Software Design Document

for

Insight Write

Version 2 approved

Prepared by Miguel Gonzalez Torres, Tony Lau, Hoang Le, Keyvan Mahmoodzadeh Kani, Will May, Hayk Vardapetyan, Jian Verdad, Niyusha Zarnegar

California State University - Los Angeles

May 10th, 2023

Revision History

Name	Date	Reason For Changes	Version
Tony	4/15/24	Minor updates to 4.3	1.1
Tony	4/21/24	Changes to Section 6	1.2
Tony	4/27/24	Changes to Section 6 and 7	1.3
Everyone	4/28/24 - 5/3/24	Final Documentation. Changes to all sections.	2

<Add rows as necessary when the document is revised. This document should be consistently updated and maintained throughout your project. If ANY requirements are changed, added, removed, etc., immediately revise your document.>

1. Introduction (Miguel)

1.1 Purpose

Identify the product whose software requirements are specified in this document, including the revision or release number. Describe the scope of the product that is covered by this SRS, particularly if this SRS describes only part of the system or a single subsystem.
The software requirements specified in this documents pertain to the journaling app, Insight write. The scope of this SRS covers the entire system of the journal app. This document details the requirements for the functionalities, user interface design, database, testing procedures, and maintenance of the application.

1.2 Document Conventions

Describe any standards or typographical conventions that were followed when writing this SRS, such as fonts or highlighting that have special significance. For example, state whether priorities for higher-level requirements are assumed to be inherited by detailed requirements, or whether every requirement statement is to have its own priority.
This SRS follows the following typographical conventions of using bold for headings, bullet points for shorter information, size 12 and Times New Romans.

1.3 Intended Audience and Reading Suggestions

Describe the different types of reader that the document is intended for, such as developers, project managers, marketing staff, users, testers, and documentation writers. Describe what the rest of this SRS contains and how it is organized. Suggest a sequence for reading the document, beginning with the overview sections and proceeding through the sections that are most pertinent to each reader type.
The intended audience for this document includes developers, project managers, marketing staff, and users. The SRS is organized into sections that cover different aspects of the app's design and requirements.

- Developers: Start with the system overview, design considerations,
- Marketing staff: focus on system overview and user interface to understand the appl's features
- Testers: Focus on testing the software and maintaining the software for testing requirements and procedures

1.4 System Overview

Provide a general description of the software system including its functionality and matters related to the overall system and its design (perhaps including a discussion of the basic design approach or organization
Insight write is a journal app that is designed to facilitate journaling and note taking for users. Its functions include creating and managing journal entries, organizing entries by their tags, setting

reminders, and searching and filtering entries. The design emphasizes a user friendly interface design.

2. Design Considerations (Jian, Niyusha)

This section describes many of the issues which need to be addressed or resolved before attempting to devise a complete design solution.

2.1 Assumptions and Dependencies

Describe any assumptions or dependencies regarding the software and its use. These may concern such issues as:

- Related software or hardware
- Operating systems
- End-user characteristics
- Possible and/or probable changes in functionality

We expect the website to be able to run on most popular browsers such as Google Chrome and Firefox. We also intend for it to support popular Operating Systems such as Windows and Linux. The use of Railway we believe will eliminate potential issues with using different Operating Systems to work on this project.

Lack of knowledge when it comes to using a computer will always be an obstacle, however we believe that the website will be simple enough for the majority of people to easily understand how to use. This is especially important since we believe that people who use journals and such tend to not use computers very often.

The growth of social media means an increase in negativity due to the quicker spread of news. We believe our project could help people learn how to have some time to themselves. Everyone needs time to relax.

Possible changes in functionality include the info/analytics dashboard since we are still not sure as to if its implementation will improve our overall project or not. We have had to consider its pros and cons a few times.

2.2 General Constraints

Describe any global limitations or constraints that have a significant impact on the design of the system's software (and describe the associated impact). Such constraints may be imposed by any of the following (the list is not exhaustive):

- Hardware or software environment
- End-user environment

- Availability or volatility of resources
- Standards compliance
- Interoperability requirements
- Interface/protocol requirements
- Data repository and distribution requirements
- Security requirements (or other such regulations)
- Memory and other capacity limitations
- Performance requirements
- Network communications
- Verification and validation requirements (testing)
- Other means of addressing quality goals
- Other requirements described in the requirements specification

You will not need to include all of these. Only the ones that will influence the design of your software

While our target audience includes those who already like to write in journals, we fear that they might prefer the experience of writing their thoughts and feelings down as opposed to typing them on a computer. Optimizing performance could also be an issue since we're more so used to developing to achieve the desired result rather than to optimize the route to the desired result.

We hope to be able to complete this project while relying on free tools along with tools we already own. Our lack of experience will also impact our ability to adapt to what the project needs, such as how to keep the databases secure.

2.3 Goals and Guidelines

Describe any goals, guidelines, principles, or priorities which dominate or embody the design of the system's software. For each such goal or guideline, unless it is implicitly obvious, describe the reason for its desirability. Feel free to state and describe each goal in its own subsubsection if you wish. Such goals might be:

- The KISS principle ("Keep it simple stupid!")

- The Software has a mandatory delivery date that must be met (end of the cd3337 class)
- Emphasis on speed versus memory use
- The product should work, look, or "feel" like an existing product

- Code and Learn: This project will require us to touch on many unfamiliar aspects. That's why it's very important for us to learn about them, not only for this project, but for our futures as well due to their relevance to our major.
- Code Together: Develop our skills on working as a team.
- The product should be friendly and helpful. It's meant to appeal to all ages and extend a warm hand to those troubled with the demands of daily life.
- The project must be finished by the end of CS 3337!

2.4 Development Methods

Briefly describe the method or approach used for this software design. If one or more formal/published methods were adopted or adapted, then include a reference to a more detailed description of these methods. If several methods were seriously considered, then each such method should be mentioned, along with a brief explanation of why all or part of it was used or not used.

These would be things such as the 'WaterFall Development' methods, 'Agile Development', 'Unplanned Mad Scramble Development', or other development models and variations. Describe how these were applied in the case of your project.

We will be using the Agile Development approach to this project. Every week, we will assign one person in our group to be the Scrum Master/Team Coach and another to be the Product Owner. We want to make the most out of each other's strengths, so we have chosen people to work on either the frontend or backend based on how they feel about their skill.

We have decided on using scrums to manage our iterative development. Every Monday and Wednesday, our group has a meeting to discuss not only our progress but also clarify matters that group members might be confused about. We also spend these meetings figuring out how to approach other tasks that our group must take care of. We also have meetings every weekend but the day for when it happens varies quite often due to individual circumstances. These weekend meetings are usually when we assign stories for this week's sprint from the product backlog while also discussing potential updates to our project's requirements through reviewing suggested features.

3. Architectural Strategies (Will, Hayk)

Describe any design decisions and/or strategies that affect the overall organization of the system and its higher-level structures. These strategies should provide insight into the key abstractions and mechanisms used in the system architecture. Describe the reasoning employed for each decision and/or strategy (possibly referring to previously stated design goals and principles) and how any design goals or priorities were balanced or traded-off. Such decisions might concern (but are not limited to) things like the following:

- Architectural Pattern

Our web application will be using Django's framework which has support for the Model-View-Template(MVT) architecture. This architectural pattern will make our code more organized, maintainable, and scalable.

Model: The model component will represent the data structure of our application which handles logic, and also handles interactions within our database. We will create our models using Django's built-in ORM (Object Relational Mapping) which will turn our database tables into Python objects.

Our project has several models:

- User Model (This model will represent the User table in our database, as well as implementing methods related to this model for querying or functionality purposes)
- Journals Model (This model will represent the Journals table in our database, as well as implementing methods related to this model for querying or functionality purposes)
- Meditations Model (This model will represent the Meditations table in our database, as well as implementing methods related to this model for querying or functionality purposes)
- Settings Model (This model will represent the Settings table in our database, as well as implementing methods related to this model for querying or functionality purposes)

View:

In Django, the views will process user requests and generate a response. These view functions will handle different HTTP request methods such as GET, POST, etc… and will interact with models to get data, which will then be rendered with templates. Django uses the urls module to map incoming requests or url patterns to their correct view function.

Our project uses several views to pull information as well as store information to our database. For example, our login and signup views handle logging in users based on the information submitted in the form or through googles third party authentication. Our journal views will handle CRUD operations on journals for each unique user, and update the database accordingly

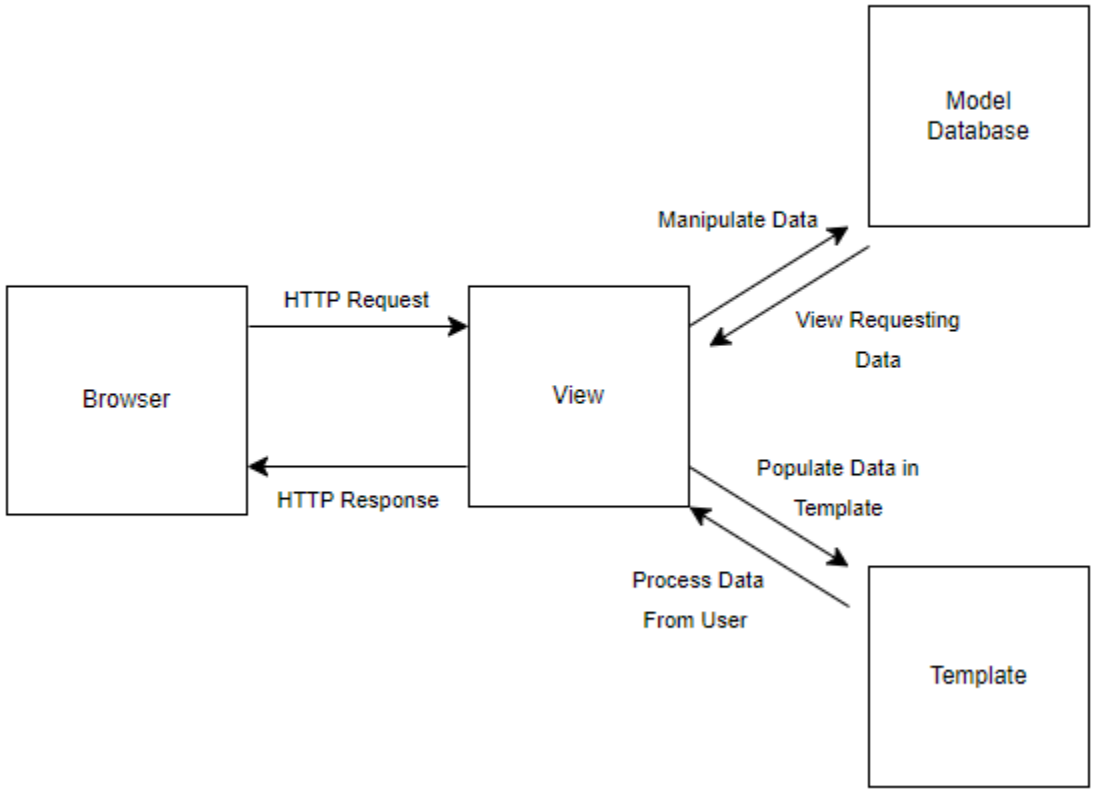
Template:

We will use Django templating to create HTML content which will be displayed to the user in response to their requests. Django's templates allow for dynamic content rendering due to the special DTL syntax. This will serve as the presentation layer.

Our project uses these templates to dynamically display data, as well as making it pretty with CSS and JS. We have several templates for each section of our website including the index.html for each major section in our project (login, signup, journals, meditations)

The reason why we decided to go with this framework as opposed to others such as Flask is because it includes tons of features built in like ORM, form handling, url routing, security features which makes Django the best choice for fast development, and due to the time constraints we have in this class, Django was the better option.

MVT Diagram:



- Use of a particular type of product (programming language, database, library, etc. ...)

Python: Easy to understand, very versatile. It also has a ton of built-in tools and libraries for pretty much everything to make development a smooth process.

- **Pytest:** Simplifies the process of testing by providing automation for writing and executing tests.

Bootstrap: Front-end framework that will make creating dynamic and responsive web applications a lot easier.

OAuth Libraries: Backend libraries that will help implement secure authentication and authorization

- Django-allauth: Used to integrate socialproviders as a way of logging in and being authenticated.

PostgreSQL: This DBMS has a bunch of features that stood out compared to others, and will make our application reliable and scalable.

- We are using Railway to manage our DB.

- Reuse of existing software components to implement various parts/features of the system

- Utility functions
- Django packages
- Database abstraction layer

- Future plans for extending or enhancing the software

- Support for journaling in multiple languages.
- Introduce social features such as sharing journals or insights with support groups or friends.

- User interface paradigms (or system input and output models)

We will be considering allowing to journal with an E-pen to directly manipulate their journal, however, since this is a web application, implementing this feature will be a lower priority compared to other essential features and we will initially stay with text. Another factor that influences this decision that we dont want to get into is the hardware dependency that is associated with the E-pen, as users will need to be on a touch screen compatible device.

- Hardware and/or software interface paradigms

Software Paradigms:

- Python OOP
- Django ORM

- Component-Based Development
- Modular-Based Development
- APIs

- Error detection and recovery

When designing our app, we want to make sure there are no errors that would break any component of the application, so we want to be able to detect and handle these errors and also have recovery strategies in place. We will implement input validation before processing any data to catch any unwanted issues early in the process, and also to verify the integrity of the flow of data. We will also constantly monitor the system using automated testing. For recovery mechanisms, we will implement exception handling, or retrying failed tasks, and also data transaction management that will follow the ACID properties being atomicity, consistency, isolation, and durability.

- External databases and/or data storage management and persistence

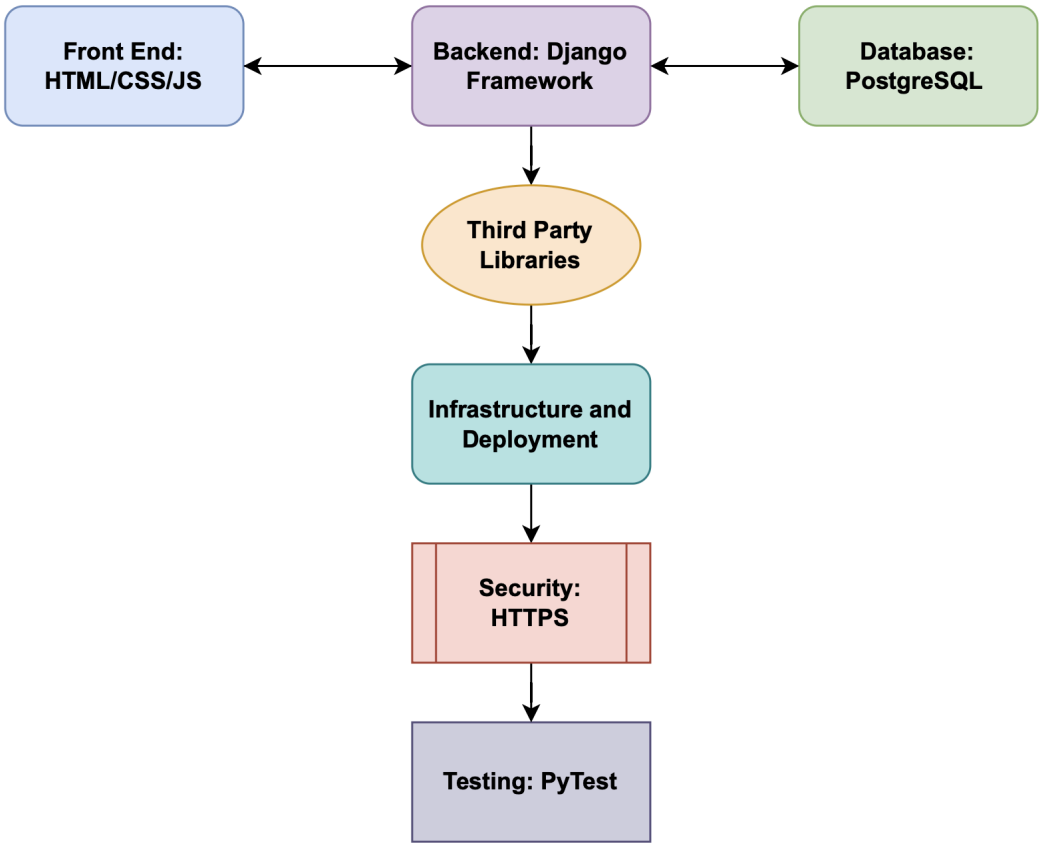
The first step in designing an external database, or managing how data will be stored is by choosing a proper DBMS. We decided to go with PostgreSQL and Railway instead of others relational DBs, because PostgreSQL is an object relational database, and has a range of built in features that other DBs otherwise wouldnt have, such as being able to store and work with JSON data, and array data types. PostgreSQL also provides better support for concurrency transactions and scalability. For data storage, we designed a data model as shown in section 8, which is an efficient way to show the structure and relationship of the data in the application. This kind of model will help reduce alot of redundancy as well as enhance the performance of our application overall. We will also have a test DB, as well as a backup and recovery mechanism in the case we make fatal or irriversible changes that would otherwise affect real user data.

Each significant strategy employed should probably be discussed in its own subsection. Make sure that when describing a design decision that you also discuss any other significant alternatives that were considered, and your reasons for rejecting them (as well as your reasons for accepting the alternative you finally chose).

4. System Architecture (Will, Hayk)

4.1 Logical View:

Key Abstractions



The system is a web application designed for journaling thoughts and experiences.

Functional Modules Overview:

Frontend: Implemented using HTML, CSS, JavaScript and Bootstrap, responsible for the user interface design as well as creating dynamic functionality for our web pages.

Backend: Developed with the Django framework, handles user authentication, journal entry CRUD operations, calendar updates, and user settings/preferences.

Database: Utilizes PostgreSQL (Railway) to store user journal entries and account information.

Third-Party Libraries: Integrated for additional functionalities (django-allauth).

Infrastructure and Deployment: Manages the deployment of the application, including Docker containerization.

Security and HTTPS: Ensures the security of user data and communication over HTTPS.

Testing: Utilizes PyTest for testing purposes.

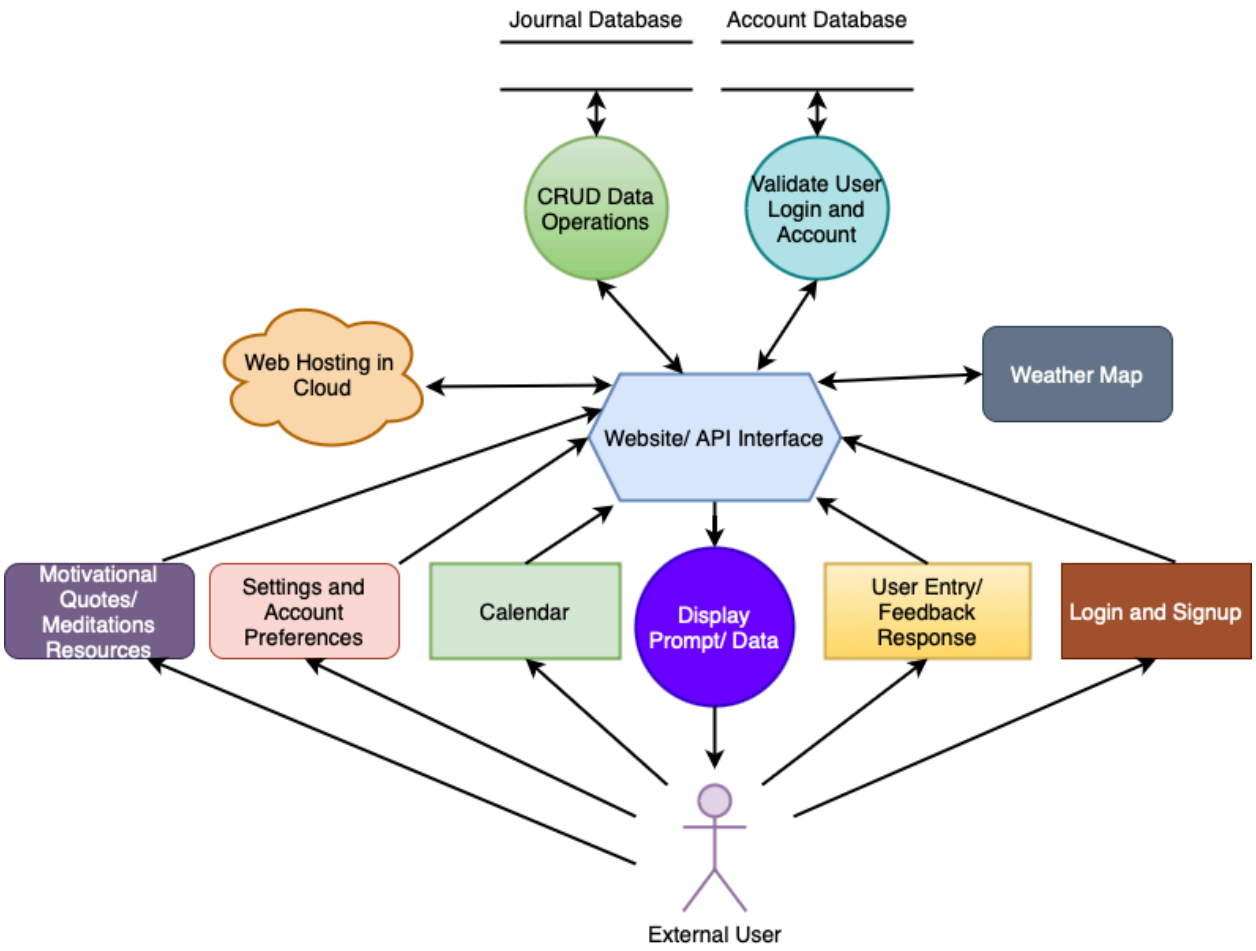
Responsibilities of Modules:

- Frontend: Rendering the user interface and handling user interactions.
- Backend: Processing user requests, interacting with the database, managing user authentication, and integrating with third party services.
- Database: Storing and retrieving user journal entries and account information.
- Third-Party Libraries: Providing additional functionalities such as weather data, socialprovider authentication, calendar apis.
- Infrastructure and Deployment: Managing the deployment environment using Docker containers. Docker involves packaging the application and its dependencies into a Docker container, then deploying the container onto a Docker-compatible host environment, providing consistent and isolated execution across different environments.
- Security and HTTPS: Ensuring secure communication and data protection.
- Testing: Implementing test cases and ensuring code quality.

Organization of Modules:

- Frontend interacts with the Backend through API interfaces for user authentication, journal operations, calendar updates, and settings/preferences.
- Backend communicates with the Database for CRUD operations on journal entries and user accounts.
- The system is organized in a layered architecture with clear separation of concerns. (Model View Template/Controller)

4.2 Development View



Level 1 Data Flow Diagrams (DFD)

Major Responsibilities of the System:

- **User Authentication and Management:** Responsible for user registration, login, and management of user accounts.
- **Journal Entry Management:** Handles the creation, retrieval, update, and deletion (CRUD) operations for user journal entries.
- **User Interface**: Renders the user interface for interacting with the system, including journal entry submission, calendar updates, and settings/preferences.
- **Database Management:** Manages the storage and retrieval of user data, including journal entries and user account information.

Modules and Subsystems:

1. **Authentication and User Management Module:** Responsible for handling user registration, authentication, and account management. This module interacts with the frontend for user login/signup operations and with the backend for validating user credentials and managing user accounts.
2. **Journal Management Module:** Handles CRUD operations for user journal entries. This module communicates with the backend to store/retrieve journal entries from the database and provides an interface for users to interact with their journals.
3. **User Interface Module:** Responsible for rendering the user interface and handling user interactions. This module utilizes HTML, CSS, and Bootstrap to create the frontend interface and communicates with the backend through API interfaces for user actions such as submitting journal entries or updating settings/preferences.
4. **Database Management Module:** Manages the storage and retrieval of user data using PostgreSQL with Railway. This module interacts with the backend to store/retrieve user journal entries and account information from the database.

Parallel Development and Deployment:

The system design allows for parallel deployment by not allowing key modules to be too dependent on each other. The functional modules at the highest level of the system as well as their subsystems are development in modular, self contained instances and only grow to interact with the other modules as the application develops. The modules, documentation, and source code are all shared using Git version control. Through Git and Github, the system will be shared amongst the involved parties and developed in smaller, local repositories. Once development on a module, subsystem, and process are tested and cleared for deployment, the new additions from the local repositories will be pushed to the main repository hosted on Github.

4.3 Process View

Component Collaboration

User Interface (UI): Responsible for rendering the user interface and handling user interactions.

Backend Services: Handles the business logic of the application, including user authentication, journal management.

Database Management: Manages the storage and retrieval of user data, including journal entries and user account information.

Concurrency, Communication, and Synchronization

Concurrency is managed through asynchronous communication between components, allowing the system to handle multiple user requests simultaneously. Synchronization mechanisms ensure data consistency and integrity, especially when accessing and updating shared resources in the database.

Handling User Requests and Task Execution

When a user interacts with the system through the UI, their requests are processed by the backend services. Authentication requests are first routed to the Authentication Service, which verifies user credentials and issues authentication tokens. Subsequent requests for journal management are processed by the respective services, which interact with the database to retrieve or store data as needed.

Key Processes, Threads, or Services

1. **Authentication Service:** Responsible for handling user authentication requests and issuing authentication tokens.
2. **Journal Service:** Manages CRUD operations for user journal entries, interacting with the database to store and retrieve data.
3. **Database Service:** Provides access to the database for storing and retrieving user data.
4. **Django All-Authentication**: Allows for google authentication, where the user will be able to log-in to the website using their google account.

Rationale for Decomposition:

Model-View-Controller (MVC) Design

MVC mostly relates to the UI / interaction layer of an application.The decomposition aims to balance between simplicity and scalability, allowing for easier maintenance and future enhancements while still meeting the system's requirements effectively. Additionally, the use of popular design patterns such as the Model-View-Controller (MVC) pattern in the backend services promotes code reusability and maintainability.

Our project uses this similar concept but slightly altered (MVT) where the views act similarly to the controller in most traditional MVC frameworks.

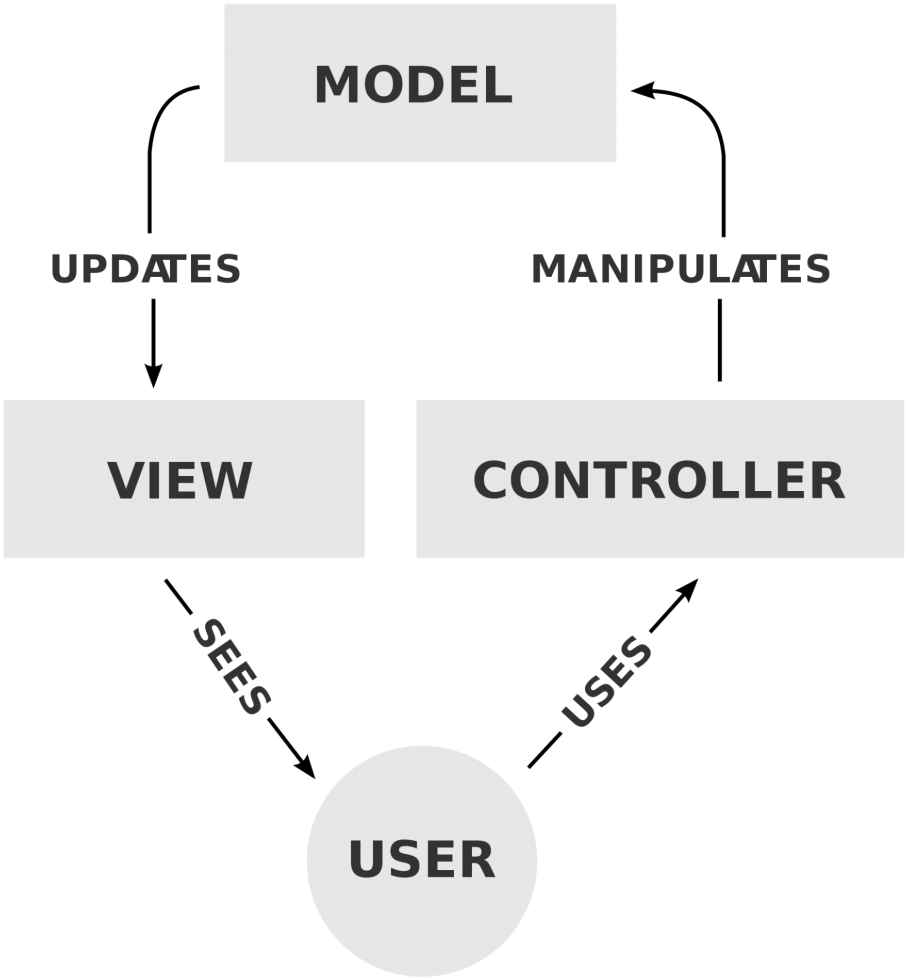
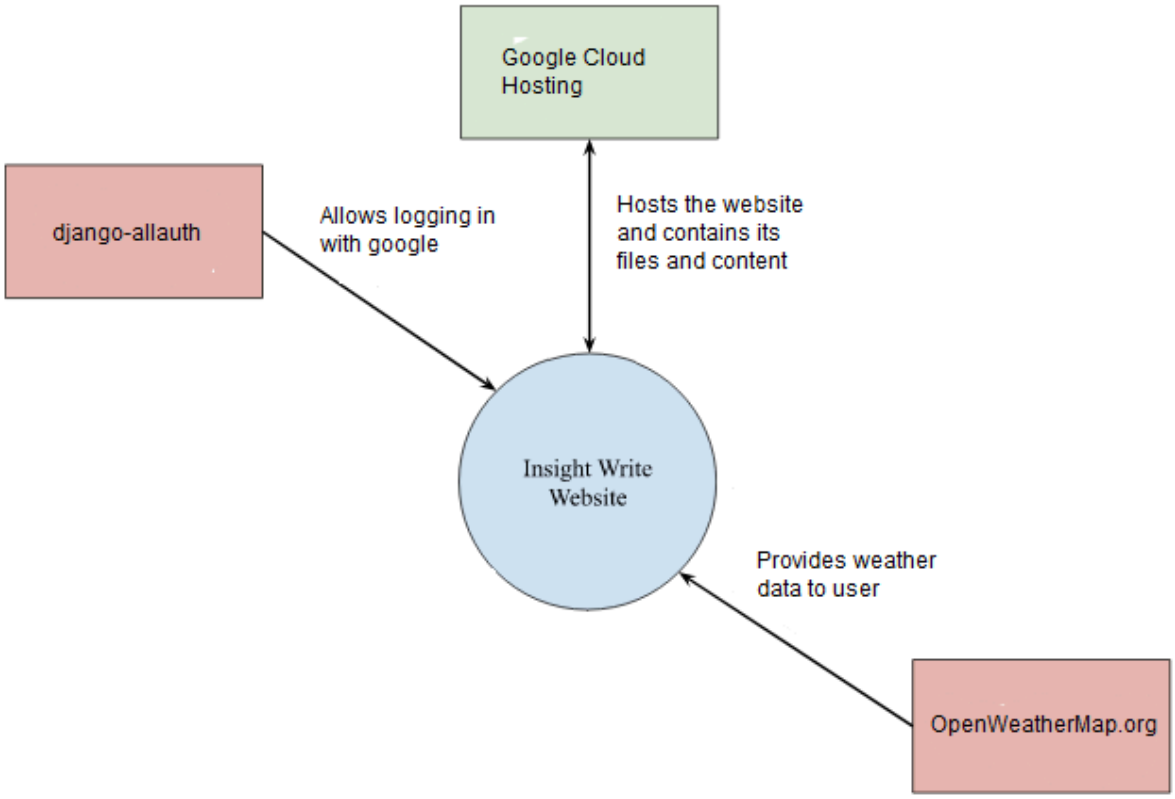


Diagram of interactions from a simple MVC pattern

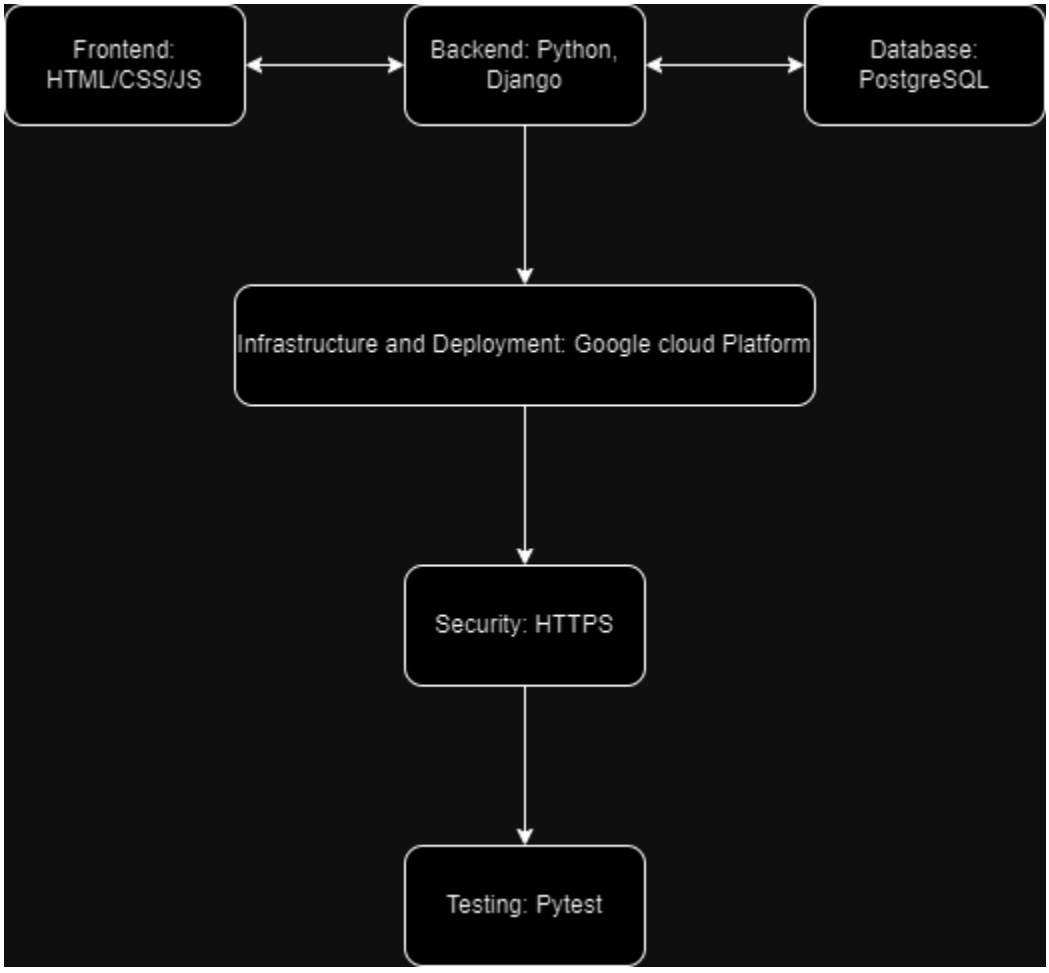
4.4 Physical view

Context Model (Hoang Le):


Google Cloud Hosting
django-allauth
Allows logging in with google
Hosts the website and contains its files and content
Insight Write Website
Provides weather data to user
OpenWeatherMap.org

5. Policies and Tactics (Niyusha, Keyvan, & Hoang)

5.1 Choice of which specific products used



Source: Will May created this diagram above

The coding platforms we will use are all listed in the diagram above which are utilized in the following products: Django Framework and Visual Studio Code. We went with these products because Django Framework allows for Python backend implementation along with frontend web-based programming and Visual Studio Code will enable us to use its compiler and interface to code. Lastly, our database is handled through PostGreSQL in Django Framework, and we can use third-party libraries from Python to generate data at will.

5.2 Plans for ensuring requirements traceability

To ensure that we track our requirements properly throughout the software development lifecycle, we will meet every sprint on Sunday,Monday, and Wednesday to check on each other for what stories we have completed and how they contribute to the project. In essence, we want to ensure that, throughout our sprints, we follow our GitHub roadmap to complete most of the features in each increment that add to the application. Ultimately, ensuring that we follow through and trace all requirements that must be completed throughout the application's life cycle.

5.3 Plans for testing the software

In this section, we will describe our detailed methods of testing approach to ensure the functionality, reliability and performance for our journal website. Our testing strategy is to test at every step, to check for compatibility while trying to maintain a safe security against cyber threats.

1. **Testing Methods:**
    - Unit Testing
    - Integration Testing
    - Regression Testing
    - Security Testing
    - UX/UI Testing
    - End-to-End Testing
2. **Testing Tools:** We will utilize Pytest for unit testing, beside that, our development team will work together for integration and system testing. Security testing will include user authentication and encryption(Https).
3. **Testing Environments:** We will set up regression testing, security testing, and accessibility testing for the development process.

5.3.1 Development trade-offs

During the development process, we will take into consideration several trade-offs in order to balance performance, functionality and development time. One of our decisions within the sprint was to organize a meeting to set-up our environment for everyone to start working at the same pace on coding instead of jumping right into the process. We also prioritize features such as journal entry management (create/edit/deletion), user registration (login, sign up), setting up backend functions for them to bring out a minimal viable webpage for further envisions in the process.

5.3.2 Coding guidelines and conventions

As mentioned above, the purpose of the meeting for everyone to set up docker and follow the django template on github is to help our team to maintain code consistency and readability across the project for all of our team members by simply deploying code files and projects into our group repository. Beside that, we could also be flexible on updating each other about new code documentations during our meeting in the week. This helps us work together efficiently and make sure that our codebase stays manageable and structured during the development process.

5.3.3 The protocol of one or more subsystems, modules, or subroutines

The architecture of our journal website is made up of a number of subroutines, modules, and subsystems that work together to provide functions for the features. We will use subsystem controls, CRUD operations for the journal entries, and have a user authentication module to handle user registration, login, and password management.

5.3.4 Architecture and Design Patterns

As mentioned in section 3, our team's strategy is to implement Django's Model-View-Template (MVT) architecture for dividing our code into models, views and templates to improve efficiency and scalability. Moveover, we will prioritize user privacy and security in our website with encryption and access controls algorithms to protect users' sensitive data in the journal entries.

5.3.5 Long-term Maintenance and Sustainability

For the website's long-term sustainability, we will try to have software maintenance with regular updates such as addressing security vulnerabilities, feature enhancements based on user feedback, and optimizations to improve the website speed.

5.3.6 User Interfaces and Source Code Organization

With intuitive interfaces such as user-friendly login and registration, improved journal entry management, and a responsive meditation section, the website will offer a dynamic experience for end-users' interaction.

5.3.7 Hierarchical organization of the source code into its physical components (files and directories).

Our team will work together to carefully organize source code into logical parts, directories, and files in a hierarchical structure. We will make sure to separate folders for both front-end and back-end code, which are then further subdivided based on its functionalities and feature sets. Additionally, after the environment setup meeting, we made sure that everyone was equipped with the necessary tools after cloning our GitHub repository to our personal devices for easier collaboration and individual project contributions.

5.3.8 Building and Generating Deliverables

According to the DFD level 0, our team will use various tools to construct Insight Write's deliverables, the back-end components will be built and packaged using Django Framework for functionality and maintainability. For optimization and interaction, we will have a front-end HTML/CSS/JS to work on and the compiler and packager for front-end resources will be decided by our team later-on.

5.3.9 Database Interface Abstraction

As suggested, we can implement an abstract genericDatabaseInterface class to make switching between different databases easier. Let's say we can implement CRUD operations as a generic interface for interacting with the database in order to make the system become independent from its database system. By doing this, we can easily change databases from MySQL to PostgreSQL with just updating the DatabaseInterface class.

6. Detailed System Design (Will, Jian)

Most components described in the System Architecture section will require a more detailed discussion. Each subsection of this section will refer to or contain a detailed description of a system software component. The discussion provided should cover the following software component attributes:

This is where Level 2 (or lower) DFD's will go. If there are any additional detailed component diagrams, models, user flow diagrams or flowcharts they may be included here.

6.1 User Interface (UI)

6.1.1 Responsibilities

Simplifies the steps that the user needs to do to interact with the website by consisting of visuals. These visuals accept user input and then the UI acts accordingly with the inputs.

6.1.2 Constraints

The type of input that the UI element is necessary for impacts the way it is visually represented. For example, if the UI element is meant to accept the user's input on a question that has only two answers (no option to not decide), then the UI element can be visually represented as an on/off switch. The UI element must be able to not only contain all possible inputs but also limit the amount of inputs down to those that cause no issues with the website. Back to the example, the on/off switch visual for the UI element has two possible inputs: on and off. It does not leave any possible user input for "half on, half off" or other possible inputs. Another constraint is on how much resources the website requires from one's hardware attempting to access it. We want this website to require as little resources as possible since it's meant to be very accessible.

6.1.3 Composition

The UI uses Cascading Style Sheets(CSS), Javascript, and Bootstrap. CSS and Javascript are front-end programming languages. Bootstrap is a front-end framework that uses CSS. Frameworks provide resources and tools that assist in web development.

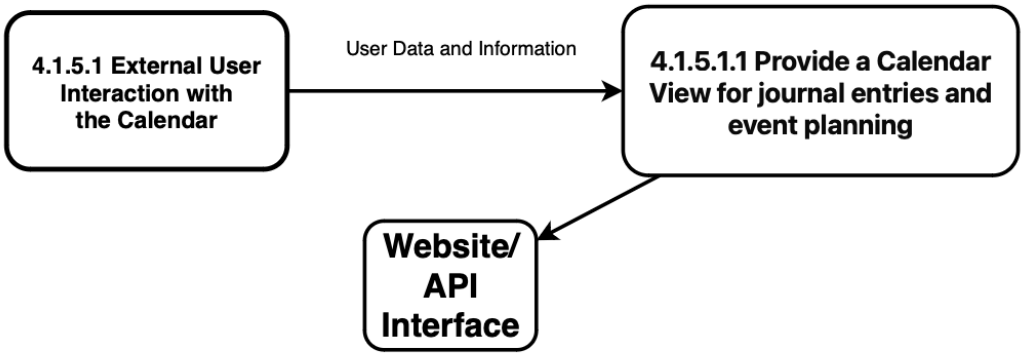
6.1.4 Uses/Interactions

While the UI does interact with the Database Management and Backend Services components, it is mainly the UI that sends requests to these components to accomplish tasks.

6.1.5 Resources

The UI requires memory, processors, and databases to work. Memory is needed for caching. A processor allows the website to perform faster. While parts of the website

could theoretically work without the databases, it would eventually hit a dead end in which accessing the database is required to proceed. Example: Calendar



6.1.6 Interface/Exports

Allows data to be interacted with using visuals that represent an input or a process. Most visuals are, in a way, a constant. They can be interacted with at any time on the webpage they are featured on. There are exceptions to this, however we do not plan on having buttons that are a one-time-only click.

6.2 Database Management

6.2.1 Responsibilities

Responsible for storing and accessing data. These main responsibilities are required for changes due to any kind of input to be made.

6.2.2 Constraints

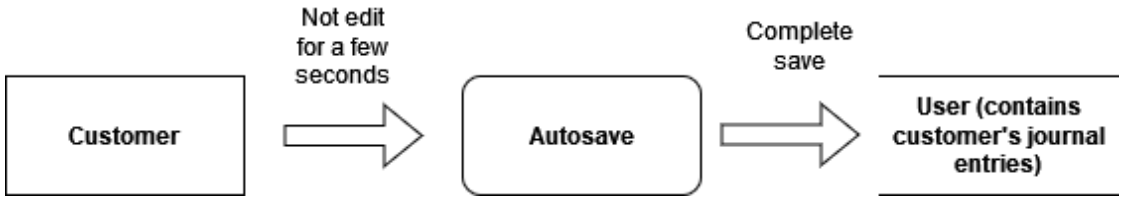
Our team's knowledge on SQL is limited compared to our knowledge in frontend and backend languages.

6.2.3 Composition

Uses SQL as the programming language. PostgreSQL is what the databases are based on and Railway is a deployment platform that will be connecting our PostgreSQL databases to the repository.

6.2.4 Uses/Interactions

Will constantly be called on by the Backend Services component to perform its responsibilities. Example: Autosave



6.2.5 Resources

Resources required include memory and processors. Memory is required for caching, which stores files in a temporary storage location in the browser so that they can be accessed more quickly.

6.2.6 Interface/Exports

Holds data of many data types for the backend to access at any time. We will mainly be using standard data types such as Strings, integers, and booleans. The classes do count as data types, however they will mainly be used to interact with the standard data types that inhabit the class.

6.3 Backend Services

6.3.1 Responsibilities

Contains and executes the logistics and instructions necessary to perform tasks that require the UI and the database. The two components working together with this component linking them is how the website can properly perform their functions.

6.3.2 Constraints

Our backend team has had less time to do anything due to the issues of finding a way to implement a database without any monetary costs.

6.3.3 Composition

Uses Python as the programming language. Also imports modules such as the datetime module for certain functions.

6.3.4 Uses/Interactions

Is called by the UI to perform functions. Most of these functions require accessing the database.

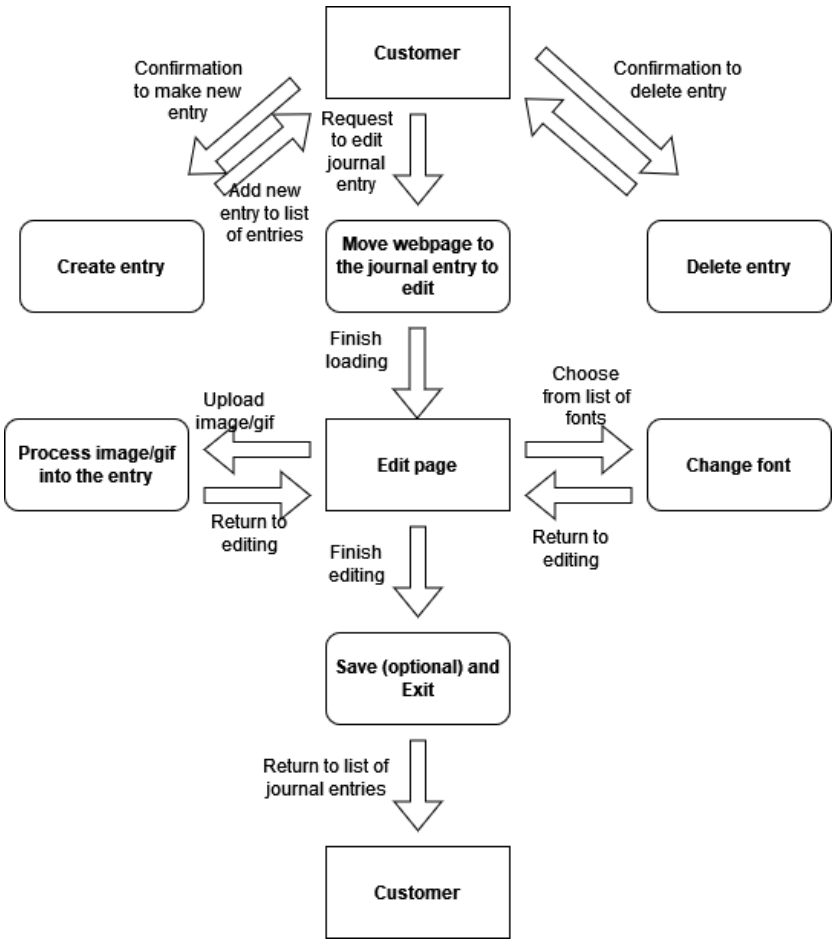
6.3.5 Resources

Resources required include the memory, processors, and databases.

6.3.6 Interface/Exports

Consists of all the class objects which allows the data taken from the database to be in a form that the backend can interact with and adjust accordingly. The backend services also

contain functions that, step-by-step, complete a task. There is a function for each task. Sometimes, the function only completes part of a task for another function so that it can complete it. Example: Create/Edit/Delete Journal Entries



6.4 Register/Login

6.4.1 Responsibilities

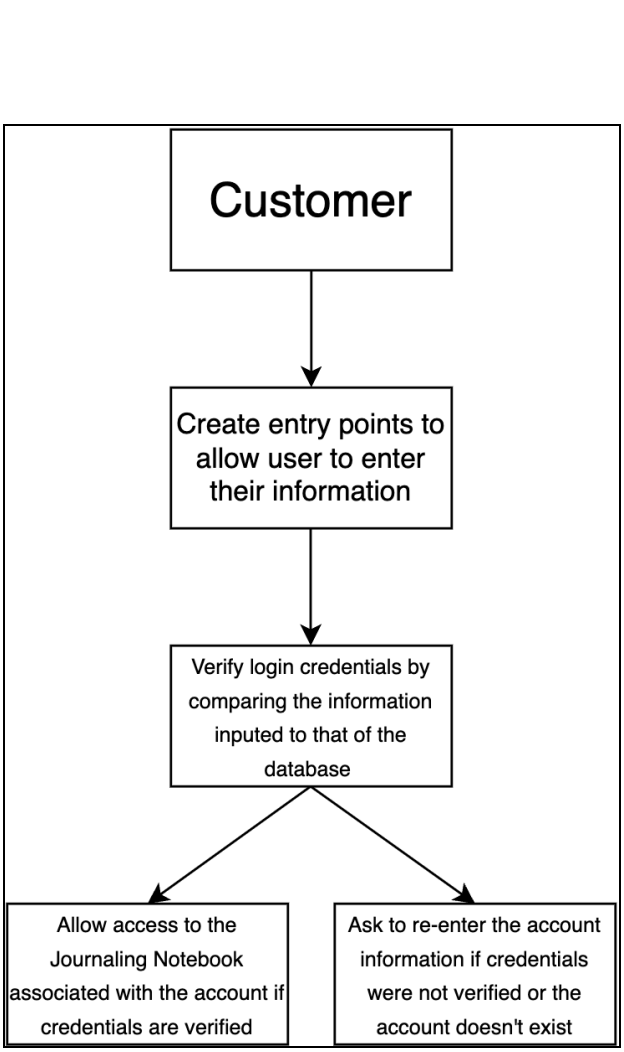
Logging in by creating an account with a username, email, and password is needed for the user to access the website. This allows users to have ownership of the journal entries they create by having an account that contains them all.

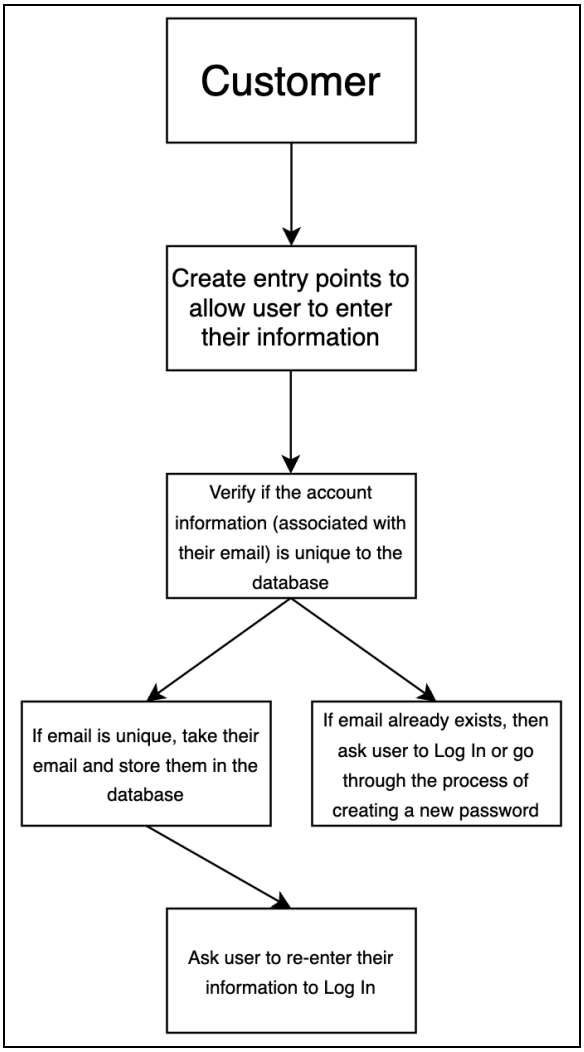
6.4.2 Constraints

There aren't many constraints, besides the user needing an email. It does require higher security on the user database however, since it contains users' private information. The security and reliability of the user database is necessary for this component to function.

6.4.3 Composition

Login and signup Level-2 DFDs.





6.4.4 Uses/Interactions

Logging in doesn't depend on other components, however other components greatly depend on logging in. Like mentioned before, one must login to access the website. If the user doesn't log in, all the other components of the website won't be usable.

6.4.5 Resources

Signing up and logging in requires interacting with the user database. The simplicity of the function makes it not very labor-intensive on the processors and memory. Race conditions would likely not occur since there shouldn't be any other function attempting to access the user database that is on the same webpage as the logging in feature. Same goes for signing up. The login and sign-up are on different webpages.

6.4.6 Interface/Exports

Many other functions require the data that the login/signup component creates in the user database. Without a primary key (data that distinguishes a group of data from other groups in a database), it becomes more difficult to define ownership. The variable that would likely see the most use is the user_ID (int), which serves as the primary key. There is also username (varchar), email (varchar), verifiedEmail (boolean), password (varchar), account_created_date (varchar), firstName (varchar), lastName (varchar), birthday (varchar), theme (boolean), notifications (boolean), analytics (boolean), meditation_tab (boolean).

6.5 Journal CRUD (Create, Read, Update, Delete)

6.5.1 Responsibilities

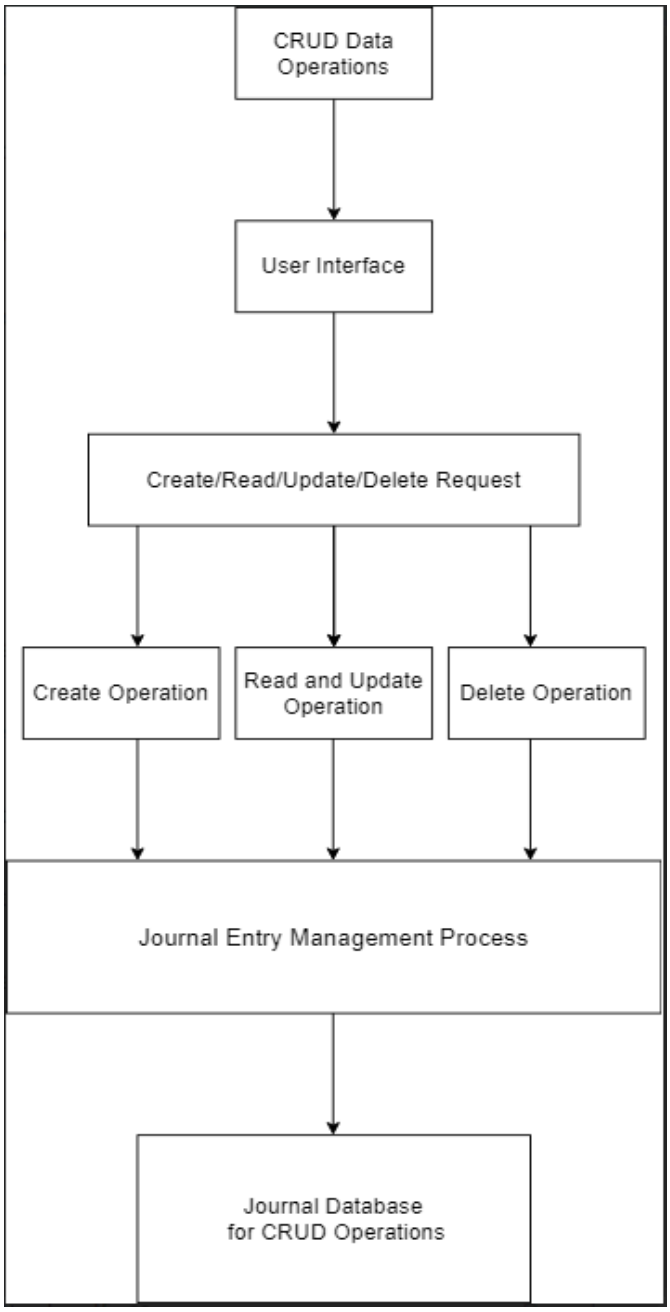
This component has the responsibility of everything related to journals: creating, reading, updating, and deleting them.

6.5.2 Constraints

The UI has to be built around these four functions. It has to be built around journal entries being able to be updated and/or deleted. There are also potential database storage issues if users create too many journal entries. The user must have an account to access Journal CRUD.

6.5.3 Composition

Level-2 DFD:



6.5.4 Uses/Interactions

Journal CRUD requires the Register/Login component to know who owns the journal entries that it is doing the CRUD functions for.

6.5.5 Resources

This component will be affecting the memory and databases the most. It will have the most frequent use and so, is expected to be changing the database very often. Race

conditions and deadlocks are more likely to occur due to constant changes to the journal database, however adjustments to how the primary key is registered should alleviate the chances of that happening.

6.5.6 Interface/Exports

The exports of this component don't impact other components. Its services such as the aforementioned CRUD are meant only to be used and interacted with by the component itself. Contains data types journal_id (int), user_id (int), tag_id (int), title (varchar), entry_date (varchar), last_modified_date (varchar), content (varchar), font (varchar), color (varchar)

6.6 Access Meditation Resources

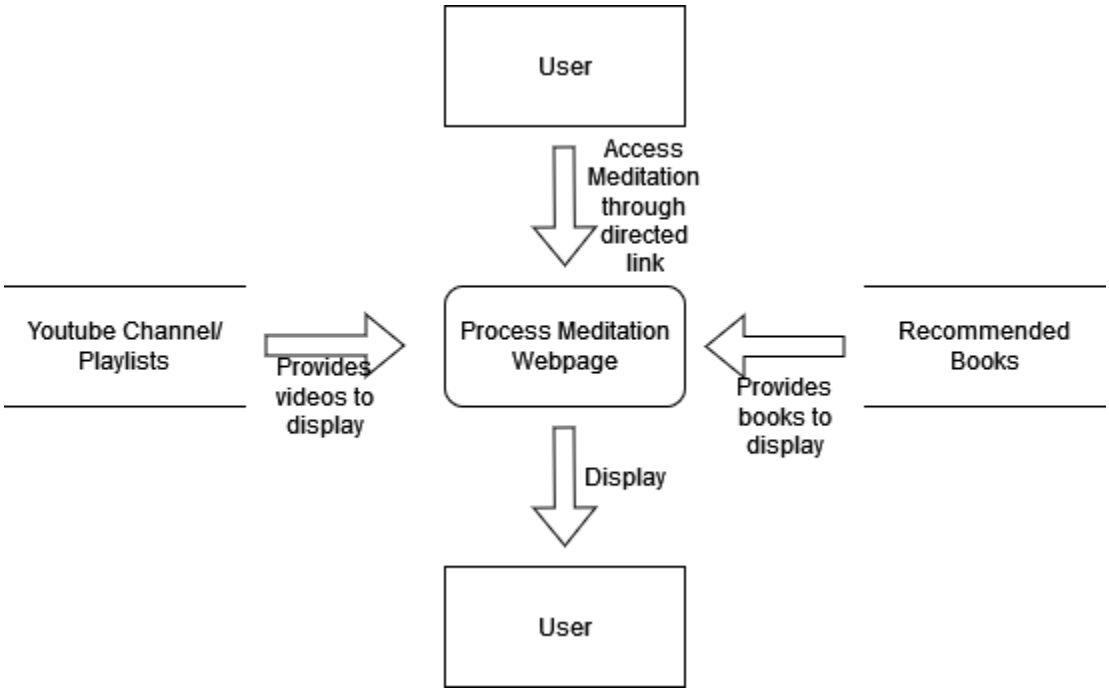
6.6.1 Responsibilities

Plays the role of assisting the user's mental health. This component allows the user to access the meditation webpage and make use of all its functions including video, audio, and book recommendations.

6.6.2 Constraints

Requires a bit of additional maintenance due to a need to refresh the video + audio + book lineup now and then.

6.6.3 Composition



6.6.4 Uses/Interactions

This component is self-contained so it does not interact with other components.

6.6.5 Resources

Does not require as big of a database as other components. Race conditions and/or deadlocks are to not be expected since the user is meant to view what is displayed rather than interact with it.

6.6.6 Interface/Exports

Mostly done frontend, due to lack of input needed from the user. Contains data types meditation_ID (int), user_ID (int), video_url (varchar), audio_url (varchar).

6.7 Access Local Weather Data

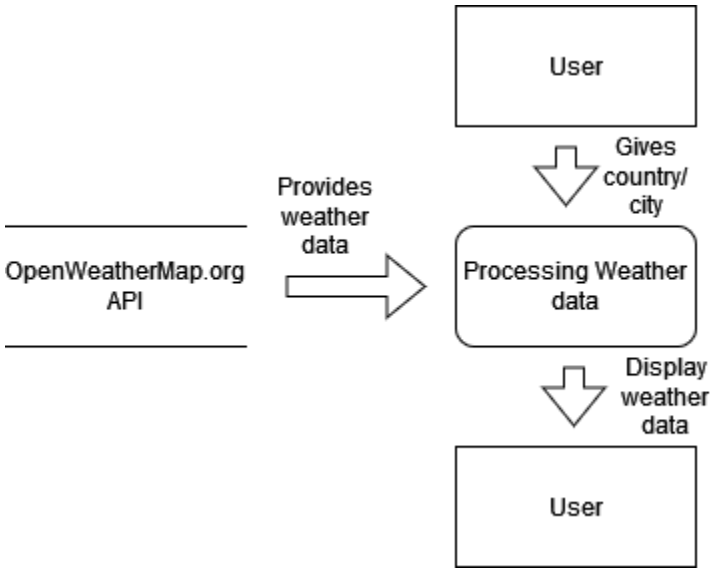
6.7.1 Responsibilities

Allows users to access the local weather data simply by typing in their location using an API called OpenWeatherMap.org

6.7.2 Constraints

It requires users to know their location, though that is likely to not be an issue. It assumes that the API is fully functional and has little bugs. It could be difficult to fix the API as the matter would be out of our hands since all we can do is call the API provider about the issue.

6.7.3 Composition



6.7.4 Uses/Interactions

This component is self-contained, so it doesn't interact with other components.

6.7.5 Resources

Requires little resources and does not interact with the database.

6.7.6 Interface/Exports

Does not contain any data types. Entirely done frontend.

6.8 Google Authentication

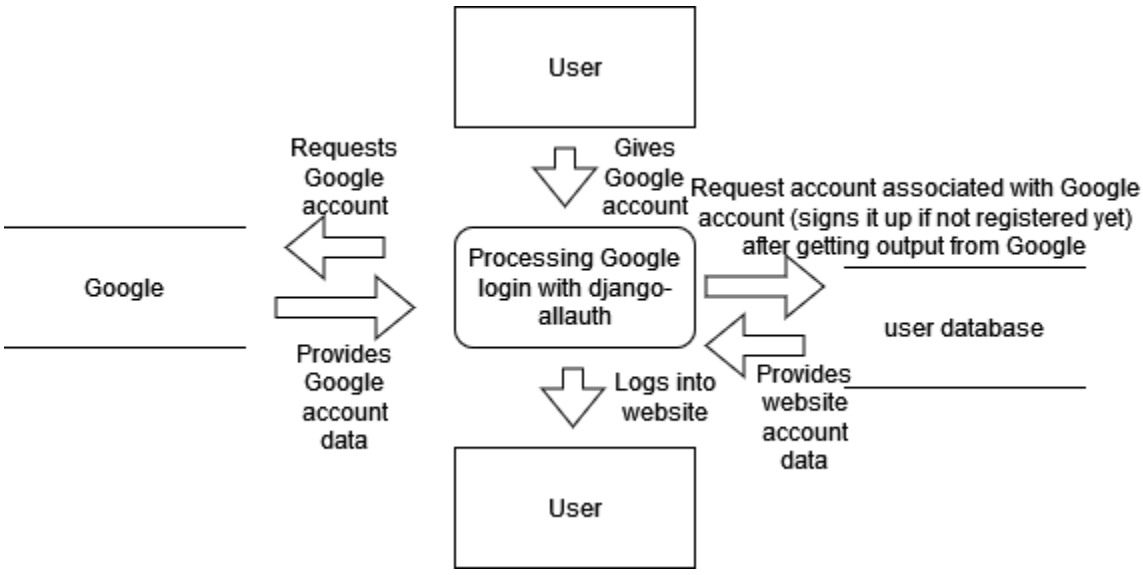
6.8.1 Responsibilities

Google authentication allows users to login to the website with a google account by using the django-allauth API. This component helps users who are not as used to tech to more easily access the website. It also is a quality–of-life feature for those who want to access many websites with just a single account.

6.8.2 Constraints

Using a google account instead of signing up on the website creates some odd interactions with the database, however that is the extent of the constraints since this feature is optional.

6.8.3 Composition



6.8.4 Uses/Interactions

Interacts with most other components, just like the register/login component for the same reason: logging in is necessary to interact with them.

6.8.5 Resources

Uses the user database, but interacts with it in a special way compared to the usual way of signing up and logging in.

6.8.6 Interface/Exports

Consists of the same data types as the register/login component. Is able to more-or-less fulfill the same functions as the register/login component, with the only difference being what the user has to provide to do it. Many other functions require the data that the login/signup component creates in the user database. Without a primary key (data that distinguishes a group of data from other groups in a database), it becomes more difficult to define ownership.

6.x Name of Component (Module)

6.x.1 Responsibilities

The primary responsibilities and/or behavior of this component. What does this component accomplish? What roles does it play? What kinds of services does it provide to its clients? For some components, this may need to refer back to the requirements specification.

6.x.2 Constraints

Any relevant assumptions, limitations, or constraints for this component. This should include constraints on timing, storage, or component state, and might include rules for interacting with this component (encompassing preconditions, post conditions, invariants, other constraints on input or output values and local or global values, data formats and data access, synchronization, exceptions, etc.)

6.x.3 Composition

A description of the use and meaning of the subcomponents that are a part of this component.

6.x.4 Uses/Interactions

A description of this components collaborations with other components. What other components is this entity used by? What other components does this entity use (this would include any side-effects this entity might have on other parts of the system)? This concerns the method of interaction as well as the interaction itself. Object-oriented designs should include a description of any known or anticipated subclasses, superclass's, and metaclasses.

6.x.5 Resources

A description of any and all resources that are managed, affected, or needed by this entity. Resources are entities external to the design such as memory, processors, printers,

databases, or a software library. This should include a discussion of any possible race conditions and/or deadlock situations, and how they might be resolved.

6.x.6 Interface/Exports

The set of services (classes, resources, data, types, constants, subroutines, and exceptions) that are provided by this component. The precise definition or declaration of each such element should be present, along with comments or annotations describing the meanings of values, parameters, etc. For each service element described, include (or provide a reference) in its discussion a description of its important software component attributes (Classification, Definition, Responsibilities, Constraints, Composition, Uses, Resources, Processing, and Interface).

Much of the information that appears in this section is not necessarily expected to be kept separate from the source code. In fact, much of the information can be gleaned from the source itself (especially if it is adequately commented). This section should not copy or reproduce information that can be easily obtained from reading the source code (this would be an unwanted and unnecessary duplication of effort and would be very difficult to keep up-to-date). It is recommended that most of this information be contained in the source (with appropriate comments for each component, subsystem, module, and subroutine). Hence, it is expected that this section will largely consist of references to or excerpts of annotated diagrams and source code.

7. Detailed Lower level Component Design (Jian, Keyvan)

Other lower-level Classes, components, subcomponents, and assorted support files are to be described here. You should cover the reason that each class exists (i.e. its role in its package; for complex cases, refer to a detailed component view.) Use numbered subsections below (i.e. "7.1.3 The ABC Package".) Note that there isn't necessarily a one-to-one correspondence between packages and components.

Note that, the classes below reflect how each Django App currently made (insightwrite, journals, login_signup, meditations, & settings) in our project works. Meaning, each app have the following folders & classes that give its functionality.

Given folders and classes that make up a Django App
Django folders: migrations, static, templates
Django files: admin.py, models.py, urls.py, views.py

7.1 Django App Framework

7.1.1 Classification
The Django folders and files listed constitute the foundational components of a Django web application, serving various purposes within the app's architecture.

7.1.2 Processing Narrative (PSPEC)
The processing narrative involves utilizing Django's built-in functionalities to handle web requests and responses efficiently. Each component plays a vital role in this narrative, contributing to execute of the application's logic and presentation layers.

7.1.3 Interface Description
These components collectively provide interfaces for managing database models, handling URL routing, rendering views, and managing static files such as CSS, JavaScript, and images. They encapsulate the core functionalities required for developing dynamic web applications using Django.

7.1.4 Processing Detail

7.1.4.1 Design Class Hierarchy
models.py: Defines the data models for the application, representing the structure and behavior of the database entities.

admin.py: Registers the application's models with the Django admin interface, enabling administrators to manage database records through a user-friendly interface.

urls.py: Specifies URL patterns and their corresponding view functions, facilitating the mapping of incoming requests to appropriate handlers.

views.py: Implements the view logic for processing requests and generating responses, typically rendering HTML templates and interacting with models as needed.

migrations/: Contains database migration files generated by Django's migration framework, enabling schema changes to be propagated across different environments consistently.

static/: Stores static files such as CSS, JavaScript, and images, which are served directly to clients by the web server.

templates/: Houses HTML templates used for rendering dynamic content, allowing for the separation of presentation and business logic.

7.1.4.2 Restrictions/Limitations
The components listed do not inherently impose specific restrictions or limitations but adhere to Django's conventions and best practices for web development.

7.1.4.3 Performance Issues
Performance considerations may arise based on factors such as database query optimization, caching strategies, and efficient template rendering. However, the components themselves do not introduce significant performance issues when used appropriately.

7.1.4.4 Design Constraints
The design constraints are primarily dictated by Django's architecture and conventions, promoting modularity, reusability, and maintainability of code.

7.1.4.5 Processing Detail For Each Operation
models.py: Handles database operations such as creating, querying, updating, and deleting records using Django's ORM (Object-Relational Mapping) features.

admin.py: Facilitates CRUD (Create, Read, Update, Delete) operations on database records via the Django admin interface.

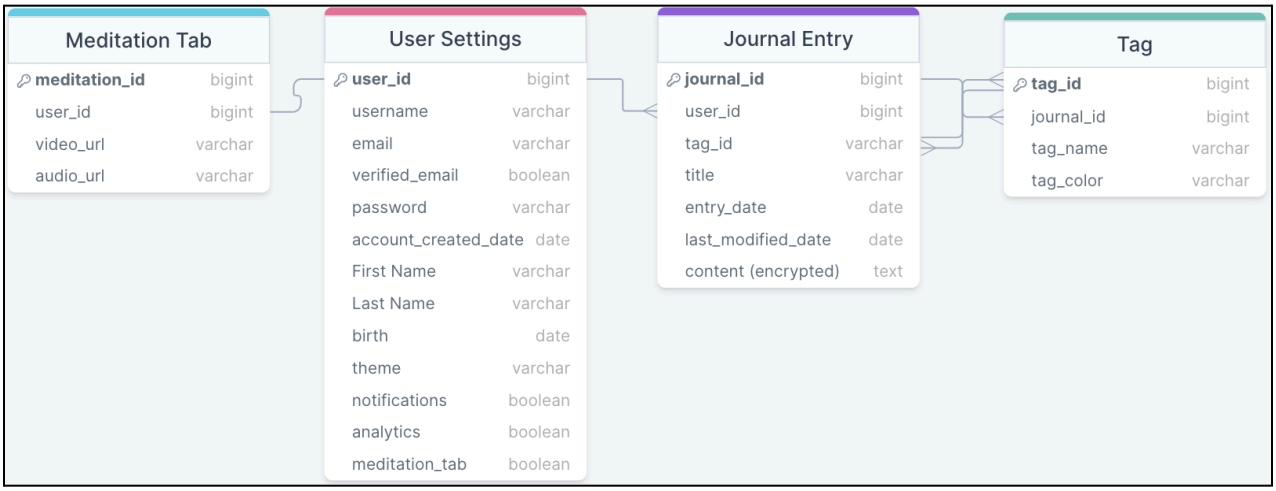
urls.py: Defines URL patterns and corresponding view functions, routing incoming requests to the appropriate handlers based on the requested URL paths.

views.py: Processes incoming requests, interacts with models and other resources as needed, and generates HTTP responses typically by rendering HTML templates with dynamic data.

migrations/: Manages database schema changes by generating and applying migration scripts to ensure database consistency across different environments.

static/: Serves static files directly to clients upon request, reducing server load and improving page load times.
templates/: Renders HTML templates with dynamic content, enabling the presentation of data retrieved from the database or processed by view functions.

8. Database Design (Keyvan)



9. User Interface (Hoang, Keyvan)

9.1 Overview of User Interface

- In order to fully interact with the website, the user will be required to have an internet connection, a device with a browser installed, and either a touchscreen with a digital keyboard or a touchpad/mouse and hardware keyboard.
- As a web application, the UI will mainly be driven with forms, buttons, links, and interactive parts of the webpage.
- The user will be able to create an account, login, access the journaling dashboard, and submit journal entries to be analyzed by the Natural Language Processor API and get feedback.

9.2 UX Standards

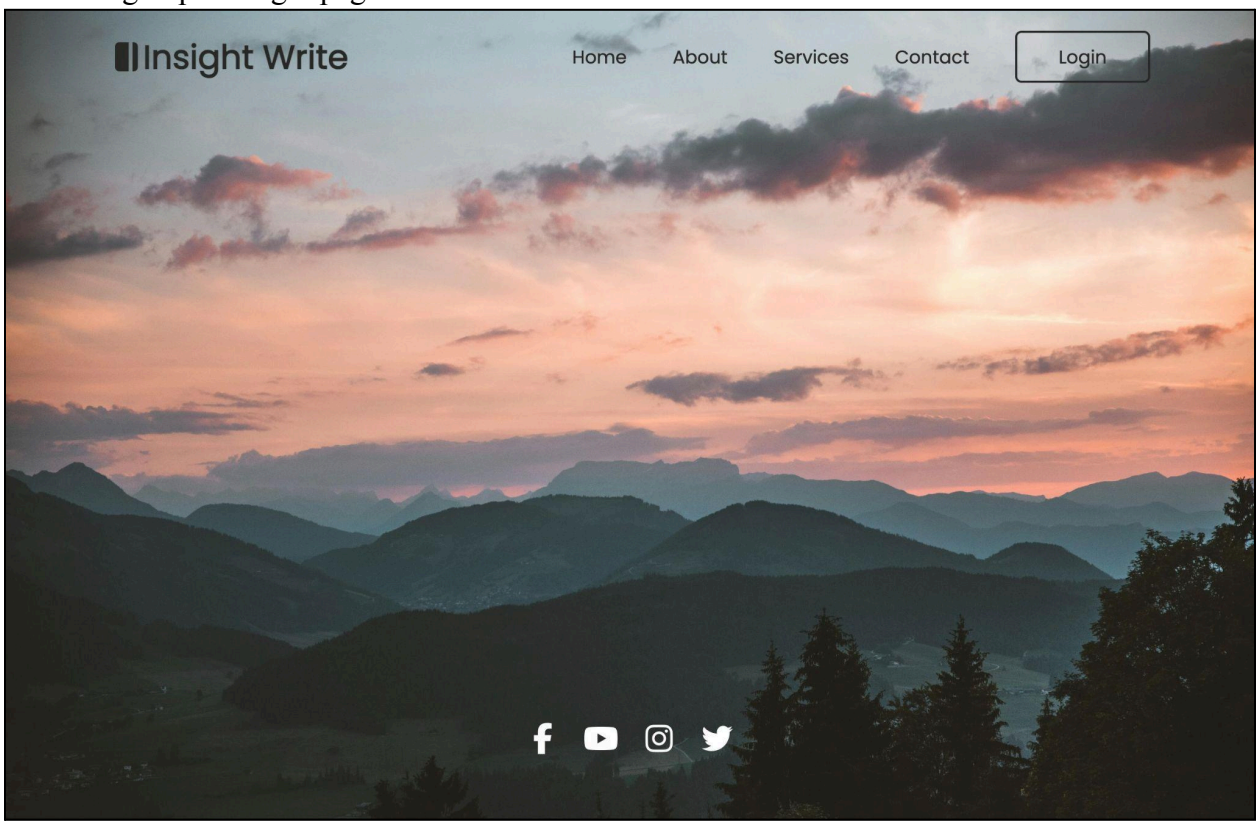
- 1. Content Reigns Supreme
    - Content is original, not plagiarized
    - Content provides genuine value to users' inquiries
    - Content is explicit and attractive to the eye
    - Easy to read and composed in a simple language
    - Appropriate structures (headings, paragraphs, other elements)
    - Use effective visual content
- 2. Homepage Design
    - Simple, comprehensive design
    - Use high-res photos/videos
    - Include links to each subpage
    - Updates/changes are visible on the home page
    - Provide a link to terms and conditions and privacy page
        - Terms and Conditions:
        - Privacy Page:
    - Page loads quickly and as intended
- 3. Website Layout
    - Responsive web design (for different screen resolutions: mobile, tablet, desktop)
    - Important information appears first
    - All content is displayed in suitable resolutions
    - Include links to relevant pages
- 4. Website Navigation
    - Primary navigation is simple to locate and comprehend (like a nav bar)
    - Home page has a link to the logo
    - Users can quickly determine where they are on a website
    - Optimize the structure of URLs
    - Every significant link is visible
    - Navigational labels are clear and present
    - On each page, include a clear call to action
        - Let the user know what part of the website they are on

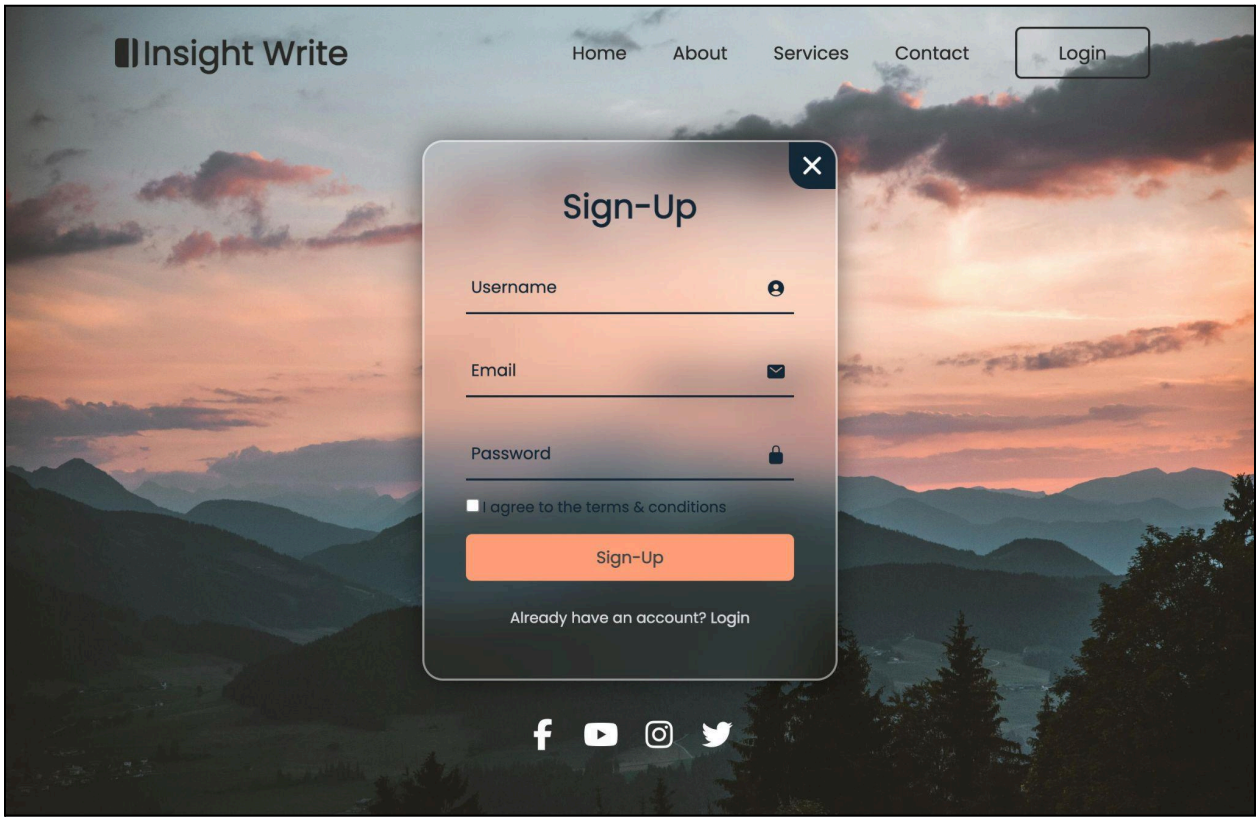
- 5. Functionality over Everything
    - Test page functionality
        - Pytest
- 6. Automated Visual Testing
    - Either manual tests or utilizing one of many visual testers:
        - TestGrid (end-to-end visual regression testing for mobile/websites)
        - LambdaTest (image-to-image comparison)
        - Percy (automated visual review platform)
- 7. Website Accessibility Testing
    - Must be ADA (Americans with Disabilities Act) compliant
        - Use alternative text on visual assets
        - Add contrasting colors
        - Accommodate keyboard navigation
        - Resize text to be viewable
        - Interactive elements should be operable
        - Effective headlines and descriptions
        - Subtitles/captions or transcript to videos
        - Avoid flashing lights or blinking bright elements
    - Automated accessibility tester:
        - Axe
- 8. Performance & Scalability
    - Page load-up time averages at 2 seconds and should not exceed much longer.
    - Can handle traffic load appropriately for smooth and seamless performance.

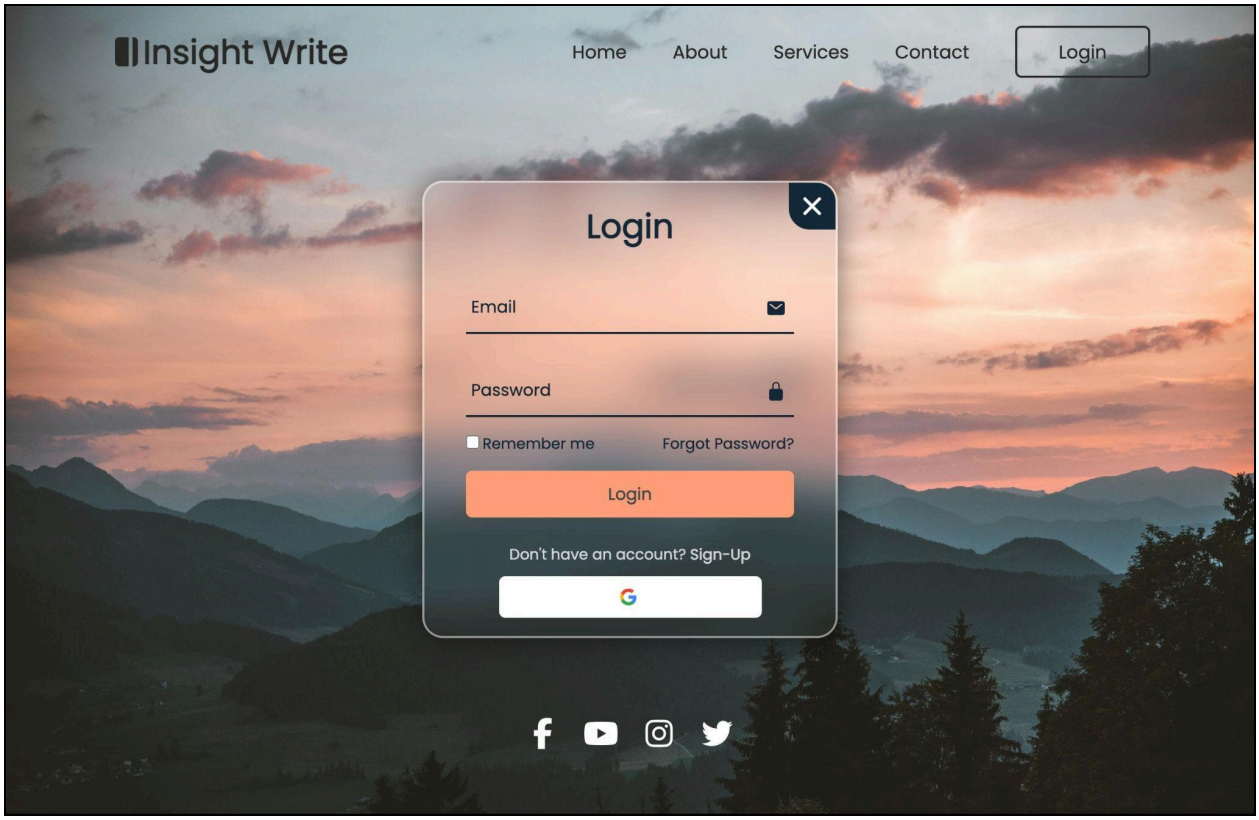
9.3 Screen Frameworks or Images

- Web pages displayed below:

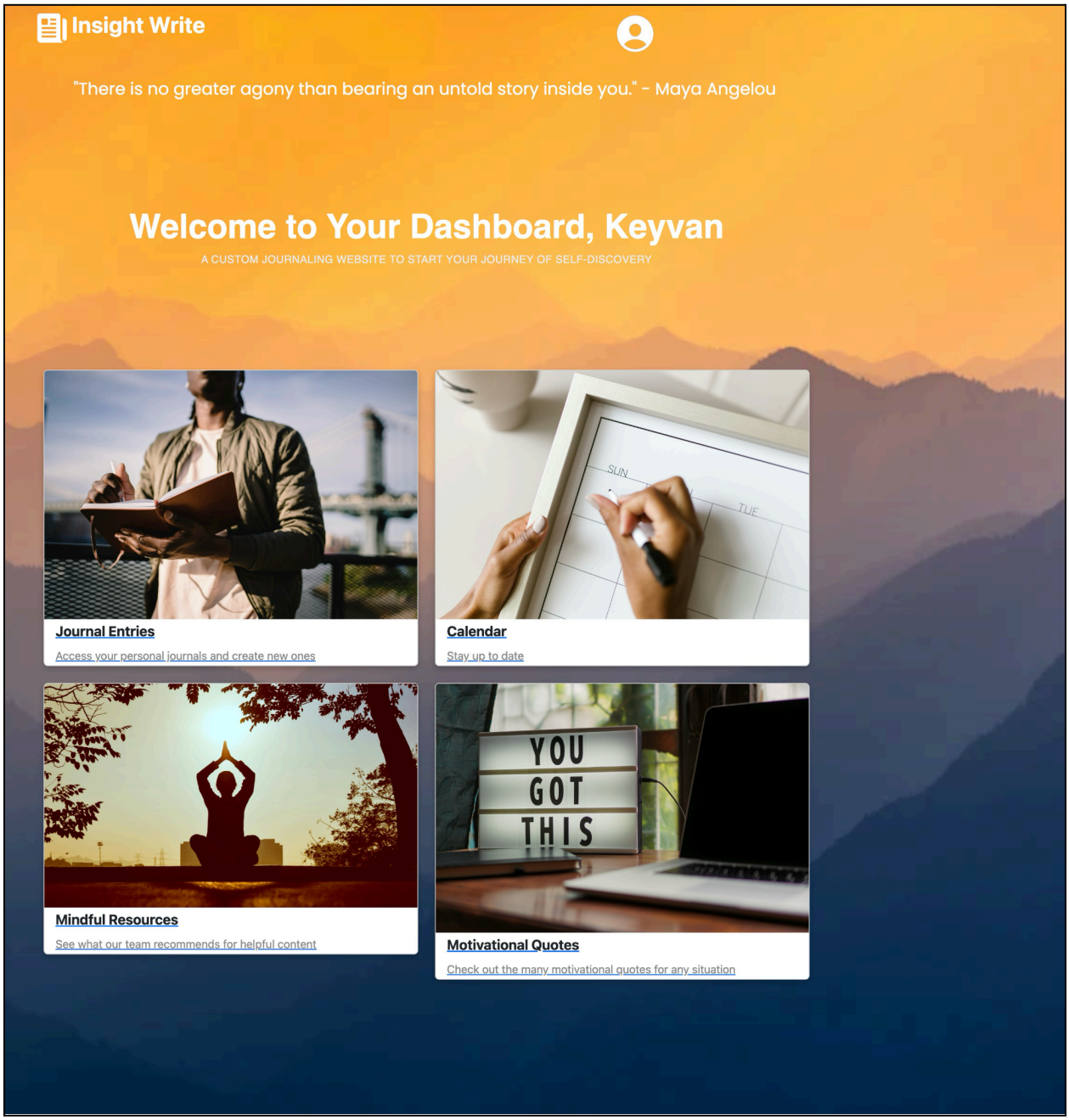
- Sign up & Login page



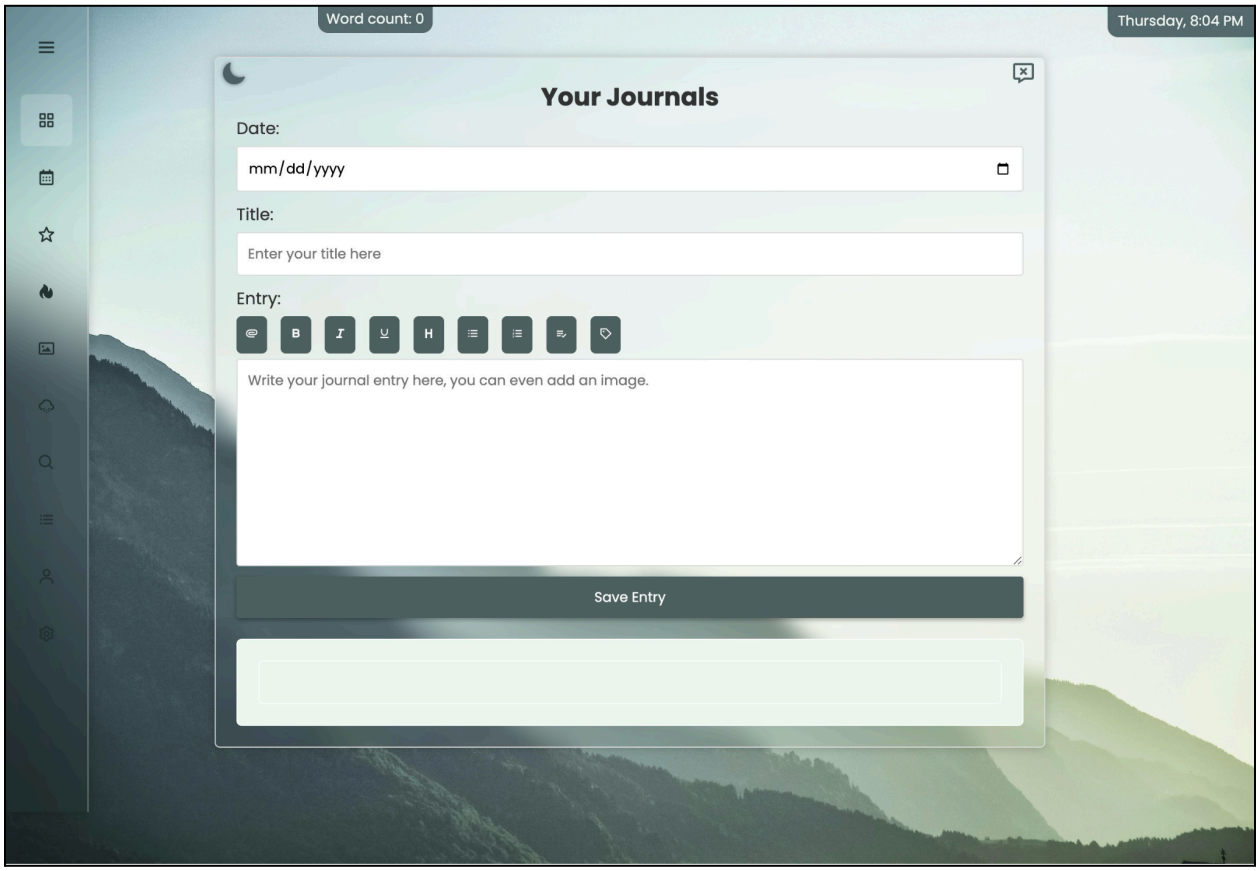



Insight Write
Home
About
Services
Contact
Login
Login
Email
Password
Remember me
Forgot Password?
Login
Don't have an account? Sign-Up

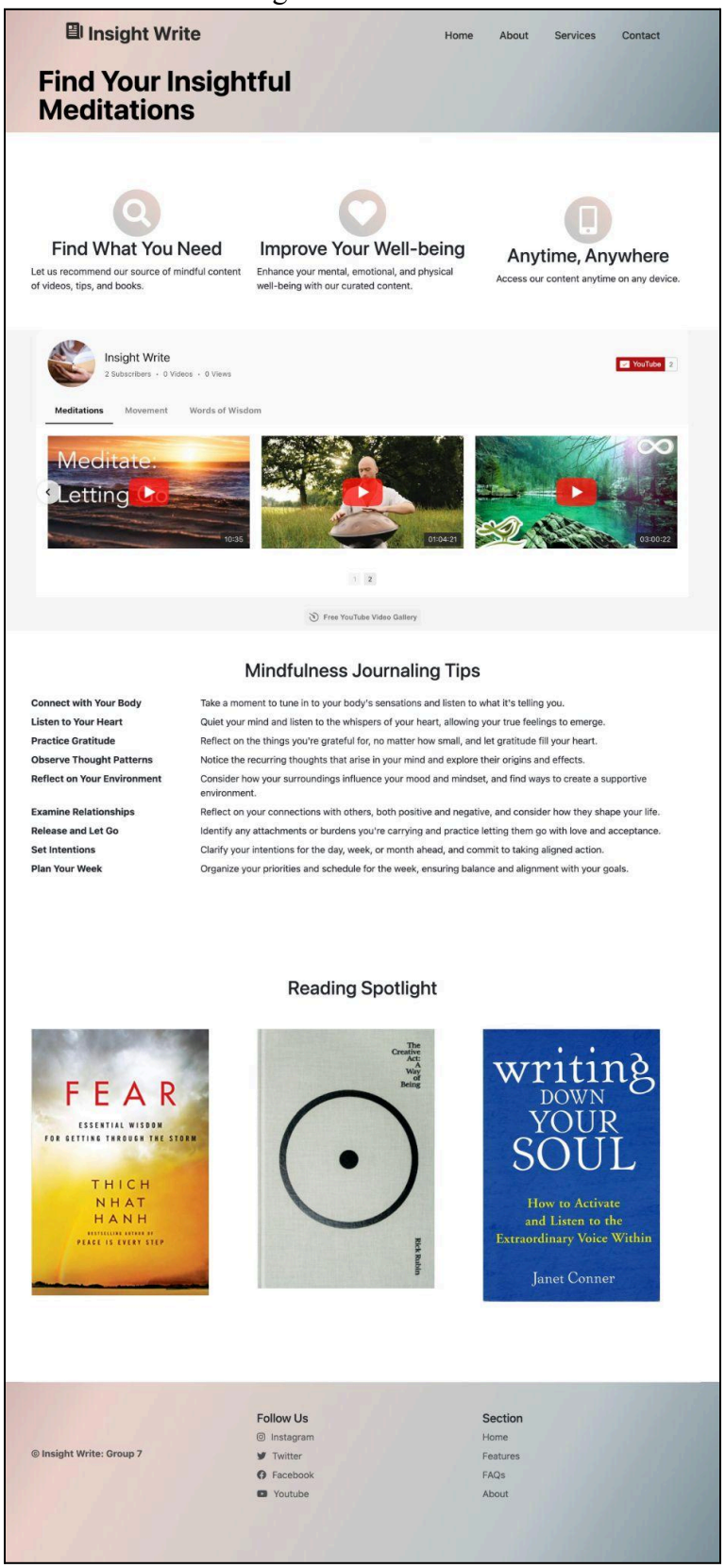
- Journals Page (Dashboard)



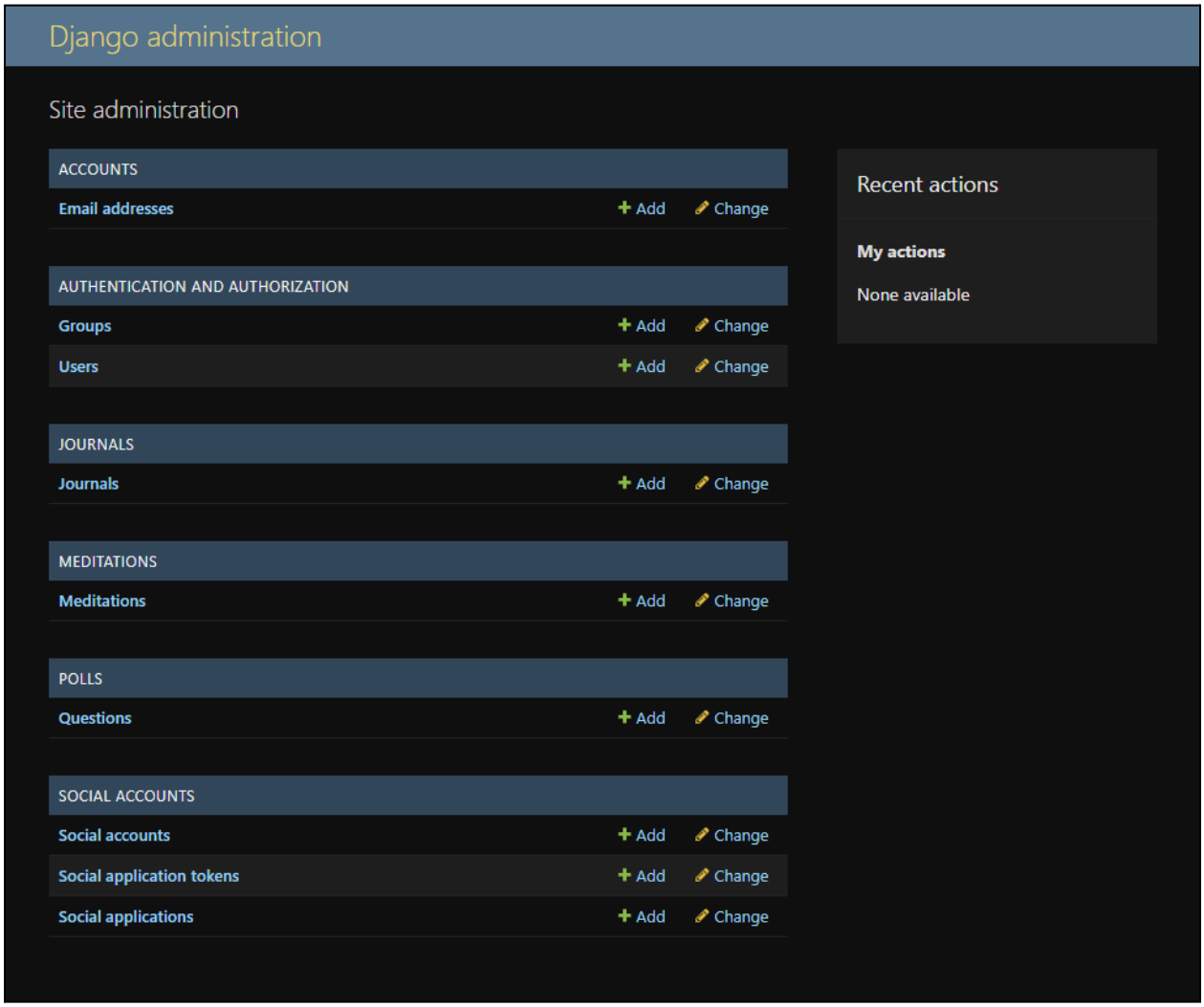
- Journal Entries Page



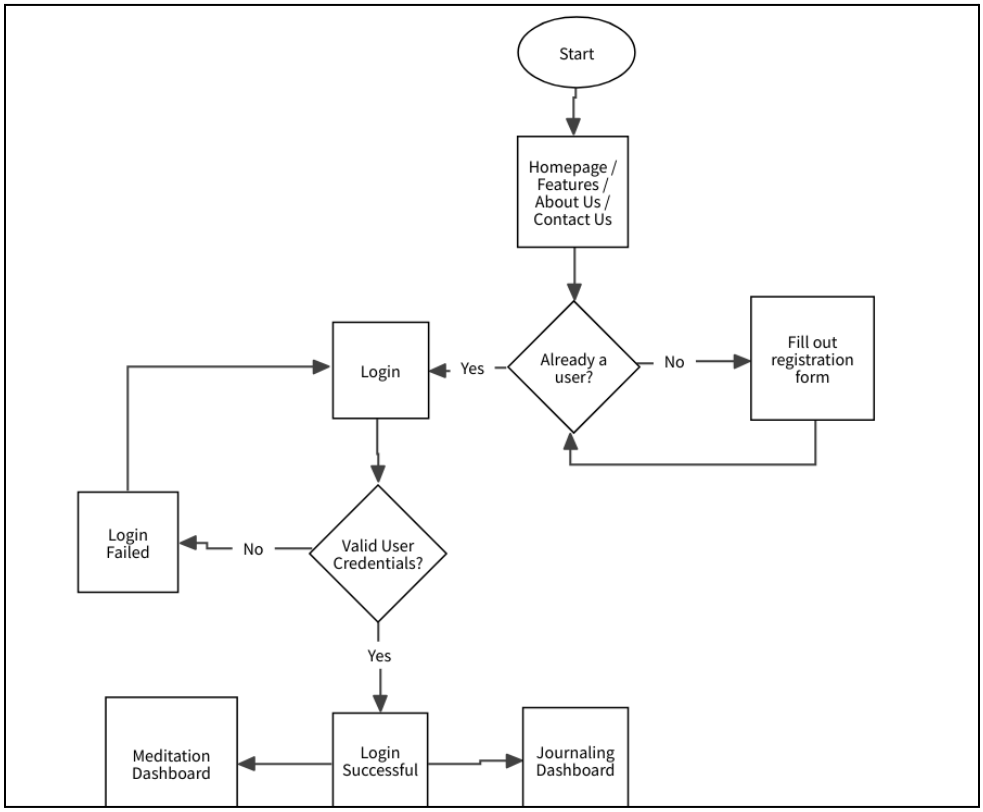
- Meditations Page



- Django Admin Page



9.4 User Interface Flow Model



10. Requirements Validation and Verification (Miguel, Niyusha)

Requirement	Element / Component	Testing Methodology
Can login	Sign Up, Log In, Validate User Login	Test log in using data already in the database. Then, test sign up by attempting to create an account on the login page.
Can interact with journal entries and save edits to the entries in them	Display User Prompt, CRUD Data Operations, User Journal Entry, Create/Edit/Delete entries, Autosave	Have a test case of journal entries in the database. Attempt to edit and/or delete them. Then, attempt to create a journal entry and try to edit and delete it.

11. Glossary - (Tony, Niyusha)

-**ADA (Americans with Disabilities Act):** A civil rights law that outlawed discrimination based on one's disability. Meant to aid the disabled in situations where their disability affects their ability to experience something.

- **ADA (Americans with Disabilities Act):** A civil rights law that outlawed discrimination based on one's disability. Meant to aid the disabled in situations where their disability affects their ability to experience something.
- **API (Application Programming Interface):** Referring to how multiple computer programs or components can communicate with each other.
- **Bootstrap:** A free and open-source front-end framework for designing websites and web applications. It contains HTML- and CSS-based design templates for typography, forms, buttons, navigation and other interface components, as well as optional JavaScript extensions.
- **CRUD Operations:** CRUD stands for Create, Read, Update, Delete. It refers to functions used for the journal management system.
- **Django Framework:** A free and open-sourced Python-based web framework. Used for web development.
- **Docker:** Runs containers, which simplifies the setup of the project such as eliminating many problems that might come with coding alongside different operating systems.
- **JavaScript:** A high-level, dynamic, untyped, and interpreted programming language JavaScript enables interactive web pages and is an essential part of web applications.
- **HTTP (Hypertext Transfer Protocol):** Refers to a process for loading webpages with hypertext links. It is what allows users to connect to servers.
- **MVT Architecture:** Stands for Model-View-Template, which is a software architectural pattern used in web development where the model represents the data structure, the view handles user interface logic, and the template manages the presentation layer.
- **ORM (Object Relational Mapping):** A programming technique for converting data between incompatible types of systems that use object-oriented programming languages. In this case, we use Django to allow our team members to interact with the database using Python objects.
- **PostgreSQL:** An open source data management system.
- **Pytest:** A testing framework for Python that simplifies writing and executing unit testings.
- **Scrum:** An agile management framework.
- **Sprint:** Part of a scrum. In this case, it refers to one week of work on the project. However, the length of a sprint is up to what the team decides.
- **UI (User Interface):** It includes graphical elements such as buttons, menus, forms… for users to interact within the website application.

- **UX (User Experience):** The overall experience of the user when using the website.

12. References (Niyusha)

SRD:
https://docs.google.com/document/d/1BSysiPqd0mos6ig-L6apamdcb_qzO4uQ--NmEtf-37A/edit

STP:
https://docs.google.com/document/d/1NDO9_zvlWWT_CTjCZ3LceB8EiaLNJ6mnGZUBfWgVEI4/edit

Django Documentation:
https://docs.djangoproject.com/en/5.0/

Django Docker Template
https://github.com/amerkurev/django-docker-template/

Bootstrap Documentation:
https://getbootstrap.com/docs/5.3/getting-started/introduction/

Brad Appleton <brad@bradapp.net> http://www.bradapp.net
https://www.cs.purdue.edu/homes/cs307/ExampleDocs/DesignTemplate_Fall08.doc