

# **NEANOTE:**

## **A CUSTOMIZABLE NOTETAKING WEB APP FOR PRODUCTIVITY**

**BY: Roman Sasinovich**

CANDIDATE NUMBER: 6032  
CENTRE NUMBER: 74561  
CENTRE NAME: IMS  
QUALIFICATION CODE: 7517/C  
DATE: 30/04/2025

# Table of Contents

1 Analysis.....	7
1.1 Identifying the Problem.....	7
1.1.1. Background:.....	7
1.1.2. Identification.....	7
1.2 Identifying the End Users.....	8
1.3 Research.....	8
1.3.1. Key Insights from Research.....	9
1.3.1.1 Books.....	9
1.3.1.1.1 Building a Second Brain.....	9
1.3.1.1.1.1 The CODE Method.....	9
1.3.1.1.1.2 The PARA Method.....	10
1.3.1.1.2 The Bullet Journal Method.....	10
1.3.1.2 In-depth analysis of existing software.....	10
1.3.1.2.1 Notion.....	10
1.3.1.2.2 Google Keep:.....	11
1.3.1.2.3 Evernote:.....	11
1.3.1.2.4 Apple Notes:.....	12
1.3.2. Survey results.....	12
1.3.2.1 How do you currently organise your tasks and notes?.....	12
1.3.2.2 What features do you value most in a notetaking app? (Multiple choices allowed).....	13
1.3.2.3 What are the biggest challenges with your current notetaking method?.....	13
1.3.2.4 How likely are you to use an app that combines notetaking and statistics?.....	14
1.3.2.5 Conclusion.....	14
1.4 Interview with Anton Kogun.....	15
1.5 Modelling the problem.....	16
1.6 Coding objectives.....	17
1.6.1. Database.....	17
1.6.1.1 Configuration and Security.....	17
1.6.1.2 Create tables for:.....	17
1.6.2. Design the backend functionality:.....	17
1.6.3. User Interface.....	18
1.6.4. Validation.....	18
1.6.4.1 Frontend Validation.....	18
1.6.4.2 Backend Validation.....	19
1.7 Proposed solution.....	19
2 Documented Design.....	20
2.1 User data flow.....	20
2.1.1. Determining frontend-backend communication.....	20
2.1.1.1 Optimistic updating.....	20
2.1.1.2 Pessimistic updating.....	20
2.1.1.3 Why I chose optimistic updating.....	20
2.1.2. Creating data flow diagrams.....	21
2.1.2.1 Note creation.....	21
2.1.2.2 Editing a note.....	22

2.1.2.3 Fetch all notes of a specific type.....	23
2.1.2.4 Fetch one note.....	24
2.1.2.5 Delete a note.....	26
2.1.2.6 Searching for a note.....	27
2.1.2.7 Tag workflow.....	28
2.1.2.8 Registering and logging users in.....	29
2.2 Backend.....	29
2.2.1. Designing the database.....	29
2.2.1.1 Detecting patterns in data representation.....	29
2.2.1.2 Choosing the database.....	29
2.2.1.3 Efficiency.....	30
2.2.1.4 Data Types and Functions:.....	30
2.2.1.5 Why use UUID?.....	30
2.2.1.6 Performance:.....	30
2.2.1.7 Security:.....	30
2.2.2. Database relationships.....	30
2.2.2.1 User data.....	31
2.2.2.2 Notes.....	32
2.2.2.3 Tagging.....	34
2.2.2.4 Statistics.....	34
2.2.2.5 Widgets.....	35
2.2.2.6 Conclusion.....	35
2.2.3. Backend file structure.....	36
2.2.4. OOP.....	37
2.2.4.1 Constructing note processing code.....	37
2.2.4.2 Simplifying validation schemas.....	38
2.3 Frontend.....	39
2.3.1. User data on the frontend.....	39
2.3.2. Frontend file structure.....	40
2.4 User Interface.....	42
2.4.1. The 10 principles of good design.....	42
2.4.2. Design as an iterative process.....	42
2.4.3. Creating wireframes.....	43
2.4.3.1 A generic note viewing and editing page, which supports organization by tagging, additional form fields and timestamps.....	44
2.4.3.1.1 Viewing all notes.....	44
2.4.3.1.2 Editing a note.....	44
2.4.3.2 An account page where the user can edit their details.....	45
2.4.3.3 A sidebar and other popups, such as alerts or search widgets.....	46
2.4.3.4 A note statistics page.....	46
2.4.3.4.1 Charts.....	47
2.4.3.4.2 Numbers.....	48
2.4.3.4.3 Progress Bars.....	48
2.4.3.4.4 Streaks.....	49
2.4.3.4.5 Mobile design.....	49
2.4.4. Choosing the color and fonts.....	52
2.4.4.1 Core color palette.....	52
2.4.4.2 Typeface: Open Sans.....	54
2.4.4.2.1 Introduction.....	54

2.4.4.2.2 Character set support.....	54
2.4.4.2.3 Licensing.....	55
2.4.4.2.4 Performance.....	55
3 Technical solution.....	55
3.1 Backend.....	55
3.1.1. app.py.....	55
3.1.1. modules/universal.py.....	57
3.1.1.1 BaseNote class.....	57
3.1.1.2 Universal note functionality.....	59
3.1.1.3 Searching.....	59
3.1.1.4 Recents.....	61
3.1.2. modules/notes.py.....	65
3.1.2.1 Create Note.....	67
3.1.2.2 Update Note.....	67
3.1.2.3 Delete Note.....	68
3.1.2.4 Get a specific note.....	69
3.1.2.5 Get Note previews.....	70
3.1.3. modules/tasks.py.....	72
3.1.3.1 Initialization.....	72
3.1.3.2 Task creation.....	73
3.1.3.3 Updating tasks.....	75
3.1.3.4 Deleting a task.....	77
3.1.3.5 Toggling task fields.....	78
3.1.3.6 Task Previews.....	80
3.1.3.7 Retrieving a single task.....	82
3.1.4. modules/archive.py.....	84
3.1.4.1 Archive and Restore.....	84
3.1.4.2 GetAll.....	85
3.1.5. modules/tags.py.....	87
3.1.5.1 CRUD tags.....	87
3.1.5.2 Linking Tags.....	91
3.1.6. Widget-related modules.....	92
3.1.6.1 models/widget.py.....	92
3.1.6.2 repositories/widget_repo.py.....	95
3.1.6.2.1 WidgetRepository initialisation cache.....	96
3.1.6.2.2 Getting user widgets.....	96
3.1.6.2.3 Other CRUD functions.....	97
3.1.6.2.4 Widget data sources.....	98
3.1.6.3 modules/widgets.py.....	99
3.1.7. modules/users.py.....	101
3.1.7.1 Registration.....	101
3.1.7.2 Login.....	102
3.1.7.3 Updating user data.....	103
3.1.7.4 Update user password.....	104
3.1.7.5 Delete user.....	105
3.1.8. modules/ honorable mentions.....	106
3.1.9. utils/utils.py.....	106
3.1.9.1 Security.....	106
3.1.9.2 Note ownership.....	107

3.1.9.3 Universal utilities.....	109
3.1.9.4 utils/word2vec.py.....	111
3.1.9.5 Training the model.....	112
3.1.9.6 Text tokenization.....	113
3.1.9.7 Processing tokens with the model.....	114
3.1.9.8 Looking notes up with cosine similarity.....	115
3.1.10. utils/priorityqueue.py.....	116
3.1.10.1 What is Heap?.....	117
3.1.10.2 Implementation.....	117
3.1.11. utils/recentsList.py.....	121
3.1.11.1 Implementing a doubly linked list.....	121
3.1.11.2 Implementing the recent notes manager.....	122
3.1.12. utils/userDeleteGraph.py.....	124
3.1.12.1 Setup.....	124
3.1.12.2 DFS.....	127
3.1.12.3 Deleting data with backoff.....	128
3.1.13. Conclusion.....	130
3.2 Frontend.....	130
3.2.1. Setup and configuration.....	132
3.2.1.1 ./index.html.....	132
3.2.1.2 ./package.json.....	133
3.2.1.2.1 Libraries used.....	135
3.2.2. ./src.....	135
3.2.2.1 ./src/main.tsx.....	135
3.2.2.1.1 Providers.....	136
3.2.2.1.2 ./components/providers/ThemeProvider.tsx.....	136
3.2.2.1.3 ./components/providers/DisplayContext.tsx.....	138
3.2.2.2 ./src/routes.tsx.....	139
3.2.2.2.1 ./components/providers/token-check.tsx.....	141
3.2.2.3 ./src/formValidation.tsx.....	142
3.2.3. ./src/api.....	145
3.2.3.1 ./src/api/api.ts.....	145
3.2.3.2 ./src/api/users.ts.....	146
3.2.3.3 ./src/api/tagsApi.ts.....	148
3.2.3.4 ./src/api/goalsApi.ts.....	151
3.2.3.5 ./src/api/universalApi.ts.....	153
3.2.4. ./src/api/types.....	155
3.2.4.1 What is an interface?.....	155
3.2.4.2 ./src/api/types/userTypes.ts.....	155
3.2.4.3 ./src/api/types/tagTypes.ts.....	156
3.2.4.4 ./src/api/types/taskTypes.ts.....	157
3.2.4.5 ./src/api/types/habitTypes.ts.....	158
3.2.5. Pages.....	159
3.2.5.1 Landing.....	159
3.2.5.2 Register.....	161
3.2.5.2.1 ./src/Pages/Register/Register.tsx.....	162
3.2.5.2.2 ./src/Pages/Register/useRegister.tsx.....	165
3.2.5.3 Login.....	166
3.2.5.3.1 ./src/Pages/Login/Login.tsx.....	167

3.2.5.3.2 ./src/Pages/Login/useLogin.tsx.....	169
3.2.5.4 Account.....	170
3.2.5.4.1 ./src/Pages/Account/Account.tsx.....	170
3.2.5.4.2 ./src/Pages/Account/useUser.tsx.....	173
3.2.5.4.3 ./src/Pages/Account/Components/PasswordDrawerDialog.tsx	
.....	179
3.2.5.4.4	
./src/Pages/Account/Components/DeleteAccountDrawerDialog.tsx....	182
3.2.5.5 Tags.....	186
3.2.5.5.1 ./src/Pages/Tags/Tags.tsx.....	187
3.2.5.5.2 ./src/Pages/Tags/useTags.tsx.....	191
3.2.5.5.2.1 Binary search.....	194
3.2.5.5.3 ./src/Pages/Tags/TagsMenu.tsx.....	195
3.2.5.5.3.1 Quicksort.....	198
3.2.5.6 Tasks.....	199
3.2.5.6.1 ./src/Pages/Tasks/Tasks.tsx.....	200
3.2.5.6.2 ./src/Pages/Tasks/CreateTasks.tsx.....	202
3.2.5.6.3 ./src/Pages/Tasks/Fromcomponents/FormInputs.tsx.....	205
3.2.5.6.4 ./src/Pages/Tasks/Fromcomponents/FormInputs.tsx.....	206
3.2.5.6.5 ./src/Pages/Tasks/FromComponents/FormInputs.tsx.....	207
3.2.5.6.6 ./src/Pages/Tasks/FromComponents/Subtasks.tsx.....	207
3.2.5.6.7 ./src/Pages/Tasks/useTasks.ts.....	210
3.2.5.6.8 ./src/Pages/Tasks/DatePicker/DatePicker.tsx.....	218
3.2.5.7 Goals.....	221
3.2.5.7.1 ./src/Pages/Goals/Goals.tsx.....	222
3.2.5.7.1.1 ./src/Pages/Goals/GoalCard/GoalCard.tsx.....	224
3.2.5.7.1.2 ./src/Pages/Goals/GoalCard/GoalCard.css.....	227
3.2.5.7.2 ./src/Pages/Goals/useGoals.tsx.....	227
3.2.5.7.3 ./src/Pages/Goals/EditGoals.tsx.....	235
3.2.5.8 Dashboard.....	239
3.2.5.8.1 ./src/Pages/Dashboard/Dashboard.tsx.....	239
3.2.5.8.2 ./src/Pages/Dashboard/EditDashboard.tsx.....	241
3.2.5.8.3 ./src/Pages/Dashboard/Widgets/Widget.tsx.....	243
3.2.5.8.4 ./src/Pages/Dashboard/Widgets/WidgetGrid.tsx.....	245
3.2.5.8.5 ./src/Pages/Dashboard/Widgets/EditPicker.tsx.....	248
3.2.5.8.6 ./src/Pages/Dashboard/Widgets/ChartWidget.tsx.....	253
3.2.5.8.7 ./src/Pages/Dashboard/useDashboard.tsx.....	254
3.2.5.9 Archive.....	258
3.2.5.9.1 ./src/Pages/Archive/Components/ArchiveCard.tsx.....	259
3.2.5.9.2 ./src/components/DeleteDialog/DeleteDialog.tsx.....	261
3.2.5.9.3 ./src/Pages/Archive/Archive.tsx.....	262
3.2.5.9.4 ./src/Pages/Archive/useArchive.tsx.....	263
3.2.5.10 Calendar.....	266
3.2.5.10.1 ./src/Pages/Calendar/Calendar.tsx.....	266
3.2.5.10.2 ./src/Pages/Calendar/useCalendar.tsx.....	269
3.2.5.11 Miscellaneous.....	271
3.2.5.11.1 ./src/components/NavBar/NavBar.tsx.....	271
3.2.5.11.2 Search.....	272
3.2.5.11.2.1 ./src/components/SearchBar/SearchBar.tsx.....	272

3.2.5.11.2.2 ./src/components/SearchBar/Components/SearchCard.tsx	275
3.2.5.11.2.3 ./src/components/Universal/UniversalCard.tsx.....	275
3.2.5.11.3 Theme selector.....	276
3.2.5.11.3.1 ./src/components/NavBar/Components/ThemeSwitcher.tsx .....	277
3.2.5.11.4 Sidebar.....	278
3.2.5.11.4.1 ./src/components/SideBar/modules.tsx.....	279
3.2.5.11.4.2 ./src/components/SideBar/SideBar.tsx.....	281
4 Testing.....	283
4.1 Hands-on testing.....	283
4.1.1. Setup.....	283
4.2 Registration and login.....	284
4.3 Notes.....	290
4.4 Tags.....	303
4.5 Tasks.....	312
4.6 Goals.....	312
4.7 Habits.....	312
4.8 Widgets.....	312
4.9 Calendar.....	312
4.10 Archive.....	312
4.11 Miscellaneous.....	312
4.12 Account.....	312

# 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 Carroll. 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

### **1.3.1.1.2 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.

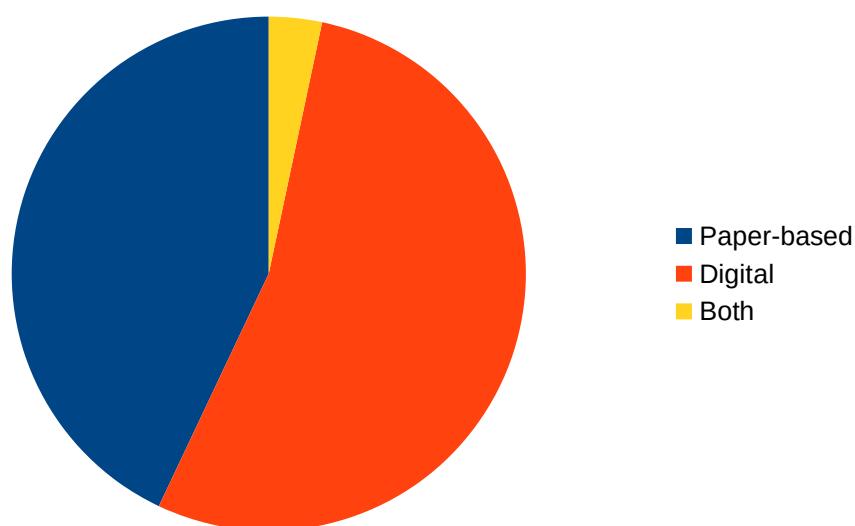
##### **Disadvantages:**

- 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.

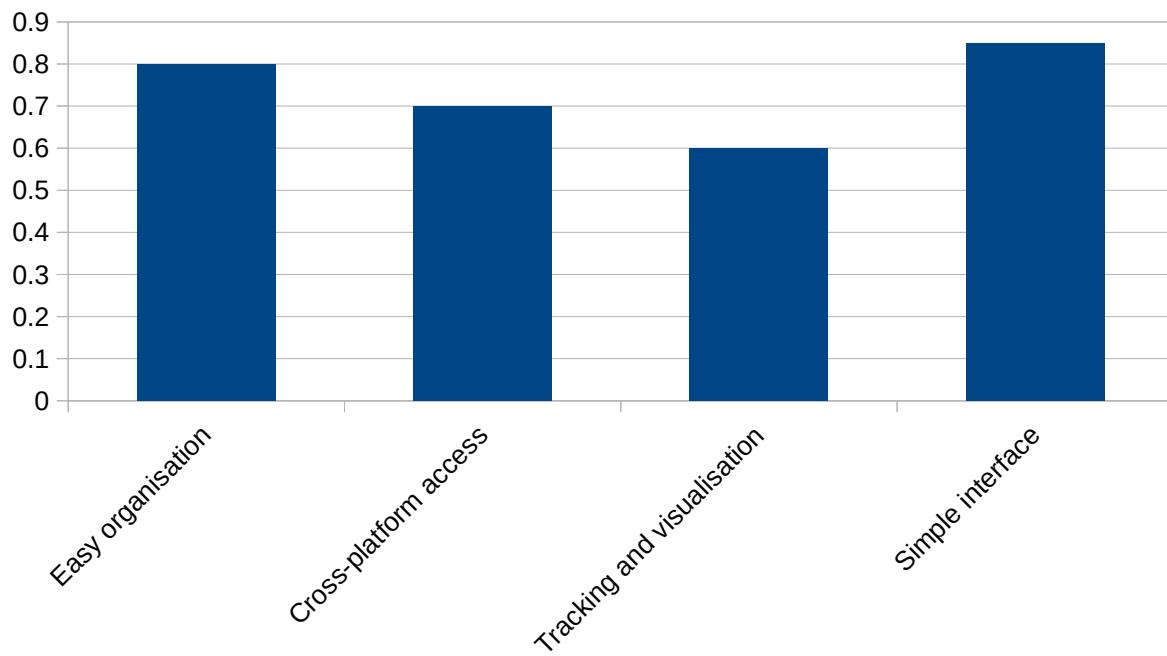
### **1.3.2. Survey results**

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:

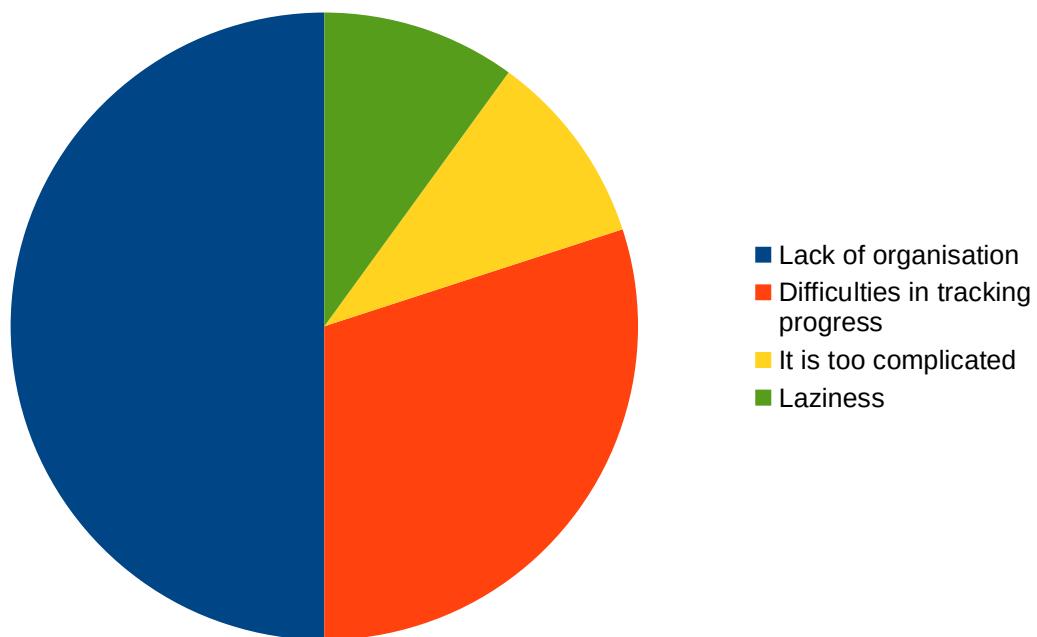
#### **1.3.2.1 How do you currently organise your tasks and notes?**



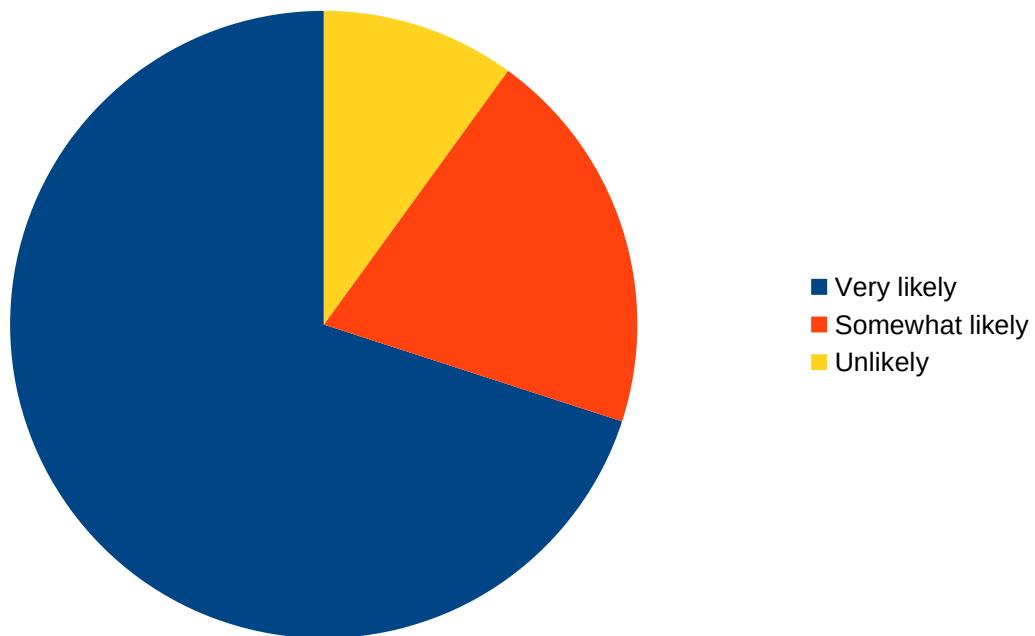
### **1.3.2.2 What features do you value most in a notetaking app? (Multiple choices allowed)**



### **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
- Widgets
- 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 authenticate 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 to 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 accessibility 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.

## **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**

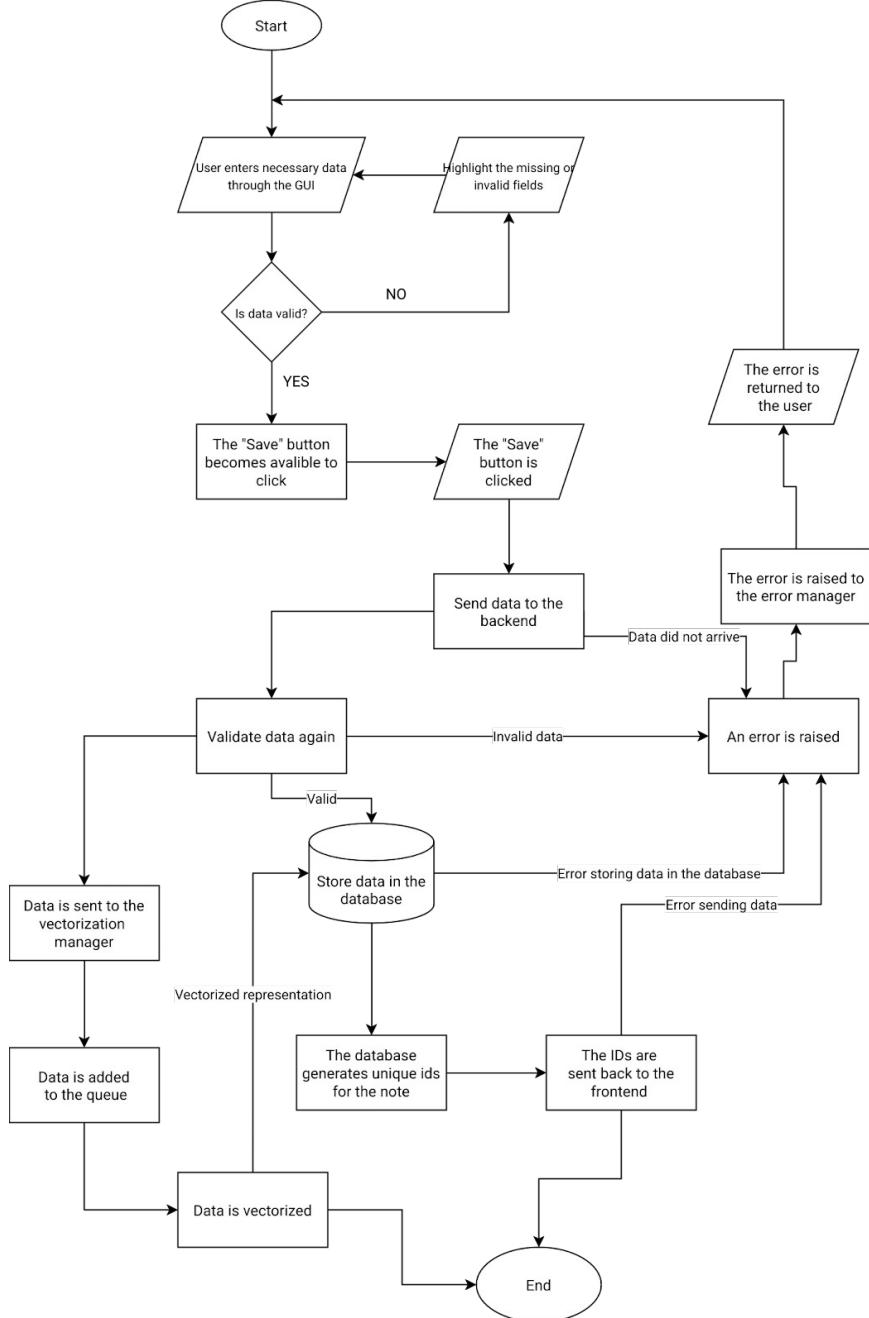
Pessimistic updating is a design pattern where the user interface is not updated until the backend operation has been successfully completed. This approach ensures data consistency but can lead to perceived delays, especially in scenarios with slow network connections or complex backend processing, which, 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

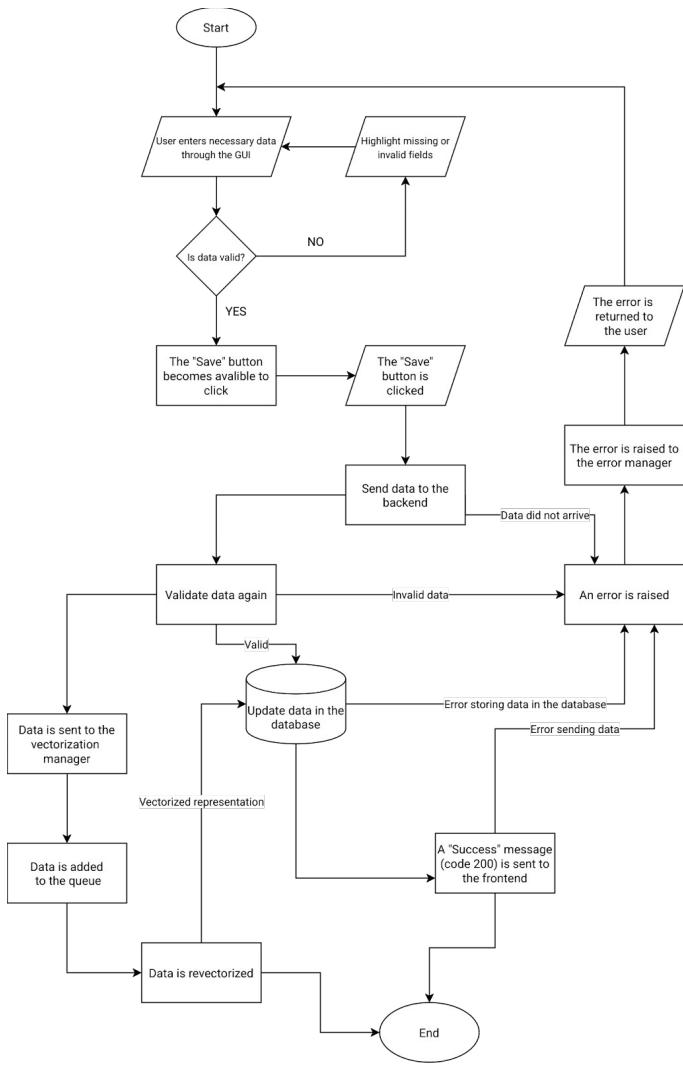
### 2.1.2.1 Note creation



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.

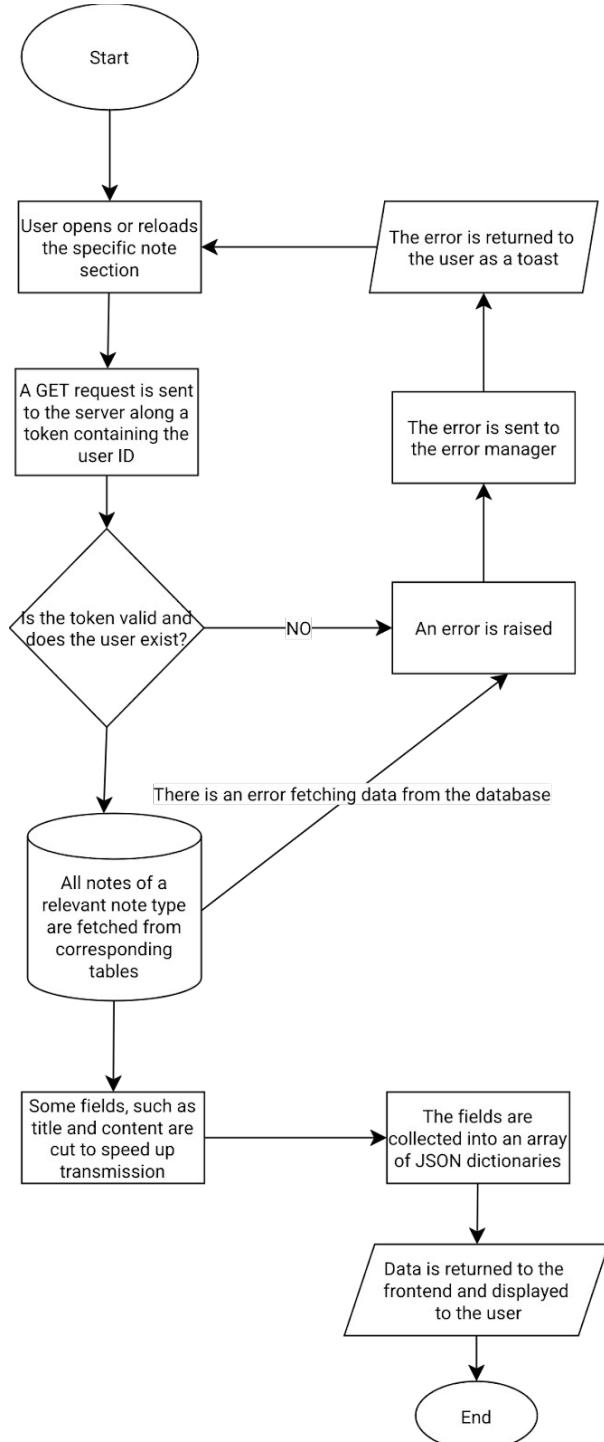
### 2.1.2.2 Editing a note



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.

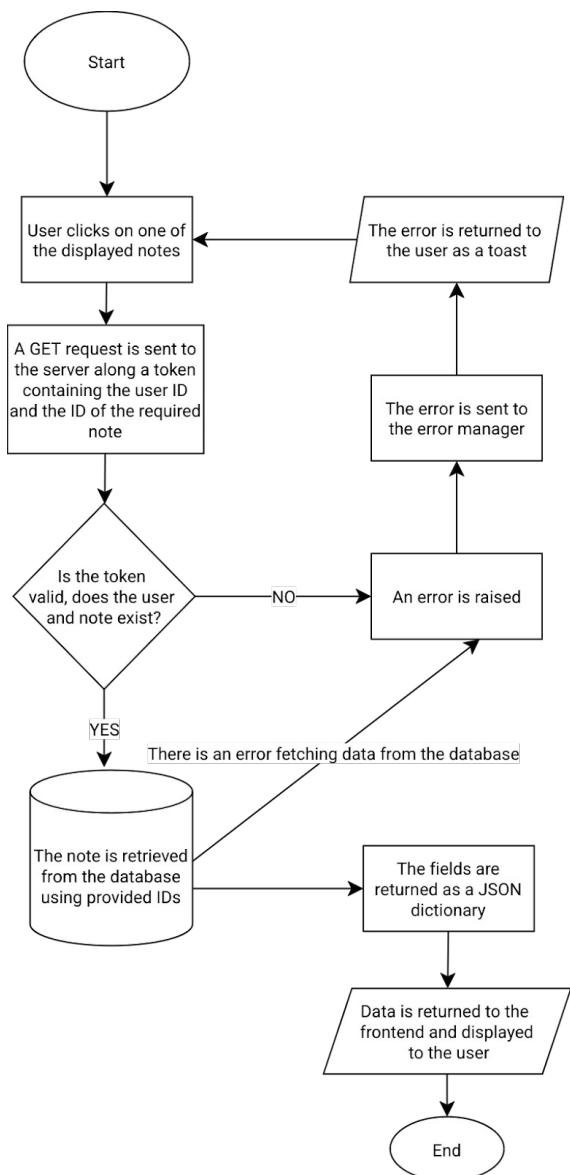
### 2.1.2.3 Fetch all notes of a specific type



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

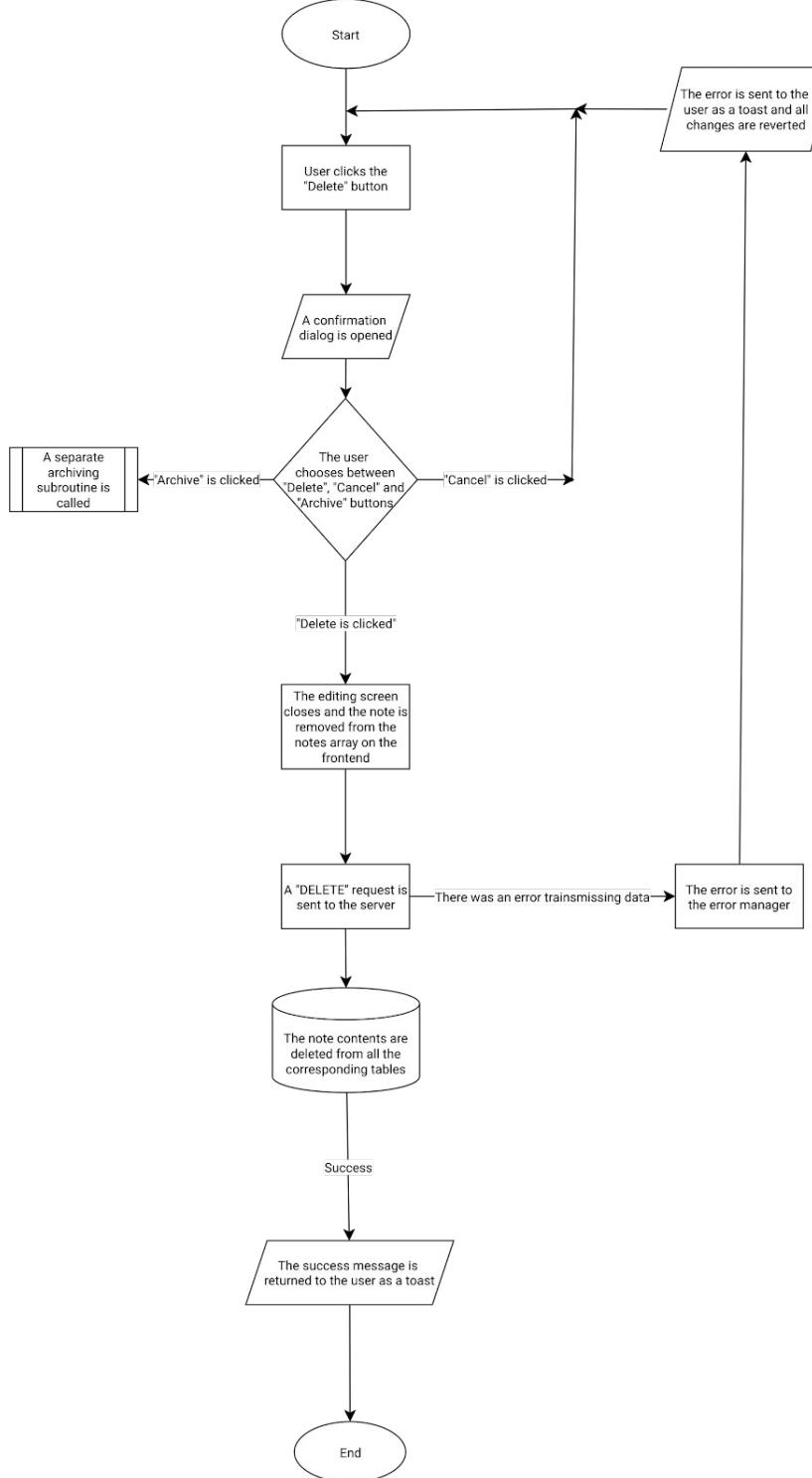
notes from a specific note type are fetched from the main Notes table and connected tables, such as Tasks and Subtasks for a Tasks note type. If the retrieval is successful. Since this page would only be used to find the correct note and then click on it to open a note editing page, there is no need 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,

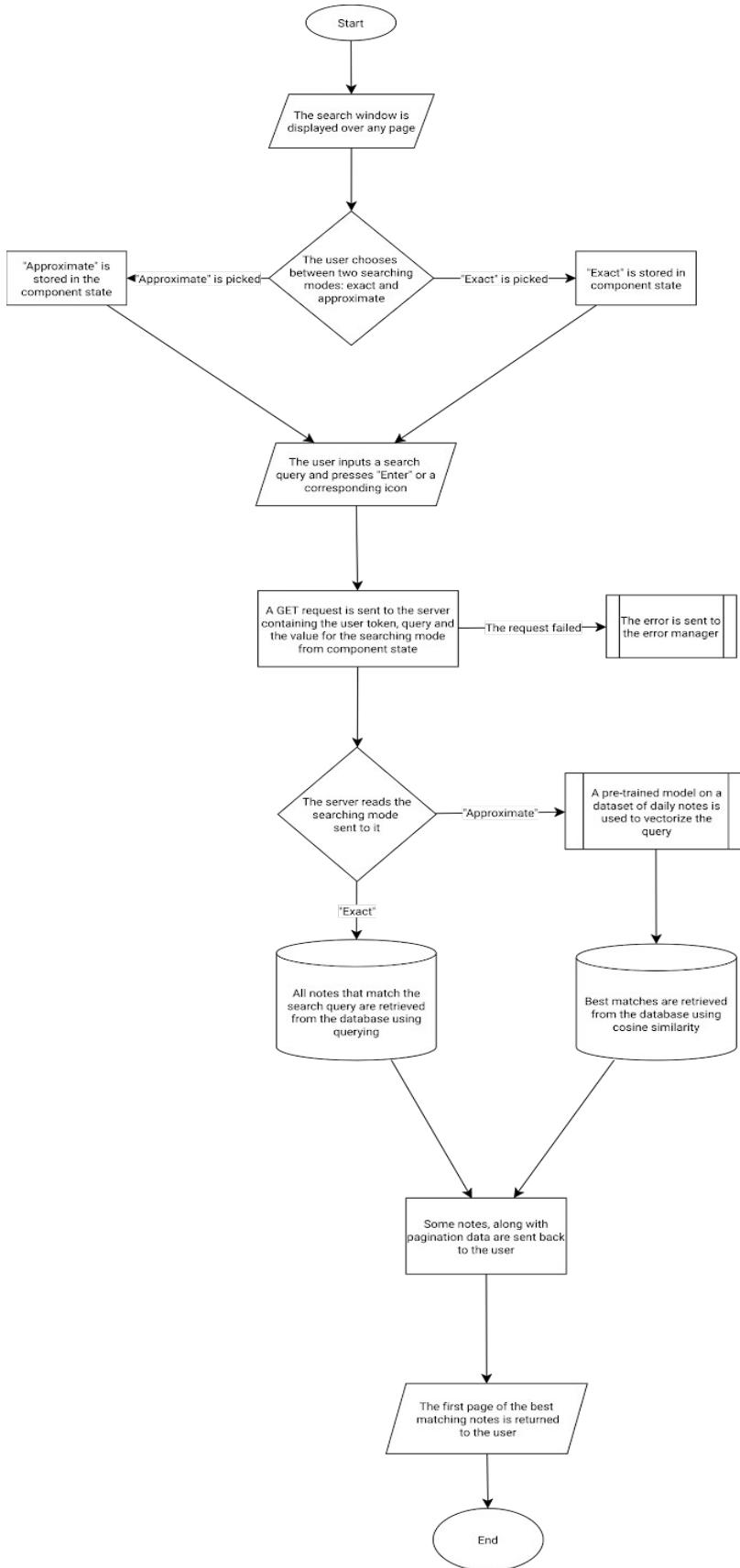
containing a user id and the ID of the note to be deleted. If the token is valid and the user is the owner of the note, its data is deleted from all corresponding tables. Finally, the success or error message is returned to the user as a toast.

### **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:

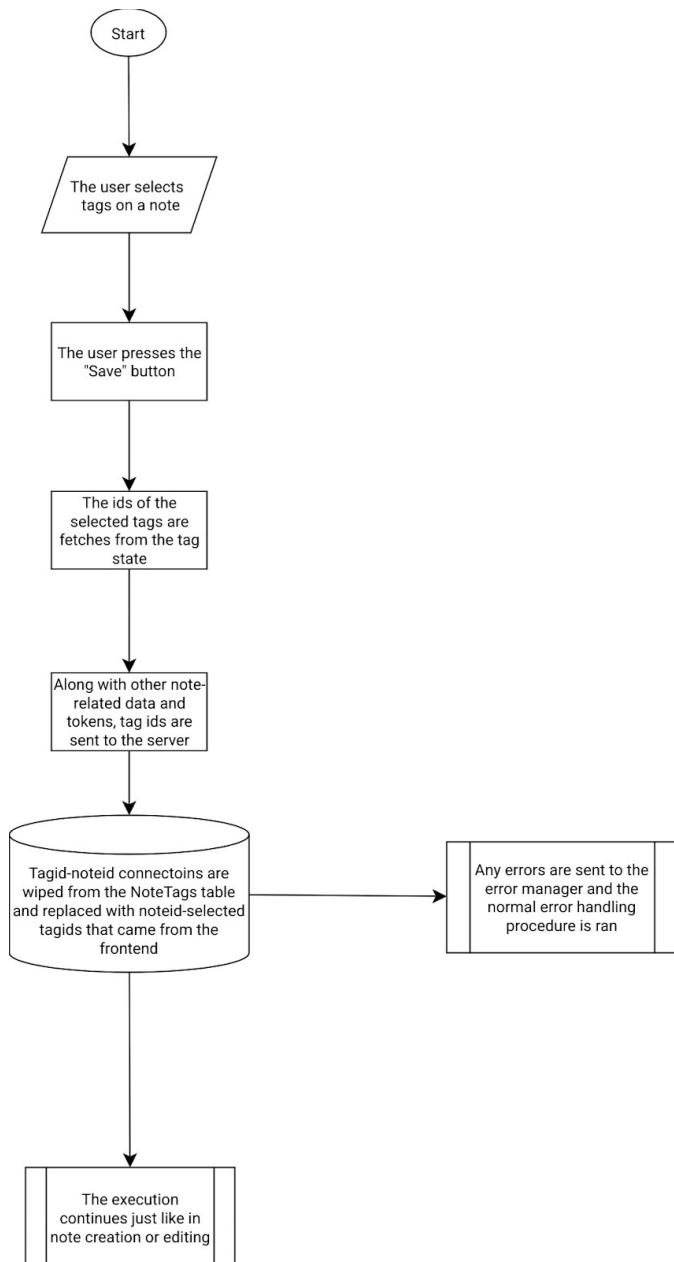
- a)**    “Exact” search mode. The query will be used to look up title or content fields in the database using SQL querying
- b)**    “Approximate” search mode. The query will be vectorized with a pretrained model and its vector representation will be compared to the stored vector representations of 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.



### 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 accessed 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.

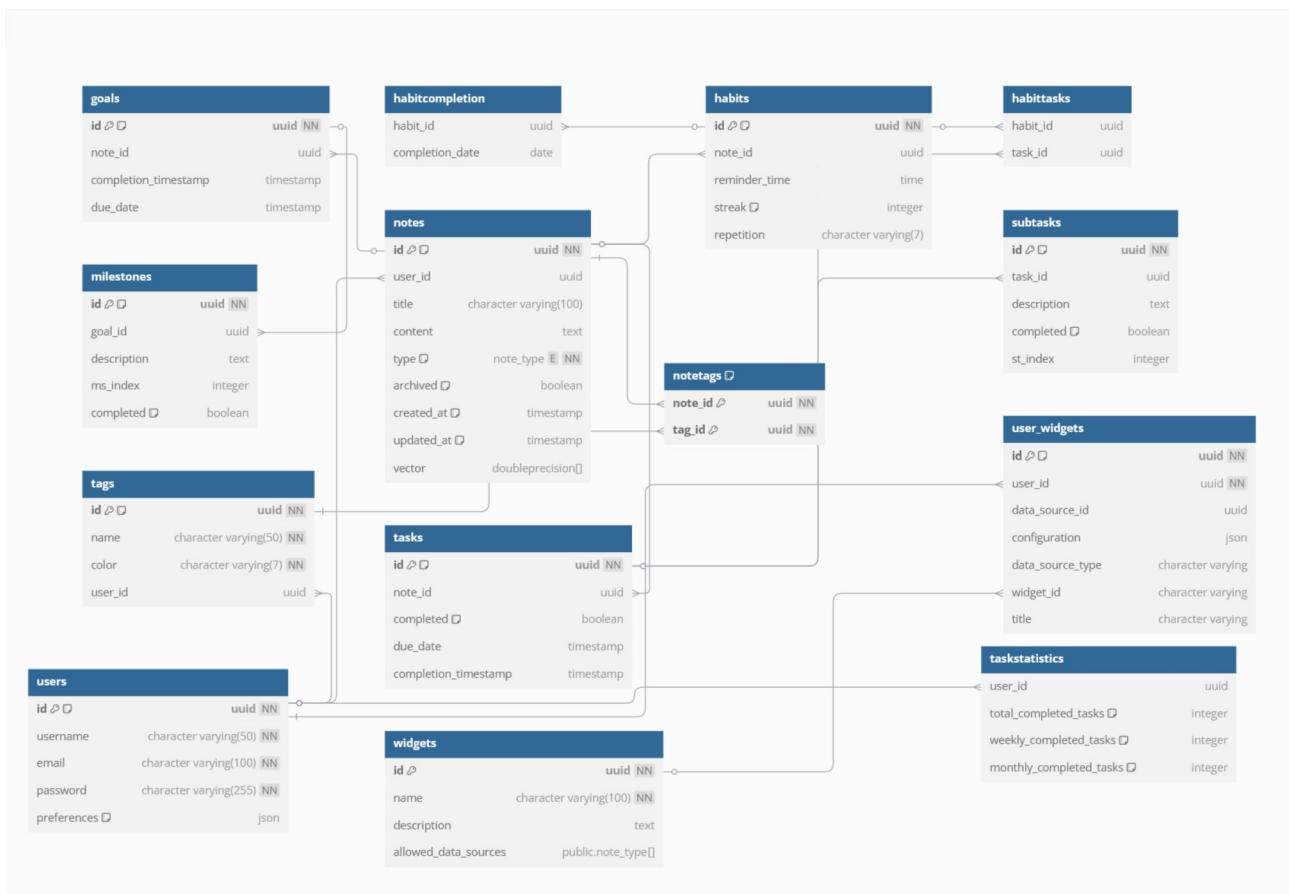
## **2.2.2 Database relationships**

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 it contains confidential configuration settings, such as database connection credentials and other environment 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.

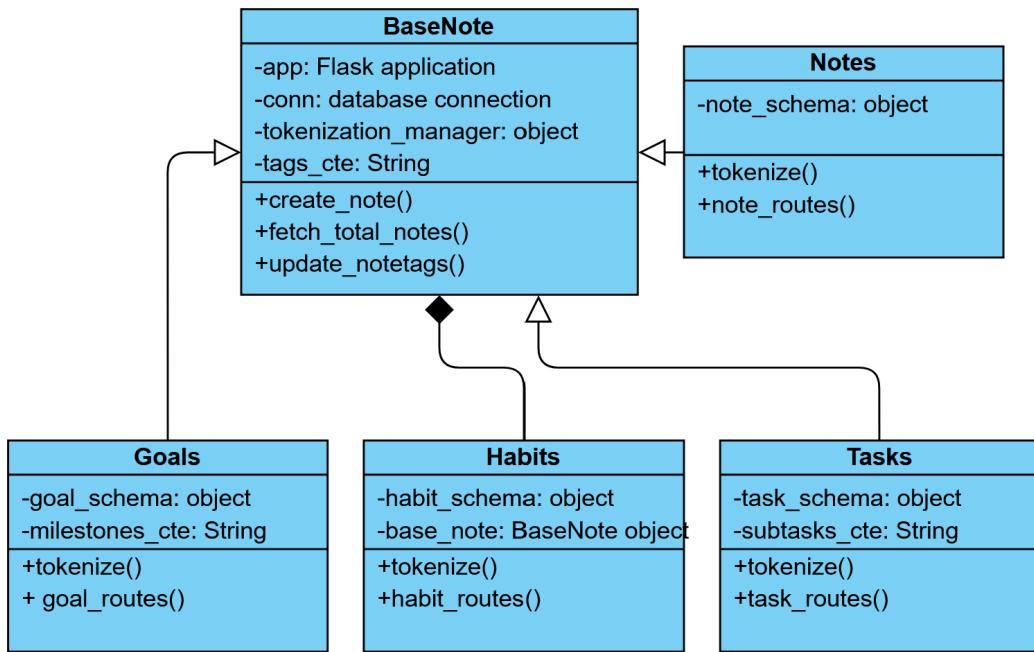
## **2.2.4. OOP**

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.

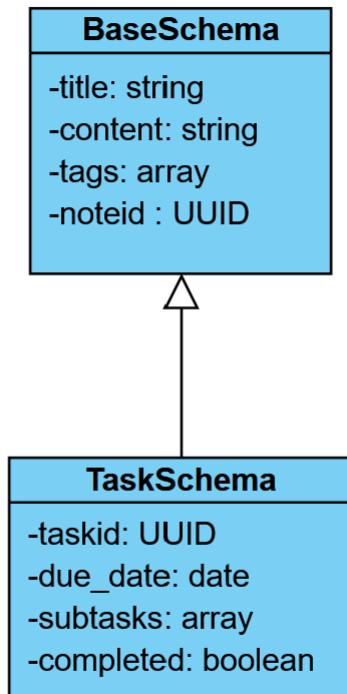
However, I noticed that most notes share similar base characteristics, such as tags, content and title and decided to use classes to mix and match actual API components to create an object that will be able to work with a specific note type. By doing that, I will encapsulate all logic for a note type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

## 2.4 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.

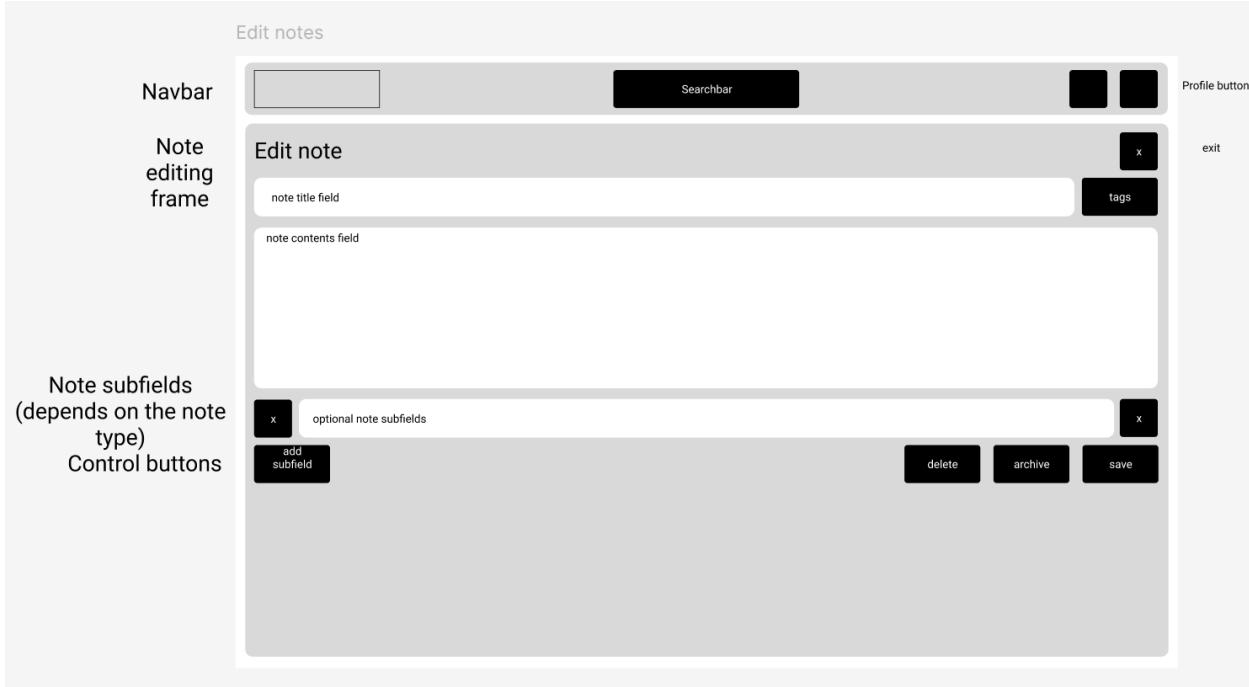
#### 2.4.3.1.1 Viewing all notes



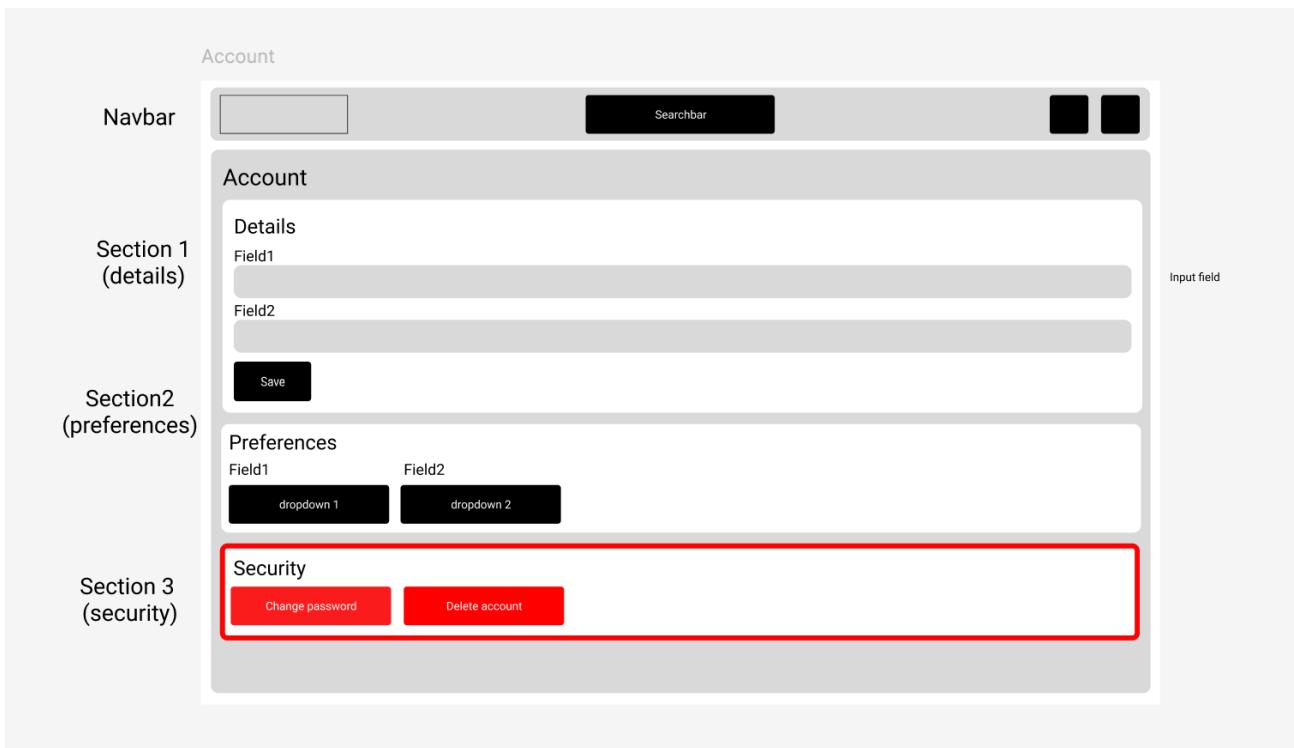
This wireframe represents the screen which will be containing all of user's notes of a specific type, along with pagination controls, ability to create new notes, view their previews (such as a cropped title and content, if they exceed a certain character limit) and open each note in a separate screen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

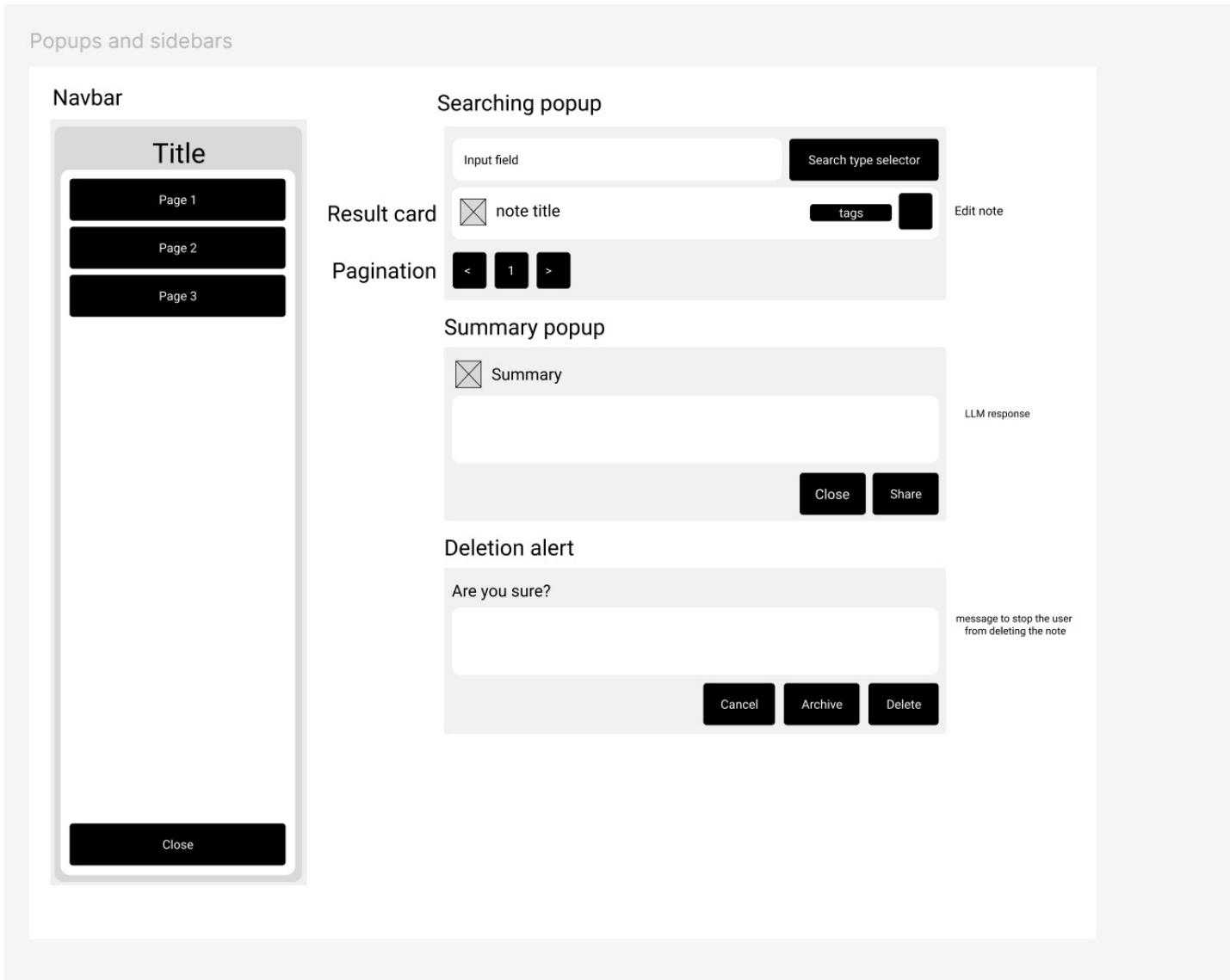


### 2.4.3.2 An account page where the user can edit their details



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.

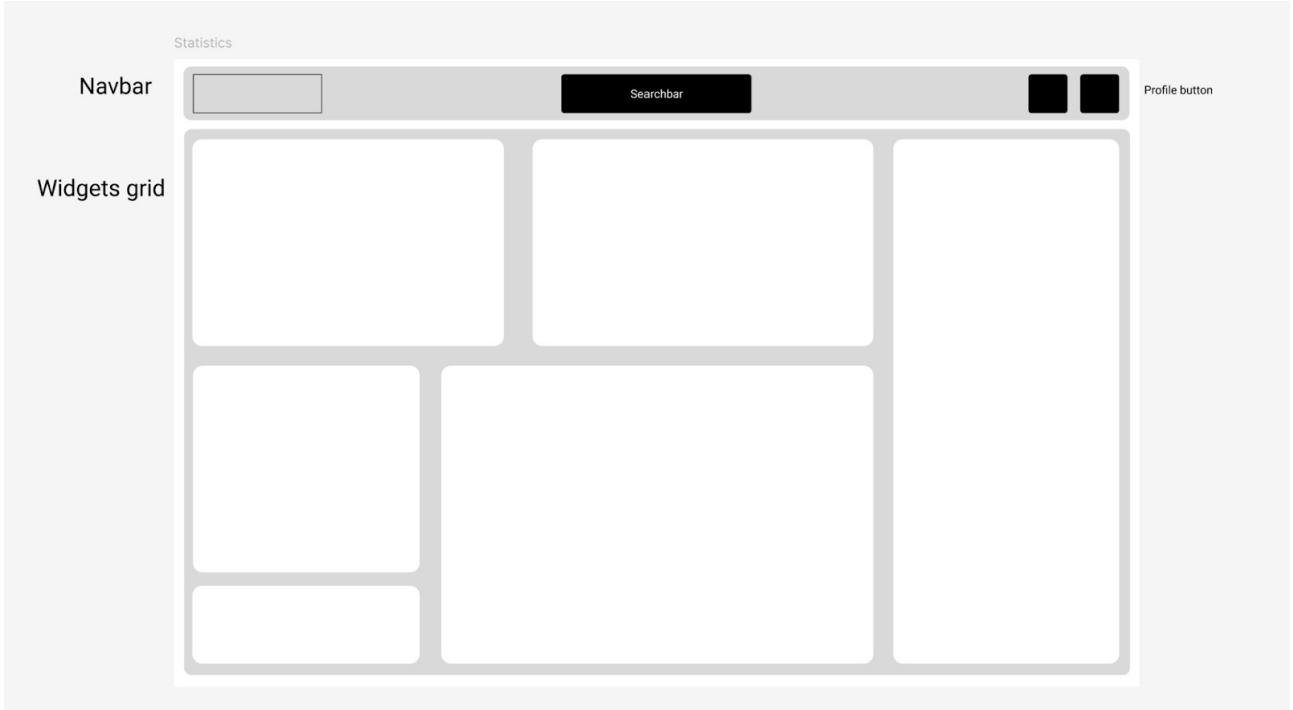
### 2.4.3.3 A sidebar and other popups, such as alerts or search widgets



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:



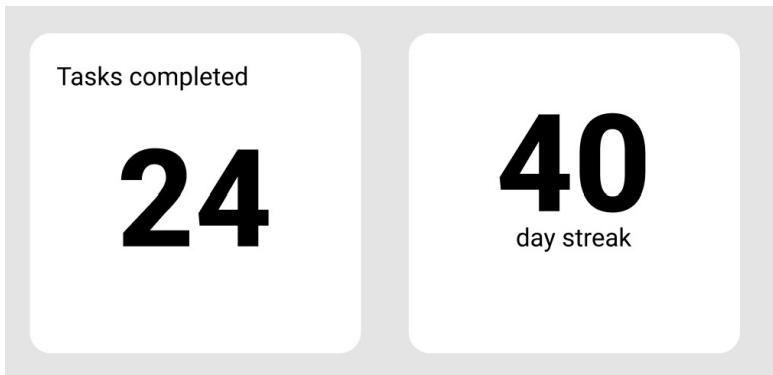
The page can be a grid with customizable widgets of various sizes. There are plenty of widget options, but most of them can be simplified to several types:

#### 2.4.3.4.1      Charts



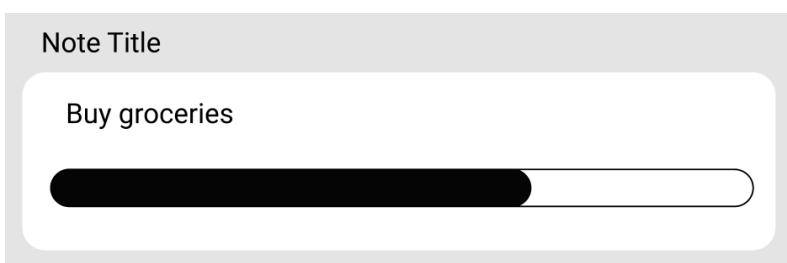
Charts can take the form of bar or graph charts and may be used to represent data such as task completion, activity, streaks etc. Depending on the size of the widget, several key features such as the scale may be dynamically added as the user scales the widget.

#### 2.4.3.4.2 Numbers

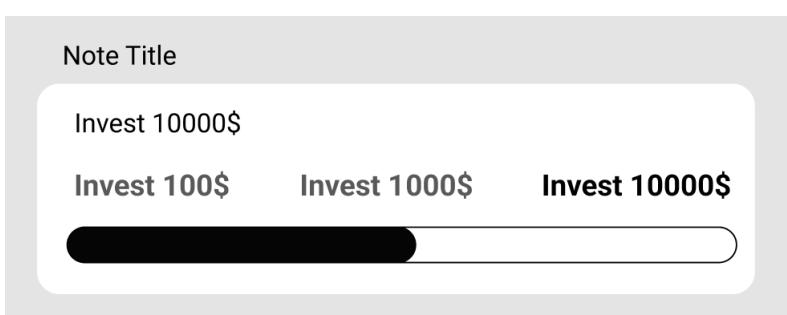


The smallest possible gridcell may be used to store a simple numerical value, such as the numbers provided above. In the examples, the widgets represent the total of tasks completed for a user and the length of their streak of using the app or a specific habit.

#### 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.



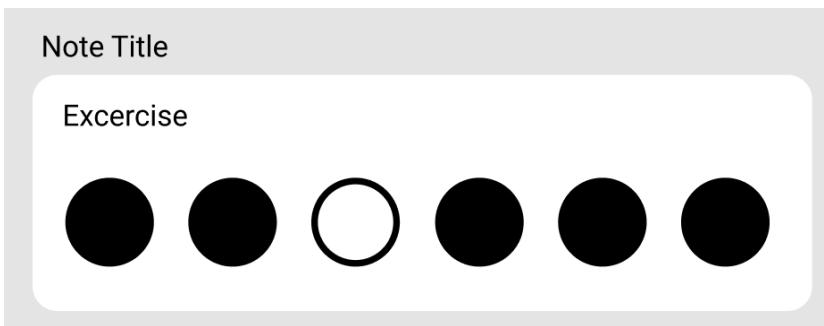
This idea may be expanded with the addition of milestones for goals, which may further motivate the user. Additionally, this example maybe used to highlight the ease of adapting these components for different sizes by stacking the milestones vertically:



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**

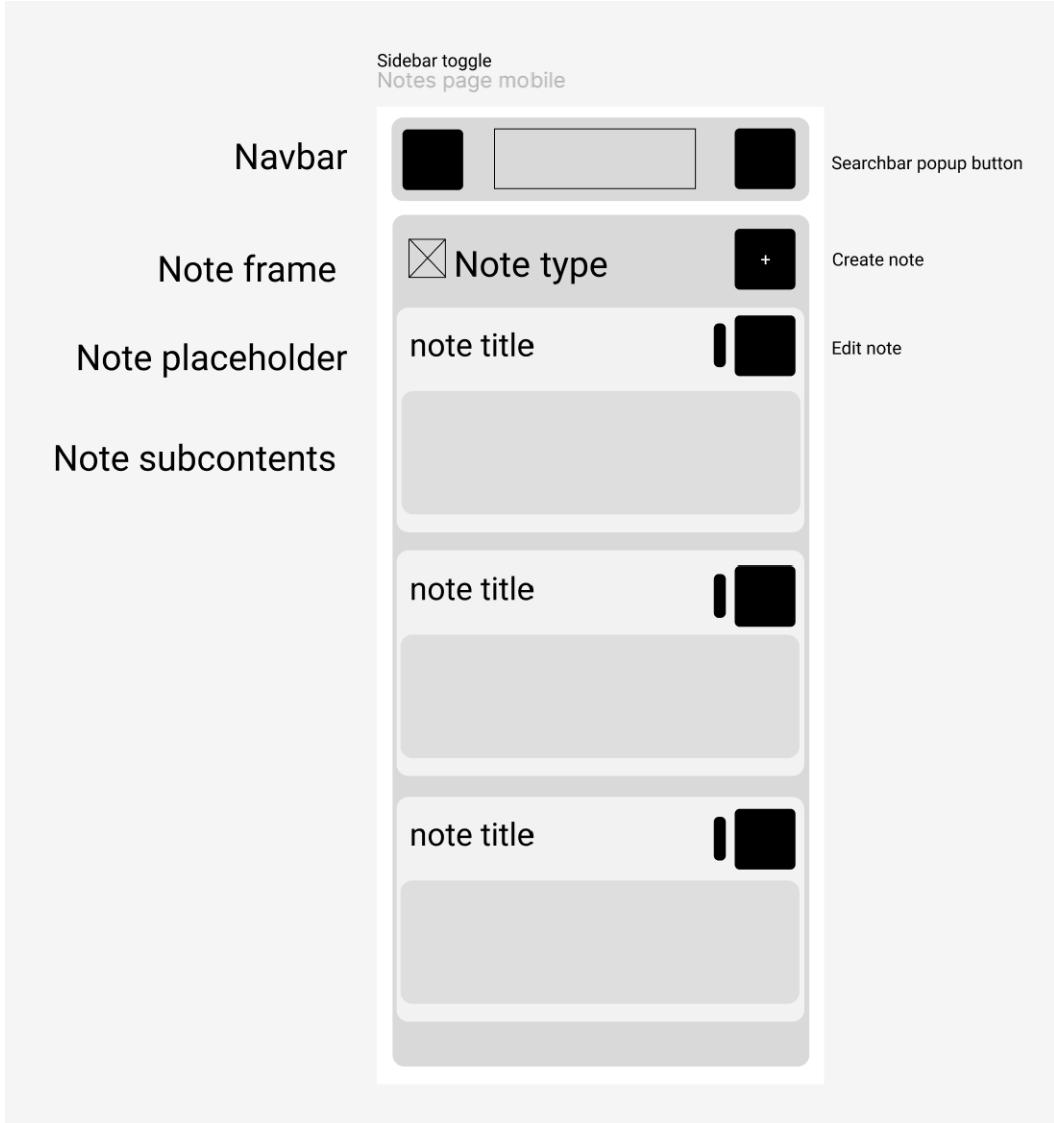
The following type can only be used to very specific notes that require daily commitment, allowing the user to visualise their progress in a simple way.



#### **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.

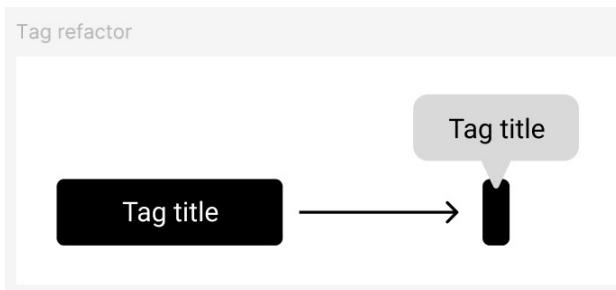
Several elements of the interface have to be rearranged in order to keep everything readable. The biggest change was done to the navbar, where the theme toggle and account buttons have been moved to the sidebar to make more space for the logo on the narrower navbar.



## Navbar mobile

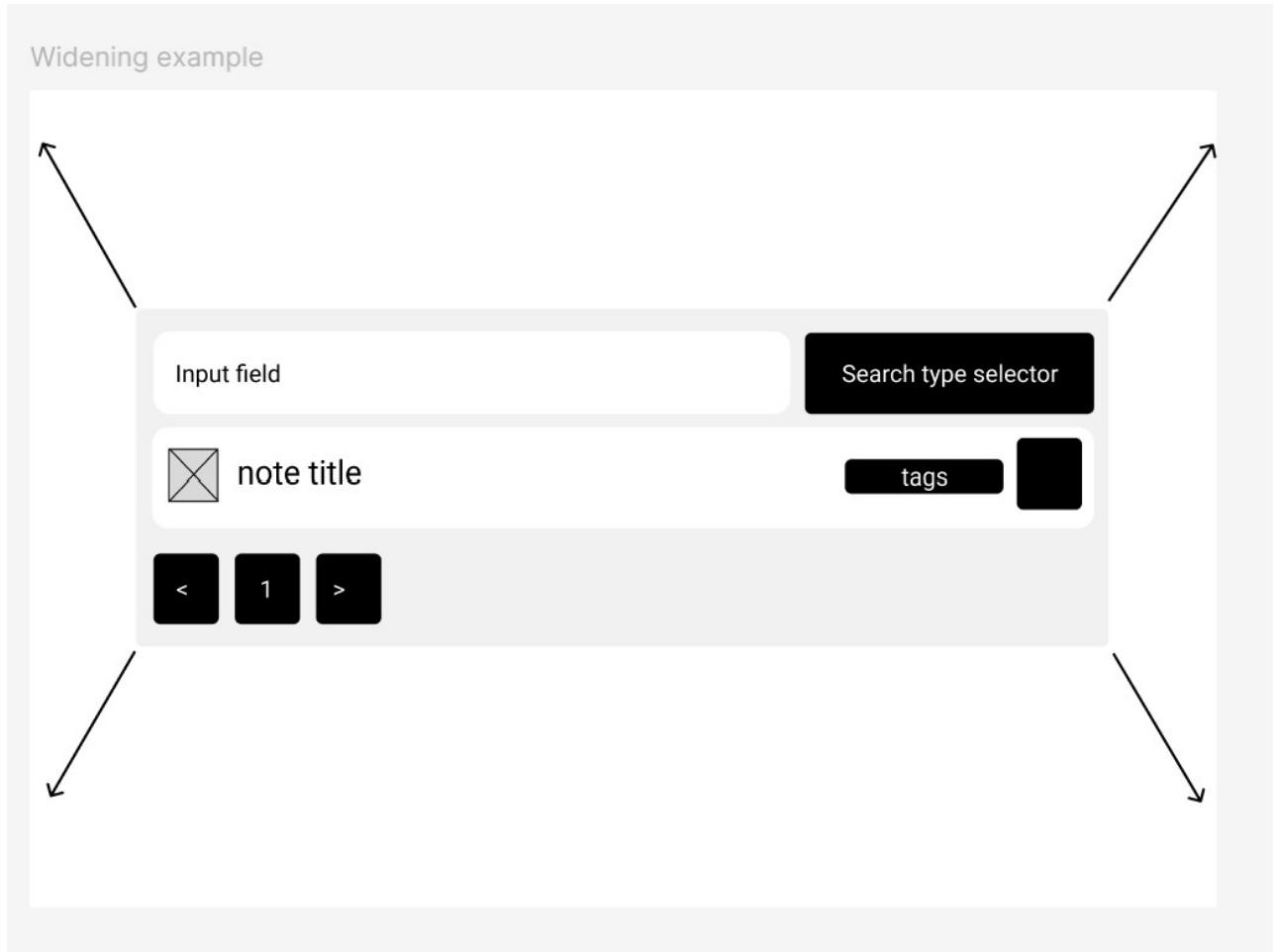


A similar refactoring happened to the note cards themselves. For example, there are less characters displayed per line as the screen shrinks to make the note cards grow in height instead of width. Moreover, additional whitespace can be found by shrinking the tag labels and just leaving small colored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

### **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={"/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 managers.

```

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 %
            """", (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 %
    """, (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'][0:50] + '...' 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 = %
        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 calendar 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}, delimited 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 schema is 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

/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
            """)
            note = cur.fetchone()
            if note:
                self.recents_manager.add_note_to_recents(note['note_id'])
                return jsonify(note), 200
            else:
                return jsonify({'message': 'Note not found'}), 404
    except Exception as e:
        self.conn.rollback()
        raise

```

```

        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 committed 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 available 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
    """)

    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('noteld')

        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('noteld')
        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 belongs 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
        """.format(user_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 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 processing.
- 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_id,
            widget_data['data_source_type'],
            widget_data['data_source_id'],
            json.dumps(widget_data['configuration']),
            user_id,
            user_id
        ))
        self.conn.commit()

```

```

    """",
        (
            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 required 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 into 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 runs 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 representations 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 two 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 relevant 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 continuously monitors the heap queue, processes 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 records 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, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1],  # users
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],  # taskstatistics
    [1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0],  # notes
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0],  # tags
    [0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0],  # goals
    [1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0],  # goalhistory
    [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0],  # milestones
    [0, 0, 1, 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, 0], # habittasks
[0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0], # tasks
[0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0], # subtasks
[0, 0, 1, 1, 0, 0, 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, 0, 0], # user_widgets
])

```

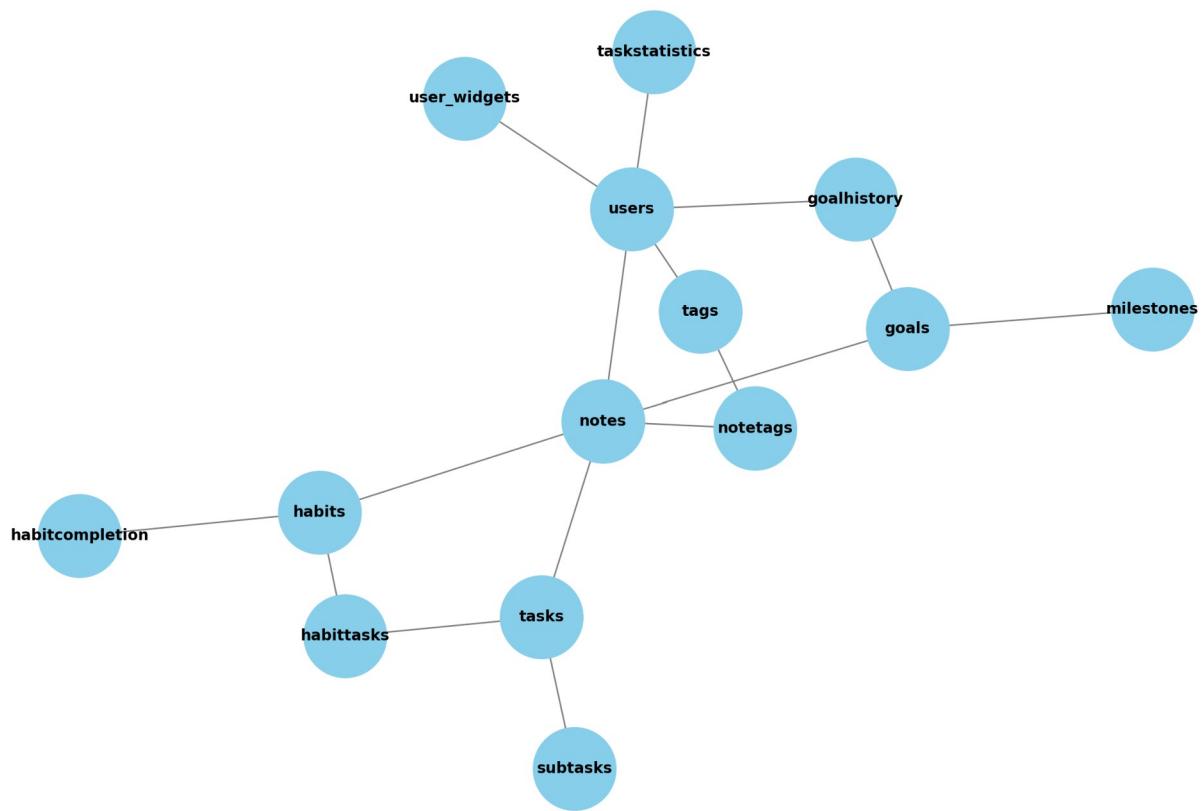
In the code, the matrix is represented as a 14x14 NumPy array, where each “1” represents a connection between two tables. Using this matrix, I can use a library like networkx to plot the relationship”

```

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 = {
  "users": "DELETE FROM users",
  "taskstatistics": "DELETE FROM taskstatistics",
  "notes": "DELETE FROM notes",
  "tags": "DELETE FROM tags",
  "goals": "DELETE FROM goals",
  "goalhistory": "DELETE FROM goalhistory",
  "milestones": "DELETE FROM milestones",
  "habits": "DELETE FROM habits",
  "habitcompletion": "DELETE FROM habitcompletion",
  "habittasks": "DELETE FROM habittasks",
  "tasks": "DELETE FROM tasks",
  "subtasks": "DELETE FROM subtasks",
  "notetags": "DELETE FROM notetags",
  "user_widgets": "DELETE FROM user_widgets"
}
  
```

```

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

```

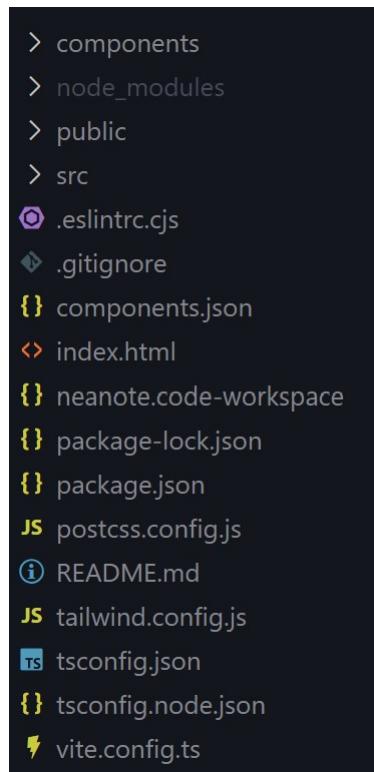
The `delete_notes_with_backoff` function is almost the same, but it targets note-related data instead of user data. It takes a note ID and a stack of nodes as input, instead of user ID. The SQL queries for note deletion are stored in the `NOTE_DELETE_QUERIES` dictionary. The function follows the same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

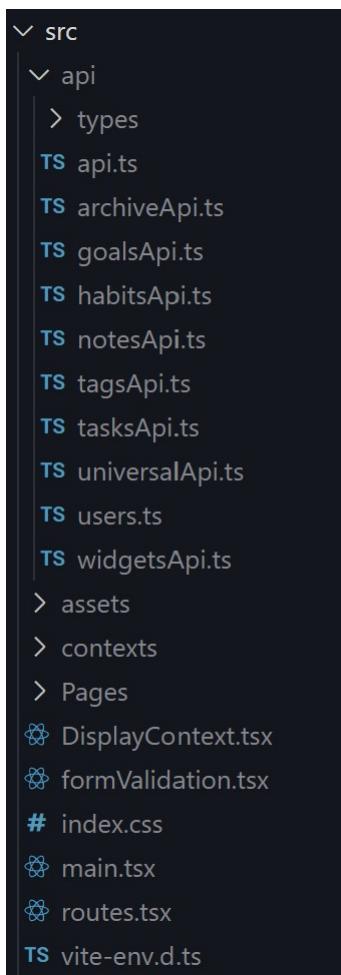
Due to the nature of React, there are many TypeScript, HTML, TSX files containing various code, such as the building elements of components, that are then combined into webpages. In addition to that, each page in my application has several other files containing the logic, predetermined types and backend 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:



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.



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:

```
<!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-collapse": "^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 throughout 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/DisplayContext.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 resize 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/>} />
    </Routes>
  </Router>
)

```

```

<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 repetitive. 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 dictionaries 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**

The types files consolidate interfaces for each module, specifying datatypes or allowed values for each field. They can be treated as CREATE SQL statements, which also determine the datatypes of each field in the database, but in the setting of a frontend app. Again, for simplicity, some types will be omitted from this section. I will just describe the principles of operation of these files with a few examples. Anything else may be found on GitHub or at the end of this file.

#### **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 during development, helping to catch errors early and improve code clarity and maintainability. The application will not error out if the datatype in an interface does not correspond to the actual datatype of the data, but if such interfaces do not correspond, I will see an error in my IDE. 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 ./src/api/types/userTypes.ts**

Generally, there are two interface types that each file contains: the actual data interface and the response interface, which combines the data interface with extra fields, like messages, status codes or pagination settings, that are returned alongside the data from the backend and also must be type-safe.

```

export interface UserSettings {
    username: string;
    email: string;
    password?: string
    preferences: {

```

```

        theme?: 'light' | 'dark' | 'system';
        model?: string;
        //add other fields later
    }
}

export interface UserGetResponse {
    data: UserSettings;
    message: string;
}

export interface UserLoginResponse {
    preferences: UserSettings['preferences'];
    message: string;
}

```

An example of this is `UserGetResponse` and `UserLoginResponse`, which are used to check the responses for a user login and a GET request to get user data in the settings page. Some fields can be optional, such as the `theme` field, so they are denoted by an "?" at the end. Additionally, `theme`, instead of having a, let say, a "string" datatype, is instead only allowed to have 'light' or 'dark' or 'system', which are the themes used in the theme provider.

### 3.2.4.3 ./src/api/types/tagTypes.ts

```

import { UUID } from "crypto";

export interface Tag {
    tagid: UUID;
    name: string;
    color: string;
};

export interface TagResponse {
    data: Tag[] | undefined;
    message: string;
}

export interface TagCreateResponse {
    data: {
        id: UUID
    };
    message: string;
}

```

Tags do not have many components and API calls, so the interface `Tag` just mirrors the datatypes in the CREATE statement of the `Tags` table. Additionally, it ensures that returned messages are of type `string` and that the `ID` property of the data returned has the `UUID` datatype.

### 3.2.4.4 ./src/api/types/taskTypes.ts

```
import { UUID } from "crypto";
import { Tag } from "./tagTypes";

export interface Subtask {
    subtaskid: UUID;
    description: string;
    completed: boolean;
    index: number;
};

export interface TaskUpdate extends Task {
    updateType: 'add' | 'update' | 'delete';
}

export interface Task {
    noteid: UUID;
    taskid: UUID;

    title: string;
    content: string;
    completed: boolean;
    due_date: Date | undefined;
    subtasks: Subtask[];
    tags: UUID[] | Tag[];
};

export interface TaskResponse {
    task: Task;
    message: string
}

export interface TaskwithIds extends Task {
    tags:UUID[]
}

export interface TaskCreateResponse {
    message: string
    data: {taskid: UUID, noteid: UUID, subtasks: Subtask[]};
}

export interface TaskPreviewResponse {
    tasks: Task[];
    pagination : {
        nextPage: number | null;
        perPage: number;
        total: number;
        page: number;
    };
    message: string;
}
```

Now Tasks have a few extra interfaces. For example, a Task, apart from other properties, also may store an array of subtasks, which also are objects with their own datatypes. This

is why Subtask is followed by a [ ] to denote that it is an array, as well as tags of Tag[ ] type or arrays of Tasks[ ] in response interfaces. If we make a connection to Python classes, "subtasks: Subtask[]" may be considered composition, while some interfaces inherit the parent interface, such as

```
export interface TaskUpdate extends Task {  
    updateType: 'add' | 'update' | 'delete';  
}
```

, which inherits the Task interface and extends its by adding a new field signifying the update type.

Moreover, as mentioned previously, pagination datatypes are also added to responses with paginations, so code knows which datatypes it should expect.

### 3.2.4.5 ./src/api/types/habitTypes.ts

Wrapping up this section, I will show the types file for the Habits module. Here, the Habit interface is also composed of several other interfaces, which simplifies the code and allows me to break each interface down into components, that I can use in other parts of the application.

```
import { UUID } from "crypto";  
import { Tag } from "./tagTypes";  
import { Task } from "./taskTypes";  
  
export interface Habit {  
    habitid:UUID;  
    noteid:UUID  
  
    title: string;  
    content:string  
    reminder: ReminderTime;  
    streak: number;  
    tags: UUID[] ;  
    linked_tasks : Task[]  
    completed_today: boolean;  
  
    isNew?: boolean;  
}  
  
export interface HabitPreview {  
    habitid:UUID;  
    noteid:UUID  
  
    title: string;  
    content: string;  
    streak: number;  
    completed_today: boolean;  
    tags: UUID[] ;  
}  
  
export interface HabitCreateResponse {
```

```
    message: string
    data: {habitid: UUID, noteid: UUID};
}

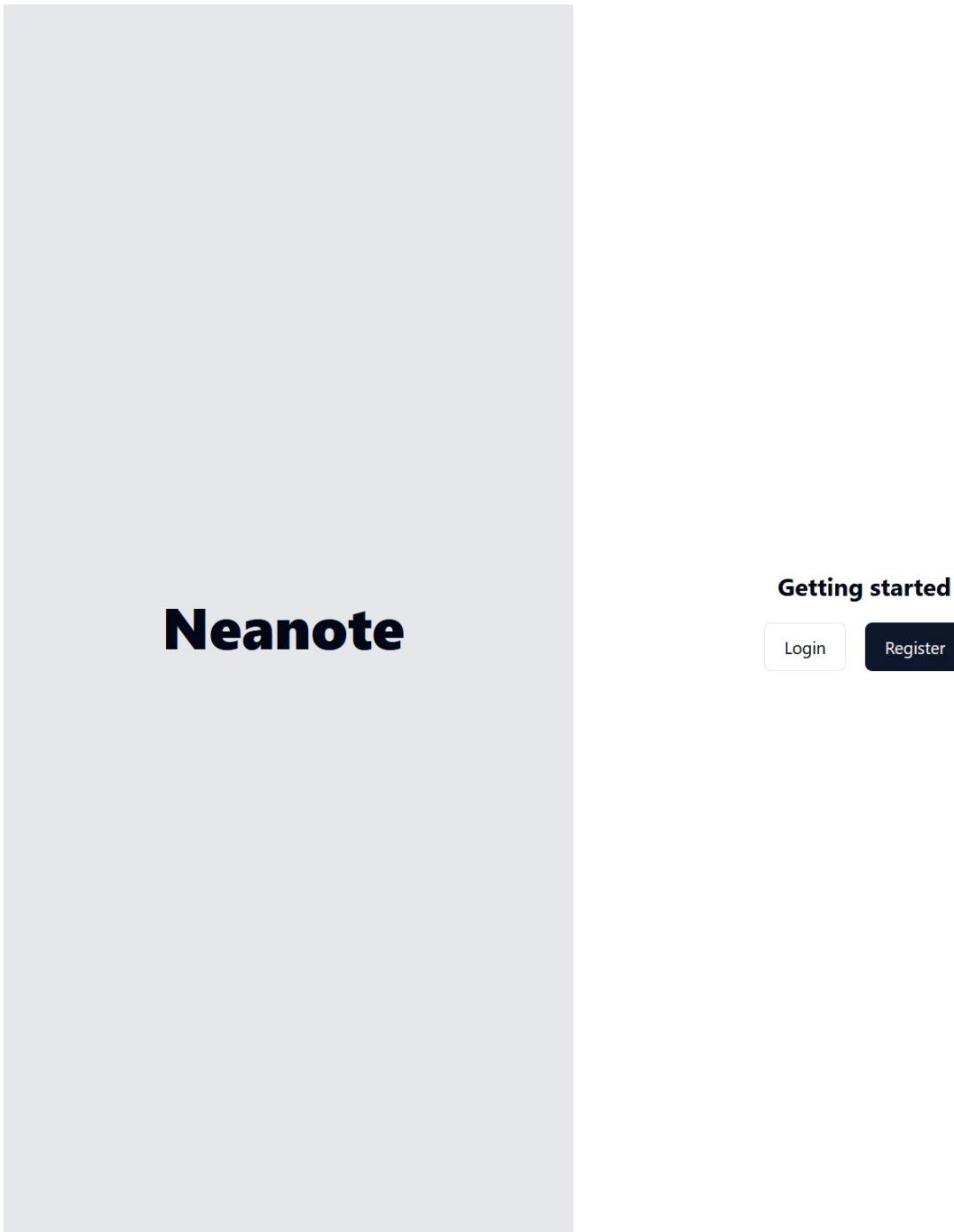
export interface HabitResponse {
    message: string
    data: Habit
}

export interface HabitPreviewResponse {
    message: string
    data: HabitPreview[];
    pagination: {
        total: number;
        per_page: number;
        page: number;
        next_page: number;
    }
}

export interface ReminderTime {
    reminder_time: string | undefined;
    repetition: string | undefined;
}
```

### 3.2.5. Pages

#### 3.2.5.1 Landing



The landing page of my notetaking app is purely decorative. It acts as a place where any unverified or logged out users end up at and provides an opportunity for expansion of the project and marketing it by adding reviews, pricelists and other information for the Internet to see. However, this is beyond the scope of the current project. In this implementation, the landing page lets users to log in or register into the app.

```
import React from 'react'  
import { Link } from 'react-router-dom'  
import { Button } from '../../components/@/ui/button'  
  
function Landing() {
```

```

return (
    <div className='flex'>
        <div className="w-1/2 flex justify-center items-center min-h-screen bg-gray-200">
            <h1 className="text-5xl font-extrabold ">
                Neanote
            </h1>
        </div>
        <div className=" flex w-1/2 justify-center min-h-screen bg-background " >
            <div className="flex flex-col justify-center items-center">
                <h1 className='text-xl pt-2 font-bold'>Getting started</h1>
                <div className="flex flex-row gap-4 p-4">
                    <Link to="/login">
                        <Button variant={'outline'}>Login</Button>
                    </Link>
                    <Link to="/register">
                        <Button>Register</Button>
                    </Link>
                </div>
            </div>
        </div>
    </div>
)
}

export default Landing

```

Additionally, it provides a simple example of how a React component looks like. The code uses React's JSX syntax to define the structure of the page. The outermost div with the class name "flex" establishes a flexible layout, arranging its children side-by-side. Inside this, there are two main div elements, each occupying half of the screen width ("w-1/2"). The left div displays the "Neanote" title, centering it both horizontally and vertically using "flex justify-center items-center" and setting a minimum height to fill the screen. It also has a light gray background. The right div provides the login and register options, also centering its content and using the application's background color.

The login and register options are presented within a nested div that arranges them in a row with some spacing and padding. The Link component (from the react-router library) is used to create clickable links that navigate the user to the "/login" and "/register" routes without the unaesthetic blue hyperlink we commonly see. The Button component is used to render the login and register buttons, with different visual styles applied using the "variant" prop. The "Login" button is styled as an outline button, while the "Register" button uses the default style. Both of these styles are provided by the shadcn library.

### 3.2.5.2 Register

If the user does not have an account, they are sent from the landing page to the registration page.

---

# Neanote

## Register

Username

Email

Password

New to Neanote? Log in

**Submit**

The registration page allows users to create a new account. It consists of two main components: `Register.tsx` and `useRegister.tsx`.

### 3.2.5.2.1        [./src/Pages/Register/Register.tsx](#)

This component defines the user interface for the registration page. It uses React Hook Form and Zod for form validation.

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

```

import React from 'react';
import { useForm } from "react-hook-form";
import { z } from 'zod';
import { Button } from "../../components/@/ui/button";
import { Form, FormControl, FormField, FormItem, FormMessage } from
'../../components/@/ui/form';
import { Input } from "../../components/@/ui/input";
import { Label } from "../../components/@/ui/label";
import { registerFormSchema } from '../../formValidation';
import { Link, useNavigate } from 'react-router-dom';
import { useRegister } from './useRegister';

function Register() {
    const {formHandler, register} = useRegister()
    const navigate = useNavigate()
    // Defines the form
    const form = useForm<z.infer<typeof registerFormSchema>>({
        resolver: zodResolver(registerFormSchema),
        defaultValues: {
            username: "",
            password: "",
            email: ""
        },
    })
    //Defines the form submit
    function onSubmit(values: z.infer<typeof registerFormSchema>) {
        console.log(values)
        formHandler(values)
        register()
        navigate('/get-started')
    }

    return (
        <div className=' p-3 '>
            <div className='items-center flex-col flex min-h-screen min-w-full
bg-background max-h-screen rounded-xl border-[2px]'>
                <Link to="/get-started">
                    <h1 className="text-4xl pb-40 pt-5 font-extrabold ">
                        Neanote
                    </h1>
                </Link>
                <h1 className='text-2xl pt-4 pb-3 font-bold'>Register</h1>
                <Form {...form}>

                    <form onSubmit={form.handleSubmit(onSubmit)} className="space-y-4
w-[250px]">
                        <div className='space-y-1'>
                            <FormField
                                control={form.control}
                                name="username"
                                render={({ field }) => (
                                    <FormItem>
                                        <Label>Username</Label>
                                        <FormControl>
```

```

        <Input
          {...field}
        />
      </FormControl>
      <FormMessage />
    </FormItem>)))/>
  </div>
<div className='space-y-1'>
  <FormField
    control={form.control}
    name="email"
    render={({ field }) => (
      <FormItem>
        <Label>Email</Label>
        <FormControl>
          <Input
            {...field}
          />
        </FormControl>
        <FormMessage />
      </FormItem>)))/>
    </div>
<div className='space-y-1'>
  <FormField
    control={form.control}
    name="password"
    render={({ field }) => (
      <FormItem>
        <Label>Password</Label>
        <FormControl>
          <Input
            {...field}
          />
        </FormControl>
        <FormMessage />
      </FormItem>)))/>
    </div>

  <Link to="/login">
    <p className='text-sm text-foreground'>New to Neanote?
      Log in</p>
    </Link>
    <Button type="submit">Submit</Button>
  </form>
</Form>
</div>
</div>
)
}

export default Register

```

The component uses react-hook-form to manage the form state and validation. zodResolver is used to integrate Zod schema for validation. The form consists of three fields: username, email, and password. When the user presses the “submit” button, the onSubmit function is called, which calls the formHandler and register functions from the useRegister hook, and navigates to the /get-started route.

### 3.2.5.2.2        [./src/Pages/Register/useRegister.tsx](#)

In React, all files starting with “use” are called hooks. Essentially, hooks allow you to use state and other React capabilities without needing to write class components. This means that function components, which were traditionally simpler and stateless, can now manage their own state, perform side effects (like calling an api mentioned previously), and access context. This hook manages the registration logic. It uses Zustand for state management and calls the register component of the users API to send a registration request to the backend. The types of each function and constant are passed into the store through <RegisterState> for type safety.

```
import Cookies from 'js-cookie';
import { create } from 'zustand';
import users from '../api/users';

type RegisterState = {
    form: {
        username: string;
        password: string;
        email: string
    };
    formHandler: (form) => void;
    register: () => Promise<boolean>;
};

export let useRegister = create<RegisterState>((set, get) => ({
    form: {username: '', password: '', email: ''},
    formHandler: (form) => {
        set({form})
    },
    register: async () => {
        let {form} = get();
        let response = await users.register(form);
        if(response){
            return true
        } else {
            return false
        }
    }
}))
```

### 3.2.5.3 Login

Alternatively, the user has an opportunity to opt to log in instead, providing an exit point, consistent with good design principles.

The login page allows users to log in to their existing account. It consists of two main components: Login.tsx and useLogin.tsx.

Neanote

Login

Username

Password

Already have an account? [Register](#)

Submit

### 3.2.5.3.1        ./src/Pages/Login/Login.tsx

```
import { zodResolver } from '@hookform/resolvers/zod';
import React from 'react';
import { useForm } from "react-hook-form";
import { Link, useNavigate } from 'react-router-dom';
import { z } from 'zod';
import { Button } from "../../components/@ui/button";
import { Form, FormControl, FormField, FormItem, FormMessage } from
'../../../../../components/@ui/form';
import { Input } from "../../../../../components/@ui/input";
import { Label } from "../../../../../components/@ui/label";
import { loginFormSchema } from '../../formValidation';

import Cookies from 'js-cookie';
import { useLogin } from './useLogin';
import { useTheme } from '../../../../../components/providers/theme-provider';

function Login() {
    const {formHandler, login} = useLogin()
    const {setTheme} = useTheme()
    const navigate = useNavigate()
    // Defines the form
    const form = useForm<z.infer<typeof loginFormSchema>>({
        resolver: zodResolver(loginFormSchema),
        defaultValues: {
            username: "",
            password: ""
        },
    })
}

async function onSubmit(values: z.infer<typeof loginFormSchema>) {
    if (!Cookies.get('token')) {
        formHandler(values);
        login().then(loginResult => {
            if (loginResult) {
                setTheme(loginResult)
                navigate("/")
            }
        })
    }
}

return (
    <div className=' p-3 '>
        <div className='items-center flex-col flex min-h-screen min-w-full
bg-background max-h-screen rounded-xl border-[2px]'>
            <Link to="/get-started">
                <h1 className="text-4xl pb-40 pt-5 font-extrabold ">
                    Neanote
                </h1>
            </Link>
            <h1 className='text-2xl pt-4 pb-3 font-bold'>Login</h1>
        </div>
    </div>
)
```

```

<Form {...form}>

<form onSubmit={form.handleSubmit(onSubmit)} className="space-y-4
w-[250px]">

    <div className='space-y-1'>
        <FormField
            control={form.control}
            name="username"
            render={({ field }) => (
                <FormItem>
                    <Label>Username</Label>
                    <FormControl>
                        <Input
                            {...field}
                        />
                    </FormControl>
                    <FormMessage />
                </FormItem>
            )}/>
    </div>
    <div className='space-y-1'>
        <FormField
            control={form.control}
            name="password"
            render={({ field }) => (
                <FormItem>
                    <Label>Password</Label>
                    <FormControl>
                        <Input
                            {...field}
                        />
                    </FormControl>
                    <FormMessage />
                </FormItem>
            )}/>
    </div>
    /* Add more form fields here... */
    <Link to="/register">
        <p className='text-sm pt-2 text-foreground'>Already
have an account? Register</p>
    </Link>
    <Button type="submit">Submit</Button>
</form>
</Form>
</div>
</div>
)
}

export default Login

```

This component defines the user interface for the login page. It uses React Hook Form and Zod for form validation. It also uses the `useLogin` hook to handle the login logic and the `useTheme` hook to set the theme after successful login.

The component initializes the form using useForm hook from react-hook-form. The zodResolver is used to validate the form against the loginFormSchema defined in src/formValidation.tsx. The onSubmit function is called when the form is submitted. It calls the formHandler function from the useLogin hook to update the form state, and then calls the login function from the useLogin hook to log in the user. If the login is successful, the setTheme function from the useTheme hook is called to set the theme based on the user's preferences, and the user is navigated to the root route. The Link component is part of the react-router library and can wrap any component to make it navigate the user to a specific link upon being clicked.

### 3.2.5.3.2        [./src/Pages/Login/useLogin.tsx](#)

```
import Cookies from 'js-cookie';
import { create } from 'zustand';
import users from '../../../../../api/users';
import { useTheme } from '../../../../../components/providers/theme-provider';

type LoginState = {
  form: {
    username: string;
    password: string;
  };
  formHandler: (form) => void;
  login: () => Promise<'light' | 'dark' | 'system' | false>;
};

export let useLogin = create<LoginState>((set, get) => ({
  form: {username: '', password: ''},
  formHandler: (form) => {
    set({form})
  },
  login: async () => {
    let {form} = get();
    let response = await users.login(form);
    if (response && response.preferences) {
      return (response.preferences.theme ? response.preferences.theme : 'system')
    }
    else {
      showToast("e", 'Invalid username or password') // Fix test 6
      return false
    }
  }
}))
```

The useLogin hook manages the form state. The formHandler function updates the form state with the form data. The login function calls the users.login web service API to log in

the user. The `users.login` API is a simple client-server model that calls a web service API and parses JSON to service the client-server model. The API returns the theme preference of the user, which is then used to set the theme using the `useTheme` hook. Finally, after the user redirect, a simple toast is shown indicating a successful login.

### 3.2.5.4 Account

Now we have reached a more complex page, which may require more explanation.

The screenshot shows the Neanote application's account settings page. At the top, there is a dark header bar with the Neanote logo, a search bar, and two small icons. Below the header, the main content area has a light gray background. It features a title 'Account' with a person icon. Underneath, there are three sections: 'Details', 'Preferences', and 'Security'. The 'Details' section contains fields for 'Username' (with 'test' entered) and 'Email' (with 'test@example' entered, accompanied by a purple asterisk icon). A red error message 'Invalid email address.' is displayed below the email field. A 'Save' button is located at the bottom of this section. The 'Preferences' section has a dropdown menu set to 'Light'. The 'Security' section is highlighted with a red border and contains two buttons: 'Change Password' and 'Delete account'.

By clicking the top right icon in their navbar, the user is sent to the /account endpoint, where they can change their username, email, theme preferences, password and have an option to delete their account. Username and email fields are also validated and an example of an invalid email address is shown on the screenshot.

#### 3.2.5.4.1 [./src/Pages/Account/Account.tsx](#)

```
import React, { useEffect, useState } from 'react'
import { useUser } from './useUser'
import TitleComponent from '../../../../../components/TitleComponent/TitleComponent'
import { FaSave, FaUser } from "react-icons/fa";
import { Label } from '../../../../../components/@/ui/label';
```

```

import { Input } from '../../../../../components/@/ui/input';
import { Button } from '../../../../../components/@/ui/button';
import { Separator } from '../../../../../components/@/ui/separator';
import { PasswordDrawerDialog } from './Components/PasswordDrawerDialog';
import { DeleteAccountDrawerDialog } from './Components/DeleteAccountDrawerDialog';
import { ThemeSelector } from './Components/ThemeSelector';
import { useTheme } from '../../../../../components/providers/theme-provider'

```

We start by importing tools and components. Such components could be React, useEffect, and useState, that handle page updates and state. useUser is a custom hook created to manage user data and will be gone over in the next section. Other components like Label, Input, and Button are styled elements for displaying text, taking user input, and performing actions, respectively. Sometimes, such components are more easily combined into other components like "PasswordDrawerDialog" and "DeleteAccountDrawerDialog" which are specialized components that handle password changes and account deletion by grouping other components and sometimes logic in a reusable manner.

```

function Account() {
  const { currentUser, getUser, updateCurrentUser, pendingChanges,
validationErrors, loading, handleUpdateDetails, handleUpdatePreferences } =
useUser();
  const [isValidationErrorsEmpty, setIsValidationErrorsEmpty] = useState(true);
  const { setTheme } = useTheme();

  useEffect(() => {
    setIsValidationErrorsEmpty(
      Object.keys(validationErrors).every(key => !validationErrors[key])
    );
    console.log(validationErrors);
  }, [validationErrors]);

  const handleThemeChange = (value) => {
    updateCurrentUser('preferences', { ...currentUser.preferences, theme: value });
    setTheme(value);
    handleUpdatePreferences();
  }

  useEffect(() => {
    getUser();
  }, [getUser]);
}

```

The main part of the code is the Account function. This function defines what the user sees and interacts with on the Account page. It uses the useUser hook to get the user's current information, any changes they've made, and any errors in their input.

useState is used to manage whether there are any errors in the user's input, and useEffect is used to keep this information updated whenever the errors change. Another useEffect is used to fetch the user's data when the page loads. The hook listens to the getUser

function, so it calls it whenever the function is mounted. Lastly handleThemeChange function updates the user's theme preference and applies the new theme.

```
return (
  <>
  <div className="flex flex-row gap-3 items-center pb-2">
    <TitleComponent><FaUser size={'20px'} /> Account</TitleComponent>
  </div>

  <div className='flex flex-col gap-3'>
    <div className='bg-secondary p-3 rounded-xl flex flex-col gap-2'>

      <h2 className='font-bold text-xl'>Details</h2>
      <Separator/>

      <Label htmlFor="username">Username: </Label>
      <Input id="username" className='50vw' value={currentUser?.username} onChange={(e) => updateCurrentUser('username', e.target.value)} />
      {validationErrors['username'] && (
        <Label htmlFor="username" className='text-destructive'>{validationErrors['username']}</Label>
      )}

      <Label htmlFor="email">Email:</Label>
      <Input id="email" value={currentUser?.email} onChange={(e) => updateCurrentUser('email', e.target.value)} />
      {validationErrors['email'] && (
        <Label htmlFor="email" className='text-destructive'>{validationErrors['email']}</Label>
      )}
      {/* <Label htmlFor="password">Password:</Label>
      <Input id="password" type="password" value={currentUser?.password} onChange={(e) => updateCurrentUser('password', e.target.value)} /> */}

      <Button className='gap-2 w-fit' disabled={!pendingChanges || !isValidationErrorsEmpty} onClick={handleUpdateDetails}>
        <FaSave /> {loading ? 'Saving...' : 'Save'}
      </Button>
    </div>
    <div className='bg-secondary p-3 rounded-xl flex flex-col gap-2'>
      <h2 className='font-bold text-xl'>Preferences</h2>
      <Separator />
      <ThemeSelector
        theme={currentUser.preferences.theme}
        onChange={handleThemeChange} />
    </div>
    <div className='border-[2px] border-destructive p-3 rounded-xl flex flex-col gap-2'>
      <h2 className='font-bold text-xl'>Security</h2>
      <Separator/>
      <div className='flex flex-row gap-3'>
        <PasswordDrawerDialog/>
        <DeleteAccountDrawerDialog/>
      </div>
    </div>
  </div>
)
```

```

        </div>
    </div>
</>
);
}

export default Account;

```

The return section of the code describes the visual layout of the page. It's organized into sections using div elements, which act like containers. The page is divided into Details, Preferences, and Security sections. The Details section displays the user's username and email form, allowing them to edit this information. Validation messages are displayed if the user enters incorrect information and have been showcased previously. A Save button lets the user save their changes and is enabled only when there are changes and there are no validation errors using the disabled={!pendingChanges || !isValidationsEmpty} inline comparison. The last two sections contain the theme preference change menu and the buttons that trigger popups containing the UI to change a password or to delete a user's account. Before we discuss the popups, it is necessary to take a look at the logic behind the user. This is where useUser hook comes into play.

### **3.2.5.4.2        ./src/Pages/Account/useUser.tsx**

```

import { create } from "zustand";
import { immer } from "zustand/middleware/immer";
import { UserSettings } from "../../api/types/userTypes";
import users from "../../api/users";
import { PasswordSchema, UserSettingsSchema } from "../../formValidation";

interface Userstate {
    user: UserSettings | undefined;
    getUser : () => Promise<void>;
    deleteUser: (password:string) => Promise<boolean>;

    currentUser: UserSettings
    updateCurrentUser: (key: keyof UserSettings, value: any) => void;
    updateUser: (updateData: object) => void;

    pendingChanges:boolean

    validationErrors: Record<string, string | undefined>;
    validateUser: () => boolean;
    validatePassword: (password: string, newPassword: string) => boolean;

    loading: boolean;
    handleUpdateDetails: () => void;
    handleUpdatePreferences: () => void;
    handleChangePassword: (password: string, newPassword:string, setOpen:(boolean)=>void) => void
}

```

After all necessary imports, the code defines an interface called Userstate. An interface is like a blueprint that describes the data and functions that will be managed by the useUser hook. It includes:

- user: The complete user data, conforming to the UserSettings type.
- getUser(): A function to retrieve user data from the backend.
- deleteUser(password: string): A function to delete a user.
- currentUser: A subset of the user data that is currently being edited or displayed.
- updateCurrentUser(key, value): A function to update a specific property of the currentUser data.
- updateUser(updateData): A function to send updated user data to the backend.
- pendingChanges: A boolean value that tracks whether the user has made any unsaved changes to their data.
- validationErrors: An object that stores any errors that occur when validating user input (e.g., incorrect email format).
- validateUser(): A function to validate the currentUser data against the UserSettingsSchema.
- validatePassword(password, newPassword): A function to validate a password against the PasswordSchema.
- loading: A boolean value that indicates whether a background operation (like fetching data or saving changes) is currently in progress.
- handleUpdateDetails(): A function to trigger the update of user details.
- handleUpdatePreferences(): A function to trigger the update of user preferences.
- handleChangePassword(password, newPassword, setOpen): A function to handle changing the user's password.

```
export const useUser = create<Userstate>()(  
  immer((set, get) => ({  
    user: undefined,  
    pendingChanges: false,  
    validationErrors: {},  
    loading: false,  
  
    currentUser: {  
      username: '',  
      email: '',  
      password: '',  
      preferences: {  
        theme: 'system',  
      },  
    },  
  }),  
);
```

```
        model: 'default',
    }
},
```

The rest of the code is inside of the `useUser` Zustand store and is wrapped in the `immer` middleware, allowing the code to be updated directly instead of being copied every time a change is made. The `set` and `get` functions are provided by Zustand to update or retrieve the current state and are used throughout the code to achieve this.

Inside the function passed to `immer`, the code defines the initial values for the state variables and the functions that manage the user data:

- `user: undefined`
- `pendingChanges: false` (no unsaved changes)
- `validationErrors: {}`: ( no validation errors)
- `loading: false`: (no background operations in progress)
- `currentUser: An object with empty values for username, email, and default preferences`. This is used to store the user's data as they edit it in the form.

Now, let's see what each function does.

```
validateUser: () => {
  const { currentUser } = get();
  const result = UserSettingsSchema.safeParse(currentUser);
  if (!result.success) {
    set((state) => {
      const errors = Object.fromEntries(
        Object.entries(result.error.flatten().fieldErrors).map(([key, value]) => [key, value.join(", ", "")])
      );
      state.validationErrors = errors;
    });
    return false;
  } else {
    set((state) => {
      state.validationErrors = {};
    });
    return true;
  }
},
```

The `validateUser` function is responsible for ensuring that the user data being entered, or about to be saved, conforms to the expected structure and rules. It uses a ZOD schema to check if fields like `username` and `email` are in the correct format. If any errors are found, it updates the `validationErrors` object within the Zustand store, making these errors available to be displayed in the user interface, and returns `false`. If the data is valid, it clears any existing errors and returns `true`.

```
validatePassword: (password: string, newPassword: string) => {
```

```

const result = PasswordSchema.safeParse({ password, newPassword });
if (!result.success) {
  set((state) => {
    const errors = Object.fromEntries(
      Object.entries(result.error.flatten().fieldErrors).map(([key, value])
=> [key, value.join(", ")])
    );
    state.validationErrors = errors;
  });
  return false;
} else if (password === newPassword) {
  set((state) => {
    state.validationErrors = { newPassword: "New password must be different
from the old password" };
  });
  return false;
}
else {
  set((state) => {
    state.validationErrors = {};
  });
  return true;
}
},

```

This function checks if a provided password, and a new password, meet the criteria defined by the PasswordSchema. This could include checks for minimum length, required characters, and other security rules. It also specifically checks if the new password is the same as the old password, and raises an error if they match. Like validateUser, it updates the validationErrors object in the store if there are any issues and returns false, otherwise clearing errors and returning true. This function is used in the password update dialog that is mentioned later.

```

updateCurrentUser: <K extends keyof UserSettings>(key: K, value:
UserSettings[K]) => {
  set((state) => {
    if (state.currentUser) {
      state.currentUser[key] = value;
    }
    if (!state.pendingChanges) {
      state.pendingChanges = true;
    }
  });
  get().validateUser();
},

```

The updateCurrentUser function is used to modify the currentUser object in the store. This object holds the user's data as they are editing it in the form. It takes a key (the name of the field to update, like "username" or "email") and a value. It directly updates the corresponding field in the currentUser object. It also sets the pendingChanges flag to true, indicating that the user has made changes that need to be saved. Finally, it calls the validateUser function to ensure that the change doesn't introduce any validation errors.

```

updateUser: async (updateData: object) => {
    const { currentUser, validateUser } = get();
    if (validateUser()) {
        set((state) => {
            state.loading = true;
        });
        const response = await users.updateUserDetails(updateData);
        if (response) {
            set((state) => {
                state.loading = false;
                state.pendingChanges = false;
            });
        }
    }
},

```

When the user presses the save button, updateUser takes an updateData object, which contains the user data to be saved to the backend. It first calls the validateUser function to ensure that the data is valid. If the data is valid, it sets the loading state to true to indicate that an operation is in progress. It then uses the users.updateUserDetails function, which comes from the UsersAPI module to send the updated data to the backend. After the update is complete, it sets loading back to false and pendingChanges to false, allowing for the details changing action to be repeated again.

```

handleUpdateDetails: async () => {
    const {updateUser, currentUser} = get();
    await updateUser({
        username: currentUser.username,
        email: currentUser.email,
    });
},
handleUpdatePreferences: async () => {
    const {updateUser, currentUser} = get();
    await updateUser({
        preferences: currentUser.preferences,
    });
},

```

The handleUpdateDetails function is a helper function that specifically calls the updateUser function to save the user's main details, which are the username and email. It extracts these values from the currentUser object in the store and passes them to the updateUser function. handleUpdatePreferences is similar to handleUpdateDetails, but it's specifically for saving the user's preferences (like their theme or other settings). It extracts the preferences from the currentUser object and passes them to the updateUser function.

```

handleChangePassword: async (password:string, newPassword: string, setOpen: (boolean)=> void) => {
    const { validatePassword } = get();
    if (validatePassword(password, newPassword)) {
        const response = await users.changePassword(password, newPassword);
        if (response && response.success) {

```

```

        setOpen(false)
        //close dialog
    }
}
},

```

This handles the process of changing a user's password. It takes the old password and the new password as arguments, and a setOpen function, which is used to close the dialog box after successfully changing the password. It first calls validatePassword to check if both passwords are valid. If they are, it calls the users.changePassword function to send the password change request to the backend. If the password is changed successfully, it calls the setOpen function, which closes the dialog.

```

getUser: async () => {
  const response = await users.getUser();
  if (response) {
    set((state) => {
      state.user = response;
      state.currentUser = {
        username: response.username,
        email: response.email,
        preferences: response.preferences,
      };
    });
  }
},

```

The getUser function retrieves user details from the backend and stores them in the currentUser object.

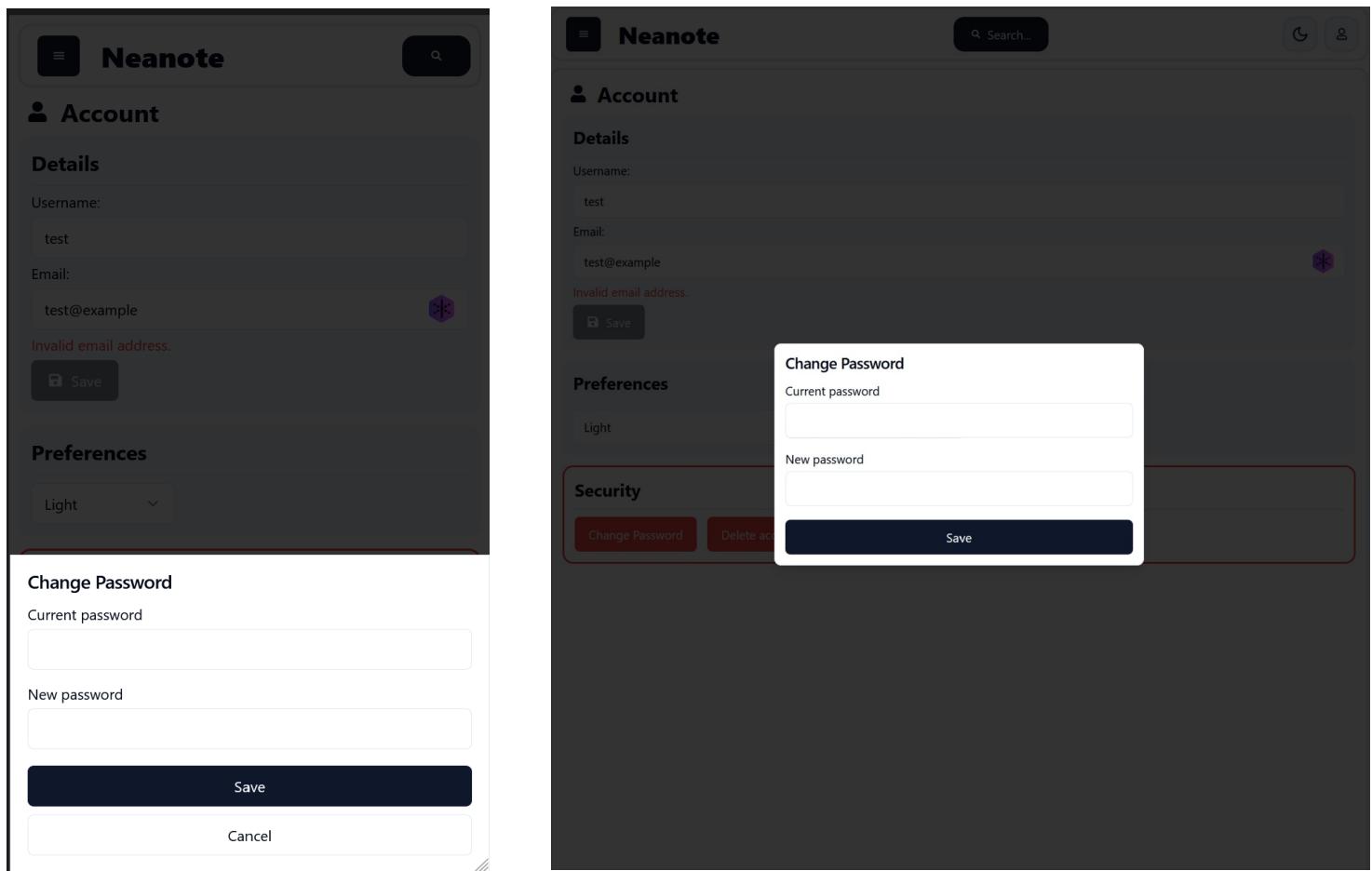
```

deleteUser: async (password:string) => {
  const response = await users.deleteUser(password);
  if (response && response.success) {
    return true
  }
  else {
    return false
  }
}
);

```

Finally, deleteUser function handles the deletion of a user's account. It takes the user's password as an argument, and calls the users.deleteUser function to send the deletion request to the backend. It returns true if the deletion is successful, and false otherwise.

### 3.2.5.4.3 ./src/Pages/Account/Components/PasswordDrawerDialog.tsx



As one of the compatibility functions of my notetaking app, each dialog must be able to adapt to different screen sizes and layouts. For example, a dialog is converted to a drawer on small screens, but its functionality remains the same.

```
import * as React from "react"

import { cn } from "../../../../../components/@/lib/utils"
import { Button } from "../../../../../components/@/ui/button"
import {
  Dialog,
 DialogContent,
  DialogDescription,
  DialogHeader,
  DialogTitle,
  DialogTrigger,
} from "../../../../../components/@/ui/dialog"
import {
  Drawer,
}
```

```

DrawerClose,
DrawerContent,
DrawerDescription,
DrawerFooter,
DrawerHeader,
DrawerTitle,
DrawerTrigger,
} from "../../../../../components/@/ui/drawer"
import { Input } from "../../../../../components/@/ui/input"
import { Label } from "../../../../../components/@/ui/label"
import { useScreenSize } from "../../../../../DisplayContext"
import { useUser } from "../useUser"

```

The codet uses React's useState to manage the visibility of a dialog or drawer, and it utilizes a custom hook, useScreenSize, to determine the current screen size. It also uses components from the shadcn library, such as Button, Dialog, DialogTrigger,DialogContent, DialogHeader,DialogTitle, Drawer, DrawerTrigger, DrawerContent, DrawerHeader, DrawerTitle, DrawerFooter, DrawerClose, Input, and Label.

```

export function PasswordDrawerDialog() {
  const [open, setOpen] = React.useState(false)
  const {screenSize} = useScreenSize()

  if (screenSize !=='small') {
    return (
      <Dialog open={open} onOpenChange={setOpen}>
        <DialogTrigger asChild>
          <Button variant="destructive" className="w-fit">Change Password</Button>
        </DialogTrigger>
        <DialogContent className="sm:max-w-[425px]">
          <DialogHeader>
            <DialogTitle>Change Password</DialogTitle>
          </DialogHeader>
          <ProfileForm setOpen={setOpen}/>
        </DialogContent>
      </Dialog>
    )
  }

  return (
    <Drawer open={open} onOpenChange={setOpen}>
      <DrawerTrigger asChild>
        <Button variant="outline">Change Password</Button>
      </DrawerTrigger>
      <DrawerContent>
        <DrawerHeader className="text-left">
          <DrawerTitle>Change Password</DrawerTitle>
        </DrawerHeader>
        <ProfileForm setOpen={setOpen} className="px-4" />
        <DrawerFooter className="pt-2">
          <DrawerClose asChild>
            <Button variant="outline">Cancel</Button>
          </DrawerClose>

```

```

        </DrawerFooter>
    </DrawerContent>
</Drawer>
)
}

```

The PasswordDrawerDialog component first uses the `useScreenSize` hook to get the current screen size. If the screen size is not small, it renders a Dialog component. The Dialog component displays a modal window, triggered by a Change Password button on the Account page. The DialogContent contains a DialogHeader with a title and the ProfileForm component, which handles the actual password change form. If the screen size is small, the component renders a Drawer instead. A Drawer is a panel that slides in from the bottom of the screen, as shown on the first screenshot, also triggered by a Change Password button. The DrawerContent contains a DrawerHeader with a title, the ProfileForm, and a DrawerFooter with a "Cancel" button that closes the drawer. Essentially, the component provides a responsive way to change the password, using a dialog on larger screens and a drawer on smaller screens for better accessibility

```

function ProfileForm({ className, setOpen }: React.ComponentProps<"form"> &
{ setOpen: React.Dispatch<React.SetStateAction<boolean>> }) {
    const {handleChangePassword, validationErrors} = useUser()
    const [form, setForm] = React.useState({
        password: '',
        newPassword: '',
    })

    const handleSubmit = (event: React.FormEvent) => {
        event.preventDefault()
        handleChangePassword(form.password, form.newPassword, setOpen)
    }

    return (
        <form className={cn("grid items-start gap-4", className)}>
            <div onSubmit={handleSubmit}>
                <div>
                    <Label htmlFor="password">Current password</Label>
                    <Input type="password"
                        value={form.password} onChange={e=>setForm({...form, password: e.target.value})} id="password" />
                    {validationErrors.password && <p className="text-sm text-destructive">{validationErrors.password}</p>}
                </div>
                <div>
                    <Label htmlFor="new password">New password</Label>
                    <Input value={form.newPassword} type="password"
                        onChange={e=>setForm({...form, newPassword: e.target.value})} id="confirm password" />
                    {validationErrors.newpassword && <p className="text-sm text-destructive">{validationErrors.newpassword}</p>}
                </div>
            </div>
        </form>
    )
}

```

```

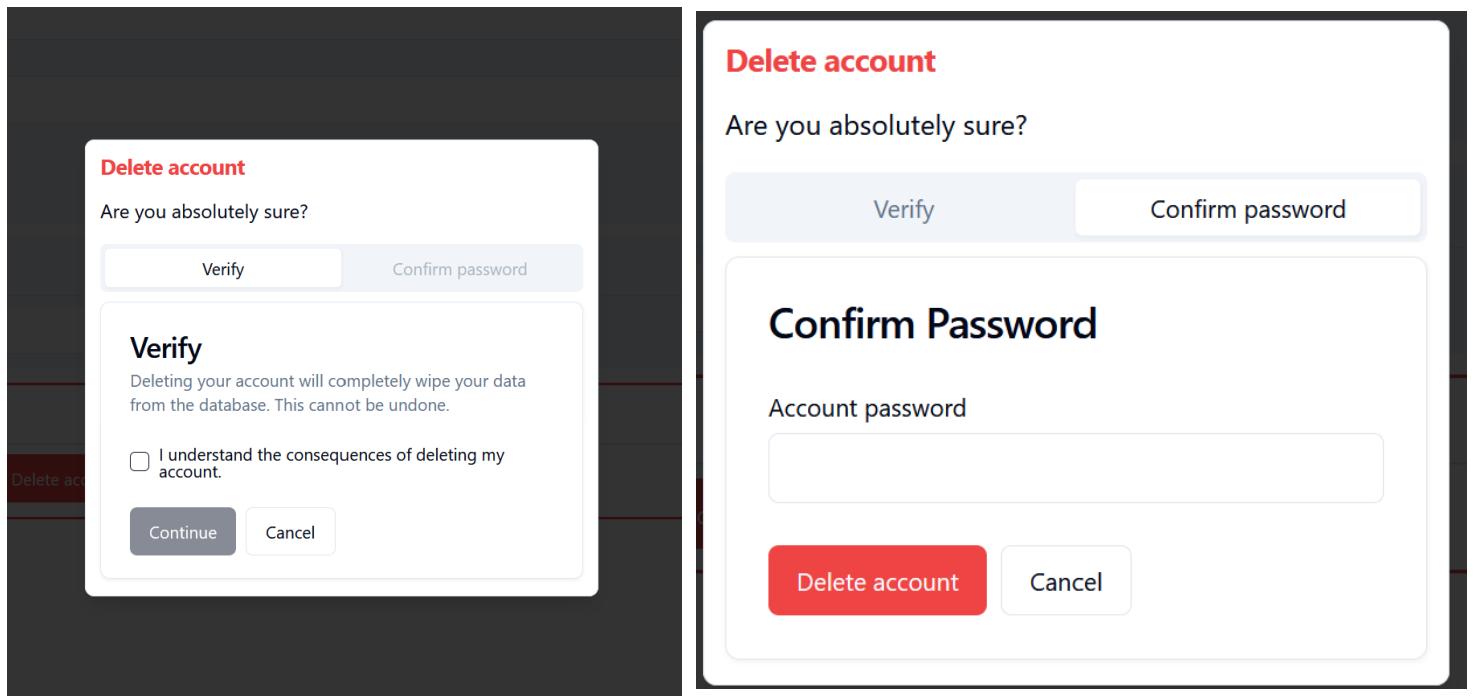
        <Button type="submit">Save</Button>
      </form>
    )
}

```

The ProfileForm component is a form that allows the user to enter their current password and their new password. It receives a setOpen function as a prop, which is used to close the dialog or drawer after a successful password change, and a className prop for additional styling using inline CSS provided by TailwindCSS. The component renders a form with two input fields for the current and new passwords, along with labels and error messages (from validationErrors) and a Save button to submit the form. The handleSubmit function is called when the form is submitted. It prevents the default HTML form submission behavior (which reloads the page) and calls the handleChangePassword function from the useUser hook, passing the password values and the setOpen function to close the modal after the password change. An appropriate toast is returned if the password is successfully reset.

### 3.2.5.4.4 ./src/Pages/Account/Components/DeleteAccountDrawerDialog.tsx

The account deletion popup uses a similar drawer-dialog mechanic. Only this time, there are several barriers the user must overcome in order to delete their account, ensuring that it is done in a mindful manner. Firstly, the user must verify that they understand the consequences of account deletion by clicking a checkmark and a “continue” button, which sends the user to the next page, where they must confirm their password again:



```
import React from "react"
```

183, Roman Sasinovich, 74561 IMS

```

import { useScreenSize } from "../../../../../DisplayContext"
import { Button } from "../../../../../components/@/ui/button"
import { Dialog, DialogContent, DialogHeader,DialogTitle, DialogTrigger } from
"../../../../../components/@/ui/dialog"
import { DrawerTrigger, DrawerContent, DrawerHeader, DrawerTitle, DrawerFooter,
DrawerClose, Drawer, DrawerDescription } from "../../../../../components/@/ui/drawer"
import { DialogDescription } from "@radix-ui/react-dialog"
import {
  Card,
 CardContent,
  CardDescription,
  CardFooter,
  CardHeader,
  CardTitle,
} from "../../../../../components/@/ui/card"
import { Input } from "../../../../../components/@/ui/input"
import { Label } from "../../../../../components/@/ui/label"
import {
  Tabs,
  TabsContent,
  TabsList,
  TabsTrigger,
} from "../../../../../components/@/ui/tabs"
import { Checkbox } from "../../../../../components/@/ui/checkbox"
import { useUser } from "../useUser"
import Cookies from 'js-cookie';
import { useNavigate } from "react-router-dom"
import { showToast } from "../../../../../components/Toast"

```

After extensive imports of components that are required for this component, the DeleteAccountDrawerDialog function applies the same logic to determine the screen size as described in the previous component. Additionally, named constants are used to define data in the text fields, that informs the user of the action they are about to undertake.

```

export function DeleteAccountDrawerDialog() {
  const [open, setOpen] = React.useState(false)
  const {screenSize} = useScreenSize()
  const title = 'Delete account'
  const desc = "Are you absolutely sure?"

  if (screenSize !=='small') {
    return (
      <Dialog open={open} onOpenChange={setOpen}>
        <DialogTrigger asChild>
          <Button variant="destructive" className="w-fit">{title}</Button>
        </DialogTrigger>
        <DialogContent className="sm:max-w-[425px]">
          <DialogHeader>
            <DialogTitle className="text-destructive
font-bold">{title}</DialogTitle>
          </DialogHeader>
          <DialogDescription>
            {desc}

```

```

        </DialogDescription>
        <DeletionTabs setOpen={setOpen}/>

        </DialogContent>
    </Dialog>
)
}

return (
    <Drawer open={open} onOpenChange={setOpen}>
        <DrawerTrigger asChild>
            <Button variant="outline">{title}</Button>
        </DrawerTrigger>
        <DrawerContent>
            <DrawerHeader className="text-left">
                <DrawerTitle className="text-destructive font-bold">{title}</DrawerTitle>
            </DrawerHeader>
            <DrawerDescription>
                {desc}
            </DrawerDescription>
            <DeletionTabs setOpen={setOpen}/>

            <DrawerFooter className="pt-2">
                <DrawerClose asChild>
                    <Button variant="outline">Cancel</Button>
                </DrawerClose>
            </DrawerFooter>
        </DrawerContent>
    </Drawer>
)
}

const DeletionTabs = ({setOpen}) => {
    const [verified, setVerified] = React.useState(false)
    const [value, setValue] = React.useState<string>('verify')
    const [password, setPassword] = React.useState<string>('')
    const {deleteUser} = useUser()
    const navigate = useNavigate()

    const handleAccountDeletion = async () => {
        if( password.length >= 6) {
            const deleted = await deleteUser(password)
            if (deleted) {
                setOpen(false)
                Cookies.remove('token')
                navigate('/get-started')
            }
        }
    }
}

return (
    <Tabs defaultValue="verify" value={value} className="w-[400px] h-[280px]">
        <TabsList className="grid w-full grid-cols-2">
            <TabsTrigger value="verify">Verify</TabsTrigger>

```

```

        <TabsTrigger disabled={!verified} value="confirm">Confirm
password</TabsTrigger>
    </TabsList>
    <TabsContent value="verify">
        <Card className="h-[230px]">
            <CardHeader>
                <CardTitle>Verify</CardTitle>
                <CardDescription>
                    Deleting your account will completely wipe your data from the
database. This cannot be undone.
                </CardDescription>
            </CardHeader>
            <CardContent className="space-y-2">
                <div className="flex items-center space-x-2">
                    <Checkbox onClick={()=>setVerified(!verified)} id="accept" />
                    <Label
                        htmlFor="accept"
                        className="text-sm font-medium leading-none peer-
disabled:cursor-not-allowed peer-disabled:opacity-70"
                    >
                        I understand the consequences of deleting my account.
                    </Label>
                </div>
            </CardContent>
            <CardFooter className="flex flex-row gap-2">
                <Button disabled={!verified} onClick={()=>
setValue('confirm')}>Continue</Button>
                <Button variant="outline"
onClick={()=>setOpen(false)}>Cancel</Button>
            </CardFooter>
        </Card>
    </TabsContent>
    <TabsContent value="confirm">
        <Card className="h-[230px]">
            <CardHeader>
                <CardTitle>Confirm Password</CardTitle>
            </CardHeader>
            <CardContent className="space-y-2">
                <div className="space-y-1">
                    <Label htmlFor="password">Account password</Label>
                    <Input value={password} onChange={e =>
setPassword(e.target.value)} id="password" type="password" />
                </div>
            </CardContent>
            <CardFooter className="flex flex-row gap-2">
                <Button variant={'destructive'}
onClick={handleAccountDeletion}>Delete account</Button>
                <Button variant="outline"
onClick={()=>setOpen(false)}>Cancel</Button>
            </CardFooter>
        </Card>
    </TabsContent>
</Tabs>
)
}

```

The DeletionTabs component manages the two-step deletion process. It uses React's useState to manage the currently displayed tab (value), a verification flag (verified), and the user's password (password). It also uses the useUser hook to access the deleteUser function and the useNavigate hook from React Router to redirect the user after successful deletion. The component employs the Tabs, TabsList, and TabsContent components to structure the process. The handleAccountDeletion function is called when the user confirms the deletion in the second tab. After a basic password length validation, it calls the deleteUser function from the useUser hook, and if the deletion is successful, it closes the dialog/drawer, removes the user's token from cookies, and redirects the user to the /get-started page. The rest of the code in the return statement consists of UI components. In some cases, their theme is edited using inline CSS. The components are connected to the logic through onClick properties, which take a function that will be activated if the component is interacted with. Alternatively, some components are disabled based on the state of the app. For example, the "Continue" button is off unless the checkbox is clicked. Moreover, the checkbox is connected to the label with htmlFor="accept". This attribute is crucial for accessibility. It associates the label with the checkbox element that has the ID accept. This association means that when a user clicks on the label text, the browser will also toggle the checkbox, just as if the user had clicked directly on the checkbox.

### **3.2.5.5 Tags**

One of the main requirements for this project is a system to group notes regardless for their type by some user-defined property. This is what tags are for.

The screenshot shows the Neanote application interface. At the top, there is a header with the logo and the word "Neanote". To the right of the header is a search bar with the placeholder "Search...". Further to the right are three small icons: a refresh symbol, a user profile, and a plus sign for adding new items. Below the header, the main content area is titled "Tags". It contains four entries, each consisting of a colored circular badge followed by the tag name and an edit icon. The entries are: "Sample Tag 1" (red), "Sample Tag 2" (blue), "Sample Tag 3" (green), and "Sample Tag 4" (black).

Tags have just two properties: Their name and color, which are displayed as badges on top of each note. The color allows the user to distinguish between tags without reading them.

### 3.2.5.5.1      [./src/Pages/Tags/Tags.tsx](#)

```
import React, { useEffect, useState } from 'react'  
import PageContainer from '../../../../../components/PageContainer/PageContainer'  
import { useTags } from './useTags'  
import { Button } from '../../../../../components/@ui/button';  
import { Separator } from '../../../../../components/@ui/separator';
```

```

import { FaPlus } from 'react-icons/fa6';
import TagCard from '../../../../../components/TagCard/TagCard';
import { HexColorPicker } from "react-colorful";
import { MdCancel } from 'react-icons/md';
import { Input } from '../../../../../components/@/ui/input';
import { Popover } from '@radix-ui/react-popover';
import { PopoverContent, PopoverTrigger } from '../../../../../components/@/ui/popover';
import { FaTags } from "react-icons/fa";
import TitleComponent from '../../../../../components/TitleComponent/TitleComponent';

function Tags() {
  const {
    tags,
    fetchTags,
    section,
    setSection,
    tagTitle,
    setTagTitle,
    color,
    setColor,
    handleSaveTag,
    handleEditTag,
    handleDeleteTag
  } = useTags()
}

useEffect(() => {
  fetchTags();
}, [fetchTags]);

let allTags = (
  <>
    <div className='flex flex-row justify-between'>
      <TitleComponent><FaTags size={'20px'} /> Tags</TitleComponent>
      <Button size="icon" onClick={() => setSection("create")}>
        <FaPlus />
      </Button>
    </div>

    {tags.map((tag, index) => (
      <div key={index} className="py-2">
        <TagCard tag={tag} />
      </div>
    )));
  </>
);

```

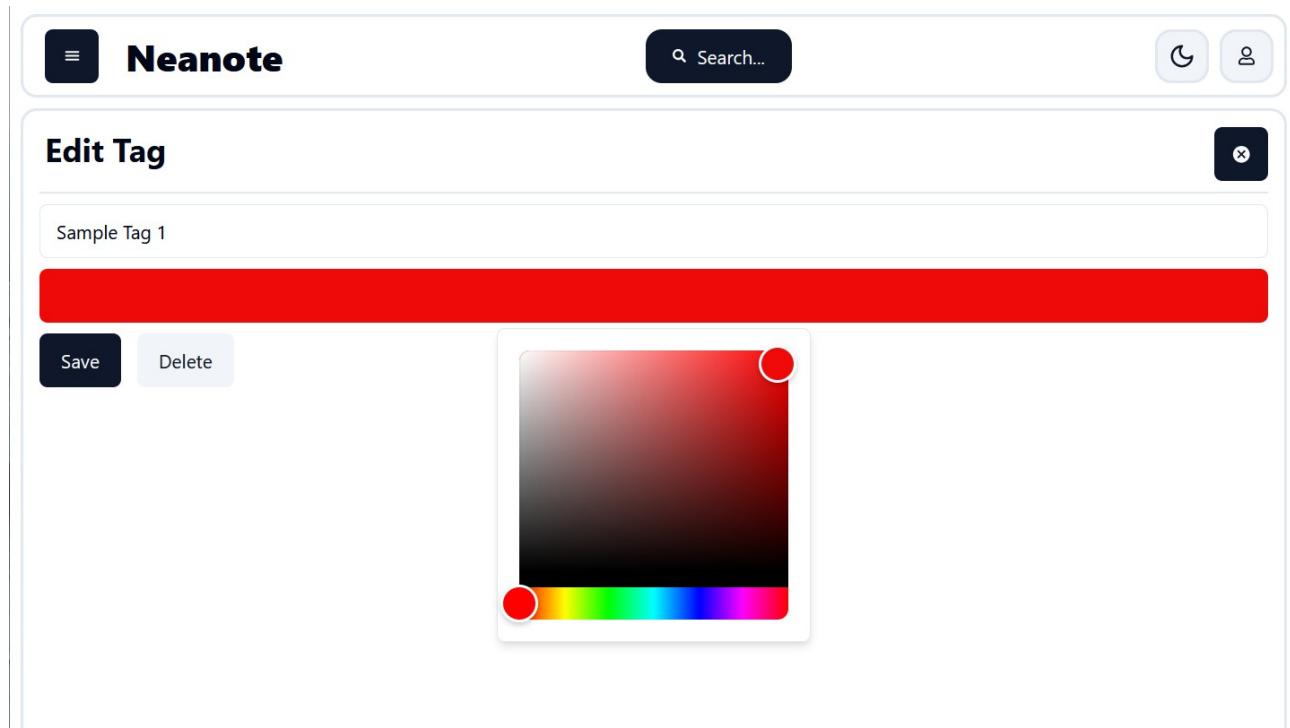
This code implements a tag management page using React, allowing users to view, create, and edit tags. The component relies on React's useState for local UI state (like which section is displayed) and a custom hook, useTags, for managing tag data and operations. The useTags hook encapsulates all the tag-related logic, allowing for component reusability and separation of concerns. This hook provides the all the necessary functions to update, retrieve, create and delete tags through appropriate

functions, that will be discussed in more detail in the next section. The component uses conditional rendering to display different UI elements based on the section state. If the section is "all tags", it displays a list of tags. If it's "create" or "edit", it shows a form for creating or editing a tag. This avoids navigating to a different page, which may be time consuming and bad for user experience. Furthermore, useEffect is used to fetch tags when the component mounts. allTags section uses the map function to iterate over the tags array and render a TagCard component for each tag. The key prop provided into each tag is essential for React to efficiently update the list. This is because React may not have to rerender the whole list, if it knows which item in it was changed.



Each tag card has a color, tag name and an icon button that will open the tag editing page. Additionally, a plus icon is on the top right corner of the page if the user wants to edit the tag. This design will stay consistent throughout the app with minor changes.

Clicking the edit or create button renders the createTagForm:



```
const createTagForm = (
  <>
    {/* Navbar */}
    <div className='flex flex-row justify-between'>
```

```

    <p className='pl-1 text-2xl font-bold'>{section === 'create' ? "Create Tag" : "Edit Tag"}</p>
    {/* Date Picker */}
    <div className='flex flex-row gap-2'>
        <Button size="icon" onClick={() => setSection("all tags")}>
            <MdCancel size={15} />
        </Button>
    </div>

    </div>
    <div className='pt-2'>
        <Separator />
    </div>
    <div className='py-2 flex flex-col gap-2'>

        <Input
            className='border rounded-md w-full h-10 leading-tight focus:outline-none
focus:shadow-outline'
            placeholder='Title'
            type='text'
            value={tagTitle}
            onChange={(e) => setTagTitle(e.target.value)}
        />
        <Popover>
            <PopoverTrigger asChild>
                <Button style={{backgroundColor:color }}/>
            </PopoverTrigger>
            <PopoverContent className='w-200'>
                <HexColorPicker color={color} onChange={setColor} />
            </PopoverContent>
        </Popover>
    </div>
    <div className='flex flex-row gap-3'>

        <Button disabled={tagTitle.length == 0} disabled={tagTitle.length == 0}
onClick=
{
    section === "create" ? handleSaveTag: handleEditTag
}>
            Save
        </Button>
        {section === "edit" && <Button variant={'secondary'}>
        onClick={handleDeleteTag}>Delete</Button>}
    </div>

    </>
);

```

The form contains an input field to enter the tag title, as well as the HexColorPicker component from the "react-colorful" library that allows users to select a color. The color state variable holds the selected color, and the onChange handler updates this state using

the setColor function. The selected color is also used to style a button, which acts as a trigger for the color picker. Another button is used to save any changes the user made. The onClick attributes on the button helps with event handling by calling functions like handleSaveTag to communicate changes to the backend.

```
return (
  <>
  {section === 'all tags' && allTags}
  {(section === 'create' || section === 'edit') && createTagForm}
</>
);
}

export default Tags
```

Finally, the forms are conditionally rendered based on the section variable, which changes whenever the user clicks the create or edit buttons. This variable is also used to dynamically adjust the title of the form from Create to Edit, depending on the action that the user is undertaking.

### 3.2.5.5.2 ./src/Pages/Tags/useTags.tsx

The useTags hook, again using the Zustand library, provides a centralized mechanism for managing tag-related data and operations within the application. The hook contains all tag-related logic and can be called by other components.

```
type TagState = {
  section: string;
  setSection: (section: string) => void;
  tags: Tag[];
  setTags: (tags: Tag[]) => void;
  order: string
  setOrder: (order: string)=>void;
  fetchTags: () => Promise<void>;
  tagTitle: string;
  setTagTitle: (title: string) => void;
  color: string;
  setColor: (color: string) => void;
  currentTagId: UUID | undefined;
  setCurrentTagId: (tagId: UUID) => void;
  selectedTagIds: UUID[];
  setSelectedTagIds: (tagIds: UUID[]) => void;
  // updateState: (key: keyof TagState, value: any) => void;
  handleSaveTag: () => void;
  handleDeleteTag: () => void;
  handleEditTag: () => void;
};
```

It begins with a type that defines all the functions and constants in the store and their datatypes.

```

tags: [],
tagTitle: '',
currentTagId: undefined,
selectedTagIds: [],
order: 'ascending',
color: '#000000',
section: 'all tags',
setSection: (section) => set({ section }),
setTagTitle: (title) => set({ tagTitle: title }),
setTags:(tags)=>set({tags}),
setColor: (color) => set({ color }),
setOrder: (order) => set({ order }),
setSelectedTagIds: (tagIds) => set({ selectedTagIds: [...tagIds] }),

setCurrentTagId: (tagId) => set({ currentTagId: tagId }),

```

Then, some basic constants and functions are defined. Functions above use Zustand's set method to directly update their corresponding state properties (section, tagTitle, tags, color, and order). This allows to quickly modify properties of tags in the state.

```

fetchTags: async () => {
  const fetchedTags = await tagsApi.getAll();
  if (fetchedTags) {
    set({ tags: fetchedTags.data });
  }
},

```

This asynchronous function, marked with the `async` keyword, performs an API call to retrieve tag data from the backend. It utilizes the `tagsApi`, and upon successful retrieval of the data, the function updates the `tags` array within the store.

```

handleSaveTag: async () => {
  const { tagTitle, color, tags } = get();
  const response = await tagsApi.create(tagTitle, color);
  if (response && response.success) {
    set({ tagTitle: '', color: '#000000', section: 'all tags' });
    set((state) => {
      state.tags.push({
        tagid: response.tagid,
        name: tagTitle,
        color: color
      });
      return state;
    })
  }
},

```

`handleSaveTag` is another asynchronous function which creates a new tag. It retrieves the `tagTitle` and `color` from the store, and then makes an API call to create the tag on the server using `tagsApi.create()`. After a successful response, default values for tag properties are set and the new tag is added to the store.

```

handleDeleteTag: async () => {
  const { currentTagId, tags } = get();

```

```

        if (currentTagId === undefined) return;
        const response = await tagsApi.delete(currentTagId);
        if (response && response.success) {
            set({tags:tags.filter(tag => tag.tagid != currentTagId ) , section:
'all tags' });
        }
    },

```

This function performs a similar task but for deleting a tag. It also applies a filter function which removes tags that match the ID of the recently deleted tag.

```

handleEditTag: async () => {
    const { currentTagId, tagTitle, color, tags } = get();
    if (currentTagId === undefined) return;
    const response = await tagsApi.edit(currentTagId, tagTitle, color);
    if (response.success) {
        const sortedTags = [...tags].sort((a, b) =>
            a.tagid.toString().localeCompare(b.tagid.toString())
        );

        // Define comparator for binary search
        const comparator = (tag: Tag, id: UUID) =>
            tag.tagid.toString().localeCompare(id.toString());

        const tagIndex = binarySearch(sortedTags, currentTagId, comparator);

        // Create a new tags array with the updated tag
        let updatedTags = [...tags];
        if (tagIndex !== -1) {
            const originalTag = sortedTags[tagIndex];
            const originalIndex = tags.findIndex(t => t.tagid ===
originalTag.tagid);

            if (originalIndex !== -1) {
                updatedTags[originalIndex] = { ...originalTag, name: tagTitle,
color };
            }
        }

        set({
            tags: updatedTags,
            tagTitle: '',
            color: '#000000',
            section: 'all tags'
        });
    }
},

```

The last function in the file is handleEditTag. This asynchronous function handles the updating of an existing tag. It calls tagsApi's edit method to send the updated tag data to the server. If the response is a success, the function locates the tag to be updated within the tags array. To optimize this search, it first sorts the tags array and then uses a binarySearch function, along with a custom comparator, to efficiently find the tag's index. A

new updatedTags array is then constructed, and the located tag is updated within this new array.

### 3.2.5.5.2.1 Binary search

The binary search function is implemented in a reusable manner in `utils.ts`, which contains a few other functions that will be discussed in context of their use cases. For now, lets take a look at the binary search:

```
// Binary search function
export function binarySearch<T>(
  arr: T[],
  target: T,
  comparator: (a: T, b: T) => number
): number {
  let low = 0;
  let high = arr.length - 1;

  while (low <= high) {
    const mid = Math.floor((low + high) / 2);
    const compareResult = comparator(arr[mid], target);

    if (compareResult === 0) {
      return mid; // Found the element
    } else if (compareResult < 0) {
      low = mid + 1; // Search in the right half
    } else {
      high = mid - 1; // Search in the left half
    }
  }

  return -1; // Element not found
}
```

This function can be used to find an element in an abstract array of a TypeScript type `T`. It takes the array of items of type `T`, a target of type `T` and a comparator

The comparator function is a crucial aspect of this binary search's flexibility. It determines the relationship between the element at the middle index (`arr[mid]`) and the target element. The comparator function should return:

- 0 if `arr[mid]` is equal to target.
- A negative value if `arr[mid]` is less than target.
- A positive value if `arr[mid]` is greater than target.

The function first defines variables `low` and `high`, which are used to calculate the remaining array after each fold. Then, a loop iteratively narrows down the search space. The loop continues as long as there's a valid interval to search within. Inside the loop, `const mid = Math.floor((low + high) / 2)` calculates the middle index of the current search interval.

Math.floor ensures that mid is always an integer, even if (low + high) / 2 results in a floating-point number. Based on the result of the comparator, the interval is adjusted as follows:

- If compareResult === 0, the target element is found, and its index (mid) is returned.
- If compareResult < 0, the target element must be in the right half of the interval (since arr[mid] is less than target). The low index is updated to mid + 1 to exclude the left half.
- If compareResult > 0, the target element must be in the left half of the interval (since arr[mid] is greater than target). The high index is updated to mid - 1 to exclude the right half.

If the element is not found, the function returns -1, highlighting that the element is not in the array.

### 3.2.5.5.3        [./src/Pages/Tags/TagsMenu.tsx](#)

Another very important task to accomplish was the connection between any note and tags. On the database, it was done with the NoteTags table, but here it needs to be shown to the user as a UI element:



This menu contains all existing tags and allows the user to sort them in an ascending or descending order

```
import * as React from "react";
import {
  Dialog,
  DialogContent,
  DialogHeader,
  DialogTrigger,
} from "../../../../../components/@/ui/dialog";
import { Button } from "../../../../../components/@/ui/button";
import { useTags } from "../useTags";
import TagLabel from "./TagLabel";
import { Tag } from "../../../../../api/types/tagTypes";
import { quicksort } from "../../../../../components/utils";
import SortMenu from "../../../../../components/SortMenu/SortMenu";
```

```
interface Props {
  onChange: () => void
}
```

Apart from importing components from the UI library and custom components, TagsMenu also takes in an arbitrary onChange function, which allows me to alter the logic of a user tag selection based on the section this specific component is a part of.

```
function TagsDropdownMenu({onChange}:Props) {
  const { tags, setTags, fetchTags, order, setOrder } = useTags();

  React.useEffect(() => {
    if (tags.length < 1) {
      fetchTags();
    }
  }, [tags.length, fetchTags]);

  React.useEffect(() => {
    sort_tags(tags, order)
  }, [order])
}
```

After passing the onChange function inside the component and taking some necessary functions and variables from the useTags store, useEffect hooks are used to fetch and sort tags upon them being mounted or the user opting for a different order of tags. The latter useEffect calls the sort\_tags function, which takes in the tags array and the order and defines a comparator for a quicksort, based on the order. Then, the quicksort function is called and the sorted tags replace the previous array in the store.

```
function sort_tags(tags: Tag[], order: string) {

  const ascendingComparator = (a: Tag, b: Tag) => a.name.localeCompare(b.name);
  const descendingComparator = (a: Tag, b: Tag) => b.name.localeCompare(a.name);

  const comparator = order === 'ascending' ? ascendingComparator :
  descendingComparator;

  const sortedTags = quicksort(tags, comparator);
  setTags(sortedTags); // Update the tags array using setTags
}

return (
  <Dialog >
    <DialogTrigger asChild>
      <Button variant="secondary">Tags</Button>
    </DialogTrigger>
    <DialogContent className="mx-auto w-[50vw]" >
      <DialogHeader className="px-2 pt-2 text-xl font-bold">
        Tags
      </DialogHeader>
      <div className="flex flex-wrap gap-2">
        {tags.map((tag, index) => (

```

```

        <TagLabel
          key={index}
          tagId={tag.tagid}
          title={tag.name}
          color={tag.color}
          onChange={onChange}
        />
      ))
    </div>
    <SortMenu order={order} setOrder={setOrder}/>
  </DialogContent>
</Dialog>
);
}
export default TagsDropdownMenu;

```

The return statement takes the sorted tags and maps it onto Tag Labels, which are interface components that display tag details to the user:

```

import { IoIosCheckbox, } from "react-icons/io";
import { MdOutlineCheckBoxOutlineBlank } from "react-icons/md";
import { Button } from '../../../../../components/@/ui/button';
import { useTags } from '../useTags';
import { UUID } from 'crypto';

interface Props {
  tagId: UUID;
  color: string;
  title: string;
  onChange: () => void;
}

function TagLabel({ color, title, tagId, onChange }: Props) {
  const { selectedTagIds, setSelectedTagIds } = useTags();
  const checked = selectedTagIds.includes(tagId);

  const handleClick = () => {
    const newSelectedTagIds = checked
      ? selectedTagIds.filter((id) => id !== tagId)
      : [...selectedTagIds, tagId];
    setSelectedTagIds(newSelectedTagIds);
    onChange()
  }

  return (
    <Button
      variant="ghost"
      className="justify-left flex gap-2 border-2 rounded-md p-2 flex-row items-center"
      onClick={handleClick}
    >
      {checked ? <IoIosCheckbox /> : <MdOutlineCheckBoxOutlineBlank />}
    
```

```

        <div style={{ backgroundColor: color, width: "15px", height: "15px",
borderRadius: "50%" }}/>
        <div className='max-w-[100px] overflow-hidden overflow-ellipsis'>
            {title}
        </div>
    </Button>
);
}

export default TagLabel;

```

These labels include a checkbox , which, upon being clicked, updates the selectedTagIds array with the ID of the selected tag and calls an arbitrary onChange function, that was passed into the component as props.

### 3.2.5.5.3.1 Quicksort

```

// QuickSort function
export function quicksort<T>(arr: T[], comparator: (a: T, b: T) => number): T[] {
    if (arr.length <= 1) {
        return arr;
    }

    const pivot = arr[Math.floor(arr.length / 2)];
    const left = arr.filter(item => comparator(item, pivot) < 0);
    const right = arr.filter(item => comparator(item, pivot) > 0);
    const middle = arr.filter(item => comparator(item, pivot) === 0);

    return [...quicksort(left, comparator), ...middle, ...quicksort(right,
comparator)]; // recursion
}

```

QuickSort is also a part of utils.py and is used to quickly sort large arrays of data. It is a textbook example of the divide and conquer paradigm. It breaks down the sorting problem into smaller subproblems and solves them recursively until all the solutions combine into a sorted array. QuickSort is similar to the binary search, as it also takes an array of some type T (that can be changed by replacing `<T>` at the start of the function with a different type; For example: `quicksort<Note>` will replace type T with type Note dynamically), and a comparator. The `if (arr.length <= 1)` condition defines the base case for the recursion. If the array has zero or one element, it's already sorted, so the function returns it directly. This prevents infinite recursion. The line `const pivot = arr[Math.floor(arr.length / 2)]` selects the pivot element. In this implementation, the pivot is chosen as the element in the middle of the array. And a floor math operation is used to return an integer value. The core of Quicksort is the partitioning step. The code uses the filter TypeScript method to partition the array into three subarrays:

- left: Contains elements that are less than the pivot
- right: Contains elements that are greater than the pivot
- middle: Contains elements that are equal to the pivot.

The final return statement combines the sorted subarrays, and the spread syntax (...) is used to concatenate the three arrays into a single sorted array.

- ...quicksort(left, comparator): Recursively sorts the left subarray.
- ...middle: Includes the elements equal to the pivot (they are already in their correct sorted position).
- ...quicksort(right, comparator): Recursively sorts the right subarray.

### 3.2.5.6 Tasks

The screenshot shows the Neanote application interface with the following components:

- Header:** A dark header bar with the "Neanote" logo, a search bar containing "Search...", and user icons for light/dark mode and profile.
- Section Header:** A "Tasks" section header with a "Tasks" icon and a "Add Task" button.
- Task Item 1:** A task titled "Lorem ipsum" with a due date of "Mar 1, 2025, 12:00 AM". It contains two subtasks: "1" (checked) and "2" (unchecked).
- Task Item 2:** A task titled "Task sample" with a due date of "Mar 26, 2025, 12:00 AM". It contains three subtasks: "Subtask 1" (unchecked), "Subtask 2" (checked), and "Subtask 3" (unchecked).
- Task Item 3:** A task titled "Gardening Notes" with a due date of "Mar 26, 2025, 12:00 AM". It contains one subtask: "a bee that works" (checked).
- Pagination:** At the bottom left, there are navigation buttons for "1" (highlighted in black), "1", and "2".

The tasks component adheres to similar design principles. All tasks have a title, optional content, due date, may have tags and any amount of subtasks. Subtasks and tasks can be completed by the user in any order, and a completion of all subtasks automatically completes the task. Since tasks may take a lot of place in the page, pagination was also implemented.

### 3.2.5.6.1      ./src/Pages/Tasks/Tasks.tsx

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

201, Roman Sasinovich, 74561 IMS

```

import { FaTasks } from "react-icons/fa";
import { FaPlus } from 'react-icons/fa6';
import { useNavigate } from 'react-router-dom';
import { Button } from '../../../../../components/@ui/button';
import PaginationSelector from '../../../../../components/Pagination/PaginationSelector';
import TaskCard from '../../../../../components/TaskCard/TaskCard';
import TitleComponent from '../../../../../components/TitleComponent/TitleComponent';
import { useTasks } from './useTasks';

const Tasks: React.FC = () => {
  const { tasks, fetchTaskPreviews, resetCurrentTask, nextPage, page } =
    useTasks();
  const navigate = useNavigate();

  const [lastFetchTime, setLastFetchTime] = useState<Date | null>(null);

  useEffect(() => {
    const fetchIfNeeded = () => {
      // Check if never fetched or if 5 minutes have passed since the last fetch
      if (!lastFetchTime || new Date().getTime() - lastFetchTime.getTime() >
        300000) {
        fetchTaskPreviews(page);
        setLastFetchTime(new Date());
      }
    };
    fetchIfNeeded();

    // Set up a timer to refetch every 5 minutes
    const intervalId = setInterval(fetchIfNeeded, 300000);

    // Clean up the interval on component unmount
    return () => clearInterval(intervalId);
  }, [fetchTaskPreviews, lastFetchTime]);

  const handleAddTaskClick = () => {
    resetCurrentTask();
    navigate('/tasks/create')
  };

  return (
    <>
      <div className="flex flex-row justify-between pb-2">
        <TitleComponent>
          <FaTasks size={'20px'} /> Tasks
        </TitleComponent>
        <Button size="sm" className="gap-2" onClick={handleAddTaskClick}>
          <FaPlus />
          Add Task
        </Button>
      </div>
      <div className="flex flex-col gap-3 flex-grow">
        {tasks.map((task) => (
          <TaskCard key={task.taskid} task={task} />
        ))}
      </div>
    </>
  );
}

```

```

        </div>
        <div className="p-1 pt-2">
          <PaginationSelector fetchingFunction={fetchTaskPreviews}>
            nextPage={nextPage} page={page}/>
          </div>
        </>
      );
    };
  };

export default Tasks;

```

The implementation of the Tasks TypeScript page is also similar to the implementation in Tags.tsx. The only difference here is the updated fetching function, which still uses the useEffect hook. Additionally, the component stores the time tasks were last fetched and automatically runs a refetch script every 5 minutes. If the user clicks the Add Task button or the edit icon on any of the task cards, they are navigated to the /create or /edit endpoint.

### **3.2.5.6.2       ./src/Pages/Tasks/CreateTasks.tsx**

```

import React, { useEffect, useState } from 'react';
import { FaPlus } from 'react-icons/fa';
import { MdCancel } from 'react-icons/md';
import { useNavigate } from 'react-router-dom';
import { Button } from '../../../../../components/@/ui/button';
import { Label } from '../../../../../components/@/ui/label';
import FormButtons from '../../../../../components/FormButtons/FormButtons';
import PageContainer from '../../../../../components/PageContainer/PageContainer';
import { DatePicker } from './DatePicker/DatePicker';
import EditTasksSkeleton from './EditTasksSkeleton';
import FormInputs from './FormComponents/FormInputs';
import Subtasks from './FormComponents/Subtasks';
import { useTasks } from './useTasks';

function EditTasks() {
  const {
    currentTask,
    pendingChanges,
    loading,
    fetchTask,
    updateCurrentTask,
    archive,
    handleAddSubtask,
    handleEditTask,
    handleDeleteTask,
    resetCurrentTask,
    validationErrors,
  } = useTasks();

  const navigate = useNavigate();
  const [isValidValidationErrorsEmpty, setIsValidationErrorsEmpty] = useState(true);

  useEffect(() => {
    const noteId = localStorage.getItem('currentTaskId');
    if (noteId) {

```

```

        fetchTask(noteId);
    }
}, [fetchTask, localStorage.getItem('currentTaskId')]);

```

After all necessary imports, a `useEffect` hook runs and checks the browser's local storage for a "currentTaskId", which should have been put there when the user clicked a specific task's edit button. If such ID exists, the hook runs a `fetchTask` function, which sends a GET request to the server and retrieves fully the selected task.

```

useEffect(() => {
    setIsValidationErrorsEmpty(
        Object.keys(validationErrors).every((key) => !validationErrors[key])
    );
    console.log(validationErrors)
}, [validationErrors]);

```

This hook checks if there are validation errors in the task (like empty title, etc.) and dynamically updates a boolean variable, which determines whether the task can be submitted or not. The user cannot press the "Save" button if there are any validation errors present

```

const handleClose = () => {
    localStorage.removeItem('currentTaskId');
    resetCurrentTask();
    navigate('/tasks');
};

const handleSave = async () => {
    await handleEditTask();
};

const handleDelete = async () => {
    await handleDeleteTask(currentTask?.taskid, currentTask?.noteid);
    localStorage.removeItem('currentTaskId');
    navigate('/tasks');
};

const handleArchive = async () => {
    await archive(currentTask?.noteid);
    navigate('/tasks');
}

```

The above handler functions call the appropriate asynchronous functions from `useTasks`, and navigate the user to the appropriate page using react-router's `useNavigate` hook. Additionally, some functions wipe the local storage from the ID of the currently open task.

```

if (loading) return <EditTasksSkeleton />;
if (!currentTask) return null;

```

If there is no current task, the component returns nothing. However, there may not be any currentTask if the task is still being retrieved from the server (for example, the user's internet connection is bad, so the loading variable is true and a skeleton is rendered instead. A skeleton is just a UI element that resembles the contents of the page, but has nothing rendered in it. It is purely decorative and shows the user that the page is loading. Such skeletons are used on most note-related pages.

```

return (
  <>
  <div className="flex flex-row justify-between">
    <p className="pl-1 text-2xl font-bold">Edit Task</p>
    <div className="flex flex-row gap-2">
      <DatePicker
        onDateChange={(date) => updateCurrentTask('due_date', date)}
        data={currentTask.due_date}
        includeTime={true}>
      />
      <Button size="icon" onClick={handleClose}>
        <MdCancel size={15} />
      </Button>
    </div>
  </div>

  <FormInputs title={currentTask.title} content={currentTask.content}
  validationErrors={validationErrors} withCheckBox/>

  <div className="rounded-md">
    <Subtasks task={currentTask}/>
    {validationErrors['subtasks'] && (
      <Label
      className="text-destructive">{validationErrors['subtasks']}</Label>
    )}
    <div className="flex py-3 justify-between">
      <Button onClick={handleAddSubtask}>
        <div className="flex flex-row items-center gap-2">
          <FaPlus />
          Add Subtask
        </div>
      </Button>
      <FormButtons
        pendingChanges={pendingChanges}
        isValidationErrorsEmpty={isValidationErrorsEmpty}
        loading={loading}
        hasDelete
        handleSave={handleSave}
        handleArchive={handleArchive}
        handleDelete={handleDelete} //add this to all other forms
      />
    </div>
  </div>
</>
);
}

```

```
export default EditTasks;
```

Then, the actual page is rendered. It primarily renders validation errors and note type-specific components like subtasks. All other buttons, the inputs for task title, content and tags are encapsulated in FormInputs and FormButtons components, which are reused throughout the programme.

### 3.2.5.6.3        **./src/Pages/Tasks/Formcomponents/FormInputs.tsx**

```
import React from "react"
import { Input } from "../../../../../components/@/ui/input"
import { Textarea } from "../../../../../components/@/ui/textarea"
import { Task } from "../../../../../api/types/taskTypes"
import TagsDropdownMenu from "../../Tags/components/TagsDropdownMenu"
import { useTasks } from "../useTasks"
import CheckBox from "../../../../../components/CheckBox/CheckBox"
import { Label } from "../../../../../components/@/ui/label"
import AutoResizeTextBox from
"../../../../../components/AutoResizeTextBox/AutoResizeTextBox"

interface Props {
    content: string
    title: string
    validationErrors: { [key: string]: string | undefined}
    withCheckBox?: boolean
}

function FormInputs({content, title, withCheckBox, validationErrors}: Props) {
    const {updateCurrentTask, setPendingChanges, toggleTaskCompleted, currentTask} = useTasks()

    return (
        <div className="pt-2 h-full">
            <div className="flex flex-row gap-2 ">
                {withCheckBox && <CheckBox checked={currentTask.completed} onChange={()=>toggleTaskCompleted(currentTask.taskid)} />}
                <Input
                    id="title"
                    name="Title*"
                    required
                    type="text"
                    value={title}
                    placeholder='Title'
                    onChange={(e) => updateCurrentTask('title', e.target.value)}
                    className="w-full p-2 border rounded"
                />
                <TagsDropdownMenu onChange={()=>setPendingChanges(true)}/>
            </div>
            {validationErrors['title'] && (
                <Label htmlFor="title" className='text-destructive'>{validationErrors['title']}
            )}
        </div>
    )
}
```

```

        <AutoResizeTextBox<Task> content={content} update={updateCurrentTask}
placeholder='Describe your task here' />

        {validationErrors['content'] && (
            <Label htmlFor="content" className='text-destructive'>{validationErrors['content']}

```

### **3.2.5.6.4       ./src/Pages/Tasks/Fromcomponents/FormInputs.tsx**

```

import React from 'react'
import DeleteDialog from '../DeleteDialog/DeleteDialog';
import { Button } from '../@/ui/button';
import { FaSave, FaTrash, FaArchive } from 'react-icons/fa';

interface Props {
    handleArchive?: () => void;
    handleDelete?: () => void
    handleSave: () => void
    pendingChanges: boolean
    loading: boolean
    isValidationErrorsEmpty: boolean
    hasDelete?: boolean
}

function FormButtons({handleArchive, loading, handleDelete, handleSave,
pendingChanges, isValidationErrorsEmpty, hasDelete}: Props) {
    return (
        <div className="flex flex-row gap-2">
            {hasDelete && (
                <>
                    <DeleteDialog handleArchive={handleArchive}>
handleDelete={handleDelete}>
                    <Button size={"icon"} variant="outline">
                        <FaTrash />
                    </Button>
                    </DeleteDialog>
                    <Button className='gap-2' variant="outline" aria-label='archive' disabled={pendingChanges} onClick={handleArchive}>
                        <FaArchive /> Archive
                    </Button>
                </>
            )
        }
    )
}

```

```

        <Button className='gap-2' disabled={!pendingChanges || !isValidationsEmpty} onClick={handleSave}>
            <FaSave /> {loading ? 'Saving...' : 'Save'}
        </Button>
    </div>
)
}

export default FormButtons

```

### **3.2.5.6.5       ./src/Pages/Tasks/FromComponents/FormInputs.tsx**

```

import React from 'react'
import {Skeleton} from '../../../../../components/@/ui/skeleton'
import PageContainer from '../../../../../components/PageContainer/PageContainer';

function EditTasksSkeleton() {
    return (
        <>
            <div className='flex row justify-between'>
                <Skeleton className="w-40"></Skeleton>
                <div className='flex gap-2'>
                    <Skeleton className='w-20 h-10' />
                    <Skeleton className='w-10' />
                </div>
            </div>
            <div className="space-y-2 py-2">
                <Skeleton className="h-10 w-full" />
                <Skeleton className="h-20 w-full" />
                <Skeleton className="w-full h-10" />
                <Skeleton className="w-full h-10" />
            </div>
        </>
    );
}

export default EditTasksSkeleton

```

The above three components perform no logic on their own. They just take some props in to let the parent component define the logic required for them to work. They just serve to encapsulate some repetitive code and simplify their parent component by being reusable, independent blocks of code.

### **3.2.5.6.6       ./src/Pages/Tasks/FromComponents/Subtasks.tsx**

To implement subtasks, I had to make use of another library named dnd-kit. This library adds drag and drop functionality to subtasks, so that the user can rearrange them to their liking.

```

import { useMemo } from "react";
import { Task } from "../../api/types/taskTypes";
import { arrayMove, SortableContext, useSortable } from "@dnd-kit/sortable";
import { DndContext, closestCenter } from "@dnd-kit/core";
import React from "react";
import { MdDragIndicator } from "react-icons/md";
import CheckBox from "../../../../../components/CheckBox/CheckBox";
import { Input } from "../../../../../components/@ui/input";
import { Button } from "../../../../../components/@ui/button";
import { FaRegTrashAlt } from "react-icons/fa";
import { useTasks } from "../useTasks";
import { CSS } from '@dnd-kit/utilities';

```

```
function Subtasks({ task }: { task: Task }) {
```

Firstly, the **Subtasks** component receives task data (of type **Task**) as a prop.

```
const subtasksId = useMemo(() => task.subtasks.map(st => st.subtaskid), [task.subtasks]);
```

The **useMemo** hook is used to memoize the **subtasksId** array. This optimization prevents unnecessary re-creation of the array if the task's subtasks don't change, improving performance.

```

const {updateCurrentTask, section, handleRemoveSubtask, toggleSubtaskCompleted} = useTasks();

const onDragEnd = (event) => {
  const { active, over } = event;
  if (active.id !== over.id) {
    const oldIndex = subtasksId.indexOf(active.id);
    const newIndex = subtasksId.indexOf(over.id);
    const newSubtasks = arrayMove(task.subtasks, oldIndex, newIndex).map((subtask, index) => ({
      ...subtask,
      index,
    }));
    updateCurrentTask('subtasks', newSubtasks);
  }
};
```

This handler is called when a drag-and-drop operation ends. It calculates the new order of subtasks and updates the task data, using the **map** function.

```

return (
  <DndContext onDragEnd={onDragEnd} collisionDetection={closestCenter}>
    <SortableContext items={subtasksId}>
      {task.subtasks
        .slice()
        .sort((a, b) => a.index - b.index)
        .map((subtask) => (
          <SortableItem
            key={subtask.subtaskid}
            task={task}

```

```

        subtask={subtask}
        section = {section}
        handleRemoveSubtask={handleRemoveSubtask}
        toggleSubtaskCompleted={toggleSubtaskCompleted}
        updateCurrentTask={updateCurrentTask}

      />
    )}
  </SortableContext>
</DndContext>
);
}

```

DndContext establishes the context for drag-and-drop operations. It wraps the entire section of the component where drag-and-drop is enabled. It takes a function for its onDragEnd prop. This function will be called by dnd-kit whenever a drag-and-drop operation within this context is completed. It's crucial for handling the reordering logic. The collisionDetection prop takes a value "closestCenter". This is a collision detection strategy provided by dnd-kit. It determines how the library detects when a dragged item is over another item. closestCenter means that the library will consider an item as being over another when the center of the dragged item is closest to the center of the other item.

To render the subtasks, the subtasks array of the task constant is taken and several methods are applied to it:

- .slice(): This creates a copy of the task.subtasks array. This is important because the sort() method mutates the array it's called on. By creating a copy, we avoid directly modifying the original task.subtasks array, which is good practice for data management in React.
- .sort((a, b) => a.index - b.index): This sorts the copied subtasks array based on the index property of each subtask. The index property likely represents the order of the subtasks. This ensures that the subtasks are rendered in the correct order, even if they have been reordered by the user. The sort function compares the index property of two subtasks (a and b). If a.index is less than b.index, a should come before b in the sorted array, and vice-versa.
- .map((subtask) => ...): This maps over the sorted subtasks array. For each subtask, it renders a SortableItem component.

The subtasks component itself is composed of SortableItems. These items contain all necessary input fields and buttons, as well as a "handle" that the user can grab with their mouse and move around, and the subtask would follow. In other words, it defines the container for the sortable items. It tells dnd-kit which items within this container can be reordered.

```

function SortableItem({ task, subtask, section, handleRemoveSubtask,
toggleSubtaskCompleted, updateCurrentTask }) {
  const { attributes, listeners, setNodeRef, transform, transition } =
useSortable({ id: subtask.subtaskid });

```

**useSortable** is provided by the **dnd-kit** library and uses the subtask ID as a key to distinguish sortable items.

```

const style = {
  transform: CSS.Transform.toString(transform),
  transition,
};

return (
  <div ref={setNodeRef} style={style} {...attributes} className="flex pt-3 gap-2 items-center">
    <MdDragIndicator size={"25px"} className="cursor-grab mr-1" {...listeners} />
    <CheckBox
      checked={subtask.completed}
      onChange={() => toggleSubtaskCompleted(subtask.subtaskid, task.taskid)}
    />
    <Input
      id="subtask"
      name="Subtask description*"
      required
      type="text"
      value={subtask.description}
      onChange={(e) =>
        updateCurrentTask('subtasks', task.subtasks.map((st) =>
          st.subtaskid === subtask.subtaskid ? { ...st, description: e.target.value } : st
        ))
      }
    />
    <Button onClick={() => handleRemoveSubtask(subtask.subtaskid)} variant="secondary" size="icon">
      <FaRegTrashAlt />
    </Button>
  </div>
);
}

export default Subtasks;

```

### 3.2.5.6.7        **./src/Pages/Tasks/useTasks.ts**

```

import { create } from 'zustand';
import { immer } from 'zustand/middleware/immer';
import tasksApi from '../api/tasksApi';
import { Task, TaskResponse } from '../api/types/taskTypes';
import { useTags } from '../Tags/useTags';
import { v4 as uuidv4 } from 'uuid';

```

```

import { UUID } from 'crypto';
import { TaskSchema } from '../../formValidation';
import { z } from 'zod';
import { showToast } from '../../components/Toast';
import utilsApi from '../../api/archiveApi';

const generateNewCurrentTask = () => {

  return {
    taskid: uuidv4(),
    noteid: uuidv4(),
    title: '',
    tags: [],
    content: '',
    subtasks: [],
    due_date: undefined,
    completed: false,
  };
};

type TaskState = {
  loading:boolean;
  setLoading: (loading: boolean) => void;

  currentTask: Task;
  // setCurrentTask: (task: Task) => void;
  resetCurrentTask: () => void;
  updateCurrentTask: (key: keyof Task, value: any) => void;

  tasks: Task[];
  handleAddSubtask: () => void;
  handleRemoveSubtask: (subtaskId:UUID) => void;
  handleSaveTask: () => Promise<boolean>;
  handleEditTask: () => Promise<void>;
  handleDeleteTask: (taskId:UUID, noteId:UUID) => Promise<void>;
  toggleTaskCompleted: (taskId:UUID) => Promise<void>;
  toggleSubtaskCompleted: (subtaskId:UUID, taskId:UUID) => Promise<void>;
  archive: (noteId:UUID) => Promise<void>;

  fetchTaskPreviews: (pageParam:number) => Promise<void>;
  fetchTask: (noteId:string) => Promise<void>;
  nextPage: number | null;
  page:number

  pendingChanges:boolean
  setPendingChanges(value: boolean): void;

  validationErrors: Record<string, string | undefined>;
  validateTask: () => boolean;
};

```

We start by defining a function that generates new default values for a task and creating a type, to which all functions and variables in `useTasks` will adhere to. Then, we create the store using Zustand and Immer and assign default values to some variables, as well as

creating basic state management functions using set to update boolean loading and pendingChanges variables, which show whether there is an active API request or if there are any changes to the file. The hook maintains a complex data model, representing tasks and their associated subtasks

```
export const useTasks = create<TaskState>()(  
  immer((set, get) => ({  
    selectedTagIds: [],  
    tasks: [],  
    loading: false,  
    validationErrors: {},  
    currentTask: generateNewCurrentTask(),  
    page: 1,  
    nextPage:null,  
  
    pendingChanges: false,  
  
    setPendingChanges: (value) => set({pendingChanges: value}),  
    setLoading : (loading) => set({ loading }),  
  
    validateTask: () => {  
      const { currentTask } = get();  
      const result = TaskSchema.safeParse(currentTask);  
      if (!result.success) {  
        set((state) => {  
          const errors = Object.fromEntries(  
            Object.entries(result.error.flatten().fieldErrors).map(([key, value])  
=> [key, value.join(", ", "")])  
          );  
          state.validationErrors = errors;  
        });  
        return false;  
      } else {  
        set((state) => {  
          state.validationErrors = {};  
        });  
        return true;  
      }  
    },  
,
```

The validateTask function employs a Zod schema to validate the currentTask object, and if validation fails, it populates the validationErrors record with detailed automatically generated error messages.

```
updateCurrentTask: <K extends keyof Task>(key: K, value: Task[K]) => {  
  set((state) => {  
    if (state.currentTask) {  
      state.currentTask[key] = value;  
    }  
  })
```

```

        if (!state.pendingChanges) {
            state.pendingChanges = true;
        }

    });
    get().validateTask();
},

```

To perform any update on the selected task, `updateCurrentTask` is used. It uses a generic type `K`, which must be fully contained in the `Task` type to safely update some specified property of the `CurrentTask` object.

```

resetCurrentTask: () => {
    useTags.getState().selectedTagIds = [];
    set((state) => {
        state.currentTask = generateNewCurrentTask()
        state.pendingChanges = false;
    })
},

```

This function is responsible for resetting the `currentTask` state to its default, empty state. It achieves this by calling `generateNewCurrentTask()` to create a fresh task object and then using the `Zustand` `set` function to update the `currentTask` within the store. Additionally, it clears any selected tag IDs by directly manipulating the `selectedTagIds` array in the `useTags` store, ensuring that the tag selection state is also reset. Finally, the `pendingChanges` flag is set to `false`.

```

archive: async (noteId: UUID) => {
    const response = await utilsApi.archive(noteId);
    if (response.success) {
        set((state) => {
            state.tasks = state.tasks.filter((task) => task.noteid !== noteId);
        });
        showToast('success', 'Task archived successfully');
    } else {
        showToast('error', response.message);
    }
},

```

The `archive` function handles the process of archiving a task. It's an asynchronous operation, indicated by the `async` keyword, which calls the `utilsApi.archive` function to communicate with the backend and perform the archiving operation. Upon a successful API call, the function updates the `tasks` array in the store by filtering out the archived task, effectively removing it from the list of active tasks. A success toast containing a message is then displayed to the user. If the API call encounters an error, the function displays an error message to the user using `showToast`.

```

fetchTaskPreviews: async (pageParam: number) => {

```

```

        set({ loading: true });
        try {
            const response = await tasksApi.getTaskPreviews(pageParam);
            if (response && response.success) {
                set({ tasks: response.data, nextPage: response.nextPage, page:
response.page });
            }
            else {
                showToast('error', response.message);
            }
        } finally {
            set({ loading: false });
        }
    },

```

This asynchronous function is designed to retrieve a paginated list of task previews from the backend. It uses try-else block for error handling and begins by setting the loading state to true to indicate that data is being fetched. The function then calls tasksApi.getTaskPreviews with the provided pageParam to request the data. If the API call is successful, the function updates the tasks, nextPage, and page state variables in the Zustand store with the received data. If the API call fails, an error message is displayed using showToast. Regardless of success or failure, the loading state is set to false within a finally block to ensure that the loading indicator is always updated.

```

fetchTask: async (noteId: string) => {
    const{setSelectedTagIds} = useTags.getState();
    set({ loading: true });
    try {
        const response = await tasksApi.getTask(noteId);
        if (response.success && response.data) {
            const dueDate = response.data.due_date ? new
Date(response.data.due_date) : undefined;
            set((state) => {
                state.currentTask = { ...response.data, due_date: dueDate };
            });
            if (response.data.tags)
                setSelectedTagIds(response.data.tags.map(tag=>tag.tagid));
        } else {
            showToast('error', response.message);
        }
    } finally {
        set({ loading: false });
    }
},

```

fetchTask is responsible for retrieves the complete details of a single task from the backend. It calls tasksApi.getTask with the provided noteId to fetch the task data. If the API call is successful, the function updates the currentTask state in the Zustand store with the retrieved data. If the task includes a due date, it converts the date string from the API response (which in turn is the PostgreSQL DATE datatype) into a JavaScript Date object. Moreover, it also calls useTags.getState().setSelectedTagIds() to update the selected tag

IDs in the `useTags` store, ensuring that the tag selection mimics the tags associated with the fetched task.

```
handleAddSubtask: () => {
  set((state) => {
    const subtasks = state.currentTask.subtasks;
    subtasks.push({ subtaskid: uuidv4(), description: '', completed: false,
index: subtasks.length });
  });
  get().validateTask();
},

handleRemoveSubtask: (subtaskid: string) => {
  set((state) => {
    const subtasks = state.currentTask.subtasks.filter((subtask) =>
subtask.subtaskid !== subtaskid);
    subtasks.forEach((subtask, index) => (subtask.index = index));
    state.currentTask.subtasks = subtasks;
  });
  get().validateTask();
},
```

Above few functions edit the state by pushing dictionaries containing default data when a new subtask is created and filtering the subtasks array to remove the subtask of a provided ID.

```
handleSaveTask: async () => {
  const { currentTask } = get();
  const { selectedTagIds } = useTags.getState();
  if (get().validateTask()) {
    const response = await tasksApi.create(currentTask.title, selectedTagIds,
currentTask.content, currentTask.subtasks, currentTask.due_date);
    if (response.data && response.success) {
      set((state) => {
        state.tasks.push({ ...currentTask, taskid: response.data.taskid,
noteid: response.data.noteid });
        state.pendingChanges = false;
      });
      localStorage.setItem('currentTaskId', response.data.noteid.toString());
      showToast('success', 'Task created successfully');
      return true;
    } else {
      showToast('error', response.message);
    }
  } else {
    showToast('error', 'Validation failed');
  }
  return false;
},
```

This code saves a new task to the backend. It retrieves the current task data and the selected tag IDs from the `useTags` store. If the task is valid, the function calls `tasksApi.create()` to send the new task data to the backend. If the API call is successful,

the function updates the tasks array in the Zustand store by adding the newly created task (including the taskid and noteid received from the backend). The pendingChanges flag is set to false, and the new task's noteid is stored in localStorage. A success or error message is displayed to the user.

```
handleEditTask: async () => {
  const { currentTask } = get();
  const { selectedTagIds } = useTags.getState();
  try {
    if (get().validateTask()) {
      const updatedTask = { ...currentTask, tags: selectedTagIds };
      const response = await tasksApi.update(updatedTask);

      if (response && response.success) {
        set((state) => {
          state.tasks = state.tasks.map((task) => (task.taskid ===
currentTask.taskid ? currentTask : task));
          state.pendingChanges = false;
          state.loading = false;
        });
      } else {
        showToast('error', response.message);
      }
    } else {
      showToast('error', 'Validation failed');
    }
  }
},
```

handleEditTask function is used to update an existing task. It retrieves and validates the task. If the task data is valid, it calls the tasksApi to send the updated task data to the server, with the same response mechanism using toasts.

```
handleDeleteTask: async (taskId: UUID, noteId: UUID) => {
  if (taskId && noteId) {
    const previousTasks = get().tasks;

    set((state) => {
      state.tasks = state.tasks.filter((task) => task.taskid !== taskId);
    });

    const response = await tasksApi.delete(taskId, noteId);
    if (response && response.success) {
      showToast('success', 'Task deleted successfully')
    } else {
      set({ tasks: previousTasks }); //revert
      showToast('error', response.message);
    }
  }
},
```

After taking taskId and notId of the task to be deleted as arguments, handleDeleteTask first optimistically updates the UI by removing the task from the tasks array in the Zustand store. It then calls tasksApi.delete() to delete the task from the backend.

```
toggleTaskCompleted: async (taskId: UUID) => {
  set((state) => {
    state.tasks = state.tasks.map((task) => (task.taskid === taskId ?
{ ...task, completed: !task.completed } : task));
  });

  const response = await tasksApi.toggleCompleteness(taskId, null);
  if (!response || !response.success) {
    set((state) => {
      state.tasks = state.tasks.map((task) => (task.taskid === taskId ?
{ ...task, completed: !task.completed } : task));
    });
    showToast('error', 'Failed to toggle task completeness');
  }
},
```

This function toggles the completion status of a task. It also optimistically updates the tasks array in the Zustand store by toggling the completed status of the specified task, before calling the API to update the task's completion status on the backend. If the API call fails, the function reverts the optimistic update by toggling the completed status back to its original value and displays an error toast.

```
toggleSubtaskCompleted: async (subtaskId, taskId) => {
  let all_completed=true
  set((state) => {
    state.tasks = state.tasks.map((task) => {
      if (task.taskid === taskId) {
        const newSubtasks = task.subtasks.map((subtask) => {
          if (subtask.subtaskid === subtaskId) {
            subtask = { ...subtask, completed: !subtask.completed };
          }
          if (subtask.completed !== true) {
            all_completed = false;
          }
        })
        return subtask;
      });
      return { ...task, completed:true, subtasks: newSubtasks };
    })
    return task;
  });
  try {
    if (all_completed) {
      await tasksApi.toggleCompleteness(taskId, null);
    }
  }
```

```

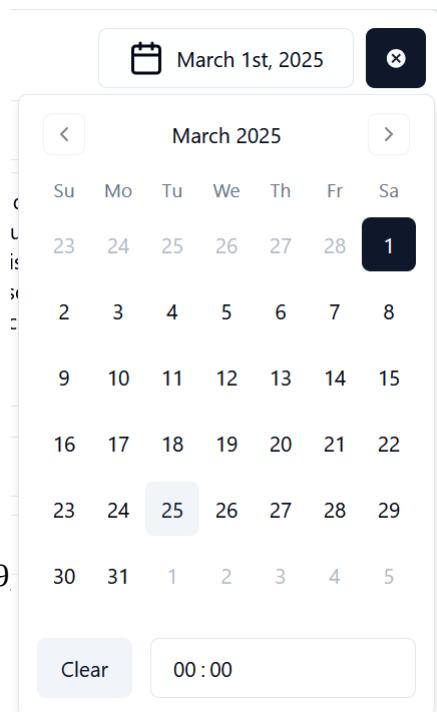
        await tasksApi.toggleCompleteness(taskId, subtaskId);
    } catch (error) {
        // Revert subtask completion on failure
        set((state) => {
            state.tasks = state.tasks.map((task) => {
                if (task.taskid === taskId) {
                    const revertedSubtasks = task.subtasks.map((subtask) =>
                        subtask.subtaskid === subtaskId ? { ...subtask, completed: !subtask.completed } : subtask
                    );
                    return { ...task, completed:false, subtasks: revertedSubtasks };
                }
                return task;
            });
        });
    },
),
));
);

```

Finally, a similar method is used to toggle a subtask. The function first updates the completion of a selected subtask. Then, the .map method is used to iterate over the whole array and check if all subtasks were completed. If this is true, apart from the subtask completion, a task completion request is sent to the server. If any errors occur, the optimistic update of the task itself and the subtask is reverted.

### 3.2.5.6.8        [./src/Pages/Tasks/DatePicker/DatePicker.tsx](#)

The date picker is used to select a due date on many different notetypes throughout the website. It has two types: including and excluding time. The two types, obviously, differ by containing a clock or omitting it. The user can select any month or year, which will be updated in the component's state.



```

import * as React from "react"
import { format } from "date-fns"
import { Calendar as CalendarIcon } from "lucide-react"

import { cn } from "../../../../../components/@/lib/utils"
import { Button } from "../../../../../components/@/ui/button"
import { Calendar } from "../../../../../components/@/ui/calendar"
import {
  Popover,
  PopoverContent,
  PopoverTrigger,
} from "../../../../../components/@/ui/popover"
import { Input } from "../../../../../components/@/ui/input"
import { useState, useEffect } from "react"

interface DatePickerProps {
  onDateChange: (newDate: Date | undefined) => void;
  data?: Date;
  includeTime?: boolean;
}

export function DatePicker({ onDateChange, data, includeTime = false }: DatePickerProps) {

```

Having passed the `onChange` function into the picker, `includeTime` is defaulted to `false`. The calendar is imported from the shadcn UI library, as well as some other components. To convert different date formats, the `date-fns` library is used

```

const [dateTime, setDateTime] = useState<Date | undefined>(data);

useEffect(() => {
  if (data instanceof Date && !isNaN(data.getTime())) {
    setDateTime(data);
  } else {
    setDateTime(undefined);
  }
  console.log(data)
}, [data]);

```

This `useEffect` hook validates that the passed data is of TypeScript type `Date` and that its `time` property is a number.

```

const handleDateSelect = (newDate: Date | undefined) => {
  if (newDate && dateTime) {
    newDate.setHours(dateTime.getHours(), dateTime.getMinutes());
  }
}

```

```

    }
    setDateDateTime(newDate);
    onDateChange(newDate || undefined);
};


```

HandleDateSelect uses an if statement to check if both newDate and dateTime are defined. If so, the newDate object is modified using setHours() and getMinutes() to combine the selected date with the existing time from dateTime. Finally, the state is updated and the function calls the onDateChange prop to notify the parent component of the date change, passing either the newDate or undefined.

```

const handleTimeSelect = (timeString: string) => {
  const timeParts = timeString.split(':').map(Number);
  const newDateTime = dateTime ? new Date(dateTime) : new Date();

  if (timeParts.length === 2) {
    newDateTime.setHours(timeParts[0], timeParts[1]);
    setDateDateTime(newDateTime);
    onDateChange(newDateTime);
  }
};


```

TimeSelect uses string manipulation to split the string into hours and minutes. Then, after checking if the string is of expected format, it creates a new Date object, updates the component state and calls the onDateChange prop

```

const handleClear = () => {
  setDateDateTime(undefined);
  onDateChange(undefined);
};

const formattedTime = (dateTime && includeTime) ? format(dateTime, "HH:mm") : "";

return (
  <div>
    <Popover>
      <PopoverTrigger asChild>
        <Button
          variant="outline"
          className={cn("w-[170px] justify-center font-normal", !dateTime && "text-muted-foreground")}>
          <CalendarIcon className="mr-2 min-h-4 min-w-4" />
          {dateTime ? format(dateTime, "PPP") : <span>Reminder</span>}
        </Button>
      </PopoverTrigger>
      <PopoverContent className="w-auto p-0">
        <Calendar
          mode="single"
          selected={dateTime}
        </Calendar>
      </PopoverContent>
    </Popover>
  </div>
);


```

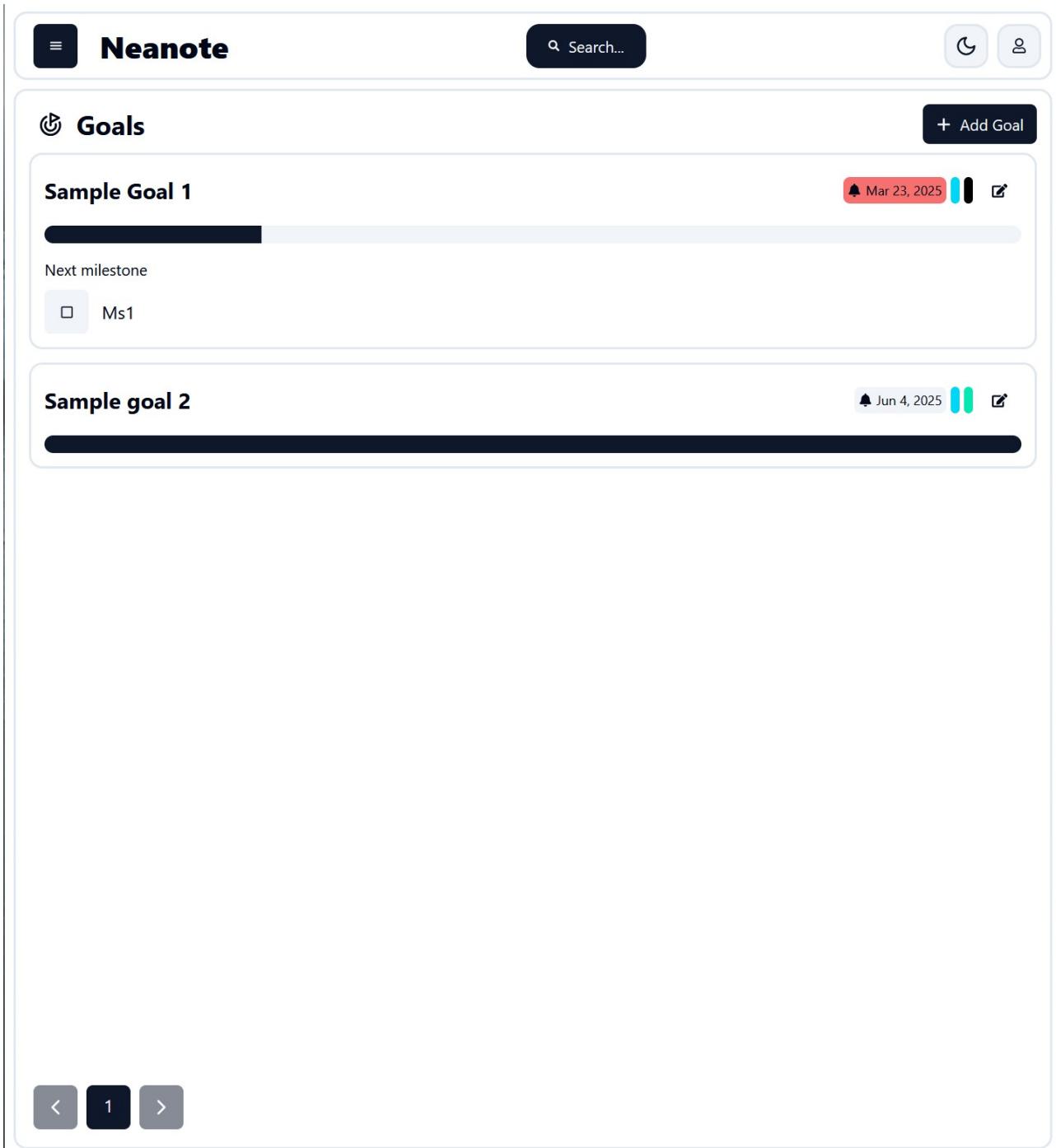
```

        onSelect={handleDateSelect}
        initialFocus
    />
    <div className="flex flex-row gap-3 p-3">
        <Button className="w-1/4 items-center" variant="secondary"
onClick={handleClear}>Clear</Button>
        {includeTime && (
            <Input
                id="date"
                name="dateTime"
                type="time"
                value={formattedTime}
                onChange={(e) => handleTimeSelect(e.target.value)}
                className="w-3/4 justify-center"
            />
        )}
    </div>
    </PopoverContent>
</Popover>
</div>
);
}

```

Lastly, the component is rendered, with some parts like the time input being conditionally rendered based on the includeTime prop.

### 3.2.5.7 Goals



The goals component has a nearly identical structure to the Tasks component. What differs here is the structure of the note, so other files must be made to adapt for new fields and datatypes that the Goals type brings. Therefore, I will skim through this and other note types. For any in depth explanations, refer to [3.2.5.6](#)

### 3.2.5.7.1        [./src/Pages/Goals/Goals.tsx](#)

```
import React, { useEffect, useState } from 'react'  
import PageContainer from '.../.../.../components/PageContainer/PageContainer'  
import { Button } from '.../.../.../components/@/ui/button'  
import { FaPlus } from 'react-icons/fa6'
```

```

import { useNavigate } from 'react-router-dom';
import { useGoals } from './useGoals';
import GoalCard from './GoalCard/GoalCard';
import { LuGoal } from "react-icons/lu";
import TitleComponent from '../../../../../components/TitleComponent/TitleComponent';
import PaginationSelector from '../../../../../components/Pagination/PaginationSelector';

function Goals() {
    const navigate = useNavigate();
    const {resetCurrentGoal, goalPreviews, fetchGoalPreviews, nextPage, page} =
useGoals();

    const handleAddGoalClick = () => {
        resetCurrentGoal();
        navigate('/goals/create')
    }
    const [lastFetchTime, setLastFetchTime] = useState<Date | null>(null);

    useEffect(() => {
        const fetchIfNeeded = () => {
            // Check if never fetched or if 5 minutes have passed since the last fetch
            if (!lastFetchTime || new Date().getTime() - lastFetchTime.getTime() >
300000) {
                fetchGoalPreviews(page);
                setLastFetchTime(new Date());
            }
        };
        fetchIfNeeded();

        // Set up a timer to refetch every 5 minutes
        const intervalId = setInterval(fetchIfNeeded, 3000);

        // Clean up the interval on component unmount
        return () => clearInterval(intervalId);
    }, [fetchGoalPreviews, lastFetchTime]);

    return (
        <>
            <div className="flex flex-row justify-between pb-2">
                <TitleComponent><LuGoal size={'23px'} /> Goals</TitleComponent>
                <Button size="sm" className="gap-2" onClick={handleAddGoalClick}>
                    <FaPlus />
                    Add Goal
                </Button>
            </div>
            <div className="flex flex-col flex-grow gap-3">
                {goalPreviews.map((goal) => (
                    <div key={goal.goalid} >
                        <GoalCard goal={goal}/>
                    </div>
                ))}
            </div>
            <div className="p-1 pt-2">

```

```

        <PaginationSelector fetchingFunction={fetchGoalPreviews}
nextPage={nextPage} page={page}/>
      </div>
    </>
  )
}

export default Goals

```

The Goals component renders the Goals page, which contains all user's notes of the "goal" type. After all imports, a useEffect hook checks if there are fetched notes on the page and automatically refetches them every 5 minutes to keep the data up to date. Then, the goals page is rendered using components from the UI library, as well as <div> elements with inline CSS styling to make the page aesthetic. Finally, the retrieved goal preview objects are iteratively sent into a GoalCard component as props, using the .map function, which goes over each element in the goalPreviews array.

### **3.2.5.7.1.1    .src/Pages/Goals/GoalCard/GoalCard.tsx**

Unlike other components that do a similar task: displaying a single goal, GoalCard needs to perform a few extra steps to correctly render a Goal note type. The code defines a functional React component named GoalCard that accepts a goal object as a prop. The component receives a goal object, which contains information about the goal, including its title, due date, tags, milestones, and content.

```

import React, { useEffect, useState } from 'react';
import { FaEdit } from 'react-icons/fa';
import { useNavigate } from 'react-router-dom';
import { Button } from '../../../../../components/@ui/button';
import DateLabel from '../../../../../components/DateLabel/DateLabel';
import TagLabel from '../../../../../components/TagLabel/TagLabel';
import SkeletonCard from '../../../../../components/TaskCard/SkeletonCard';
import { Goal } from '../../../../../api/types/goalTypes';
import { useGoals } from '../useGoals';
import { Progress } from '../../../../../components/@ui/progress';
import './GoalCard.css';
import { Label } from '../../../../../components/@ui/label';
import CheckBox from "../../../../../components/CheckBox/CheckBox";
import { UUID } from 'crypto';
import { useScreenSize } from '../../../../../DisplayContext';

```

```

function GoalCard({ goal }: { goal: Goal }) {
  const {
    loading,
    handleMilestoneCompletion
  } = useGoals()

  const navigate = useNavigate()

  function handleEditClick(noteId : UUID) {

```

```

        localStorage.setItem('currentGoalId', noteId);
        navigate('/goals/edit');
    }
    const {isDateCollapsed, isTagCompressed} = useScreenSize()

```

A custom hook is used to access the loading state and the handleMilestoneCompletion function. This hook manages goal-related state and logic, just like useTasks in the Tasks section. Additionally, useScreenSize is a provider which gives the component two boolean variables, which determine the UI state of tags and the due date, as they compress if the user opens the page on their phone, where horizontal screen real estate is limited. The useNavigate hook from react-router-dom is used to programmatically navigate to the goal editing page when the user clicks the edit button.

```

useEffect(() => {
    const progress = calculateProgress();

    var next_milestone =  goal.milestones
        .filter(milestone => !milestone.completed)
        .sort((a, b) => a.index - b.index)[0],[goal.milestones])
//recalculate progressbars on milestone change

    const calculateProgress = () => {
        const sortedMilestones = [...goal.milestones].sort((a, b) => a.index - b.index);
        const completedMilestones = sortedMilestones.filter(milestone => milestone.completed).length;
        return (completedMilestones / sortedMilestones.length) * 100;
    };

    const progress = calculateProgress();

    var next_milestone =  goal.milestones
        .filter(milestone => !milestone.completed)
        .sort((a, b) => a.index - b.index)[0]

```

This code snippet calculates and stores the progress of a goal and determines the next incomplete milestone. CalculateProgress is a helper function that calculates the completion percentage of a goal based on its milestones. [...goal.milestones]: Creates a copy of the goal.milestones array using the spread syntax. This is crucial to avoid mutating the original goal.milestones array. Then, the array is sorted and filtered using the .sort and .filter methods, so that a new array of only completed milestones is created. Then, the .length property of the array is taken to get the total number of milestones and the number of completed milestones. Finally, the progress percentage is calculated with (completedMilestones / totalMilestones) \* 100.

Meanwhile, the useEffect hook is used to perform side effects in a functional component. In this case, it recalculates the next\_milestone every time the component renders. It uses array manipulation to access the next uncompleted milestone to show to the user, while the [goal.milestones] dependency array ensures that this useEffect hook runs only when

the goal.milestones array changes. This is important for performance, as it prevents unnecessary recalculations.

```
if (loading) {
    return <SkeletonCard />;
}

return (
    <div className='p-3 w-full rounded-xl border-[2px]'>
        <div className='flex flex-row items-center gap-3 justify-between'>
            <div className='flex flex-row items-center gap-3'>
                <h3 className="goal-title">{goal.title}</h3>
            </div>
            <div className='flex flex-row items-center gap-1'>
                {goal.due_date && <DateLabel includeTime={false}
collapsed={isDateCollapsed} date={goal.due_date} />}
                {goal.tags.map((tag, index) => (
                    <TagLabel key={index} name={tag.name} color={tag.color}
compressed={isTagCompressed} />
                ))}
                <Button variant="ghost" size={"icon"} onClick={() =>
handleEditClick(goal.noteid)}><FaEdit /></Button>
            </div>
        </div>
        <div className='mt-3'>
            <Progress value={progress}/>
            {next_milestone && (
                <div className='pt-4 flex flex-col gap-3'>
                    <Label>Next milestone</Label>
                    <div className='flex flex-row gap-2'>
                        <CheckBox
checked={next_milestone.completed} onChange={() =>
handleMilestoneCompletion(goal.goalid, next_milestone.milestoneid)} />
                        <p className="text-md pl-1 pt-2">
                            {next_milestone.description}
                        </p>
                    </div>
                </div>
            )}
        </div>
        {goal.content && <p className="text-md pl-1
pt-2">{goal.content}</p>}
    </div>
);
}

export default GoalCard;
```

After conditionally rendering the SkeletonCard component if data is still being fetched, the actual component is rendered with a progressbar, which takes the recently calculated percentage value as props. Another list rendering with .map is used to render the tags corresponding to this note.

If the user clicks a button or a checkbox, appropriate functions are called with onClick and onChange.

### 3.2.5.7.1.2 .src/Pages/Goals/GoalCard/GoalCard.css

Even though I primarily use TailwindCSS for inline CSS styling, some components have more complex styles, that are harder to implement using Tailwind syntax. For this, files like GoalCard.css exist, which encapsulate any leftover styles for this component:

```
.goal-title {  
    font-size: 1.25rem; /* text-xl */  
    font-weight: bold; /* font-bold */  
    overflow: hidden;  
    text-overflow: ellipsis;  
    white-space: nowrap;  
    max-width: 50vw; /* Default max-width for larger displays */  
    flex-grow: 1;  
}
```

### 3.2.5.7.2 .src/Pages/Goals/useGoals.tsx

This file also uses Zustand and immer middleware, as well as binary search and quicksort to optimize goal handling tasks.

```
import { create } from "zustand"  
import { immer } from "zustand/middleware/immer"  
import { Goal, GoalResponse, GoalsPreview } from "../../api/types/goalTypes";  
import { v4 as uuidv4 } from 'uuid';  
import { useTags } from "../Tags/useTags";  
import goalsApi from "../../api/goalsApi";  
import { UUID } from "crypto";  
import { GoalSchema } from "../../formValidation";  
import utilsApi from "../../api/archiveApi";  
import { showToast } from "../../components/Toast";  
import { quicksort, binarySearch } from "../../components/utils";  
  
// Function to generate a new current goal object  
const generateNewCurrentGoal = () => {  
  
    return {  
        goalid: uuidv4(),  
        noteid: uuidv4(),  
        title: '',  
        content: '',  
        due_date: undefined,  
        tags: [],  
        milestones: [  
            { milestoneid: uuidv4(), description: '', completed: false, index: 0, isNew: true },  
            { milestoneid: uuidv4(), description: '', completed: false, index: 1, isNew: true }  
        ]  
    };  
};
```

This subroutine creates a new Goal object and populates it with two default empty milestones.

```
type GoalState = {
    goalPreviews: Goal[];
    currentGoal: Goal;
    resetCurrentGoal: () => void;
    updateCurrentGoal: <K extends keyof Goal>(key: K, value: Goal[K]) => void;
    handleCreateGoal: () => Promise<boolean>;
    handleUpdateGoal: () => Promise<void>;
    handleDeleteGoal: (goalid: UUID, noteid: UUID) => Promise<void>;
    fetchGoalPreviews: (pageParam: number) => Promise<void>;
    fetchGoal: (noteId: string) => Promise<void>

    handleAddMilestone: () => void
    handleRemoveMilestone: (milestoneid:UUID) => void
    handleMilestoneCompletion: (goalid:UUID, milestoneid:UUID) => Promise<void>
    archive: (noteId: UUID) => Promise<void>;
    loading:boolean;
    setLoading: (loading: boolean) => void;
    pendingChanges:boolean
    setPendingChanges(value: boolean): void;
    validationErrors: Record<string, string | undefined>;
    validateGoal: () => boolean;
    page:number
    nextPage:number | null
}

export const useGoals = create<GoalState>()(  
    immer((set, get) => ({  
        goalPreviews: [],  
        validationErrors:{},  
        pendingChanges: false,  
  
        loading: false,  
        setLoading: (loading) => set({loading}),  
        setPendingChanges: (value) => set({pendingChanges: value}),  
  
        page:1,  
        nextPage:null,  
  
        resetCurrentGoal: () => {  
            set(() => ({  
                currentGoal: generateNewCurrentGoal()  
            }));  
        },  
    }),
```

After creating variables with default names, `resetCurrentGoal` resets the `currentGoal` state to a new, default goal object.

```
currentGoal: generateNewCurrentGoal(),

handleCreateGoal: async () => {
  const { currentGoal } = get();
  const { selectedTagIds } = useTags.getState();
  if (get().validateGoal()) {
    const { title, content, due_date, milestones } = currentGoal;
    const response = await goalsApi.create(title, selectedTagIds, content,
due_date, milestones);

    if (response.data && response.success) {
      set((state) => {
        state.currentGoal = { ...currentGoal, goalid: response.data.goalid,
noteid: response.data.noteid };
        state.goalPreviews.push({
          goalid: response.data.goalid,
          noteid: response.data.noteid,
          title,
          content,
          due_date,
          milestones: response.data.milestones,
          tags: [],
        });
      });
      localStorage.setItem('currentGoalId',
response.data.noteid.toString());
      showToast('success', 'Goal created successfully');
      return true;
    } else {
      showToast('error', response.message);
    }
  } else {
    showToast('error', 'Validation failed');
  }
  return false;
},
```

This function creates a new goal via the API and updates the state, using `async`. After validating the goal and setting its ID in local storage, it shows the user a success message and returns a boolean indicating success or failure.

```
handleUpdateGoal: async () => {
  const { currentGoal, resetCurrentGoal } = get();
  const { tags, selectedTagIds } = useTags.getState();

  if (get().validateGoal()) {
    const { goalid, noteid, title, content, due_date, milestones } =
currentGoal;
```

```

const preparedMilestones = milestones.map(milestone => {
  const { isNew, ...rest } = milestone;
  if (isNew) {
    // For new milestones, exclude the milestoneid when sending to
the backend
    const { milestoneid, ...newMilestoneRest } = rest;
    return newMilestoneRest;
  }
  return rest;
});

const updatedGoal = {
  goalid,
  noteid,
  title,
  tags: selectedTagIds,
  content,
  due_date,
  milestones: preparedMilestones,
};

const previousGoals = get().goalPreviews;

// optimistic update
set((state) => {
  state.goalPreviews = state.goalPreviews.map((goal) =>
(goal.goalid === goalid ? {goalid, noteid, title, tags, due_date, milestones, content} : goal));
});

const response = await goalsApi.update(updatedGoal);
if (!response || !response.success) {
  // revert update
  set({ goalPreviews: previousGoals});
} else {
  set({pendingChanges: false })
}

} else {
  showToast('error', 'Validation failed');
}
},

```

This asynchronous function optimistically updates a goal with new user-entered data. After validating the input fields and retrieving the IDs of selected tags, the milestones array is manipulated to send the new milestones without an ID, as it will be generated on the backend. Then, the goal object is sent to the backend and reverted if errors arise, with a corresponding error message shown to the user

```

validateGoal: () => {
  const { currentGoal } = get();
  const result = GoalsSchema.safeParse(currentGoal);

```

```

        if (!result.success) {
            set((state) => {
                const errors = Object.fromEntries(
                    Object.entries(result.error.flatten().fieldErrors).map(([key, value]) => [key, value.join(", ")])
                );
                state.validationErrors = errors;
            });
            return false;
        } else {
            set((state) => {
                state.validationErrors = {};
            });
            return true;
        }
    },
}

handleAddMilestone: () =>
set((state) => {
    if (state.currentGoal) {
        const milestones = state.currentGoal.milestones;
        milestones.splice(milestones.length - 1, 0, {
            milestoneid: uuidv4(),
            description: '',
            completed: false,
            index: milestones.length,
            isNew: true,
        });
        // Sort milestones by index using quicksort
        state.currentGoal.milestones = quicksort(milestones, (a, b) =>
a.index - b.index);
        // Update indices
        state.currentGoal.milestones.forEach((ms, idx) => ms.index =
idx);
    }
}),

```

This adds a new empty milestone to the milestones array. `.splice()` Inserts the new milestone at the correct position. Then, milestones are quicksorted by index and the `forEach` method updates the `index` property of each milestone

```

fetchGoalPreviews: async (pageParam: number) => { //Fetch goal previews
useGoals.getState(). setLoading(true);
const response = await goalsApi.getGoalPreviews(pageParam);
if (response && response.data) {

    const previewsWithFormattedDates = response.data.map(preview => ({
        ...preview,
        due_date: preview.due_date ? new Date(preview.due_date + Z) :
undefined,
    }));
}

```

```

        set({ goalPreviews: previewsWithFormattedDates, nextPage:
response.nextPage, page: response.page });
    } else{
        showToast('error', response.message);
    }
    useGoals.getState().setLoading(false);
},
fetchGoal: async(noteId:string) => { //Fetch a single goal of known ID
    const {setSelectedTagIds} = useTags.getState();
    try{
        useGoals.getState().setLoading(true);
        const response = await goalsApi.getGoal(noteId);
        if (response && response.data) {
            const dueDate = response.data.due_date ? new
Date(response.data.due_date + Z) : undefined;
            const goalWithFormattedDate = {
                ...response.data,
                due_date: dueDate, //format goal date to TS Date format
            };
            set((state) => {
                state.currentGoal = goalWithFormattedDate;
            });
            setSelectedTagIds(response.data.tags.map(tag=>tag.tagid));
        } else {
            showToast('error', response.message);
        }
    } finally {
        set({ loading: false });
    }
},
handleRemoveMilestone: (milestoneid) => {
    set((state) => {
        if (state.currentGoal) {
            const milestones =
state.currentGoal.milestones.filter((milestone) => milestone.milestoneid !==
milestoneid);
            milestones.forEach((ms, idx) => ms.index = idx); //Creates a
new array of milestones without the milestone of provided id
            state.currentGoal.milestones = milestones;
        }
    })
},
handleMilestoneCompletion: async (goalid, milestoneid) => {
    const toggleMilestoneCompletion = (goalid, milestoneid) => {
        set((state) => {
            state.goalPreviews = state.goalPreviews.map((goal) => {
                if (goal.goalid === goalid) {
                    const newMilestones = goal.milestones.map((milestone) =>
                        milestone.milestoneid === milestoneid ? { ...milestone,
completed: !milestone.completed } : milestone
                );
            });
        });
    };
}

```

```

        return { ...goal, milestones: newMilestones };
    }
    return goal;
});

state.currentGoal = {...state.currentGoal, milestones:
state.currentGoal.milestones.map((milestone) =>
    milestone.milestoneid === milestoneid ? { ...milestone,
completed: !milestone.completed } : milestone
)
}
})
},
}

// Toggle completion before API call
toggleMilestoneCompletion(goalid, milestoneid);

const response = await goalsApi.completeMilestone(goalid,
milestoneid);
if (!response || !response.success) {
    // Revert completion if API call fails
    toggleMilestoneCompletion(goalid, milestoneid);
}
},

```

Milestones must be completed for the progressbar to move. This function toggles the completion status of a milestone. This function is also optimistic, so the UI is updated before the API call and is reverted back if the call fails. To update the completed property of a goal, the currentGoal object is taken from the state and broken down into its components using the spread ... operator. There, the .map method is used to find a correct milestone, which boolean value under its completed property is reverted to an opposite value.

```

archive: async (noteId: UUID) => {
    const response = await utilsApi.archive(noteId);
    if (response && response.success) {
        set((state) => {
            state.goalPreviews = state.goalPreviews.filter((goal) =>
goal.noteid !== noteId);
        });
        showToast('success', 'Goal archived successfully');

    } else {
        showToast('error', response.message);
    }
},

```

Archiving requires only removing the selected goal from the goalPreviews array, since everything else is done on the backend, so .filter is used/

```
updateCurrentGoal: <K extends keyof Goal>(key: K, value: Goal[K]) => {
```

```
    set((state) => {
      if (state.currentGoal) {
        state.currentGoal[key] = value;
        state.pendingChanges = true;
      }
    });
  get().validateGoal();
}
```

`updateCurrentGoal` works just like a similarly named function in `useTasks` by taking a goal of generic type `K` and updating some property from the provided key: new value pair. It also sets `pendingChanges` to true to signify that some changes were made to the currently edited goal.

```
handleDeleteGoal: async (goalid, noteid) => {
    const previousGoals = get().goalPreviews;

    set((state) => {
        // Use binary search to efficiently find and remove the goal
        const sortedGoals = quicksort([...state.goalPreviews], (a, b) =>
            a.goalid.toString().localeCompare(b.goalid.toString()))
    });

    const comparator = (goal: Goal, id: UUID) =>
        goal.goalid.toString().localeCompare(id.toString());

    const index = binarySearch(sortedGoals, goalid, comparator);

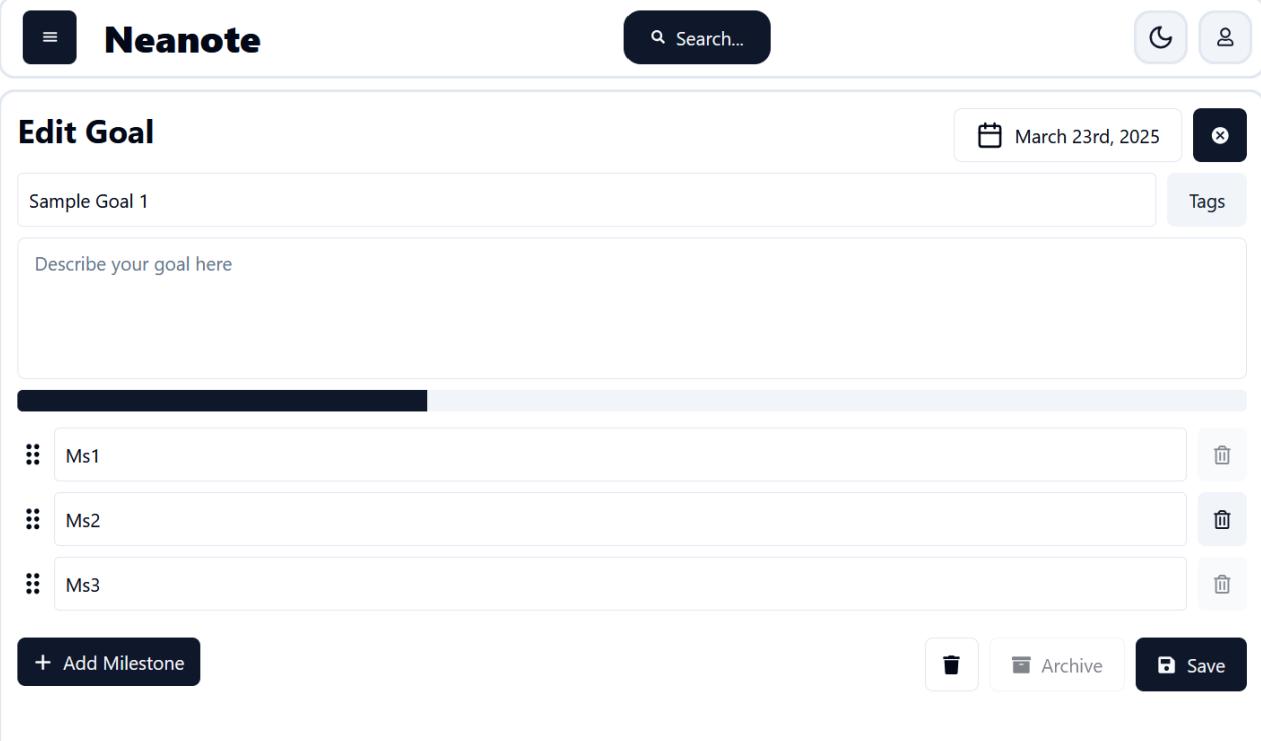
    if (index !== -1) {
        // Find the actual index in the unsorted array
        const foundGoal = sortedGoals[index];
        const actualIndex = state.goalPreviews.findIndex(g => g.goalid
        === foundGoal.goalid);

        if (actualIndex !== -1) {
            state.goalPreviews.splice(actualIndex, 1);
        }
    });
};

const response = await goalsApi.delete(goalid, noteid);
if (!response) {
    // revert deletion
    set({ goalPreviews: previousGoals });
}
},
})))
```

Lastly, binary search and quicksort are used to find and delete a specific goal. After defining appropriate comparator functions, and sorting the `goalPreviews` array, binary search is used to find the index of the goal to be deleted. If such index exists, the goal is removed from the array and an optimistic API call on the `goalsAPI` is sent. If it fails, the action is reverted.

### 3.2.5.7.3      ./src/Pages/Goals/EditGoals.tsx



The screenshot shows the 'Edit Goal' page of the Neanote application. The page has a header with the Neanote logo and a search bar. A date indicator shows 'March 23rd, 2025'. The main content area is titled 'Edit Goal' and contains a 'Sample Goal 1' input field, a large text area for goal description, and a list of three milestones: Ms1, Ms2, and Ms3. Each milestone has a delete icon. At the bottom, there are buttons for '+ Add Milestone', 'Archive', and 'Save'.

The EditGoal component contains all of the inputs and toggles to create or edit a goal. This specific component is tweaked to edit a goal, but there are little differences from the CreateGoal component, except for some UI elements

```
import React, { useEffect, useState } from 'react';
import { FaPlus } from 'react-icons/fa6';
import { MdCancel } from 'react-icons/md';
import { useNavigate } from 'react-router-dom';
import { Button } from '../../../../../components/@/ui/button';
import { Progress } from '../../../../../components/@/ui/progress';
import DeleteDialog from '../../../../../components/DeleteDialog/DeleteDialog';
import PageContainer from '../../../../../components/PageContainer/PageContainer';
import { useTags } from '../Tags/useTags';
import { DatePicker } from '../Tasks/DatePicker/DatePicker';
import EditGoalsSkeleton from './EditGoalsSkeleton';
import Inputs from './FormComponents/Inputs';
import Milestones from './FormComponents/Milestones';
import { useGoals } from './useGoals';
import { Label } from '../../../../../components/@/ui/label';
import FormButtons from '../../../../../components/FormButtons/FormButtons';

function EditGoals() {
  const {currentGoal, loading, pendingChanges, archive, validationErrors,
  handleDeleteGoal, fetchGoal, resetCurrentGoal, handleUpdateGoal,
  handleAddMilestone, updateCurrentGoal} = useGoals();
  const navigate = useNavigate();

  useEffect(() => {
```

```

const noteId = localStorage.getItem('currentGoalId');
if (noteId) {
  fetchGoal(noteId);
  if (currentGoal.tags) {
    const mappedTagIds = currentGoal.tags.map(tag => tag.tagid);
    useTags.setState({
      selectedTagIds: mappedTagIds,
    });
  }
}, []);

```

When the user clicks on a goal card, its ID is stored in local storage. This component reads the ID from the local storage and fetches the goal from the database using `fetchGoal` discussed previously

```

const [isValidationsEmpty, setIsValidationsEmpty] = useState(true);

useEffect(() => {
  setIsValidationsEmpty(
    Object.keys(validationErrors).every(key => !validationErrors[key])
  );
}, [validationErrors]);

```

Just like during creation, user inputs must be validated through a schema. This code does not allow the user to submit his goal until all validation errors were resolved

```

const handleArchive = async () => {
  await archive(currentGoal?.noteid);
  navigate('/goals');
}

const calculateProgress = () => {
  const sortedMilestones = [...currentGoal.milestones].sort((a, b) => a.index - b.index);
  const completedMilestones = sortedMilestones.filter(milestone => milestone.completed).length;
  return (completedMilestones / sortedMilestones.length) * 100;
};

const progress = calculateProgress();

```

Since this component also has a progressbar, it must perform the same logic as the `goalCard` component to calculate the percentage progress

```

const handleClose = () => {
  localStorage.removeItem('currentGoalId');
  useGoals.setState({
    pendingChanges: false,
    validationErrors: {},
  })
}

```

```

useTags.setState({
  selectedTagIds: [],
})
resetCurrentGoal()
navigate('/goals');
};

const handleSave = async () => {
  await handleUpdateGoal();
}

const handleDelete = async () => {
  await handleDeleteGoal(currentGoal?.goalid, currentGoal?.noteid)
  navigate('/goals');
}

```

These handlers run an appropriate async function and use the useNavigate hook from react-router to navigate the user to a different page, if an action like delete or archive implies a removal of the goal from the list of goals

```

if (loading) return <EditGoalsSkeleton/>

return (
  <>
    <div className='flex row justify-between'>
      <h1 className="text-2xl font-bold mb-4">Edit Goal</h1>
      <div className='flex gap-2'>
        <DatePicker onDateChange={(date) => updateCurrentGoal('due_date', date)} data={currentGoal.due_date} includeTime={false} />
        <Button size='icon' onClick={handleClose}>
          <MdCancel size={15} />
        </Button>
      </div>
    </div>
    <Inputs content={currentGoal.content} title={currentGoal.title}/>
    <div className="mb-3">
      <Progress className='rounded-sm mb-3' value={progress}/>
      <Milestones goal={currentGoal}/>
    </div>
    {validationErrors['milestones'] && (
      <Label className='text-destructive'>{validationErrors['milestones']}

```

```
        handleDelete={handleDelete}
      />
    </div>
  </div>
</>
);
}

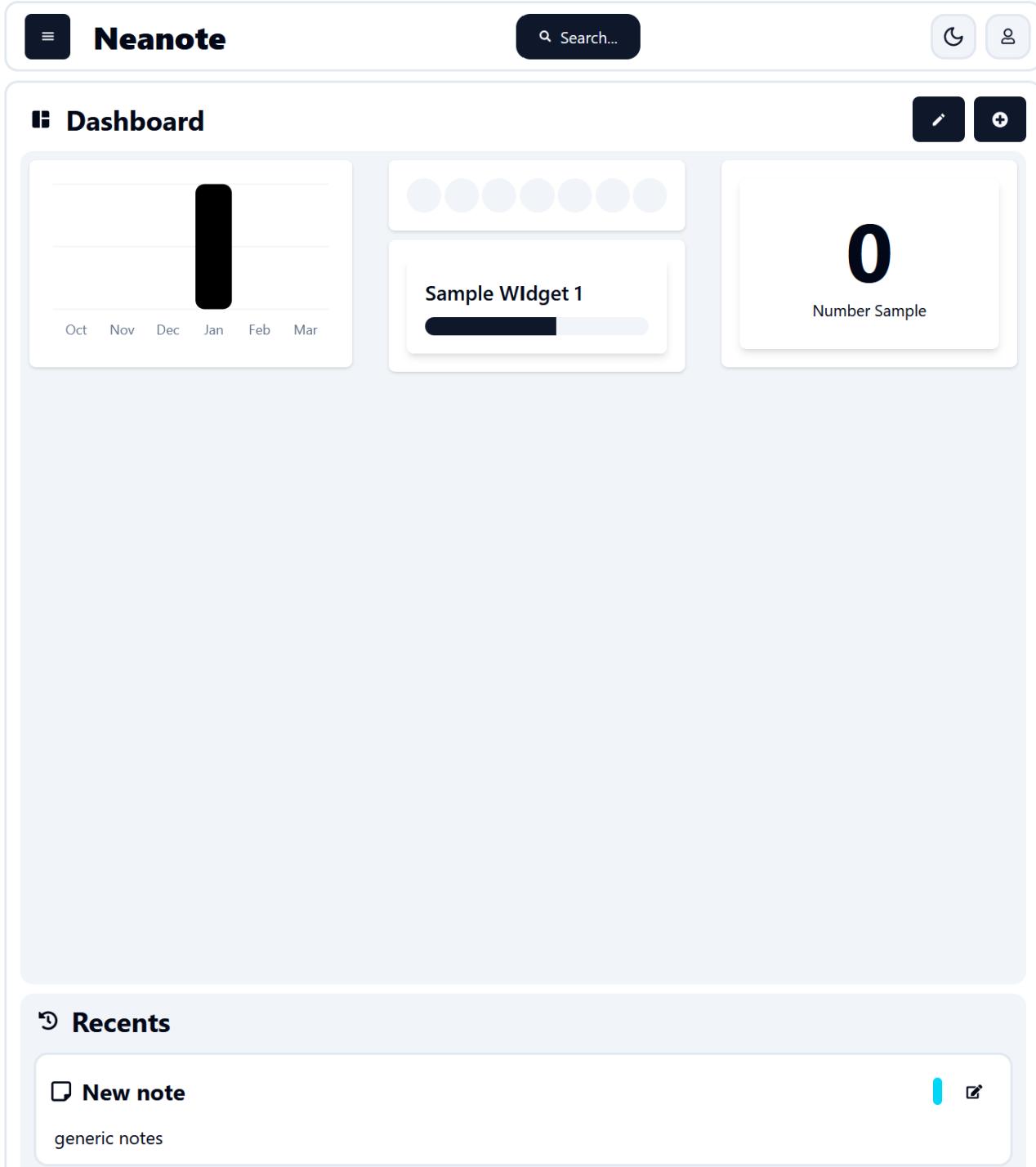
export default EditGoals;
```

To avoid unnecessary repetition, most buttons and inputs are combined into components like FormButtons and reused throughout the app. There is no need to go over them again.

The next two notetypes, habits and notes, contain similar code and adhere to the same design principles, use optimistic updates, Zustand, and other techniques, so I believe that it is unnecessary to go over the same code again.

### 3.2.5.8 Dashboard

#### 3.2.5.8.1 ./src/Pages/Dashboard/Dashboard.tsx



```
import React, { useEffect } from 'react';
import { FaHistory } from "react-icons/fa";
import { MdEdit, MdOutlineCheck, MdSpaceDashboard } from "react-icons/md";
import { useNavigate } from "react-router-dom";
import { Button } from '../../../../../components/@ui/button';
```

```

import { Label } from '../../../../../components/@/ui/label';
import TitleComponent from '../../../../../components/TitleComponent/TitleComponent';
import UniversalCard from '../../../../../components/Universal/UniversalCard';
import EditPicker from './Components/EditPicker';
import WidgetGrid from './Components/WidgetGrid';
import { useDashboard } from './useDashboard';

function Dashboard() {
  const {recents, getRecents, loading, editMode, setEditMode, fetchWidgets} =
  useDashboard()
  const navigate = useNavigate();

  useEffect(() => {
    getRecents();
  },[getRecents])

  function handleEditClick(noteId, type) {
    localStorage.setItem(`current${type.charAt(0).toUpperCase() + type.slice(1)}Id`, noteId.toString());
    navigate(`/${type + 's'}/edit`);
  }

  return (
    <>
    <div className="flex flex-row justify-between pb-2">

      <TitleComponent>
        <MdSpaceDashboard size={'20px'} /> Dashboard
      </TitleComponent>
      <div className="flex flex-row gap-2">
        <Button onClick={() => setEditMode(!editMode)}>
          {editMode ? <MdOutlineCheck /> : <MdEdit />}
        </Button>
        <EditPicker/>
      </div>
    </div>
    {/* <KanbanBoard/> */}
    <WidgetGrid/>
    <div className='bg-secondary rounded-xl p-2'>
      <div className="flex flex-row gap-3 items-center pb-2">
        <TitleComponent><FaHistory size={'17px'} /> Recents</TitleComponent>
      </div>
      <div className='flex flex-col flex-grow p-1 gap-2'>
        {loading && <p>Loading...</p>}
        {recents.length>0 &&
        <>
          <div className='flex-col flex gap-2'>
            {recents.map((item, index) => (
              <UniversalCard key={index} note={item}
handleEditClick={()=>handleEditClick(item.noteid, item.type)} />
            )))
          </div>
        </>
      }
    
```

```

        {recents.length == 0 && !loading && <Label>No recent notes
found.</Label>}
    </div>
</div>
</>
)
}

export default Dashboard

```

The Dashboard component contains two primary sections: the widgets and the recents section, which are conditionally rendered based on the loading variable.

### 3.2.5.8.2      [./src/Pages/Dashboard/EditDashboard.tsx](#)

The Widget grid works as any app layout grid on your mobile phone. This means that widgets can be moved up and down in their column, as well as being draggable between columns at the user's disposal. To avoid accidental dragging events, a specific edit mode is required to move the widgets around. To let all the widgets know that they are now draggable, a provider is used:

```

import React, { createContext, useContext, useState } from 'react';
import { WidgetT } from '../../api/types/widgetTypes';
import { Column } from './useDashboard';

interface DashboardContextType {
    columns: Column[];
    widgets: WidgetT[];
    editMode: boolean;
    setEditMode: (mode: boolean) => void;
    setColumns: (columns: Column[]) => void;
    setWidgets: (widgets: WidgetT[]) => void;
    addWidget: (widget: WidgetT) => void;
    removeWidget: (widgetId: string) => void;
    moveWidget: (widgetId: string, overId: string, columnId: string) => void;
}

export const DashboardContext = createContext<DashboardContextType | undefined>(undefined);

export const DashboardProvider: React.FC<{ children: React.ReactNode }> =
({ children }) => {
    const [columns, setColumns] = useState<Column[]>([]);
    const [widgets, setWidgets] = useState<WidgetT[]>([]);
    const [editMode, setEditMode] = useState(false);

    const addWidget = (widget: WidgetT) => {
        setWidgets(prev => [...prev, widget]);
    };

    const removeWidget = (widgetId: string) => {
        setWidgets(prev => prev.filter(w => w.id !== widgetId));
    };
}

```

```

const moveWidget = (widgetId: string, overId: string, columnIndex: string) => {
  setWidgets(prev => {
    const widgets = [...prev];
    const widgetIndex = widgets.findIndex(w => w.id === widgetId);
    const widget = widgets[widgetIndex];

    if (!widget) return prev;

    // Remove widget from current position
    widgets.splice(widgetIndex, 1);

    if (columnIndex) {
      // Moving to a new column
      widget.columnId = columnIndex;
      widgets.push(widget);
    } else if (overId) {
      // Moving within same column or between widgets
      const overIndex = widgets.findIndex(w => w.id === overId);
      widgets.splice(overIndex, 0, widget);
    }

    // Update order values
    return widgets.map((w, index) => ({
      ...w,
      order: index
    }));
  });
};

const value = {
  columns,
  widgets,
  editMode,
  setEditMode,
  setColumns,
  setWidgets,
  addWidget,
  removeWidget,
  moveWidget,
};

return (
  <DashboardContext.Provider value={value}>
    {children}
  </DashboardContext.Provider>
);
};

export const useDashboard = () => {
  const context = useContext(DashboardContext);
  if (context === undefined) {
    throw new Error('useDashboard must be used within a DashboardProvider');
  }
  return context;
}

```

```
};
```

This code establishes a dashboard context using interfaces, React's createContext and useContext hooks, along with the useState hook for managing state. The DashboardContext provides a way for components within the dashboard to access and modify dashboard-related data, such as columns and widgets. The DashboardProvider component wraps the application or a section of it, making the context available to its children. It initializes the columns and widgets state using useState, and also manages an editMode state. The provider component defines several functions: addWidget to add a new widget, removeWidget to delete a widget, and moveWidget to reorder widgets within or between columns. The moveWidget function demonstrates immutable state updates by creating copies of the state arrays, modifying them, and then setting the state with the modified copies. The useDashboard hook simplifies accessing the context, throwing an error if a component attempts to use it outside of a DashboardProvider.

### 3.2.5.8.3        [./src/Pages/Dashboard/Widgets/Widget.tsx](#)

Now that we have a way to communicate the state of the page to each widget, its time to render an appropriate widget based on the widget type. The Widget component conditionally renders a trash icon if the edit mode to allow the user to delete the widget. Additionally, a switch-case (alternative selection) is used to properly decide the widget to render based on the provided widget type.

```
import React from 'react';
import { FaTrash } from 'react-icons/fa';
import { Button } from '../../../../../components/@ui/button';
import { ChartWidget } from '../../../../../components/Widgets/Chart/ChartWidget';
import { ProgressWidget } from '../../../../../components/Widgets/ProgressWidget/ProgressWidget';
import NumberWidget from '../../../../../components/Widgets/Number/NumberWidget';
import HabitWeek from '../../../../../components/Widgets/HabitWeek/HabitWeek';

interface WidgetProps {
  id: string;
  data: any;
  type: string;
  title: string;
  editMode: boolean;
  onRemove: () => void;
  handleEditClick: (noteId: string, type: string) => void;
}

export const Widget: React.FC<WidgetProps> = ({  
  id,  
  data,  
  type,  
  title,  
  editMode,  
  onRemove,  
  handleEditClick  
) => {
```

```

// Render the appropriate widget based on type
const renderWidget = () => {
  switch (type) {
    case 'Chart':
      return (
        <ChartWidget
          label={title}
          color="primary"
          sample={false}
          data={data?.monthly_data || []}
        />
      );
    case 'Progress':
      const progressValue = data?.completed_milestones !== undefined &&
                           data?.total_milestones !== undefined &&
                           data.total_milestones > 0
                           ? (data.completed_milestones /
data.total_milestones) * 100
                           : 0;
      return (
        <ProgressWidget
          title={title}
          progress={progressValue}
        />
      );
    case 'Number':
      return (
        <NumberWidget
          caption={title}
          number={data?.value || 0}
        />
      );
    case 'HabitWeek':
      return (
        <HabitWeek title={title}
          data={data?.days || [false, false, false, false, false, false, false]}
        />
      );
    default:
      return <div>Unknown widget type: {type}</div>;
  }
};

return (
  <div className='relative min-h-50 bg-background rounded-md p-4 shadow'>
    {editMode && (
      <Button
        onClick={onRemove}
        className='absolute h-10 w-10 top-2 right-2 text-red-500'
        variant='ghost'
      >
        <FaTrash />
      </Button>
    )}
    {renderWidget()}
  </div>
);

```

```
        </div>
    );
};
```

### 3.2.5.8.4 ./src/Pages/Dashboard/Widgets/ WidgetGrid.tsx

The WidgetGrid component functions as a dynamic layout manager for interactive widgets, employing the dnd-kit library to facilitate drag-and-drop reordering within a grid structure. This component is designed to handle the complexities of arranging and manipulating user interface elements, specifically widgets, within a dashboard-like environment.

```
import {
  closestCenter,
  DndContext,
  DragOverlay,
  PointerSensor,
  useSensor,
  useSensors,
} from '@dnd-kit/core';
import {
  rectSortingStrategy,
  SortableContext
} from '@dnd-kit/sortable';
import React, { useEffect, useMemo, useState } from 'react';
import { createPortal } from 'react-dom';
import ColumnContainer from './ColumnContainer';

import { useScreenSize } from '../../../../../DisplayContext';
import { useDashboard } from '../useDashboard';
import { WidgetT } from '../../../../../api/types/widgetTypes';
import { Widget } from './Widget';

const WidgetGrid = () => {
  const {
    columns,
    widgets,
    addColumn,
    removeColumn,
    addWidget,
    removeWidget,
    moveWidget,
    setColumns,
    editMode,
    setEditMode,
    fetchWidgets
  } = useDashboard();

  const [activeWidget, setActiveWidget] = useState<WidgetT | null>(null);
  const { screenSize } = useScreenSize(); // Get screen size
```

```

useEffect(() => {
  fetchWidgets();
}, [fetchWidgets]
);

const sensors = useSensors(
  useSensor(PointerSensor, {
    activationConstraint: {
      distance: 5,
    },
  })
);

```

The PointerSensor, configured with an activation constraint, detects drag events. This sensor specifically monitors pointer events, such as mouse movements and touch gestures, to initiate drag-and-drop interactions. The activation constraint, prevents accidental drags by requiring the user to move the dragged element a minimum distance before the drag operation is considered active. This improves the user experience by reducing unintended behavior.

```

const columnsId = useMemo(() => columns.map((col) => col.id), [columns]);

useEffect(() => {
  const determineColumns = () => {
    let numColumns;
    if (screenSize === 'small') {
      numColumns = 1;
    } else if (screenSize === 'medium') {
      numColumns = 3;
    } else {
      numColumns = 6;
    }

    const initialColumns = Array.from({ length: numColumns }, (_, index) => ({
      id: `column-${index + 1}`,
      title: `Column ${index + 1}`,
    }));
    setColumns(initialColumns);
  };
  determineColumns();
}, [screenSize, setColumns]);

```

This component makes rendering and manipulation of these widgets possible, leveraging React hooks such as useEffect, useMemo, and useState for state management. useState is used for managing local component state, such as tracking the currently dragged widget. useEffect enables side effects, such as fetching widget data or adjusting the column layout based on screen size : 1 column for small displays, 3 for average and 6 for large. useMemo optimizes performance by memoizing calculated values, like the array of column IDs, preventing unnecessary recalculations.

```

const handleDragEnd = (event: any) => {
  const { active, over } = event;

  if (!over) return;

  const activeId = active.id;
  const overId = over.id;

  if (activeId === overId) return;

  const isActiveWidget = active.data.current?.type === 'Widget';
  const isOverWidget = over.data.current?.type === 'Widget';
  const isOverColumn = over.data.current?.type === 'Column';

  // Handle widget to widget movement
  if (isActiveWidget && isOverWidget) {
    moveWidget(activeId, overId, '');
  }

  // Handle widget to column movement
  if (isActiveWidget && isOverColumn) {
    moveWidget(activeId, '', overId);
  }

  setActiveWidget(null); // Clear active widget after drag ends
};

const handleDragStart = (event: any) => {
  const { active } = event;
  if (active.data.current?.type === 'Widget') {
    setActiveWidget(widgets.find(widget => widget.id === active.id) || null);
  }
};

```

Drag-and-drop interactions are enabled through the DndContext, SortableContext, and related sensor components from dnd-kit. The DndContext establishes the context for drag-and-drop operations, making the necessary functionality available to its descendants. SortableContext defines a container within which elements can be sorted, in this case, the columns.

Widget dragging is handled via handleDragStart and handleDragEnd, where the latter uses the moveWidget function from the Dashboard provider to update the widget's position. The handleDragStart function is called when a drag operation begins. The handleDragEnd determines the new position of the dragged widget and uses the moveWidget function to update the application's state, reflecting the widget's new location within the grid.

```

return (
  <>
  <DndContext
    sensors={sensors}

```

```

        collisionDetection={closestCenter}
        onDragEnd={handleDragEnd}
        onDragStart={handleDragStart}
    >
    <div className='flex bg-secondary rounded-xl mb-2 gap-4'>
        <SortableContext items={columnsId} strategy={rectSortingStrategy}>
            {columns.map((column) => (
                <ColumnContainer
                    key={column.id}
                    column={column}
                    widgets={widgets.filter((widget) => widget.columnId === column.id)}
                    removeWidget={removeWidget}
                    editMode={editMode}
                    removeColumn={removeColumn}
                />
            )))
        </SortableContext>
    </div>
    {createPortal(
        <DragOverlay>
            {activeWidget && (
                <div className='relative bg-background rounded-md p-4 shadow'>
                    <Widget
                        id={activeWidget.id}
                        data={activeWidget.content}
                        type={activeWidget.type}
                        title={activeWidget.title}
                        editMode={false}
                        onRemove={() => {}}
                        handleEditClick={() => {}}
                    />
                </div>
            )}
        </DragOverlay>,
        document.body
    )}
    </DndContext>
</>
);
};

export default WidgetGrid;

```

### **3.2.5.8.5       ./src/Pages/Dashboard/Widgets/ EditPicker.tsx**

The user has to take a few steps to add a new widget.

```

import React from 'react';
import { Button } from '../../../../../components/@/ui/button';
import { useDashboard } from '../useDashboard';
import { FaPlusCircle } from 'react-icons/fa';
import {
    Drawer,

```

```

DrawerContent,
DrawerDescription,
DrawerHeader,
DrawerTitle,
DrawerTrigger,
} from "../../../../../components/@ui/drawer";
import { ChartWidget } from "../../../../../components/Widgets/Chart/ChartWidget";
import { ProgressWidget } from
'../../../../../components/Widgets/ProgressWidget/ProgressWidget';
import NumberWidget from '../../../../../components/Widgets/Number/NumberWidget';
import WidgetPreviewContainer from
'../../../../../components/Widgets/WidgetPreviewContainer';
import { WidgetSetup } from './WidgetSetup';
import HabitWeek from '../../../../../components/Widgets/HabitWeek/HabitWeek';
import widgetsApi from '../../../../../api/widgetsApi';

function EditPicker() {
  const {
    columns,
    addWidget,
    selectedWidgetType,
    setSelectedWidgetType,
    setWidgetConfig,
  } = useDashboard();

  const handleWidgetClick = (widgetType: 'Chart' | 'Number' | 'Progress' |
'HabitWeek') => {
    setSelectedWidgetType(widgetType); // Show setup screen
    setWidgetConfig({ title: '', dataSource: '' }); // Reset widget config
};

const handleSave = async (widgetData: any) => {
  try {
    // Create the widget in the backend
    const response = await widgetsApi.createUserWidget(widgetData);
    if (response.success) {
      // Add widget to the grid
      const columnId = columns[0]?.id || 'column-1';
      addWidget(
        columnId,
        response.data.id,
        widgetData.widget_id,
        widgetData.configuration.title,
        widgetData.data_source_type,
        widgetData.data_source_id,
        response.data.source_data // Pass the source_data as content
      );
      setSelectedWidgetType(null);
    }
  } catch (error) {
    console.error('Failed to create widget:', error);
  }
};

const handleCancel = () => {

```

```

// Return to widget selection
setSelectedWidgetType(null);
};

return (
<Drawer>
  <DrawerTrigger asChild>
    <Button>
      <FaPlusCircle />
    </Button>
  </DrawerTrigger>
  <DrawerContent>
    <DrawerHeader>
      <DrawerTitle className='flex flex-row justify-between'>Add
      widgets</DrawerTitle>
      <DrawerDescription>Select a widget type and source</DrawerDescription>
    </DrawerHeader>

    <div className="mx-auto w-full min-w-full pb-4 px-4">
      {selectedWidgetType === null ? (
        // 1) WIDGET SELECTION GRID
        <div className='grid grid-cols-1 sm:grid-cols-1 md:grid-cols-3 lg:grid-
        cols-4 bg-secondary rounded-xl min-w-full min-h-[40vh]'>
          <WidgetPreviewContainer
            title={'Chart'}
            description='Visualise your task completion rates.'
            type='Chart'
            onClick={handleWidgetClick}
          >
            <ChartWidget label='Sample' color='destructive' sample={true}-
            data={null}/>
          </WidgetPreviewContainer>

          <WidgetPreviewContainer
            title={'Progress'}
            description='Track your progress.'
            type='Progress'
            onClick={handleWidgetClick}
          >
            <ProgressWidget title='Investments' progress={75} />
          </WidgetPreviewContainer>

          <WidgetPreviewContainer
            title={'Number'}
            description='Display your streaks and averages.'
            type='Number'
            onClick={handleWidgetClick}
          >
            <NumberWidget caption={'Habit streak'} number={25}/>
          </WidgetPreviewContainer>

          <WidgetPreviewContainer
            title={'Habit Week'}
            description='View your weekly habit progress.'
            type='HabitWeek'
          >
        </div>
      ) : (
        // 2) PREVIEW GRID
        <div className='grid grid-cols-1 sm:grid-cols-1 md:grid-cols-3 lg:grid-
        cols-4 bg-secondary rounded-xl min-w-full min-h-[40vh]'>
          <WidgetPreviewContainer
            title={'Chart'}
            description='Visualise your task completion rates.'
            type='Chart'
            onClick={handleWidgetClick}
          >
            <ChartWidget label='Sample' color='destructive' sample={true}-
            data={null}/>
          </WidgetPreviewContainer>

          <WidgetPreviewContainer
            title={'Progress'}
            description='Track your progress.'
            type='Progress'
            onClick={handleWidgetClick}
          >
            <ProgressWidget title='Investments' progress={75} />
          </WidgetPreviewContainer>

          <WidgetPreviewContainer
            title={'Number'}
            description='Display your streaks and averages.'
            type='Number'
            onClick={handleWidgetClick}
          >
            <NumberWidget caption={'Habit streak'} number={25}/>
          </WidgetPreviewContainer>

          <WidgetPreviewContainer
            title={'Habit Week'}
            description='View your weekly habit progress.'
            type='HabitWeek'
          >
        </div>
      )
    </div>
  </DrawerContent>

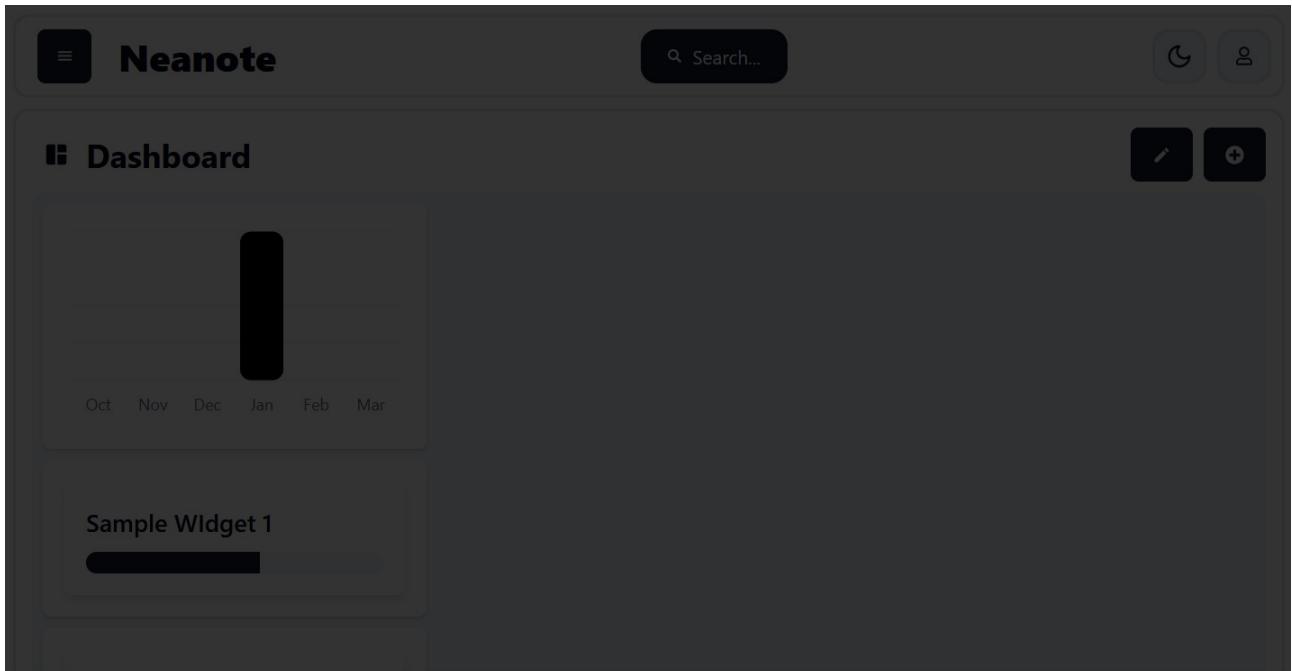
```

```
        onClick={handleWidgetClick}
      >
      <HabitWeek data={[true,true,false,true,true,true,false]}/>
    </WidgetPreviewContainer>

    {/* Add more preview widgets here if needed */}
  </div>
) : (
  // 2) SETUP SCREEN
  <WidgetSetup
    widgetType={selectedWidgetType}
    onSave={handleSave}
    onCancel={handleCancel}
  />
)
)
</div>
</DrawerContent>
</Drawer>
);
}

export default EditPicker;
```

It conditionally renders a drawer with two slements: a gallery with example widgets and a setup screen.



### Add widgets

Select a widget type and source

**Chart**  
Visualise your task completion rates.

A bar chart with six bars representing the months from January to June. The bars are black and vary in height. The x-axis is labeled with the months: Jan, Feb, Mar, Apr, May, and Jun. The bars for Jan, Mar, and May are significantly taller than the others.

Month	Completion Rate (approx.)
Jan	85%
Feb	10%
Mar	90%
Apr	5%
May	75%
Jun	15%

**Progress**  
Track your progress.

**Investments**

A horizontal progress bar with a dark blue segment and a light grey remainder. The text "Investments" is displayed above the bar.

**Number**  
Display your streaks and averages.

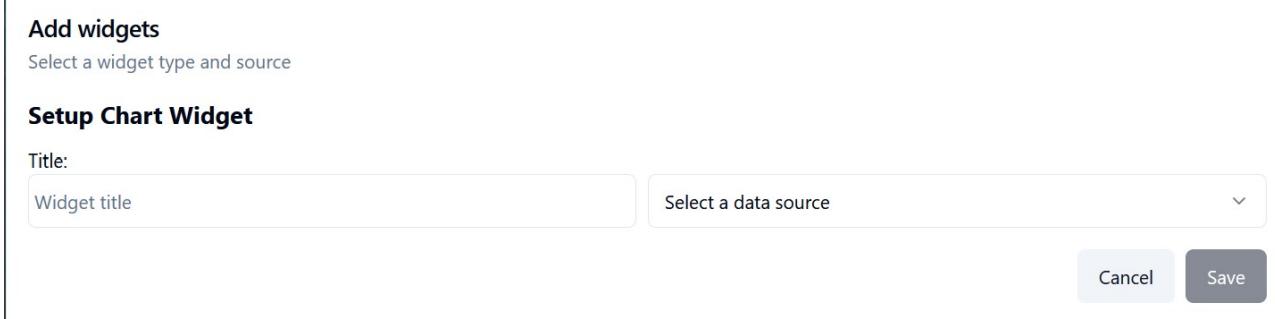
**25**

Habit streak

**Habit Week**  
View your weekly habit progress.

A horizontal progress bar consisting of seven circular segments. The first four segments are dark, and the last three are light, representing a weekly habit streak.

Once the user chooses the widget, they click on it and are prompted to name the widget and select the datasource from a list, which dynamically fetches notes that can be used as datasources for this specific widget:



Widgets rendered in the drawer have sample data pre-written into them to provide a visual representation of the widget the user is about to click. Additionally, an `async handleSave` function is written in the component to directly run a `CREATE` request through the API

### 3.2.5.8.6     ./src/Pages/Dashboard/Widgets/ ChartWidget.tsx

```
import React from "react"
import { Bar, BarChart, CartesianGrid, XAxis } from "recharts"

import {
  ChartConfig,
  ChartContainer,
  ChartTooltip,
  ChartTooltipContent,
} from "../../@/ui/chart"
import { chartData } from "../sample_data"

interface ChartWidgetProps {
  label : string,
  color : string,
  sample : boolean,
  data : any
}

export function ChartWidget({label, color, sample, data}: ChartWidgetProps) {

  const chartConfig = {
    completed: {
      label: label,
    },
  } satisfies ChartConfig
  // Resolve theme-dependent color
  const resolvedColor =
getComputedStyle(document.documentElement).getPropertyValue(`--color-$
{color}`).trim() || color;

  return (
    <ChartContainer config={chartConfig} className="min-h-[150px] w-full">
      <BarChart accessibilityLayer data={sample ? chartData : data}>
```

```

        <CartesianGrid vertical={false} />
        <XAxis
            dataKey="month"
            tickLine={false}
            tickMargin={8}
            axisLine={false}
            tickFormatter={(value) => value.slice(0, 3)}
        />
        <ChartTooltip content={<ChartTooltipContent />} />
        <Bar dataKey="completed" fill={resolvedColor} radius={8} />
    </BarChart>
</ChartContainer>
)
}

```

Here is the chart widgets. This component uses the recharts library to render charts based on the provided data. The widget uses JavaScript to get the current style of the page and adjust the color of the bars on the chart in accordance with the theme of the whole page.

### 3.2.5.8.7        [./src/Pages/Dashboard/useDashboard.tsx](#)

This custom hook uses Zustand and Immer to manage the state of the dashboard page. It stores several main state variables, such as the list of recent notes, as well as widgets and widget columns and defines logical operations that need to be done with the state to reflect the user moving a widget from one column to another in the edit mode. It applies the same principles and techniques as other components.

```

import { create } from "zustand";
import { immer } from "zustand/middleware/immer";
import { universalApi } from "../../api/universalApi";
import { UniversalType } from "../../api/types/ArchiveTypes";
import { showToast } from "../../components/Toast";
import { arrayMove } from "@dnd-kit/sortable";
import { WidgetT, WidgetType, DataSourceType } from "../../api/types/widgetTypes";
import widgetsApi from "../../api/widgetsApi";

export interface Column {
    id: string;
    title: string;
}

interface DashboardState {
    getRecents: () => void;
    recents: UniversalType[];
    loading: boolean;
    columns: Column[];
    widgets: WidgetT[];
    setWidgets: (widgets: WidgetT[]) => void;
    editMode: boolean;
    selectedWidgetType: WidgetType | null;
    widgetConfig: { title: string; dataSource: string } | null;
    addColumn: () => void;
    removeColumn: (id: string) => void;
    addWidget: (

```

```

        columnId: string,
        widgetId: string,
        type: WidgetType,
        title: string,
        dataSourceType: DataSourceType,
        dataSourceId?: string,
        content?: any
    ) => void;
    removeWidget: (id: string) => void;
    moveWidget: (activeId: string, overId: string, overColumnId: string) => void;
    setColumns: (columns: Column[]) => void;
    setEditMode: (editMode: boolean) => void;
    setSelectedWidgetType: (widgetType: WidgetType | null) => void;
    setWidgetConfig: (config: { title: string; dataSource: string } | null) => void;
    fetchWidgets: () => void;
}

export const useDashboard = create<DashboardState>()(

    immer((set, get) => ({
        recents: [],
        loading: false,
        columns: [],
        widgets: [],
        selectedWidgetType: null,
        widgetConfig: null,
        editMode: false,
        getRecents: async () => {
            const response = await universalApi.getRecents();
            if (response && response.success) {
                set((state) => {
                    state.recents = response.data.data;
                });
            } else {
                showToast('error', 'An error occurred while fetching recently accessed notes.');
            }
        },
        setWidgets: (widgets: WidgetT[]) => {
            set((state) => {
                state.widgets = widgets;
            });
        },
        addColumn: () => {
            const newColumnId = `column-${Date.now()}`;
            const newColumn: Column = {
                id: newColumnId,
                title: `Column ${get().columns.length + 1}`,
            };
            set((state) => {
                state.columns.push(newColumn);
            });
        },
    }),

```

The `addColumn` function adds a new column to the dashboard layout. It generates a unique ID for the new column using `Date.now()` and creates a `Column` object with this ID and a default title. The new column is then added to the `columns` array in the store's state, effectively expanding the dashboard's layout.

```
setEditMode: (editMode: boolean) => {
  set((state) => {
    state.editMode = editMode;
  });
},
removeColumn: (id: string) => {
  set((state) => {
    state.columns = state.columns.filter((column) => column.id !== id);
    state.widgets = state.widgets.filter((widget) => widget.columnId !== id);
  });
},
addWidget: (columnId, widgetId, type, title, dataSourceType, dataSourceId,
content = {}) => {
  set((state) => {
    state.widgets.push({
      id: widgetId,
      columnId,
      type,
      title,
      content,
      dataSourceType,
      dataSourceId,
      order: state.widgets.length,
    });
  });
},
removeWidget: async (id: string) => {
  try {
    const response = await widgetsApi.deleteUserWidget(id);
    if (response.success) {
      set((state) => {
        state.widgets = state.widgets.filter((widget) => widget.id !== id);
      });
      showToast('success', 'Widget removed successfully');
    } else {
      showToast('error', 'Failed to remove widget');
    }
  } catch (error) {
    showToast('error', 'An error occurred while removing the widget');
  }
},
moveWidget: (activeId: string, overId: string, overColumnId: string) => {
  set((state) => {
    const activeIndex = state.widgets.findIndex((widget) => widget.id === activeId);
    const overIndex = state.widgets.findIndex((widget) => widget.id === overId);

    if (activeIndex === -1) return;
  });
}
```

```
        if (overIndex === -1) {
            state.widgets[activeIndex].columnId = overColumnId;
        } else {
            state.widgets[activeIndex].columnId = state.widgets[overIndex].columnId;
            state.widgets = arrayMove(state.widgets, activeIndex, overIndex);
        }
    });
},
},
```

The `moveWidget` function repositions a widget within the dashboard, either within the same column or to a different column. It uses `arrayMove` from the `dnd-kit` library to adjust the order of widgets within the `widgets` array. The function also updates the `columnId` of the moved widget, ensuring that it is displayed in the correct column.

```
setColumns: (columns: Column[]) => {
  set((state) => {
    state.columns = columns;
  });
},
setSelectedWidgetType: (widgetType: WidgetType | null) => {
  set((state) => {
    state.selectedWidgetType = widgetType;
  });
},
const response = await widgetsApi.getUserWidgets();
if (response.success && response.data) {
  set((state) => {
    state.widgets = response.data.map((widget) => ({
      id: widget.id,
      columnId: widget.configuration?.position?.x || "column-1", // Use
      default column
      type: widget.widget_id,
      title: widget.title || widget.configuration?.title,
      content: widget.source_data, // Use the source_data as the content
      dataSourceType: widget.data_source_type,
      dataSourceId: widget.data_source_id,
      order: widget.configuration?.position?.y || state.widgets.length,
    }));
  });
} else {
  showToast('error', 'An error occurred while fetching widgets');
}
}
```

The `fetchWidgets` function retrieves user-specific widgets from the backend using `widgetsApi.getUserWidgets()`. It then transforms the received data into an appropriate format and updates the widgets state, populating the dashboard with the user's saved widgets. Default column and order values are used if they are not provided by the backend.

```
setWidgetConfig: (config: { title: string; dataSource: string } | null) => {
  set((state) => {
    state.widgetConfig = config;
  });
},
})),
);

```

### 3.2.5.9 Archive

**Neanote**

Search...

Archive

Sample Goal 1

Task sample

Lorem ipsum dolor sit amet.

Archived notes will be removed after 30 days if not in use

< 1 >

The archive section may contain archived notes of all types. The notes can be restored or deleted within a 30 day period, before being deleted automatically. The user can distinguish notes by type using an appropriate icon, which is associated with a specific note type.

### 3.2.5.9.1        [./src/Pages/Archive/Components/ArchiveCard.tsx](#)

```
import React from 'react'  
import { UniversalType } from '../../../../../api/types/ArchiveTypes'
```

260, Roman Sasinovich, 74561 IMS

```

import { FaTasks } from 'react-icons/fa'
import { MdRepeat } from 'react-icons/md'
import { LuGoal } from 'react-icons/lu'
import { Button } from '../../../../../components/@ui/button'
import { MdOutlineRestore } from "react-icons/md";
import { FaRegNoteSticky, FaTrash } from 'react-icons/fa6'
import DeleteDialog from '../../../../../components/DeleteDialog/DeleteDialog'
import { useArchive } from '../useArchive'
import TagLabel from '../../../../../components/TagLabel/TagLabel'

function ArchiveCard({note}:{note:UniversalType}) {
    const {handleDelete, handleRestore} = useArchive();

    const onDelete = async () => {
        await handleDelete(note.type, note.noteid, note.secondaryid);
    }
    const onRestore = async () => {
        await handleRestore(note.noteid);
    }

    return (
        <div className='p-3 w-full rounded-xl border-[2px]'>
            <div className='flex flex-row justify-between'>
                <div className='flex flex-row items-center gap-2' >
                    {note.type === 'note' && <FaRegNoteSticky size={'20px'} />}
                    {note.type === 'task' && <FaTasks size={'20px'} />}
                    {note.type === 'habit' && <MdRepeat size={'20px'} />}
                    {note.type === 'goal' && <LuGoal size={'22px'} />}
                </div>
                <h3 className="task-title">{note.title}</h3>
            </div>
            <div className='flex flex-row items-center gap-2' >
                {note.tags.map((tag, index) => (
                    <TagLabel key={index} name={tag.name} color={tag.color}
                    compressed={true}/>
                ))}
                <DeleteDialog handleDelete={onDelete}>
                    <Button size="icon" variant="destructive">
                        <FaTrash />
                    </Button>
                </DeleteDialog>

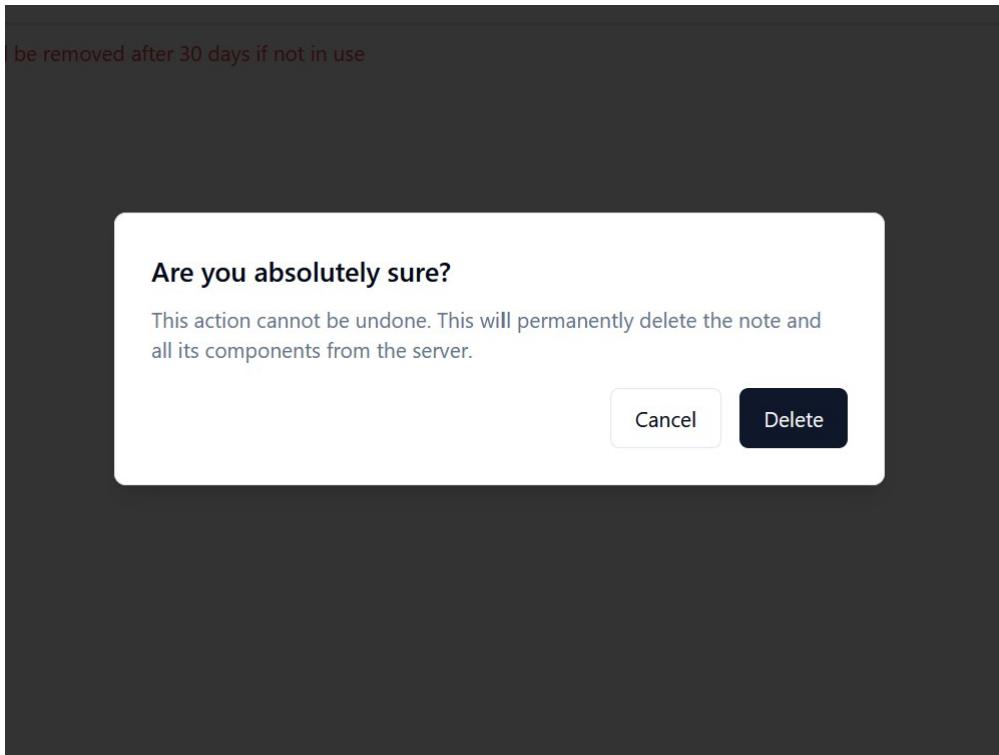
                <Button onClick={onRestore} size="icon">
                    <MdOutlineRestore size={'20px'} />
                </Button>
            </div>
        </div>
        {note.content && <p className="text-md pl-1 pt-2">{note.content}</p>}
    </div>
)
}

export default ArchiveCard

```

This is the component that is mapped across the Archive page and renders each note. It is passed a note that was fetched after the /api/archive GET request and conditionally renders an appropriate icon based on the note type. Restore and delete buttons are also created and the Delete button is actually passed as children into a dialog. The component only renders the note title, content and tags, because these properties are present on all notetypes.

### 3.2.5.9.2        ./src/components/DeleteDialog/ DeleteDialog.tsx



```
import React, { ReactElement, ReactNode } from 'react'
import { AlertDialog, AlertDialogTrigger, AlertDialogContent, AlertDialogHeader,
AlertDialogTitle, AlertDialogDescription, AlertDialogFooter, AlertDialogCancel,
AlertDialogAction } from '../@/ui/alert-dialog'
import { Button } from '../@/ui/button'

interface Props {
  children: ReactNode,
  handleDelete: () => void,
  handleArchive?: () => void
}

function DeleteDialog({children, handleDelete, handleArchive}:Props)  {
  return (
    <AlertDialog>
      <AlertDialogTrigger asChild>
        {children}
      </AlertDialogTrigger>
```

```

        <AlertDialogContent>
            <AlertDialogHeader>
                <AlertDialogTitle>Are you absolutely sure?</AlertDialogTitle>
                <AlertDialogDescription>
                    This action cannot be undone. This will permanently delete the note
                    and all its components from the server. {handleArchive && 'Maybe archive
                    instead?' }
                </AlertDialogDescription>
                </AlertDialogHeader>
                <AlertDialogFooter>
                    <div className='flex flex-row gap-3'>
                        <AlertDialogCancel>Cancel</AlertDialogCancel>
                        { handleArchive &&
                            <AlertDialogAction onClick={handleArchive}>Archive
                            instead</AlertDialogAction>
                        }
                        <AlertDialogAction
                            onClick={handleDelete}>Delete</AlertDialogAction>
                    </div>
                </AlertDialogFooter>
            </AlertDialogContent>
        </AlertDialog>
    )
}

```

export default DeleteDialog

This dialog takes in a deletion function and sometimes an archive function. This is because the dialog also conditionally renders an “Archive instead” button, which sends the note to the archive. But in this specific case, the user can only delete the note, since it is already archived.

### 3.2.5.9.3        [./src/Pages/Archive/Archive.tsx](#)

```

import React, { useEffect } from 'react'
import PageContainer from '../../../../../components/PageContainer/PageContainer'
import { useArchive } from './useArchive';
import ArchiveCard from './Components/ArchiveCard';
import { FaArchive } from 'react-icons/fa';
import TitleComponent from '../../../../../components/TitleComponent/TitleComponent';
import PaginationSelector from '../../../../../components/Pagination/PaginationSelector';

function Archive() {
    const { fetchArchivedNotes, nextPage, archive, page } = useArchive();

    useEffect(() => {
        fetchArchivedNotes(page);
    }, [fetchArchivedNotes])

    return (
        <>
            <div className="flex flex-row gap-3 items-center pb-2">
                <TitleComponent><FaArchive size={'20px'} /> Archive</TitleComponent>

```

```

        </div>
        <div className='flex flex-col flex-grow gap-2'>
        { archive.map((note) => (
            <ArchiveCard key={note.noteid} note={note}/>
        )))
        <p className='pl-1 text-destructive text-sm ml-1'>Archived notes will be
removed after 30 days if not in use</p>
        </div>
        <div className="p-1 pt-2">
            <PaginationSelector fetchingFunction={fetchArchivedNotes}>
nextPage={nextPage} page={page}</PaginationSelector>
        </div>
    </>
)
}
}

export default Archive

```

This page calls the fetching function upon mount and displays the ArchiveCard components with retrieved data.

### **3.2.5.9.4        ./src/Pages/Archive/useArchive.tsx**

As always, this hook defines datatypes and creates default variable values with Zustand and Immer

```

import { create } from "zustand"
import { immer } from "zustand/middleware/immer"
import { UniversalType } from "../../api/types/ArchiveTypes"
import archiveApi from "../../api/archiveApi"
import { showToast } from "../../../../components/Toast"
import { UUID } from "crypto"
import habitsApi from "../../../../api/habitsApi"
import tasksApi from "../../../../api/tasksApi"
import goalsApi from "../../../../api/goalsApi"
import notesApi from "../../../../api/notesApi"

type ArchiveState = {
    archive: UniversalType[]

    handleDelete: (noteType:string,noteId:UUID,secondaryId:UUID) => Promise<void>
    handleRestore: (noteId:UUID) => Promise<void>
    fetchArchivedNotes: (pageParam:number) => void
    nextPage: number | null
    page:number

    loading:boolean
}

export const useArchive = create<ArchiveState>()(

    immer((set, get) => ({
        archive: [],

```

```

nextPage:null,
page:1,

loading:false,

fetchArchivedNotes: async (pageParam: number) => {
    set({ loading: true });
    try {
        const response = await archiveApi.getAll(pageParam);
        if (response.success) {
            set({ archive: response.data, nextPage: response.nextPage,
page: response.page });
        } else {
            showToast('error', response.message);
        }
    } finally {
        set({ loading: false });
    }
},
handleDelete: async (noteType: string, noteId: UUID, secondaryId: UUID) =>
{
    try {
        let response = { success: false };

        switch (noteType) {
            case 'habit':
                response = await habitsApi.delete(secondaryId, noteId);
                break;
            case 'task':
                response = await tasksApi.delete(noteId, secondaryId);
                break;
            case 'note':
                response = await notesApi.delete(noteId);
                break;
            case 'goal':
                response = await goalsApi.delete(noteId, secondaryId);
                break;
            default:
                showToast('e', 'Invalid note type');
                return;
        }

        if (response.success) {
            showToast('s', `${noteType.charAt(0).toUpperCase() + noteType.slice(1)} has been deleted successfully`);
            set((state) => {
                state.archive = state.archive.filter((note) =>
note.noteid !== noteId);
            });
        } else {
            showToast('e', `Failed to delete ${noteType}`);
        }
    } catch (error) {

```

```
        showToast('e', `An error occurred while deleting the ${noteType}: $ {error.message || error}`);
    }
},
),

```

In this function, switch-case is used again to call different apis based on the note type provided, as different note types are spread across different tables.

```
handleRestore: async (noteId: UUID) => {
    const response = await archiveApi.restore(noteId);
    if (response.success) {
        showToast('s', 'Restored successfully');
        set((state) => {
            state.archive = state.archive.filter((note) => note.noteid !== noteId);
        });
    } else {
        showToast('e', response.message);
    }
}
}))
```

### 3.2.5.10 Calendar

The screenshot shows the Neanote application's Calendar page for March 2025. The interface includes a header with the Neanote logo, a search bar, and navigation buttons for 'Previous', 'March 2025', and 'Next'. The main area is a 6x5 grid representing the months of March. Each cell contains a date. On March 23rd, there is a note titled 'Sample Goal 1' with an edit icon. On March 26th, there is a note titled 'Task sample' with an edit icon. The rest of the cells are empty.

25	26	27	28	1
2	3	4	5	6
7	8	9	10	11
12	13	14	15	16
17	18	19	20	21
22	23	24	25	26
27	28	29	30	31

Another way to connect notes from different note types is the Calendar page. Here, all notes with a specific due date are shown. The user can navigate the calendar back and forth. Each note is clickable and a click would send the user to an appropriate note editing page.

#### 3.2.5.10.1 ./src/Pages/Calendar/Calendar.tsx

```
import React, { useEffect, useState } from 'react'  
import TitleComponent from '../../../../../components/TitleComponent/TitleComponent'
```

```

import { FaRegCalendar } from 'react-icons/fa'
import { Button } from '../../../../../components/@ui/button'
import DayCard from './DayCard';
import { useCalendar } from './useCalendar';
import { useNavigate } from 'react-router-dom';

const Calendar = () => {
  const { currentDate, daysInMonth, handlePrevMonth, handleNextMonth,
  handleDateClick, fetchNotes, notes } = useCalendar();

```

After all the imports, values and functions of the useCalendar hook are destructured

```

useEffect(() => {
  const startDate = new Date(currentDate.getFullYear(), currentDate.getMonth(),
1);
  const endDate = new Date(currentDate.getFullYear(), currentDate.getMonth() + 1,
0);
  fetchNotes(startDate, endDate);
}, [currentDate, fetchNotes]);

```

This effect runs whenever currentDate or fetchNotes changes. It calculates the start and end dates of the currently displayed month, then calls fetchNotes to retrieve notes for that month.

```

const renderDays = (): JSX.Element[] => {
  const year = currentDate.getFullYear();
  const month = currentDate.getMonth();
  const daysInCurrentMonth = daysInMonth(month, year);
  const daysInPrevMonth = daysInMonth(month - 1, year);
  const totalDays = 35;
  const daysNeededFromPrevMonth = totalDays - daysInCurrentMonth;

  const days: JSX.Element[] = [];

  // Helper function to get notes for a specific day
  const getNotesForDay = (day: number, month: number, year: number) => {
    return notes.filter(note => {
      const noteDate = new Date(note.due_date);
      return noteDate.getDate() === day && noteDate.getMonth() === month &&
noteDate.getFullYear() === year;
    });
  };

  // Days from previous month
  for (let i = daysNeededFromPrevMonth; i > 0; i--) {
    const day = daysInPrevMonth - i + 1;
    const dayNotes = getNotesForDay(day, month - 1, year);
  }
}

```

```

    days.push(
      <DayCard
        key={`prev-${day}`}
        day={day}
        year={year}
        month={month - 1}
        handleDateClick={handleDateClick}
        notes={dayNotes}
        secondary
      />
    );
}

// Days in current month
for (let day = 1; day <= daysInCurrentMonth; day++) {
  const dayNotes = getNotesForDay(day, month, year);
  days.push(
    <DayCard
      key={`current-${day}`}
      day={day}
      year={year}
      month={month}
      handleDateClick={handleDateClick}
      notes={dayNotes}
    />
  );
}

return days;
};

```

The renderDays function is responsible for generating the visual representation of the calendar's days within the Calendar component. It calculates which days to display, including days from the previous and current months, and creates an array of DayCard components, each representing a single day in the calendar.

Firstly, retrieves the year and month from the currentDate state variable, which is managed by the useCalendar hook. It calculates the number of days in the current month and the previous month using the daysInMonth function. The total days are 35, which assumes a 5-week calendar view (7 days a week x 5 weeks). Then, days required to be rendered from the previous month are calculated by subtracting the two variables.

getNotesForDay is a nested function, which retrieves any notes associated with a specific day. It takes the day, month, and year as input. It filters the notes array (which contains all notes for the displayed month) to find notes whose due dates match the provided day, month, and year.

Then, a for loop iterates to generate DayCard components for the days from the previous month that are needed to fill the beginning of the calendar grid. The loop starts from the last day needed from the previous month and counts down to the first day needed.

Inside, the day variable is calculated and getNotesForDay is called to get the notes for that specific day in the previous month. A DayCard component is created with a unique key (using prev-{\$day}) and is pushed into the days array. The function returns the days array, which now contains all the DayCard components needed to render the calendar view for the current month, including the trailing days from the previous month.

```

return (
  <>
  <div className='flex flex-row gap-2 mb-2 items-center justify-between'>
    <TitleComponent>
      <FaRegCalendar size={'18px'} /> Calendar
    </TitleComponent>
    <div className='flex flex-row gap-2 items-center justify-center'>
      <Button onClick={handlePrevMonth}>Previous</Button>
      <span className='flex bg-primary h-10 rounded-md p-2 text-sm text-secondary items-center justify-center'>
        {currentDate.toLocaleString('default', { month: 'long' })}
      </span>
      <Button onClick={handleNextMonth}>Next</Button>
    </div>
  </div>
  <div className="grid grid-cols-4 sm:grid-cols-4 md:grid-cols-5">
    {renderDays()}
  </div>
</>
);
};

export default Calendar;

```

Finally, the Days array is rendered on the webpage.

### **3.2.5.10.2      ./src/Pages/Calendar/useCalendar.tsx**

```

import { create } from "zustand";
import { immer } from "zustand/middleware/immer";
import { UniversalTypeWithDate } from "../../api/types/ArchiveTypes";
import { universalApi } from "../../api/universalApi";

type CalendarState = {
  currentDate: Date;
  selectedDate: Date | null;
  notes: UniversalTypeWithDate[];
  daysInMonth: (month: number, year: number) => number;
  handlePrevMonth: () => void;
  handleNextMonth: () => void;
  handleDateClick: (date: Date) => void;
  fetchNotes: (startDate: Date, endDate: Date) => void;
};

export const useCalendar = create<CalendarState>()(

  immer((set, get) => ({

```

```

    currentDate: new Date(),
    selectedDate: null,
    notes: [],

    daysInMonth: (month: number, year: number) => {
        return new Date(year, month + 1, 0).getDate();
    },

```

This function calculates the number of days in a given month and year. By creating a new Date object with the year, month + 1, and day 0. In JavaScript, the day 0 of a month refers to the last day of the previous month. It then calls getDate() on this date object, which returns the day of the month.

```

    handlePrevMonth: () => {
        const { currentDate } = get();
        set((state) => {
            state.currentDate = new Date(currentDate.getFullYear(),
currentDate.getMonth() - 1, 1);
        });
    },
    handleNextMonth: () => {
        const { currentDate } = get();
        set((state) => {
            state.currentDate = new Date(currentDate.getFullYear(),
currentDate.getMonth() + 1, 1);
        });
    },

```

These two functions use the set function to update the state, creating a new Date object for the next or previous months.

```

    fetchNotes: async (startDate: Date, endDate: Date) => {
        try {
            const response = await
universalApi.getDue(startDate.toISOString(), endDate.toISOString());
            if (response) {
                set((state) => {
                    state.notes = response.data.data;
                });
            }
        } catch (error) {
            console.error("Failed to fetch notes:", error);
        }
    },
});

```

Lasty, the fetchNotes function takes the start and end dates converted from JavaScript Date objects to an ISO string format and calls the API.

### 3.2.5.11 Miscellaneous

#### 3.2.5.11.1 ./src/components/NavBar/NavBar.tsx



On large displays, the navbar contains the trigger to open a sidebar, the logo of the project, which also acts as a link to the dashboard, the searchbar trigger, the theme switcher and the account menu.

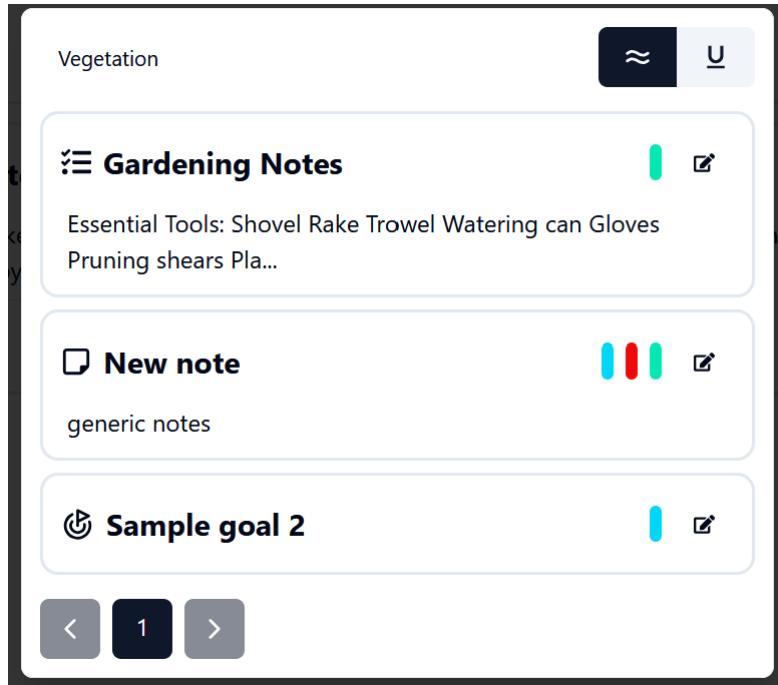
```
import React, { useState } from 'react';
import ThemeSwitcher from './ThemeSwitcher/ThemeSwitcher';
import LoginButton from './LoginButton/LoginButton';
import Sidebar from '../Sidebar/Sidebar';
import Title from '../Sidebar/Title';
import SearchBar from '../SearchBar/SearchBar';
import { useScreenSize } from '../../src/DisplayContext';

function NavBar() {
  const { screenSize } = useScreenSize()
  return (
    <div className='p-2 rounded-xl flex flex-row justify-between items-center border-[2px] '>
      <div className='flex flex-row pl-2 gap-5 items-center'>
        <Sidebar/>
        <Title font={"28px"} />
      </div>
      <SearchBar/>
      {screenSize != 'small' &&
        <div className='flex flex-row gap-2'>
          <ThemeSwitcher/>
          <LoginButton/>
        </div>
      }
    </div>
  )
}

export default NavBar
```

The latter are conditionally rendered based on the screen size.

### 3.2.5.11.2 Search



As mentioned previously, the search component has two modes: exact and approximate. Here is the example of the approximate mode, which matches the word “vegetation” to a note talking about gardening, which does not have the exact word in it.

#### 3.2.5.11.2.1 ./src/components/SearchBar/SearchBar.tsx

The SearchBar component implements a search interface, utilizing React's state management and component lifecycle features, instead of using a separate library like Zustand, along with custom UI elements and an external API for search functionality. The component employs the useState hook to manage several aspects of the search process, including the user's search query (searchQuery), the selected search mode (activeMode), the search results (results), the loading state (loading), any errors that occur (error), the visibility of the search dialog (isDialogOpen), and pagination information (pagination). This allows the component to maintain its state across user interactions and asynchronous operations.

```
import React, { useEffect, useState } from 'react';
import { FaSearch } from "react-icons/fa";
import { universalApi } from '../../src/api/universalApi';
import { UniversalType } from '../../src/api/types/ArchiveTypes';
import { useScreenSize } from '../../src/DisplayContext';
import { Button } from "../../ui/button";
import {
  Dialog,
  DialogContent,
  DialogTrigger
} from "../../ui/dialog";
```

```

import { Input } from "../@/ui/input";
import { Label } from "../@/ui/label";
import PaginationSelector from '../Pagination/PaginationSelector';
import SearchCard from './Components/SearchCard';
import ToggleButtons from './ToggleButtons';

function SearchBar() {
    const [searchQuery, setSearchQuery] = useState<string>('');
    const [activeMode, setActiveMode] = useState<'approximate' | 'exact'>('exact');
    const [results, setResults] = useState<UniversalType[]>([]);
    const [loading, setLoading] = useState<boolean>(false);
    const [error, setError] = useState<string | null>(null);
    const [isDialogOpen, setIsDialogOpen] = useState<boolean>(false);
    const [pagination, setPagination] = useState<{ nextPage: number | null; page: number; } | undefined>(undefined);

    const { screenSize } = useScreenSize()

    useEffect(() => {
        if (searchQuery.length < 2 || !isDialogOpen)
            setResults([]);
    }, [searchQuery, isDialogOpen]);
}

```

This hook is used to clear the search results if the dialog is closed or the search query is short.

```

const handleSearch = async (pageIndex:number) => {
    setLoading(true);
    setError(null);
    try {
        const response = await universalApi.search(searchQuery, activeMode,
pageIndex);
        if (response && response.data) {
            setResults(response.data);
            setPagination(response.pagination)
        }
    } catch (err) {
        setError('An error occurred while searching.');
    } finally {
        setLoading(false);
    }
};

```

The handleSearch function contains the search logic. It is an asynchronous function that takes a page index as an argument. When called, it first sets the loading state to true and clears any previous errors. It then calls the universalApi.search function, passing the search query, active search mode, and page index. After receiving a successful response, it updates the results and pagination state with the data from the API. If an error occurs during the API call, the function sets the error state to display an error message.

```

const handleKeyPress = (event: React.KeyboardEvent<HTMLInputElement>) => {
    if (event.key === 'Enter' && searchQuery.length > 1) {

```

```

        handleSearch(1);
    }
};
```

The handleKeyPress function is an event handler attached to the search input field. It listens for the Enter key press. When the user presses Enter and the search query is at least two characters long, it triggers the handleSearch function with page index 1. This provides a convenient way for users to initiate a search by pressing Enter, in addition to clicking a search button.

```

const handleOpenDialog = () => {
    setIsDialogOpen(true);
    setResults([]);
}

const handleCloseDialog = () => {
    setIsDialogOpen(false);
    setResults([]);
};

return (
    <Dialog open={isDialogOpen} onOpenChange={setIsDialogOpen}>
        <DialogTrigger asChild>
            <Button size={screenSize == 'small' ? 'icon' : 'default'}
                className={`rounded-xl ${screenSize == 'small' ? 'w-[65px]' : ''} px-5 gap-2`}
                onClick={() => setIsDialogOpen(true)}
            >
                <FaSearch className='justify-self-start' size={10} />
                {screenSize != 'small' && <Label>Search...</Label>}
            </Button>
        </DialogTrigger>
        <DialogContent className="max-w-[500px] h-fit items-center">
            <div className="flex w-full items-center space-x-2">
                <Input
                    type="string"
                    className='border-0'
                    onChange={e => setSearchQuery(e.target.value)}
                    onKeyPress={handleKeyPress}
                    placeholder='Type here'
                />
                <ToggleButtons activeMode={activeMode}
                    setActiveMode={setActiveMode} />
            </div>
            {loading && <p>Loading...</p>}
            {error && <p>{error}</p>}
            {results.length>0 &&
            <>
                <div className='flex-col flex gap-2'>
                    {results.map(result => (
                        <SearchCard key={result.noteid} note={result}
                            onCloseDialog={handleCloseDialog} />
                    )))
                </div>
            </>
        </DialogContent>
    </Dialog>
);
```

```

        {(pagination && results) && <PaginationSelector
page={pagination?.page} nextPage={pagination?.nextPage}
fetchingFunction={handleSearch}/>}
        </>
    }
</DialogContent>
</Dialog>
);
}

export default SearchBar;

```

The rest of the code conditionally renders error or loading labels or cards which contain the fetched results

### **3.2.5.11.2.2 ./src/components/SearchBar/ Components/SearchCard.tsx**

These cards reuse the UniversalCard component to handle the UI behind displaying a note of some type. It also uses string manipulation in the click handler to create a cookie in the local storage appropriately named for the note type and to navigate to the url of the appropriate note type editing page.

```

import React from "react";
import { useNavigate } from "react-router-dom";
import { UniversalType } from "../../src/api/types/ArchiveTypes";
import UniversalCard from "../../Universal/UniversalCard.tsx";
import './SearchCard.css';

function SearchCard({ note, onCloseDialog }: { note: UniversalType, onCloseDialog: () => void }) {
    const navigate = useNavigate();

    function handleEditClick(noteId, type) {
        localStorage.setItem(`current${type.charAt(0).toUpperCase() + type.slice(1)}Id`, noteId.toString());
        navigate(`/${type + 's'}/edit`);
        onCloseDialog();
    }

    return (
        <UniversalCard note={note} handleEditClick={handleEditClick} />
    );
}

export default SearchCard;

```

### **3.2.5.11.2.3 ./src/components/Universal/ UniversalCard.tsx**

This component is reused across the application and renders basic note data like title and content, alongside a conditionally rendered note type-specific icon.

```

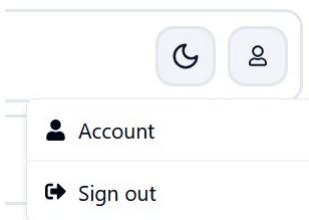
import React from 'react'
import { UniversalType } from '../../src/api/types/ArchiveTypes'
import { FaTasks, FaEdit } from 'react-icons/fa'
import { FaRegNoteSticky } from 'react-icons/fa6';
import { LuGoal } from 'react-icons/lu'
import { MdRepeat } from 'react-icons/md'
import { Button } from '../@/ui/button'
import TagLabel from '../TagLabel/TagLabel'
import { UUID } from 'crypto'

function UniversalCard({note, handleEditClick}: {note: UniversalType, handleEditClick: (noteId: UUID, type: string) => void}) {
    return (
        <div className='p-3 w-full bg-background rounded-xl border-[2px]'>
            <div className='flex flex-row justify-between'>
                <div className='flex flex-row items-center gap-2'>
                    {note.type === 'note' && <FaRegNoteSticky size={'20px'} />}
                    {note.type === 'task' && <FaTasks size={'20px'} />}
                    {note.type === 'habit' && <MdRepeat size={'20px'} />}
                    {note.type === 'goal' && <LuGoal size={'22px'} />}
                    {/* add other types here */}
                    <h3 className="note-title">{note.title}</h3>
                </div>
                <div className='flex flex-row items-center gap-2'>
                    {note.tags.map((tag, index) => (
                        <TagLabel key={index} name={tag.name} color={tag.color}
                        compressed={true} />
                    ))}
                    <Button variant="ghost" onClick={() =>
                        handleEditClick(note.noteid, note.type)} size="icon">
                        <FaEdit />
                    </Button>
                </div>
            </div>
            {note.content && <p className="text-md max-w-[400px] overflow-hidden
            overflow-ellipsis pl-1 pt-2">{note.content}</p>}
        </div>
    )
}

export default UniversalCard

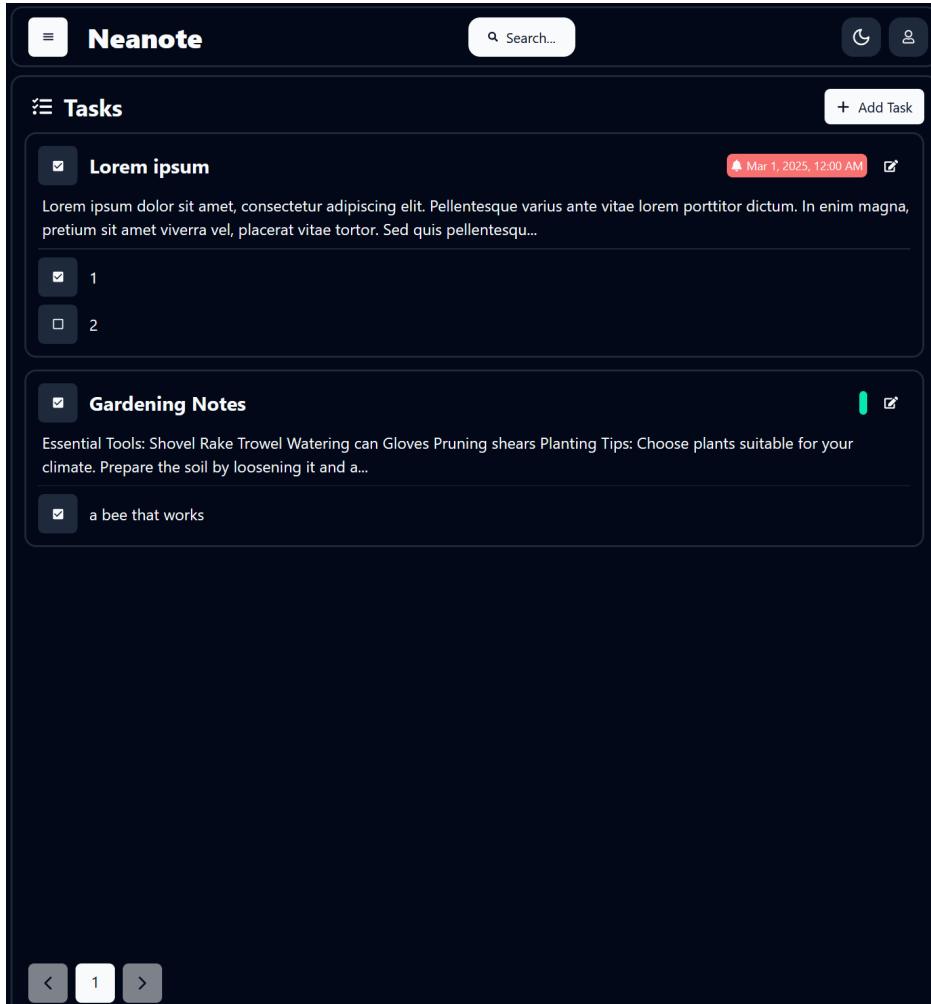
```

### 3.2.5.11.3 Theme selector



The navbar also contains a theme switcher and an account button, which opens a small menu that allows the user to sign out of their account by removing the JWT from the

browser's cookies. The next button is the theme selector, which, upon clicking, switches the website's theme from light to dark for night reading:



### 3.2.5.11.3.1 ./src/components/NavBar/Components/ThemeSwitcher.tsx

```
import React from 'react'
import { Button } from "../../ui/button"
import { LuMoon, LuSun } from "react-icons/lu";
import { useTheme } from "../../providers/theme-provider"

function ThemeSwitcher() {
  const [themeIsDark, setThemeIsDark] =
    React.useState( localStorage.getItem('chakra-ui-color-mode') === 'dark' ? true : false)
  const { setTheme } = useTheme()
  return (
    <Button className='rounded-xl border-[2px]' variant="secondary" onClick={() =>
      {setTheme(themeIsDark ? "light" : "dark")}
    }
  )
}
```

```

        setThemeIsDark( !themeIsDark ) }
    } size="icon">
        {themeIsDark ? <LuMoon size={20}/> : <LuSun size={20}/> }
    </Button>
)
}

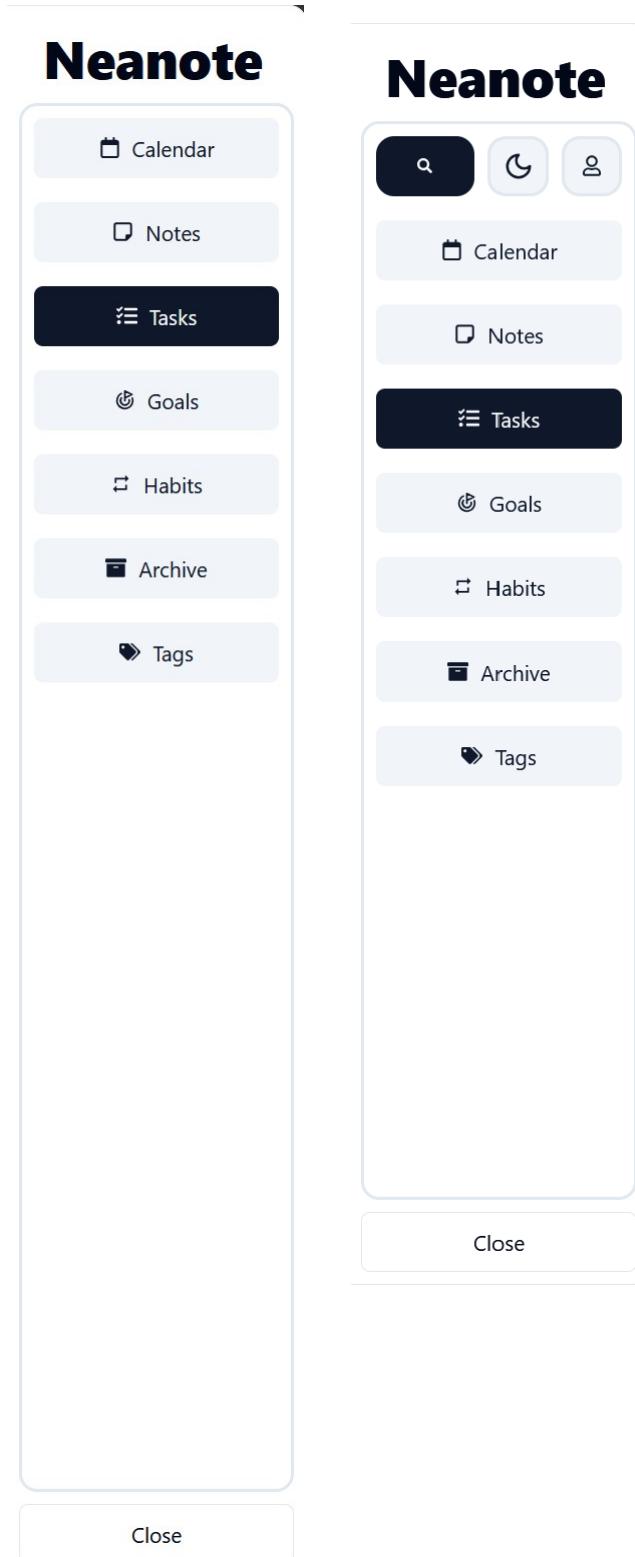
export default ThemeSwitcher

```

The switcher uses the `useTheme` hook from the theme-provider that was discussed previously to switch the theme between light and dark based on whether the theme is dark or not using negation and selection .

### **3.2.5.11.4 Sidebar**

I will finish this section by outlining the workings of the sidebar, as it is the main way users navigate through the website and contributes to the websites accessibility. This is because it is built to dynamically adjust based on the display size. On larger displays, the navbar can be wide, so the searchbar, theme switcher and the account button are located there. However, they cannot be placed there on small displays, so they are moved to the sidebar instead



### 3.2.5.11.4.1 ./src/components/SideBar/modules.tsx

```
import React from "react";  
280, Roman Sasinovich, 74561 IMS
```

```
import { FaTasks, FaArchive, FaTags } from "react-icons/fa";
import { LuGoal } from "react-icons/lu";
import { MdRepeat } from "react-icons/md";
import {FaRegNoteSticky,FaRegCalendar } from 'react-icons/fa6';

export const modules = [
  { link: 'calendar', text: (
    <div className="flex items-center gap-2">
      <FaRegCalendar /> Calendar
    </div>
  ), disabled: false },
  { link: 'notes', text: (
    <div className="flex items-center gap-2">
      <FaRegNoteSticky /> Notes
    </div>
  ), disabled: false },
  {
    link: 'tasks',
    text: (
      <div className="flex items-center gap-2">
        <FaTasks /> Tasks
      </div>
    ),
    disabled: false,
  },
  {
    link: 'goals',
    text: (
      <div className="flex items-center gap-2">
        <LuGoal /> Goals
      </div>
    ),
    disabled: false,
  },
  {
    link: 'habits',
    text: (
      <div className="flex items-center gap-2">
        <MdRepeat /> Habits
      </div>
    ),
    disabled: false,
  },
  {
    link: 'archive',
    text: (
      <div className="flex items-center gap-2">
        <FaArchive /> Archive
      </div>
    ),
    disabled: false,
  },
  {
    link: 'tags',
    text: (
```

```

        <div className="flex items-center gap-2">
          <FaTags /> Tags
        </div>
      ),
      disabled: false,
    },
];

```

The modules.tsx file serves as the main directory from which the icons and names are taken. It contains an array of dictionaries with link, text and disabled properties. The link property points to an appropriate route on the frontend, while the text property contains a styled JSX component with an icon and a name of the component. The disabled property was used during development to add some component to the Sidebar without it being implemented yet.

### **3.2.5.11.4.2 .src/components/SideBar/SideBar.tsx**

```

import React, { useState } from 'react';
import { IoMenu } from "react-icons/io5";
import { useLocation, useNavigate } from "react-router-dom";
import { Button } from "../@/ui/button";
import {
  Drawer,
  DrawerClose,
  DrawerContent,
  DrawerFooter,
  DrawerHeader,
  DrawerTrigger
} from "../@/ui/drawer";
import Title from "./Title";
import { modules } from './modules';
import { useScreenSize } from '../../src/DisplayContext';
import SearchBar from '../SearchBar/SearchBar';
import ThemeSwitcher from '../NavBar/ThemeSwitcher/ThemeSwitcher';
import LoginButton from '../NavBar/LoginButton/LoginButton';

function Sidebar() {
  const [open, setOpen] = useState(false);
  const location = useLocation();
  const navigate = useNavigate()
  const { screenSize } = useScreenSize();

  const handleLinkClick = (link) => {
    setOpen(false);
    navigate(`/${link}`);
  };
}

```

Here the link property of each module is added to the redirect link using JavaScript's alternative to f-strings in Python

```

return (
  <Drawer direction="left" open={open} onOpenChange={setOpen}>
    <DrawerTrigger asChild>
      <Button size="icon" variant="default"><IoMenu /></Button>
    </DrawerTrigger>
    <DrawerContent className="flex flex-col p-2 rounded-r-xl h-full w-[200px] mt-24 fixed bottom-0 right-0">
      <DrawerHeader className="pt-0 pb-1">
        <div className="flex flex-row gap-2">
          <Title font={"35px"} />
        </div>
      </DrawerHeader>
      <div className="flex rounded-xl w-full h-full flex-col border-[2px] p-2">
        <div className="flex flex-col gap-4">
          {screenSize === 'small' && (
            <div className='flex justify-between gap-2'>
              <SearchBar/>
              <ThemeSwitcher/>
              <LoginButton/>
            </div>)}
          {modules.map((module) => (
            <Button
              variant={`${location.pathname.includes(module.link) ? "default" : "secondary"}`}
              key={module.link}
              onClick={() => handleLinkClick(module.link)}
              disabled={module.disabled}
              className={}
            >
              {module.text}
            </Button>
          )))
        </div>
      </div>
      <DrawerFooter className="p-0 pt-2">
        <DrawerClose asChild>
          <Button className="w-full" variant="outline">Close</Button>
        </DrawerClose>
      </DrawerFooter>
    </DrawerContent>
  </Drawer>
)
}

export default Sidebar

```

This return statement renders the drawer component from shadcn UI library and maps the modules onto the sidebar as individual buttons that call the redirect function upon being clicked. The name of the application also exists on the drawer and also navigates the users to the dashboard.

## 4 Testing

After many hours of tiring work writing, debugging and documenting the code, I have, despite skipping a half of the code out of the total number of lines of this project, hopefully, shown the full range of techniques, practices and paradigms used in the application.

I have fixed most of the errors in my code prior to writing this document, but it is never wrong to re-check everything again. The two main ways I will be testing my code are:

- **Hands-on testing:** I will construct a series of tests of almost, if not all functions of the app from **1.6** and perform the tests by myself through the user interface, to see how the user would actually see the program. I will use normal, as well as boundary and erroneous data to see how well does my validation work. I will also attempt to trigger errors on the backend or throttle my connection to evaluate error handling of the web app. Additionally, some tests will be conducted at a different aspect ratio to simulate a mobile phone. Features, that are made for smaller displays will also be tested.
- **Unit testing:** Unit testing is a form of testing in which the smallest testable parts of an application (like a component or a subroutine), called units, are individually tested to ensure that there will be no side effects when they are combined together.
  - For the frontend, I will use Jest and React Testing Library. These libraries can simulate button clicks, inputs, connection issues and many other things.
  - For Python unittest it is a built in module and will be used to see how boundary, erroneous and normal data is treated on the backend.
  - Unit tests can also mock an API response to simulate an error on the backend or an external API to test for appropriate error handling. Each test is a function or a separate file, which will be included either in this section or at the end of the project. When I run an appropriate trigger function, each test in my application will run and produce a result. It will be displayed as a list containing test names, the time they took to run and whether an expected value was received. The results of these runs will be shown in this section.

### 4.1 Hands-on testing

#### 4.1.1. Setup

Firstly, let's run the web app in development mode. To do that, I navigate into the root directory of the React app (`./neanote`) and run

```
$ npm run dev
> neanote@0.0.0 dev
> vite

VITE v5.2.13 ready in 2616 ms
→ Local: http://localhost:5173/
→ Network: use --host to expose
→ press h + enter to show help
```

npm run dev

in the command line to start the frontend.

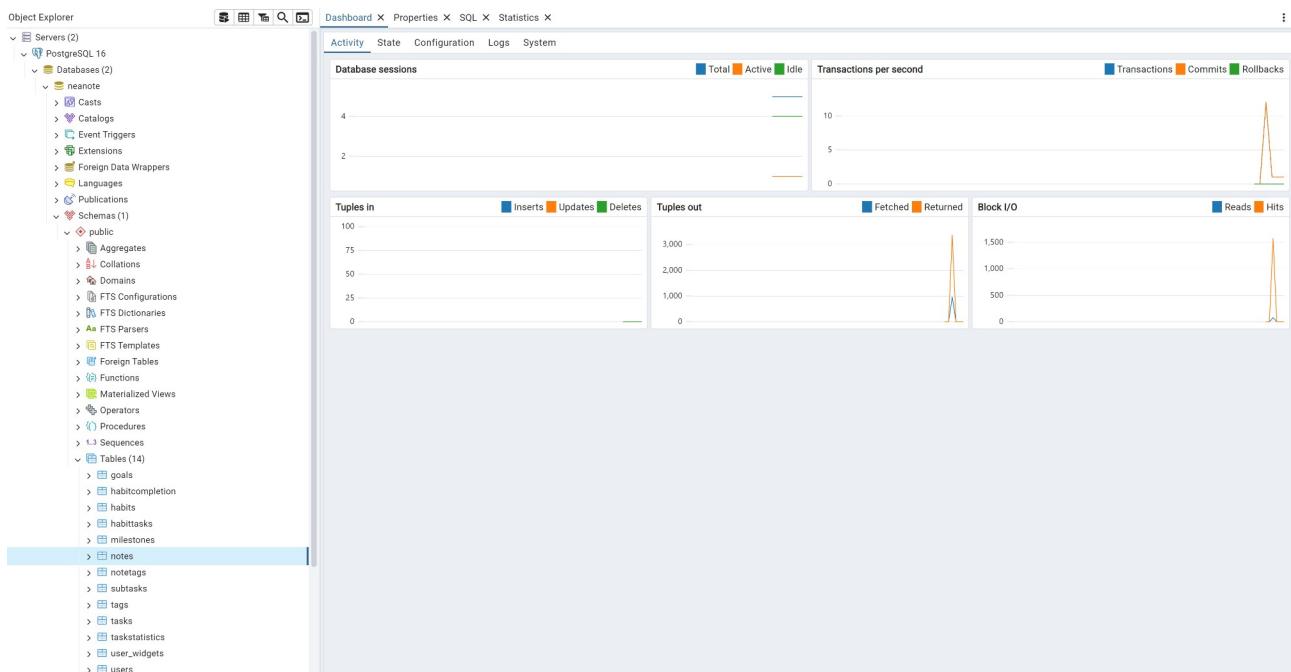
Node package manager (npm) detects the command and calls vite to start the development

server. As can be seen from the screenshot, the frontend is running at localhost:5173. Navigating to this link in my browser, we are instantly redirected to the get-started endpoint as I do not have a valid JWT token. However, the backend server is not running yet.

Therefore, we need to start the backend server by navigating to the ./backend directory and running app.py

```
$ C:/Users/Roman/AppData/Local/Programs/Python/Python312/python.exe h:/Code/Note/backend/app.py
Loading existing model...
* Serving Flask app 'app'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on http://127.0.0.1:5000
Press CTRL+C to quit
* Restarting with stat
Loading existing model...
* Debugger is active!
* Debugger PIN: 900-508-336
```

This will launch the Flask app in development mode. The last step is to check that the database is running, which can be done with pgAdmin – a Postgres database management tool.



The database is active, so its time to start testing.

## 4.2 Registration and login

Test #	Details	Example	Test data type	Expected result	Result
1	Register a user with a	Username:test Email:	Erroneous	An error is communicated to	Pass

	short password	<u>test@example.com</u> Password: 1234		the user, registration does not pass	
2	Register a user with a weak password	Username:test Email: <u>test@example.com</u> Password: 12AB21	Erroneous	An error is communicated to the user, registration does not pass	Pass
3	Register a user with an erroneous email	Username:test Email: <u>@examplecom</u> Password: UI2!sd23	Erroneous	An error is communicated to the user, registration does not pass	Pass
4	Register a user with a short username (<4)	Username: t Email: <u>test@example.com</u> Password: UI2!sd23	Erroneous	An error is communicated to the user, registration does not pass	Pass
5	Register a user with a short username (5)	Username: 12345 Email: <u>test@example.com</u> Password: UI2!sd23	Boundary	Registration event passes, the user is redirected for login	Pass
6	Log in with a username that doesn't exist	Username: 123456 Password: UI2!sd23	Erroneous	Login is denied, an error is communicated to the user	Fail, Passed with corrections
7	Log in with a wrong password	Username: 12345 Password: 123456	Erroneous	Login is denied, an error is communicated to the user	Pass
8	Log in with correct credentials	Username: 12345 Password: UI2!sd23	Normal	Login passes, the user is redirected to the dashboard	Pass
9	Log out	Navigate to the Account iconbutton on the navbar and press "log out:	Normal	The user is logged out and redirected to the /get-started page	Pass

## Test 1

**Result:**

An error is displayed to the user, test passes.

# Neanote

## Register

Username

Email

Password

Password must be at least 6 characters.

New to Neanote? Log in

**Submit**

## Register

Username

**Test 2****Result:**

Email

Password

28 Password must contain at least one uppercase letter, one lowercase letter, one digit, and one special character.

S

New to Neanote? Log in

**Submit**

An explanatory error message guides the user to create a strong password. Pass

### Test 3

#### Result:

The user is informed of their mistake. Pass

## Register

Username

test

Email

@examplecom



Invalid email address.

Password

UI2!sd23

New to Neanote? Log in

Submit

## Register

Username

t

### Test 4

#### Result:

Username must be at least 4 characters.

Email

test@example.com



Password

UI2!sd23

New to Neanote? Log in

Submit

Registration denied, user is informed. Pass

## Test 5

### Result:

2 | 37f6f404-bbe8-4dd4-a63f-de0d485eed02 | 12345 | test@example.com | \x2432622431322456744c304b4443444d6d44796c45655a662e502e4265423155745a6538456b512e6d5767682e6f4363633869525231474c506e...

Registration was successful. A new record was added to the Users table containing the entered user data and an encrypted password. Pass

## Test 6

### Result:

#### Login

Username

123456

Password

UI2!sd23

Already have an account? Register

Submit

No error message was communicated to the user. Fail. Upon closer inspection, the login() function in the useLogin.ts file lacked a showToast() function, which displays an error. Fix applied at **3.2.5.3.2**.

**Pass**

## Test 7

### Result:

#### Login

Username

Validation did not allow me to enter a simple password, but with a wrong complex password an error was communicated to the user. Pass

Password

Already have an account? [Register](#)

## Test 8

### Result:

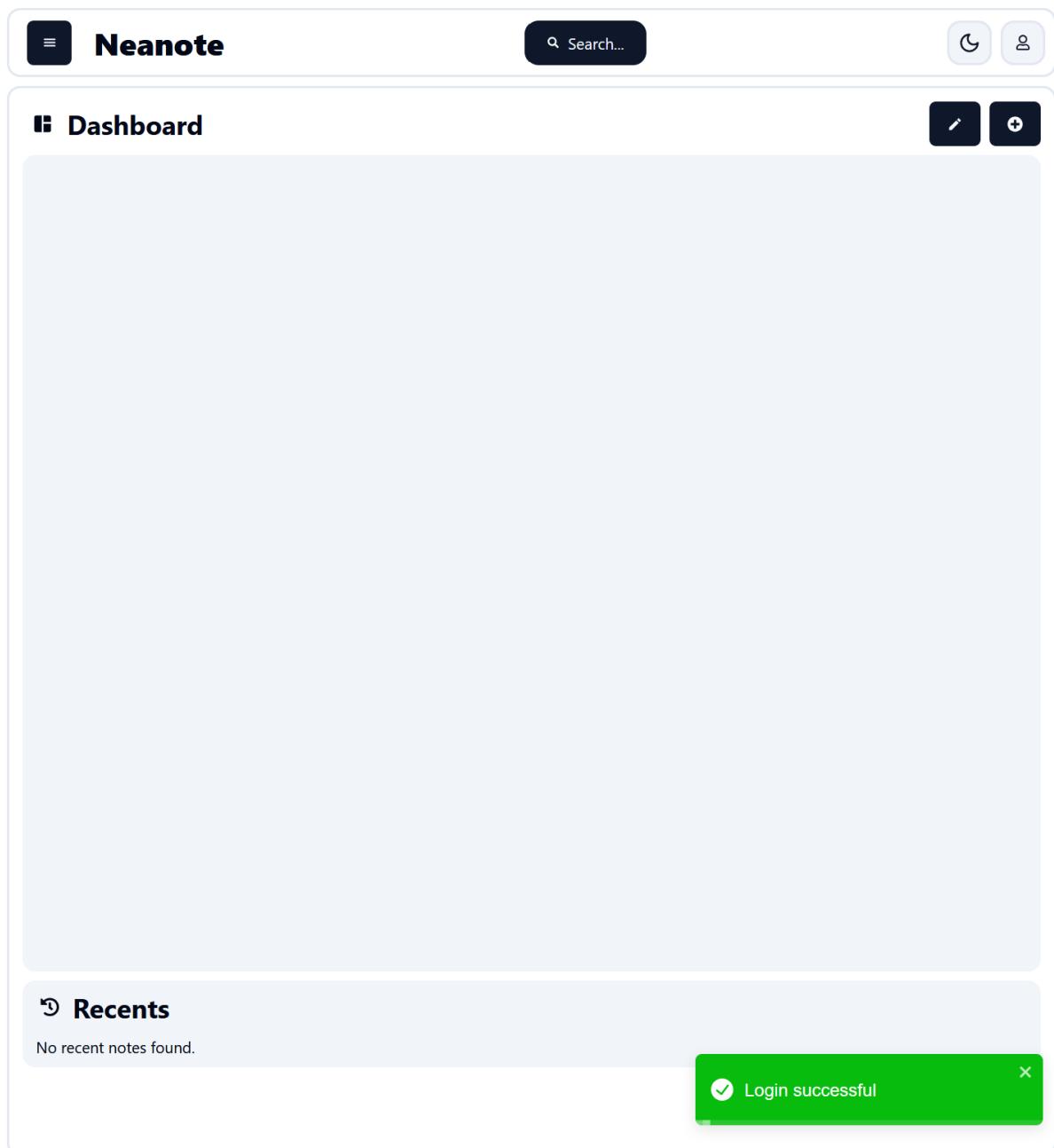
#### Login

Username

The login works, the user is redirected to the dashboard and a success message is shown. Pass

Password

Already have an account? [Register](#)



## Test 9

### Result:

The user is redirected to the get-started page. Pass

## 4.3 Notes

Test	Details	Example	Test data type	Expected result	Result

#					
10	Create a note	Title: Normal Test 1 Content: Description Test	Normal	Note is created, a success message is displayed and the user is redirected from the creation page to the edit page	Pass
11	Attempt to create a note with empty title	Title : Content: Description Test	Erroneous	Note cannot be created	Pass
12	Edit a note	Title: Normal Test 1 Content: Description Test 2	Normal	Note edited successfully	Pass
13	Attempt to edit a note by removing its title	Title : Content: Description Test 2	Erroneous	Edit is not allowed	Pass
14	Create a note with a title of 1 character	Title: T Content: Description Test 3	Boundary	Creation is allowed	Pass
15	Create a note with a very long title and description	Title: *string of 200 characters* (maximum allowed is 100) Content: *string of 2000 characters* (maximum allowed is 1000)	Erroneous	Creation is forbidden	Pass
16	Check whether newly created notes were vectorized in the database		Normal	Each note has a multidimensional vector in the Notes table of the database	Pass
17	Check whether all notes are on the notes page		Normal	Each note is displayed on the Notes page	Pass
18	Evaluate how notes		Normal	Interface is scaled	Pass

	are displayed on the Notes page and Edit pages on a small screen			appropriately and each field is editable	
19	Delete a note		Normal	Note removed from the Notes page	Pass
20	Archive a note		Normal	Note removed from the Notes page and is instead available on the Archive page	Pass

## Test 10

Result:

The screenshot shows the Neanote application interface. At the top, there is a header bar with the Neanote logo, a search bar, and user profile icons. Below the header, the title "Edit Note" is displayed. The main area contains two text input fields: one labeled "Normal Test 1" containing the text "Normal Test 1" and another labeled "Description Test" containing the text "Description Test". At the bottom of the screen, there are three buttons: "Delete" (trash icon), "Archive" (archive icon), and "Save" (disk icon). A green success message box at the bottom right corner states "Note created successfully" with a checkmark icon.

The note has been successfully created and the edit note page is now open. A success message is seen at the bottom of the screen. Pass

**Test 11:**

**Result:**

The screenshot shows the Neanote application interface. At the top, there is a header with the Neanote logo, a search bar, and user profile icons. Below the header, the title "Create Note" is displayed. There are two input fields: "Title" and "Description". The "Title" field is empty and has a red validation error message "Title is required" below it. The "Description" field contains the text "Description Test". At the bottom left is a "Save" button with a disabled icon. On the right side of the screen, there is a "Tags" section.

The save button is disabled, as there is a validation error under the “title” field. Pass

**Test 12:**

**Result:**

The screenshot shows the Neanote application interface. At the top, there is a header with the Neanote logo, a search bar, and user profile icons. Below the header, the title "Edit Note" is displayed. There are two input fields: "Normal Test 1" and "Description". The "Description" field contains the text "Description Test 2". At the bottom left are three buttons: a trash can icon, an archive icon, and a "Save" button with a disabled icon. On the right side of the screen, there is a "Tags" section.

Note’s description was successfully changed. The Save button is now disabled as there are no new pending changes.

**Test 13:**

**Result:**

295, Roman Sasinovich, 74561 IMS

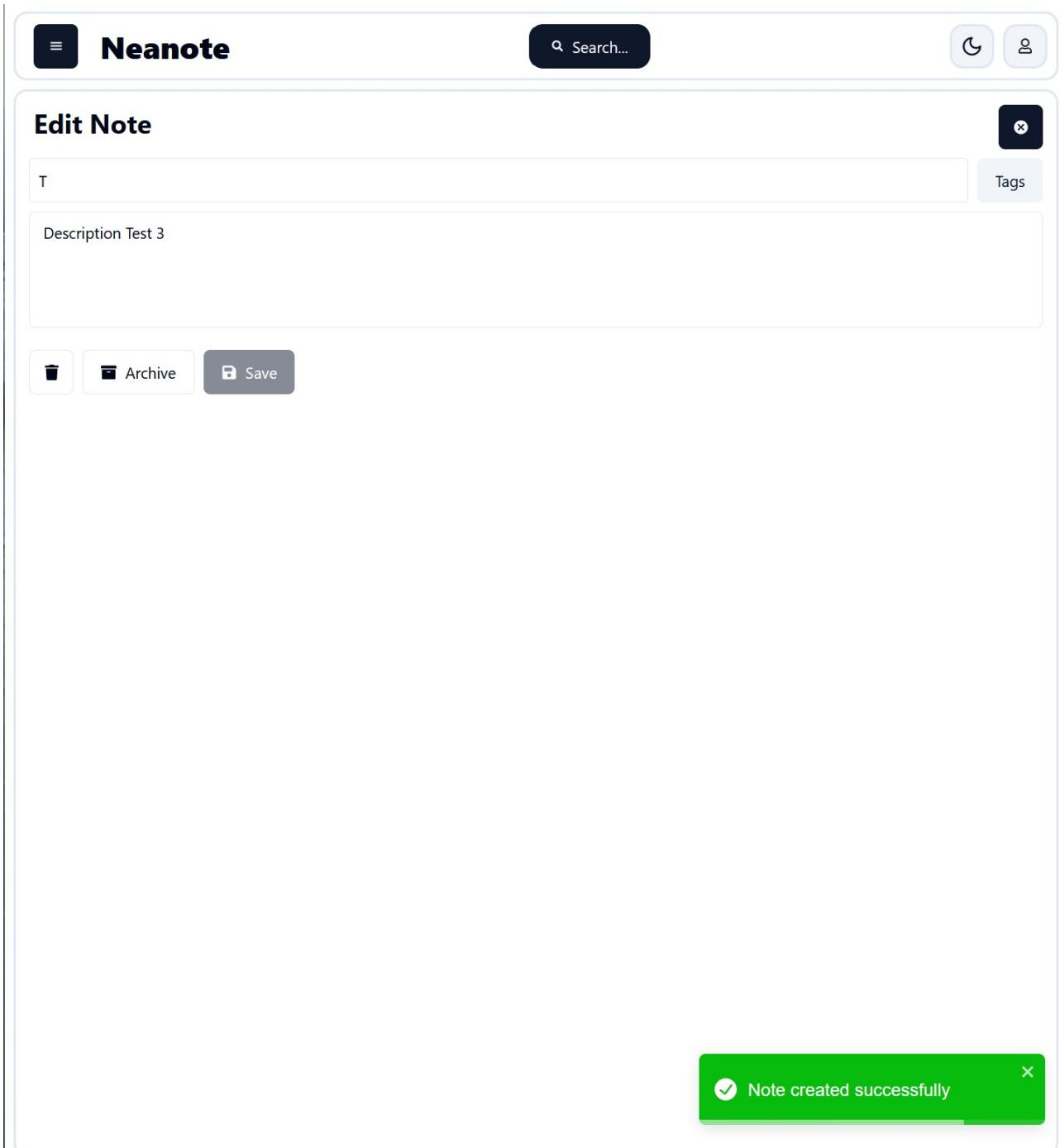
The screenshot shows the Neanote application interface. At the top, there is a dark header bar with the 'Neanote' logo on the left, a search bar with the placeholder 'Search...', and two small circular icons on the right. Below the header, the main content area has a title 'Edit Note' and a close button (X) in the top right corner. A large input field labeled 'Title' is present, with a red error message 'Title is required' below it. A text area labeled 'Description Test' follows. At the bottom of the form are three buttons: a trash icon, an 'Archive' icon, and a 'Save' button.

A similar error is shown as in test 11. Pass

#### Test 14:

The screenshot shows the Neanote application interface. At the top, there is a dark header bar with the 'Neanote' logo on the left, a search bar with the placeholder 'Search...', and two small circular icons on the right. Below the header, the main content area has a title 'Create Note' and a close button (X) in the top right corner. A large input field labeled 'T' is present, with a 'Tags' button to its right. A text area labeled 'Description Test 3' follows. At the bottom of the form is a single 'Save' button.

Result:



Note successfully created. Pass

### Test 15:

I used an online Lorem ipsum placeholder text generator to generate long paragraphs of text that I later pasted into the title and content fields.

### Result:

297, Roman Sasinovich, 74561 IMS

**Create Note**

s egestas velit massa sed tellus. Curabitur dapibus placerat maximus. Maecenas sit amet neque posuere, varius diam at, gravida turpis.

Tags

Title cannot exceed 100 characters

Lo&rem ipsum dolor sit amet, consectetur adipisic&ing elit. Duis at ullamcorper risus. Nulla ornare tristique est, nec auctor orci gravida sit amet. Maecenas id felis sed nibh sodales aliquet sed at mauris. Sed ut risus tellus. Donec non dolor sit amet tellus semper efficitur ut et quam. Ut orci nunc, viverra et imperd&et vitae, consectetur sit amet purus. Morbi faucibus vulputate arcu e&et tristique. In hac habitasse platea dictumst. Praesent imperd&et ligula libero, eget pellentesque ex luctus eu. Fusce ultrices lectus eu metus aliquam, sed aliquet urna commodo.

In vestibulum lacus eu lacus maximus, sed commo&do arcu sodales. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Quisque eu placerat sem. Praesent consequat lectus sed lectus bibendum consequat. Duis in pretium nulla. Fusce lorem arcu, pretium ac neque non, facilisis cursus lorem. Sed porta turpis ac leo placerat, eu vestibulum orci sollicitudin. Interdum et malesuada fames ac ante ipsum primis in faucibus. Praesent placerat in lacus in efficitur. Mauris ac volutpat dui. Donec iaculis lobortis urna, at porttitor orci vehicula vitae. Praesent tincidunt enim vel tincidunt ullamcorper. Donec cursus metus neque, at finibus nibh sagittis at.

Sed nibh sapien, malesuada sit amet velit convallis, ullamcorper ullamcorper orci. Quisque lacus dui, suscipit vel rutrum sit amet, aliquam nec felis. Sed sapien nunc hendrerit in lorem nec tempus pretium ligula. Phasellus dolor tellus, placerat ut orci nec, sollicitudin iaculis mi. Ut sem nibh vulputate vitae ultrices nec sagittis fringilla orci. Etiam elementum suscipit gravida. Quisque et varius mi. Praesent sit amet hendrerit libero. Etiam nec eleifend est.

Suspendisse potenti. Proin facilisis tortor nec hendrerit feugiat. Nulla suscipit malesuada mattis. Sed ante massa, cursus ut ornare nec fringilla at massa. Duis tincidunt aliquam est. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia curae; Etiam placerat non ex id viverra. Morbi vulputate porttitor commo&do. Curabitur consequat tristique semper. In quis risus sapien. Nullam quis egestas lorem.

Nulla blandit cursus odio vel dignissim. Nulla tristique nisi enim. Suspendisse eu quam malesuada, porta purus et, gravida urna. Donec quis enim metus. Nulla erat augue, imperd&et nec condimentum ac, interdum sit amet est. Aenean vulputate, est at scelerisque bibendum, felis eros dapibus enim, sit amet ornare nisl sapien ut massa. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Sed id vestibulum neque. Proin efficitur, est at luctus condimentum, nunc velit posuere risus, at iaculis mi est ut nunc. Sed a mauris tincidunt, cursus neque nec ornare purus. Aenean mi tortor, vulputate vitae nisi e&et, venenatis accumsan tellus. Ut tempor, quam et tempor vulputate enim nunc vulputate mauris, rhoncus egestas velit massa sed tellus. Curabitur dapibus placerat maximus. Maecenas sit amet neque posuere, varius diam at, gravida turpis.

Content cannot exceed 1000 characters

 Save

Save is disabled as title and content have too many characters. Pass

**Test 16**

Running the SELECT \* FROM notes produces the following table:

**Result:**

```
{-0.04191935434937477,0.15557605028152466,0.010106091387569904,0.04353015869
{-0.04193319380283356,0.1413801908493042,0.005597058683633804,0.044927448034
```

The vector property of each Note record contains an array of real numbers (vectors). Pass.

**Test 17**

298, Roman Sasinovich, 74561 IMS

## Result:

The screenshot shows the Neanote application interface. At the top, there is a header with the logo and the word "Neanote". To the right of the logo is a search bar with the placeholder "Search...". Further to the right are two small icons. Below the header, there is a section titled "Notes" with a sub-section title "T". Inside the "T" section, there is a note with the text "Description Test 3" and a small checkmark icon in the top right corner. Below the "T" section, there is another section titled "Normal Test 1" with a note containing the text "Description Test 2" and a small checkmark icon in the top right corner. At the bottom left of the screen, there are three navigation buttons: a left arrow, a central button with the number "1", and a right arrow.

The two notes I was allowed to create by the programme are displayed on the Notes page.  
Pass

## Test 18:

In my browser's developer tools, I press the Responsive Design Mode and select one of the mobile phone presets like Galaxy Note 20, which resizes the screen to match the aspect ratio of the phone.

**Result:**

The image displays two screenshots of the Neanote application interface.

**Left Screenshot (Notes List):**

- The title bar says "Neanote".
- A search icon is in the top right corner.
- The main area shows a list of notes:
  - T**: Description Test 3
  - Normal Test 1**: Description Test 2
- A button "+ Add Note" is at the top right of the list area.

**Right Screenshot (Edit Note):**

- The title bar says "Neanote".
- A search icon is in the top right corner.
- The main area shows the details of the note "Normal Test 1":
  - Description: Description Test 3
  - Tags: Tags
- Action buttons at the bottom include: a trash bin icon, an archive icon labeled "Archive", and a save icon labeled "Save".

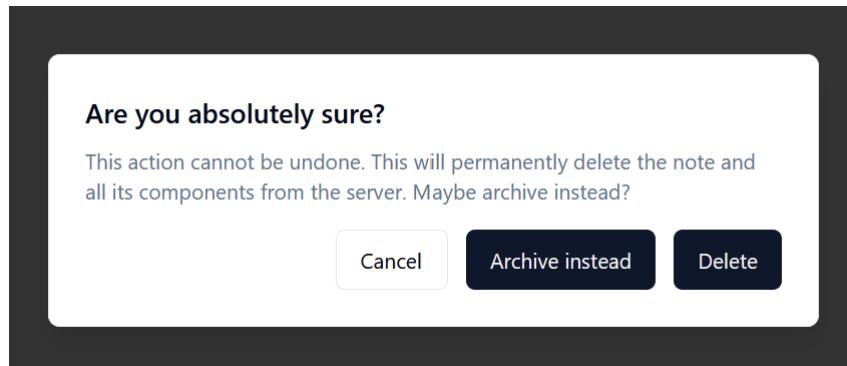
**Pagination:**

At the bottom left, there are three buttons: a left arrow, the number "1" (highlighted in dark blue), and a right arrow.

On both main and edit screens, the note cards and input fields are compressed, while still being clickable. Pass

### Test 19

After clicking the Delete button, I get warning message about the irreversability of this action:



**Result:**

The screenshot shows the Neanote application interface. At the top, there is a header with the brand name "Neanote". To the right of the header are three icons: a search bar, a refresh symbol, and a user profile icon. Below the header, a navigation bar contains a "Notes" button and an "Add Note" button. The main content area displays a note card for "Normal Test 1". The note has a title "Normal Test 1" and a description "Description Test 2". To the right of the note card is a small edit icon. At the bottom of the screen, there is a pagination control with three buttons: a left arrow, a central number "1", and a right arrow. To the right of the pagination control is a green success message box with a checkmark icon and the text "Note deleted successfully".

When I clicked the “Delete” button, I got redirected to the main Notes page and the note titled T (that I was deleting) is gone and a success message is shown.

## Test 20

### Result:

Pressing the Archive button on the remaining note removes it from the main page:

The screenshot shows the Neanote application interface. At the top left is the logo 'Neanote'. To its right is a search bar with the placeholder 'Search...'. Further to the right are two small circular icons: one with a refresh symbol and another with a user profile icon. Below the header, a dark blue navigation bar contains a 'Notes' section with a list icon and a 'Add Note' button with a '+' icon. The main content area is currently empty, indicating no notes have been added. At the bottom left, there are three small grey buttons with arrows pointing left, right, and right respectively. On the far right, a green success message box is displayed, containing a checkmark icon and the text 'Note archived successfully'.

And adds it to the /archive page:

The screenshot shows the Neanote application interface. At the top, there is a header with the logo and the word "Neanote". To the right of the logo is a search bar with the placeholder "Search...". Further to the right are three icons: a circular arrow, a user profile icon, and a gear icon.

The main content area is titled "Archive". Below the title, there is a card for a note titled "Normal Test 1". The note has a red trash can icon and a blue refresh/circular arrow icon to its right. The note's content is "Description Test 2".

A red text message at the bottom of the screen states: "Archived notes will be removed after 30 days if not in use".

At the bottom of the archive list, there are navigation buttons: a left arrow, a central number "1" (which is dark blue), and a right arrow.

Pass.

#### 4.4 Tags

Test #	Details	Example	Test data type	Expected result	Result
21	Create a tag	Title: Tag Test 1 Color: Blue	Normal	Tag is created, a success message is displayed and the user is	Pass

				redirected to the Tags page	
22	Attempt to create a tag with no title	Title : Color: black	Erroneous	Tag cannot be created	Fail  See fix at 3.2.5.5. 1 or here
23	Create a tag with a short title	Title: 1 Color: Purple	Boundary	Tag is created	Pass
24	Edit a tag's color and title	Title: Tag Test 2 Color: Red	Normal	Tag edited successfully	Pass
25	Assign the tag to a note		Normal	Tag is now displayed along the note's title on the Notes page	Pass
26	Delete Tag		Normal	Tag is no longer on the Tags page and any other notes that it was associated with.	Pass
27	Test tags for different display types		Normal	Tags are resized appropriately	Pass

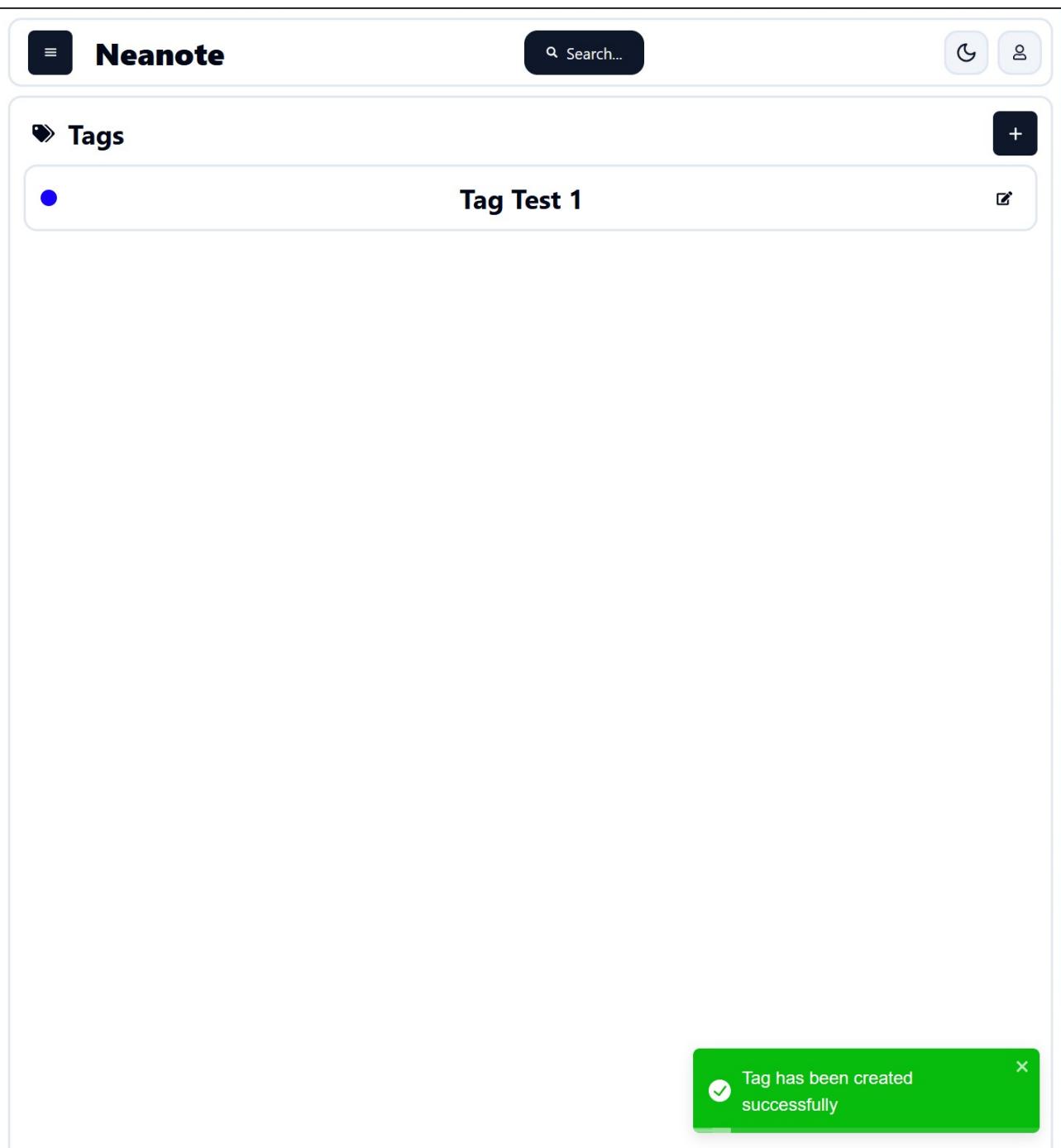
## Test 21

The screenshot shows the Neanote application interface. At the top, there is a header with a menu icon, the brand name "Neanote", a search bar with a magnifying glass icon, and two user account icons. Below the header, the main content area has a title "Create Tag". In the center, there is an input field containing the text "Tag Test 1". Directly beneath the input field is a horizontal color picker bar, which is currently filled with a solid blue color. At the bottom left of the content area, there is a dark button labeled "Save".

I navigated to the Tags page, pressed the “Create Tag” button and entered the title value in the input box and picked a color from the color picker bar.

### Result:

305, Roman Sasinovich, 74561 IMS



Tag is created and is shown on the main page with a blue icon (which is the same as the color I chose). Pass

**Test 22:**  
**Result:**

The screenshot shows the Neanote application interface. At the top, there is a header with the logo and the title "Neanote". Below the header is a search bar and some user account icons. The main content area is titled "Create Tag". It contains a "Title" input field, which is currently empty and highlighted with a black rectangle. Below the input field is a "Save" button. At the bottom of the page, the browser's developer tools Network tab is open, showing a list of network requests. One request is selected, showing a POST request to "http://localhost:5000/api/tags/create" with a status of 500 INTERNAL SERVER ERROR. The response body is empty.

Despite the title field being empty, a request is still sent, but is caught and handled by the backend instead. The Save button should be disabled.

## Fix:

To fix this, I had to add the following line of code to the Save Button component in Tags.tsx:

```
disabled={tagTitle.length == 0}
```

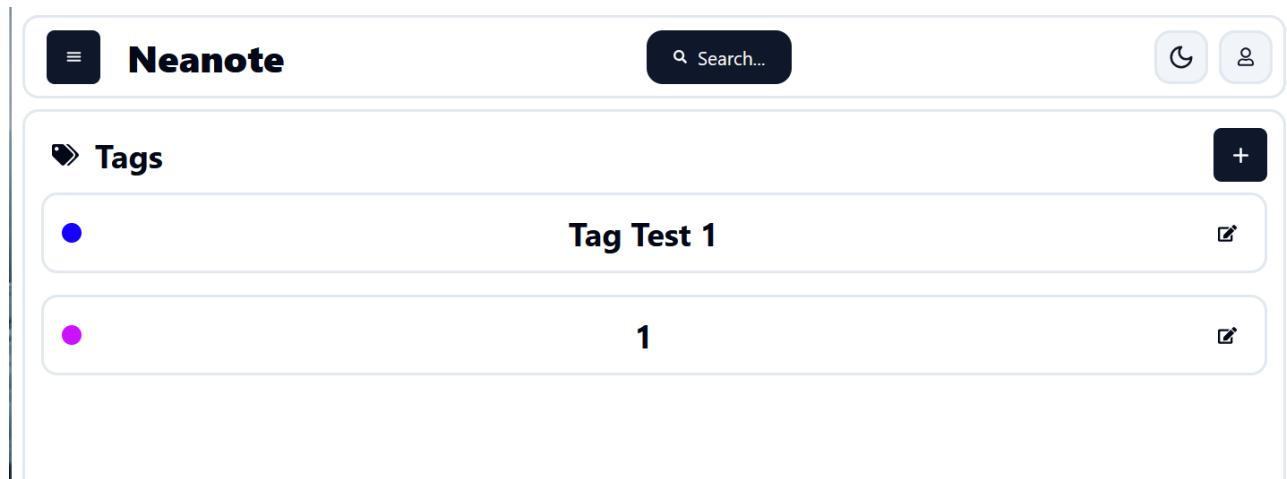
Now, everything works as it should and the button cannot be pressed.

## Test 23

### Result:

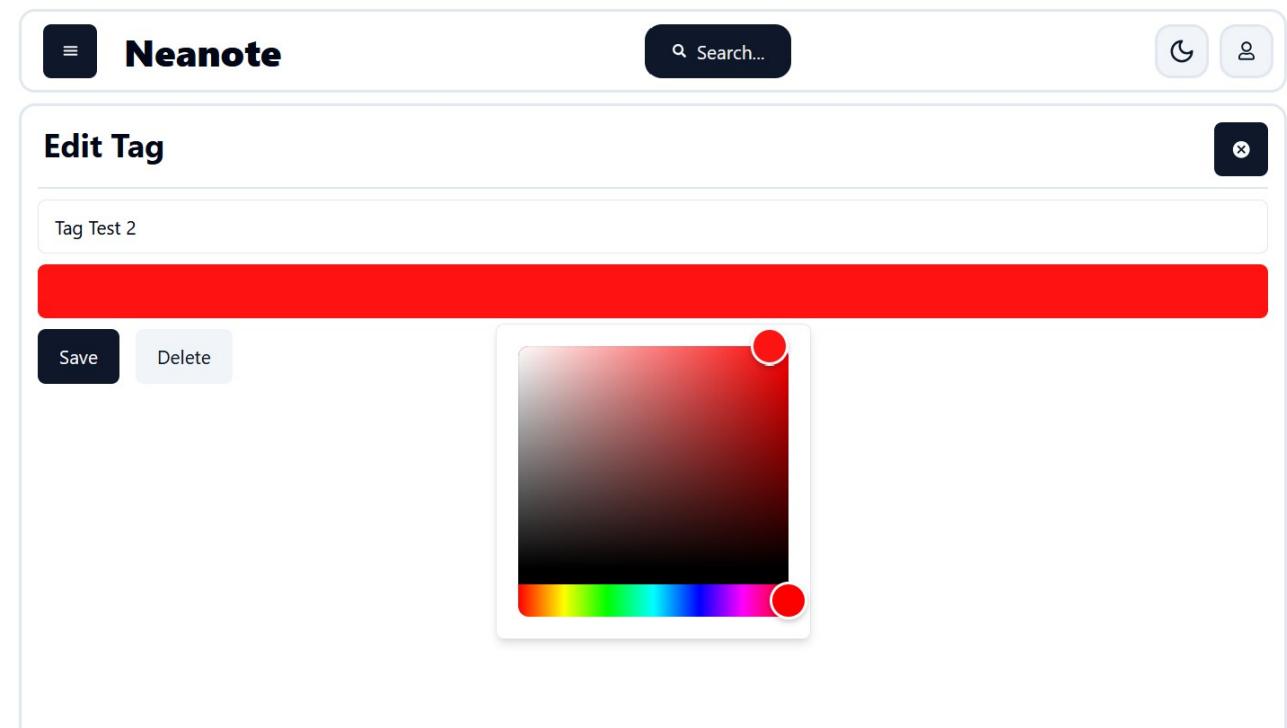
307, Roman Sasinovich, 74561 IMS

A new tag with a correct title and color is added to the Tags page:



The screenshot shows the Neanote application interface. At the top, there is a header with the logo and the word "Neanote". Below the header is a search bar with the placeholder "Search...". On the right side of the header are three small icons: a refresh symbol, a user profile, and a plus sign for adding new items. The main content area is titled "Tags". It displays two tags: "Tag Test 1" with a blue circular icon and "1" with a purple circular icon. Each tag has a small edit icon (pencil) to its right.

## Test 24



The screenshot shows the "Edit Tag" page for "Tag Test 2". The title "Edit Tag" is at the top, along with a close button. Below it is a text input field containing "Tag Test 2". Underneath the input field is a large red rectangular area. To the left of this area are two buttons: "Save" (dark blue) and "Delete" (light gray). To the right is a color picker interface. The color picker shows a gradient from red to black at the top, followed by a color wheel with a red dot indicating the selected color. A color bar at the bottom shows a gradient from yellow to red. The entire "Edit Tag" page is enclosed in a light gray border.

For this test, I edit the title of the purple tag and select red on the color picker.

### Result:

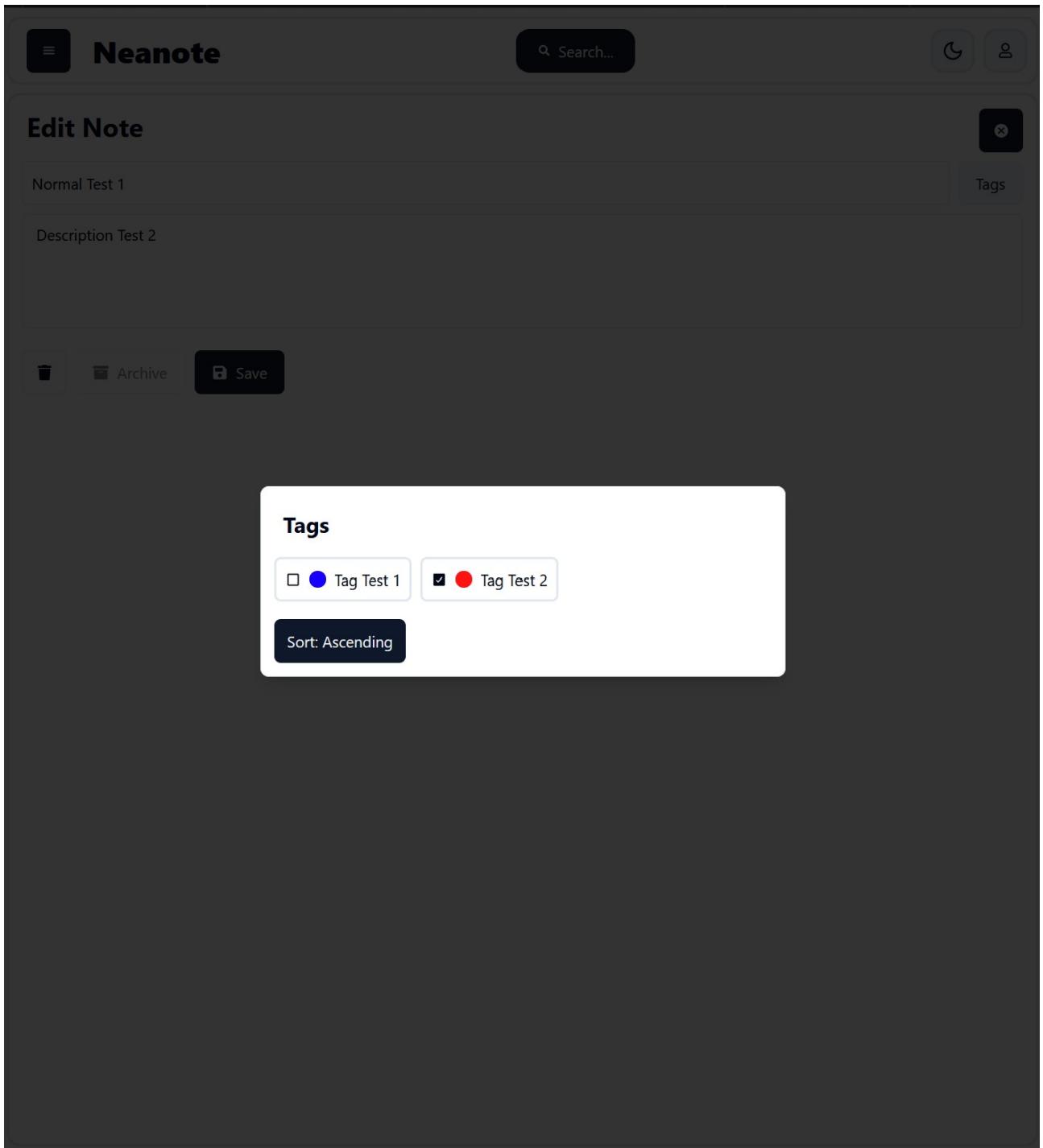
Tag successfully changes color and title on the main page:



The screenshot shows the Neanote application interface again. It displays a single tag with a red circular icon, a red title "Tag Test 2", and a small edit icon to its right. This indicates that the tag's color and title have been updated successfully.

Pass.

## Test 25



To conduct this test, I navigated to the Notes page and clicked the “Edit” button on one of the notes, after which I selected the Tags menu and ticked one of the tags that I want to associate with this note. Pressing the “Save” button and exiting the Edit mode, this tag is now displayed on the Note card at the main Notes page as a badge of the correct color. Upon clicking the tag, the correct title is shown:

## ☐ Notes

+ Add Note

### Normal Test 1

Description Test 2



Tag Test 2

## Test 26

Result:



Neanote

Search...



## Tags



Tag Test 1



✓ Tag has been deleted successfully X

Tag is removed from the Tags page and is no longer shown on the note it was associated with:

### Normal Test 1

Description Test 2



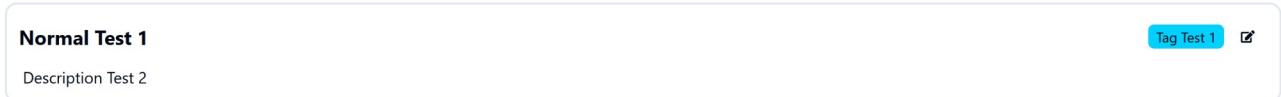
Pass.

## Test 27

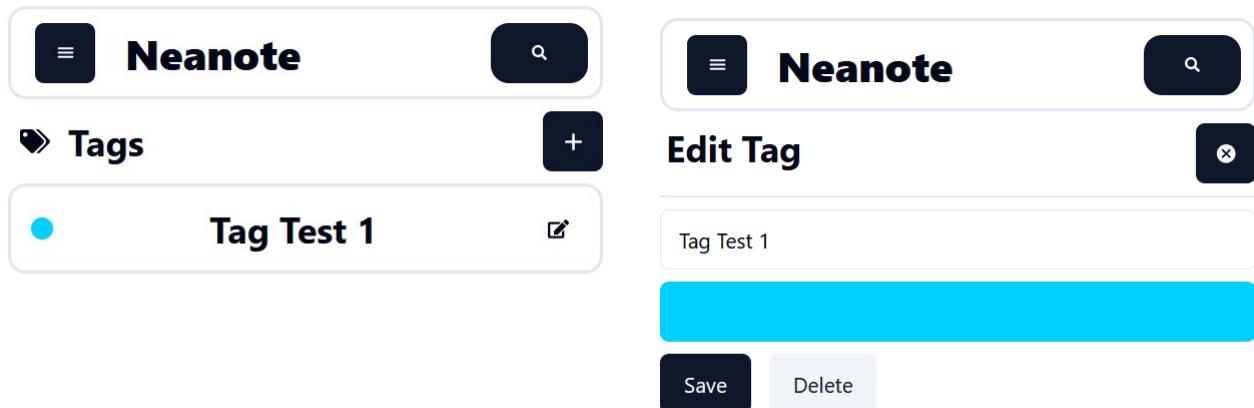
310, Roman Sasinovich, 74561 IMS

## Result:

If I connect a different tag to the previously mentioned note and resize the window so it is wider, the colored badge expands into a tag, containing its name:



On screens of a smaller size, the Tags page is properly resized and the tag Edit or Create pages are also usable:



Pass.

## 4.5 Tasks

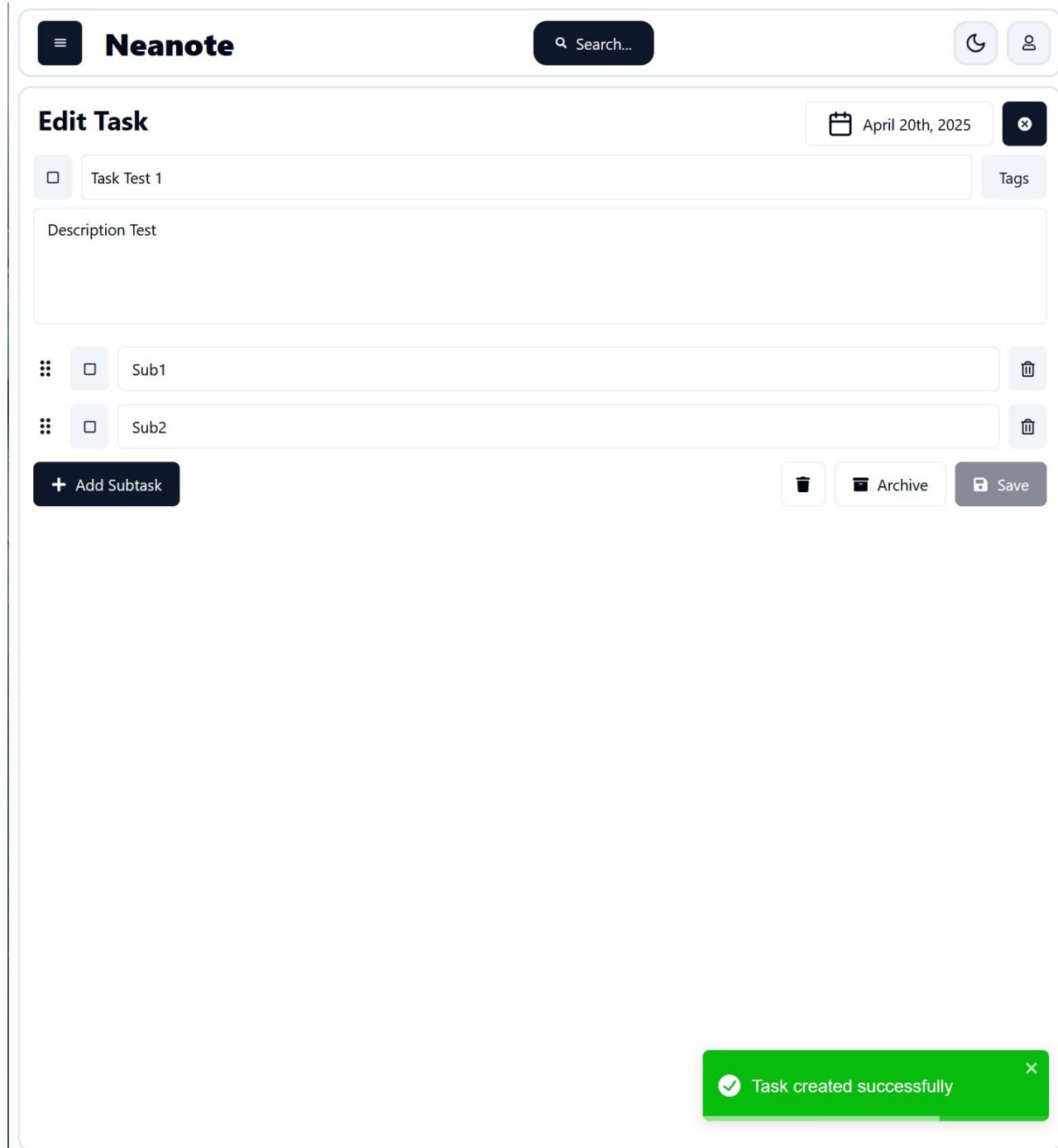
Since tasks contain similar methods and schemas for validation and other features, I believe it is unnecessary to separately document validation of fields like content and title, archiving, vectorization and tagging. Instead, I will focus on new features from now on and let the unit tests ensure that these functions still work.

Test #	Details	Example	Test data type	Expected result	Result
28	Create a valid task with subtasks and assigned tags	Title: Task Test 1 Content: Description Test Subtask 1: Sub1 Subtask 2: Sub2 Reminder: 20/04/2025 at 00:00 +assign any of the available tags	Normal	Task is successfully created and user is redirected to the Edit Task page	Pass
29	Rearrange subtasks	Rearrange subtasks using the drag and drop functionality	Normal	Subtasks rearranged on the Edit task section and on the Tasks page	Pass
30	Evaluate how tasks are displayed on the Task page and Edit pages on a small screen		Normal	Interface is scaled appropriately and each field is editable, it is possible to use drag and drop functionality to rearrange Subtasks	Pass
31	Delete a task		Normal	Task removed from the Tasks page	Pass
32	Create a task with a long title	Title: *100 character string* Content: Description Test Subtask 1: Sub3 Subtask 2: Sub4 Reminder: 20/04/2025 at 00:00 +assign any of the available tags	Boundary	Task is successfully created and user is redirected to the Edit Task page	Pass
33	Attempt to enter an incorrect time	Time: test	Erroneous	Non-number characters should not be allowed to be entered	Pass
34	Add an empty	Subtask:	Erroneous	An error should be displayed and	Pass

	subtask			save button disabled	
35	Edit a task to have an expired reminder	Reminder : 10/02/2025	Normal	The edit should be allowed and the reminder should be set to red on the Tasks page	Pass
36	Evaluate how the task is displayed on the Tasks page	Edit the task again to have a valid reminder	Normal	Task should be displayed with an appropriate title, tags, reminder, content and subtasks displayed	Pass
37	Complete tasks and subtasks	Click the checkboxes to complete subtasks	Normal	Completing all subtasks must automatically complete the task	Fail

## Test 28

### Result:



The screenshot shows the Neanote application interface. At the top, there is a header with the Neanote logo, a search bar, and user profile icons. Below the header, the main area is titled "Edit Task". Inside this area, there is a list item for "Task Test 1" with a checkbox and a delete icon. Below this, there is a text input field labeled "Description Test". Underneath the main task, there are two subtask entries, each with a checkbox and a delete icon: "Sub1" and "Sub2". At the bottom left of the main area, there is a button labeled "+ Add Subtask". To the right of the subtask list, there are three buttons: a trash can icon, an "Archive" button, and a "Save" button. A green notification bar at the bottom right of the main area contains a checkmark icon and the text "Task created successfully".

The task saved without any errors and I was redirected to the Edit page with an appropriate notification. Pass.

## Test 29

Using the handles on the left, I rearranged the subtasks, which is considered an edit, so the Save button became enabled.



## Result:

The screenshot shows the "Edit Task" page. At the top, there is a date field set to "April 20th, 2025" and a close button. Below the date field is a "Tags" button. The main area contains a title "Task Test 1" and a description "Description Test". Underneath the main task, there are two subtasks: "Sub2" and "Sub1". Each subtask has a checkbox, a delete icon, and a trash icon. At the bottom left is a "+ Add Subtask" button, and at the bottom right are three buttons: a trash icon, an "Archive" icon, and a "Save" icon.

The subtasks remain rearranged even after a page reload, so their new order has been saved successfully. Pass.

## Test 30

Going into the responsive development mode on my browser, I saw the buttons and the navbar shrink to fit the thinner display. Pressing the exit button at the top right of the page navigates us to the Tasks page, where the reminder has been replaced with an icon to give the title more space.

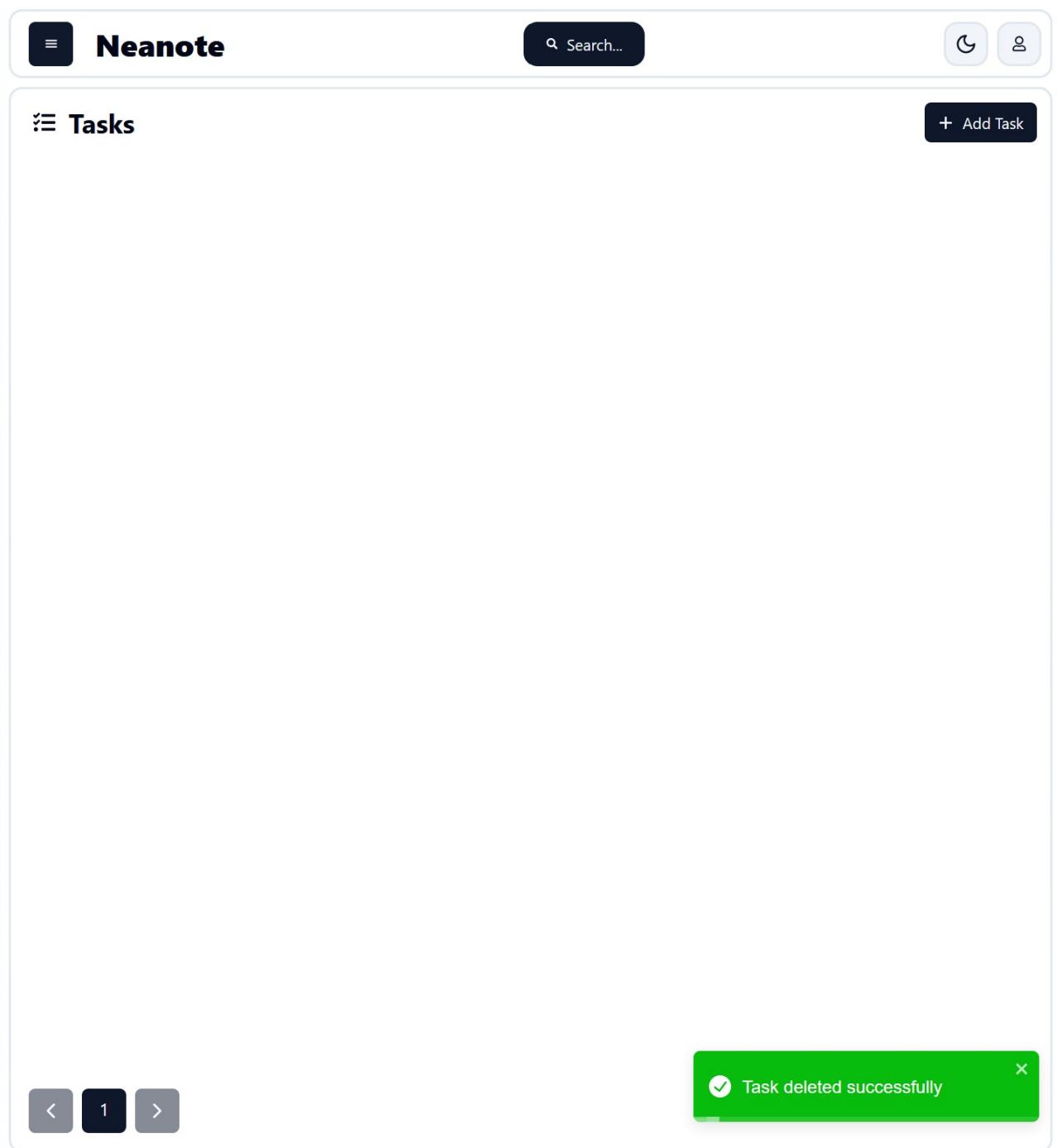
## Result:

The screenshot shows the "Edit Task" page on the left and the "Tasks" page on the right. The "Edit Task" page is identical to the previous screenshot, showing "Task Test 1", "Description Test", and subtasks "Sub2" and "Sub1". The "Tasks" page shows a single task "Task Test 1" with a description "Description Test". The subtasks "Sub2" and "Sub1" are listed below it. The "Tasks" page includes a "Tasks" header, a "+ Add Task" button, and icons for a bell, a blue bar, and a pen.

The interface is resized appropriately and the handles are still holdable to rearrange the subtasks. Pass

**Test 31**

**Result:**



Deleting a task removes it from the Tasks page and a success message is displayed.

### Test 32

For the title, I randomly generated the following 100 letter string:  
dtEfGsSgpqPZoOYKHocfahkRkHwNOytjCbSotWRknNtYCCnVpzaCFsvDktBdrEqgEMfZM  
TBbqwNxTqWXoATawdoeDKqFodBWNoKG

### Result:

317, Roman Sasinovich, 74561 IMS

**Neanote**

Search... ⟳ 👤

### Edit Task

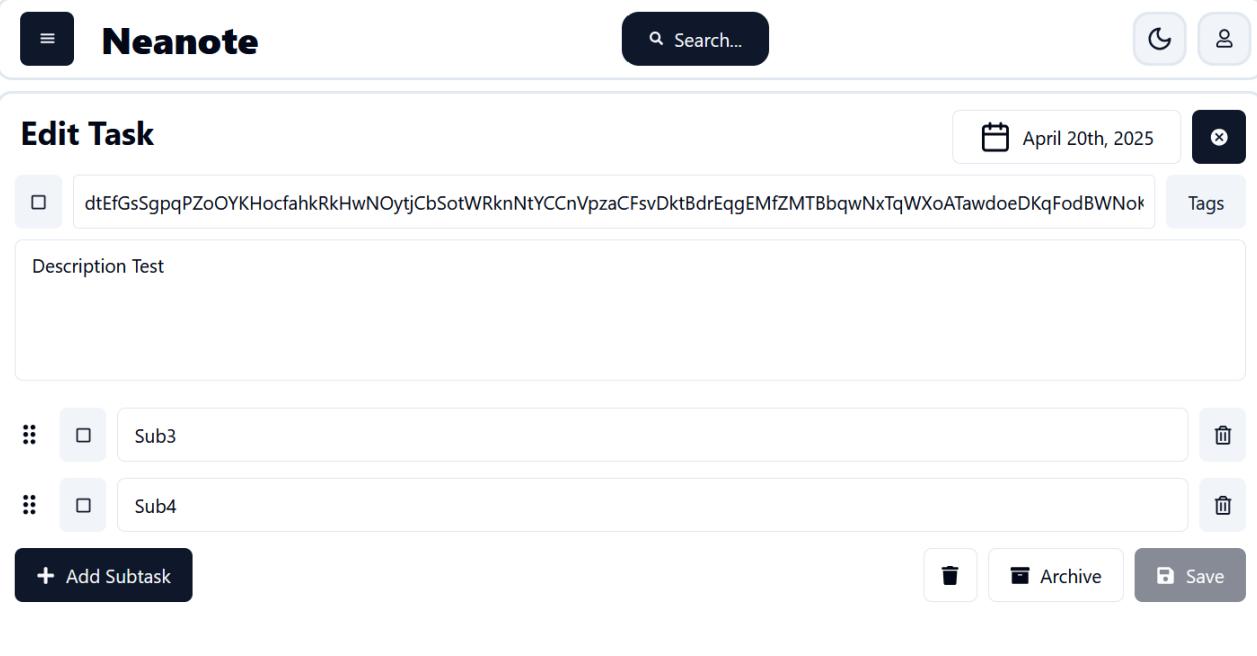
dtEfGsSgpqPZoOYKHocfahkRkHwNOytjCbSotWRknNtYCCnVpzaCFsvDktBdrEqgEMfZMTBbqwNxTqWXoATawdoeDKqFodBWNoK

Description Test

Sub3

Sub4

+ Add Subtask trash Archive Save



The task is allowed to be saved with this title, while adding one letter above it triggers a validation error. Pass

### Test 33

#### Result:

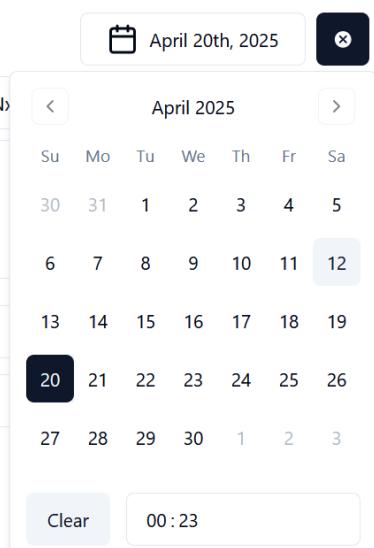
The nature of the input field does not allow any characters but numbers entered by definition, so a user cannot enter a string into the time input field:

April 20th, 2025

April 2025

Su	Mo	Tu	We	Th	Fr	Sa
30	31	1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	1	2	3

Clear 00:23



### Test 34

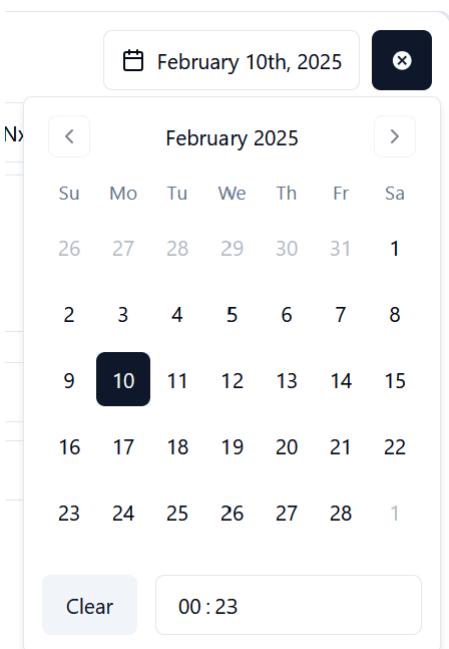
318, Roman Sasinovich, 74561 IMS

## Result:

The screenshot shows a task management interface. At the top, there are three input fields: a dropdown menu, a checkbox, and a text input field. Below these is a red error message: "Description is required". A large black button labeled "+ Add Subtask" is positioned below the input fields. To the right are three buttons: a trash can icon, an "Archive" icon, and a "Save" icon. The "Save" icon is currently grayed out.

Adding an empty subtask automatically triggers a validation error and disables the “save” button until a title has been added. Pass.

## Test 35



Using the calendar’s controls, I navigated to February and selected a random date. The task can be saved, as sometimes users may want to put projects that are past their deadline to remind them of it. Pass.

## Test 36

### Result:

## ☰ Tasks

+ Add Task

A screenshot of a mobile-style 'Tasks' application. At the top right is a dark button labeled '+ Add Task'. Below it is a list item with a small square checkbox and the truncated title 'dtEfGsSgpqPZoOYKHocfahkRkHwNOytjCbSotW...'. To the right of the title is a red rounded rectangle containing a bell icon and the text 'Feb 10, 2025, 12:23 AM'. Further to the right are two small blue icons: a checkmark and a pen. The main content area shows the truncated title again, followed by a horizontal line and a list of subtasks: 'Sub3' and 'Sub4', each preceded by a small square checkbox.

On the Tasks page, the reminder changes color from gray to red to signal that the task is past its due date, while the tag is compressed to give the title more screen real estate. The title is also shortened to avoid fetching the whole 100-character string, which cannot be viewed in full from the Tasks page anyways. Pass.

### Test 37

A screenshot of the same 'Tasks' application interface. It shows a list item with a checked square checkbox and the truncated title 'dtEfGsSgpqPZoOYKHocfahkRkHwNOytjCbSotW...'. To the right is a red rounded rectangle with a bell icon and the text 'Feb 10, 2025, 12:23 AM'. There are two small blue icons at the far right. The main content area shows the truncated title again, followed by a horizontal line and a list of subtasks: 'Sub3' and 'Sub4', each preceded by a small square checkbox. This screenshot is identical to the one above it.

For some reason, completing a single subtask resulted in the automatic completion of the whole task. In my browser's network inspector, I noticed that there are two requests sent to the backend: to complete the subtask and to complete the whole task. Turns out, an IF statement was omitted on the frontend which by accident called both endpoints even if not all subtasks were completed:

```
//call task api to toggle completeness of the whole task
if (all_completed) {
    await tasksApi.toggleCompleteness(taskId, null);
}
await tasksApi.toggleCompleteness(taskId, subtaskId);
```

## Tasks

+ Add Task

dtEfGsSgpqPZoOYKHocfahkRkHwNOytjCbSotW...

Feb 10, 2025, 12:23 AM



Description Test

Sub3

Sub4

A fix was implemented in **3.2.5.6.7**. Now, the main checkmark is only filled when both subtasks are completed:

## 4.6 Goals

Test #	Details	Example	Test data type	Expected result	Result
38	Create a valid goal with milestones and assigned tags	Title: Goal Test 1 Content: Description Test Milestone 1: Ms1 Milestone 2: Ms2 Reminder: 20/04/2025 at 00:00 +assign any of the available tags	Normal	Goal is successfully created and user is redirected to the Edit Goal page	Minor error discovered
39	Rearrange milestones	Rearrange milestones using the drag and drop functionality	Normal	Milestones rearranged on the Edit task section and on the Tasks page	Pass
40	Evaluate how goals are displayed on the Goals main page and Edit pages on a small screen		Normal	Interface is scaled appropriately and each field is editable, it is possible to use drag and drop functionality to rearrange Milestones	Pass
41	Delete a goal		Normal	Goal removed from the Goals page	Pass

42	Create a goal with long milestones and title	Title: *100 character string* Content: Description Test Milestone 1: *500 character string* Milestone 2: *500 character string* Reminder: 20/04/2025 at 00:00 +assign any of the available tags	Boundary	Goal is successfully created and user is redirected to the Edit Goal page	Pass
43	Complete a milestone and evaluate how the progress bar moves	One out of two milestones should be completed	Normal	The progress bar should move to roughly 50% of its total length	Pass
44	Add an empty milestone	Milestone:	Erroneous	An error should be displayed and save button disabled	Pass
45	Evaluate how goals are displayed on the main page and whether pagination works correctly	I will add around 6 more milestones to enable pagination	Normal	Each page should have different milestones displayed with correct information	Pass

## Test 38

The screenshot shows the Neanote application interface. At the top, there is a header with the Neanote logo, a search bar containing the placeholder "Search...", and two small circular icons. Below the header, the main title "Create Goal" is displayed. Underneath the title, there are two input fields: one for "Goal Test 1" containing the text "Goal Test 1" and another for "Description Test" containing the text "Description Test". To the right of these fields are two buttons: "Tags" and a date picker set to "April 20th, 2025" with a clear button next to it. Below the input fields, there are two milestone entries: "Ms1" and "Ms2", each with a delete icon to its right. At the bottom left is a button labeled "+ Add Milestone", and at the bottom right is a large "Save" button.

The goal comes pre-built with two default milestones, which must be used for calculating progress. To enter this data, I also tested an accessibility feature of modern browsers, that was integrated into my app, which allows me to press the Tab key on my computer and navigate through the buttons and inputs on the page without having to use my mouse.

**Result:**

## Edit Goal

April 19th, 2025



Goal Test 1

Tags

Description Test

Ms1



Ms2



+ Add Milestone



Archive

Save

The goal is created and an empty progress bar is spawned above the milestones. However, upon closer inspection, the date for some reason is set at a day before April 20. This error does not occur when similar steps are performed in the Tasks section.

Time	Domain	Path	Initiator	HTTP Method	Transfer Size	Request Headers	Request Cookies	Response Headers	Response Cookies	Timings	Stack Trace
2023-04-19T21:00:00.000Z	localhost	/create	axios.js	POST	760 B	Content-Type: application/json	Content-Type: application/json	Content-Type: application/json	Content-Type: application/json	2 ms	Raw
2023-04-19T21:00:00.000Z	localhost	/	xhr	GET	408 B	Content-Type: application/json	Content-Type: application/json	Content-Type: application/json	Content-Type: application/json	2 ms	Raw
2023-04-19T21:00:00.000Z	localhost	/goal?noteld=84bebfb	axios.js	POST	886 B	Content-Type: application/json	Content-Type: application/json	Content-Type: application/json	Content-Type: application/json	2 ms	Raw
2023-04-19T21:00:00.000Z	localhost	/goal?noteld=84bebfb	axios.js	POST	886 B	Content-Type: application/json	Content-Type: application/json	Content-Type: application/json	Content-Type: application/json	2 ms	Raw
2023-04-19T21:00:00.000Z	localhost	/goal?noteld=84bebfb	xhr	GET	399 B	Content-Type: application/json	Content-Type: application/json	Content-Type: application/json	Content-Type: application/json	2 ms	Raw
2023-04-19T21:00:00.000Z	localhost	/goal?noteld=84bebfb	xhr	GET	399 B	Content-Type: application/json	Content-Type: application/json	Content-Type: application/json	Content-Type: application/json	2 ms	Raw

Examining the POST request of a new goal shows that the due\_date is set at the 19<sup>th</sup> instead of the 21<sup>st</sup>, which signals an error on the frontend.

After a closer look, the error is in the timezones. Since I am currently on Eastern European Summer Time (GMT+3), I set the reminder at 00:00 of April 20 GMT+3, but in the database, time is stored in the GMT+0 format and then should be converted to the local timezone on the frontend, which doesn't happen.

The fix turned out very simple: I just had to add timezone information to each DATE string returned from my database and let the JavaScript's DateConstructor do its thing:

```
Date(response.data.due_date + 'Z')
```

Now, the reminder is set at the correct date:

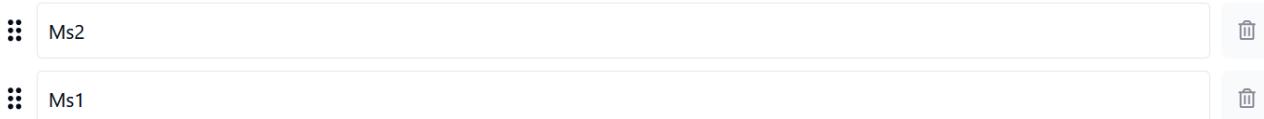
Apr 20, 2025

### Test 39

Milestones can be rearranged the same way as subtasks in the Tasks component: by using drag and drop handles.

**Result:**

Milestones are rearranged on the Edit Goal page:



And the “Next Milestone” of the Goal preview on the Goals page is switched from Ms1 to Ms2, as Ms2 is now first in the queue of milestones. Pass.

Next milestone

Ms2

## Test 40

Result:

The screenshot shows the Neanote application interface. On the left, the Goals page displays three goals: Goal Test 4, Goal Test 3, and Goal Test 2F. Each goal card includes a title, a progress bar, a 'Next milestone' section (with a checkbox and label), and a 'Description Test' section. Goal Test 4's next milestone is Ms1. Goal Test 3's next milestone is Ms1. Goal Test 2F's next milestone is Ms1. On the right, the Edit Goal page is open for Goal Test 4. It shows the goal title 'Goal Test 1', a description 'Description Test', and a milestones section. In the milestones section, Ms2 is listed above Ms1, indicating it is the next milestone. There are buttons for '+ Add Milestone', 'Archive', and 'Save' at the bottom right. The top right of the screen shows the date 'April 20th, 2025'.

The interface is appropriately readjusted and is convenient to use on a small screen. Pass.

## Test 41

### Result:

The DELETE request is successful and the goal is removed from the Goals page:

```
DELETE http://localhost:5000/api/goals/delete?goalid=2a30d8ed-ed50-4365-9d5b-da2083f18973&noteid=84beb6f0-8f58-495c-9288-3c2a91bf6290
```

JSON

```
message: "Goal deleted successfully"
```

## Test 42

### Result:

#### Edit Goal

The screenshot shows a 'Reminder' button, a close button, and tabs for 'Tags'. A large input field contains a long string of characters. Below it is a text area labeled 'Describe your goal here'. Underneath these are two milestone sections, each with a delete icon. The first section has a red validation message: 'Milestone description cannot exceed 500 characters'. At the bottom are 'Add Milestone', 'Archive', and 'Save' buttons.

dtEfGsSgpqPZoOYKHocfahkRkHwNOytjCbSotWRknNtYCCnVpzaCFsvDktBdrEqgEMfZMTBbqwNxTqWXoATawdoeDKqFodBWNoKG

Describe your goal here

IoKGdtEfGsSgpqPZoOYKHocfahkRkHwNOytjCbSotWRknNtYCCnVpzaCFsvDktBdrEqgEMfZMTBbqwNxTqWXoATawdoeDKqFodBWNoKG

QGdtEfGsSgpqPZoOYKHocfahkRkHwNOytjCbSotWRknNtYCCnVpzaCFsvDktBdrEqgEMfZMTBbqwNxTqWXoATawdoeDKqFodBWNoKG A

Milestone description cannot exceed 500 characters

+ Add Milestone

Archive Save

Goal is successfully created, while any extra character in the milestones or title inputs results in a validation error.

## Test 43

If I click the checkbox on any of the milestones on the Goal Preview page, it is completed and the progressbar moves. This goal has only two milestones, so the progress bar is half full.

### Result:

The screenshot shows a 'Goal Test 2F' title, a date 'Apr 16, 2025', and a checkbox. Below is a progress bar with a dark blue segment. To its right is a 'Next milestone' section with a checkbox labeled 'Ms 2'. At the bottom is a 'Description Test' section.

Goal Test 2F

Apr 16, 2025

Next milestone

Ms 2

Description Test

Pass.

## Test 44

326, Roman Sasinovich, 74561 IMS

## Result:

### Edit Goal

The screenshot shows a user interface for editing a goal. At the top right are buttons for saving (blue), canceling (red), and deleting (black). Below them is a date field showing "April 16th, 2025". A "Tags" button is also present. The main area contains a title "Goal Test 2F" and a description "Description Test". A progress bar at the bottom is mostly black, indicating low completion. Below the bar is a list of milestones: "Ms 1", "Ms 2", and "Milestone 2", each with a delete icon. A "Save" button at the bottom right is disabled.

Goal Test 2F

Description Test

Ms 1

Ms 2

Milestone 2

+ Add Milestone

Archive

Save

Adding a new milestone automatically recalculates progress (now it is 1/3 of the total length of the bar), but the Save button is disabled until the milestone is given a title. Pass.

## Test 45

### Result:

When the total number of goals exceeds 5, pagination is used to avoid fetching all goals at once. Instead, the bottom pagination controller enables its buttons to allow the user to switch between pages. Clicking on the buttons sends requests for different pages to the backend, which returns different goals for each page as shown below. The pagination controller also shows the current page.

GET	localhost:5000	previews?page=1
OPTIONS	localhost:5000	previews?page=1
GET	localhost:5000	previews?page=2
OPTIONS	localhost:5000	previews?page=2



## Goals

[+ Add Goal](#)

### Another goal



Next milestone

 12123

### Goal 20555124



Next milestone

 141241234

### Goal



Next milestone

 yet another Milestone

### Goal Test



Next milestone

 Milestone 1

### Goal Test 2F

Apr 16, 2025



Next milestone

 Ms 2

Description Test

The screenshot shows the Neanote application interface. At the top, there is a header bar with the Neanote logo, a search bar containing the placeholder "Search...", and three icons: a refresh symbol, a user profile, and a settings gear.

The main content area is titled "Goals" and displays a single goal entry:

- Goal Test 1** (bolded title)
- Next milestone:  Ms2
- Description: Description Test
- Timestamp: Apr 20, 2025
- Status: A small checkmark icon is present next to the timestamp.

At the bottom of the screen, there is a navigation bar with three buttons: a left arrow, the number "2", and a right arrow.

Pass.

## 4.7 Habits

Test #	Details	Example	Test data type	Expected result	Result
47	Create a habit	Title: Habit Test 1 Content: Description Test Repeats: daily at 5	Normal	Habit is created, a success message is displayed and the user is redirected from the creation page to the edit page	Pass
48	Edit a habit	Title: Habit Test 2 Content: Description Test 2	Normal	Habit edited successfully	Pass
49	Complete a habit		Normal	The streak should be increased by 1 and the checkbox should be disabled until the next day	Pass
50	Evaluate how a habit is displayed on the main Habits page		Normal	The habit with updated information and completed checkbox should be seen on the Habits page	Pass

### Test 47

#### Create Habit

Habit Test 1

Description Test

Repeats:

- 17:00
- Daily
- Weekly
- Monthly

Save

The “Repeats” dropdown allows a user to select daily, monthly and weekly modes and input a time for the checkbox to unlock

**Result:**

## Edit Habit

Habit Test 1

Description Test

Archive Save

Repeats:



Tags

The habit was created and a checkbox appeared, which would allow the user to complete the habit. Pass.

## Test 48

### Result:

## Edit Habit

Habit Test 2

Description Test 2

Archive Save

Repeats:



Tags

Editing works great as well, with the save button turning on and off whether there are any validation errors or changes made to the habit. Pass.

## Test 49

## ⌚ Habits

+ Add Habit

Habit Test 2

Streak: 0

Description Test 2

On the Habits page, a streak is also displayed.

**Result:**

## ⌚ Habits

+ Add Habit

Habit Test 2

Streak: 1

Description Test 2

After the checkbox button is pressed, the streak is highlighted and the checkbox is disabled until 5pm the next day. The checkbox is disabled on the Edit Habits page too.

On the next day, the streak remains at 1 but the checkbox is enabled, allowing to press it again to update the streak:

## ⌚ Habits

+ Add Habit

Habit Test 2

Streak: 1

Description Test 2

 Neanote



+ Add Habit

## ⌚ Habits

+ Add Habit

Streak: 2

Habit Test 2

Streak: 2

Description Test 2

Pass.

## Test 50

**Result:**

As seen in the previous tests, the habit is properly displayed on the Habits page in both landscape and mobile cases:

}

The screenshot shows the Neanote application interface. At the top, there is a header with the logo 'Neanote' and a search bar labeled 'Search...'. To the right of the search bar are two small icons: a refresh symbol and a user profile symbol. Below the header, the title 'Habits' is displayed next to a back arrow icon. On the far right, there is a button labeled '+ Add Habit'. The main content area contains a single habit entry: 'Habit Test 2' with a checked checkbox, a description 'Description Test 2', and a green button labeled 'Streak: 2' with a checkmark icon. At the bottom of the screen, there are three navigation buttons: '<' (left), '1' (center), and '>' (right).

Pagination data is also returned with the GET request, so pagination works as well:

```
JSON
▶ data: [ {...} ]
  message: "Habit previews fetched successfully"
▶ pagination: Object { page: 1, per_page: 5, total: 1, ... }
```

Pass.

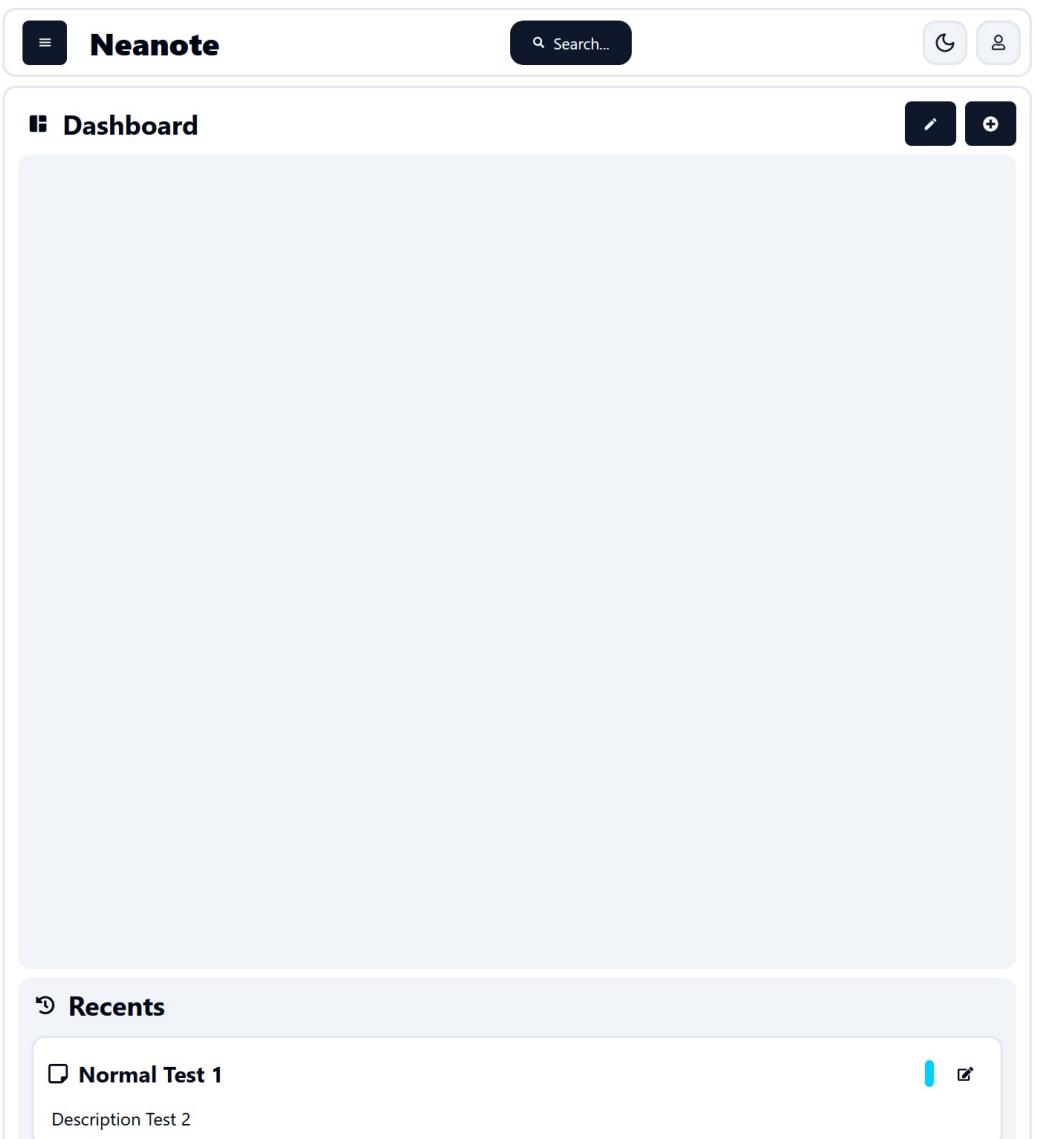
## 4.8 Widgets

Test #	Details	Example	Test data type	Expected result	Result
51	Create a Chart widget	Title: Chart1 Datasource: goals	Normal	A chart widget is created with the last bar having some value in relation to completed goals	Pass
52	Create a Progress widget	Title: Progress2 Datasource: any of the goals	Normal	A progressbar that is as full as the progressbar of a related goal	Pass
53	Create a Number widget	Title: Number3 Datasource: habit	Normal	A number widget containing the streak of the selected habit	Pass
54	Create a HabitWeek widget	Title: Week4 Datasource: habit	Normal	A week widget with two last days filled out depending on the streak of the habit selected	Pass
55	Trigger an update on the widgets		Normal	Where possible, the widget numbers/values are incremented	Pass
56	Attempt to create widgets with empty titles or with long titles	1: Title: an empty string 2: Title: *some long string exceeding 50 characters*	Erroneous	The strings are not allowed to be entered	Pass
57	Rearrange widgets within their own column and between columns	This can only be done in edit mode	Normal	The widgets are moved around and snapped into gridcells on the dashboard	Pass
58	Attempt to drop a widget outside of the page		Erroneous	The widget should return back to the grid	Pass

59	Delete a widget		Normal	The widget should be removed from the Dashboard	Pass
60	Evaluate how the dashboard acts on different screens		Normal	Widgets should be reassembled into a single column on a small screen and more columns should be created on large screens	Pass

### Test 51

To add any widget, we navigate to the Dashboard page and press the plus icon at the top right corner:



This will open a drawer menu, which has several widgets a user can add:

335, Roman Sasinovich, 74561 IMS

## Add widgets

Select a widget type and source

### Chart

Visualise your task completion rates.



### Progress

Track your progress.

#### Investments



### Number

Display your streaks and averages.

**25**

Habit streak

### Habit Week

View your weekly habit progress.



Each widget is clickable and leads to a setup page. For this test, I will add a chart widget.

## Add widgets

Select a widget type and source

### Setup Chart Widget

Title:

Widget title

Select a data source

Cancel

Save

The following prompt appears, where I will unput the title and select goals as the datasource from the dropdown menu. Only then will the Save button enable, allowing me to add a widget.

## Add widgets

Select a widget type and source

### Setup Chart Widget

Title:

Chart1

goal (6 items)

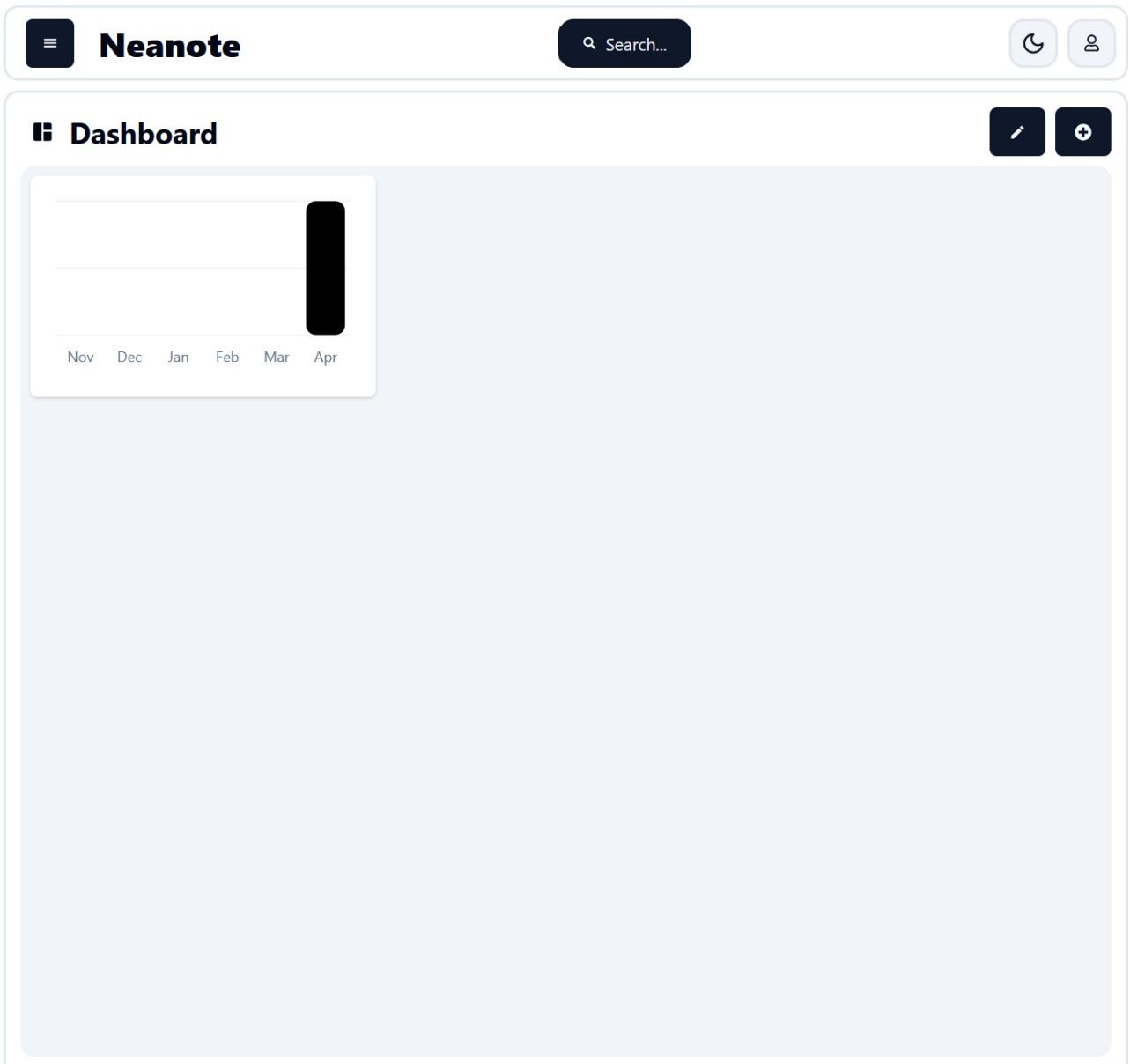
Cancel

Save

## Result

336, Roman Sasinovich, 74561 IMS

The widget is added to the grid, containing a single bar in April. Hovering over the bar shows the value of 1, as I only have fully completed a single goal from the goals added in the previous tests.



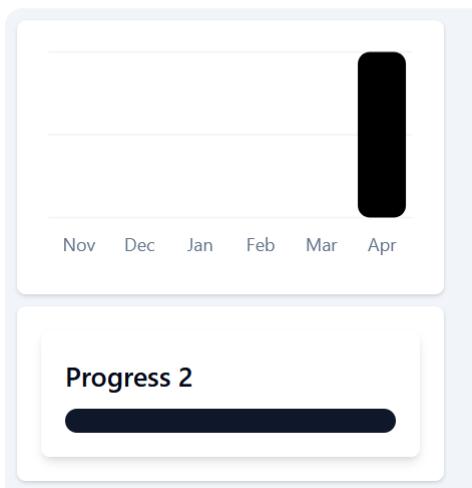
Pass.

\*I will avoid this process for the next tests and just show how the widget would look on the dashboard.

## Test 52

### Result

## ■ Dashboard

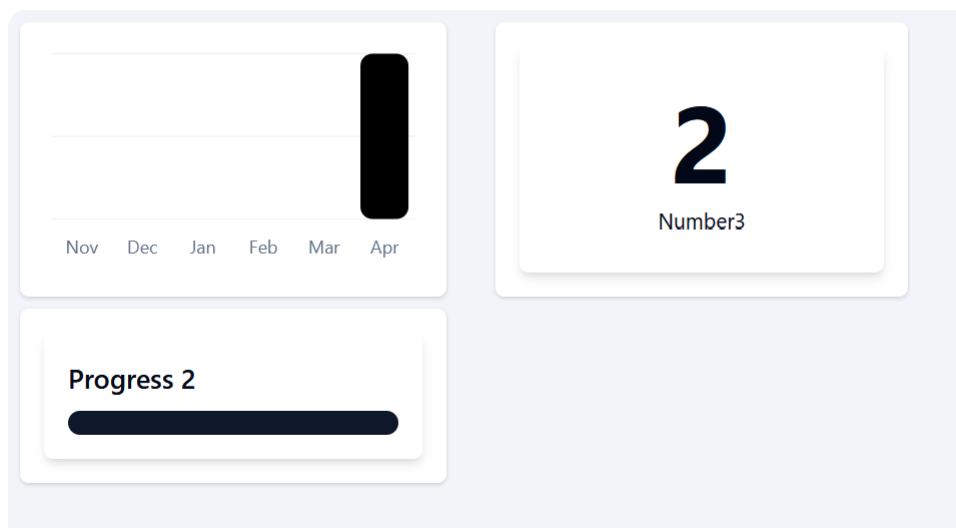


A completed progress bar is added to the dashboard, since the source goal had all of its milestones completed. Pass.

### Test 53

#### Result

## ■ Dashboard



The widget with a number matching the streak on the Habit mentioned previously is added. Pass.

### Test 54

#### Result

338, Roman Sasinovich, 74561 IMS

A HabitWeek widget is added with the two last days filled up, as the connected habit has been completed for the last two days in a row. Pass.



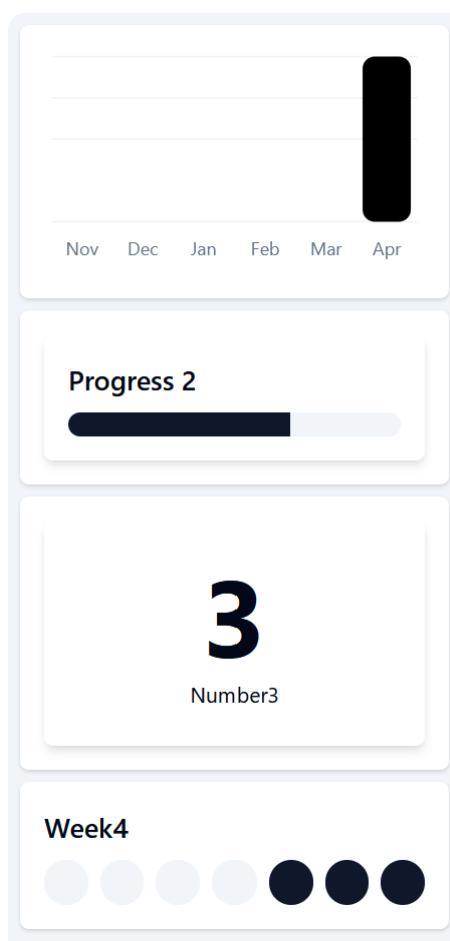
## Test 55

To trigger updates, I had to do a few things on the app.

For the Chart and Progress widgets, I added a new milestone to a goal that Progress widget is related to in order to move the progress bar back and completed another goal. For habit-related widgets, I ran the following SQL command to artificially add another day to the streak :

```
insert into habitcompletion  
values('ef0a1171-1e16-4d0b-9247-6be0cc63472d', '2025-04-12')
```

## Result



The Chart widget is visually unchanged, but hovering over the column shows "2" instead of "1", as two goals are completed.

The progressbar is now at 2/3, since there are 3 total milestones now, while the habit related widgets have been incrementer or another day has been added to the week, when the habit was completed. Pass.

## Test 56

## Result

The `WidgetSetup` component does not allow to create a widget if the title is empty:

**Add widgets**

Select a widget type and source

**Setup Progress Widget**

Title:

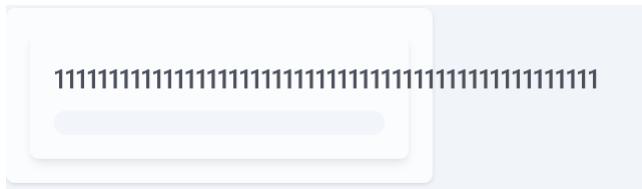
Widget title

Select a data source

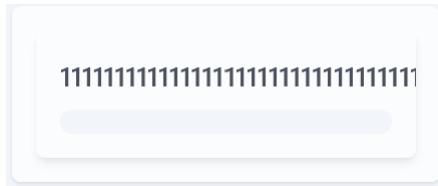
Cancel Save

If I continue adding characters, the input will stop accepting any new values at 50 characters, which is the maximum allowed character count.

However, the boundary value of 50 results in the title going out of bounds of the widget:



This has a simple fix: adding an `overflow-clip` inline css to the title of the widget's styling, which removes any overflowing characters:

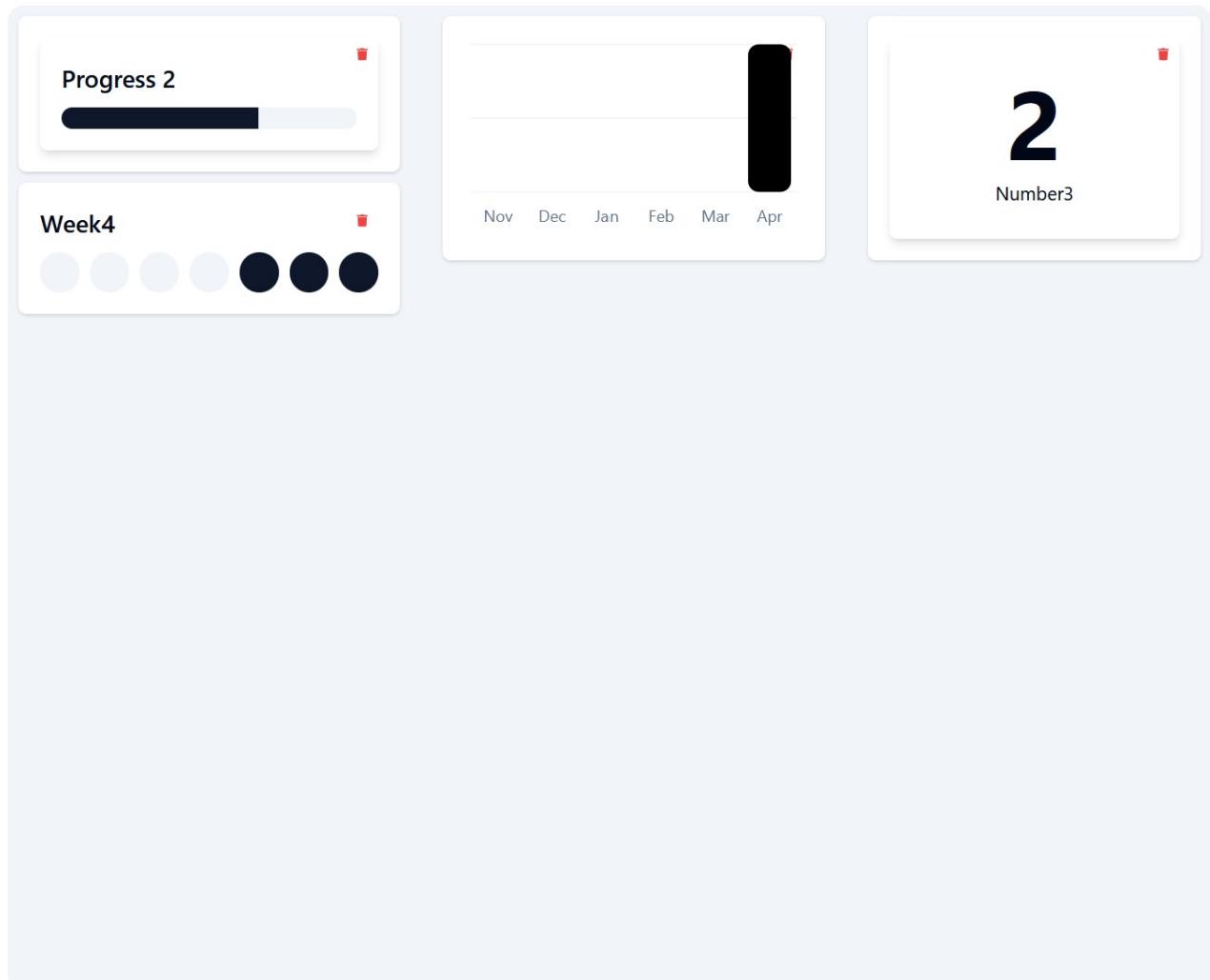


## Test 57

The edit mode button, located next to the “Add widget” button, turns on the drag and drop functionality of widgets. It allows the user to drag the widgets between columns and arrange them within a column into a design of their choice:

## Result

## Dashboard



You may also watch a quick video showcase of this and the next tests [here](#).

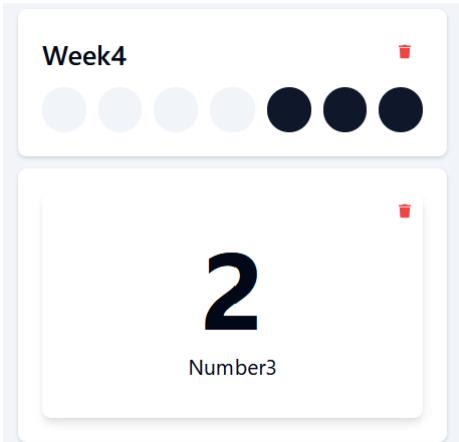
### Test 58

#### Result

As can be seen from the video, the Number widget that was dragged out of bounds is returned to the nearest column. Pass.

### Test 59

The edit mode displays small delete icons on each widget:



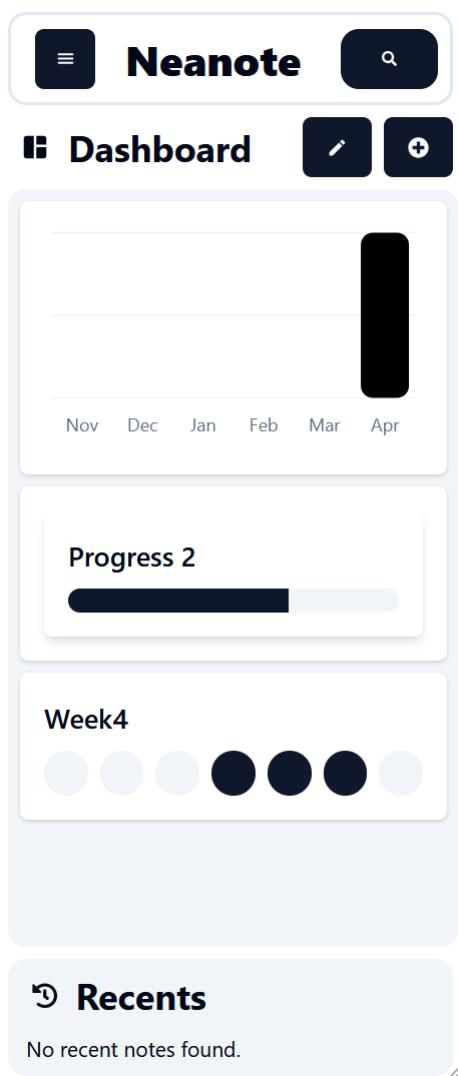
## Result

The image displays the dashboard view of the Neanote application. At the top left is the logo "Neanote". To its right is a search bar with the placeholder "Search...". On the far right are three icons: a circular arrow, a user profile, and a plus sign. Below the header, there are several cards arranged in a grid. One card on the left is titled "Progress 2" and shows a progress bar with a dark blue segment. Another card on the right is titled "Week4" and features a horizontal progress bar with seven circles, where the last three are dark blue. In the center, there is a large card with a blank lined page and a vertical black sidebar. At the bottom of this central card, there is a timeline with months: Nov, Dec, Jan, Feb, Mar, Apr. The overall interface has a clean, modern design with a light gray background and white cards.

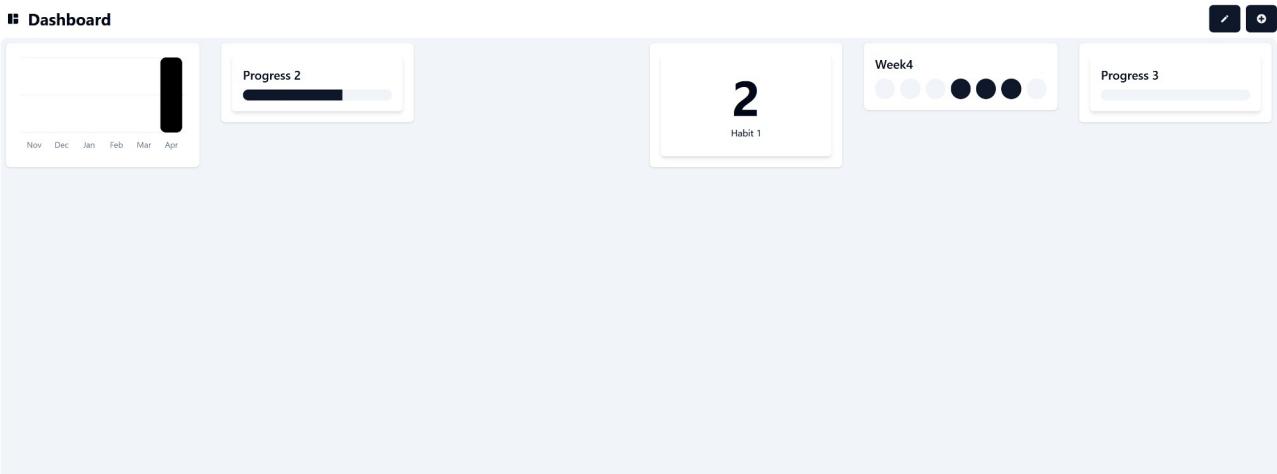
Pressing the button removes the widget with an appropriate message. Pass.

## Test 60

**Result:**

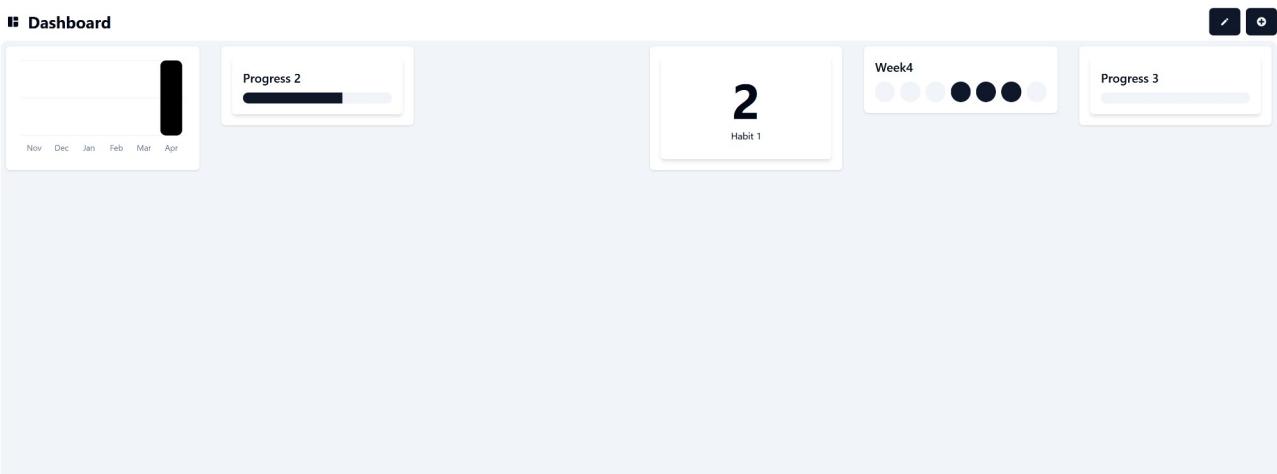


On a small screen, there is only one column, while on large displays, widgets can be spread out through more columns:



Pass.

## 4.9      Calendar



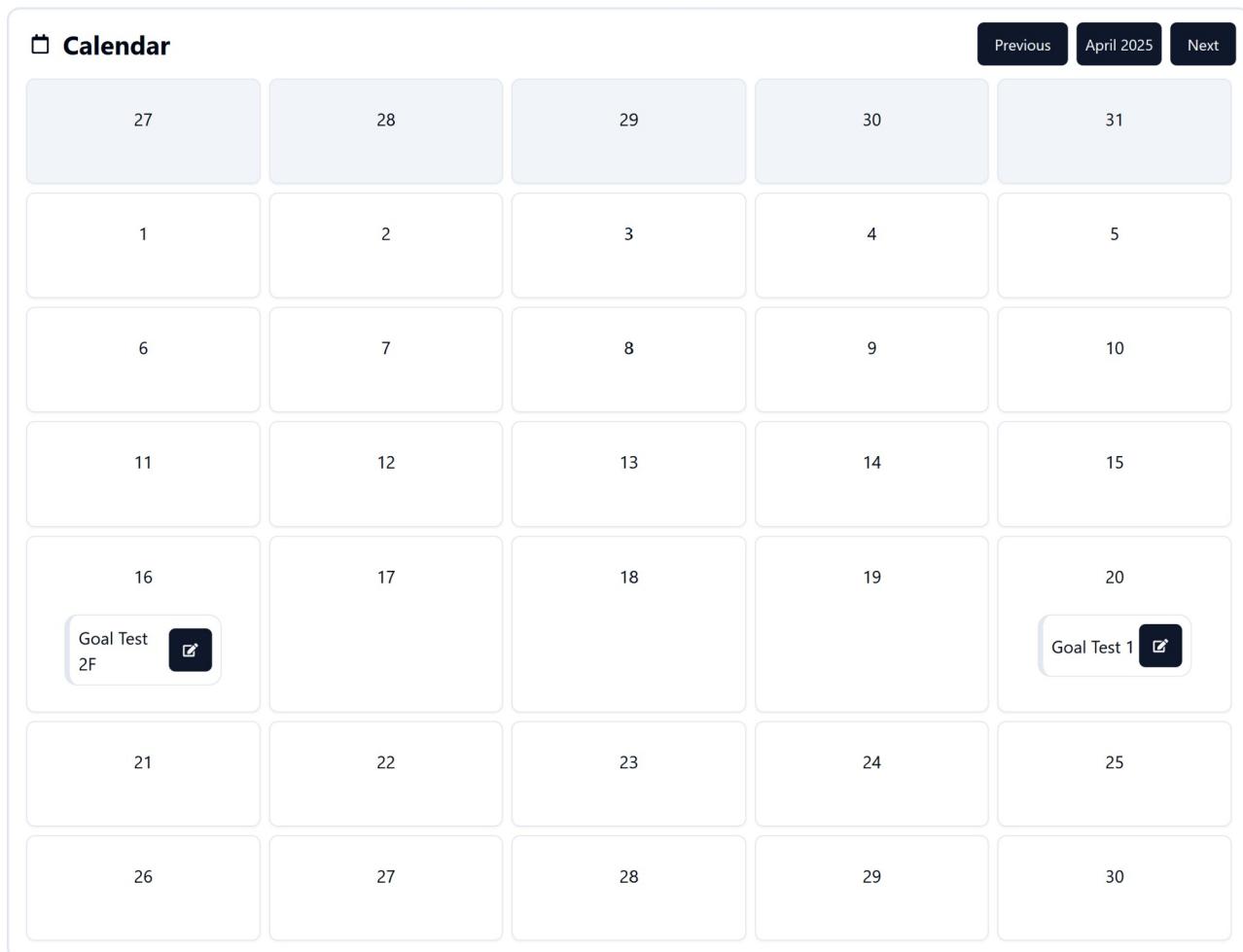
## 4.10     Archive

Test #	Details	Example	Test data type	Expected result	Result
61	Evaluate how notes with deadlines are shown on the calendar		Normal	Notes with deadlines are shown on appropriate days in the calendar	Pass
62	Evaluate whether a correct number of days is shown on		Normal	The number of days in a month matches is correct	Pass

	the calendar				
63	Check if notes are clickable		Normal	A click should redirect the user to the appropriate note	Pass

## Test 61

### Result

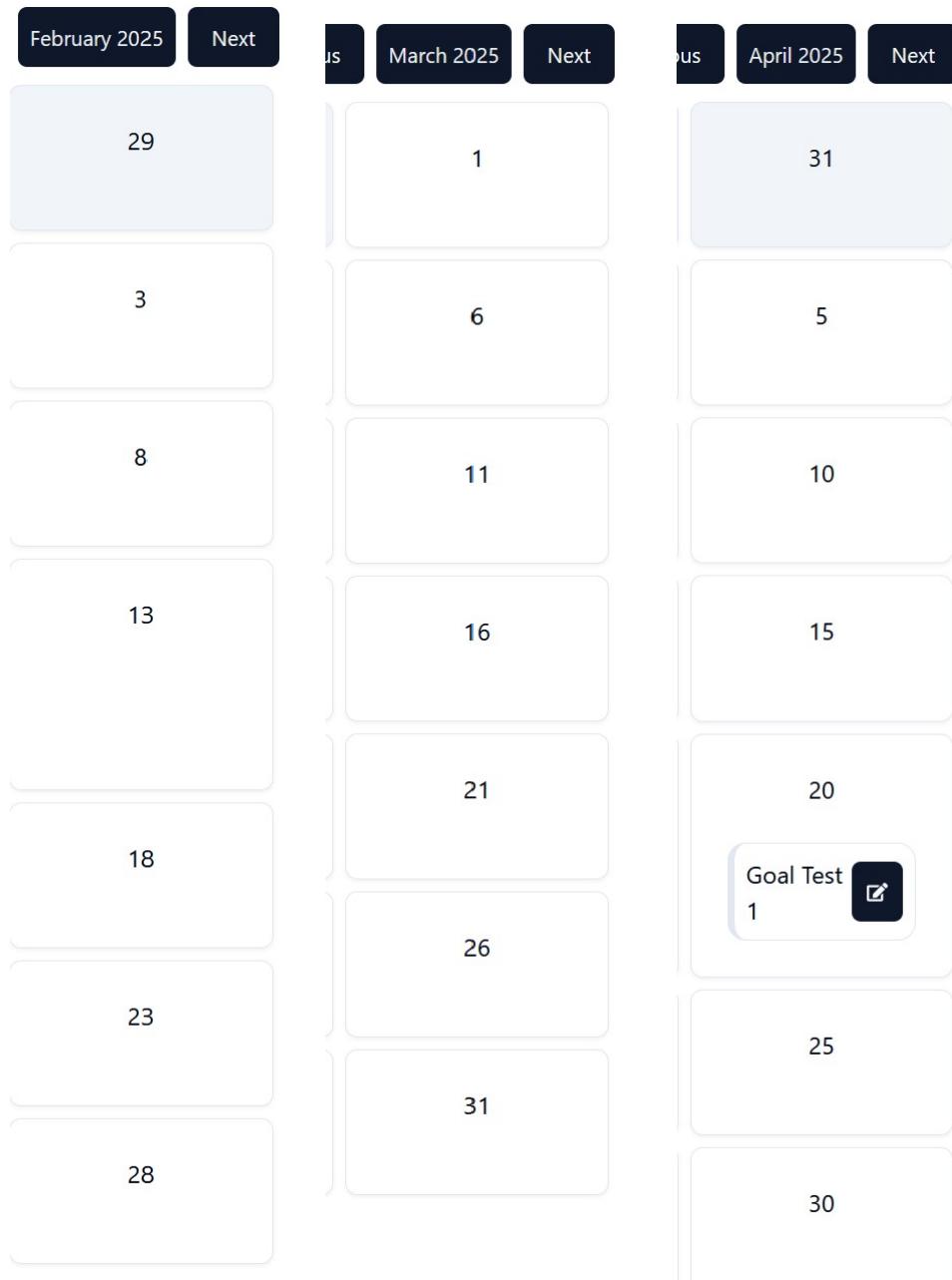


The two goals that have their due dates set in April are shown on the April page and their titles are displayed correctly. Pass.

## Test 62

Using the “Previous” and “Next” buttons, the user can navigate between months and the number of days in a month is changed accordingly. For example, February has 28 days, while April has 30.

### Result



Pass.

### Test 63

Each note has a button, which, upon clicking, redirects the user to the edit page of the note.

### Result

Goal Test 1

Description Test

Ms2

Ms1

+ Add Milestone

Archive

Save

In this case, since it is a goal, the link to the edit page contained /goals/ : [localhost:5173/goals/edit](http://localhost:5173/goals/edit)

Pass.

## 4.11 Miscellaneous

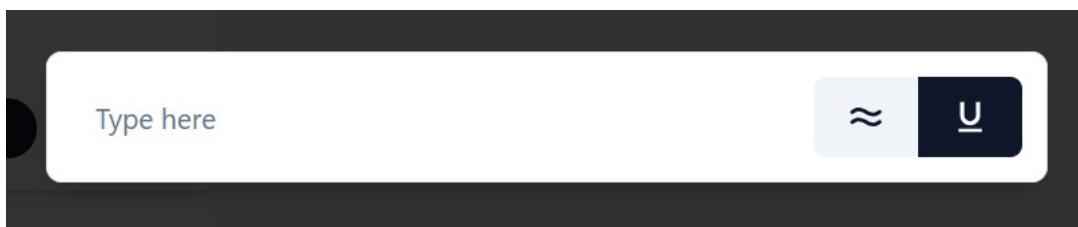
### 4.11.1. Search

Test #	Details	Example	Test data type	Expected result	Result
64	Search one character	Search: a	Erroneous	A search will not occur since this is below the limit for characters	Pass
65a	Search for 3 characters with results	Search: goa	Boundary	The “Exact” search mode returns a list of notes that have “goa” in them	Pass
65b	Search without results	Search: rrr	Erroneous	No results are shown since no notes contain “rrr”	Pass
66	Search for a specific note	Search: Goal Test 2F	Normal	Only one note is returned	Pass
67	Evaluate pagination		Normal	Different notes are shown on each page	Pass

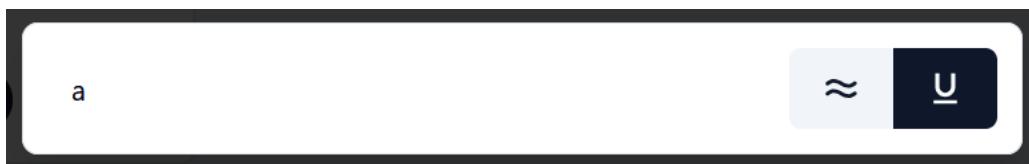
68	Test context-based search	Create a note talking about a topic and use one words that are similar to the topic e.g. “movies” and “cinema” to search	Normal	The search should return the note talking about movies first	<b>Pass</b>
----	---------------------------	--	--------	--	-------------

### Test 64

The search button is always on the navbar, on any screen resolution. A click on it will open a search popup, which defaults in “exact” search mode.



### Result:



Entering one character does nothing since the search is too ambiguous. Pass.

### Test 65a

Now two or more characters can be used more effectively during searching, so entering “goa” would return all notes that have these characters in it, such as goals, which have the word “goal” in the title.

### Result:

goa ≈ U

**Goal Test 2F**

Description Test

**Goal Test 1**

Description Test

**Goal Test**

**Goal 20555124**

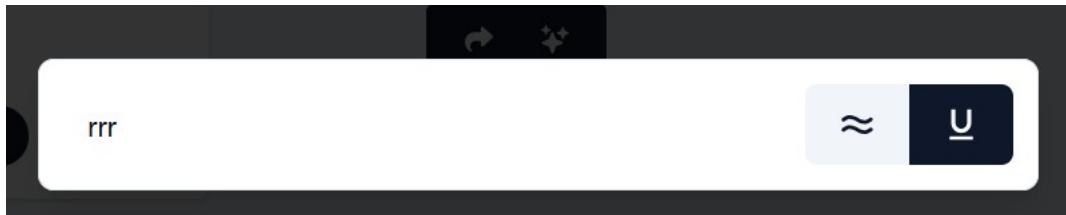
**Another goal**

< 1 >

Several notes have been returned in multiple pages. Pass

### Test 65b

**Result:**

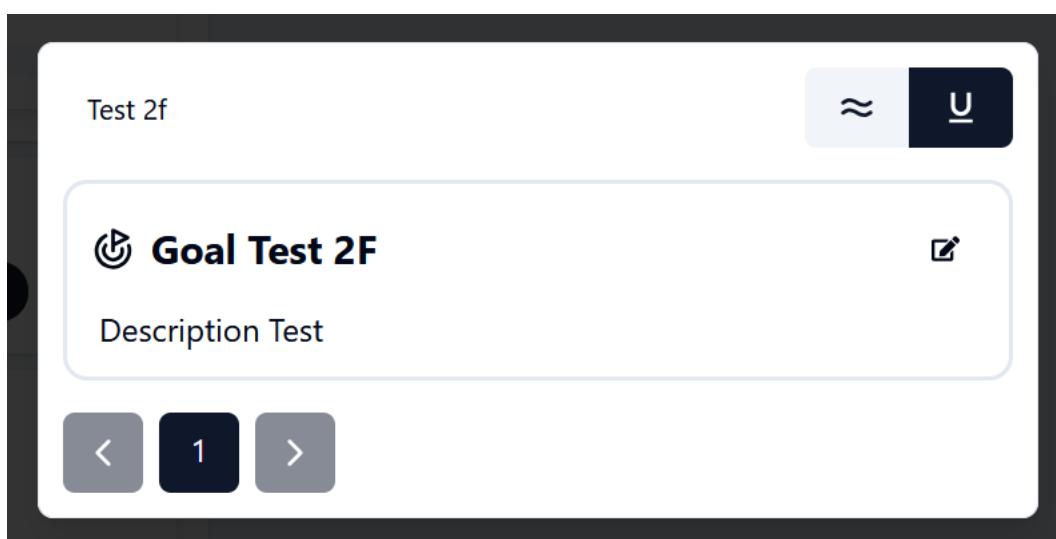


An exact search with no results does not show any notes. Pass.

### Test 66

The search mode is note case sensitive, so the note is found even though the search string does not exactly match the note.

#### Result:

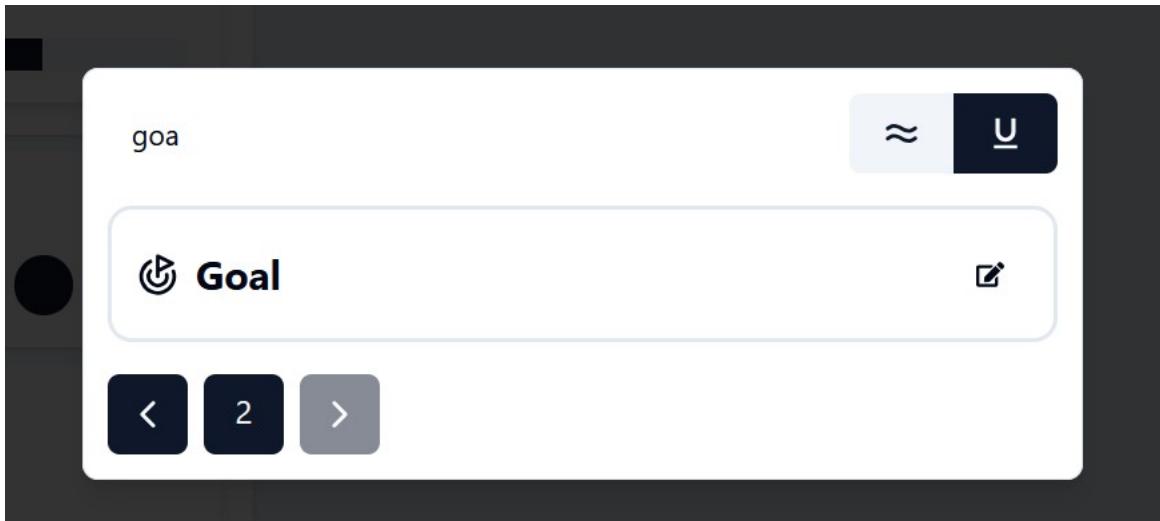


Pass.

### Test 67

Using the same search query as in 65a and clicking the “Next page” icon button fetches the next page of results:

#### Result:



Each result is unique to the page and page parameters are correctly send alongside the searchmode in the GET query to the backend:

Status	Method	Domain	File	Initiator	Type	Transferred
200	GET	localhost:5000	search?searchQuery=goa&searchMode=exact&pageParam=1	axios.js:1646 (xhr)	json	1.55 kB
200	OPTIONS	localhost:5000	search?searchQuery=goa&searchMode=exact&pageParam=1	xhr	html	399 B
200	GET	localhost:5000	search?searchQuery=goa&searchMode=exact&pageParam=2	axios.js:1646 (xhr)	json	628 B
200	OPTIONS	localhost:5000	search?searchQuery=goa&searchMode=exact&pageParam=2	xhr	html	399 B

Pass.

## Test 68

For this test, I asked Google's Gemini large language model to generate me two notes talking about going to the cinema without directly mentioning the word "movies":

### Note 1 (Past):

Just thinking about that night we went to the picture house last month. Remember how packed it was? The big screen really made the whole spectacle so immersive. Afterwards, grabbing ice cream and dissecting all the scenes was the perfect end to the evening. We should do that again sometime.

### Note 2 (Future):

Hey! Are you free next Saturday evening? I was thinking of catching the latest showing at the multiplex. They're featuring that new sci-fi presentation everyone's been talking about. Let me know if you're interested in experiencing it on the big screen!

... And then added them to the Notes section of my app.

## Cinema plans

edit

Hey! Are you free next Saturday evening? I was thinking of catching the latest showing at the multiplex. They're featuring that new sci-fi presentation everyone's been talking about. Let me know if yo...

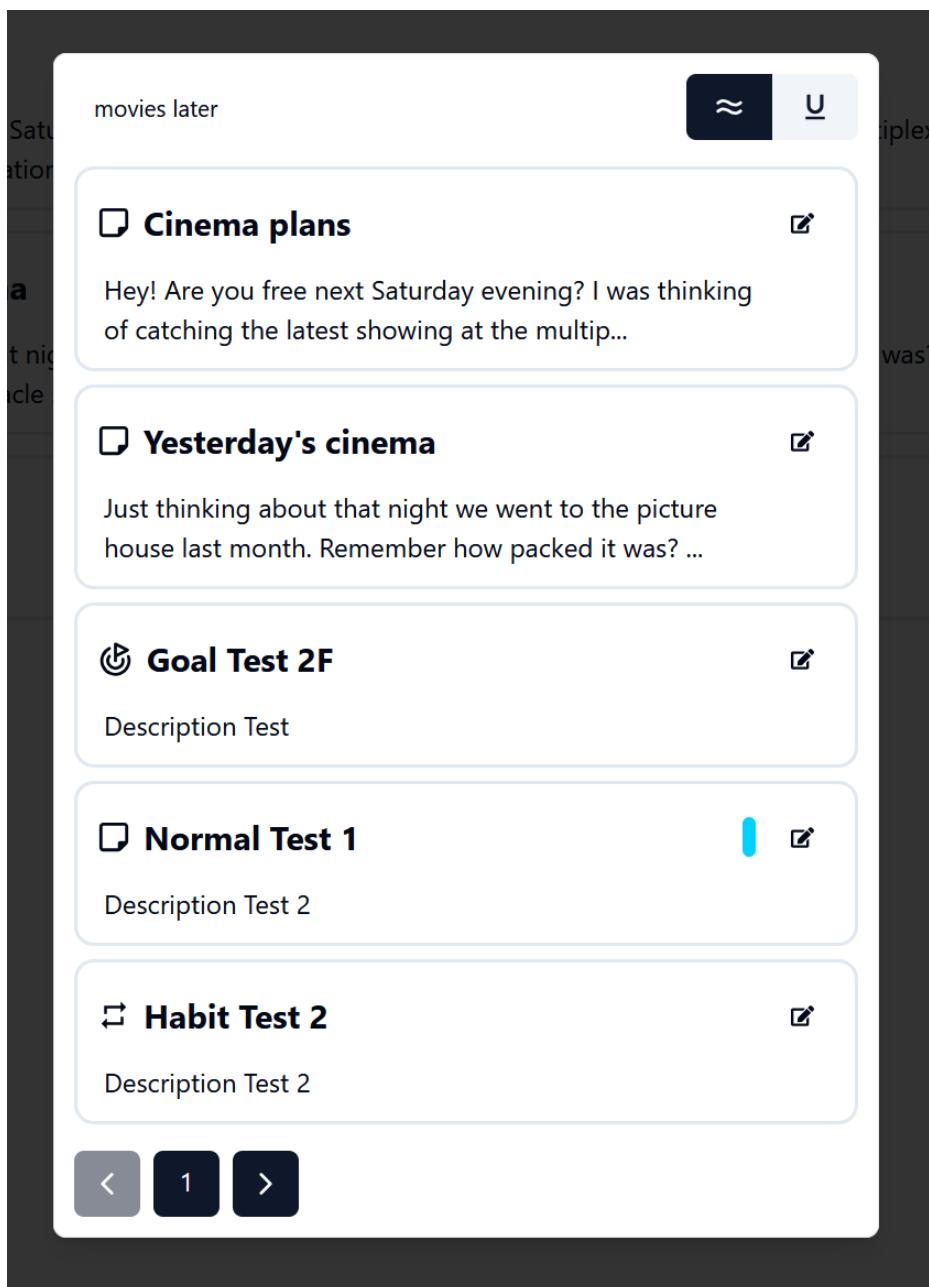
## Yesterday's cinema

edit

Just thinking about that night we went to the picture house last month. Remember how packed it was? The big screen really made the whole spectacle so immersive. Afterwards, grabbing ice cream and diss...

I created the notes and waited for them to be vectorized on the backend.

**Result:**



The “approximate” search move thrives off context. The more information about the searched note is in the search query and the note itself, the more likely it is for the search to be meaningful. In this case, the word “movies”, despite not being mentioned directly in any of the notes, was related to “cinema”, “pictures” and other words, since their vectorized representations are similar. Then, the word “later” helped bring the note about future plans, giving the note about cinema plans greater relevance in this context. Meanwhile, the other notes were not as relevant to this search, so they were not placed as high up as “cinema” notes. Pass.

#### 4.11.2. Recents

Test #	Details	Example	Test data type	Expected result	Result
69	Check if recent notes are displayed		Normal	A list of notes accessed in a chronological order is shown	Pass
70	Change order of recent notes by clicking on the oldest note		Normal	The accessed note is sent to the top of the list	Pass

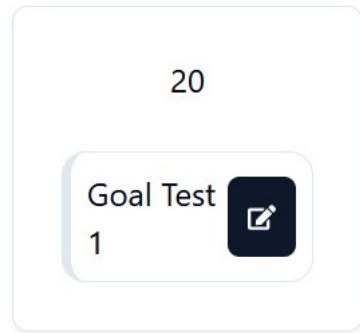
#### Test 69

On a fresh start of the web app, when the user has not yet clicked on any notes, there are no recent notes displayed:

##### ⌚ Recents

No recent notes found.

After going through some notes by clicking the “edit” button for the note on the Calendar, in the Searchbar or on the page of a corresponding note type, the visited notes are added to the Recents. To ensure that notes visited from any components are added to the server, here are the buttons I clicked in chronological order:



## ⌚ Goals

+ Add Goal

### Another goal

Next milestone

another one

cinema

≈

U

### Yesterday's cinema

Just thinking about that night we went to the picture house last month. Remember how packed it was? ...

This test encompasses notes opened from the calendar, the note type page and the search menu.

## Result

## ⌚ Recents

### Yesterday's cinema



Just thinking about that night we went to the picture house last month. Remember how packed it was? ...

### Another goal



### Goal Test 1



Description Test

The list correctly displays the notes clicked alongside an icon that suggests their note type. The most recently accessed note is on the top. Pass.

## Test 70

To update the list of recent notes, I clicked on the last note in the list titled “Goal Test 1”

## Result

## ⌚ Recents

### Goal Test 1



Description Test

### Yesterday's cinema



Just thinking about that night we went to the picture house last month. Remember how packed it was? ...

### Another goal



This brought it up in the list, keeping the order of the remaining notes.

### 4.11.3. Archive

Test #	Details	Example	Test data type	Expected result	Result
71	Check if archived notes are		Normal	Notes are displayed with all relevant	Fail

	displayed			information, including an indication of their note type alongside restore or delete buttons	
72	Restore a note		Normal	The note should be returned to its category's page	Pass
73	Delete a note		Normal	The note should be completely removed	Pass

## Test 71

If a user has no archived notes, loading the page threw a backend error.

```
"GET /api/notes/archive?pageParam=1 HTTP/1.1" 500 -
"GET /api/notes/archive?pageParam=1 HTTP/1.1" 500 -
```

A closer inspection led to a discovery that if there are no rows fetched from the database, then the variable notes being accessed in the return statement of the function will not exist:

```
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
```

To rectify this, all that had to be done is to define notes as an empty list (notes = []) somewhere in the file. Now loading the page if a user has no archived notes returns a success code 200 and an empty list:

```
"GET /api/notes/archive?pageParam=1 HTTP/1.1" 200 -
```

With this error fixed, its time to determine whether archived notes are actually displayed. To do this, I clicked the “Archive” button on one note from each note type.

### Result:

The screenshot shows the Neanote application interface. At the top, there is a header bar with the Neanote logo, a search bar containing the placeholder "Search...", and two small circular icons. Below the header, the title "Archive" is displayed next to a folder icon. The main content area contains four cards, each representing a different note type:

- Normal Test 1**: Represented by a blue square icon. The card contains the text "Description Test 2". To the right are three buttons: a blue trash can icon, a red restore icon with a white arrow, and a black circular icon with a white arrow.
- Task Test 4**: Represented by a red square icon with a white list symbol. The card contains the text "asasdasd". To the right are three buttons: a red trash can icon, a red restore icon with a white arrow, and a black circular icon with a white arrow.
- Habit Test 2**: Represented by a black square icon with a white double arrow symbol. The card contains the text "Description Test 2". To the right are three buttons: a red trash can icon, a red restore icon with a white arrow, and a black circular icon with a white arrow.
- Goal**: Represented by a red square icon with a white circular arrow symbol. The card contains no visible text. To the right are three buttons: a red trash can icon, a red restore icon with a white arrow, and a black circular icon with a white arrow.

At the bottom of the screen, there is a footer area with three navigation buttons: a left arrow, a central button with the number "1", and a right arrow.

Archived notes will be removed after 30 days if not in use

Each note's title and content (if it exists) are rendered in a card alongside an icon unique to this note type. Notes with tags, such as "Normal Test 1" also have their tags displayed on the card. Each card has a "Delete" and "Restore" buttons.

## Test 72

I pressed the "Restore" buttons on the task and the note types, which were promptly removed from the Archive and returned to their note type's main page with a success message.

## Result:

The screenshot shows the Neanote application interface. At the top, there is a header bar with the Neanote logo, a search bar, and two small icons. Below the header, the main content area is titled "Archive". It contains a note card for "Habit Test 2" which includes a title, a description, and two action buttons (trash and edit). Below this is a "Goal" section with its own card and action buttons. A red text message at the bottom of the screen states: "Archived notes will be removed after 30 days if not in use". In the bottom right corner of the main content area, there is a green success toast notification that says "Restored successfully" with a checkmark icon. At the very bottom left, there are navigation buttons for page navigation.

**Archive**

**Habit Test 2**

Description Test 2

**Goal**

Archived notes will be removed after 30 days if not in use

Restored successfully

## Notes

### Normal Test 1

Description Test 2

+ Add Note



358, Roman Sasinovich, 74561 IMS

## Tasks

+ Add Task

Task Test 4

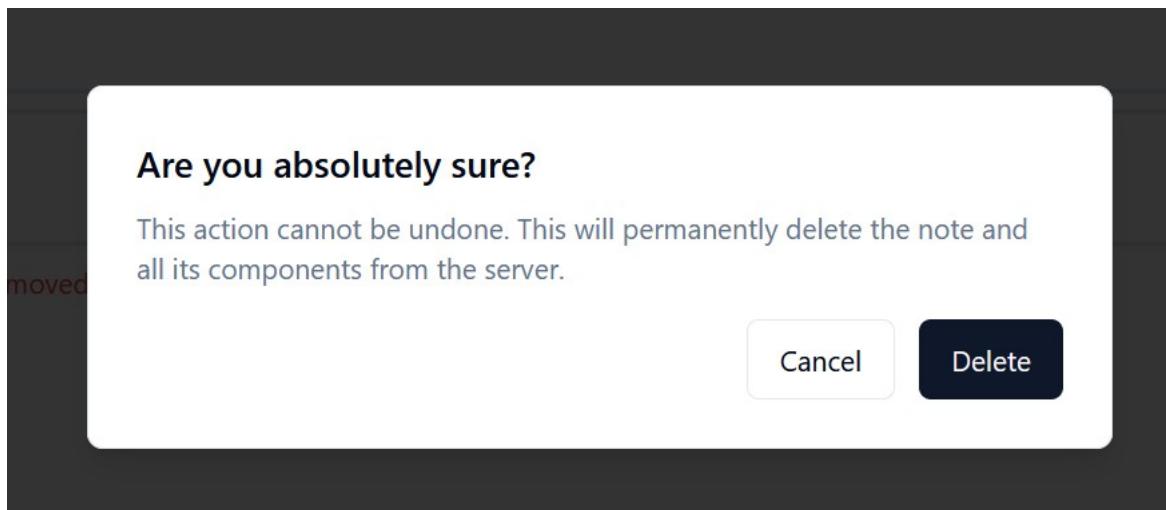
asasdasd

Apr 2, 2025, 12:00 AM

The restored notes are successfully returned to their note type's pages. Pass.

### Test 73

On the other hand, the “Delete” button press prompts the user to reconsider as the note is about to be deleted permanently:



### Result:

The deletion of a habit removes it from the Archive page and sends a DELETE request to the server:

200	DELETE	localhost:5000	delete?habitid=ef0a1171-1e16-4d0b-9247-6be0	axios.js:1646 (xhr)	json	303 B	46...
200	OPTION...	localhost:5000	delete?habitid=ef0a1171-1e16-4d0b-9247-6be0	xhr	html	396 B	0 B

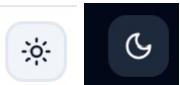
Pass.

### 4.11.4. Theme

Test #	Details	Example	Test data type	Expected result	Result
74	Evaluate how does the theme affect other		Normal	The theme switches to dark instead of light, inverting theme	Pass

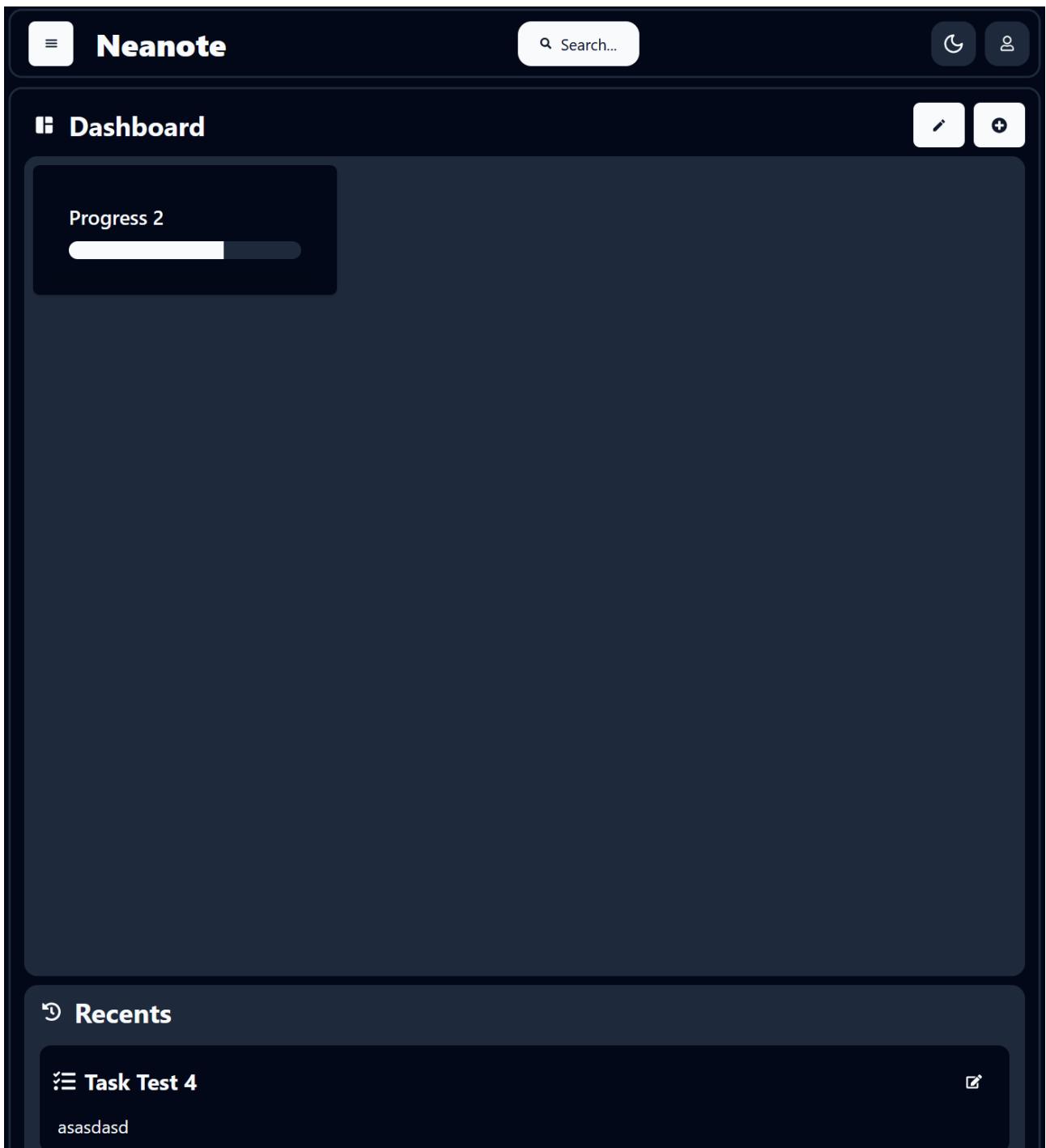
	components			colors, but not user-defined colors like tags	
--	------------	--	--	---	--

## Test 74



This button, located on the navbar, is used to switch the theme of the website from light to dark. So far, tests have been done in light mode. For this test, I will take screenshots of some components to show the changes dark mode does to them.

### Dashboard



## Tasks

The screenshot shows the Neanote application interface. At the top, there is a header bar with the Neanote logo, a search bar containing the placeholder "Search...", and two circular icons for settings and user profile. Below the header is a navigation bar with a "Tasks" icon and a "Add Task" button. The main content area displays a single task card. The task is titled "Task Test 4" and has a subtitle "asasdasd". To the right of the title is a red notification badge indicating the due date: "Apr 2, 2025, 12:00 AM" with a bell icon. Below the task card are three small navigation buttons: a left arrow, a center button labeled "1", and a right arrow.

## Goals

The screenshot shows the Neanote app interface with a dark theme. At the top, there is a header bar with the Neanote logo, a search bar, and user profile icons. Below the header, the title "Goals" is displayed next to a gear icon. A button for "Add Goal" is located in the top right corner.

The main content area displays four goal cards:

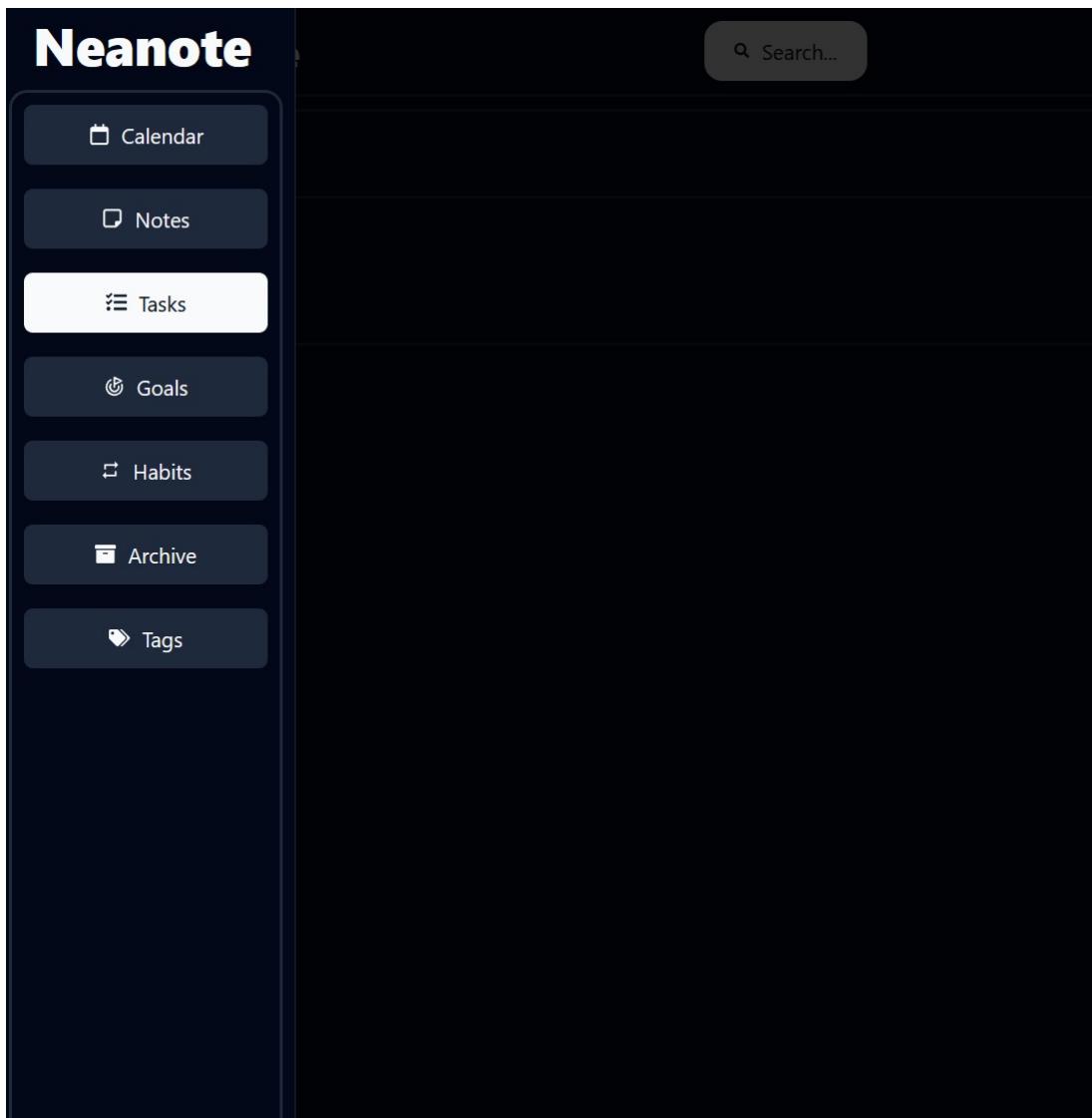
- Another goal**: Next milestone is "another one".
- Goal 20555124**: Next milestone is "141241234".
- Goal Test 2F**: Next milestone is "Ms 2". A notification badge indicates the date "Apr 16, 2025".
- Goal Test 1**: Next milestone is "Ms2". A notification badge indicates the date "Apr 20, 2025".

At the bottom left, there are navigation buttons for "1" and arrows. A green success toast message at the bottom right says "Goal archived successfully".

## Edit/Create pages

The screenshot shows the Neanote application interface. At the top, there is a header bar with a menu icon, the brand name "Neanote", a search bar, and three circular icons for refresh, settings, and user profile. Below the header, the main content area has a title "Edit Goal". Underneath the title, there is a text input field containing "Goal Test 2F" and a "Tags" button. A large text area labeled "Description Test" is present. Below this, two items are listed: "Ms 1" and "Ms 2", each with a small icon and a delete button. At the bottom left is a "Add Milestone" button. On the right side, there are three buttons: "Archive", "Save", and another delete button.

## Sidebar



## Search

cinema ≈ U

**Yesterday's cinema** |   
Just thinking about that night we went to the picture house last month. Remember how packed it was? ...

**Cinema plans** |   
Hey! Are you free next Saturday evening? I was thinking of catching the latest showing at the multip...

< 1 >

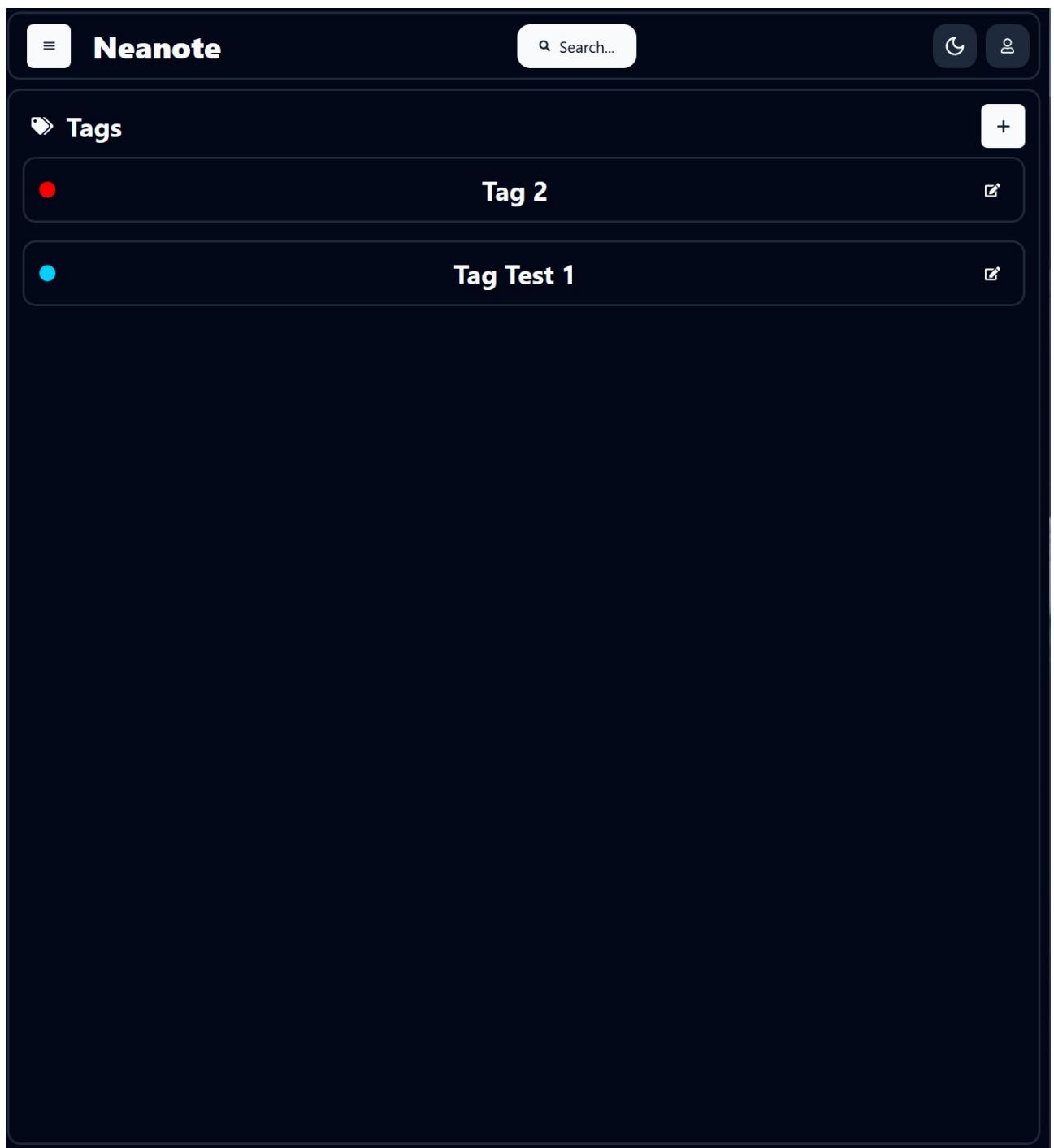
## Calendar

The screenshot shows the Neanote app's calendar view for April 2025. The interface is dark-themed with white text and icons. At the top, there is a header bar with the Neanote logo, a search bar containing "Search...", and two small circular icons. Below the header is a navigation bar with "Calendar" on the left and "Previous", "April 2025", and "Next" buttons on the right. The main area is a 6x5 grid of days from April 1st to April 30th. Most days are empty, but some contain small callout boxes with text and icons:

- Day 2: A callout box contains "Task Test" and "4" with a small edit icon.
- Day 16: A callout box contains "Goal Test" and "2F" with a small edit icon.
- Day 20: A callout box contains "Goal Test" and "1" with a small edit icon.

## Tags

367, Roman Sasinovich, 74561 IMS



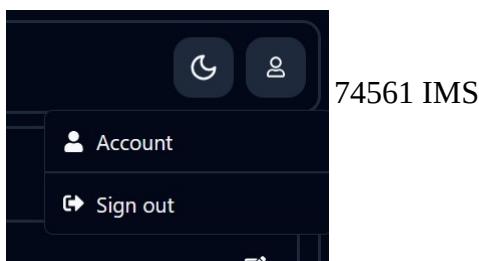
## Result

Most of the elements of the app were inverted. Most white turned into black with a blue hue, while black labels and buttons became white, while grey became dark grey with a tint of blue as well to make it easier for the eye. Even some details like the color of the overdue reminder were adjusted to a lighter version for better contrast against the dark background. What didn't change were the colors of the toast notifications, since they are bright enough, as well as user-defined tag colors. Pass.

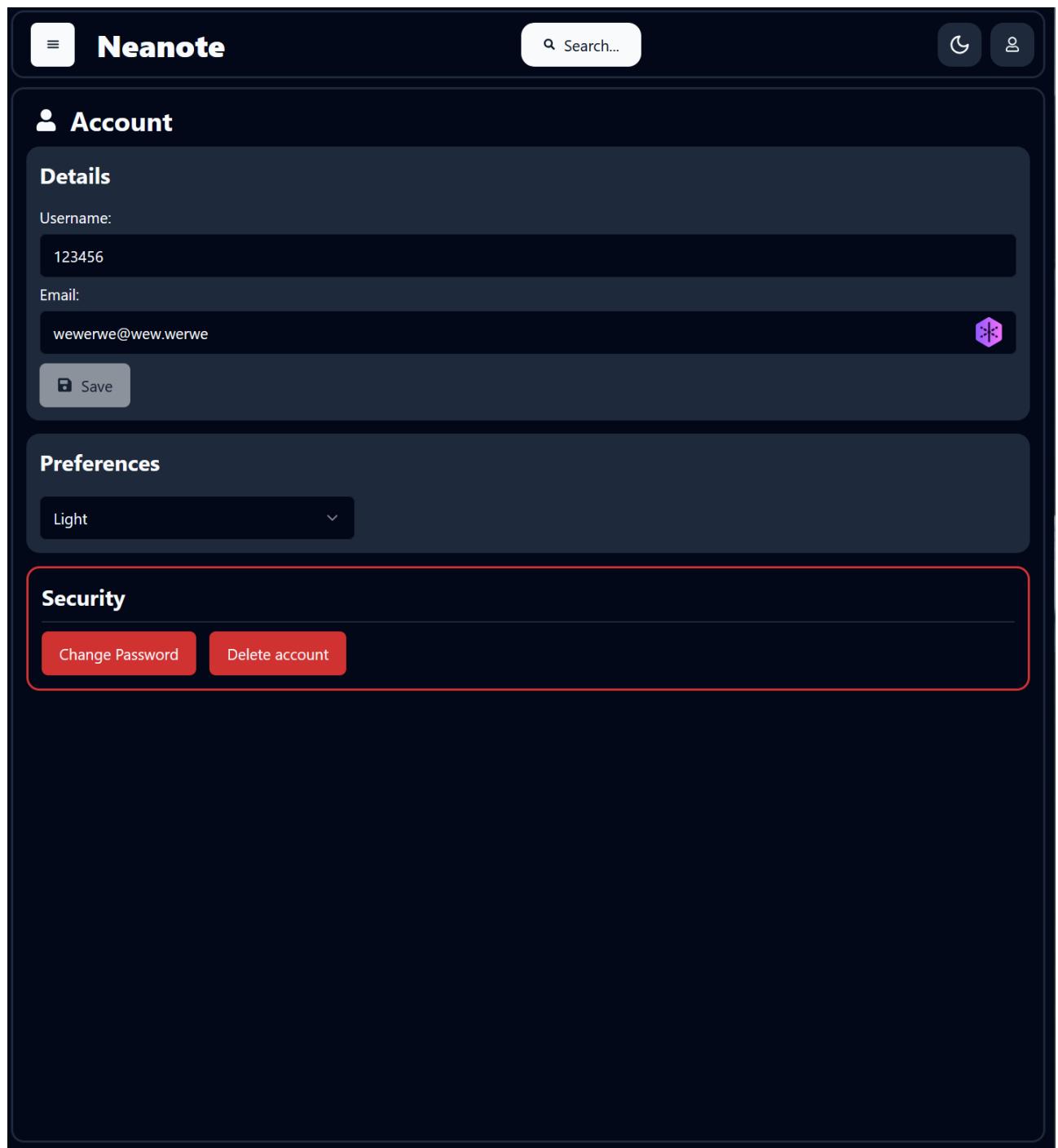
## 4.12 Account

Test #	Details	Example	Test data type	Expected result	Result
75	Attempt to change to invalid username and email	Username: 1 Email: 12321	Erroneous	Change is prohibited and an error is shown	Pass
76	Change username and email to valid information	Username: 1234567 email: example@gmail.com	Normal	Details successfully changed	Pass
77	Change user's preferred theme	Theme: Dark	Normal	Theme is set to dark upon login	Pass
78	Change password to the same password	Current Password = New Password	Erroneous	Password change is not permitted	Pass
79	Change to a different valid password		Normal	Only the new password is accepted for login	Pass
80	Delete account		Normal	User data is wiped from the database and the account can no longer be accessed	Pass

### Test 75



The button in the top right corner of the screen triggers a menu which contains a logout button and the Account button. Pressing the Account button redirects the user to the /account endpoint:



## Result

**Details**

Username:

Username must be at least 4 characters.

Email:

Invalid email address.

In the Details section of the Account page, the username and email can be edited. Entering a short username and an invalid email addresses shows validation errors and prevents the user from saving the new details. Pass.

## Test 76

## Result

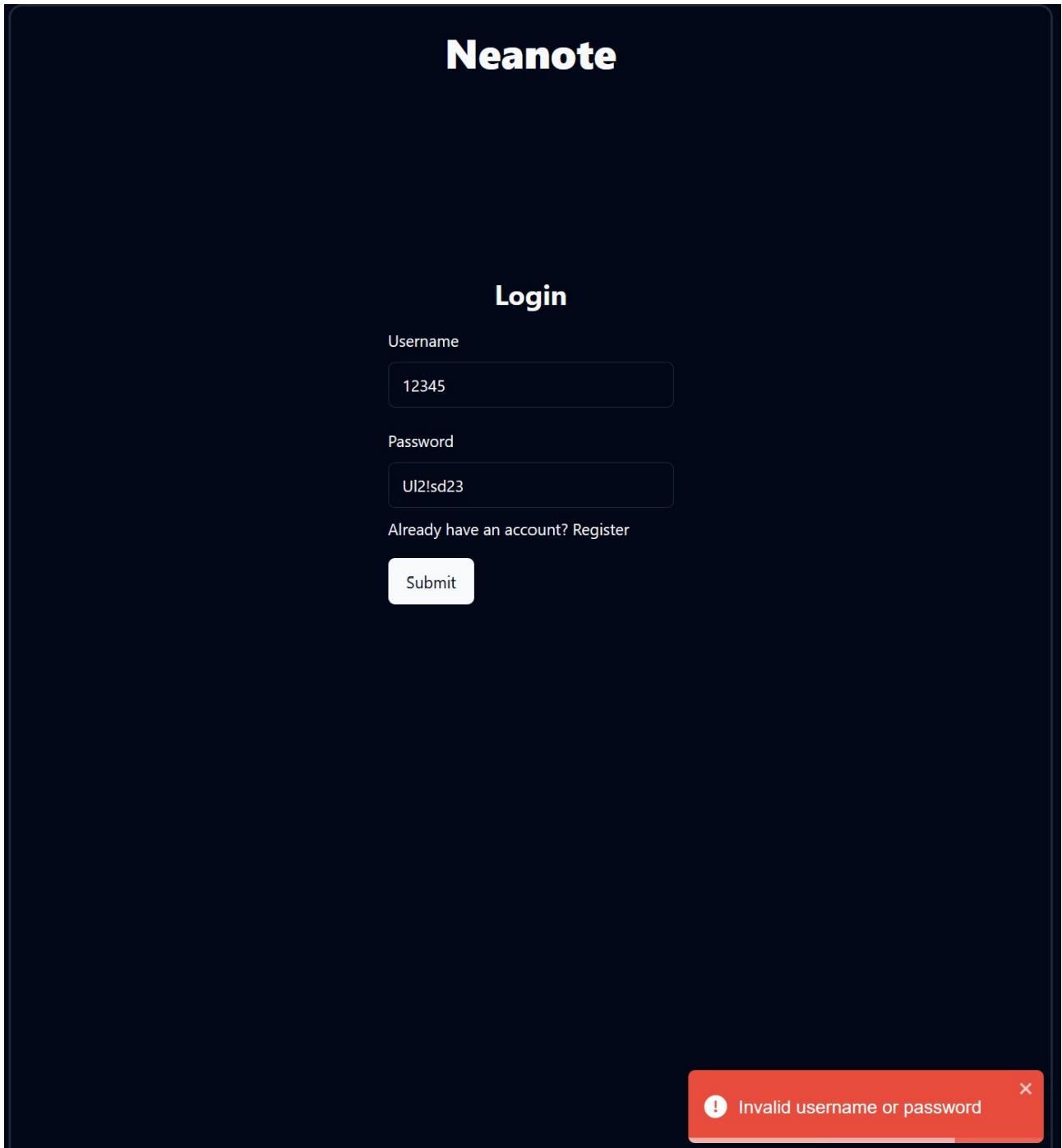
**Details**

Username:  
1234567

Email:  
example@gmail.com 

 Save

Entering valid details removes the validation errors and the new details can be saved. If I proceed to log out and attempt to log in with the old username, I get an error since this username no longer exists:



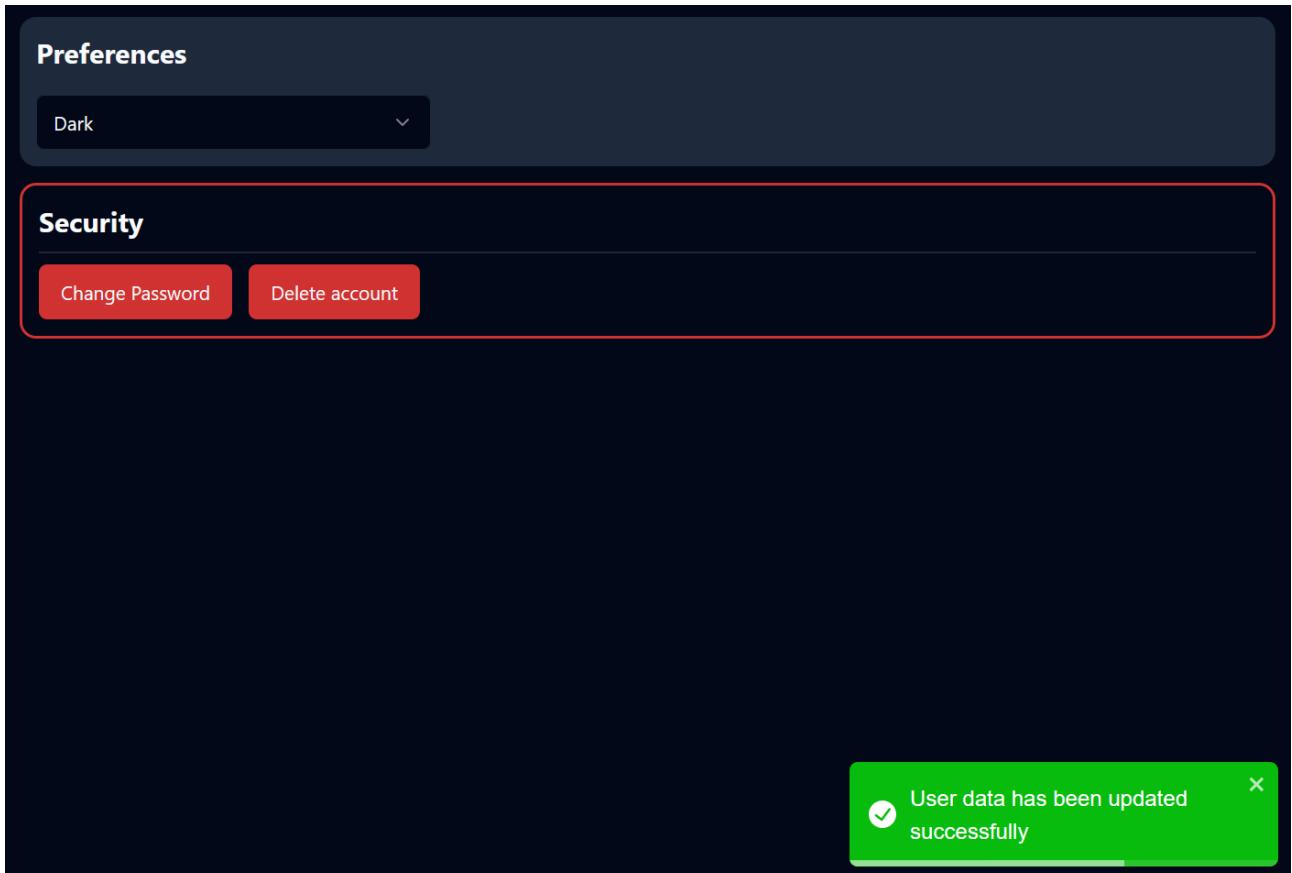
However, the login with a new username works and the theme is automatically set to light, since it is in the user's preferences, which is visible on the screenshot in test 75.

Pass.

### Test 77

### Result

373, Roman Sasinovich, 74561 IMS

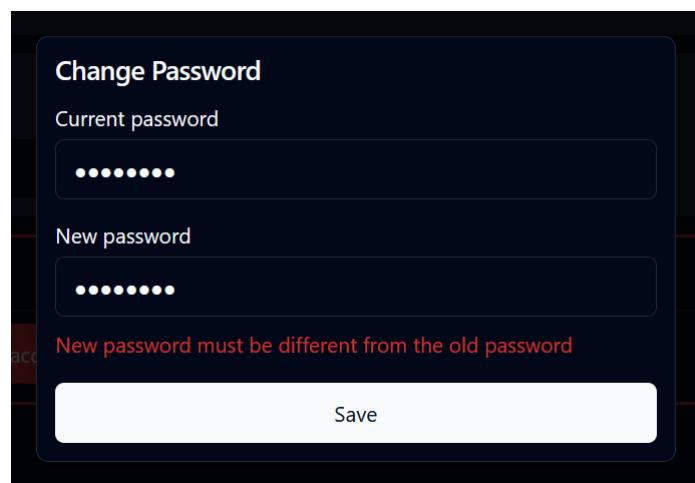


This automatically switches the current theme to dark and is persistent even after the user relogsins. Pass.

### Test 78

The “Change Password” button is in the Security section of the Account page. A press on it opens a popup menu, when the user is prompted to enter a new password. I will enter two same passwords in the field, which are equal to the password for this account: UI2!sd23

### Result



This raises a validation error. Pass

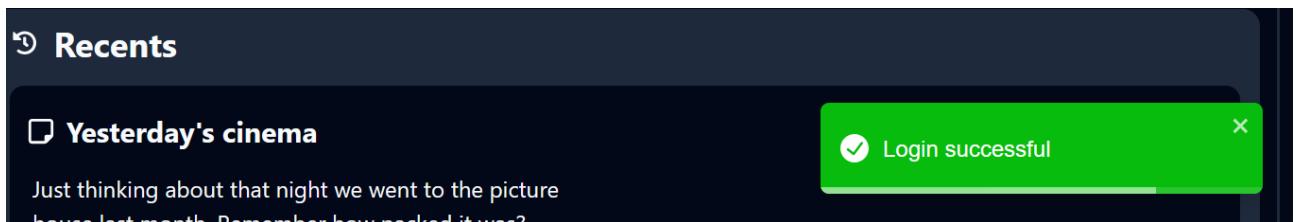
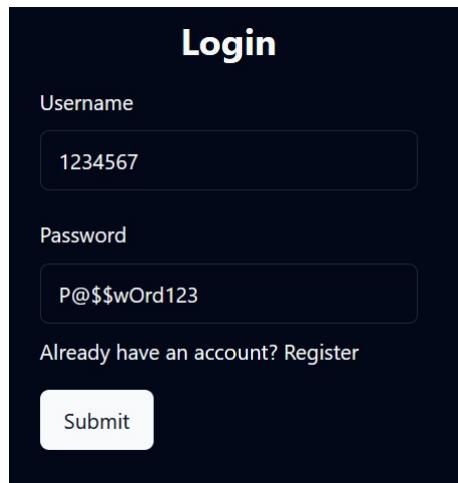
## Test 79

I made up a new password :P@\$\$wOrd123

and entered it into the dialog.

## Result

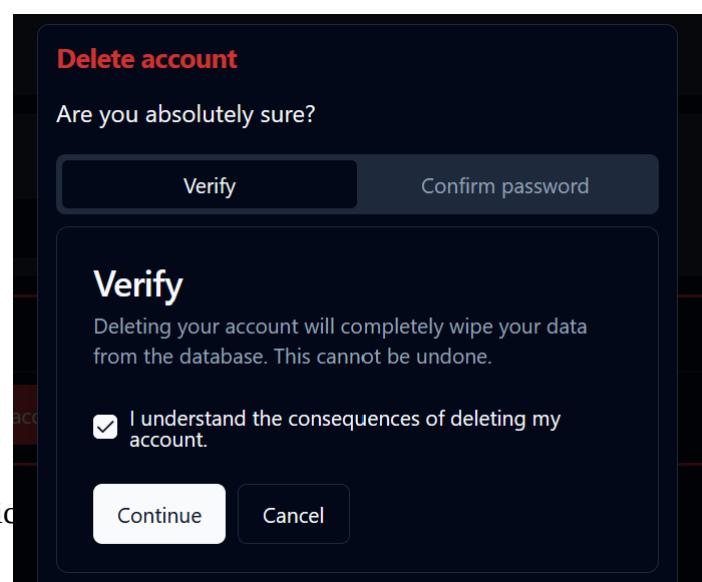
The change was successful, and now only the new password can be used for login:



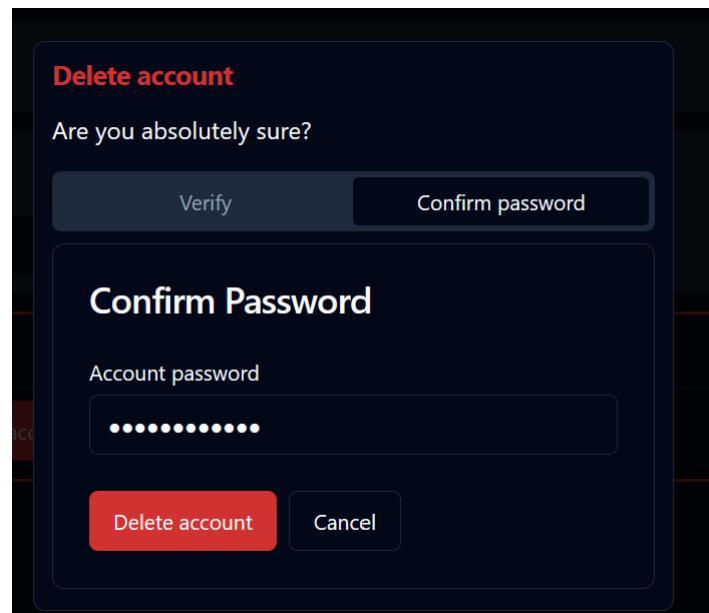
Pass.

## Test 80

Several steps need to be taken to delete the account. Firstly, the user should verify that they understand what they are doing:



Then, on the next page, they have to enter their password:



## Result

Upon user deletion, their data is removed from the database following a DELETE request, they are logged out and their credentials no longer work:

The screenshot shows a browser window with a dark theme. At the top, there is a "Login" page with fields for "Username" (containing "1234567") and "Password" (containing "P@\$\$wOrd123"). Below these fields is a link "Already have an account? Register" and a "Submit" button. Below the login form, the browser's developer tools are open, specifically the Network tab. The Network tab displays a list of network requests:

Status	Method	Domain	File	Initiator	Type	Transferred
200	DELETE	localhost:5000	delete	axios.js:1646 (xhr)	json	302 B
200	OPTIONS	localhost:5000	delete	xhr	html	410 B
500	POST	localhost:5000	login	axios.js:1646 (xhr)	json	412 B
200	OPTIONS	localhost:5000	login	xhr	html	408 B

Pass.

## 5 Evaluation

## 6 Appendix