Computing NEA - Journal App

Xichao Wang

March 25, 2024

Contents

Chapter 1

Abstract

This document presents the A Level NEA Project, providing an overview of the problem area, the proposed solution's features, requirements specification, critical path, design details, technical implementation, testing strategy, evaluation, and an appendix containing code listings. [**?**]

This document presents my A level NEA project, which includes the following section with the aim to create a full picture

Chapter 2

Analysis

2.1 Problem Area

The issue identified is the lack of mindfulness and increased forgetfulness. Keeping a journal acts as a way to document significant moments within an individual's life, while providing a personal and private way for people to express their emotions. I find myself a perfect example of somebody that is lost in 'modern life,' leading me to forget even simple events which had occurred within my day.

In a hectic world full of day-to-day distractions, the creation of a consumerist culture through the rise of social media has increased the likelihood for individuals to feel less content with their day-to-day lives. We are consuming content through different forms of media, whether that be through our computers, phones or televisions. As a result, individuals are left constantly craving to ingrain more information, but none of this information is retained. The society of the modern-world has evolved the extent that it has made it inevitable for individuals to be exposed to the high-volume of content. Such content is easily accessible through our phones, inevitably leading individuals to become overwhelmed and overloaded. If we consider the Covid-19 quarantine and its long lasting impact on teenagers mental health, [**?**] it has

left a void for individuals needing to relieve their anxieties and reduce the overall stress incorporated into their daily lives. journaling can induce a positive impact on individuals struggling with the overwhelming nature of social media and how it has clearly merged itself with day-to-day life, bringing mindfulness as a positive method in documenting and organising a person's life.

Keeping a Journal increases productivity and mindfulness. I have always wanted to incorporate this habit into my daily routine, and through my NEA, I hope to create an easily accessible platform for myself and those alike. For my NEA, I want to create a minimal, easily navigated web-interface where individuals can add entries, storing the data, and through my web-interface, these entries can be safely stored for personal use for whoever is accessing the website.

2.1.1 Benefits of journaling

Here are some benefits of journaling include:

- It is a method of mindful writing that can help you to become more aware of your thoughts and feelings. Therefore come to terms with them.
- Simply jot down your thoughts and ideas, and you can come back to them later.
- It can help you to become more organised and productive by making you more aware of your goals and aspirations.

2.2 Client / End User

journaling is an excellent habit anyone could incorporate into their daily lives. However, my primary target end-user for my app would be people like

myself who are more digitally orientated, and are comfortable with accessing an online platform to manage their daily lives. Teenagers and young adults are more likely to be my target audience, as they are more likely to be comfortable with using online platforms.

I've wanted to journal for quite some time now, but due to my busy schedule and overall forgetfulness, I have never been consistent with the habit for over a week. To finally properly start jouranling once and for all, I will build a website front end that can be accessed as long I have an internet-connected device. This way, I can journal no matter where I am, reducing the friction which prevents me from cultivating the habit of journaling.

2.2.1 Survey Result

I have created a survey for friends and family to fill out to understand the needs and people's thoughts about journaling. Through this I learned about more benefits of the habit and I also gained insight into people's online usage which will help me to design the app. Here are some of the results from the survey:

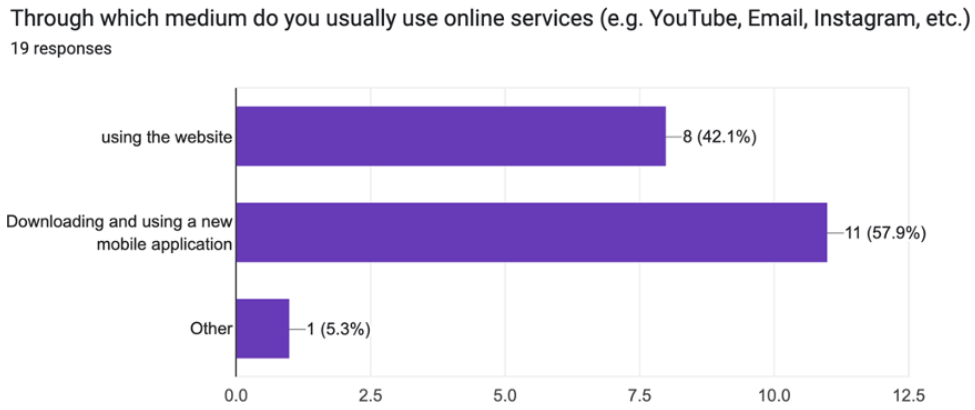

Figure 2.1: Survey Results Showing the Platform People Use to Access Online Services

In the figure above it shows that although most people use mobile application to access online services, but a significant number of people use the website. I will be creating a website for the journal app because its universally accessible, but in the future I can create a mobile app to cater to the needs of the users.

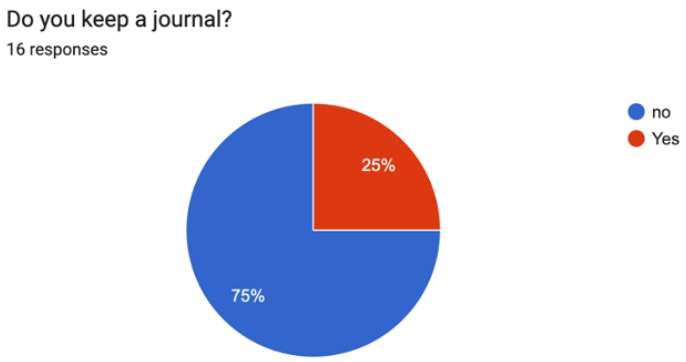

Figure 2.2: Survey Results Showing the Ratio of People who Keep a Journal

As expected, majority of people do not keep a journal. Creating my app could potentially help people to start journaling and experience the benefits of the habit.

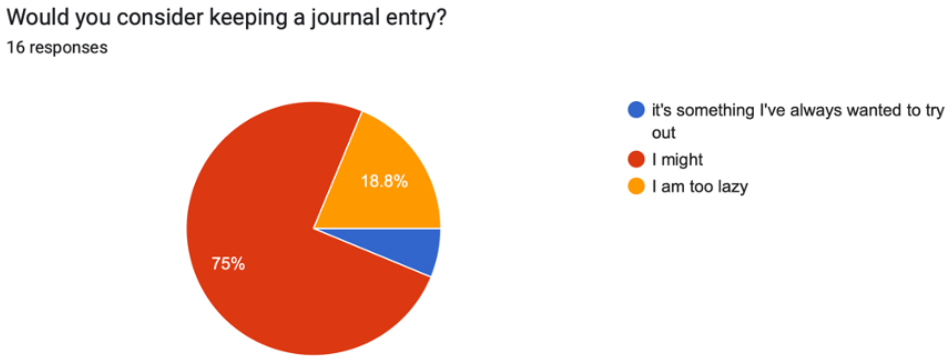

Figure 2.3: Survey Results Showing the Ratio of People who Would Consider Journaling

Interestingly, a significant number of people would consider journaling. This shows that there is a demand which I could potentially fill with my app.

Q1. *I asked the participants who who journal what inspired them to start journaling.*

A1. "I overthink too much"

"Was more of a note taking thing I used to do then it evolved into writing - not all the time, but on occasion"

"Having a busy period in my life where so much was going on that I realised that I wanted to remember as much of it as possible. However, I wasn't certain that I'd be able to recall everything on my own, so I wrote things down as they happened to ensure that I wouldn't forget them."

"I was walking across the river Thames one day and I sat down for 30 minutes and just stared as the river flowed extremely peacefully. I found so many thoughts racing through my head and got restless without any digital media to entertain me. This experience inspired me to have that peace of mind and articulate my thoughts."

- This confirms my belief that journaling is a good habit with many benefits.

Q2. *I then asked everyone for the benefits they see in journaling.*

A2. Results

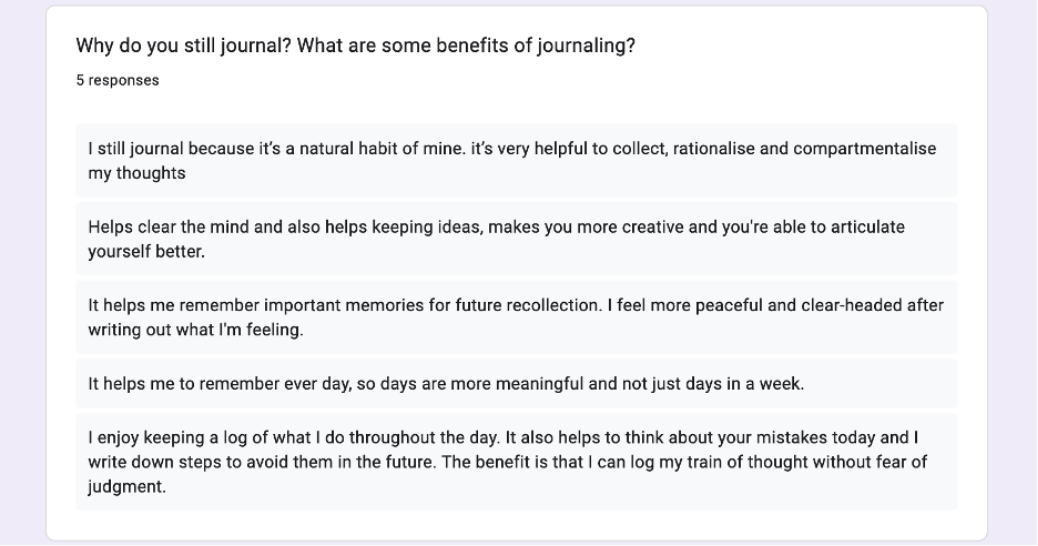

Figure 2.4: Survey Results Showing the Benefits of Journaling Highlighted by People Who Journal

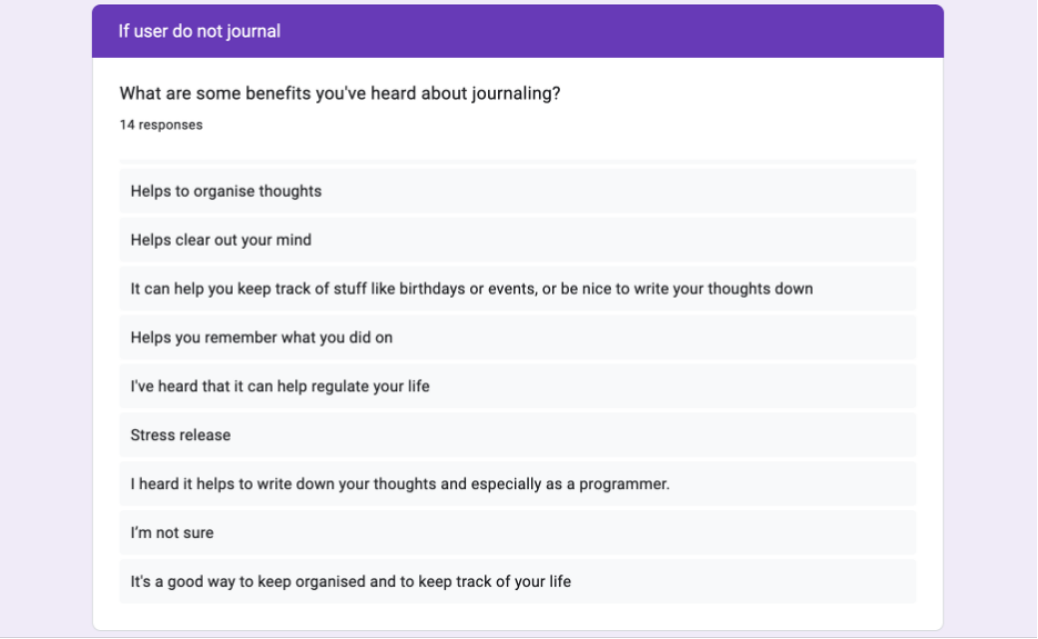

Figure 2.5: Survey Results Showing the Benefits of Journaling Highlighted by People Who Do Not Journal

Regardless of whether people journal or not, they all see the benefits of journaling. Almost every single person has an unanimous agreement that journaling is a good habit to have. This is a good sign for my app as I was correct for thinking that people would be interested in journaling.

Q3. *Finally, I had asked the participants for "What would be some important and fundamental features of a digital journal app?"* Here a few selected response which represent all the responses as a whole:

A3. "Security, I don't want my data to get breached and leaked."

- This is a very important feature to have in the app. I will need to ensure that the data is stored securely and sensitive information are encrypted.

A3. "Writing entries, being able to select and view a past date, low-key it would be really interesting to see my journaling pattern in the long term, if you can do that itd be kinda cold"

- The thing to highlight here is that the user wants to be able to refer back to past entries and being able to see the statistics of their journaling. This is a good idea and I will need to implement this feature in the app.

Interview

I had an interview with a teacher at my school who represents the someone who's slightly older than me and represent the wider user base of the app. I have picked out some responses which is particularly relevant for the application.

Q1. *How do you feel about the security of your personal information in digital platforms? Are there expectations you have?*

A1. "...Yeah, surely my password and logins should be protected and private"

- Users of the platform want to have a way of securely storing their sensitive information. Therefore, I will need to implement a secure way of not only storing user but also encrypting the data that is stored in there. For demonstrating purposes I will be encrypting the most sensitive data such as the user's password.

Q2. *Can you describe what you look for in a good app interface? Are there any design elements or navigation styles you find particularly helpful or annoying?*

A2. "...I would like to have an easy-to-use menu with options instead of a gimmicky and too-visual website."

- The user wants a simple and easy to use interface. Therefore, I will need to design a clean and feature rich interface. I think a navigation bar would be a good idea to implement as it clearly shows all the options available to the user.

Q3. *How important is an app's speed and responsiveness to your overall user experience? Have you stopped using an app because it was too slow or unresponsive?*

A3. "Yes, very much it's a waste of my time, I would rather use something else."

- Performance is a key factor in the user experience. Therefore, I will need to ensure that the app is responsive and fast. I will need to use a framework that is known for its speed and performance.

Q4. *How has your experience been with digital note-taking or journaling tools compared to traditional methods?*

A4. "No one carries a pen and paper anymore, dont be silly... Yeah [digital journaling] would be very useful for people on the go."

- Website is accessible through all devices with an internet connection. Creating a website lays the fundation of the app, providing a good way for the user to do the journaling. In the future I can add different forms of application.

Q5. *Any additonal comments?*

A5. "I find it frustrating with other apps to enter the date every time I want to log something. Can't it just record it automatically Automatically - you can have your morning meeting then later you can have your evening meeting it automatically enters the date . It would be quite handy."

A5. "I like it the data sync between different devices."

- User wants to be able to have consistent data for ease of access. Building an API and backend means I can store the data in a way that is easily accessible and consistent. These data can then be accessed through all forms of frontend interfaces. Th proves that in the future I can create other interfaces such as a mobile app and be able to have consistent data via the backend.

2.3 Research Methodology

Initially what inspired me to create a mindful journal app was when I watched a Youtube Video which introduced the concept of journaling, it highligthed the vast array of benefits that journaling can bring to an individual. Then, I looked through various online articles to understand the concept abit more.

A reason for the creation of the app was because I wanted to create a habit of journaling, and I thought that creating an app would be a good way to do so. I have also conducted a survey and an interview to understand the needs of the user and to understand the requirements of the app.

Some of the research methods I have used include:

- Surveys
- Interviews
- Watching YouTube videos
- Online Research

- Personal Experience

2.4 Features of Proposed Solution

Below I have synthesised the core features of the proposed solution. I have broken down the features into two categories - frontend and backend.

2.5 Requirements Specification

ID	Requirement Description	How to Evidence
1.1	Develop a login page to authenticate users, include forms for user input and dynamically provide error/feedback when necessary.	Demonstrate through UI screenshots or video, and code snippets.
1.2	Implement a registration page enabling new users to create an account, take in input for email, password, last name, first name and interact/update backend. Provide feedback to the user after submission.	Demonstrate through UI screenshots or video, and code snippets.
1.3	Develop a place where the user can see details about themselves ie their personal information and journal statistics	Provide screenshot and code snippet
2.1	Create a page for users to compose new journal entries. Input parameters are title and content of the entry. Interact with the corresponding endpoint and provide feedback to the user.	Video of the UI as well as checking database to verify successful submissions as well as code snippets.
2.2	Design a retrieve journal page that lists all the user's entries, with options for sorting the entries in ascending or descending order based on creation date or other criteria in an efficient way.	Video documenting interactions with the UI, showing the features listed.
3.1	Implement navigation bar that adjusts its visibility of options based on the user's login status.	Video of the UI and code snippets.
4.1	Utilize state management techniques to keep the user logged in across different pages, preserving session information securely.	Screenshot and explanation of the implementation as well as video of functioning app.
4.2	Handle fetching, posting, and updating data through JSON responses from the backend.	Screenshot of code snippets showcasing a couple example of this in action.

Table 2.1: Frontend Requirements for Journal App

ID	Requirement Description	How to Evidence
1.1	Design data models for users, journal entries, with relationships defined.	Screenshot of my database admin panel with created models and their relationship as well as code snippet of the definition of the models.
1.2	Have an effective way to check the submitted user information, journal entries, and related data before stored in the database, ensuring data stored are in the required format.	Code snippet and testing of the functionalities.
1.3	Have a way to generate statistics of the user's journaling habits, such as the most active month of the user, average entries made per week, etc.	Code snippet of the function that generates the statistics and screenshots of the statistics being displayed.
2.1	Develop API endpoints with functionalities implemented to handle user authentication (login and registration) and other core endpoints that drives the journal app as a whole	use tools to run requests against the endpoints and capture their responses, testing different cases.
2.2	Features for secure password storage(salting and hashing) and and authenticate; token generation for session management.	Run tests to see the functionalities are working
3.1	Ensure all API endpoints are secured and accessible only to authenticated users, using tokens or similar mechanisms for session management.	screenshots of authenticated requests and their responses.
3.2	Ensure smooth and spontaneous communication with the frontend.	Provide video of the working application
4.1	Design the Backend with the room for more functionalities	Code snippet of none core functionalities being implemented.

Table 2.2: Backend Requirements for Journal App

2.6 Critical Path

The critical path refers to the sequence of the stages in the development of my application. I am doing full stack development, which includes independent backend and frontend both with many small features to add all the time. The Agile methodology would be suitable for me to following for my project. This is because it is a flexible and iterative approach which allows repeated test of all the small modules and eventually building up to a complete solution. It is also a good way to manage the project as I can easily adapt to changes, and I will need to make many changes since there will be many new things I will learn and obstacles overcome to as I am developing the app.

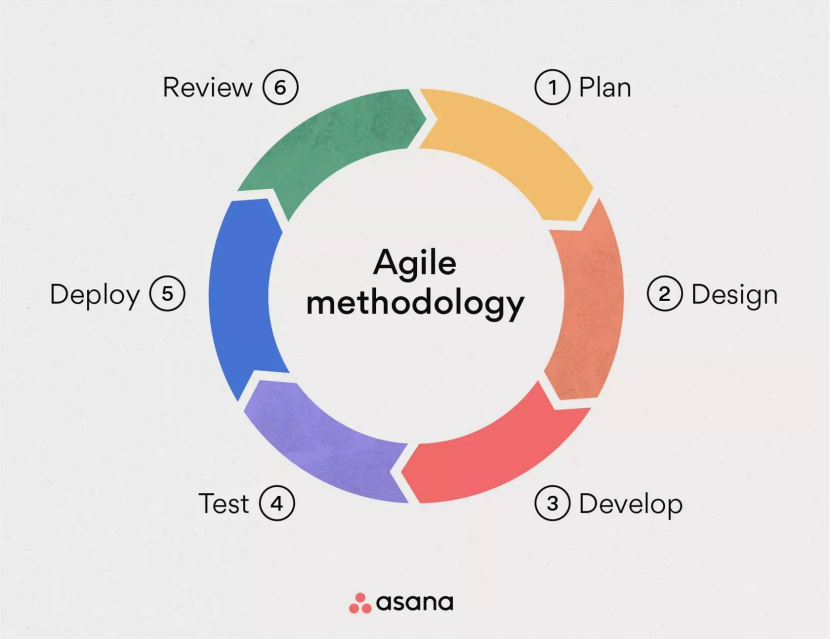

Figure 2.6: Imagine Showing the Steps of the Agile Methodology taken from [**?**]

Chapter 3

Design

In this section of my NEA I will be including details on the detailed design of my Journal app.

3.1 Introduction

The objective of my journal app is to provide users with an easily accessible and navigated app for digital journaling. To accomplish this, I will choose the Django REST framework for my backend server functionality and React, a powerful JavaScript library, for my frontend. In this section of my report I will be justifying why I have made these choices and detailing the specific designs for full stack application tailored to these frameworks

3.2 Frontend

Frontend primarily refers to the interface which the user interacts. The most common form of frontend is a webpage, which encompasses the layout and design elements of the website. Initially, webpages are created using HTML,

CSS, and JavaScript. However, as time progresses, developers have devised ways to generate them. Using Python, for example, frameworks like Django and Flask allow developers to build HTML, CSS, and Javascript pages interactively and dynamically. In the case of Django and Flask, developers can use templating engines like Jinja2 to generate dynamic HTML pages by integrating Python code into the HTML layout, as well as being able to avoid boilerplate code by building templates for the layout of the appearance of the page and then being able to make changes to different pages while maintaining consistency across the website. This approach saves time since it lets me focus on the features instead of writing boilerplate code.

Behind the scenes, frontend is not only the UI, as it also includes the logic and functionality that drives the user experience. This includes handling user interactions, making API calls, data manipulation, and updating the UI based on user input.

Javascript frameworks are the most popular amongst web developers since there exist powerful libraries like React and Angular maintained by giant tech companies, and javascript is simply more widely adopted and used in the web development world. Similarly, I have heard good things about a library called Svelt, which offers a lightweight approach to building user interfaces. What I mean by that is it uses fewer codes, is truly reactive, and has no virtual DOM

Initially, I used Django to generate my frontend pages using its built-in templating engine dynamically. However, I want to create a more modular approach and, therefore, truly isolate my backend and frontend. The benefit of doing so is that I could connect a completely different front-end interface to the same backend functionality with ease via API communication. In my application, I will try my best to maintain the Single Responsibility design principle throughout my application development process.
To achieve achieve this modular approach, I decided to use a separate frontend to my Django backend, specifically React.js. React is the most popular and in-demand ecosystem developed by Facebook for building responsive user interfaces, Facebook developed React.js and later React native for building "truly native" mobile apps using JavaScript. React.js is the javascript library I chose for my web creation, and I believe that it would be a valuable learning experience to take on my first Javascript library to build my app.

3.3 Backend

Backend refers to the server side of a web application, where the logic behind the application is. It is responsible for processing requests, interacting with databases, and algorithmically generating responses to be sent back to the client and other background processes. There are frameworks available for different programming languages that can be used for backend development.

I have considered Python, C# and Javascript frameworks for my project as I am proficient in Python, C# (having dabbled with those in my own time and studied those languages for GCSE and A-Levels, respectively), and Javascript is a language I have always wanted to explore further since it is the most popular language; hence, there's wide support for it in terms of libraries and frameworks. Specifically, I have considered Django, FastAPI, Flask, Next.js, and ASP.NET Core frameworks.

3.3.1 ASP.NET:

ASP.NET is a powerful framework developed by Microsoft for building web apps and services using the .NET framework and C#. It is capable and scalable, making it suitable for large-scale applications. Personally, however. I did not end up choosing ASP.NET due to my hardware constraint. I am running a MacOS unix system on the apple silicone architecture, which I could technically do .Net development using. However, from my experience in the past, it is a pain. In addition, ASP.NET was primarily suited for the Windows ecosystem (although they are trying to make it more platform-independent), meaning it is not optimal for me personally, and there are other better ways to create a university platform-independent backend.

3.3.2 Next.js and Express.js:

Two of the most demanding and powerful frameworks are these two are used to build the backend using Node.js. Next.js goes hand-in-hand with React

since it provides server-side rendering. This is a thin client approach, where the initial HTML is generated on the server and then sent to the client. This approach ensures the smoothness of the user experience [**?**]. Express.js is, on the other hand, a lightweight and flexible framework for building RESTful APIs and the like. It might be nice, but I am more comfortable with Python, so I decided that it's a good idea to only use Javascript for my Frontend development and Python for my Backend.

3.3.3 Flask and FastAPI:

Flask is a lightweight web framework for Python used to build APIs. I had previously dabbled with the framework and found it barebone and flexible. FastAPI is a much more modern alternative to Flask. It provides similar simplicity and flexibility but with the added benefit of high performance due to its asynchronous capabilities. I was really tempted to use FastAPI for my project, however, I had been learning a lot of Django and decided to go with it instead.

3.3.4 Django

Django is the most feature-rich mainstream backend web development framework in Python. It provides all the necessary tools I need, including URL routing, Database Object-Relational mapping, as well as overall security features preventing basic forms of attacks such as CSRF. This allows me to robustly build my application, focusing on functionalities that matter rather than reinventing the wheels. Django also allows me to have fine-tuned control over functionalities if I want to redesign, replace, or extend any parts. Even though Django might not be as performant as some other frameworks, such as FastAPI, it offers a well-established and stable platform for web development suitable for my Journal app.

3.4 Hierarchy Charts

Hierarchy charts are visual representations of a large problem being broken down into smaller components or sub-problems. They are charts in the form of an upside-down tree structure. Ideally, the problem should be broken down until each module is no longer than a page of code. In the charts I have created, I had to separate my application for two different charts. One for my Frontend and one for my Backend because the project is quite large. In the diagram, I have broken down my problems to a reasonably small size yet not too specific such that it becomes too difficult to maintain.

3.4.1 Django Server Design:

In this hierarchy chart, I have decomposed the design of my entire Django backend API into many smaller problems I will solve in my implementation. I will not be describing each individual problem in detail here. Still, the hierarchy chart provides a clear overview of how the different components and functionalities of the backend system are organised. See Figure 3.1 for the chart.

3.4.2 React Frontend

In the hierarchy chart for the JavaScript frontend, I have broken down the design of my user interface and client-side functionalities into smaller components.

These components include user authentication, data retrieval from the backend API, the UI and more. In this chart, it's hard to show how the functions communicate at an inter-component level. However, on top of what's designed in the hierarchy chart, data is passed through props and the things under my state management box shown in the diagram would be accessible throughout my application by different components through the use of React Context and Context Provider. See Figure 3.2 for the chart.

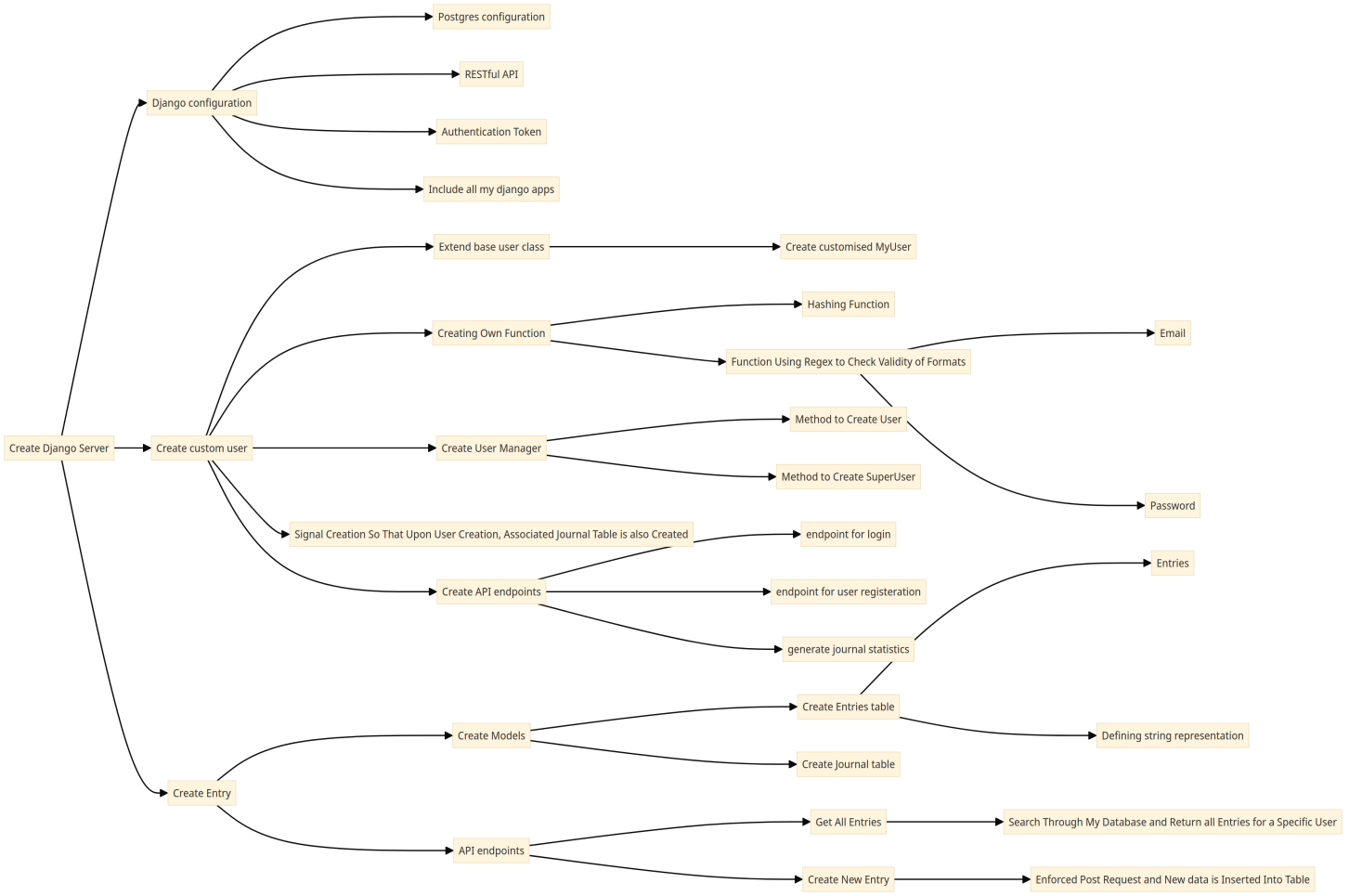

Figure 3.1: Backend Hierarchy Diagram

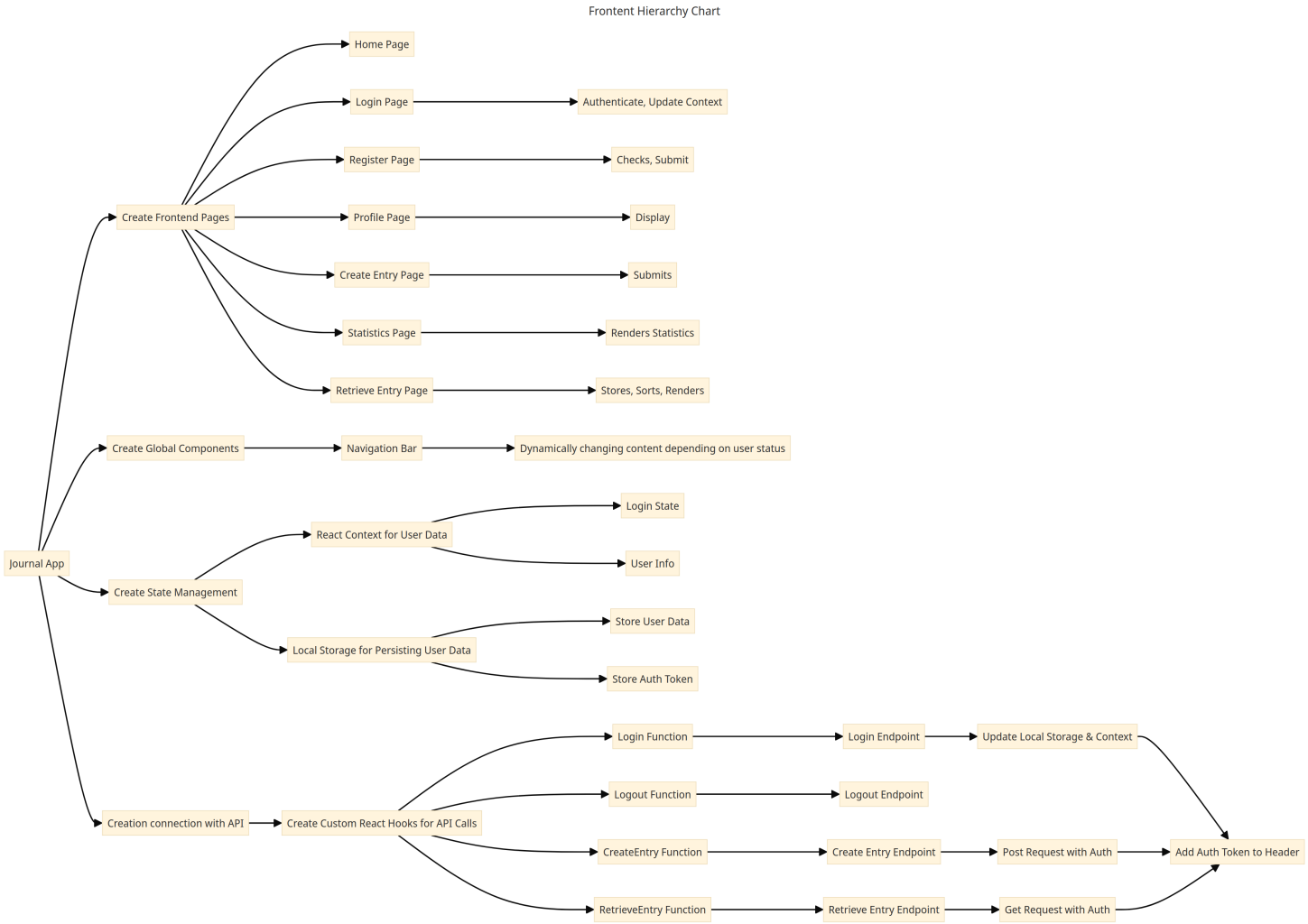

Figure 3.2: Frontend Hierarchy Diagram

3.4.3 Flowchart

Now, to bring everything together, I have designed a flowchart to depict the flow of operations across the entire system. The overall narrative of the flowchart is that the user interacts with the frontend web pages, and it would trigger functions in the frontend JavaScript code. These functions would perform operations such as making requests to the RESTful API endpoints exposed by the Backend or making changes to the storage in the browser. The front end also manipulates the storage system in the browser while the server handles the database.

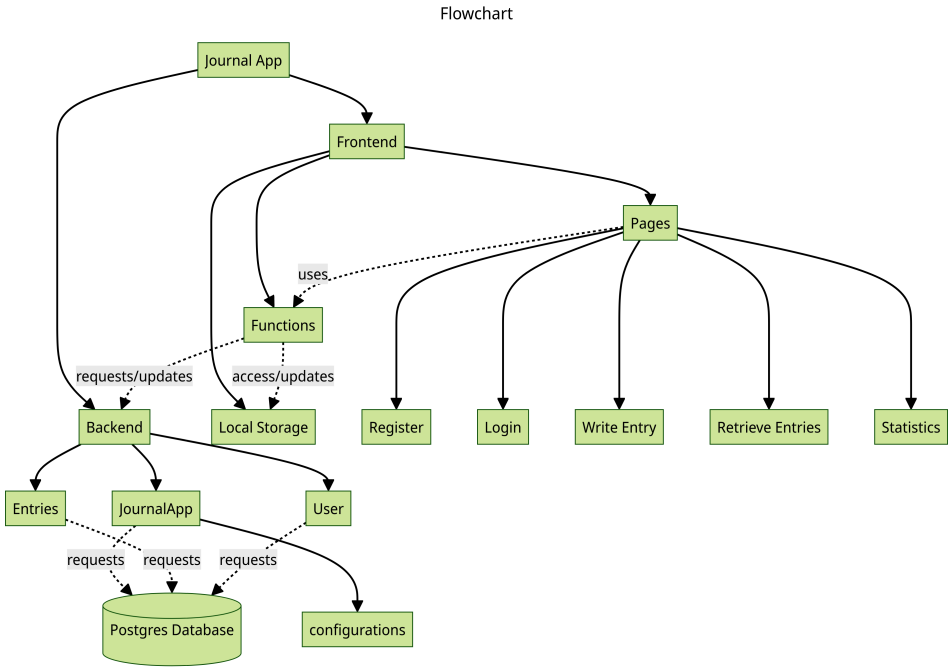

Figure 3.3: Flowchart of the System

Something to note is that users can only access webpages via HTTP(s) requests (probably through a internet browser). The backend is not intended to be accessed by the user directly. Communication with the server happens internally through private requests. This is a good design practice as it ensures that the server is secure and reduces the risk of attacks by malicious users.

3.5 Data Structures / Data Modelling

3.5.1 Entity Relationship Modelling

Storing Data

In my application, data and the flow of data are integral to the system's overall functionality. I have decided to store the data centrally in a relational database. Central storage ensures reliability, scalability and ease of access of data for me. This is good for my users, who expect their data to be reliably stored and easily accessed.

While it is possible to implement a distributed data storage system for a journal app, I have chosen a centralised approach for simplicity and ease of implementation. Not only is a centralised database more accessible for amateur developers like myself but there are also other reasons why centralised storage is better for my journal app. Keeping entries in a structured place means opportunities to analyse the data. If the user opts in for these features, I could perform sentiment analysis on the user's entries or generate statistics and insights based on their journaling patterns. In fact, Google provides a model for analysing sentiment, which is easily accessible through their API. This is an additional thing I may implement after my project is complete.

I have chosen PostgreSQL as the database management system for my application. Postgres is the most powerful Open-Source option available, and once configured, it provides excellent support for my development environment.

Moreover, If I were to host my application, I could easily connect it to a Postgres database through a cloud provider like Railway.

Looking through the specifications of my journal app, I have identified data that needed to be stored and created a database design in the Third Normal Form.

Entity Relationship Modelling

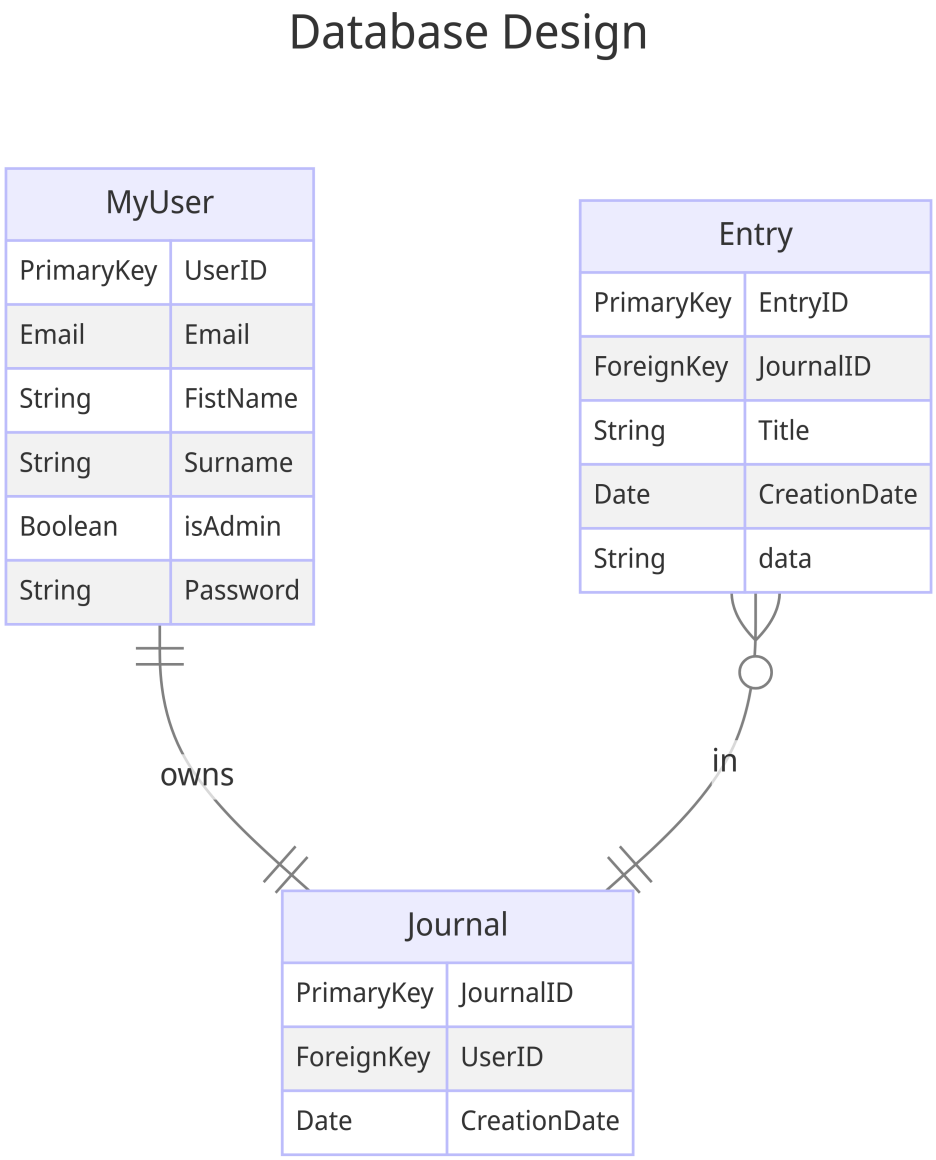

Figure 3.4: Entity Relationship Diagram

3.5.2 External Data Sources

In a bit of my code I implemented API calls to google cloud natural language API to perform sentiment analysis, it is a method included in the entry class in the backend, it can be used to generate a score for the sentiment of the user represented by a float value. This is just a gimmick feature but not core to the project.

3.5.3 OOP Model

One of the core features of Django is the built-in Object Relation Mapping, which enables me to Model my database using Objects. Initially, I explored writing SQL queries to create my database, but as my project grew in complexity, I realised that to use some powerful features provided by Django, for example, the authentication and the administrator interface, I need to integrate Django's Object Relational Mapping functionalities with my database. With this powerful feature, database schema can be defined using Python classes. Additionally, when it comes to interacting with defined data models, Django provides three ways to interact with the database: the Django ORM, raw SQL queries, and the admin panel. Enabling me to have the best of both worlds, the flexibility of powerful SQL queries and the ease of use of the Django ORM (with the bonus of a GUI interface).

In my models.py file, I have defined several classes that represent different objects in my project:

- MyUser: A class which represents my custom user model in Django, with additional fields and methods tailored to my project's requirements.

- MyUserManager: A class that manages MyUser class objects, including creating new user instances. MyUser has a dependency on MyUserManager for its functionality.

- Journal: A class representing a journal entry in my project, with fields such as title, content, and date. Journal has a one to one association with

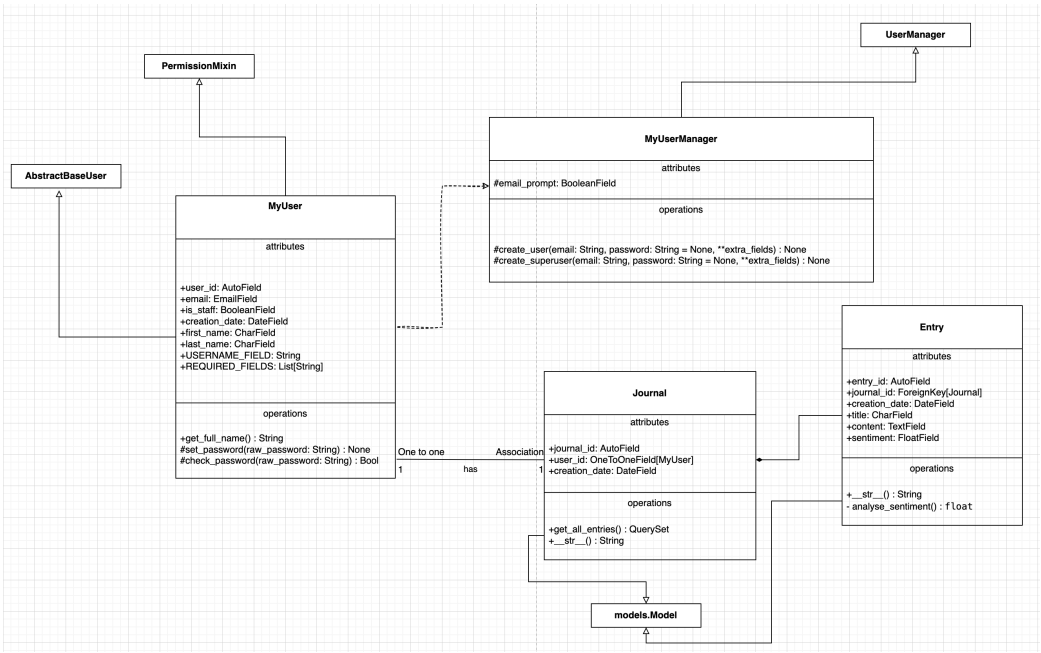

Figure 3.5: UML Class Diagram

MyUser class.

- Entry: A class with a composition relationship with the Journal class, representing a single entry within a journal.

Multiple inheritance is a feature that's not common to all OOP languages since it can lead to potential conflicts and ambiguity in the class hierarchy. In Python, it is supported, and as you can see in the UML diagram, MyUser inherits from AbstractBaseUser and PermissionsMixin. An abstract class is a class that acts as a blueprint for creating a new derived class. I have inherited Django's blueprint for creating users, and I have designed my own user class, which depends on the email field for user registration and login. The second inheritance is from the PermissionsMixin class; this enhances my user class with built-in methods for handling user permissions and authorisation, which is useful.

Both the Journal and Entry classes are inherited from the Django Model class. This allows me to utilise Django's object relational mapping capabil-

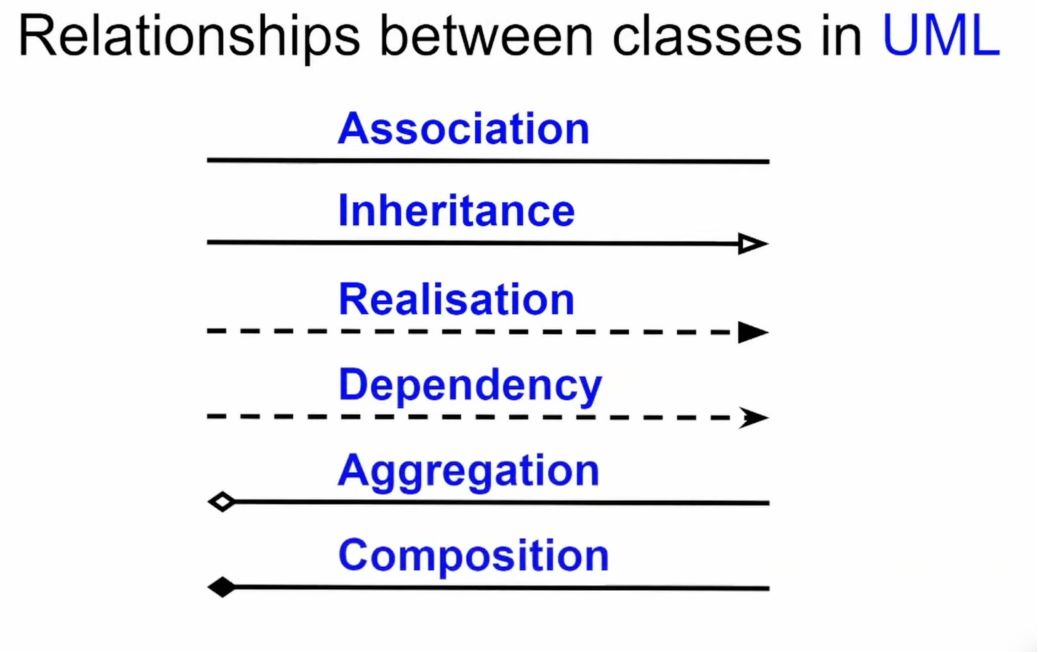

Figure 3.6: Keys explaining the arrows in the UML diagram [**?**]

ities to interact with the database smoothly. Entry has a composition relationship with a Journal because an entry is a part of a journal and cannot exist independently. Similarly, the Journal class has a one-to-one association with the MyUser class, indicating that each journal entry is associated with a specific user.

3.5.4 Handling JSON

JSON data are repeatedly used in my project. JSON are in the form of key-value pairs, similar to a dictionary in Python. In my project, I have used JSON data for communication and it is also just a nice way to store other information. One example off the top of my head is that I store the secret key for my application in a secret.json file which I render with python code.

3.5.5 Typescript Interface

Conceptually Typescript interface is very similar to the Object Oriented Principal of Interface. Both are used to layout the "shape" of an object. It allows a form of abstraction and encapsulation in the code, since only the blueprint of the object is defined and also many methods and properties are bundled into one. In my project, Interface is very useful to model the blueprint of the data I am expected to receive from the backend. By defining interfaces, I can guarantee the format of the is as expected which increase the maintainability of the code and reduce the change of unexpected errors.

3.6 Algorithms

In my frontend application, once all the entries are retrieved from my database, in order to improve the experience of the user, I need there to be a way to sort and filter the entries efficiently. While it is possible to create multiple endpoints in my backend application to return the data in multiple ways (e.g. sorted by date, filtered by category), this can result in unnecessary network requests and slower performance. To optimise the sorting and filtering of entries, I can implement an efficient algorithm in my frontend application.

For my purpose I will implement the sorting based on the entry submission date, although it can easily be modified to include other criteria such as the sentiment of the entries.

Choosing an effective algorithm can improve responsiveness of the application and it can especially do so for long term users of the application who might have a large number of entries in their journal.

Bubble sort is a simple algorithm however it has both terrible time complexity making it unsuitable for large datasets. A divide-and-conquer algorithm is more suitable for my purpose since they are generally more efficient. At first I had chosen and implemented the QuickSort algorithm in my code since I imagined that quicksort would be efficient as it has an average time

complexity of O(n log n). However, I am now rewriting my design now to use Timsort instead. This is because I realised that the entries after retrieved from my database, would be in a structured and near sorted format. Since timsort is partially insertion sort gives it the benefit of being efficient for near sorted data, and the fact it's partially merge sort gives it the benefit of being efficient for large lists of entries. Check out this graphic for a comparison of sorting algorithms.

TODO: edit from quicksort to timsort Writing my own sorting algorithm gives me greater flexibility as I am able to define what is the key that I am deriving my my entries. In my specific case, I have chosen to implement the QuickSort algorithm for sorting the entries based of the creation date of the entry.

Here a simple implementation I have written in Python based on the instructions from OCR A-Level Further Mathematics formula book [**?**], which is the first algorithm I have implemented in my project. The code is as follows:

```python
# my random array
my_array = [3, 6, 8, 10, 1, 2, 1]

def quicksort(array):
    if len(array) == 0:
        return array

    # the pivot point can either be the first element or the last element by convention
    pivot = array[0]
    leftSubarray = []
    rightSubarray = []

    for element in array[1:]:
        # iterating through the array starting from the second element to the end (because I set the pivot to be the first element, otherwise I would iterate from the first element to the second last element)
        if element < pivot:
            leftSubarray.append(element)
        else:
            rightSubarray.append(element)

    leftSubarray = quicksort(leftSubarray)
    rightSubarray = quicksort(rightSubarray)

    return leftSubarray + [pivot] + rightSubarray

# testing the quicksort function
print(quicksort(my_array))
```

Figure 3.7: QuickSort Python Implementation

3.7 User Interface

3.7.1 Wireframe Design

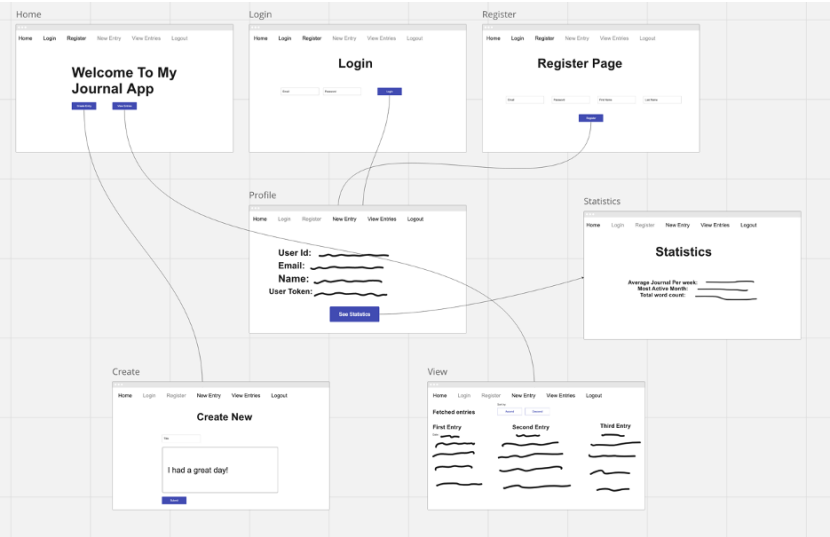

Figure 3.8: This is an overview of all the screens inside my application, it includes the home page, login page, register page, profile page, create page and view page. There are lines connecting the pages to show the flow of the application. Using the navigation bar at the top, the user can navigate to any page from any page.

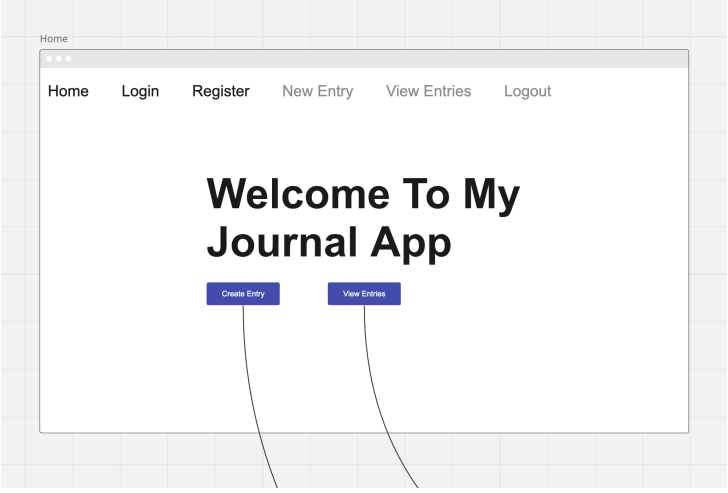

Figure 3.9: The homepage of my application, it includes a navigation bar at the top, a welcome message and buttons for creating and viewing the user's journal entries.

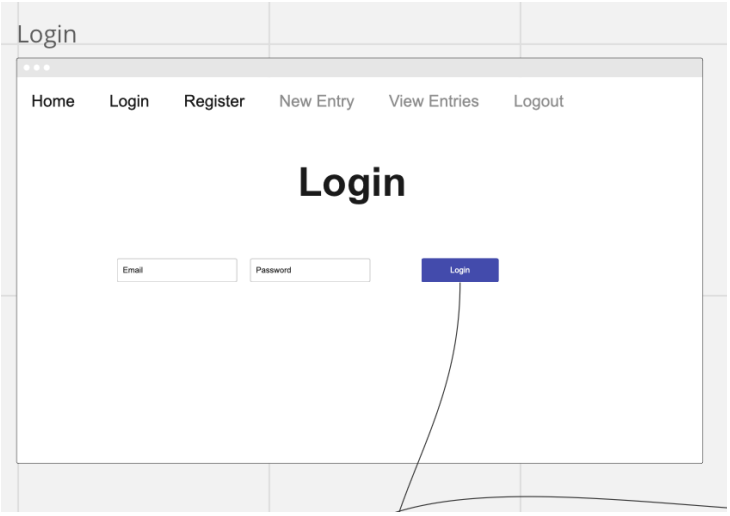

Figure 3.10: The login page of my application, it includes a navigation bar at the top, a form for the user to input their email and password and a button to submit the form. As you can see in the navigation bar, New Entry, View Entries and Logout are all greyed out, this is assuming that the user is not logged in.

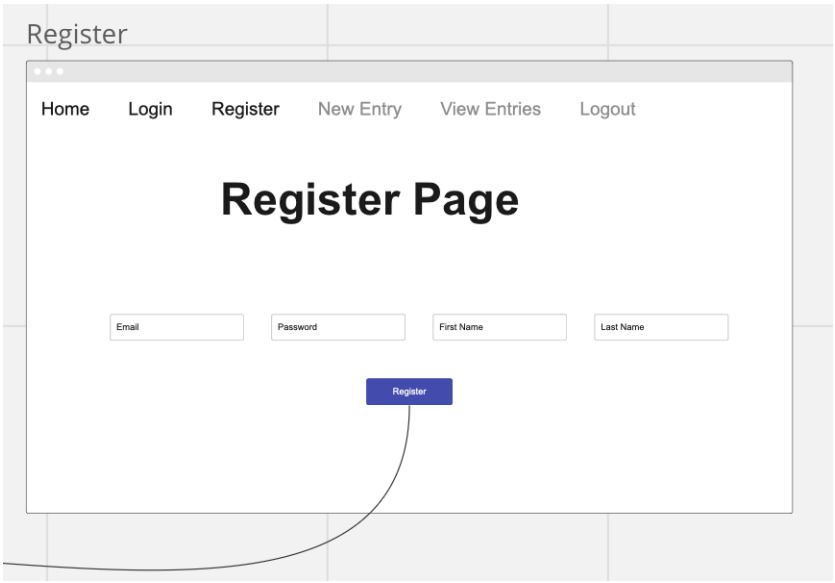

Figure 3.11: Register page is very similar to the login page, it includes a navigation bar at the top, a form for the user inputs as well as the submission button. Registration field requires a couple more fields than the login page.

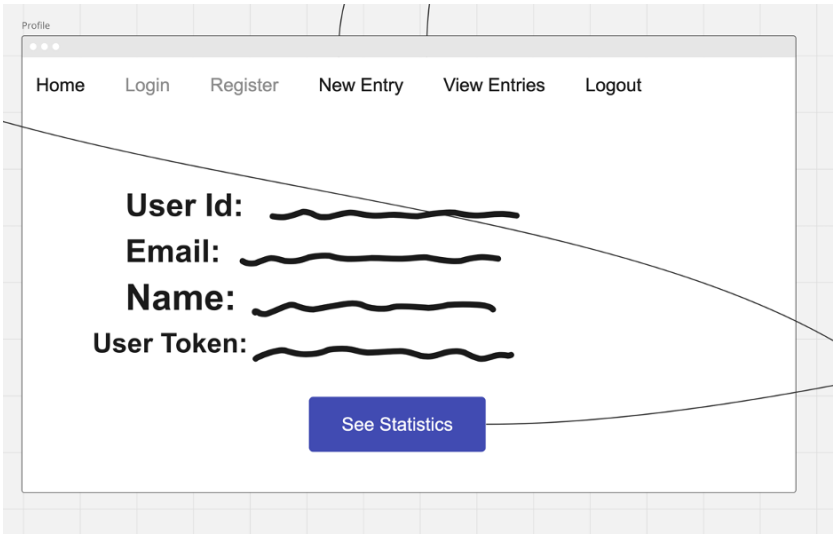

Figure 3.12: Profile page is where the user can view their profile information, as well as place where they can opt in for some additional features.

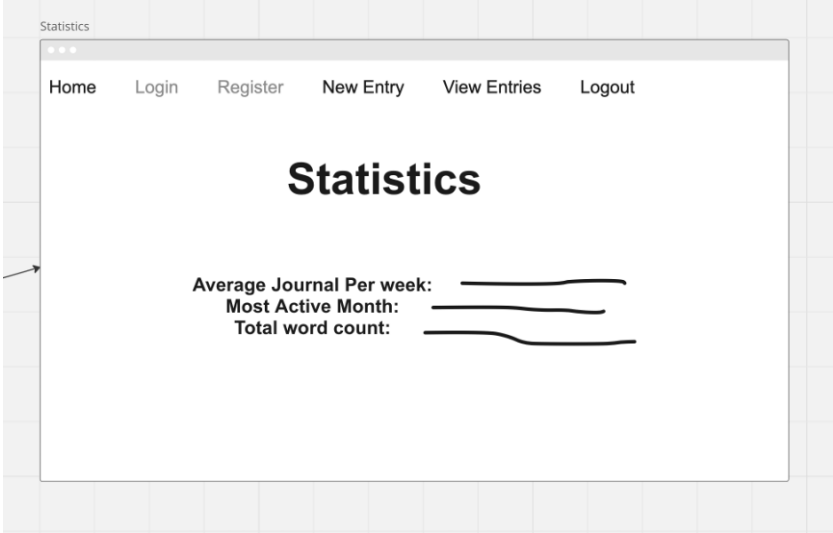

Figure 3.13: Statistics page is where the user can view statistics based on their journal entries pattern.

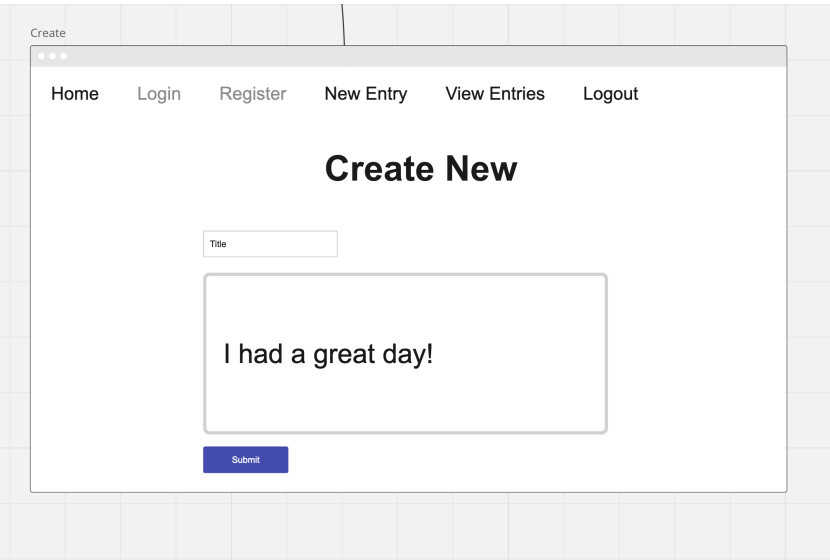

Figure 3.14: Create page is where the user can create a new journal entry. It includes a form for the user to input the title and content of the entry.

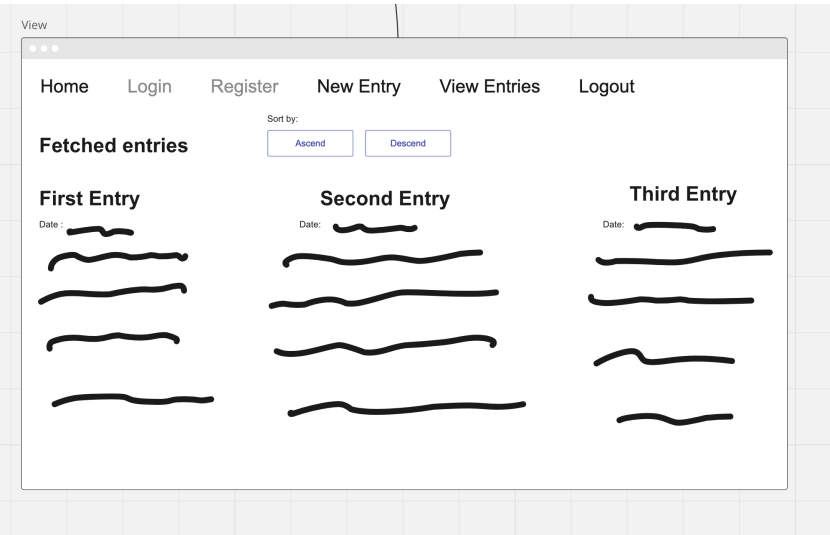

Figure 3.15: View page is where the user can view all of their journal entries. It includes buttons to enable users to sort the entries.

3.8 RESTful API Endpoints

REST effectively allows my frontend application to directly interact with the server through HTTP requests. in this section I will document some of the endpoints I have created in my Django backend. One more thing to add is that I have used Django REST framework to create my API endpoints. This is a framework on top of Django specifically designed for creating RESTful APIs.

3.8.1 Serialization and deserialization of data

Serialization refers to the process of converting complex data type into a format that can easily be transmitted. For my case I need to break down the user object into a JSON format which can then be sent to and use in my frontend. Deserialization is the opposite process, where the JSON data is

converted back into a complex data type. I would need to process the JSON data in the frontend, and the fact that I'm using javascript means that I can easily parse the JSON data, considering that JSON is a subset of JavaScript.

3.8.2 login

The login endpoint is used to authenticate the user. The user sends a POST request to the server with their email and password. The server then checks if the email and salted and hashed value of the password match with records in the database. If they do, the server generates a token and sends it back to the user. This token then can be used to authenticate future requests. For both login and register if successful serialized user data is sent back to the user along with the token.

3.8.3 register

Register is similar, but it is used to create a new user. The user sends a POST request with their email and password and other user information. The server first of perform sanity check on the format of the email via regex, then it checks if the email is already in use. If it is not, the server creates a new user with the email and password and sends back a token as well as other user details in the response.

3.8.4 get entries

This is the endpoint that retrieves all the entries of the user. The user sends a GET request, however the caveat is that the user must be authenticated and the generated token must be attached in teh header of the request. Because the user is validated, the server knows who this user is, and can then retrieve all the entries associated with this user through the foreign key relationship in the database. The server then sends back the entries in JSON format.

3.8.5 create entry

Create entry is used to create a new entry. The user sends a POST request with the title and content of the entry. For this method the user must also be authenticated. The server knows which user this is and through the one to one relationship between the user and the journal entry, the server can create a new entry and associate it with the user's journal. The server then sends back the status of the request. Otherwise it sends back an error message.

3.8.6 get statistics

This endpoint is used to obtain statistics about a user's journal. The server generate statistics based on the user's journal entries.

3.9 Interface Logic

Here I have created flowcharts demonstrating how each of these web pages are suppose to function.

3.9.1 Login and Register Page

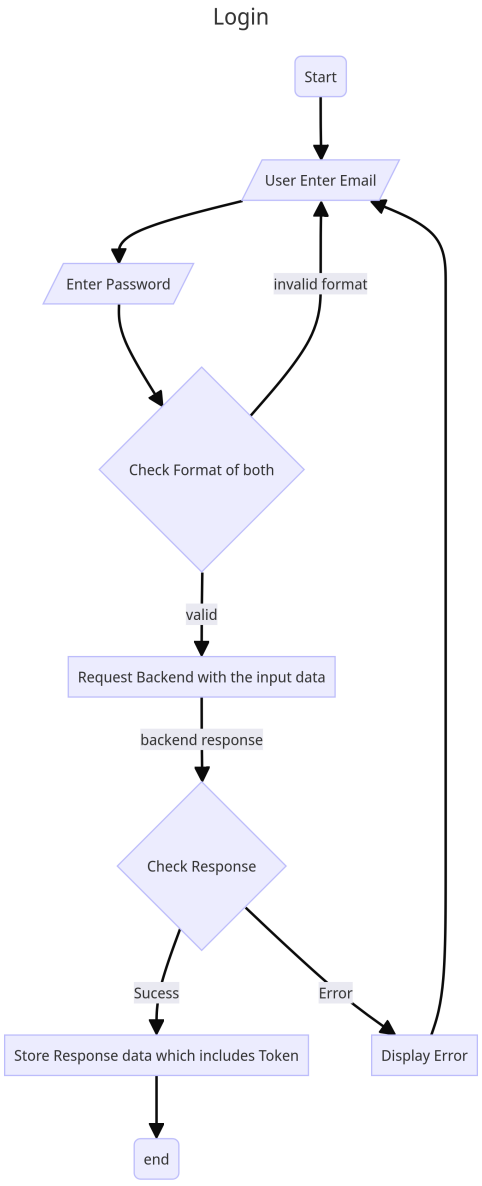

Figure 3.16: Login Page

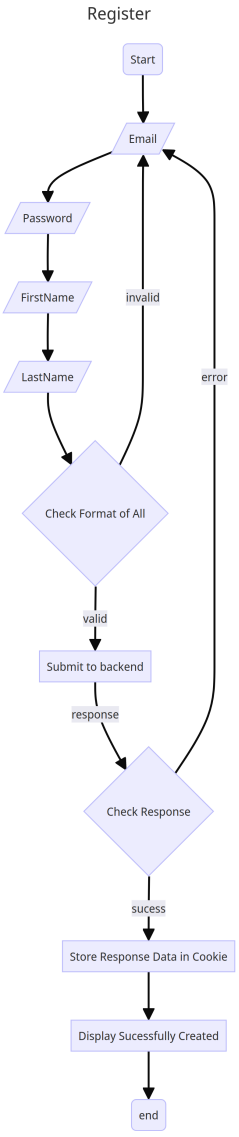

Figure 3.17: Register Page

3.9.2 Retrieve Entries Page

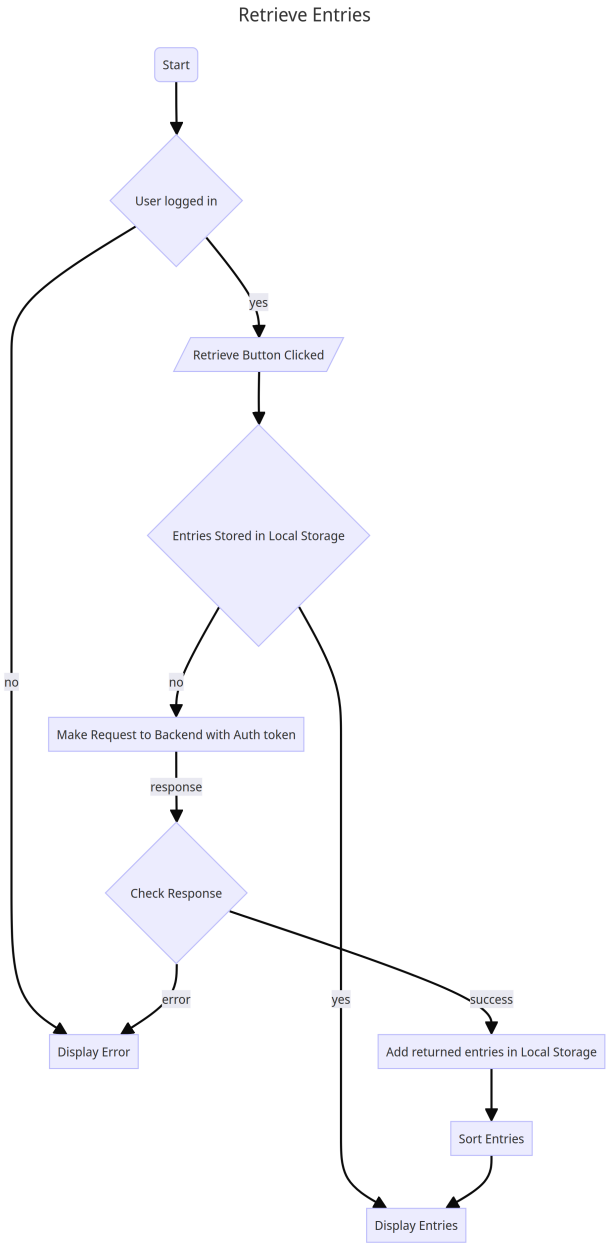

Figure 3.18: Retrieve Entries Page

3.9.3 Other

Create entries follow a similar design to login as it takes in user input and sends it to the server. The profile page is a page that displays the user's information, it reads of the data stored in my browser's storage then renders it. The home page is the first page the user sees when they visit the website. It includes buttons to navigate to other pages.

3.10 Generating User Statistics

The statistics page renders the statistics generated by the server. The server use advance SQL queries to generate the statistics, and then sends it back to the user. Here I will be detailing the design of those queries.

3.10.1 SQL Queries

The exact implementation can be found in the Implementation section of the report. Here I will be detailing how the queries are structured and how they work without delving much into the specifics of the code.

Most Active Month

For this feature, the logic of the query is as such:

1. Join the Journal and Entry tables on the foreign key relationship.
2. Group the entries by the month, and entry count that month using the COUNT function.
3. Order the table by the count in descending order, this way the most active month is at the top.

4. Limit the result to 1, this way only the most active month is returned.

To achieve this a couple of SQL functions are used, namely the EXTRACT, JOIN, GROUP BY, COUNT, ORDER BY and LIMIT functions.

EXTRACT is a function available on PostgreSQL which enable me to extract only a specific detail from a date, in this case the month. This is very handy as I could directly group the entries by the month.

As shown in the Entity Relationship Diagram 3.4, in each journal belonging to a user there are multiple entries, to collect all of these entries I could use a JOIN clause to join the two tables together, at where teh foreign key of Entry is equal to the primary key of the Journal. This way I can collect all the entries associated with a user.

Since I now have the months extracted from each entry, I can now use the GROUP BY clause to aggregate the entries by the month. In SQL if I have a GROUP BY clause, I then can use the COUNT function to count the number of entries in each group. This way I can count the number of entries in each month. The defined group and the function need to both be in the SELECT clause.

Finally, I can use the ORDER BY clause to order the table by the count in descending order, this way the most active month is at the top. I can then use the LIMIT clause to limit the result to 1 to get the most active month.

Average Entry Per Week

This query involves a sub-query which get all the weeks with the entry count, and an outer query that averages this count, the logic is as such:

1. The sub-query used is very similar to the query used to calculate the most active month. What happens here is that instead of grouping all the entries by month, they are grouped by their year and ISO week [**?**].

This enables me to get the number of entries all the individual weeks the user has ever made an entry.

2. Then an outer query perform an average on the entry count column, this way I can get the average number of entries the user makes per week.

The new function used here is the AVG function which take a field of a table as a parameter then perform an average of all the record's value in that field then an output of the average is returned. For me I got all the week's and the number of entries in that week, then I averaged the number of entries in each week to get the average number of entries per week.

Total Word count

This query I estimate the total word count by counting the number of spaces in the content of the entries. I basically assumed here that between every two words there is a single space, so if i count the number of spaces +1(accounting for the first/last word), I can get the word count .The logic is as follows:

1. In a similar way I get all the entries of a journal by performing the same join.

2. Then I apply this formula to the content of the entries: "length of the content− length of the content without spaces + 1". This way I can get the number of words in the content.

3. I then sum all the words in the content of all the entries to get the total word count.

A couple of new function used here include LENGTH, SUM, REPLACE.

Length will be used to get the length of the content of the entries, and the length of the content without spaces.

To create the content without spaces, I can use the REPLACE function to replace all the spaces in the content with an empty string. This way I can get the length of the content without spaces.

Finally, I can use the SUM function to sum all the words in the content of all the entries to get the total word count.

3.11 Hardware & Software Requirements

Specify the hardware and software requirements for the project.

Requirement ID	**Description**
1	Development Machine: Apple MacBook Pro 14-inch with M1 chip, 8 GB of RAM, and 256 GB SSD. This is more than sufficient for my development, React and Python are also well supported on Apple Silicone.
2	Server for Deployment: Initially it could be configured with 2 vCPUs, 4 GB RAM, and 50 GB SSD storage which should be enough for the backend, it should also be easily scalable.
3	End-User Devices: The app should be accessible on devices with internet connection and a web browser. This includes smartphone, laptop, even some smart IOT devices like your smart TV.

Table 3.1: Hardware Requirements for Journal App

Requirement ID	Description
1	Development Environment: Visual Studio Code with the extensions for efficient coding.
2	Terminal: I use Alacritty as my terminal emulator, the default shell on my device is zsh.
3	Package Manager: Homebrew for macOS, very useful for installing new things for example postgres was installed through this.
4	Database: PostgreSQL, as well as pgAdmin 4 for database administration and interaction.
5	Backend Framework: Python, Django REST framework, which is also compatible with my database of choice.
6	Frontend Technologies: Typescript, React, Vite.js.
7	Version Control: Git for source code management, I use it to frequency commit my code, it is also synced with Github for backup.
8	Deployment: Many options are available if I were to host my application, notably ones commonly used include Google Cloud, AWS, there are also ones built for easy deployment ones such as Railway, Heroku and Netlify.

Table 3.2: Software Requirements for Journal App

Chapter 4

Technical Implementation

In this section of my report, I will highlight section of my code with explanations. One thing I will also discuss is best practices and patterns that I have used in my project, such as defensive programming and modularity, etc.

4.1 File Structure

See the following of the tree structure of the project. This is created using the terminal tree command on a freshly cloned version of my repository so that only important files are shown:

Table 4.1: Group A Skills, Found in the Project (skills include all of Models and most of Algorithms)

Group A Skill	**Found in**
Complex data model in database (eg several interlinked tables)	These models are found in the implementation of my backend
Hash tables, lists, stacks, queues, graphs, trees or structures of equivalent standard	
Files(s) organised for direct access	Shown in this section
Complex scientific/mathematical/robotics/control/business model	User session persistance as an example complex control model
Complex user-defined use of object-orientated programming (OOP) model, eg classes, inheritance, composition, polymorphism, interfaces	Created upon user's registration and also defined when new entries are created.
Complex client-server model	Flow of data between frontend application and backend via RESTful API
Cross-table parameterised SQL	Used throughout due to the relationship between my models (design detailed in the previous section)
Aggregate SQL functions	Used when gathering statistics about user's journal
List operations	the journal is handled as an array of entries in the frontend,
Hashing	Hashing, salting of password for storage as well as validating the user password by comparing hash
Recursive algorithms	Used for the implementation of sorting algorithm
Mergesort or similarly efficient sort	
Dynamic generation of objects based on complex user-defined use of OOP model	
Server-side scripting using request and response objects and server-side extensions for a complex client-server model	Fundamental to my application as this is used betwe
Calling parameterised Web service APIs and parsing JSON/XML to service a complex client-server model	

4.1.1 Backend

```
├── ERdiagram.png
├── Email
│   ├── __init__.py
│   ├── apps.py
│   ├── cron.py
│   ├── sendEmail.py
│   └── tests.py
├── Entries
│   ├── __init__.py
│   ├── admin.py
│   ├── apps.py
│   ├── migrations
│   │   └── __init__.py
│   ├── models.py
│   ├── test.rest
│   ├── tests.py
│   ├── urls.py
│   └── views.py
├── JournalApp
│   ├── __init__.py
│   ├── asgi.py
│   ├── settings.py
│   ├── urls.py
│   ├── views.py
│   └── wsgi.py
├── README.md
├── Users
│   ├── __init__.py
│   ├── admin.py
│   ├── apps.py
│   ├── migrations
│   │   └── __init__.py
│   ├── models.py
│   ├── serializers.py
│   ├── signals.py
```

```
│   ├── test.py
│   ├── test.rest
│   ├── urls.py
│   ├── utils.py
│   └── views.py
├── manage.py
├── package-lock.json
├── package.json
├── requirements.txt
└── secret.json
7 directories, 39 files
```

4.1.2 Frontend

```
├── README.md
├── index.html
├── package-lock.json
├── package.json
├── public
│   └── vite.svg
├── src
│   ├── App.css
│   ├── App.tsx
│   ├── api
│   │   ├── axios.ts
│   │   ├── getAuthTokenFromCookie.ts
│   │   └── useApi.ts
│   ├── assets
│   │   └── react.svg
│   ├── components
│   │   └── layout
│   │       ├── NavBar.css
│   │       └── NavBar.tsx
│   ├── context
│   │   └── userContext.tsx
│   ├── hooks
```

```
│   │   ├── useAuth.ts
│   │   ├── useFetchData.tsx
│   │   ├── useUser.ts
│   │   └── useUserContext.ts
│   ├── index.css
│   ├── main.tsx
│   ├── pages
│   │   ├── HomePage.tsx
│   │   ├── ProfilePage.tsx
│   │   ├── entry
│   │   │   ├── CreateEntryPage.tsx
│   │   │   ├── JournalEntriesPage.tsx
│   │   │   └── sortEntries.ts
│   │   ├── login
│   │   │   ├── LoginPage.tsx
│   │   │   └── useLoginAPI.tsx
│   │   └── register
│   │       └── RegisterPage.tsx
│   └── vite-env.d.ts
├── tsconfig.json
├── tsconfig.node.json
└── vite.config.ts
15 directories, 36 files
```

4.2 Key Code Segments

In this section, I will be highlighting key code segments from my project which I believe are important to understand the implementation of the project. To provide context, the source code from where these snippets are taken from are also provided in the appendix.

4.3 Data Structures

4.3.1 Data Models

Data models are the backbone of my backend server. In my design section I have already discussed the data models that I have used in my project. Here I will provide the implementation of the data models in Django.

Modelling the User

```
from django.db import models
from django.contrib.auth.models import AbstractBaseUser, UserManager,
↪  PermissionsMixin

from .utils import clean_email, hash_password, validate_email,
↪  verify_password

# Create your models here.


def check_email_and_password(email: str, password: str | None) -> bool:
    if password is None:
        raise ValueError("Password is not valid")
    if not validate_email(email):
        raise ValueError("Email is not valid")
    return True


class MyUserManager(UserManager):
    def create_user(self, email, password=None, **extra_fields):
        cleaned_email = clean_email(email)
        is_valid = check_email_and_password(cleaned_email, password)
        if is_valid:
            user = self.model(email=cleaned_email, **extra_fields)
            user.set_password(password)
            user.save()

    def create_superuser(self, email, password=None, **extra_fields):
```

```
        cleaned_email = clean_email(email)
        is_valid = check_email_and_password(cleaned_email, password)
        if is_valid:
            user = self.model(email=cleaned_email, **extra_fields)
            user.set_password(password)
            user.is_staff = True
            user.is_superuser = True
            user.save()


class MyUser(AbstractBaseUser, PermissionsMixin):
    """A Users table used to store the data of my user a subclass
    ↪  implementing the default User model"""

    user_id = models.AutoField(primary_key=True)
    email = models.EmailField(unique=True)
    is_staff = models.BooleanField(default=False)
    creation_date = models.DateField(auto_now_add=True)
    first_name = models.CharField(max_length=30, null=True)
    last_name = models.CharField(max_length=30, null=True)
    email_prompt = models.BooleanField(default=False, null=True)

    # By convention, the manager attribute are named objects.
    objects = MyUserManager()

    USERNAME_FIELD = "email"
    REQUIRED_FIELDS = ["first_name", "last_name"]

    def get_full_name(self) -> str:
        return f"{self.first_name} {self.last_name}"

    def set_password(self, raw_password: str) -> None:
        self.password = hash_password(raw_password)

    def check_password(self, raw_password: str) -> bool:
        return verify_password(raw_password, self.password)

```

To improve maintainability and readability, I have defined many helper functions in a separate file called utils.py. This is a good practice and in my codebase I try to follow the Single Responsibility Principal as much as possible. This makes the code more self contained, and also makes it easier to perform

unit tests on the helper functions. In the code included in the appendix it will contain all the relevant codes relating to what I talk about in this section of the report.

TODO: Explain the code above and below.

Modelling the Journal and Entries

```
from email.policy import default
from django.db import models, connection
from Users.models import MyUser
from django.utils import timezone

# sentiment analysis
from google.cloud import language_v2


# Create your models here.


class Journal(models.Model):
    """A Journals table pointing to the owner used to link entries
    ↪  together with the user"""

    journal_id = models.AutoField(primary_key=True)
    user_id = models.OneToOneField(MyUser, on_delete=models.CASCADE)
    creation_date = models.DateField(auto_now_add=True)

    def get_all_entries(self):
        return self.entry_set.all()

    def get_most_active_month(self):
        with connection.cursor() as cursor:
            cursor.execute(
                f"""
SELECT
  EXTRACT(MONTH FROM "Entries_entry".creation_date) AS month,
  COUNT(*) AS entry_count
FROM
  "Entries_entry"
```

```
32  JOIN
33    "Entries_journal" ON "Entries_journal".journal_id =
      ↪  "Entries_entry".journal_id_id
34  WHERE
35    "Entries_journal".journal_id = {self.journal_id}
36  GROUP BY
37    month
38  ORDER BY
39    entry_count DESC
40  LIMIT 1;
41                  """
42              )
43              row = cursor.fetchone()
44              return row[0] if row else None
45
46      def get_average_entries_per_week(self):
47          with connection.cursor() as cursor:
48              cursor.execute(
49                  f"""
50  SELECT
51    AVG(entry_count)
52  FROM (
53    SELECT
54      EXTRACT(YEAR FROM "Entries_entry".creation_date) AS year,
55      EXTRACT(WEEK FROM "Entries_entry".creation_date) AS week,
56      COUNT(*) AS entry_count
57    FROM
58      "Entries_entry"
59    JOIN
60      "Entries_journal" ON "Entries_journal".journal_id =
      ↪  "Entries_entry".journal_id_id
61    WHERE
62      "Entries_journal".journal_id = {self.journal_id}
63    GROUP BY
64      year, week
65  )
66  LIMIT 1;
67                  """
68              )
69              row = cursor.fetchone()
70              return row[0] if row else None
71
72      def get_total_word_count(self):
73          with connection.cursor() as cursor:
74              cursor.execute(
```

```
                f"""
SELECT
  SUM(
    LENGTH("Entries_entry".content) -
    ↪  LENGTH(REPLACE("Entries_entry".content, ' ', '')) + 1
  ) AS total_word_count
FROM
  "Entries_entry"
JOIN
  "Entries_journal" ON "Entries_journal".journal_id =
  ↪  "Entries_entry".journal_id_id
WHERE
  "Entries_journal".journal_id = {self.journal_id};
                """
            )
            row = cursor.fetchone()
            return row[0] if row else None

    def __str__(self):
        return f"{self.user_id}'s Journal"


class Entry(models.Model):
    """An Entry table used to store the data of t journal entry and also
    ↪  tell us which journal it belongs to"""

    entry_id = models.AutoField(primary_key=True)
    journal_id = models.ForeignKey("Journal", on_delete=models.CASCADE)
    creation_date = models.DateField(default=timezone.now, editable=True)
    title = models.CharField(max_length=50, default="Untitled")
    content = models.TextField()
    sentiment = models.FloatField(default=None, blank=True, null=True)

    def __str__(self) -> str:
        return f"Entry {self.entry_id} from {self.journal_id}"

    # analyse data and return sentiment score range from -1 to 1 by
    ↪  calling google cloud api
    def analyse_sentiment(self) -> float:
        client = language_v2.LanguageServiceClient()
        document = language_v2.Document(
            content=self.content,
            ↪  type_=language_v2.Document.Type.PLAIN_TEXT
        )
        response = client.analyze_sentiment(request={"document":
        ↪  document})
```

```
115         return response.document_sentiment.score
```

TODO: add interfaces

4.4 Algorithms

Detail the implementation of algorithms.

TODO: include sorting algorithms

4.4.1 Implementation of the SQL Queries

To provide context, the following are the SQL queries that are implemented in the Django ORM as methods of the Journal model. This allow me to leverage powerful features of Python/Django enabling me to make more dynamic queries with ease. For example I can access the journal id of the current instance of the model using self.journal_id, which can then be used directly in the query through string interpolation. In action, when a user accesses the API endpoint to get their journal statistics, self refers to user making the request. Therefore, the query will return the statistics of the journal that the user who just made that request. Having these as methods is also a good practice as it allows for better modularity and readability.

Most Active Month

```
1  SELECT
2    EXTRACT(MONTH FROM "Entries_entry".creation_date) AS month,
3    COUNT(*) AS entry_count
4  FROM
5    "Entries_entry"
```

```
JOIN
  "Entries_journal" ON "Entries_journal".journal_id =
  ↪  "Entries_entry".journal_id_id
WHERE
  "Entries_journal".journal_id = {self.journal_id}
GROUP BY
  month
ORDER BY
  entry_count DESC
LIMIT 1;
```

AS is used to alias the column names which makes the query more readable and efficient to write.

Average Entry Count Per Week

```
SELECT
  AVG(entry_count)
FROM (
  SELECT
    EXTRACT(YEAR FROM "Entries_entry".creation_date) AS year,
    EXTRACT(WEEK FROM "Entries_entry".creation_date) AS week,
    COUNT(*) AS entry_count
  FROM
    "Entries_entry"
  JOIN
    "Entries_journal" ON "Entries_journal".journal_id =
    ↪  "Entries_entry".journal_id_id
  WHERE
    "Entries_journal".journal_id = {self.journal_id}
  GROUP BY
    year, week
)
```

Total Word Count

```
SELECT
    SUM(
      LENGTH("Entries_entry".content) -
      ↪  LENGTH(REPLACE("Entries_entry".content, ' ', '')) + 1
    ) AS total_word_count
  FROM
    "Entries_entry"
  JOIN
    "Entries_journal" ON "Entries_journal".journal_id =
    ↪  "Entries_entry".journal_id_id
  WHERE
    "Entries_journal".journal_id = {self.journal_id};
```

4.5 Modularity

Modularity is shown throughout my codebase and I try splitting code into modules when it makes sense. A couple of examples I thought were worth mentioning are the hooks I have created in my frontend to make call to the backend API and also the modules I created to enable persist login in the frontend using context.

4.5.1 API call functions

Here I will include the code for the hooks I have created to make API calls to the backend. This module is repeatedly reused throughout the frontend to make calls to the backend. For demonstration sake, I will be only including one example of my useApi hook in action.

useApi.ts

```
import { useState,  useCallback } from "react";
import  { AxiosError, AxiosRequestConfig, AxiosResponse } from "axios";
import { myApiCall } from "./axios";

// generic interface for the hook's return type
interface ApiHookState<T> {
  isLoading: boolean;
  error: AxiosError | null;
  data: T | null;
  fetchData: () => Promise<void>;
}

export default function useApi<T>(
  config: AxiosRequestConfig,
): ApiHookState<T> {
  const [isLoading, setIsLoading] = useState(false);
  const [error, setError] = useState(null);
  const [data, setData] = useState<T | null>(null);

  const fetchData = useCallback(async () => {
        setIsLoading(true);
        try {
          const response: AxiosResponse<T> = await myApiCall(config);
          setData(response.data);
        } catch (error: any) {
          setError(error);
        } finally {
          setIsLoading(false);
        }
      }, [config]);
      return { isLoading, error, data, fetchData };
    }
```

I have created useApi hook because whenever I make a call to the backend, I want to be able to retrieve information about wether the call is loading, if there is an error, and the data that is returned. This piece of code is built on top of the Axios library which is a promised[1] based library for

[1]Promise is an asynchronous operation that can either succeed or fail.

HTTP requests. A promise exists in one of three states: pending, fulfilled, or rejected. See my implementation to see how extract additional information from my call.

A generic interface is used to define the return type of the hook. This is useful since it increases the reusability of the module.

TODO: finish explaining A callback function is

from axios.ts

```
...
export const myApiCall: AxiosInstance = axios.create({
    baseURL: "http://127.0.0.1:8000/",
    timeout: 5000, // time before the request times out in milliseconds
    headers: {
        "Content-Type": "application/json", // I use json as the default
        ↪  content type

    },
});
```

RegisterPage.tsx

```
import { useEffect, useState } from "react";
import { useNavigate } from "react-router-dom";
import useAuth from "../../hooks/useAuth";
import useApi from "../../api/useApi";
import { User } from "../../hooks/useUser";

interface registerRequestData {
  email: string;
  password: string;
```

```
  first_name: string;
  last_name: string;
}

export default function RegisterPage(): JSX.Element {
  // my states
  const [registerRequestData, setregisterRequestData] =
    useState<registerRequestData>({} as registerRequestData);
  const navigate = useNavigate();
  const { login } = useAuth();

  const { isLoading, error, data, fetchData } = useApi<User>({
    url: "/user/register/",
    method: "post",
    data: {
      email: registerRequestData.email,
      password: registerRequestData.password,
      first_name: registerRequestData.first_name,
      last_name: registerRequestData.last_name,
    },
  });

  useEffect(() => {
    if (data) {
      // redirect to profile page
      login(data);
      navigate("/profile");
    }
  }, [data]);
  const onRegisterFunc = () => {
    console.log("Registering...");
    fetchData(); // triggers myApiCall
  };
  return (
    <div>
      <input
        placeholder="email"
        value={registerRequestData.email}
        onChange={(e) =>
          setregisterRequestData({
            ...registerRequestData,
            email: e.target.value,
          })
        }
      />
```

```
55        <input
56          type="password"
57          placeholder="password"
58          value={registerRequestData.password}
59          onChange={(e) =>
60            setregisterRequestData({
61              ...registerRequestData,
62              password: e.target.value,
63            })
64          }
65        />
66        <input
67          placeholder="first name"
68          value={registerRequestData.first_name}
69          onChange={(e) =>
70            setregisterRequestData({
71              ...registerRequestData,
72              first_name: e.target.value,
73            })
74          }
75        />
76        <input
77          placeholder="last name"
78          value={registerRequestData.last_name}
79          onChange={(e) =>
80            setregisterRequestData({
81              ...registerRequestData,
82              last_name: e.target.value,
83            })
84          }
85        />
86        {isLoading && <div>Loading...</div>}
87        {error && <div>{error.message}</div>}
88        {error && <div>{JSON.stringify(error.response?.data)}</div>}
89        {data && (
90          <div>
91            Registered successfully
92            <div>Welcome {data.first_name}</div>
93          </div>
94        )}
95
96        <br />
97
98        <input
99          disabled={isLoading}
```

```
100         type="button"
101         value="Register"
102         onClick={onRegisterFunc}
103       />
104     </div>
105   );
106 }
```

TODO: Explain the code above. This is the example of using my useApi hook. The versatility is demonstrated here .....

4.5.2 Persistent Login Session

useUser.ts

```
1  import Cookies from "universal-cookie";
2  import useUserContext from "./useUserContext";
3
4  export interface User {
5      user_id: number;
6      email: string;
7      first_name: string;
8      last_name: string;
9      authToken?: string;
10     email_prompt: boolean;
11   }
12 export default function useUser() {
13     const { updateUser } = useUserContext();
14
15     const cookies = new Cookies();
16
17     const addUser = (user: User) => {
18         cookies.set("user", user, { path: "/",
19         secure: true,
20         sameSite: "strict"
21
22     });
23         updateUser(user);
```

```
        console.log("User added");
    }

    const removeUser = () => {
        cookies.remove("user");
        updateUser(null);
        console.log("User removed");
    }
    const getUser = (): User | undefined  => {
        return cookies.get("user");
    }

    return {
        addUser,
        removeUser,
        getUser
    }
}
```

TODO: Explain the code above.

userContext.tsx

```
import Cookies from "universal-cookie";
import { User } from "../hooks/useUser";
import React, { createContext, useState, ReactNode, useEffect } from
↪  "react";

export interface UserContextType {
  user: User | null;
  updateUser: (user: User | null) => void;
}

export const UserContext = createContext<UserContextType | undefined>(
  undefined
);

export const UserProvider: React.FC<{ children: ReactNode }> = ({
```

```
  children,
}) => {
  const cookies = new Cookies();
  const [user, setUser] = useState<User | null>(() => {
    // prevent my user being lost on refresh
    const savedUser = cookies.get("user");
    if (savedUser) {
      return savedUser;
    }
  });

  const updateUser = (newUser: User | null) => {
    setUser(newUser);
    console.log("update user for context: ", newUser, user);
  };

  useEffect(() => {
    if (user) console.log("user context: ", user);
  }, [user]);

  return (
    <UserContext.Provider value={{ user, updateUser }}>
      {children}
    </UserContext.Provider>
  );
};
```

TODO: Explain the code above.

useAuth.ts

```
import { User } from "./useUser";
import useUser from "./useUser";

export default function useAuth() {
    const { getUser, addUser, removeUser } = useUser();

    const login = (user: User) => {
```

```
        addUser(user);
    }

    const logout = () => {
        removeUser();
    }

    const isAuthenticated = () => {
        return getUser() !== undefined;
    }

    return {
        login,
        logout,
        isAuthenticated
    }}

```

TODO: Explain the code above.

useUserContext.ts

```
import { useContext} from "react";
import { UserContext, UserContextType  } from "../context/userContext";

export default function useUserContext() {
    const context = useContext(UserContext) as UserContextType;

    if (context === undefined) {
        throw new Error("useUser must be used within a UserProvider");
    }

    return context;

}
```

TODO: Explain the code above.

4.5.3 Virtual Environment

Discuss the modularity of the project.

4.6 Defensive Programming / Robustness

Explain the measures taken for defensive programming and robustness.

TODO: Sanity checking TODO: Error handling TODO: Input validation TODO: Authentication and Authorization TODO: CSRF TODO: Token Authentication/ Session Management TODO: XSS (cookies vs local storage) TODO: SQL Injection TODO: Password Hashing TODO: Rate Limiting TODO: CORS

TODO: secret key management

Cross Site Request Forgery (CSRF)

4.6.1 Server Based Authentication

Cross-Site Scripting (XSS)

4.7 Patterns

4.7.1 Single Responsibility Principal

4.8 Credential Management

TODO: compare against spec for group A skills

Chapter 5

Testing

5.1 Test Strategy

In this section I will compare my my finished project with the original specification proposed in my Analysis section. Furthermore, I will demonstrate my app as a whole.

Video demonstration of the app will be provided to show the app in action, this will provide a complete overview of the app and its features. This is my acceptance testing as it will show the app in action and how it meets the original requirements.

The primary testing strategy for my backend project would be Unit testing to see if all the smaller components of my app work as intended. Django provide its own testing framework built on top of Python's unittest module. I will be using this to test my models and the methods within them, as well as the utility functions I have created in my app. Normal, boundary and edge cases will be tested where applicable. In addition, screenshots of the PostgreSQL database will also be shown.

For the API endpoints I will be using REST client provided by Visual

Studio Code to make HTTP requests to the server and compare the result with my anticipated result. I will be doing this rather than using Django's own testing framework because I want to model real world usage of my app. This is my system testing for the backend as it will test the functionality of the server as a whole.

For my frontend Interface, I will be will be testing the app manually by interacting with the components to see if the app behaves as expected. There will be screenshots of the results of the test cases.

5.1.1 Unit Testing

To test my backend code django includes its own... TODO: Unit testing

5.2 Endpoints

To model real world usage, I will be making real HTTP requests to the server and compare the result with my anticipated result. TODO: Endpoints

5.2.1 Rest Client

5.2.2 Django Test Models

5.3 Frontend Interface

TODO: Frontend Interface

5.4 Testing Video

Include a link or description of a testing video. TODO: Test video

5.5 System Tests

Conduct tests against the original requirements specification. TODO: System tests

Chapter 6

Evaluation

6.1 Requirements Specification Evaluation

Evaluate the project against the initial requirements. TODO: Requirements Specification Evaluation

6.2 Independent End-User Feedback

Include feedback from end-users. TODO: Independent End-User Feedback

6.3 Improvements

Discuss potential improvements for the project. TODO: Improvements - Mobile App - Email - better Analysis

TODO: citation TODO: grammar

Chapter 7

Appendix

7.1 Code Listing

Include relevant code listings.

7.1.1 Appendix A – Code Listing

Include code listings as an appendix.