Denisa-Gabriela Neagu

January 2nd, 2025

Bachelor's Final Project

Standard pages: 34.5
Characters: 82.819

Project supervisor:

Kim Duus Thøisen

KT@easv.dk

Participants:

Denisa-Gabriela Neagu

dennea01@easv365.dk

GitHub repository: https://github.com/denisa122/Bachelor-s-Degree-Project

Render: https://mindscribe-frontend.onrender.com/

Table of Contents

Introduction

The main purpose of this final project of the Top-Up Bachelor's Degree study program is to put in practice all the theory, skills, and knowledge that I have gained during the education. In completing this project, I not only used the skills and knowledge that I already possessed, but also stepped out of my comfort zone and had the opportunity to learn new skills and gain more knowledge across diverse areas. Creating this project has allowed me to successfully achieve the learning outcomes that the study program sets for its students.

The project focuses on the development of a website application designed to provide users with a safe space to put their thoughts into writing, in the form of journaling. Sentiment analysis is then performed on the journal entries made by the users, in order to create tailored personal insights for each individual user. Another functionality of the platform that allows for the generation of custom personal insights is the mood tracker page. This consists in a series of questionnaires that the users can fill out during the day, and which focus on the emotional states and fluctuating moods that the users might experience throughout the day. The primary goal of this website application is to provide a safe space for users and to allow them to express their feelings and moods, and to provide users with actionable insights based on their inputs.

This report aims to describe into more detail the process of designing and implementing this website application, including the challenges that I have encountered during the development process and the solutions that I came up with.

Problem statement

The development of this project was inspired by the current events that were happening around me at the time of having to decide the topic of the final project of my education. The winter weather and atmosphere had set in, and I came up with the idea of creating a platform that might help users through this hard time of the year, and not only, by providing a safe space for self-expression. Although there are other options out there that provide journaling and mood-tracking tools, I could not find anything that also has the functionality of creating tailored insights based on the user input. I decided to address this gap in other current applications and create a platform that integrates both sentiment analysis based on user input (journal entries), and a mood-tracking feature through different period of time.

Participants

The only participant in this project is me, Denisa-Gabriela Neagu. I fulfilled the roles of the designer, the developer and the evaluator, therefore undertaking all aspects of the project. This approach allowed me to utilize all the skills that I have gained during the study program, across multiple areas – design, databases, frontend and back-end implementation, testing, and deployment.

Subject

The subject of this project is to create a web application that combines features of journal writing with generating personal insights into a user's mental well-being through sentiment analysis and mood tracking.

The application that I designed and implemented aims to create an engaging and meaningful experience for its users and to promote emotional well-being and self-awareness through its features.

Problem/ Problem area

The inspiration behind this project was based on current events that were happening around me, with the winter season setting in and the general mood and feelings of people going down due to this change.

After performing some research to see the available options of applications that might help ease this transition and might provide support for those in need of it, I came to the conclusion that there is a lack of a platform that would combine the aspect of users being able to express themselves freely, with the aspect of providing the users with a clear and straightforward visualization of their actual feelings, moods and sentiments and how these change and evolve throughout periods of time. While there are many existing solutions that focus on the aspect of journaling, these solutions don't integrate meaningful feedback or a deeper analysis of the users' emotional journeys.

I decided to bridge this gap with the help of the application that I came up with. It seeks to encourage and empower users to reflect on their mental well-being and emotional well-being, with the help of the tools and features provided by the platform.

Main question and sub-questions

How can a journaling web application integrate sentiment analysis and mood tracking over periods of time, in order to provide personalized insights into its users' emotional journeys?

Sub-questions:

- What are the key features needed on the platform, to create an engaging journaling application?
- How can sentiment analysis be implemented to generate meaningful and accurate insights based on the user input?
- How can user data be managed securely and be in compliance with the GDPR regulations?

Method/ Procedure

The creation of this project followed a structures development methodology, which included the following steps:

1. Research and coming out with a concept. This consisted of researching current similar solutions and identify what they are lacking in or what they could do better, and then defining project requirements and objectives based on the findings of my research.
2. Design. This phase consisted of coming up with what pages I want there to be on my application, and then creating paper sketches and wireframes, as well as prototypes for these pages. I tested my design by performing a number of user tests. I also created my desired database design during this step of the project.
3. Development. The development step was the longest and most difficult one, since it consisted of implementing the database, the back end and the frontend of my application.
4. Testing and evaluation. This consisted of conducting integration tests on the backend components of my application, to make sure that the application performed as expected.
5. Deployment. This step meant hosting the application on a live server.

MoSCoW Prioritization

The MoSCoW prioritization is a four-step prioritization technique for managing project requirements. The acronym MoSCoW stands for the four categories of initiative: must-have, should-have, could-have a won't have, or will not have right now. (*Moscow prioritization* 2024)

I decided to follow this prioritization technique in order to divide the requirements for my project into separate categories and to get a better overview of what were the most important features to implement on the platform, and what were some features that had a lower degree of importance. The MoSCoW prioritization that I created can be seen below, in Figure 1.

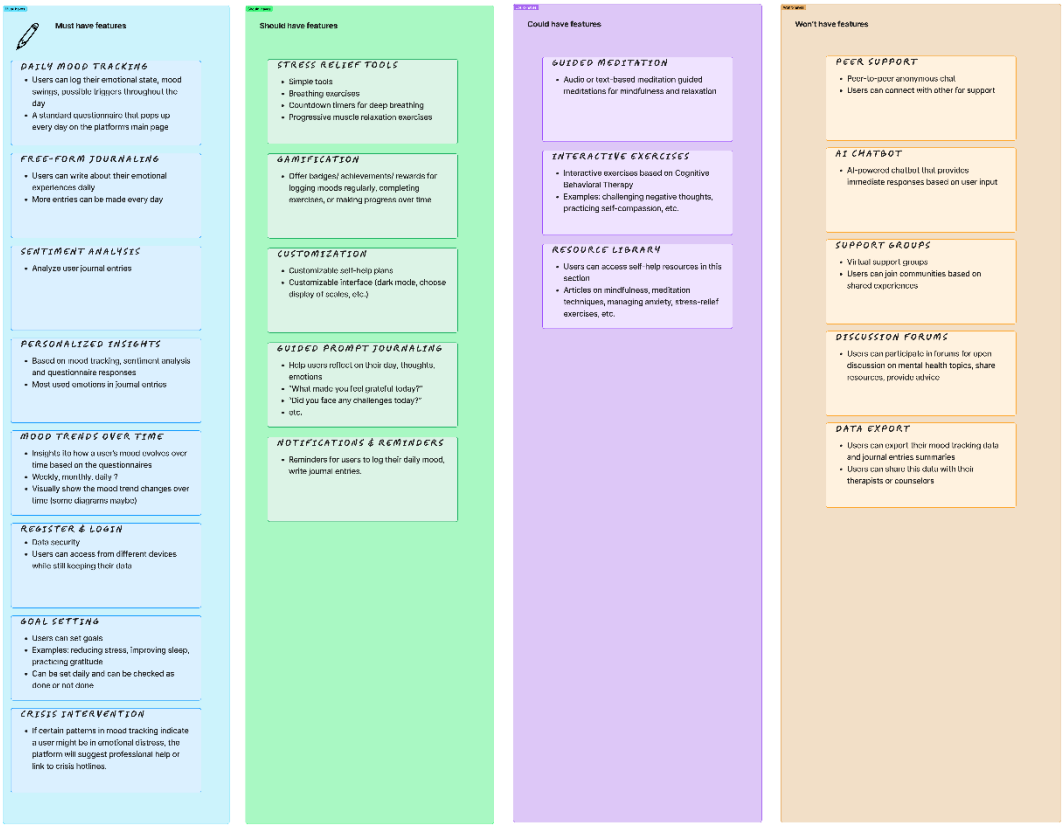

Figure 1. MoSCoW Prioritization of project requirements

M: Must have

This category includes all the project requirements that are mandatory for the project. The implementation of these requirements leads to the successful implementation of the desired functionality of the project, and without these features, my website application could not be considered finished.

Since the main focus of the platform that I was creating was on two things – users being able to express themselves through journaling, and the platform generating personalized insights based on these journal entries and also on mood tracking – the features that could not miss from the website are the following: free-form journaling,

daily mood tracking, and personalized insights. Sentiment analysis was also a must-have feature, due to the fact that this is what is used to analyze the text journal entries made by the users. Tracking mood trends over time was also a very important aspect, since it is a very useful way of allowing users to visualize how their mood changes and fluctuates over different periods of time, and better understand how certain events in time affect their emotional well-being. I decided that the option of users setting daily goals for themselves was also a feature that must be implemented on the platform, since setting clear daily objectives and celebrating small victories, in the form of completing a task or a chore, can be a mood booster for many people. Crisis intervention is also a very important feature that I thought has to be implemented on the website, since it is a way of alerting users that there is available help if they are feeling lower for a period of time. The last must-have feature that I came up with for my platform was the possibility of registering and logging in on the platform. This is important, as it provides data security – especially when dealing with such sensitive data about users – and it allows the users to access the platform from different devices, while still keeping their data.

S: Should have

This category includes the project requirements that are one step below the must-have requirements. This means that these requirements are important to completing the project, and add other value to the project, but are not vital.

Features like guided prompt journaling, a notifications system and reminders, customization (for example, customizing the user interface to either light mode or dark mode, choosing whether scales should be displayed as numerical or emoji scales, and so on), gamification (offering some kind of rewards or achievement to users for performing tasks like completing all three mood tracker questionnaires during the day, setting daily goals for a number of days in a row, entering journal entries regularly, etc.) and stress relief tools (like breathing exercises, countdown timers for deep breathing and progressive muscle relaxation exercises) would add significant value to my application and make it more robust, but the application can function and make sense as a finished product without them as well.

C: Could have

This category includes the project requirements which would be nice to have, but in the case that they are left out from the project, they will have a much smaller impact. This kind of requirements have low priority compared to must-have and should-have requirements.

What I thought are some nice features to potentially add to the platform, in order to make it more robust and engaging for the users, are adding guided meditation (both audio or text-based), having a resource library, where users can access self0help resources and read articles on different topics (such as mindfulness, meditation techniques, managing anxiety, stress-relief exercises, etc.) and having a section

dedicated to interactive exercises for the users, which they can practice individually from the comfort of their own home (like challenging negative thought, practicing self-compassion, etc.).

W: Will not have

This category includes the project requirements that are not a priority for the current project's time frame.

Some examples of what I thought are project requirements than qualify in this category are the following: peer support in the form or peer-to-peer anonymous chat, AI chatbot, support groups that users could join based on shared experiences, discussion forums that users could participate to, and the data export option (users could export their mood tracking data and journal entries, and potentially share this with a professional). Most of these features are in this category because they are not so much focused on individual growth and understanding, as they are in sharing resources and experiences with other people as a group.

Project Management

Project management was crucial for the development of this project, and I tried to ensure that I had an organized approach when it comes to this. Since I was working alone throughout the whole project, I was responsible for aspects related to both time management and to the technical development of the project.

To ensure that I kept making progress during all the stages of the project, from research to design and actual implementation and delivery, I used a combination of task managements tools.

Task management

As I was managing the project on my own, I was the one responsible for deciding the priority of the features to be implemented on my platform, estimating the duration of the implementation of these features and deciding when to implement what feature. To get a better overview of all the work that still needed to be done in order to implement an application that fulfills the requirements that I set in a timely manner, I decided to divide the project into smaller milestones. Some of these milestones included research of tools, technologies and framework, creating a guiding frontend design, creating a database schema, setting up the backend API, implementation of the frontend, and so on.

I figured out that the best approach for me would be to decide in the beginning of each week what set of tasks to focus on during that specific week. This allowed me to set clear goals for the week and stick to the implementation of those specific features. I tried to make sure that I allocated enough time for each of the tasks, depending on the complexity that I estimated them to be of, while also leaving some flexibility for issues that might arise during the process of working on these tasks. Each week, I reviewed the progress that I made during the previous week and proceeded to assign the tasks to work on for the following week.

To organize the tasks that I was working on, I used Trello, a task management tool that allowed me to create a visual board with lists and cards. I divided the tasks (cards) in three categories, "To do", "Doing", and "Done", and moved them around from one section to another based on the stage of the task at certain points in time. I assigned a due date to each task, and whenever a task was done, I was able to mark it as completed using Trello's built-in feature to do so. Trello also allowed me to make adjustments to task deadlines and to re-prioritize tasks, based on certain situations that appeared during the development process.

By being able to manage my own deadlines through the use of Trello and being able to see a clear visualization of tasks and their state, I kept track of my project's progress without feeling overwhelmed or stuck at any point.

Agile Development Methodology

In the process of managing this project, I aimed to adopt certain Agile development methodology principles. Agile is a project management approach that is centered around incremental and iterative steps to complete a project. The Agile approach prioritizes quick delivery and adapting to change. (*What is agile? and when to use it*)

Even though Agile is most often associated with teams and working collaboratively on a project, I found that some of its main principles could be applied to working individually on project as well.

Agile methodology divides work into sprints, in which specific tasks or features are completed. I didn't have time for a more formal division of my work into sprints, but I did section my project into smaller parts and try to assign realistic deadlines for each task. The weekly planning of new tasks and revision of tasks from the previous week could be considered as treating each week as a short sprint.

Another aspect of Agile development methodology is being able to adapt based on changing requirements and based on constant feedback. I tried to embrace this principle by reviewing my application's functionality frequently, more precisely by testing each feature that I implemented as if I were a real-life user of the platform, and refining these features based on this. This allowed me to have more flexibility to the needs and requirements of the application that I was developing.

Challenges

Being responsible for both project management and project development came with a few challenges, especially considering that this was the first time when I was managing or implementing a project of this scale. While I have previous experience with working on smaller projects throughout my education and throughout that two internships that I have completed, I never had to deliver such a comprehensive product on my own.

A particularly difficult aspect that I encountered was correctly estimating how long it would take to finish certain tasks. There were instances when I underestimated the time required for the development of specific features, especially when it came to backend implementation of the code, which led to some unexpected delays. There were also times when I allocated too much time for particular tasks, which I now realize that were not as important as I considered them to be initially.

Since this project was a personal project, at times I also found myself getting caught app in adding new features that were not necessarily of high priority for the scope of the project, or fine-tuning aspects of the application that were minor in comparison to others. It was important for me to recognize when these adjustments that I was making started to come in the way of the original goals and deadlines that I initially set for the project. As the project period went on, I always tried to make sure that I was not letting this happen too often and not letting it affect the final result that I was aiming to deliver for this project.

Overall, managing both the project management aspects and the actual development of the code came with its challenges, but it also provided me with some valuable insights into task prioritization, time management, and handling such a complex project through every step of the design and development process. It is my belief that I managed to navigate each obstacle that appeared during the time that I spent on this project, and that I was able to deliver a product that meets the requirements and expectations that I set for myself in the beginning.

Design

This section of the report will describe in more detail the process of designing the user interface of my website application, before the actual implementation.

Pages on the website

The first step was coming out with a layout of the website and deciding what pages should be included and later on implemented for the purpose of this project, and what each of these pages should contain. This process was based on the categorization of project requirements that I have achieved using the MoSCoW analysis, described in an earlier chapter of this report.

I decided to focus, for now, only on the pages that are essential to any website application (like homepage, log in page, register page), as well as on the pages that would encompass the main functionality that I desired for the platform. The final pages that I decided to implement for this project are: Register page, Login page, Homepage for guests, Homepage for logged in user, Journal page, Mood Tracker page and Insights page.

The Register and Login page are standard to any application that allows users to create an account and to log in to their account. These two pages both consist of just a simple, typical form.

There are two homepages for my application, one for the situation when a guest (either a user who does not have an account, or a user who is not currently logged in) is browsing the website, and one for the case when a user is logged in. These two pages serve different purposes. On one hand, the homepage dedicated to guests is meant to attract users, by introducing them to the platform and its main features, and guide them to sign up. On the other hand, the homepage dedicated to a user who is logged in to the platform is meant to provide quick access to some of the most important features of the platform (like journaling, mood tracking questionnaires, and insights). This page also contains some of the project requirements from the should-have category, like notifications, which I thought be a nice addition to a front page.

The Journal page is the place where the users can easily access the journal writing tool provided by the website. Here, they can also see their previous journal entries, categorized by month and year.

The Mood Tracker page is the one where users can access the three mood tracker questionnaires that they have the option of filling in throughout the day – Morning questionnaire, Midday questionnaire and Evening questionnaire. The users can also set their daily goals on this page and mark them as complete whenever they finish them in real life. It is also easy for users to navigate to their personal insights from this page.

The Insights page is the place where users can see and track their emotional journey. It provides visual representations for mood changes, increases or decreases in energy level or stress level, what kind of emotions they have been experiencing lately and their respective percentages, and the most used sentiments, extracted from their journal entries.

Sketches

Sketching is an invaluable tool that allows to connect the abstract concepts to the tangible visuals. When ideas are put on paper, the vision begins to take shape and a better sense of understanding of the project is provided. (Dudley, *The role of sketching in the design process* 2023)

For me, the process of creating sketches for each page of my application helped me better differentiate between the unique features of the platform and the scope and purpose of each of the main pages found on my website.

Below, the initial sketches that I created can be seen.

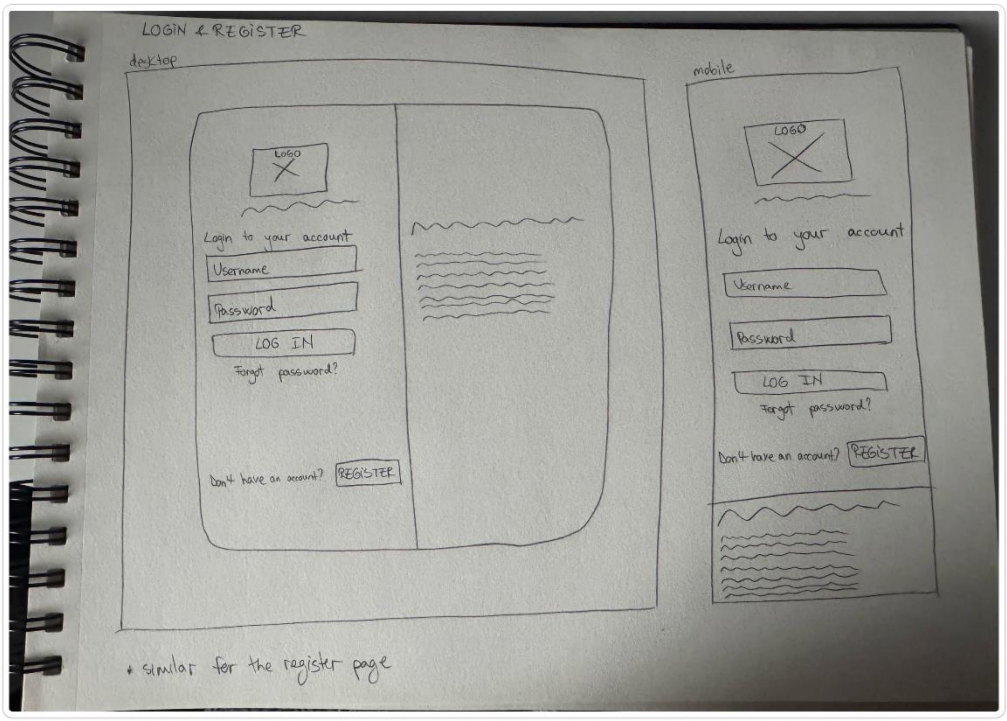

Figure 2. Sketch - Lofgin & Register pages

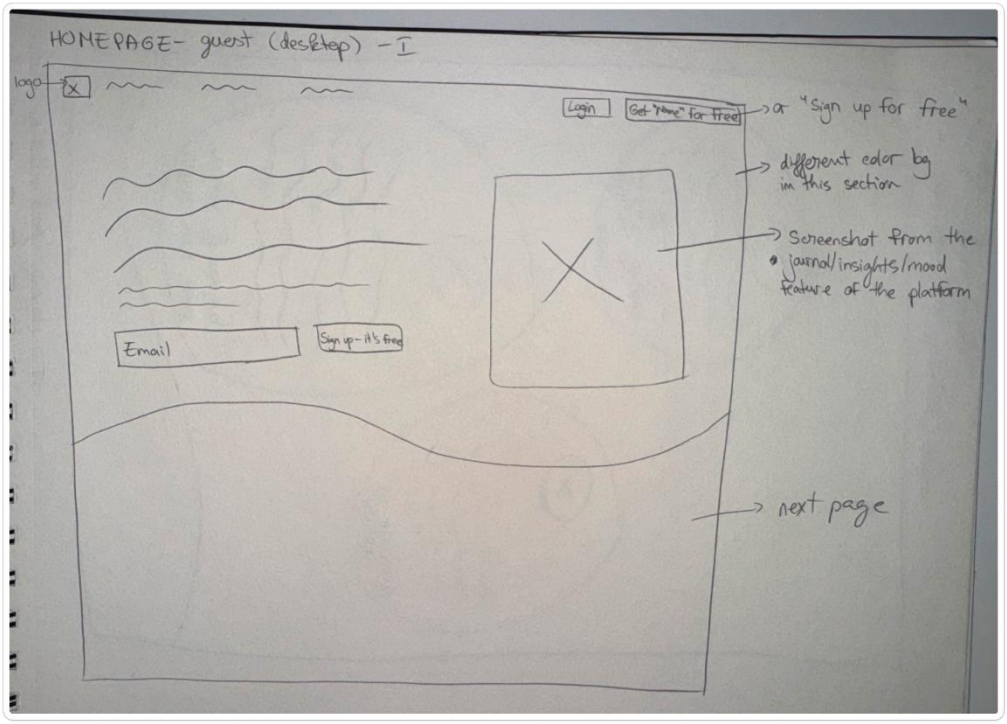

Figure 3. Sketch - Homepage, guest (1)

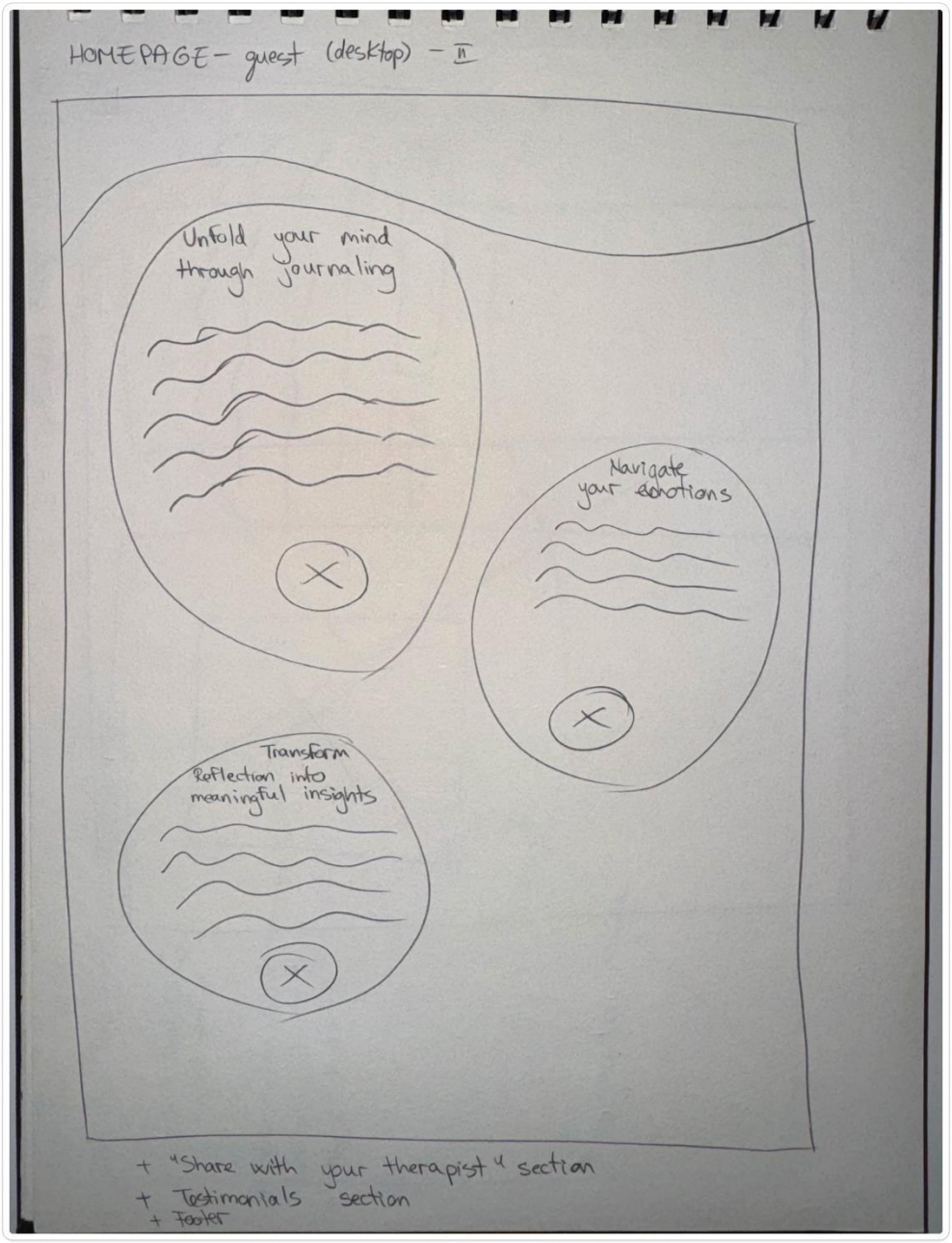

Figure 4. Sketch - Homepage, guest (2)

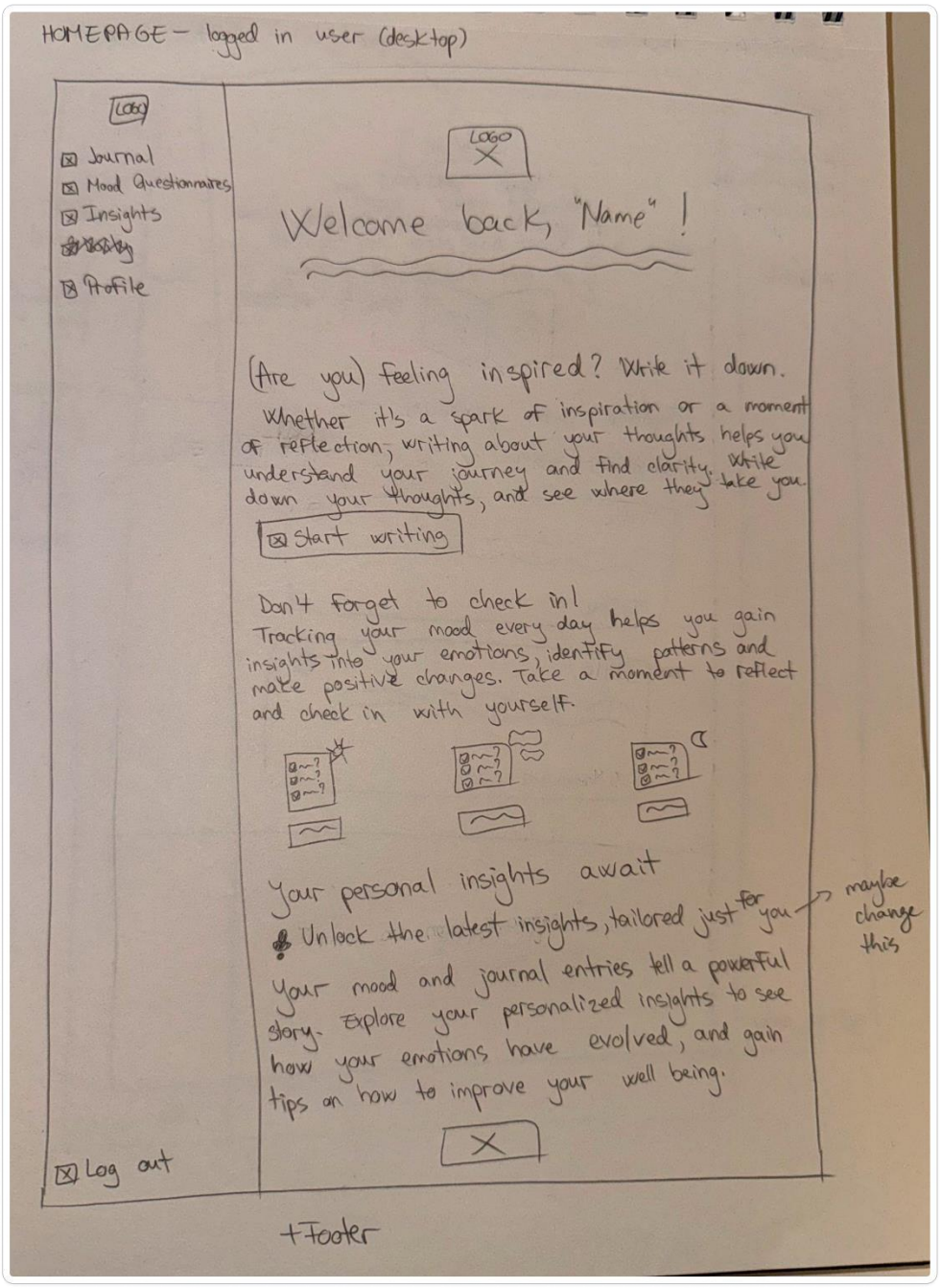

Figure 5. Sketch - Homepage, logged in user

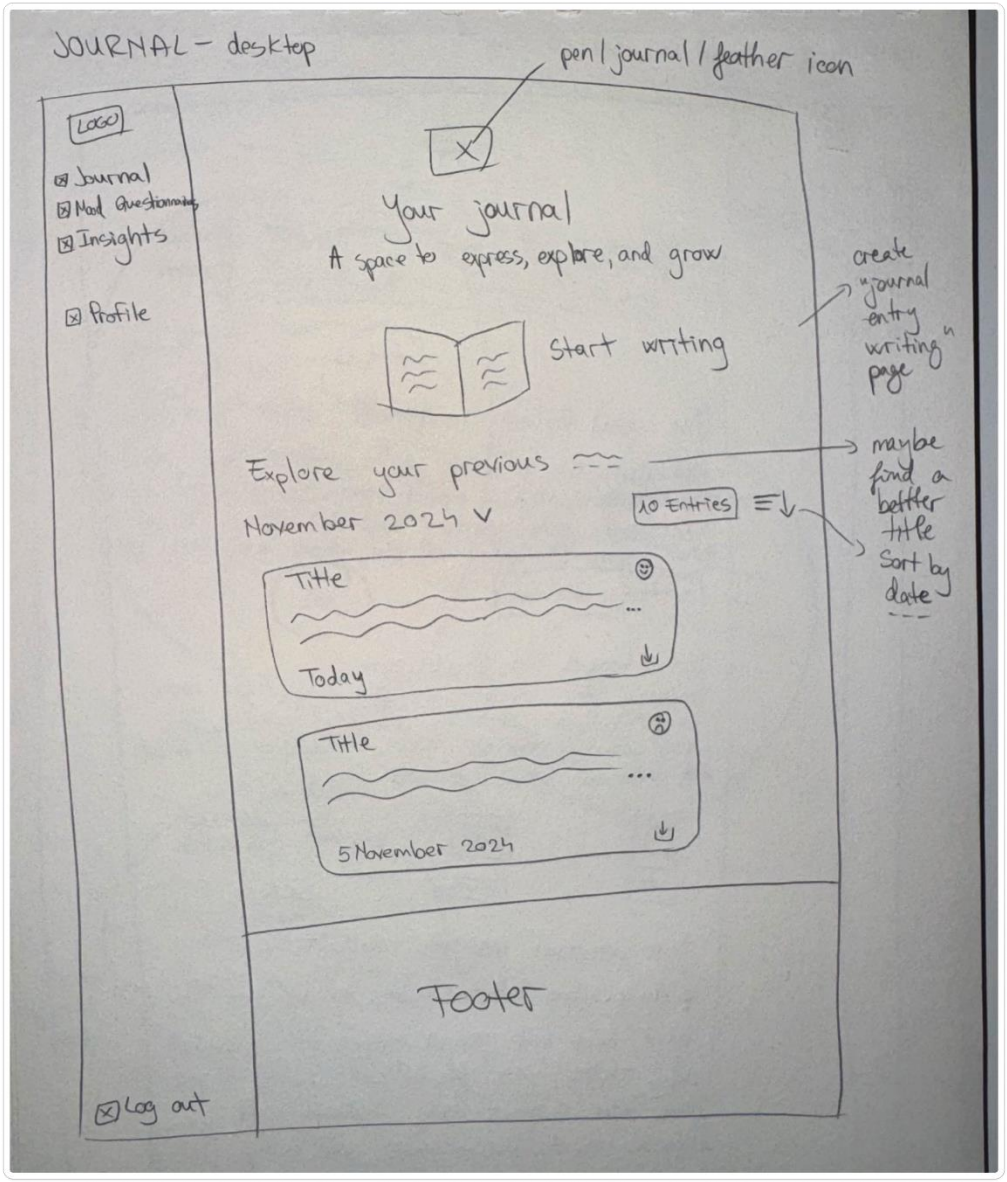

Figure 6. Sketch - Journal page

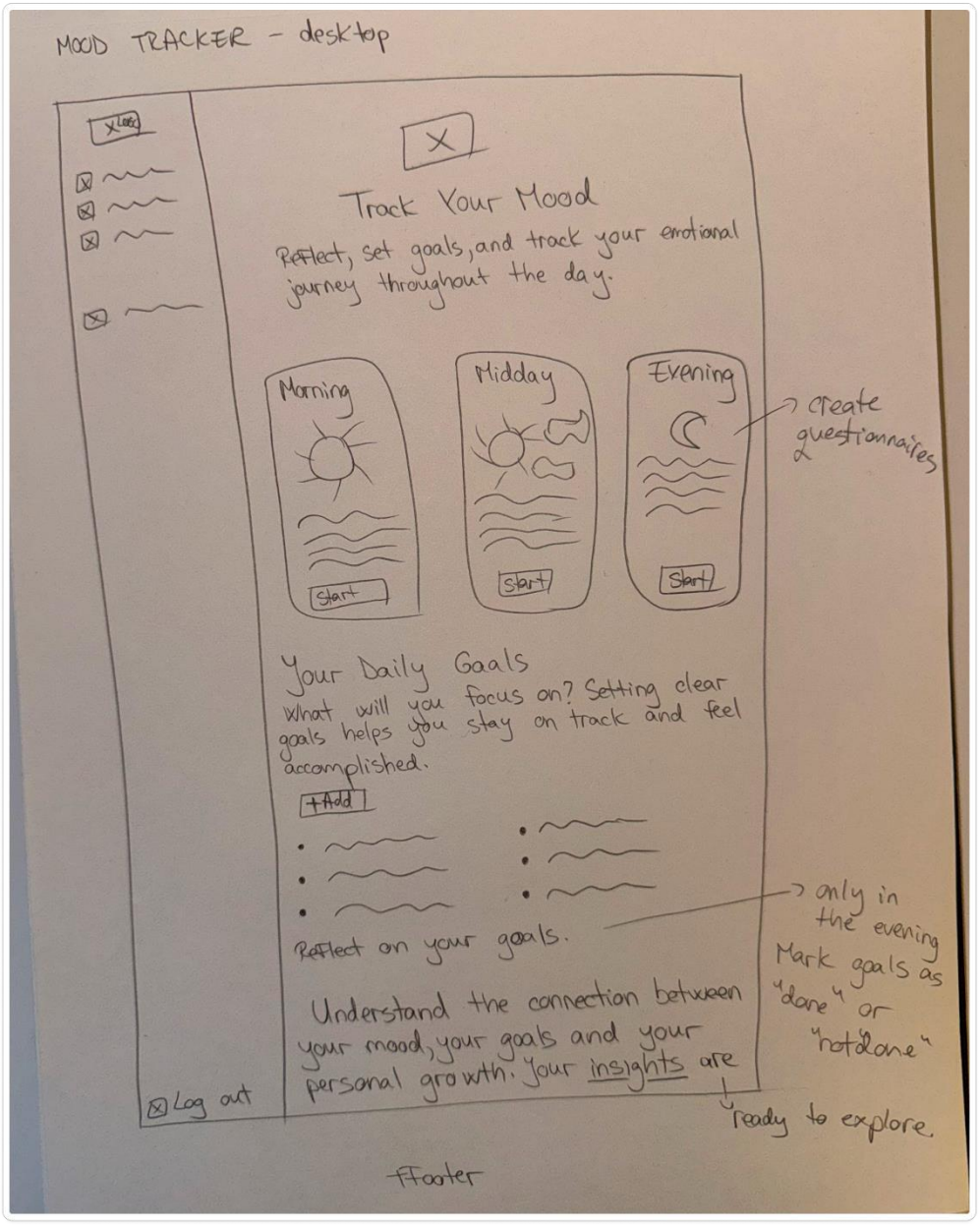

Figure 7. Sketch - Mood Tracker page

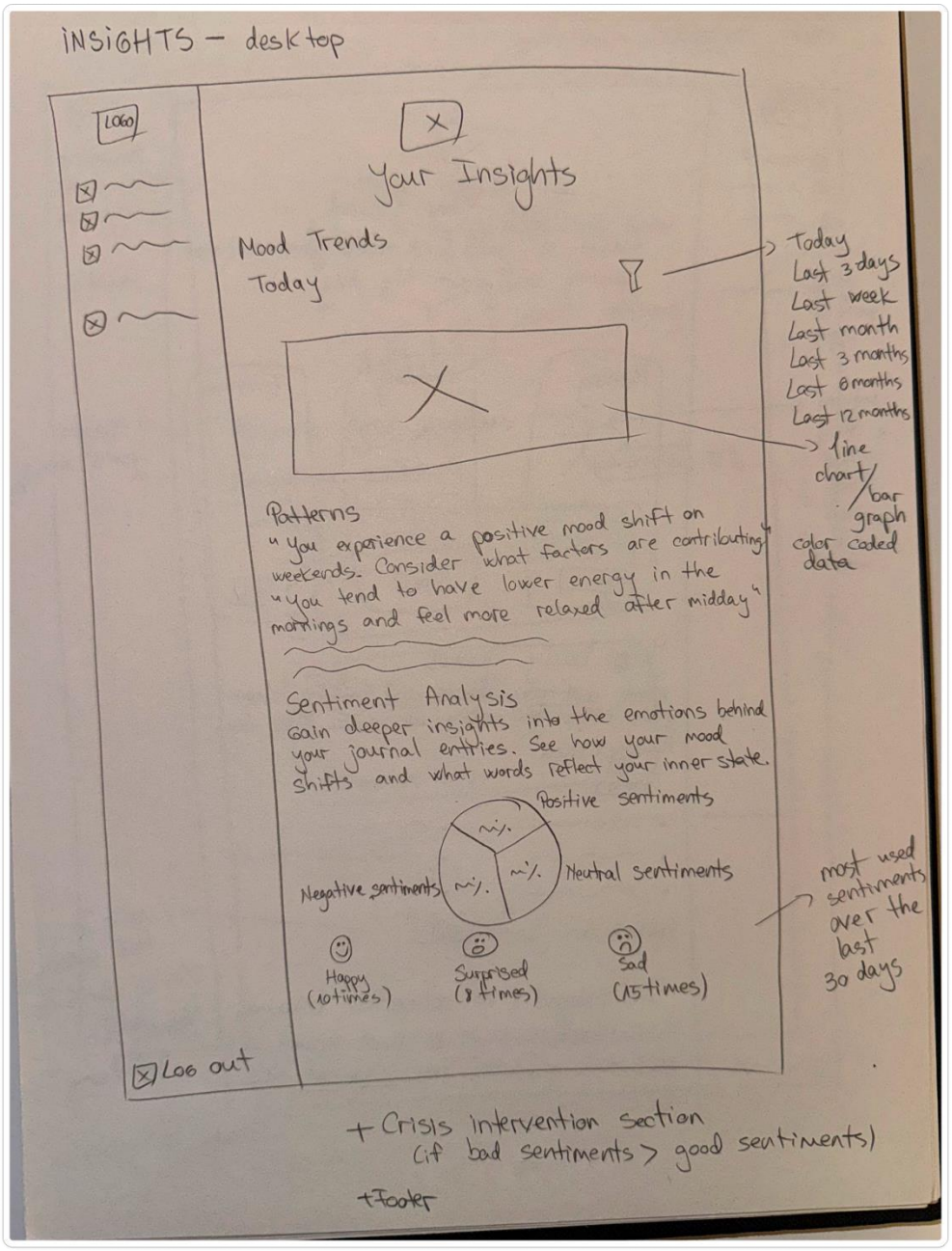

Figure 8. Sketch - Insights page

Wireframes

Wireframes represent basic visual representation of a user interface, which outline the structure and the layout of a webpage. They serve as blueprints that help understand the placement of elements on the page, the structure and flow of the design, and they don't focus on details like colors, fonts or images. (*What is wireframing? - updated 2024* 2024)

I created wireframes based on the sketches that I had previously finished. Doing this allowed me to better visualize my design and how each element or block of elements would look like on a webpage, as well as how the flow of the elements looked like and whether I was satisfied or not with the design that I previously thought about in my sketches. I kept changing the wireframes and testing how they would look like as an actual website, until I was satisfied with the results.

The final form of the wireframes can be seen below.

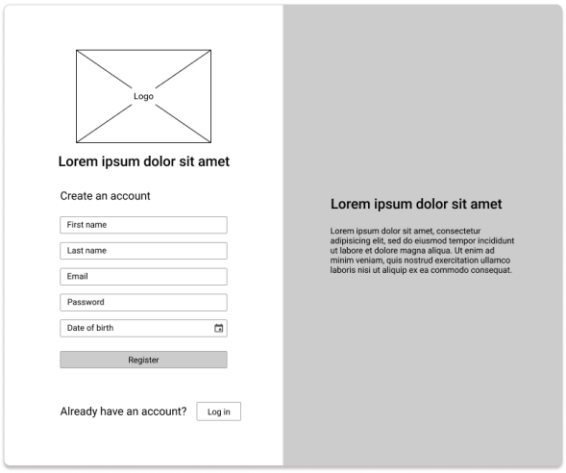

Figure 9. Wireframe - Register page

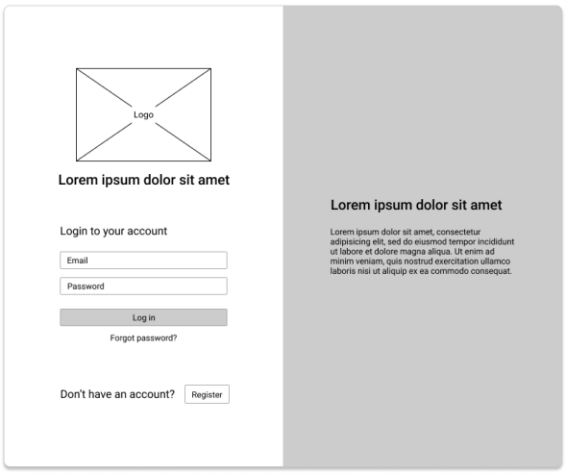

Figure 10. Wireframe - Login page

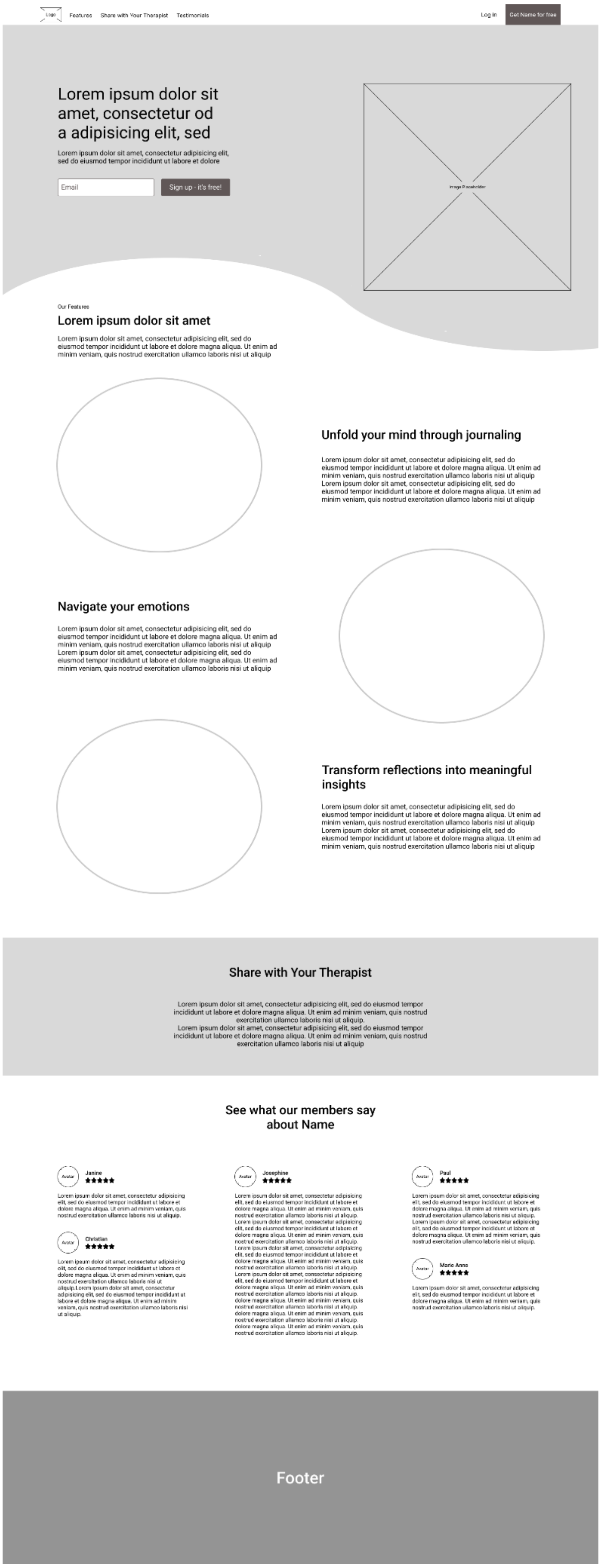

Figure 11. Wireframe - Homepage, guest (Option 1)

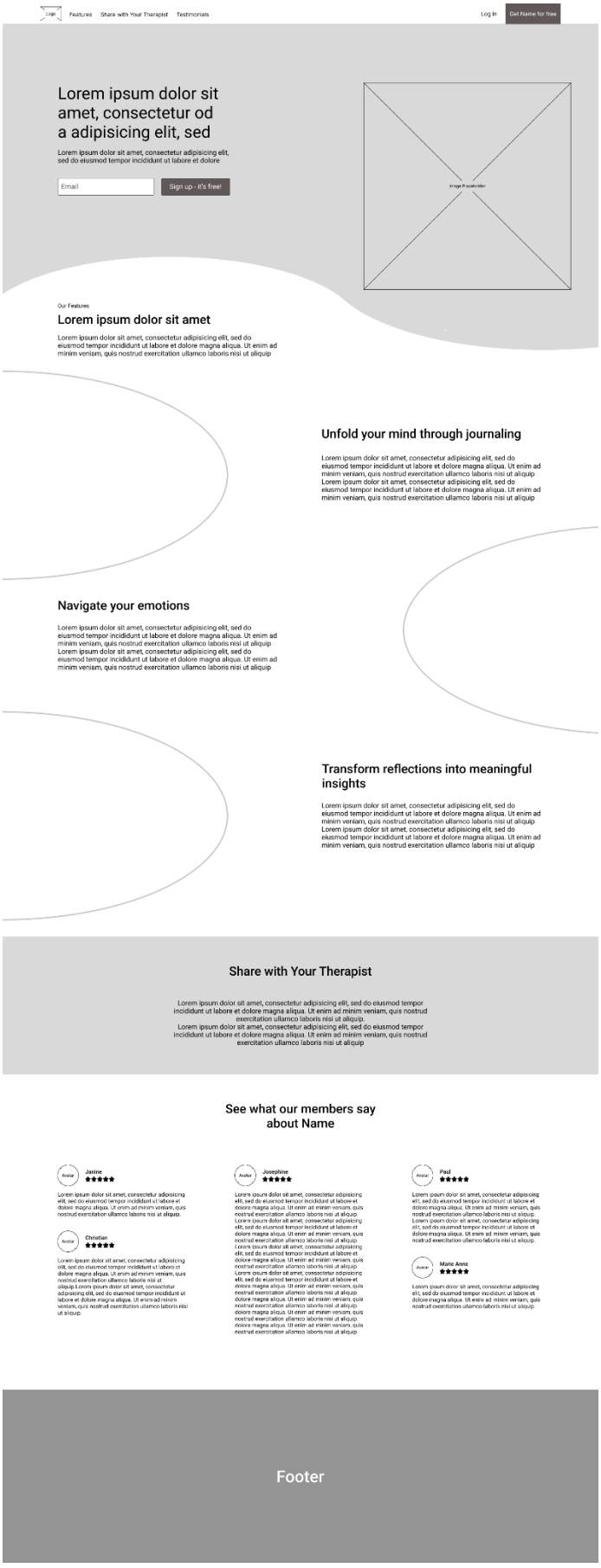

Figure 12. Wireframe - Homepage, guest (Option 2)

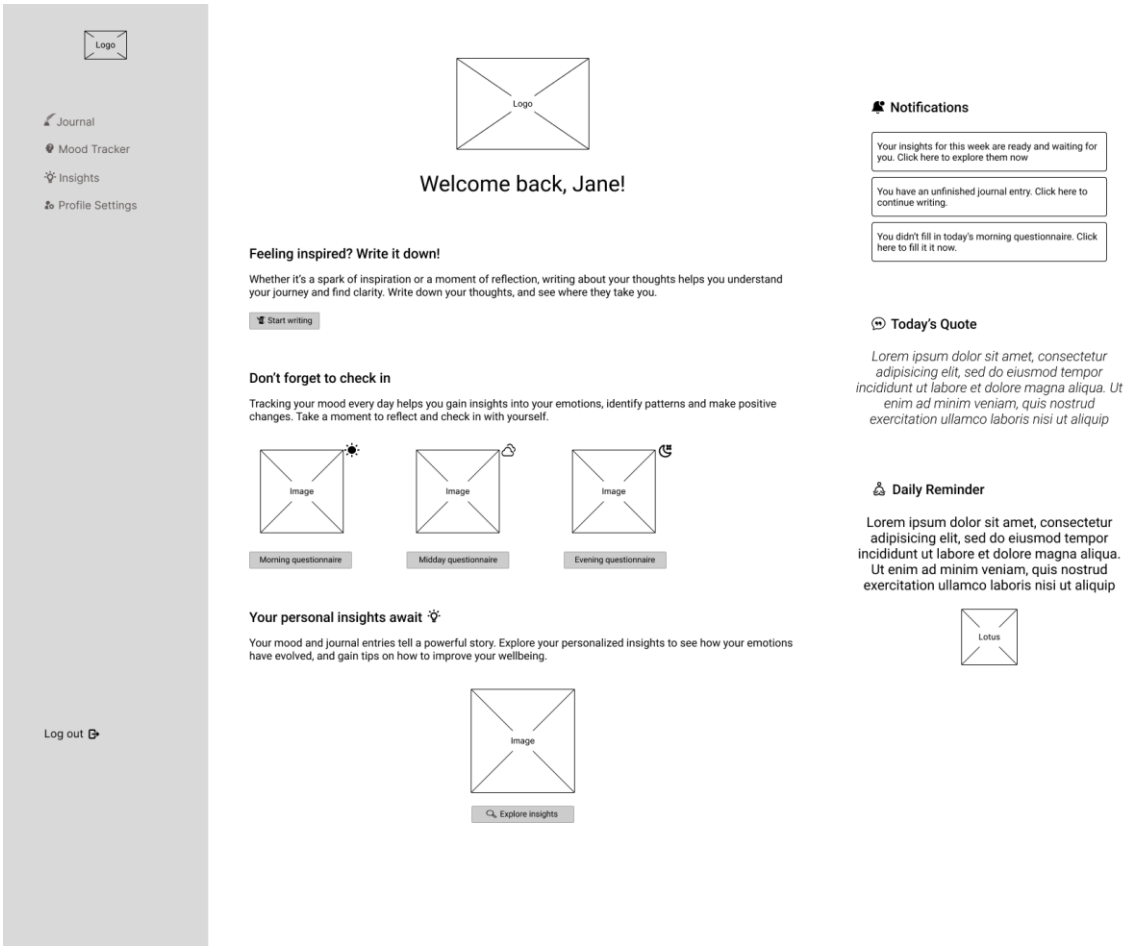

Figure 13. Wireframe - Homepage, logged in user

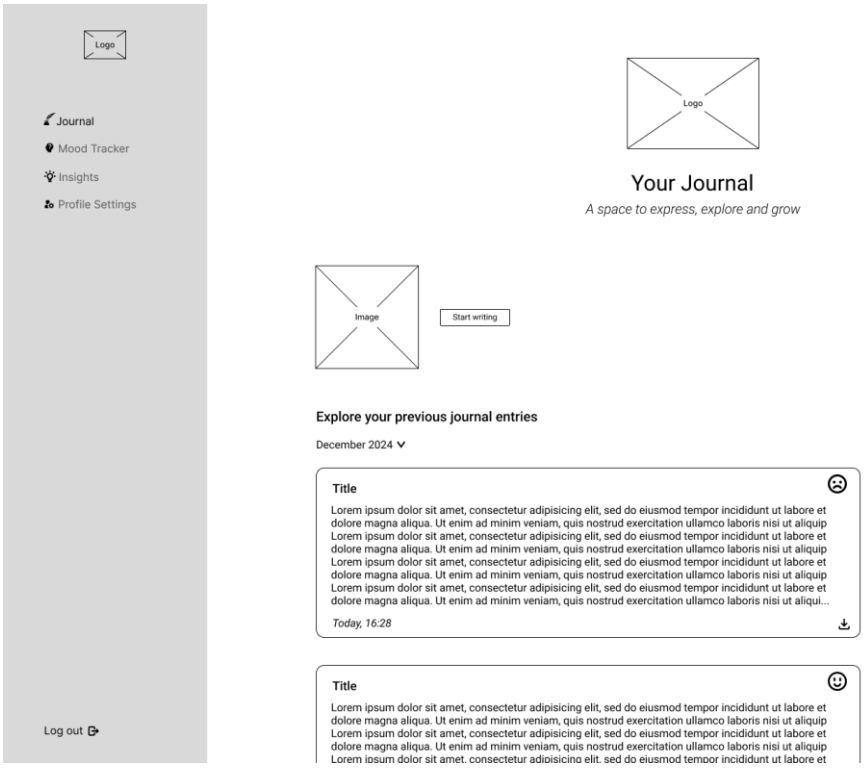

Figure 14. Wireframe - Journal page

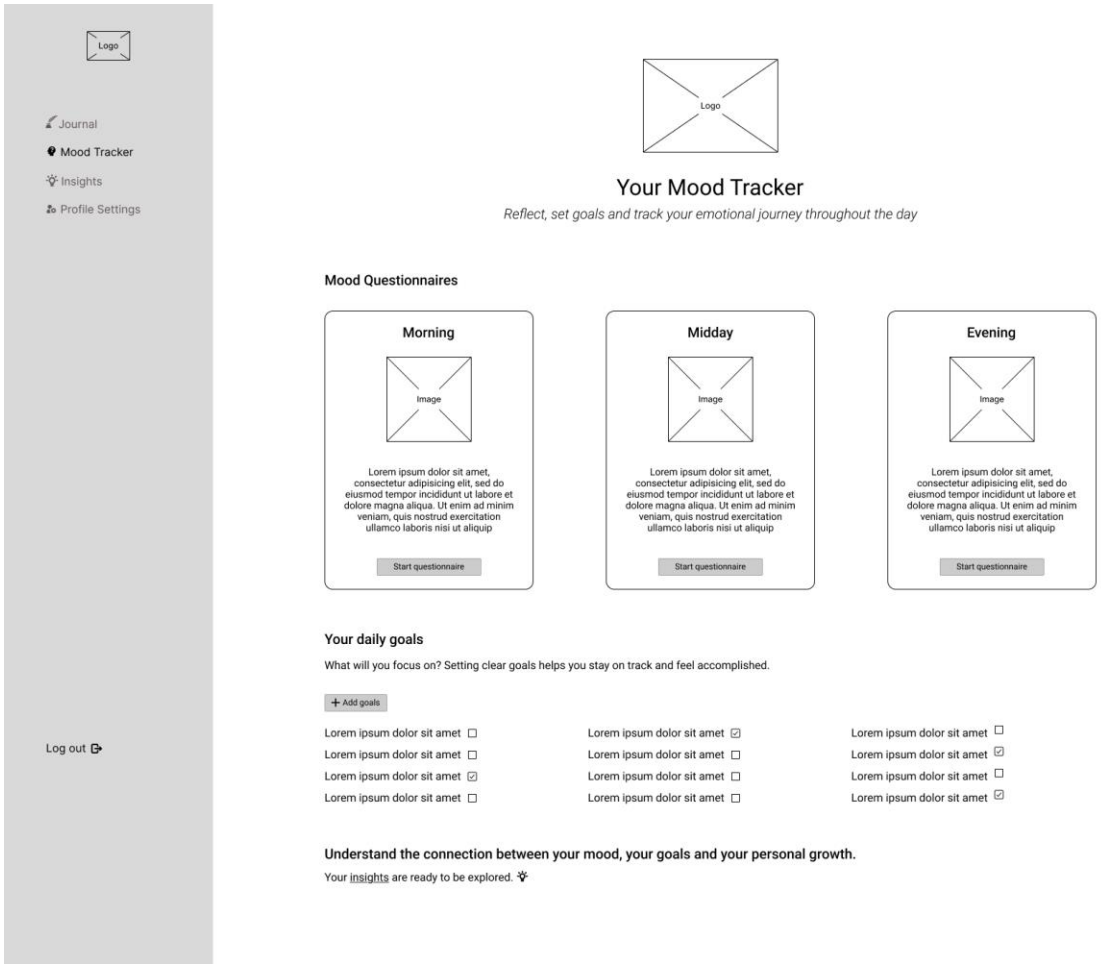

Figure 15. Wireframe - Mood Tracker page

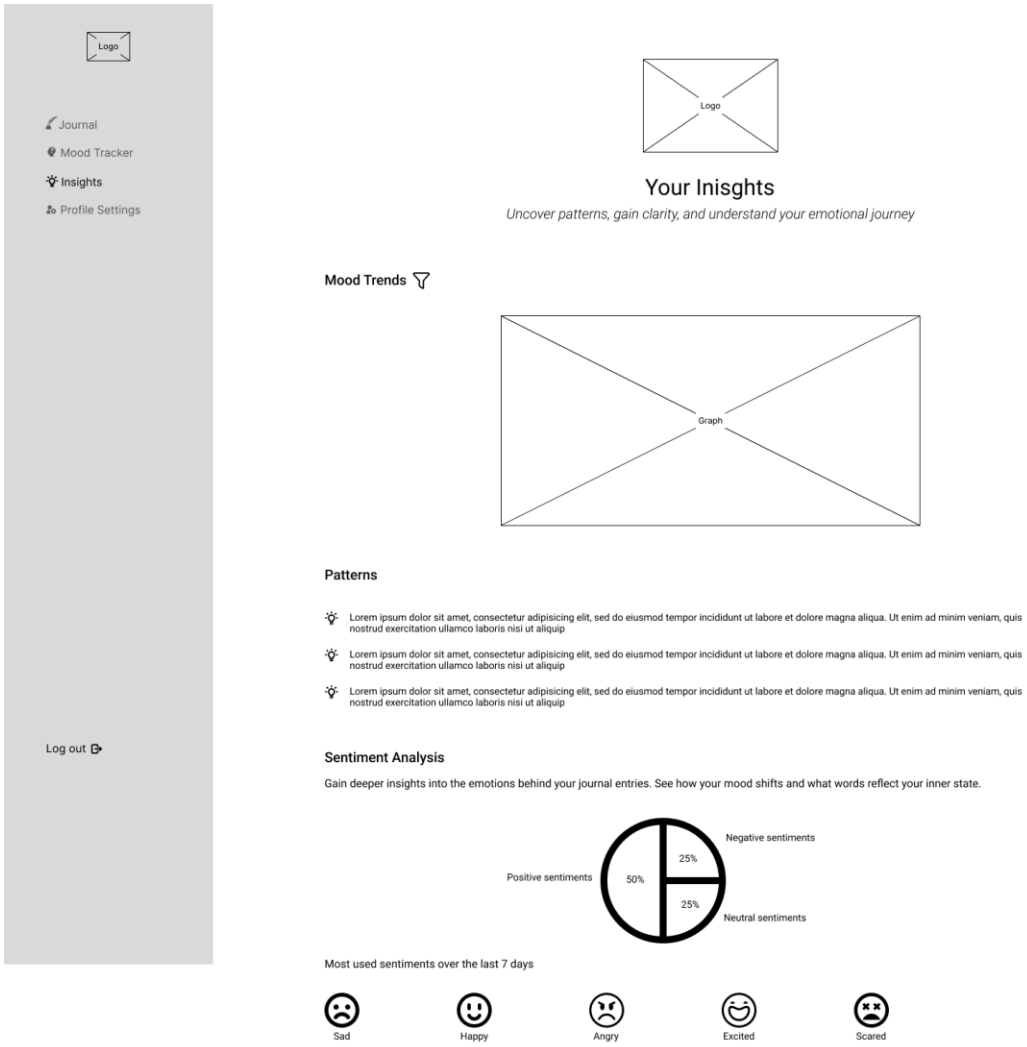

Figure 16. Wireframe - Insights page

Prototype

A prototype transforms the original concept into a digital form, representing a model of the proposed product. It acts as a small-scale example of a product, while allowing to make required changes to ensure that a product is viable before more time and resources are invested into it. It is also a tool for getting feedback from the potential users of the application, which then is used for making changes and fixing any issues before moving forward with the actual implementation of the product. (*What is a prototype?*)

I did not create a final prototype for my web application, due to the fact that I considered it to not be as high priority as the actual implementation itself. Considering the fact that I was planning to undertake a huge task with this project, including using technologies, tools, frameworks or libraries that I had not used before, I decided that skipping creating the prototype would save me valuable time and would allow be to focus more resources on the actual implementation of the project.

Although I did not create a final prototype, I did choose a palette of colors for my website, and I transformed some of the wireframes using these colors and using actual images and text that would be implemented on the platform itself. These can be seen in the images below.

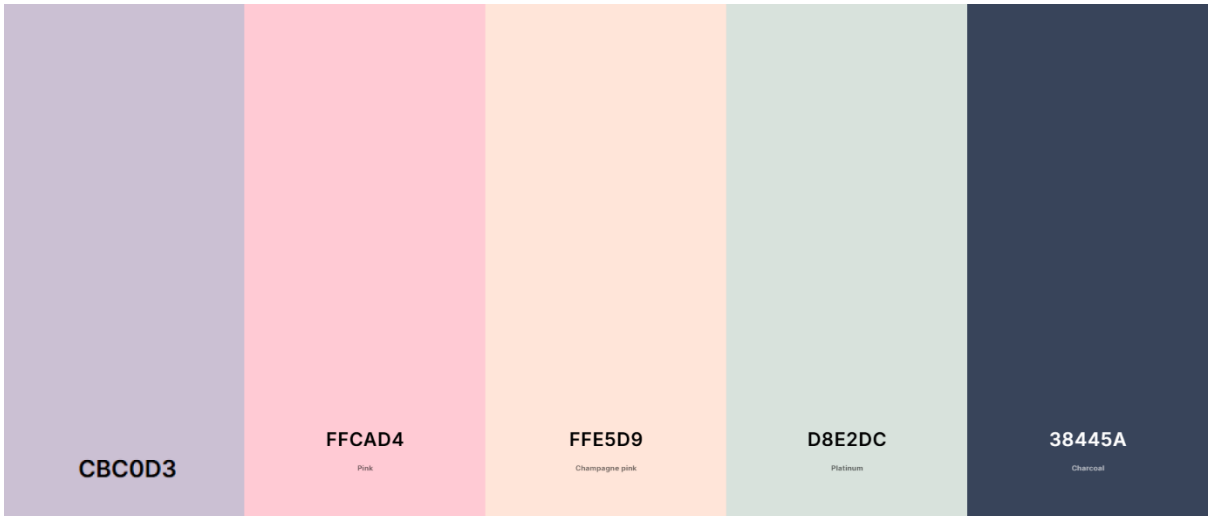

Figure 17. Color palette

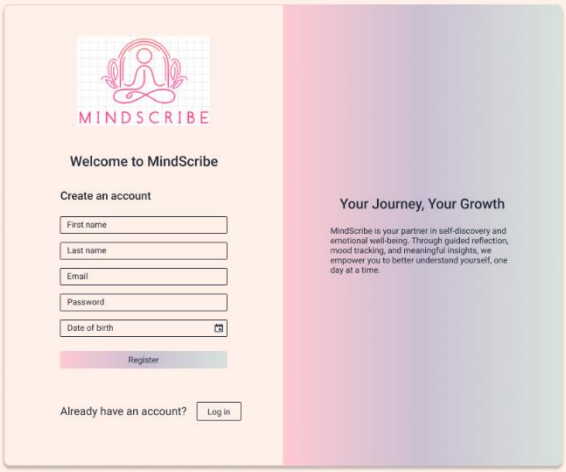

Figure 18. Prototype - Register page

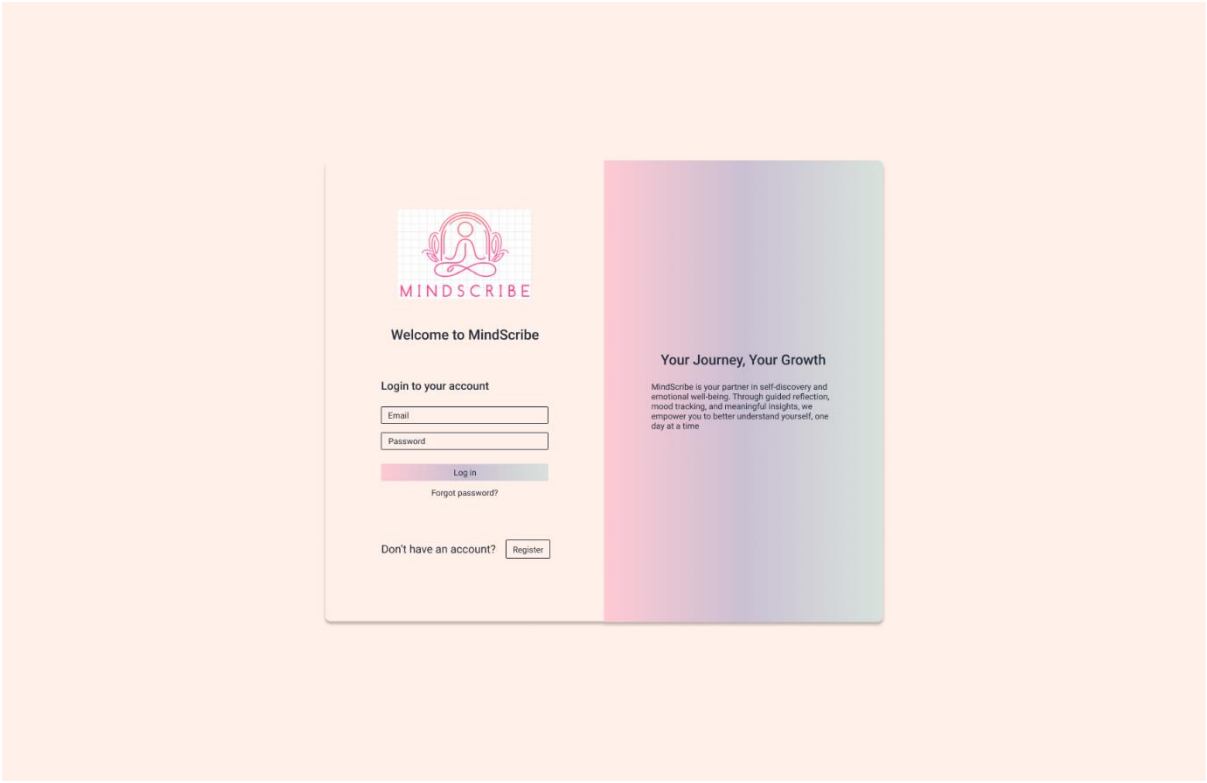

Figure 19. Prototype - Login page

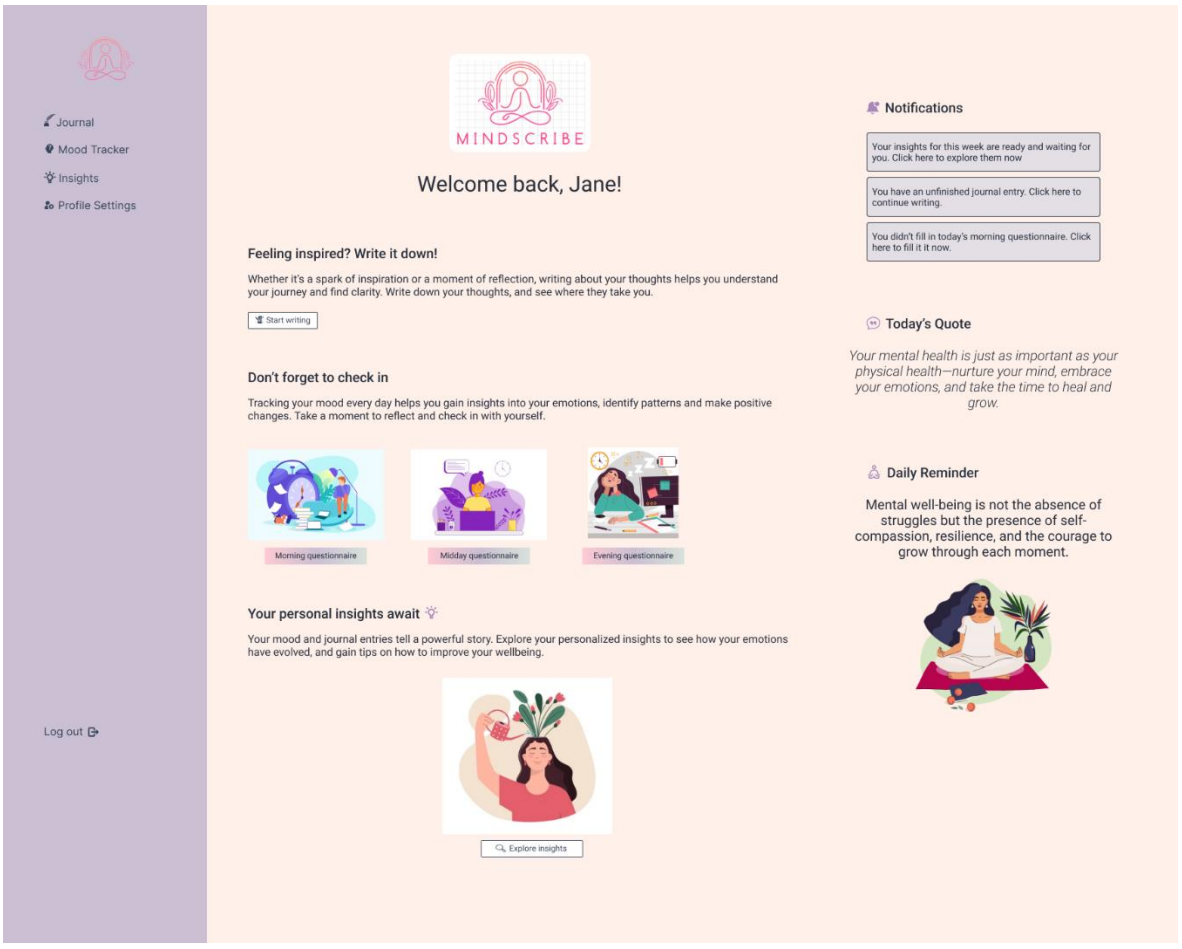

Figure 20. Prototype - Homepage, logged in user

The design that can be seen in this section is the final one that I decided on, prior to the actual implementation of the website. However, during the actual implementation, I had to make a few tweaks and changes and update the design to better fit the newly discovered needs or requirements of the application, which is why some of the designs show above are inconsistent with the final look of the frontend pages of the website.

Database – Design & Implementation

The next step in the process of creating, designing, and implementing my project was to come up with the type of database that I wanted to use, and then to design and implement it.

I decided to go for a hybrid approach, and implement a combination of two databases – one SQL database to store structured data (like user profiles, mood tracker entries, questions, questionnaires, journal metadata, etc.) and one NoSQL database to store unstructured data (like full journal entries and sentiment analysis results). I have never tried implementing this approach in previous projects, so it was a personal challenge that I wanted to fulfill.

For the SQL database, the options that I was considering choosing between were PostgreSQL and MySQL, which are two of the most popular and widely used open-source SQL-based database management systems (DBMSs), based on my research. Even though they both are widely used and offer good performance, rich features, and strong community support, I found through my research that it is better to use PostgreSQL when storing and querying structured data, along with unstructured or semi-structured data (like the options attribute from the Questions table, and the preferences attribute from the Users table, which are both of type JSON in my application). I also found out that PostgreSQL allows for more flexibility, and it is easier to scale or enhance project over time when using it, which might come in very handy when dealing with an application like the one that I was building. So, my final choice was PostgreSQL for the SQL database.

When it comes to the NoSQL database, the options which I was considering using for my project were MongoDB Atlas (the cloud-hosted version of MongoDB) and Firestore (Google Firebase Cloud Firestore), which are both document-based NoSQL databases, storing data in a document format. While both options would be appropriate for the needs of my project on a smaller scale, I found MongoDB Atlas to be the better choice for my project on a larger scale. Some reasons for that are the fact that the journal entries that I am planning to store in the database are text-heavy and semi-structured, and my research led to me believe that MongoDB Atlas has an edge over Firestore when it comes to complex querying, text-search capabilities and flexible schemas. From a scalability point of view, both MongoDB Atlas and Firestore handle large amount of data, but Firestore may incur higher costs, especially for frequent read-write operations on text-heavy data. So, my final choice for the NoSQL database was MongoDB Atlas.

For the SQL database that I used in my project, I decided to create an entity relationship diagram and a relational data model, to get a better understanding of my data and its structure, and to make the actual implementation of the database go faster. These diagrams will be discussed in what follows.

ER (Entity Relationship) Diagram

An Entity Relationship Model is a visual representation used for identifying the entities that need to be represented in the database, and how those entities are related to each other. Creating an ER diagram provides a standard solution for visualizing the data in our database in a logical way, while modeling real-world objects and the relation between the real-world objects. (Comment et al., *Introduction of ER model* 2024)

The ER diagram that I created based on my project's needs and requirements can be seen in Figure 21.

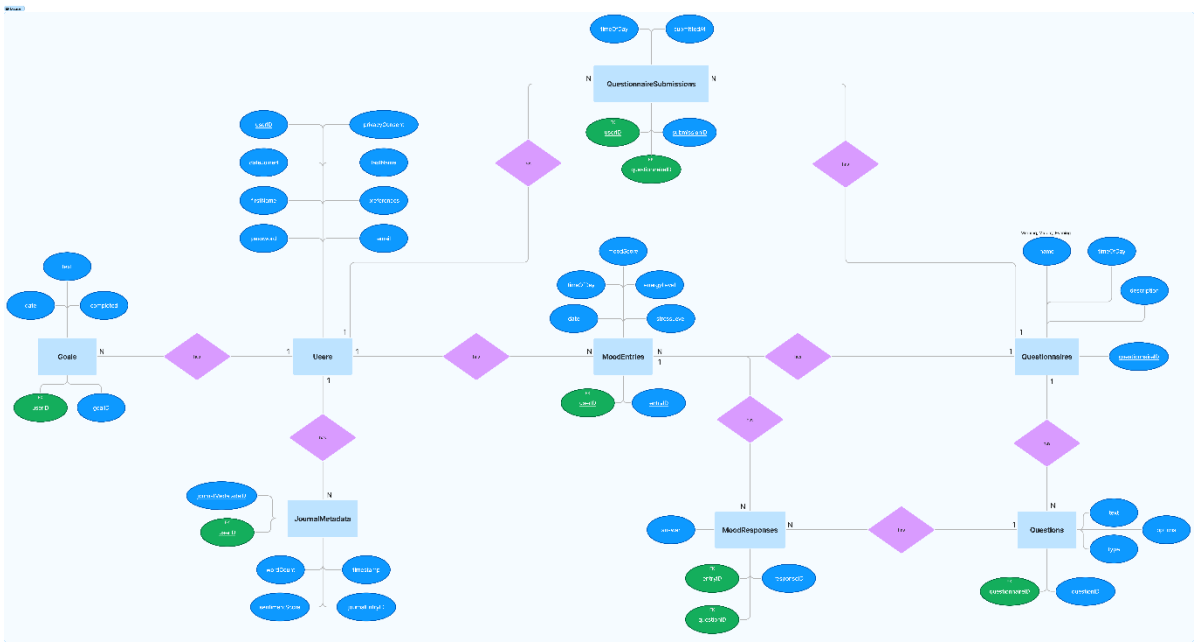

Figure 21. Entity Relationship Diagram

In order to create a robust system that ensures the desired functionality for my project, I decided to have the following entities in my relational database: Goals, Users, JournalMetadata, MoodEntries, MoodResponses, Questionnaires, Questions, and QuestionnaireSubmissions.

The Users entity represents the users of my application, who can interact with the system in various ways – like submitting the daily mood questionnaires, writing journal entries, setting daily goals and tracking their mood, mood trends and changes.

The Goals entity represents the daily goals that a user can set for themselves. These goals have the attribute "completed", which might be useful for tracking progress and generating user insights – like percentage of daily goal completion – in the future.

The Questionnaires entity is represented by the static questionnaires that can be reused by multiple users, across different dates and times of the day. The questionnaires are essential in assessing users' moods, both throughout the day and over a longer period of time.

The Questions entity is characterized by the individual questions that are found within each questionnaire. Each question is associated with one specific questionnaire, and has a predefined type (for example scale, text, yes/no, multiple choice).

The QuestionnaireSubmissions entity represents each instance of a user submitting a questionnaire. This is helpful in ensuring that each questionnaire corresponding to a specific time of day can only be submitted once per day by a user.

The MoodEntries entity represents each instance of a user filling out one of the mood questionnaires available to them daily. Each record in the MoodEntries table corresponds to a single instance of a user filling out a mood questionnaire, and is associated with multiple mood responses, one for each question in the respective questionnaire.

The MoodResponses entity represents the individual answers to each question in a mood questionnaire. Every time a user answers a question from a questionnaire, it generates a new entry in the MoodResponses table.

The JournalMetadata entity stores the metadata for each journal entry made by a user of the application. This entity is useful for allowing users to maintain their journaling history and for potentially analyzing it and including it in the personalized insights of each user.

RDM (Relational Data Model)

The next step after finalizing the Entity Relationship diagram was to convert it into a Relational Data Model (RDM). An RDM showcases a collection of tables, including the attributes and type of these attributes for each table, and the relationships between these tables. (GeeksforGeeks, *Relational model in DBMS* 2024)

The RDM that I designed based on my ER diagram can be seen in Figure 22.

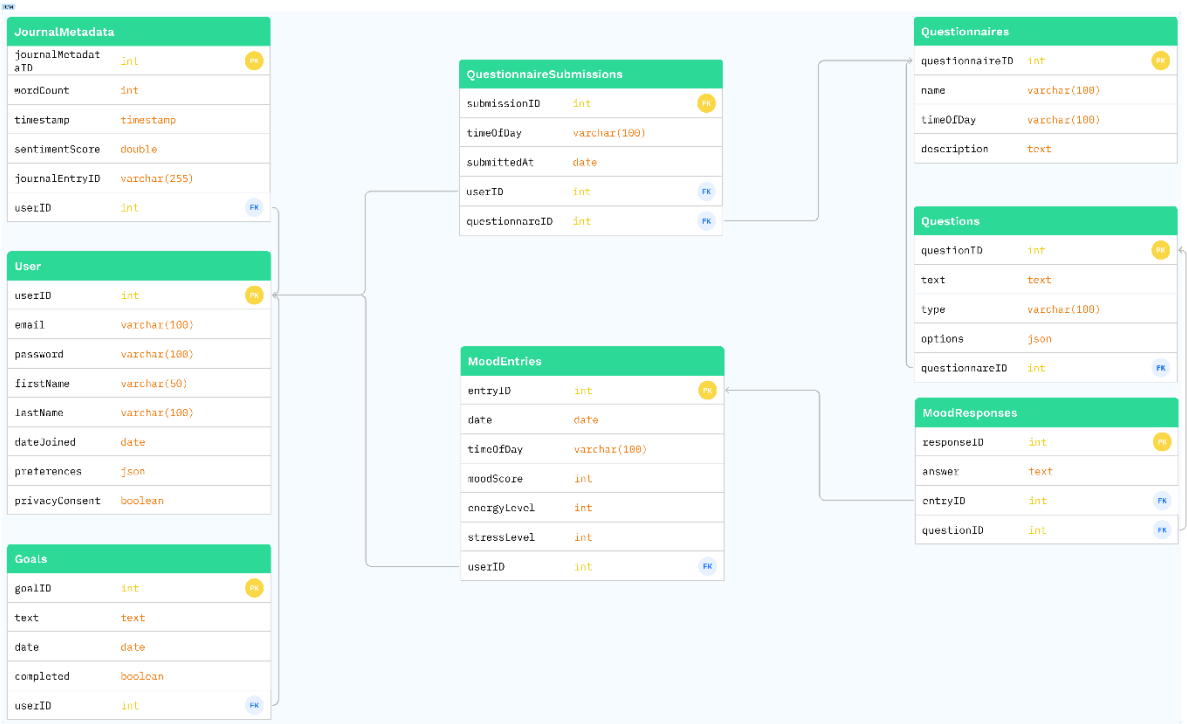

Figure 22. Relational Data Model

Each of the tables corresponds to one entity from my ER diagram. The relationships between these tables are established through the foreign key constraints, as can be seen in my RDM: a foreign key is pointing to the key that is a primary key in its original table, maintaining referential integrity between the related data.

Since all the relationships between the tables in my ER diagram were either one-to-one or one-to-many, I did not need to create any additional tables (names junction tables), which would have been needed in the case of many-to-many relationships.

The RDM that I created is designed to minimize data redundancy and to ensure data consistency, which was achieved by adhering to normalization principles. Normalization is defined to be the process of organizing data in the database and is used in order to minimize redundancy from a relation or a set of relations. (*DBMS normalization: 1NF, 2NF, 3NF and BCNF with examples - javatpoint*)

The normal form that I wanted to achieve for my database was the Third Normal Form (3NF).

The First Normal Form (1NF) states that each table cell should contain only atomic values, which are the smallest units of data, and which cannot be divided into other smaller parts. I ensured this in my RDM by already dividing attributes that could potentially be divided into smaller parts (for example, in the Users table, instead of having an attribute called "name", I created the attributes "firstName" and "lastName").

The Second Normal Form (2NF) states that a relation (table) needs to be in 1NF and that all attributes that are not keys have to be fully dependent on the primary key. These requirements are fulfilled in my RDM, since I already established that 1NF is

fulfilled and due to the fact that every non-key attribute in any of the tables is dependent on the primary key of that respective table and only makes sense in the context of that table.

The Third Normal Form (3NF) states that a table needs to be in 2NF and all the non-key attributes are dependent on the primary key but, at the same time, they are not dependent on other non-key attributes (so they are dependent on the primary key only). I already established in the previous paragraph that 2NF is fulfilled in my RDM. By looking at the tables in my RDM and at their respective attributes, it is clear that the second condition of the 3NF is fulfilled as well. Therefore, I have achieved the Third Normal Form in my Relational Data Model.

PostgreSQL Implementation

In this section, I will explain in more detail how I implemented the PostgreSQL database in my project, as a next step after designing the diagrams that I talked about in previous subchapters of this chapter. I will provide more information regarding the setup of the database connection, the creation of the necessary models, the configuration of the associations between the tables corresponding to my RDM, and seeding initial data in some of my database tables.

Database setup

The first step in the implementation of the database was to set up PostgreSQL and to configure a connection to my database. In order to facilitate the communication between my application and the PostgreSQL database, I used Sequelize ORM (Object-Relational Mapping). An ORM is a helper that allows developers to work with data in a database using objects, acting as a bridge between the application's code and the database. Sequelize is a Node.js ORM library that allows to work with different databases like MySQL, PostreSQL and SQLite. The benefits of using Sequelize are the facts that we can define models (how things look), we can introduce data validation, we can define associations (connections, for example between different models), and we are able to change things if needed through database migrations. (Singh, *"Node.js database magic: Exploring the powers of sequelize Orm"* 2023)

The connection details, such as the database user, password, host, port and name are passed into Sequelize via environment variables that I defined, therefore ensuring security and flexibility in the case where another database might be used, and these details would change. This can be seen below, in Figure 23.

```
// Setup Sequelize connection
const sequelize = new Sequelize(`postgres://${process.env.POSTGRESQL_DB_USER}:
    ${process.env.POSTGRESQL_DB_PASSWORD}@${process.env.POSTGRESQL_DB_HOST}:
    ${process.env.POSTGRESQL_DB_PORT}/${process.env.POSTGRESQL_DB_NAME}`, {
    dialect: 'postgres',
});
```

Figure 23. Setup Sequelize connection

Model definitions

Based on the previously created ER diagram, I defined several models in my code using Sequelize, each model corresponding to an entity from the diagram. An example of one model from my code is the MoodReponse model, which can be seen in the image below.

```
const { Sequelize, DataTypes } = require("sequelize");
const { sequelize } = require("../setup");

const MoodResponse = sequelize.define(
  "MoodResponse",
  {
    responseID: {
      type: DataTypes.INTEGER,
      primaryKey: true,
      autoIncrement: true,
    },
    answer: {
      type: DataTypes.TEXT,
      allowNull: false,
    },
  },
  {
    tableName: "mood_responses",
    timestamps: false,
  }
);

module.exports = MoodResponse;
```

Figure 24. Example of model - MoodResponse model

After configuring all the individual models, I also configured the relationship between these models, corresponding to the relationships between the entities in the database design diagrams, in the index.js that can be found in the same folder as the other PostgreSQL models. The associations defined in the index.js file can be seen in Figure 25.

```
// Associations

// User - Goal
Goal.belongsTo(User, {foreignKey: 'userID', onDelete: 'CASCADE'});
User.hasMany(Goal, {foreignKey: 'userID', onDelete: 'CASCADE'});

// User - JournalMetadata
JournalMetadata.belongsTo(User, {foreignKey: 'userID', onDelete: 'CASCADE'});
User.hasMany(JournalMetadata, {foreignKey: 'userID', onDelete: 'CASCADE'});

// User - MoodEntry
MoodEntry.belongsTo(User, {foreignKey: 'userID', onDelete: 'CASCADE'});
User.hasMany(MoodEntry, {foreignKey: 'userID', onDelete: 'CASCADE'});

// MoodEntry - MoodResponse
MoodResponse.belongsTo(MoodEntry, {foreignKey: 'entryID', onDelete: 'CASCADE'});
MoodEntry.hasMany(MoodResponse, {foreignKey: 'entryID', onDelete: 'CASCADE'});

// Question - MoodResponse
MoodResponse.belongsTo(Question, {foreignKey: 'questionID', onDelete: 'CASCADE'});
Question.hasMany(MoodResponse, {foreignKey: 'questionID', onDelete: 'CASCADE'});

// Questionnaire - Question
Question.belongsTo(Questionnaire, {foreignKey: "questionnaireID", onDelete: "CASCADE"});
Questionnaire.hasMany(Question, {foreignKey: "questionnaireID", onDelete: "CASCADE"});

// User - QuestionnaireSubmission
QuestionnaireSubmission.belongsTo(User, {foreignKey: "userID", onDelete: "CASCADE"});
User.hasMany(QuestionnaireSubmission, {foreignKey: "userID", onDelete: "CASCADE"});

// Questionnaire - QuestionnaireSubmission
QuestionnaireSubmission.belongsTo(Questionnaire, {foreignKey: "questionnaireID", onDelete: "CASCADE"});
Questionnaire.hasMany(QuestionnaireSubmission, {foreignKey: "questionnaireID", onDelete: "CASCADE"});
```

Figure 25. index.js – Associations

Database Synchronization

Once I defined all the necessary models and associations, I used the sync () method from Sequelize. This method ensures that the database schema reflects the model definitions that I implemented at all times. The code *force: false* means that the tables are created or updated without dropping any existing data.

```
// Sync models to the database

await sequelize.sync({ force: false }); // Change `force: false` to `force: true` for recreating tables
console.log('All sequelize models were synchronized successfully.');
```

Figure 26. Database synchronization

Seeding data

In order to populate my PostgreSQL database with some initial data for the three mood tracker questionnaires – Morning questionnaire, Midday questionnaire and Evening questionnaire – I combined an SQL file (seed_data.sql) with a Node.js script file (insertData.js).

The seed_data.sql file contains the SQL statements designed to populate the desired tables of the database (questionnaires and questions), with the data predefined by me. This file contains *INSERT INTO* queries.

The insertData.js file is a script for seeding the database. This executes the seed_data.sql file and inserts the predefined data into the database. More specifically, the script connects to the PostgreSQL database, reads the content of the seed_data.sql file, and runs the SQL queries contained in the file. To run the script, I used Node.js - *node insertData.js*.

MongoDB Atlas Implementation

In this following section, I will go through the implementation of MongoDB Atlas in my project. I will focus on the setup of the database connection, model definitions, and integration with the rest of the application.

Database setup

The first step of integrating MongoDB Atlas in my application was to set up a connection to the database. This is a pretty simple and straightforward process, since MongoDB Atlas represents a cloud solution for managing MongoDB databases, therefore eliminating the need for manual setup or maintenance of the database server. The connection was configured easily, using a MongoDB URI that contains the necessary connection details, like username, password, and database cluster information. Similarly to the implementation of the PostgreSQL database, I also environment credentials in this case to securely manage my database credentials.

In order to establish a connection to MogoDB, I used the Mongoose library, which is an ORM library for MongoDB and Node.js. This ORM manages relationship between data, as well as providing schema validation and being used to translate between objects in the code and the representation of these objects in a MongoDB database. (freeCodeCamp, *Introduction to mongoose for mongodb* 2018)

The code for connecting to my MongoDB Atlas database can be seen in Figure 27.

```
const connectToMongoDB = async () => {
    try {
        await mongoose.connect(process.env.MONGODB_URI);
        console.log('Successfully connected to MongoDB!');
    } catch (error) {
        console.error('Error connecting to MongoDB:', error);
        throw error;
    }
};
```

Figure 27. Setup MongoDB connection

Model definitions

To interact with MongoDB, I created two Mongoose models -journalEntryModel and sentimentAnalysisModel. Mongoose allows to define schemas for the data and to interact with the database in a more effective way. The two models that I created can be seen in Figure 28 and Figure 29.

```
// Journal Entry Model

const mongoose = require('mongoose');
const Schema = mongoose.Schema;

const journalEntrySchema = new Schema({
  userID: { type: String, required: true },
  content: { type: String, required: true },
  timestamp: { type: Date, default: Date.now },
});

const JournalEntry = mongoose.model('JournalEntry', journalEntrySchema);

module.exports = { JournalEntry };
```

Figure 28. Journal Entry model

```
// Sentiment Analysis Model

const mongoose = require('mongoose');
const Schema = mongoose.Schema;

let sentimentAnalysisSchema = new Schema({
    entryID: {type: mongoose.Schema.Types.ObjectId, ref: 'JournalEntry'},
    sentimentScore: {type: Number},
    analysisDate: {type: Date, default: Date.now}
});

module.exports = mongoose.model('SentimentAnalysis', sentimentAnalysisSchema);
```

Figure 29. Sentiment Analysis model

Integration with server

In my project, both PostgreSQL and MongoDB databases are integrated in the main application server, which is built using Express.js. Express.js is a web framework for Node.js that is designed to build web application and APIs. By using this framework, I was able to save time on tasks related to building a backend from scratch for my application in Node.js (tasks like setting up ports to route handlers, writing boilerplate code, and so on). (Codecademy, *What is express.js?*)

In the server.js file from my application, I used separate functions to connect to PostgreSQL and MongoDB asynchronously, which ensures that the server is only

going to start accepting requests once both of the databases are connected. The connection to PostgreSQL is managed via Sequelize, and the connection to MongoDB is handled by Mongoose. After establishing connections to both databases successfully, the server starts listening to incoming requests. The code for starting my server, which includes connecting to both databases, can be seen in the image below.

```
const startServer = async () => {
  try {
    await connectToMongoDB();
    await connectToPostgreSQLDB();

    // Sync the Sequelize models with the database
    await sequelize.sync({ force: false }); // can change this to `force: true` during development
    console.log("All Sequelize models were synchronized successfully.");

    const PORT = process.env.PORT || 5000;

    // Start the server after successful connection to the databases
    app.listen(PORT, function () {
      console.log(`Server is running on Port: ${PORT}`);
    });
  } catch (error) {
    console.error("Error during database connection", error);
  }
};

startServer();
```

Figure 30. Start server code snippet

API (Application Programming Interface)

This section of the report will be dedicated to providing more details regarding the API that I implemented in order to be able to interact with the backend of my application. An API is a mechanism that enabled two software components to communicate with each other, through a set of specific definitions and protocols. (*What is an API? - application programming interface explained - AWS*)

As will be discussed into more detail in following sections of the report, I used Express.js, a Node.js application wjatframework, to build my desired API. My goal was to implement an API that adheres to RESTful design principles, so a REST (Representational State Transfer) API. Some of these constraints that are characteristic to a REST API are: a client-server architecture, requests from the client to the server are being managed through HTTP, a stateless client-server communication.

The API that I designed and implemented represents the core component of my application, since it is responsible for managing data operations, user authentication, and the business logic. The API interacts with both the MongoDB and the PostgreSQL databases that I have implemented in my project, and it supports various operations – such as user authentication, CRUD operations for resources from my application like journal entries, questionnaires, daily goals, personalized insights and so on, and data retrieval by user.

Routes and Endpoints

The REST API that I created for my project has a modular structure, and it contains different handler for different resources. The routes are separated in my code implementation in distinct files, each of them representing one specific resource, such as authentication, goal, journal, insights, etc., and each of them managing operations for their respective resource. The routes are using the functions from the respective controller in order to handle logic, and a token verification middleware for security reasons, which further ensures modularity.

An example of routes from my application can be seen in Figure 31 . This screenshot shows the authentication related routes, which handle user registration, login, logout and checking the login status.

```
const router = require('express').Router();
const authController = require('../controllers/authController');

// Register
// /api/auth/register
router.post('/register', authController.register);

// Login
// /api/auth/login
router.post('/login', authController.login);

// Logout
// /api/auth/logout
router.post('/logout', authController.logout);

// Get login status
// /api/auth/login-status
router.get('/login-status', authController.getLoginStatus);

module.exports = router;
```

Figure 31. Authentication routes

API Calls

In the frontend of my application, API calls are made using the Axios library in order to interact with the backend of the application (the server).

In what follows, I will provide an example of how a request is sent from the client to the server, and how this request is processed and handled.

In Figure 32, the frontend calls the API to fetch journal entries for the logged in user. Axios is used to send a GET request to the specific endpoint that handles this logic from the backend (*/api/journal/entries*), with the authentication token header as part of the request, to ensure that only an authorized user can access this data. Once the response is received from the server, *fetchedEntries* are set, which will then trigger the rendering of the journal entries corresponding to the logged in user in the frontend of the application.

```
const fetchEntries = async () => {
  if (!userID) return;
  try {
    const response = await axios.get(
      `${process.env.REACT_APP_API_BASE_URL}/api/journal/entries/${userID}`,
      {
        headers: {
          "auth-token": token,
        },
      }
    );
    const fetchedEntries = response.data;
    setEntries(fetchedEntries);
```

Figure 32. Journal frontend API call

On the backend side, the request from the frontend is received by the appropriate route handler that is defined in the file that includes all the routes corresponding to journal operations. These routes can be seen in the screenshot below.

```
const router = require('express').Router();
const { model } = require('mongoose');
const journalController = require('../controllers/journalController');

const {verifyToken} = require('../middlewares/tokenVerification');

// Save journal entry
// api/journal/entries
router.post('/entries', verifyToken, journalController.saveJournalEntry);

// Fetch journal entries by userID
// api/journal/entries/:userID
router.get('/entries/:userID', verifyToken, journalController.getJournalEntries);

module.exports = router;
```

Figure 33. Journal routes

The journal controller contains the logic of fetching the journal entries. This controller processes the request by querying the two databases – MongoDB for journal entries and PostgreSQL for journal metadata – and then send back the combined data to the frontend, as can be seen in Figure 34.

```
const getJournalEntries = async (req, res) => {
  const { userID } = req.params;

  try {
    const journalEntries = await JournalEntry.find({ userID }).sort({
      timestamp: -1,
    });
    const journalEntryIDs = journalEntries.map((entry) => entry._id.toString());
    const metadata = await JournalMetadata.findAll({
      where: {
        userID,
        journalentryid: {
          [Op.in]: journalEntryIDs,
        },
      },
    });

    const metadataMap = metadata.reduce((acc, meta) => {
      acc[meta.journalentryid] = meta;
      return acc;
    }, {});

    Complexity is 3 Everything is cool!
    const combinedEntries = journalEntries.map((entry) => {
      const entryID = entry._id.toString();
      return {
        ...entry.toObject(),
        sentimentScore: metadataMap[entryID]?.sentimentScore || 0,
      };
    });

    res.status(200).json(combinedEntries);
  } catch (error) {
    console.error("Error fetching journal entries:", error);
    res.status(500).json({ error: "Something went wrong!" });
  }
};
```

Figure 34. Journal controller

Version Control

In this section, I will describe the version control of my application in more detail. Version control is an importance practice from software development that I used for tracking and managing any changes made to the code by the developers working in it (GitLab, *What is version control?* 2023). It is a way to ensure that everyone collaborating on the project always has the most recent version of the code and to keep track of who made what changes in a much easier and faster way.

Even though I was working on this project alone, I still needed to practice version control, in order to track the changes that I made to the code and to potentially revert any changes, if needed. In my project, I used Git and GitHub for version control.

Git is a version control system used for tracking code changes and collaborating on code. Git deals with repository managements, cloning projects to work on a local copy, controlling and tracking changes with Commit, allowing developers to work on different parts and version of projects with Branch and Merge, pulling the latest version of the project using Pull and pushing local updates to the main project using Push. (*W3schools.com*)

GitHub is a cloud-based platform where code can be stored in Git repositories, and which allows collaborative working. (*About github and git*)

The first step in the version control process was to create a repository on GitHub.

I used GitHub Desktop, an interface tool which makes it easier to perform Git operations like pushing changes or pulling updates, managing branches, and so on. The reason why I mostly used GitHub Desktop, as opposed to using Git from the command line, is because its interface is more intuitive.

Throughout the development server, I was only working on the *main* branch of my application, since I was developing the code on my own and I did not need anyone else to verify or approve my code before pushing it to the *main* branch. If I were to follow a better practice for branching, I would create a new branch for each specific feature, task or bug fix, which is a strategy that is especially useful when collaborating with other developers on a project. This strategy ensures that the *main* branch is the one branch that is always stable and has only code that has been thoroughly tested and is ready to be pushed to production. If I could change one thing related to version control during the development of this project, it would be using this branching strategy.

With the use of Git and GitHub, along with GitHub Desktop, I was provided with a robust version control system that ensured the development process went smoother and allowed me to keep track of changes and revert changes without any problems, when necessary.

GDPR (General Data Protection Regulation)

GDPR (General Data Protection Regulation) is a data privacy and security law established by the European Union, which includes requirements for organizations around the world, with the purpose of guarding the personal data of its citizens. (*What is GDPR, the EU's new Data Protection Law?* 2024)

Since my application is based in the European Union and collect, processes and stores data of people that reside in the European Union, in a real-life situation, GDPR compliance would be a very important aspect for an application like the one that I created. In the development of my application, I researched key aspects of GDPR and tried to adhere to them and respect them in the best possible way that I could, given the context of the project and my limited experience with this issue.

In what follows, I will describe in more detail the main principles of GDPR that I managed to successfully implement in my project.

Lawfulness, Fairness and Transparency

This principle of the GDPR states that all data that is collected from the suers should be collected and processed in a way that is lawful and transparent, and it is for a legitimate purpose.

In my application, I provided a clear privacy policy that describes what data is collected and for what purpose, how long the data is stored in the application, security measures and users' rights. I allow users to explicitly consent to this privacy policy when they register by checking the consent box, and I keep records on user consent in the PostgreSQL database of my application.

Data Minimization

This principle of the GDPR states that only the data that is necessary for the intended purpose of the application is to be collected.

In my application, I am collecting only the personal information that is relevant for authentication (like email and password), and for the delivery of the features present on my platform, like history of journal entries, daily goals, personalized insights and so on. I am not collecting unnecessary data that is not required, for example date of birth, personal address, or phone number.

Right to Access and Deletion

This principle of the GDPR states that users should have access to their data and that they should be able to delete their stored data form the application.

The users of my application have the possibility to access their personal data, such as journal entries, and they have the option to request the deletion of their data, by choosing to delete their account.

Data Protection Measures

Another principle of the GDPR is that the personal data of the users of the application should be protected against unauthorized access.

Data protection measures include encryption of sensitive data like passwords, and the use of tokens that are secure. In my application, I am using bcrypt to hash user passwords in the database and I am implementing a JWT token-based authentication, which adds a layer of security.

Code Implementation

This chapter of the report will describe into more detail the actual implementation of the code for creating the desired web application. Aspects regarding both frontend and backend code implementation will be discussed, as well as a few examples of some interesting or worth mentioning code implementations. There will also be a section where the most important tools, technologies and frameworks used throughout the project will be described.

Tools, Technologies, and Frameworks

This section will be used to briefly mention and describe the most important tools, technologies and frameworks that were used in order the achieve my goals of implementing a fully functional web application.

Task Management

Creating new tasks, editing task deadlines, sorting tasks in the corresponding category (To do, Doing or Done) was done in **Trello**. Every week, I set a goal for myself of what I wanted to achieve for both the code and the report, and then created the necessary tasks for achieving those goals. If there were any tasks that were not finished from one week to another, I moved them to the following week.

Design and Planning

When it comes to the design process, I used **Figma** to create both the wireframes and the "prototype" before implementing the actual application. Figma is very popular among UX/UI designers, and it offers free premium access to students, which was very useful for my project.

I also used Figma to create a MoSCoW analysis and the database diagrams (ER diagram and RDM). Apart from the design tool, Figma also offers various templates for designing project, managing projects, creating presentation, etc. Among these, there were templates available for the diagrams that I needed to create.

Version Control

Version control was handled through **Git**, **GitHub** and **GitHub Desktop**, like I mentioned in the chapter dedicated to this specific subject.

Databases

For the SQL database, I chose **PostgreSQL** and I used it with the **Sequelize** ORM in order to simplify the interactions with the database, to be able to define database models and to manage the database migrations that I created.

For the NoSQL database, I opted for **MongoDB**, and I used with the **Mongoose** library in order to define database schemas and to perform interactions with the database.

IDE

The Integrated Development Environment (IDE) that I used for developing the code is **Visual Studio Code** (**VS Code**). The reason behind this is the support for many programming languages that VS Code offers, the integrated terminal, the big variety of community developed plugins that are easy to install, the support for debugging and the syntax highlighting feature.

Frontend Development

The frontend application was built using **React.js**, which is a JavaScript library that allows users to build user interfaces. React.js is component-based, which means that it allows for reusable components that manage their own state to be built, and then reused to create more complex UIs. (*React – a JavaScript library for building user interfaces*)

The library that I used in order to handle client-side routing and to enable the navigation between different pages of the application is **React Router DOM**.

To style the frontend of my platform, I used a combination of **Tailwind CSS** and **Tailwind Elements**. Tailwind CSS follows a utility-first approach, which makes the development process much faster and styling the code is very easy and fast. Tailwind Elements provided built-in components that I was able to integrate seamlessly in my code with Tailwind CSS. One example of a page where I used Tailwind Elements are the Login and Register pages, more precisely the two forms used on those pages.

In order to handle HTTP requests from the frontend to the backend, I used **Axios**. Axios is a JavaScript library that allows to handle HTTP requests and it provides and API for making requests (GET, POST PUT and DELETE requests).

For the text editor needed on the page where users can write a new journal entry, I used a library named **Draft.js**. This library fit seamlessly into my React application and it was very easy to implement, without having to think about issues like rendering, input behavior, and so on.

Backend Development

The backend was built using **Node.js**. Node.js is a JavaScript runtime which is designed to help build network application. It provides an efficient runtime environment for the server to execute JavaScript. (*Node.js - about node.js*®)

Express.js, a web framework for Node.js, was also used to create a RESTful API. This framework helped to simply the process of handling routes and handling HTTP requests.

For authentication and session management, I used **JWT** (**JSON Web Tokens**). The use of JWT helped to ensure that the communication between the client and the server is secure.

In order to make it easier to handle JSON and data encoded in the URL, I used **Body-parser**.

During the development process, I used **Nodemon**. Nodemon automatically restarted the server whenever I made any changes to any file, therefore saving me the time and effort of manually restarting the server every time I made changes to files.

Sentiment Analysis and Insights

To analyze sentiment from the journal entries that users make, I used **Sentiment.js**. Even though it is not the most complex module, it was the one I could find to work with JavaScript and not cost money to use.

The libraries that I used for creating the interactive charts on the Insights page of my application are **Chart.js** and **React-Chartjs-2**. While Chart.js provided the main functionality for charting, React-Chartjs-2 provided a React wrapper that made integration within my project much easier.

Testing

Postman was used to test the API endpoints that I implemented. This helped me ensure that the backend was working correctly and the response that I was getting for my requests were the expected ones.

In order to implement backend testing, I used a combination of the **Mocha** test framework and the **Chai.js** assertion library. Mocha is a JavaScript test framework that runs on Node.js and in the browser, which makes asynchronous testing easier to perform (*The fun, simple, flexible JavaScript test framework*). Chai.js is an assertion library for Node and the browser (*Chai assertion library*). The two are very easy to pair together, in order to create both simple and complex backend test cases.

Deployment

My web application is deployed on **Render.com**, which is a cloud platform that allows for building, deploying and scaling applications with ease (*Cloud application platform*).

Others

I used **dotenv** to manage environment variables throughout my application, both for the frontend and the backend implementations.

In the process of registering a new user, I used **bcrypt** to hash password. This added an extra layer of security to my application.

To allows the frontend and the backend to communicate, I used **CORS** (**Cross-Origin Resource Sharing**).

Interesting implementations

In this section of the report, I will describe into more detail a few implementations from my code that I consider to be worth mentioning.

Authentication and Authorization

This implementation ensures that only registered and authenticated users can have access to certain features of my web application, therefore providing a secure experience. This implementation decides how users register, log in and gain access to protected resources from the application.

The authentication and authorization functionality were designed using a combination of bcrypt (for hashing passwords) and JWT (JSON Web Tokens, for session management). I also implemented a middleware to protect the restricted routes from my application.

The process of registering a new user is handled by the *register* function in the *authController.js*. The code for registering a new user can be seen in Figure 35.

```
// Register new user
Complexity is 7 It's time to do something...
const register = async (req, res) => {
  console.log(req.body);

  // Validate user input
  const { error } = registerValidation(req.body);

  if (error) {
    return res.status(400).json({ error: error.details[0].message });
  }

  // Check if email is already taken
  const emailExists = await User.findOne({ where: { email: req.body.email } });

  if (emailExists) {
    return res.status(400).json({ error: "Email is already taken!" });
  }

  // Hash password
  const salt = await bcrypt.genSalt(10);
  const hashedPassword = await bcrypt.hash(req.body.password, salt);

  // Create user object and save it in the database
  const userObject = {
    firstName: req.body.firstName,
    lastName: req.body.lastName,
    email: req.body.email,
    password: hashedPassword,
  };

  try {
    const savedUser = await User.create(userObject);
    res.json({ error: null, data: savedUser });
  } catch (error) {
    res.status(400).json({ error });
  }
};
```

Figure 35. Authentication controller – Register

The steps for the registration process are the following:

1. User input is validated using the Joi schema that can be seen in the image below.

```
// Register validation

const registerValidation = (data) => {
  const schema = Joi.object({
    firstName: Joi.string().min(2).max(100).required(),
    lastName: Joi.string().min(2).max(100).required(),
    email: Joi.string().min(6).max(100).required().email(),
    password: Joi.string().min(8).required(),
  });

  return schema.validate(data);
};
```

Figure 36. Joi validation - Register

2. The system checks if the email address is already associated with a user account.
3. Once these validations and verifications are complete, the next step is to hash password of the newly created user. This is a crucial step due to the fact that storing password in plain text in the database represents a critical security risk.
4. The new user's details, including the hashed password, are then saved in the PostgreSQL database that my application is connected to.

The process of logging in an existing user is handled by the *login* function in *authController.js*. The code for logging in a user can be seen in Figure 37.

```
// Login user
Complexity is 7 It's time to do something...
const login = async (req, res) => {
  // Validate user login credentials
  const { error } = loginValidation(req.body);
  if (error) {
    return res.status(400).json({ error: error.details[0].message });
  }

  const user = await User.findOne({ where: { email: req.body.email } });
  if (!user) {
    return res.status(400).json({ error: "Incorrect email or password!" });
  }

  const validPassword = await bcrypt.compare(req.body.password, user.password);
  if (!validPassword) {
    return res.status(400).json({ error: "Incorrect email or password!" });
  }

  // Create and assign token
  const token = jwt.sign(
    {
      id: user.userID,
      email: user.email,
    },
    process.env.TOKEN_SECRET,
    {
      expiresIn: process.env.JWT_EXPIRES_IN,
    }
  );

  // Attach token to header
  res.header("auth-token", token).json({
    error: null,
    data: { token },
  });
};
```

Figure 37. Authentication controller – Login

The steps for the login process are the following:

1. User input is validated using the Joi schema which can be seen in the image below.

```
// Login validation

const loginValidation = (data) => {
  const schema = Joi.object({
    email: Joi.string().min(6).max(100).required().email(),
    password: Joi.string().min(8).required(),
  });

  return schema.validate(data);
};

module.exports = { registerValidation, loginValidation };
```

Figure 38. Joi validation - Login

2. The email is checked in the database using *findOne* and the password provided by the user trying to log in is compared with the hashed password from the database using *bcrypt.compare*.
3. If the credentials provided by the user are correct, a JWT token is generated using the jsonwebtoken library and this token includes user specific claims (id and email) and is signed with a secret key, which I am storing in the environment variables. The token is also assigned a certain expiration time, also stored in the environment variables.
4. The token is the sent to the client as part of the response header, and the client stores it in localStorage.

Once a user is authenticated, they can access protected routes from the application. This is where the custom middleware function that I implemented, verifyToken, is used. If the token is valid, then the user is allowed to proceed and access the protected route that they are trying to access. If the token is invalid, the user is denied access to the protected route, therefore maintaining the integrity of the restricted resources in my application. The implementation of the token verification middleware can be seen in the image below.

```
// Token verification middleware

const jwt = require("jsonwebtoken");

Complexity is 5 Everything is cool!
const verifyToken = (req, res, next) => {
    const token = req.header("auth-token");

    if (!token) {
        return res.status(401).json({ error: "Access denied!" });
    }

    try {
        const verified = jwt.verify(token, process.env.TOKEN_SECRET);
        req.user = verified;

        // Pass control to the next route
        next();
    } catch (error) {
        res.status(400).json({ error: "Invalid token!" });
    }
};

module.exports = {verifyToken};
```

Figure 39. Token verification middleware

Here is an example of how the middleware is being applied in the routes corresponding to the journal functionality of my application, for which a user needs to be logged in to access:

```
const {verifyToken} = require('../middlewares/tokenVerification');

// Save journal entry
// api/journal/entries
router.post('/entries', verifyToken, journalController.saveJournalEntry);

// Fetch journal entries by userID
// api/journal/entries/:userID
router.get('/entries/:userID', verifyToken, journalController.getJournalEntries);
```

Figure 40. Example of usage of verifyToken middleware

On the frontend, the login functionality is implemented in the login component, *Login.js*. When a user submits the login form after filling in their credentials, these credentials are sent to the */api/auth/login* endpoint using the Axios library. If the login is successful, the returned token is stored in localStorage and the application state – *isAuthenticated* – is updated, which enabled the rendering of protected components, as well navigation to secure routes.

Authentication and authorization mechanisms are also integrated in the App component of my React frontend application. Different user flows are allowed based on the user's authentication status – *isAuthenticated* – and based on the user's *userId*. This component handles authentication state by determing is a user if logged in or not (using the state variables *isAuthenticated* and *userId*) and it also implements protected routes through the *PrivateRoute* component. The authorization logic is integrated by passing the *userId* to those components that need it in order to access data specific to the user. All of this can be seen in the code snippet below, in Figure 41.

```jsx
return (
  <BrowserRouter>
    <Routes>
      <Route path="/register" element={<Register />} />
      <Route
        path="/login"
        element={<Login setIsAuthenticated={setIsAuthenticated} />}
      />

      <Route path="/" element={<HomepageGuest />} />
      <Route
        path="/home"
        element={
          isLoading ? (
            <p>Loading...</p>
          ) : (
            <PrivateRoute
              element={<HomepageLoggedIn userID={userId} />}
              isAuthenticated={isAuthenticated}
            />
          )
        }
      />
      <Route
        path="/journal"
        element={
          isLoading ? (
            <p>Loading...</p>
          ) : (
            <PrivateRoute
              element={<Journal userID={userId} />}
              isAuthenticated={isAuthenticated}
            />
          )
        }
      />
```

Figure 41. App.js code snippet

Credentials to test the application

In order to see the application in the case when a user has multiple journal entries and has filled in the mood questionnaires throughout multiple days, the following credentials can be used to log in:

Email: frank123@gmail.com

Password: FranksPassword

Testing

In this chapter, I will describe in more detail the types of testing that I performed for my application.

Software testing is a very important practice that has as purpose evaluating and improving the quality of software products. Performing testing on application is essential in order to ensure that they are functioning correctly, and that they are meeting the needs and requirements and ultimately providing value to their end users.

The various types of tests that I conducted and implemented for this project will be described in what follows.

User testing

During the beginning phases of the project, it was suggested by my supervisor that I perform some simple user testing in order to validate the designs that I had created, and to identify potential issues with this prior to the actual implementation of my application.

I decided to perform user testing in the form of think aloud tests. Think aloud testing is an observation method of user testing which, as suggested by its name, involves asking users to perform certain tasks on the application and to think out loud while doing so. (*Usability body of knowledge*)

The scenarios of the think aloud tests that I conducted were regarding different pages of my application (mainly the homepage, journal page, mood tracker page and insights page).

One example of think aloud test that I performed was to show the users the homepage – the one for the case when a user is logged in – and ask them what they think is possible do to on that page and what some features of the application are. Most of the users clicked on the "Start writing" button right away and then successfully navigated to the journal entry writing page. This reaffirmed to me that one of the main features of the platform, which is represented by the journal-like functionality, is easily accessible to the users.

Another example of task that I asked the users to perform is to complete a mood tracker questionnaire, and then view the changes in mood insights that it might generate. All the users knew that in order to achieve this, they had to navigate to the Mood Tracker page (from the navigation bar), fill out one of the questionnaires available there, and then navigate to the Insights page (also from the navigation bar). This scenario brought up several suggestions form users, such as adding the option to navigate directly the desired mood tracker questionnaire from the homepage, and also to somehow link the Mood Tracker page directly with the Insights page, to suggest that there is a direct connection between the two features of my application.

After conducting this type of users testing, I took the key results and divided them in two categories – strengths and points that could use some improvement. Among the strengths that were identified by the users, were aspects such as clear and visually appealing design, interesting and valuable generated personal insights, and easy-to-use features. On the other hand, some features that could use improvement were adding an easily visible and attention catching crisis intervention on the homepage or on one of the main pages, and connecting the main features of the platform more directly.

Backend testing

When it comes to testing the backend of my application, I decided to use the Mocha JavaScript testing framework, together with the Chai.js assertion library, as I knew from previous experience that they compliment each other well and that they allow for creating clear and comprehensive test cases. Mocha provides the structure for the tests and the execution framework, while Chai.js provides assertions that verify the results of the implemented tests.

I created a file called *setup.test.js,* which can be seen in Figure 42. In this file, I initialized and configured the testing environment, to make sure that the application ran in a way that is testing-specific. I also loaded the Sequelize database instance and I imported the models used in my application, which represent the database tables needed for performing test specific operations (like creating data, updating data or cleaning up test data). I ensured that all the necessary tables were created in the test database before running the actual tests by using the *syncDatabase* function. I also implemented the *before* and the *after* hooks. Like their names suggest, the *before* hook runs once before all the tests are run, while the *after* hook runs once after all the tests are completed. The *before* hook synchronizes the database and makes sure that the specified tables are cleared, and the *after* hook cleans up the test data from the database once the tests are completed.

```
// setup.test.js

process.env.NODE_ENV = "test";

const { sequelize } = require("../db/postgresql/setup");

const User = require("../db/postgresql/models/user");
const Goal = require("../db/postgresql/models/goal");
const JournalMetadata = require("../db/postgresql/models/journalMetadata");
const MoodEntry = require("../db/postgresql/models/moodEntry");
const MoodResponse = require("../db/postgresql/models/moodResponse");
const Questionnaire = require("../db/postgresql/models/questionnaire");
const Question = require("../db/postgresql/models/question");
const QuestionnaireSubmission = require("../db/postgresql/models/questionnaireSubmission");

Complexity is 3 Everything is cool!
const syncDatabase = async () => {
  try {
    await sequelize.sync({ force: true });
    console.log("Database tables recreated successfully.");
  } catch (error) {
    console.error("Error syncing database:", error);
  }
};

// Clean the DB before and after tests
before(async () => {
  await syncDatabase();

  await sequelize.sync({ force: true });
  await User.destroy({ where: {} });
  await Goal.destroy({ where: {} });
});

after(async () => {
  await User.destroy({ where: {} });
  await Goal.destroy({ where: {} });
});
```

Figure 42. setup.test.js file

The tests that I implemented in my code are integration tests, since they verify that different parts or modules of my application are working together as I expect them to. I am testing API endpoints, such as */api/auth/register*, */api/auth/login*, and so on, as well as database interactions, such as ensuring that a user is created and inserted in the database upon user registration.

I also implemented a test that can be considered as an end-to-end workflow test, more precisely for the workflow of creating a goal and then retrieving it. This test simulates the entire workflow, from registering a user and then logging that user in, to creating a goal and then displaying the daily goals for the logged in user, which include the previously created goal. What I trying to achieve with this workflow was to ensure that the different parts of my application – models, routes, middleware and controllers – are working together seamlessly and as expected.

While I was able to implement a few backend tests for my application, I am nowhere near a percentage high enough to consider that I have most of my application tested. Since I don't have much experience with PostgreSQL applications, it took me quite some time to figure out how to perform backend testing in this context and, in the beginning, it was hard for me to find a way to use a test database instead of my development database for the testing, and I kept running into errors.

In the future, I would like to focus more on creating backend tests for my application.

Frontend testing

Unfortunately, I was not able to implement any frontend testing in my application, due to other features being more time consuming or more complex to implement then I had originally assessed, or due to running into unexpected error.

Even though I was not able to implement frontend tests, I did research possible frameworks or libraries that I would have liked to use. One of them is Cypress, which is a JavaScript-based front-end testing tool that enables developers to perform two types of frontend tests, end-to-end tests and component tests. Cypress is a useful tool when it comes to making sure that the user flow of the created application which is being tested is the correct one. This would have been useful if I wanted to perform UX/UI testing on my web application.

Another library that I researched and was hoping to use in order to perform frontend tests in Jest, a JavaScript testing framework.

Deployment and Hosting

This section of the report is dedicated to providing more information regarding the deployment process and the chosen option for hosting my web application.

Deployment is a term that refers to the mechanism through which a software is made available by developers, to users, or other programs, on a system. The options that I was considering for deploying my website were Render, Netlify and Vercel, and my final decision was to go with Render. While all three of them are reliable and widely used for deploying web applications, and they all offer a free tier, I will explain in what follows the reason why I chose to deploy my application on Render. Another option that I was also looking into and would have been suitable for my project is Heroku, but as of 2022 it does not offer a free tier anymore.

Based on the research that I conducted prior to deploying my application, I found there to be some important differences in these options, that were relevant to my project and its needs. A key factor in my decision-making process was the fact that I needed to deploy both my backend and my frontend, in order for my application to be complete and to work as desired. While deploying the frontend would have been of similar difficulty on all three platforms that I was deciding between, the same cannot be said about deploying the backend part of my web application.

I found that Render provides support for backend applications, and it actually makes the process of deploying backend more straightforward and simpler, even for developers that do not have such extended experience with this. Deploying the backend of my project was very easy due to the clear documentation and wide community support. On the other hand, I found out that both Vercel and Netlify are optimized and more commonly used for deploying frontend applications that are serverless. Neither of these are designed for RESTful APIs, like the one that I implemented for my project, and they do not provide native support for backend frameworks, such as Express, the one that I used.

Moreover, when it comes to database integration on the three platforms that I was considering, I came to the conclusion that Render was also the option that was the better fit for my project's needs and requirements. It supports directly deploying both MongoDB and PostgreSQL, the two databases that I implemented in my application. The possibility to provide built-in environment variables in order to pass database credentials was a huge advantage and help, due to the fact that I was already using environment variables in my code, and it was very easy to use the same ones on Render. When it comes to both Vercel and Netlify, my research suggested that neither of them provides database hosting and that relying on external services is needed to manage integrating the needed databases with my application.

Keeping in mind these considerations, I concluded that the clear choice for deploying my application was Render.

The next step after deciding this was to establish whether I wanted to deploy my application as a single, monolithic application, or deploy the and host the frontend and the backend separately. I first considered deploying the backend and the frontend together, as a single project on Render, due to the fact that this process would have been simpler and more straightforward, since I was developing them together and they were contained within the same project. This approach would have meant that I would only have one project to manage and deploy, which is easier to handle and debug, and does not introduce new potential issues – such as managing cross-origin issues, API calls between different services, and multiple pipelines for building and deploying the application.

However, I chose not to deploy them together, and instead I deployed each part of my application (backend and frontend) separately. The reason behind my choice was the fact that the kind of platform that I implemented for this project is one that, in the long term, would need the possibility for better scaling individually. As more users join the website, the frontend will most likely not need too scale too much compared to how it is at this stage, but the backend might need to scale considerably. If each user of the application would start writing daily journal entries, or some even multiple journal entries per day, this could potentially raise some serious issues for the application.

I also enjoyed the flexibility that this approach provided, with me being able to change or replace different parts of the application, for example in the backend, without needing to drastically change the frontend.

Before deploying the backend and the frontend, I had to create a PostgreSQL database on Render and populate it with the data for questionnaires and questions that I had in the database from my localhost. This steep of seeding the data was needed because the questionnaires in my application are static and, for the time being, there is no admin panel from which an admin user might be able to edit the questionnaires and their respective questions. Another thing that I needed to do after creating a PostgreSQL database on Render was to update the environment variables from my code with the database URL generated by Render.

Then, I deployed first my backend, and then finally my frontend.

CI/CD Pipelines

CI/CD (Continuous Integration/ Continuous Deployment) is a practice used in programming automates and streamlines the process of building, testing and deploying code. CI, or Continuous Integration, refers to the process of automatically integrating code changes made by multiple developers on a shared repository, while making sure that the new code works as it is supposed to, and that it does not introduce bugs or errors. CD, or Continuous Deployment, extends CI by automatically deploying the new code to the production environment after ensuring that it passes all the appropriate tests. (*What is Ci/CD?*)

To achieve CI/CD, I used GitHub Actions and created the necessary workflows for both my frontend and my backend. I first created a workflow for my backend, which can be seen partially in Figure 43.

```yaml
jobs:
  # test_backend:
  #   name: Test Backend
  #   runs-on: ubuntu-latest
  #   strategy:
  #     matrix:
  #       node-version: [20.x]
  #   steps:
  #     - name: Checkout repository
  #       uses: actions/checkout@v2
  #       with:
  #         fetch-depth: 0
  #     - name: Setup Node.js ${{ matrix.node-version }}
  #       uses: actions/setup-node@v2
  #       with:
  #         node-version: ${{ matrix.node-version }}
  #     - name: Install dependencies
  #       run: |
  #         npm install
  #     - name: Run tests
  #       run: |
  #         npm test
  #       env:
  #         POSTGRESQL_DB_USER: ${{ secrets.POSTGRESQL_DB_USER }}
  #         POSTGRESQL_DB_HOST: ${{ secrets.POSTGRESQL_DB_HOST }}
  #         POSTGRESQL_DB_NAME: ${{ secrets.POSTGRESQL_DB_NAME }}
  #         POSTGRESQL_DB_PASSWORD: ${{ secrets.POSTGRESQL_DB_PASSWORD }}
  #         POSTGRESQL_DB_PORT: ${{ secrets.POSTGRESQL_DB_PORT }}
  #         MONGODB_URI: ${{ secrets.MONGODB_URI }}
  deploy_backend:
    name: Deploy Backend
    runs-on: ubuntu-latest
    # needs: [test_backend]
    steps:
      - name: Checkout repository
        uses: actions/checkout@v2
        with:
          fetch-depth: 0
      - name: Deploy backend to Render
        if: github.ref == 'refs/heads/main'
        env:
          deploy_url: ${{ secrets.RENDER_BACKEND_DEPLOY_HOOK_URL }}
        run: |
          curl "$deploy_url"
```

Figure 43. Backend workflow

As can be seen in the screenshot, the section regarding the "Test Backend" job from this workflow is currently commented out. The reason for this is the fact that I was unable to connect to my PostgreSQL test database in order to run my backend tests. I spend a significant amount of time trying to fix this issue and followed different approaches for this, but I simply could not do it. After doing some research on this., I suspect that the reason for not being able to connect to the database stems from the fact that pgAdmin's firewall settings, which are probably preventing connections that are being initiated from GitHub Actions.

Due to this inconvenience, my workflow for the backend is incomplete, but the CI/CD pipeline does work, and it is successfully deploying my backend to Render every time that there is a new commit on the main branch.

I also created a workflow for my frontend, which can be seen in Figure 44. The deployment of the frontend happens upon the successful deployment of the backend. This workflow does not include a step for frontend testing, as I was not able implement any frontend testing during the development of my project. (*The importance of software testing* 2024)

```yaml
name: Deploy Frontend
on:
  workflow_run:
    workflows: ["Deploy Backend"]
    types:
      - completed

jobs:
  deploy_frontend:
    name: Deploy Frontend
    runs-on: ubuntu-latest
    if: ${{ github.event.workflow_run.conclusion == 'success' }}
    strategy:
      matrix:
        node-version: [20.x]
    steps:
      - name: Checkout repository
        uses: actions/checkout@v2
        with:
          fetch-depth: 0
      - name: Setup Node.js ${{ matrix.node-version }}
        uses: actions/setup-node@v2
        with:
          node-version: ${{ matrix.node-version }}
      - name: Install dependencies
        run: |
          npm install --legacy-peer-deps
        working-directory: frontend
      - name: Build frontend
        run: |
          CI=false npm run build
        working-directory: frontend
      - name: Deploy frontend to Render
        if: github.ref == 'refs/heads/main'
        env:
          deploy_url: ${{ secrets.RENDER_FRONTEND_DEPLOY_HOOK_URL }}
        run: |
          curl "$deploy_url"
```

Figure 44. Frontend workflow

The link to my Render frontend application is https://mindscribe-frontend.onrender.com.

Future improvements

In this section of the report, I will discuss in more detail about a few improvements or additional features that I would like to implement on my project in the future.

Notifications & Reminders

First of all, I would like to implement a notification system for my web application. This could be even a simple system that sends notifications related to new insights that have been generated for the user based on their journal entries and mood tracker questionnaires from the previous week.

I tried to implement simple notifications that tell the users that their insights have been updated with react-toastify, but my attempt was unsuccessful. This was due to the fact that react-toastify is used for sending alerts rather than notifications and I only found a way to send those alerts by triggering a button click, which was not my desired functionality.

I would also like to send users reminders, such as reminders at different times of the day to fill in the corresponding mood tracker questionnaire for that time of the day, reminders to write journal entries daily and to set goals for the day, reminders when they have an unfinished journal entry, and so on.

More complex Insights page

Another improvement that I would like to add to my platform is providing users with a more detailed and better adjustable Insights page.

An example of what I would like to add to this page is to generate insights for different periods of time, such as 3 days, one week, one month, 3 months, 6 months, one year, etc. Right now, I am generating the user insights based only on journal entries and mood questionnaire answers from the last 7 days. I think it could be valuable to allow users to see their mood changes and the development of their emotional wellbeing during more diverse and longer periods of time.

I would also like to conduct sentiment analysis on the text responses of the mood tracking questionnaires' questions. This would allow for a more complex calculation of mood score, stress level and energy level, and therefore it would provide more accurate mood trends for the users.

In addition to this, I would like to provide other types of insights as well, aside from the ones that I am currently providing, such as pattern recognition from journal entries and mood tracker questionnaires. It could also be a good idea to let users know the rate of completion of daily goals and to find out what is stopping them from successfully completing all their daily goals.

Guided prompt journaling

One more feature that I would like to add to the journaling section of my platform is guided prompt journaling. This would be a useful tool to motivate people to write daily journal entries, and it could provide an inspiration of what users can write about.

This feature would also allow users to better reflect on their feelings and get a better understanding of their thoughts and emotions.

Export journal entries

Additionally, I think it could be useful to provide users with the possibility of exporting their journal entries, or even their personalized insights. They could then share these notes with, for example, their therapists or beloved ones, and better process and assimilate this information.

This feature would also ensure that my application is even more in compliance with GDPR rules and regulations.

Personalization

Another feature that could provide a more user friendly and personal experience to the users of my web application would be allowing them to customize the appearance of the user interface. Some examples of what they could customize include the option for a dark mode, the display of elements like scales (whether they want emoji-based, number-based, text-based scales, etc.), and so on.

Homepage for guests

One other change that I would like to implement in the future to the platform is finalizing the implementation of the homepage that is dedicated to guest users (users that are not logged in or do not have an account on the website).

This would be the entry point to the platform, the page that first shows when users navigate to the website. The purpose of this page would be to attract users by introducing them to the platform and its main features, and to guide them to create an account on the platform.

The reason why I did not implement this page yet is because I did not have enough time, and I wanted to focus on other parts of the application that were of a higher importance.

Today's quote

Another feature that I would like to add to the homepage of my application is a "today's quote" section. This section would display a different motivational quote or quote related to mental wellbeing every day, in order to keep users engaged and remind them that emotional wellbeing is a journey.

Machine Learning

One last feature that could be interesting to implement for this application in the future would be to implement machine learning algorithms for analyzing and tracking user sentiments over time. The main idea behind this would be to gather data from different user interaction on the platform, such as the journal entries that users write, the daily goals that they set for themselves and their responses to the mood tracker questionnaires, and then use machine learning models to analyze different trends in the users' emotional states.

One example of this could be recognizing patterns in written content (journal entries) and detecting emotional shifts over different periods of time. Such a feature could offer more dynamic and adaptive personalized insights.

Conclusion

The management and development of this web application has been an equally rewarding and challenging process, which allowed me to apply the skills and knowledge that I have gained throughout my education, but also to gain new skills. The project combined elements of UX design, database design and implementation, and backend and frontend development, to create a unique platform.

The main objective of this project was to create a platform that provides a safe space for self-expression through journaling, while also offering its users personalized insights into their emotional wellbeing through sentiment analysis and mood tracking. With the combination of journal entries and mood tracking questionnaires that can be found in the web application, users can now reflect on their emotions and track changes in their mood, energy level and stress level over time, and receive actionable insights to support their wellbeing.

As with any project, there is, of course, room for improvement, and future iterations of the application could include more advanced features of the application that would enhance the impact of the application.

However, it is my belief that this project has provided me with the opportunity to develop a meaningful web application and has, at the same time, allowed me to grow as both a developer and a project manager. The skills and knowledge that I gained throughout this project will most definitely contribute to my professional development and will allow me to pursue more opportunities in the field of web development.

Reference list

- *Moscow prioritization* (2024) *ProductPlan*. Available at: https://www.productplan.com/glossary/moscow-prioritization/ (Accessed: 23 December 2024).
- Dudley, K. (2023) *The role of sketching in the design process*, *Toast Design Services*. Available at: https://toastdesignservices.co.uk/the-role-of-sketching-in-the-design-process/#:~:text=Sketches%20are%20a%20universal%20language,to%20share%20and%20discuss%20ideas . (Accessed: 23 December 2024).
- *What is wireframing? - updated 2024* (2024) *The Interaction Design Foundation*. Available at: https://www.interaction-design.org/literature/topics/wireframe#:~:text=They're%20a%20key%20deliverable,colors%2C%20fonts%2C%20or%20imagery . (Accessed: 23 December 2024).
- *What is a prototype?* (no date) *Coursera*. Available at: https://www.coursera.org/articles/prototype?utm_medium=sem&utm_source=gg&utm_campaign=B2C_EMEA__coursera_FTCOF_career-academy_pmax-multiple-audiences-country-multi-set2&campaignid=20882109092&adgroupid=&device=c&keyword=&matchtype=&network=x&devicemodel=&adposition=&creativeid=&hide_mobile_promo&gad_source=1&gclid=Cj0KCQiAsaS7BhDPARIsAAX5cSA5t6iz83yPUUnuxc7BDDs2ce7JAVTmmDwBbWXfiqzG267UV_Ra56IaApwyEALw_wcB (Accessed: 23 December 2024).
- Comment *et al.* (2024) *Introduction of ER model*, *GeeksforGeeks*. Available at: https://www.geeksforgeeks.org/introduction-of-er-model/ (Accessed: 26 December 2024).
- GeeksforGeeks (2024) *Relational model in DBMS*, *GeeksforGeeks*. Available at: https://www.geeksforgeeks.org/relational-model-in-dbms/ (Accessed: 26 December 2024).
- *DBMS normalization: 1NF, 2NF, 3NF and BCNF with examples - javatpoint* (no date) *www.javatpoint.com*. Available at: https://www.javatpoint.com/dbms-normalization (Accessed: 26 December 2024).
- Singh, A. (2023) *'Node.js database magic: Exploring the powers of sequelize Orm'*, *Medium*. Available at: https://medium.com/@rishu__2701/node-js-database-magic-exploring-the-powers-of-sequelize-orm-a22a521d9d9d (Accessed: 26 December 2024).
- freeCodeCamp (2018) *Introduction to mongoose for mongodb*, *freeCodeCamp.org*. Available at: https://www.freecodecamp.org/news/introduction-to-mongoose-for-mongodb-d2a7aa593c57/ (Accessed: 26 December 2024).
- Codecademy (no date) *What is express.js?*, *Codecademy*. Available at: https://www.codecademy.com/article/what-is-express-js (Accessed: 26 December 2024).
- GitLab (2023) *What is version control?*, *GitLab*. Available at: https://about.gitlab.com/topics/version-control/ (Accessed: 26 December 2024).
- *W3schools.com* (no date) *Introduction to Git and {{title}}*. Available at: https://www.w3schools.com/git/git_intro.asp?remote=github (Accessed: 26 December 2024).

- *About github and git* (no date) *GitHub Docs*. Available at: https://docs.github.com/en/get-started/start-your-journey/about-github-and-git (Accessed: 26 December 2024).
- *React – a JavaScript library for building user interfaces* (no date) – *A JavaScript library for building user interfaces*. Available at: https://legacy.reactjs.org/ (Accessed: 26 December 2024).
- *Node.js - about node.js®* (no date) *Node.js* -. Available at: https://nodejs.org/en/about (Accessed: 26 December 2024).
- *The fun, simple, flexible JavaScript test framework* (no date) *Mocha*. Available at: https://mochajs.org/ (Accessed: 31 December 2024).
- *Chai assertion library* (no date) *Chai*. Available at: https://www.chaijs.com/#:~:text=Chai%20is%20a%20BDD%20%2F%20TDD,with%20any%20javascript%20testing%20framework . (Accessed: 31 December 2024).
- *Cloud application platform* (no date) *Render*. Available at: https://render.com/ (Accessed: 31 December 2024).
- *What is Ci/CD?* (no date) *Revenera*. Available at: https://www.revenera.com/software-composition-analysis/glossary/what-is-ci-cd?utm_source=google&utm_medium=cpc&utm_campaign=EN-SCA-FCI-EMEA-Generic-SMART22&utm_content=Dynamic_Ads&id=PPC_SCA_2023&lead_source=PPC&k_clickid=_k_EAIaIQobChMIltLtrL7SigMV54KDBx3yTCHjEAAYASAAEgLGXfD_BwE&gad_source=1&gclid=EAIaIQobChMIltLtrL7SigMV54KDBx3yTCHjEAAYASAAEgLGXfD_BwE (Accessed: 31 December 2024).
- *The importance of software testing* (2024) *IEEE Computer Society*. Available at: https://www.computer.org/resources/importance-of-software-testing (Accessed: 31 December 2024).
- *Usability body of knowledge* (no date) *Welcome to the Usability Body of Knowledge*. Available at: https://www.usabilitybok.org/think-aloud-testing#:~:text=A%20direct%20observation%20method%20of,and%20feeling%20at%20each%20moment . (Accessed: 31 December 2024).
- *What is GDPR, the EU's new Data Protection Law?* (2024) *GDPR.eu*. Available at: https://gdpr.eu/what-is-gdpr/ (Accessed: 31 December 2024).
- (No date) *What is an API? - application programming interface explained - AWS*. Available at: https://aws.amazon.com/what-is/api/ (Accessed: 02 January 2025).
- *What is agile? and when to use it* (no date) *Coursera*. Available at: https://www.coursera.org/in/articles/what-is-agile-a-beginners-guide?utm_medium=sem&utm_source=gg&utm_campaign=B2C_EMEA__coursera_FTCOF_career-academy_pmax-multiple-audiences-country-multi&campaignid=20858198824&adgroupid=&device=c&keyword=&matchtype=&network=x&devicemodel=&adposition=&creativeid=&hide_mobile_promo&gad_source=1&gclid=Cj0KCQiA7NO7BhDsARIsADg_hIZbMvBkt9EynhDJ5_IvCS_o9cVrhTl8yh_ssgBw6FfWNOWxm5DwgjcaAnXTEALw_wcB (Accessed: 02 January 2025).