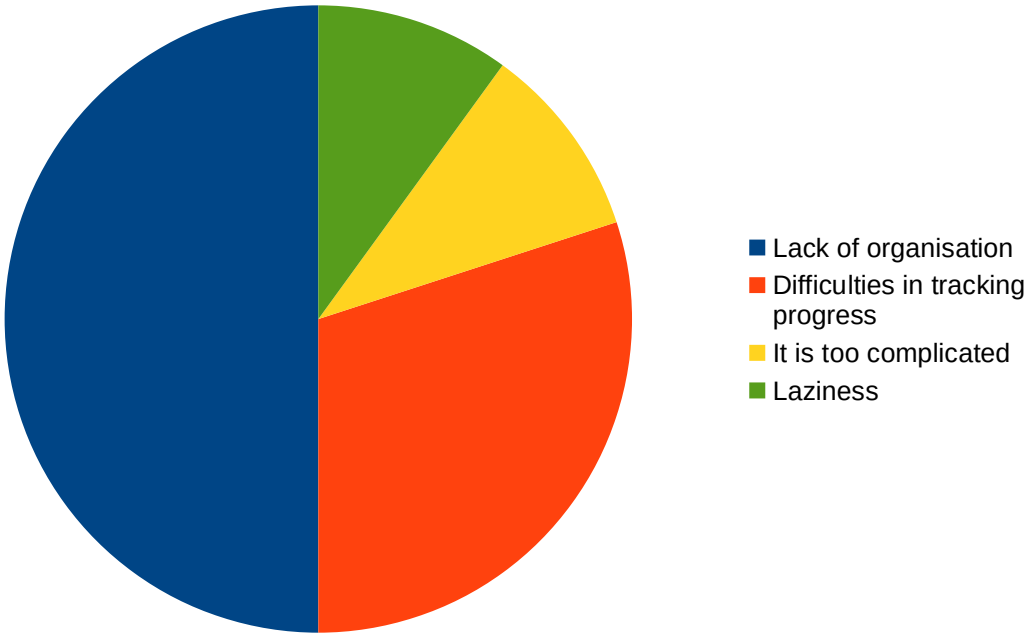
1.3.2.1 How do you currently organise your tasks and notes?



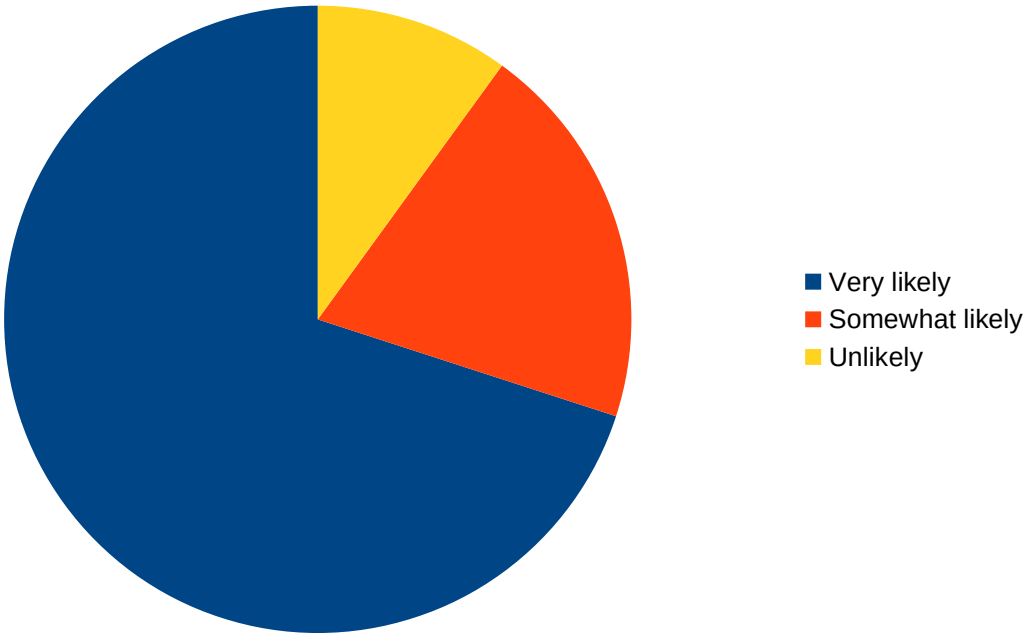
1.3.2.2 What features do you value most in a notetaking app? (Multiple choices allowed)



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 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.

1.5 Coding objectives

1.5.1. Database

1.5.1.1 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.

1.5.1.2 Create tables for:

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

1.5.2. 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

1.5.3. 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.

1.5.4. Validation

1.5.4.1 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.

1.5.4.2 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.

1.6 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.

1.7 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.

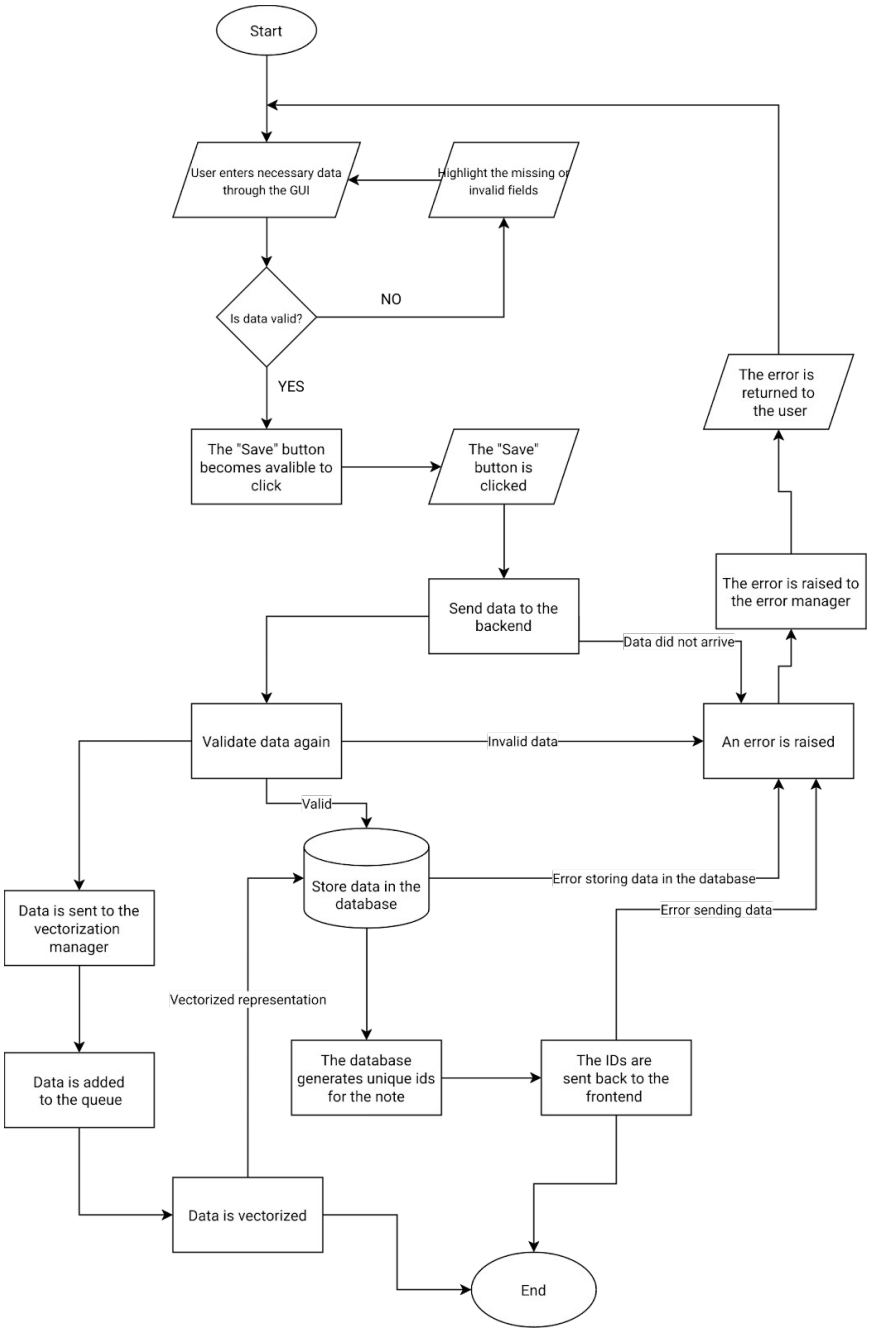
2.1.1.3 Why I chose optimistic updating

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.

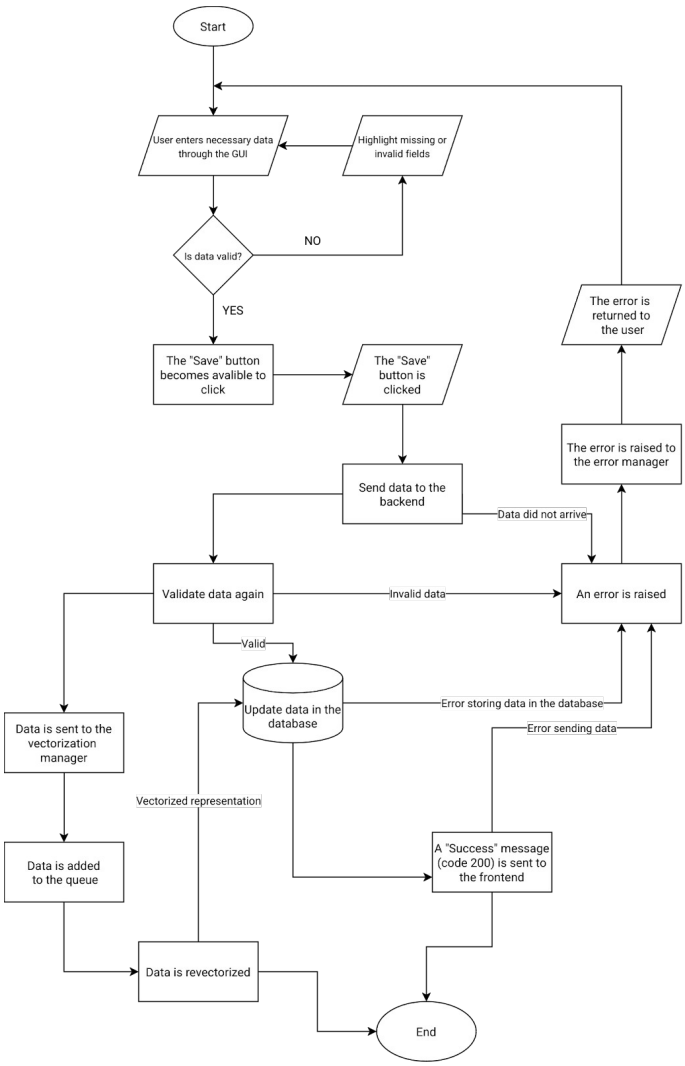
2.1.2. Creating data flow diagrams

2.1.2.1 Note creation



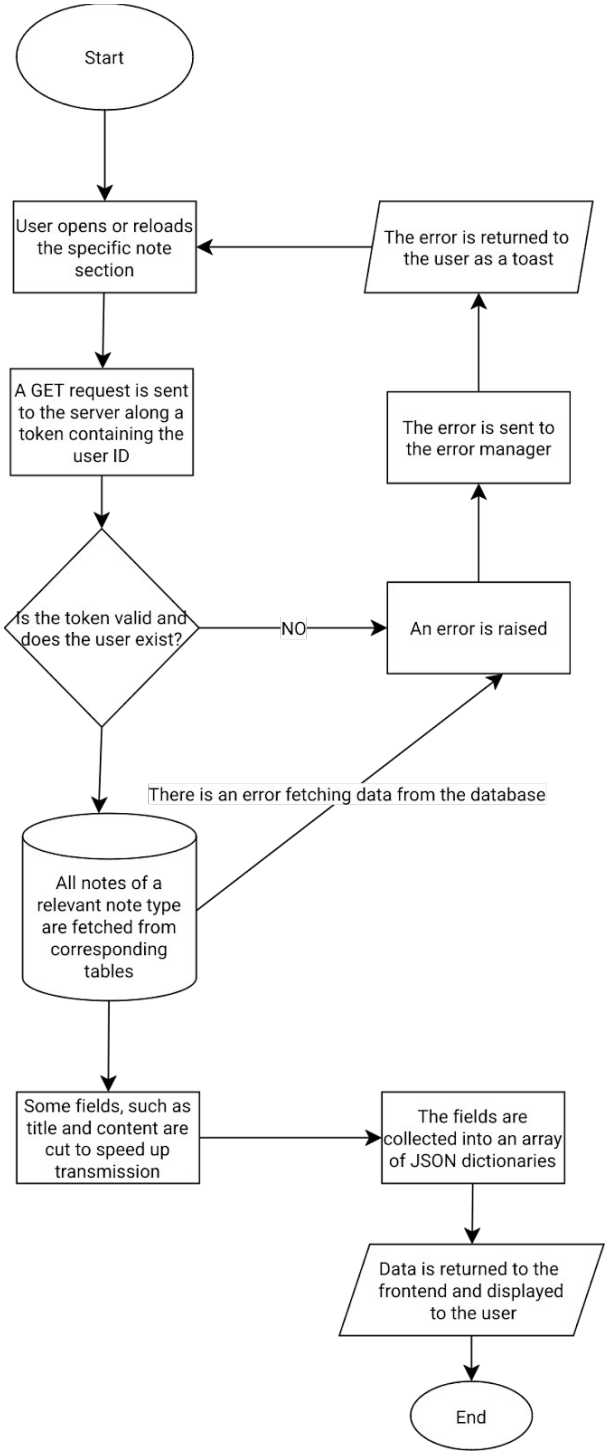
When the user clicks on “Create” button on the home page for a note type, a screen should be opened with the necessary fields, such as title, content, etc. Once the user enters all the details for a note to be considered valid by the validation script and all the validation errors have been resolved, 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



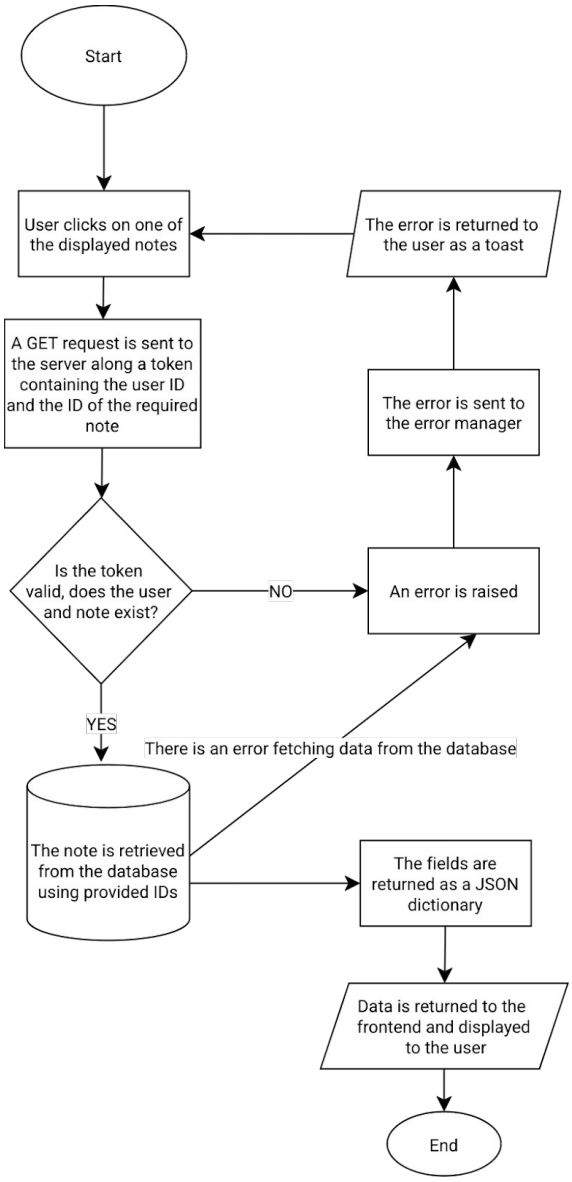
Generally, the note editing data flow is similar to creation, apart from first few details. To begin with, the “Save” button will not be enabled if no additional data was entered into the note. This is done to prevent unnecessary edit requests to the backend. In addition, if the database was updated 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



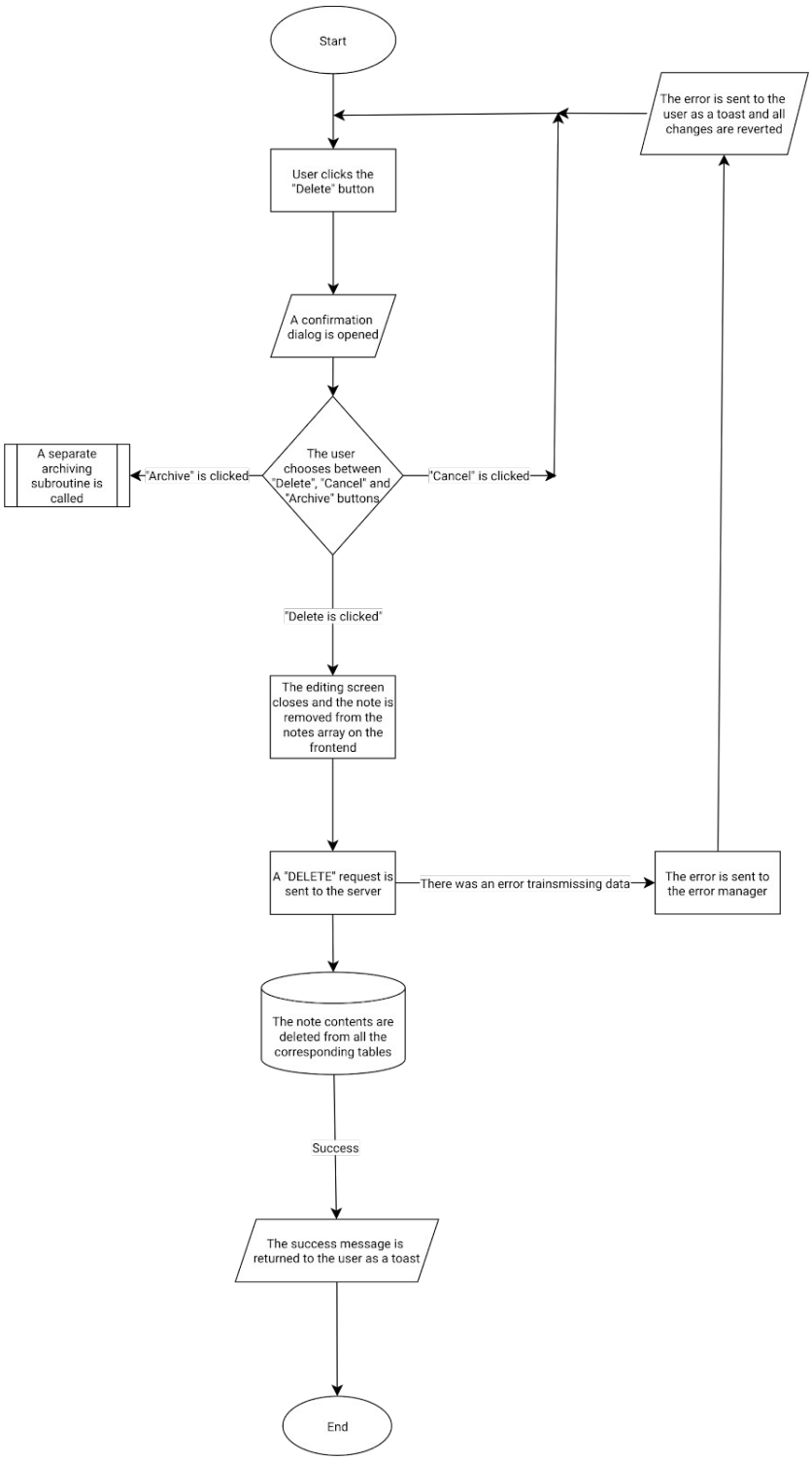
Every time a user loads a page of a specific note type, such as Tasks or Goals, a GET request is sent to the server alongside a JSON Web Token containing encoded user Id and information about the session. If the token is valid and the user exists in the database, all notes from a specific note type are 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



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



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.

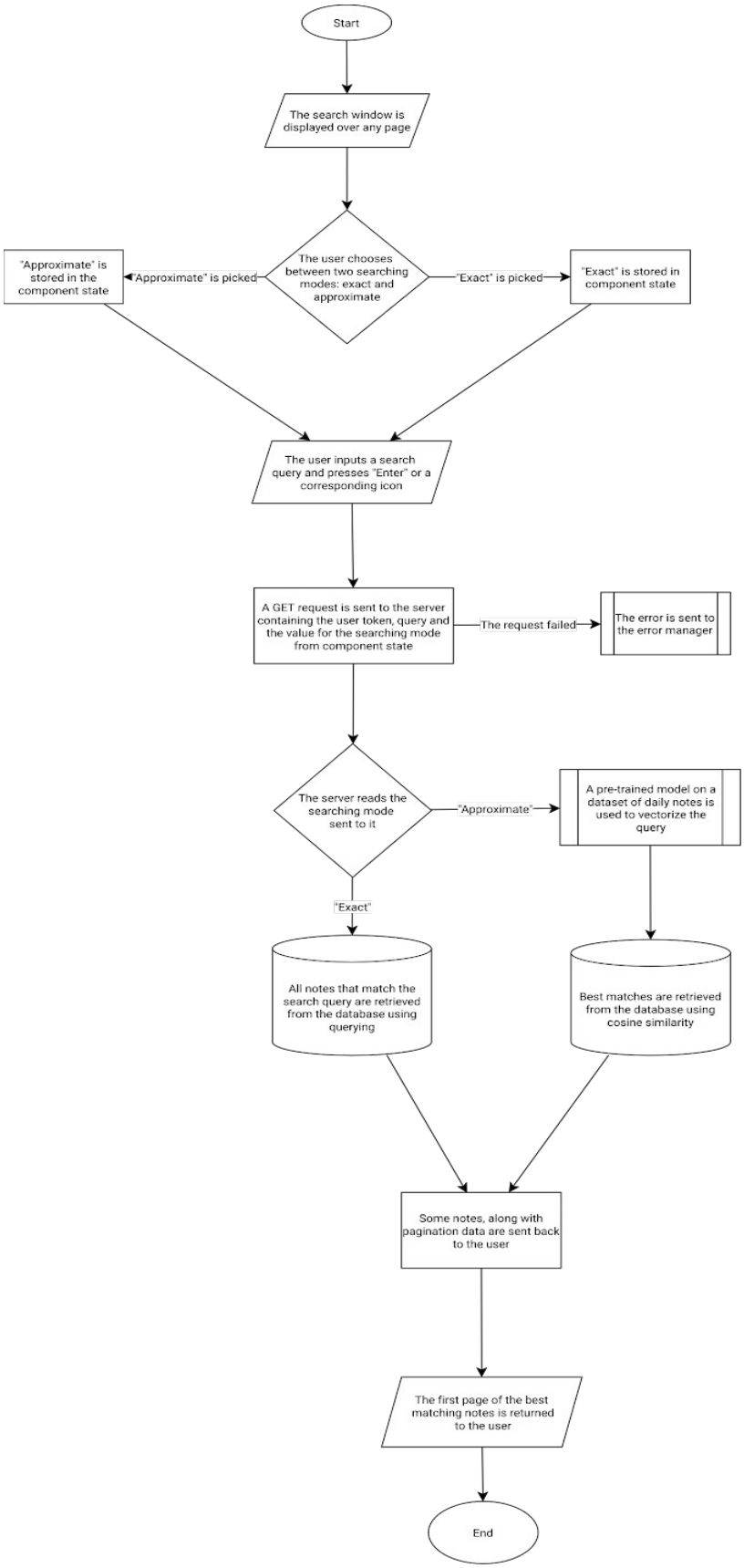
2.1.2.6 Searching for a note

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

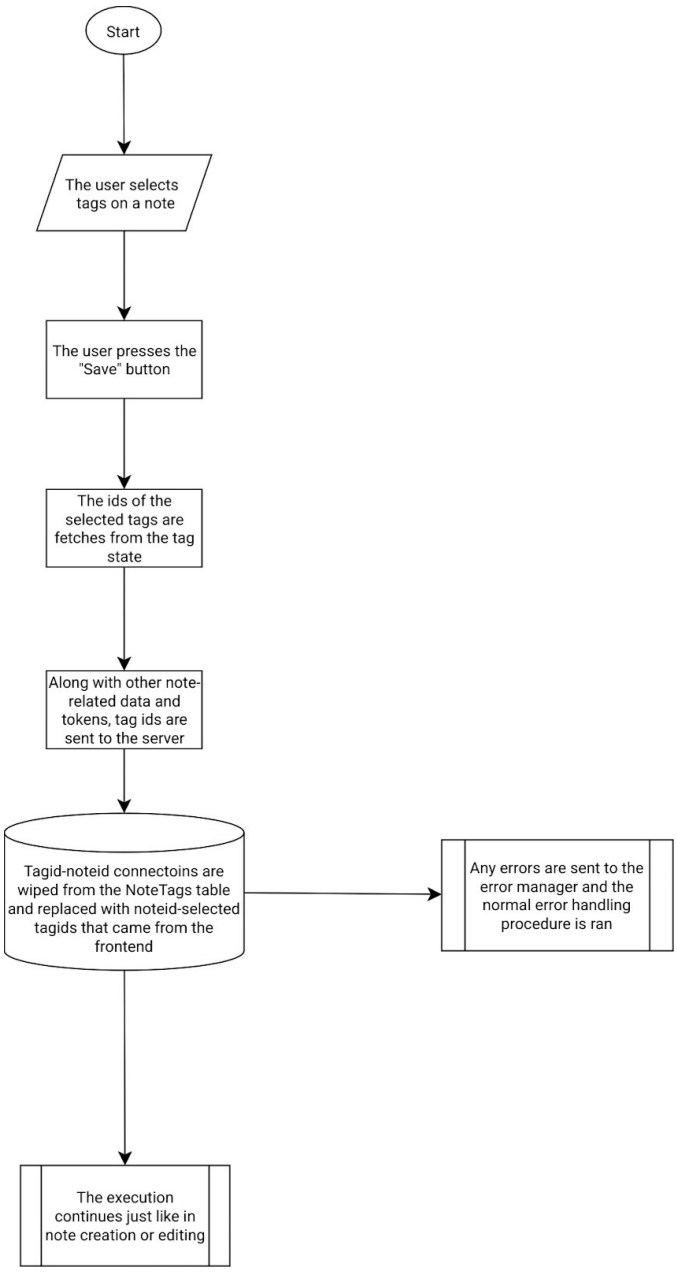
b) “Approximate” search mode. The query will be vectorized with a pretrained model and its vector representation will be compared to the stored vector representations of notes in the database using a function such as cosine similarity.

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.



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.

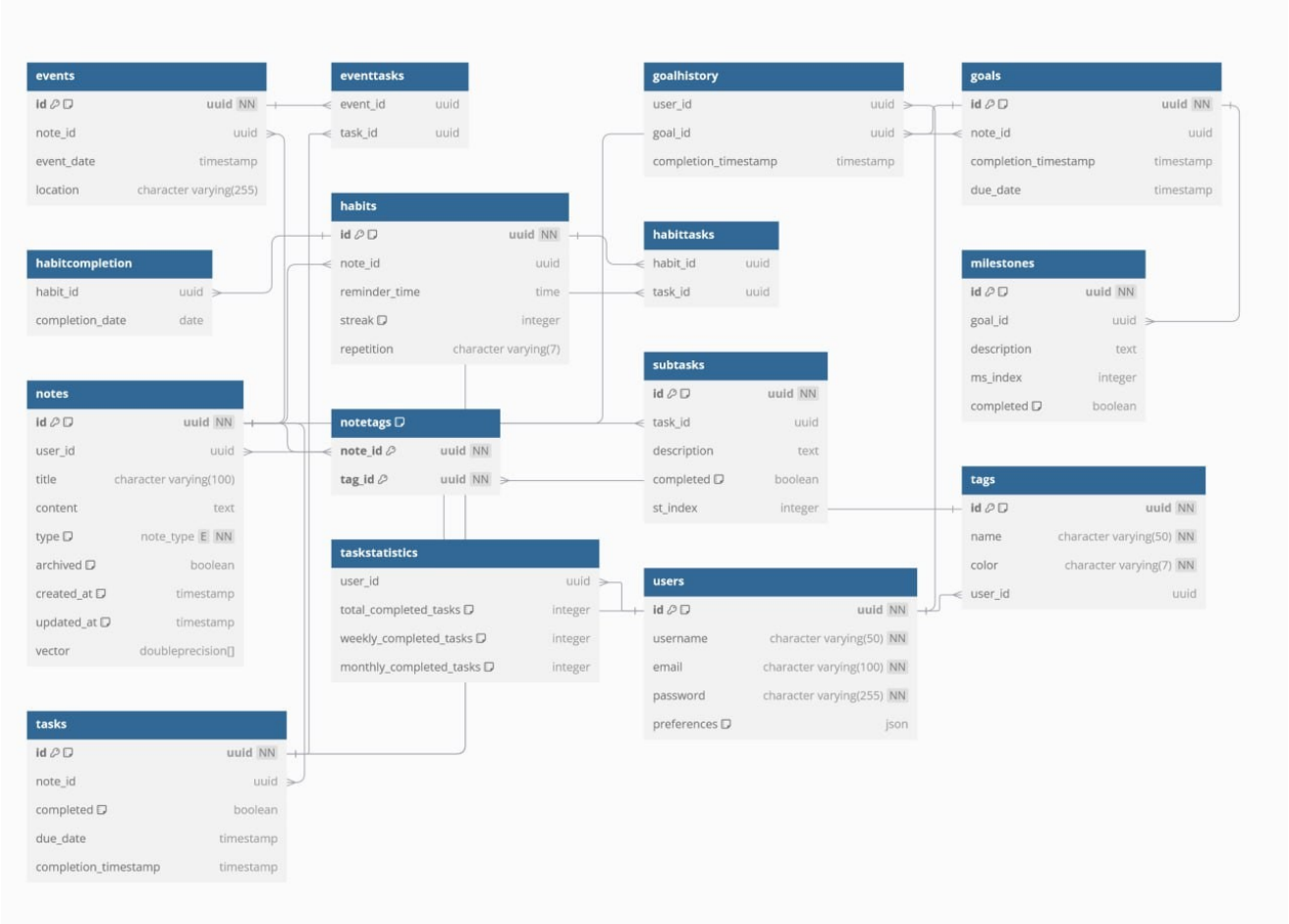
2.2.1.6 Performance:

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 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

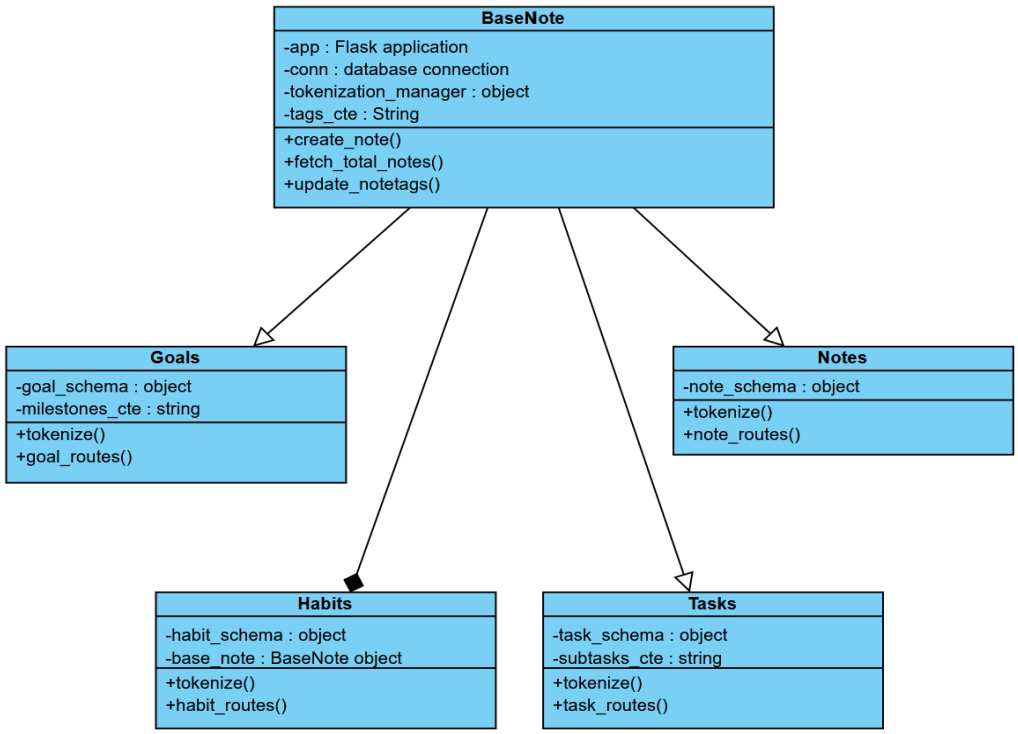
OOP (object oriented programming) is a programming paradigm that organizes software design around objects (instances of classes), rather than functions. By using OOP, we can

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.

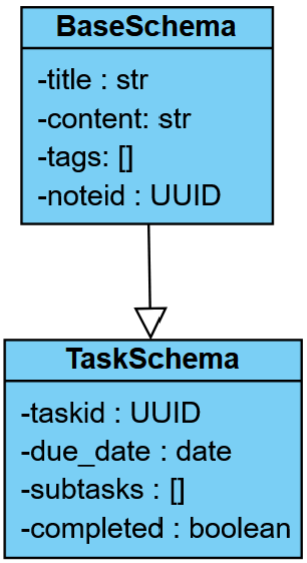
However, I noticed that most notes share similar base characteristics, such as tags, content and title and decided to use classes to mix and match actual api components to create an object that will be able to work with a specific note type. By doing that, I will encapsulate all logic for a note type 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

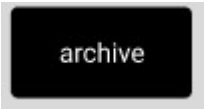
take advice from the end users of the product, keeping the final design as little as possible. Furthermore, iterative design is flexible, adapting to changing requirements or technologies, and ensures that the end product remains long-lasting, understandable and thorough.

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.

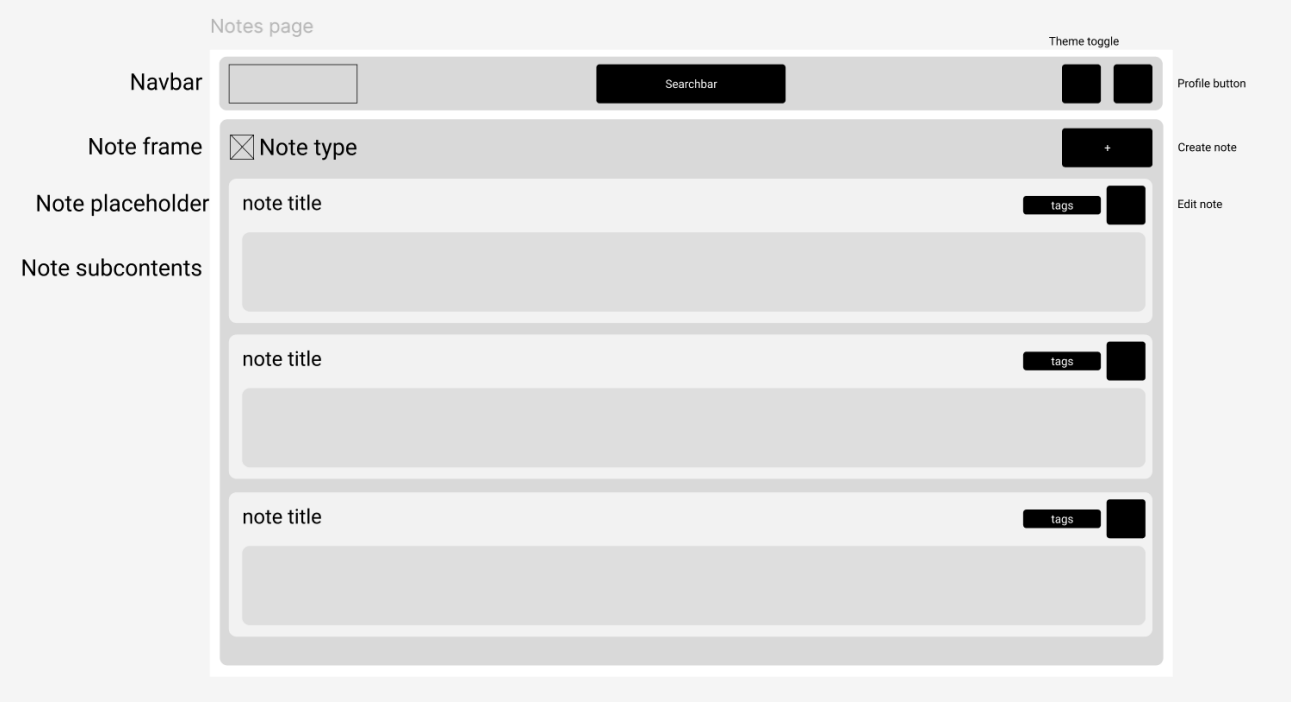
Cards

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.

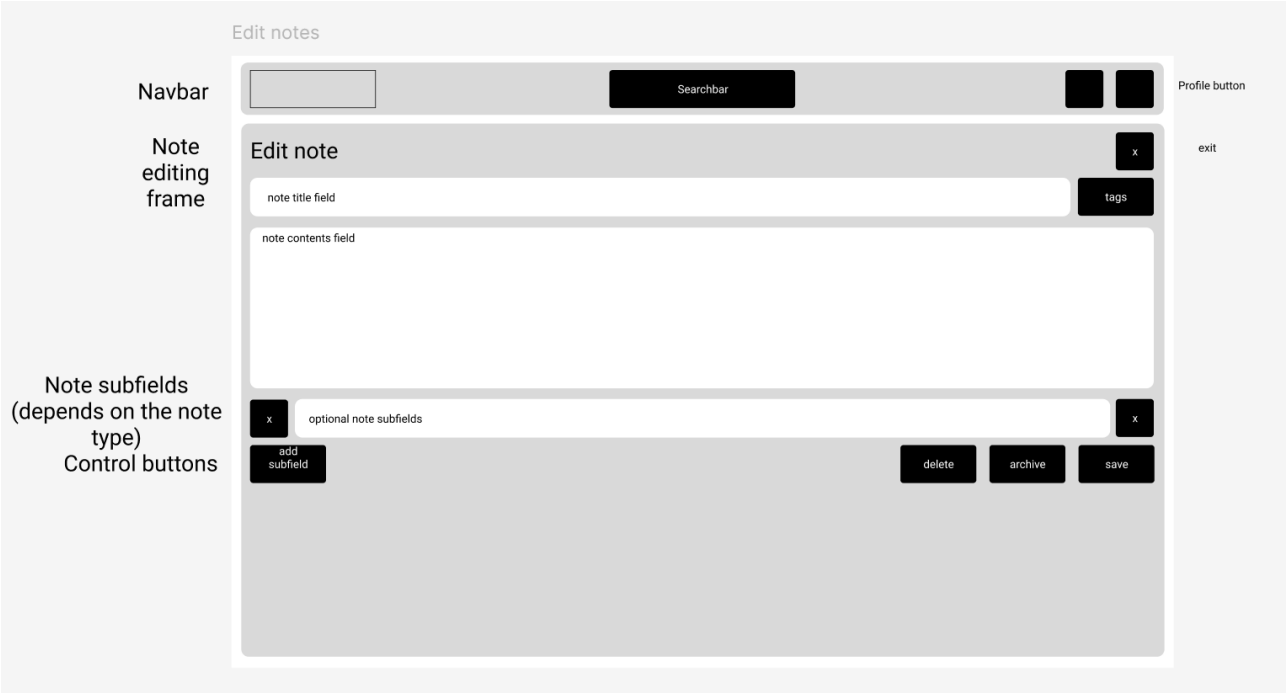
2.4.3.1.1 Viewing all notes



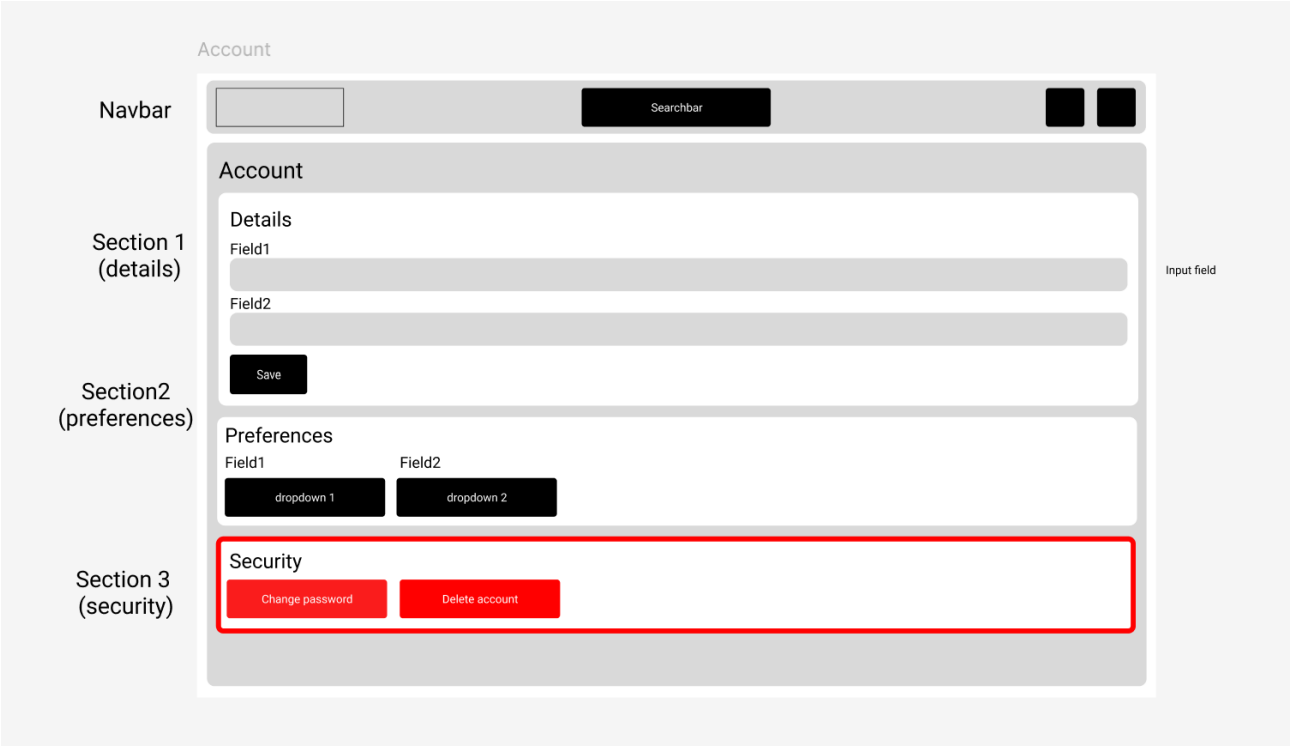
This wireframe represents the screen which will be containing all of user's notes of a specific type, along with pagination controls, ability to create new notes, view their previews (such as a cropped title and content, if they exceed a certain character limit) and open each note in a separate screen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

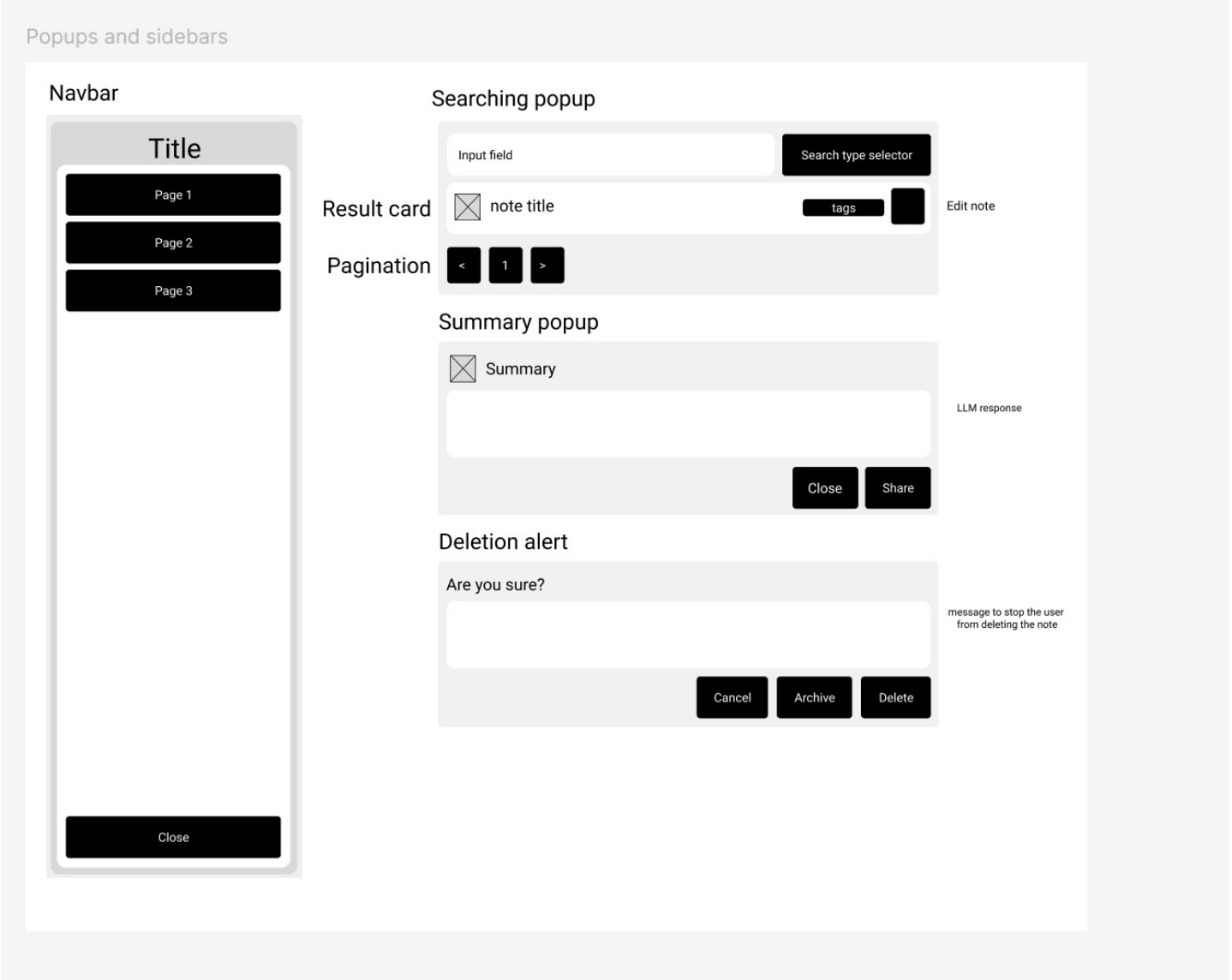


2.4.3.2 An account page where the user can edit their details



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.

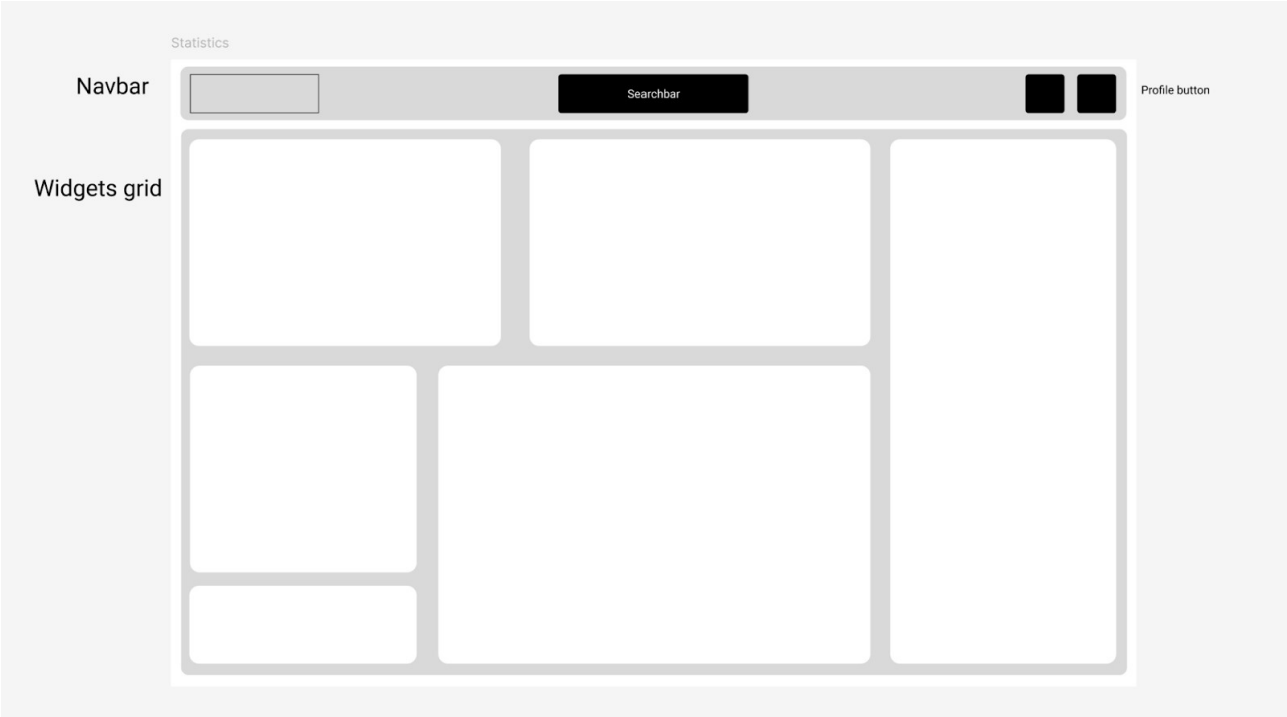
2.4.3.3 A sidebar and other popups, such as alerts or search widgets



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:

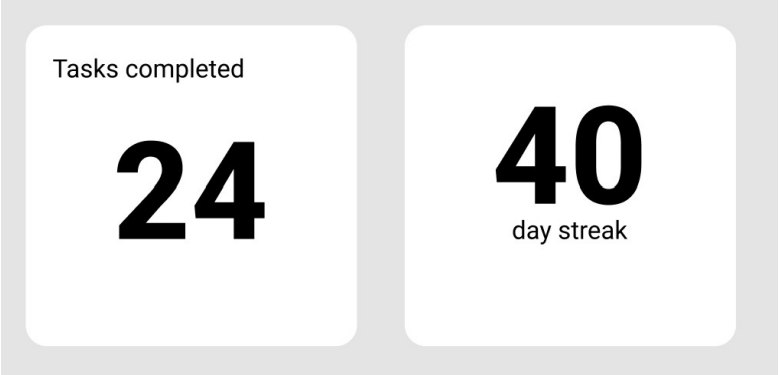


The page can be a grid with customizable widgets of various sizes. There are plenty of widget options, but most of them can be simplified to several types:

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.

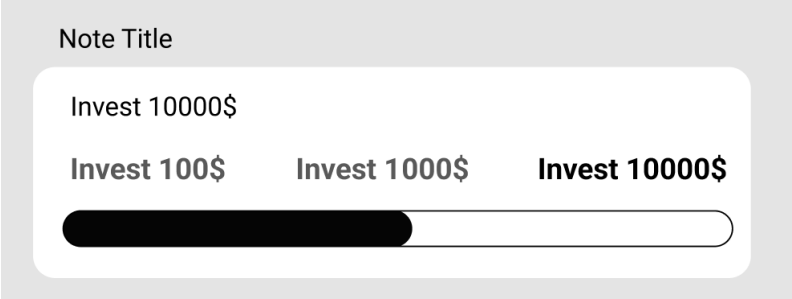
2.4.3.4.2 Numbers



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.

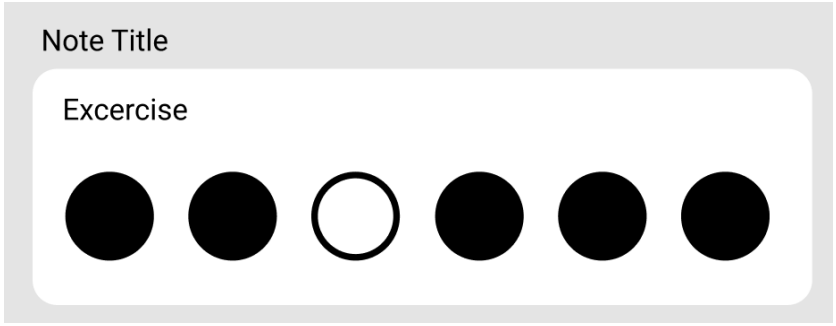


This idea may be expanded with the addition of milestones for goals, which may further motivate the user. Additionally, this example maybe used to highlight the ease of adapting these components for different sizes by stacking the milestones vertically:

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

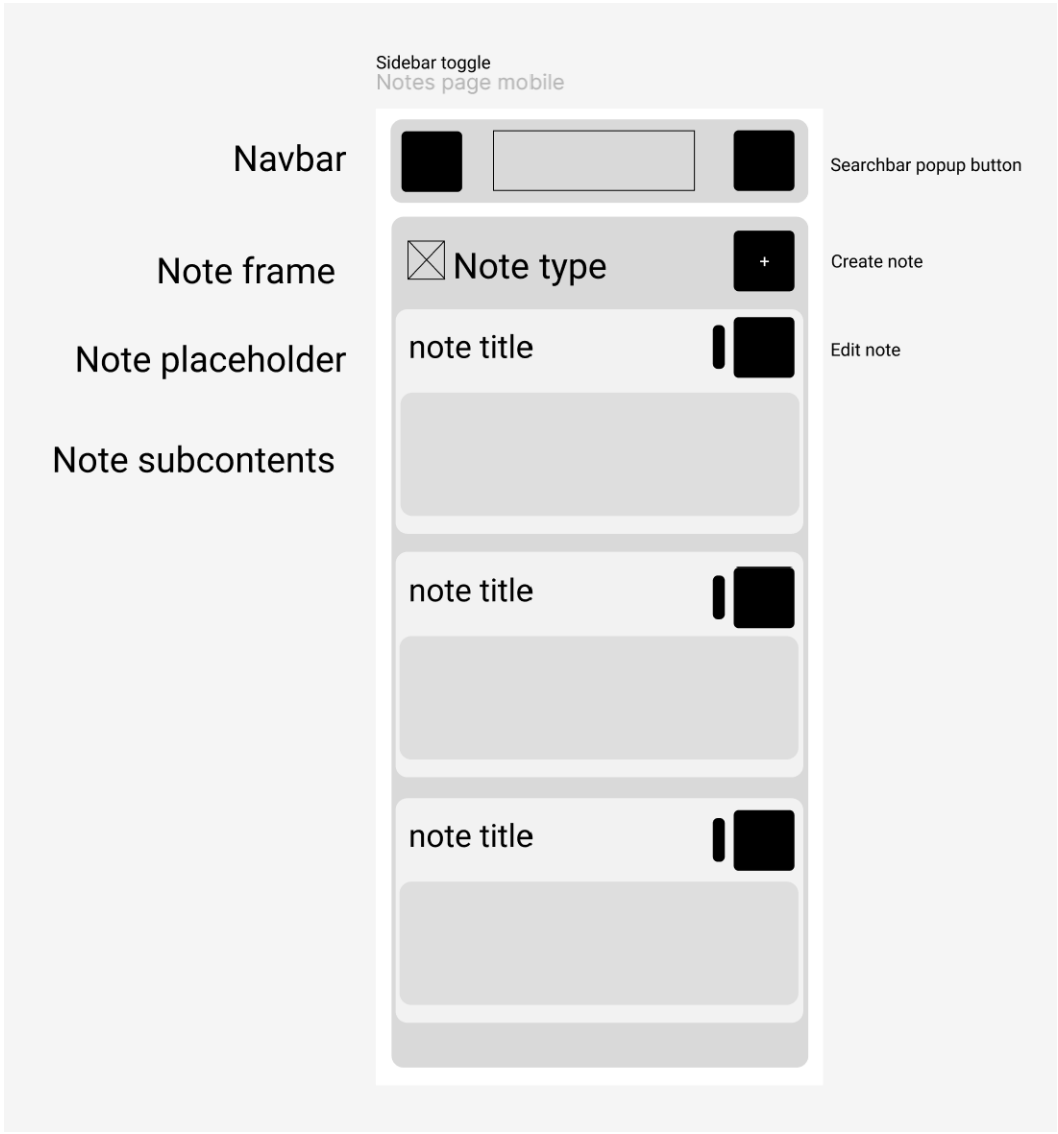
The following type can only be used to very specific notes that require daily commitment, allowing the user to visualise their progress in a simple way.

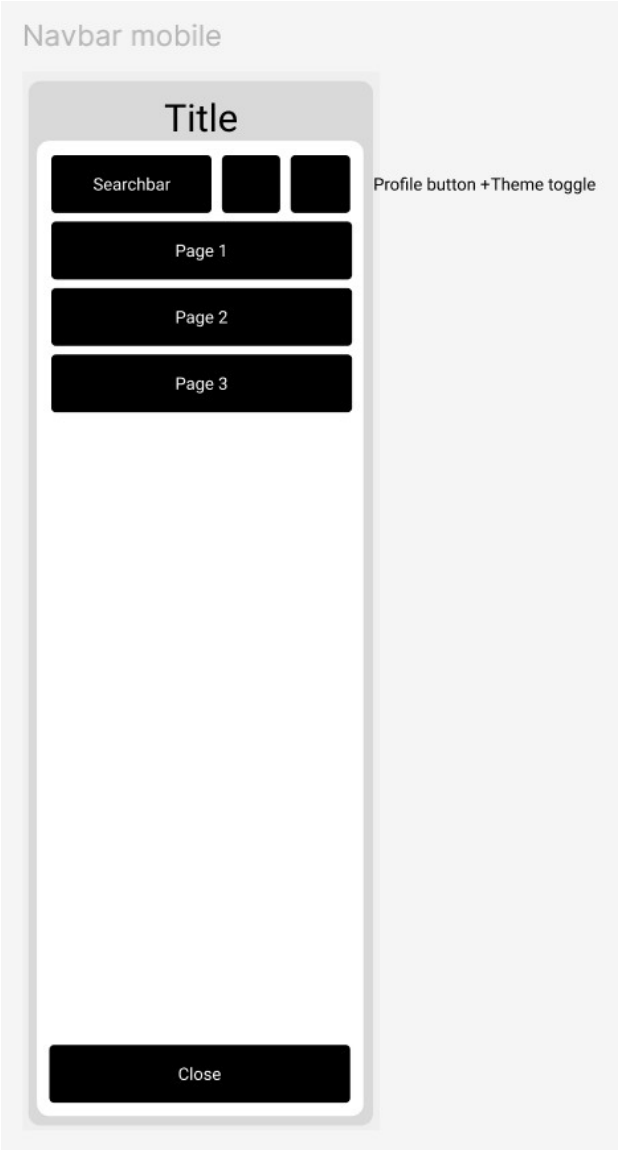


2.4.3.4.5 Mobile design

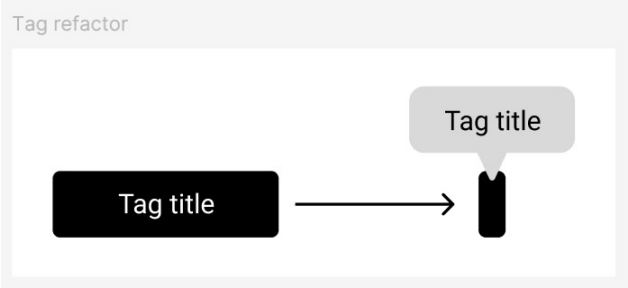
In addition, there should be design options for devices of different display sizes, to ensure that the products is long-lasting and thorough. Even though the app should be able to auto-adapt to various display sizes, some components must be moved around or replaced to accommodate for smaller screens.

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

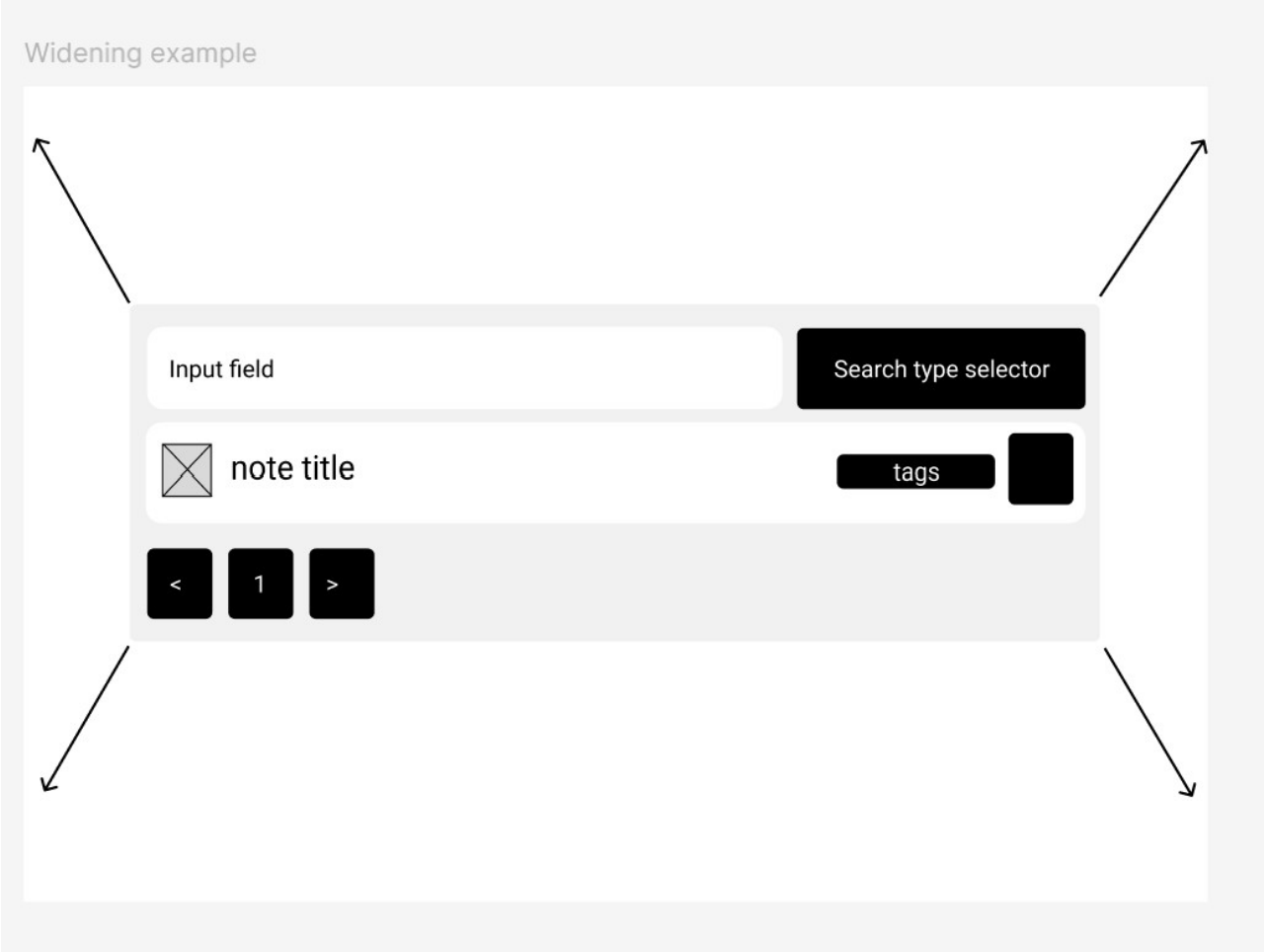


A similar refactoring happened to the note cards themselves. For example, there are less characters displayed per line as the screen shrinks to make the note cards grow in height instead of width. Moreover, additional whitespace can be found by shrinking the tag labels and just leaving small colored 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. Setup

3.1.2. 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`.

```
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.