OurVoice

PROJECT DOCUMENTATION

INTRODUCTION

The purpose of OurVoice is to create a supportive digital journaling platform for women and non-binary individuals, seeking a safe and empowering online space. This report will explore the specifications, design, implementation and execution of our application, as well as critical analysis into our testing, challenges, and future improvements.

BACKGROUND

Problem Space & Motivation, System Purpose

Most existing platforms are overly social, or performance-driven, discouraging users from sharing sensitive experiences. OurVoice was designed to bridge this gap by creating a supportive journaling and wellbeing environment. The system provides a private journal experience, optional public sharing, keyword detection, support, wellness and career hub, sentiment analysis, personalised affirmations, hashtag generation and like-functionality.

Application Overview

The system operates as a terminal-driven client app connected to a Flask server and SQL database. The server processes all data storage, analysis and routing, while the client is presented with a text-based interface.

SPECIFICATIONS & DESIGN

Functional requirements

1. **User Identification:** users enter a username at the start of the application
2. **Create Public or Private Posts:** users create posts with a title, content, visibility setting and optional hashtags which are then submitted via the /feed POST route and stored in the database
3. **Viewing Posts:** public posts are retrieved via /feed GET route
4. **Like Functionality:** users can like a public post once only to prevent duplicate likes
5. **Keyword Detection:** the system analyses journal entries for sensitive or wellbeing-related keywords and informs user of directed support
6. **Support Hub Access:** provide the user with access to a structured set of wellbeing and support resources through the /support_hub route
7. **Wellness and Career Resources:** provide access to a second resource hub containing wellness, career development and learning materials via the /wellness_career route
8. **Personalised Affirmations:** analyse the sentiment of the user's post using a sentiment analysis library, fetch and display an affirmation from an external API based on the sentiment category

Non-functional requirements

1. **Usability:** clear menus, intuitive navigation and informative error messages e.g. when an invalid post ID is entered
2. **Reliability:** stable handling of API requests and database failures, prevents duplicate likes by checking existing entries before updating the database
3. **Performance:** efficient database queries and responsive terminal interaction

4. **Security:** safe SQL queries, protected API keys and strict separation of private content to prevent harmful inputs from breaking the DB
5. **Maintainability:** modular architecture with single-responsibility components, supporting the project requirement for clean organisation and effective code decomposition
6. **Portability:** compatibility across macOS and Windows, Flask server operates on port 5001, avoiding conflicts that can occur on some operating systems
7. **Scalability:** support for future feature expansion, such as adding more resource categories, improving text analysis features or introducing new interfaces
8. **Accessibility:** simple, readable text-based interface with numbered menus to ensure it is easy to read for users with different level of digital confidence

Architectural Patterns Used

- **Client–Server Architecture:** Client communicates with Flask server via HTTP
- **Layered Architecture:** Separation of presentation, logic and data layers
- **Repository Pattern:** Centralised database access through utility functions

System Workflow

1. The user starts the programme and enters a username
2. The main menu displays several options
3. When creating a post: content is collected, hashtags are generated, keywords are checked, sentiment analysis produces affirmations, post is stored via the API
4. If the user views the public feed, the system retrieves all public posts from the API and the user may choose to like a post
5. If the user reviews their personal profile (including public and private posts), the system fetches posts linked to their username
6. Both resource hubs (support hub and wellness/career hub) can be accessed at any time
7. After every action, the system returns the user to the main menu, allowing user to continue using the application or exit

High-Level System Architecture:
Data Flow & Component Interaction

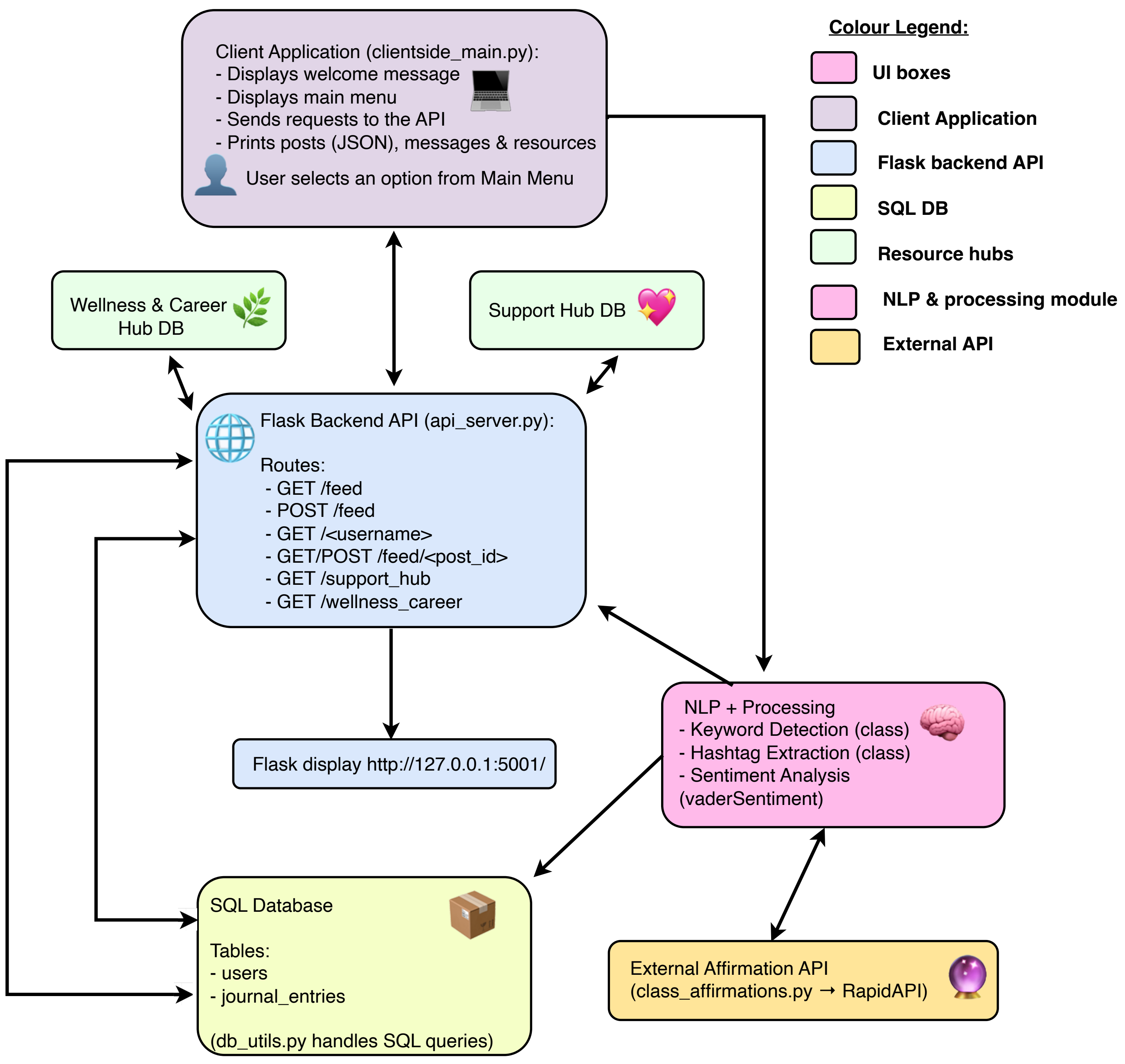
Client Application (clientside_main.py):
- Displays welcome message
- Displays main menu
- Sends requests to the API
- Prints posts (JSON), messages & resources
User selects an option from Main Menu
Wellness & Career Hub DB
Support Hub DB
Flask Backend API (api_server.py):
Routes:
- GET /feed
- POST /feed
- GET /<username>
- GET/POST /feed/<post_id>
- GET /support_hub
- GET /wellness_career
Flask display http://127.0.0.1:5001/
NLP + Processing
- Keyword Detection (class)
- Hashtag Extraction (class)
- Sentiment Analysis (vaderSentiment)
SQL Database
Tables:
- users
- journal_entries
(db_utils.py handles SQL queries)
External Affirmation API (class_affirmations.py → RapidAPI)
Colour Legend:
UI boxes
Client Application
Flask backend API
SQL DB
Resource hubs
NLP & processing module
External API

Figure 2. High-Level System Architecture: Data Flow & Component Interaction

This diagram provides an overview of how data moves through the entire system, showing communication between the client application, Flask backend API, SQL database, NLP processing module, and the external affirmation API.

IMPLEMENTATION & EXECUTION

Our team adopted a collaborative and Agile-based development approach to ensure group involvement.

We used Notion to organise and share work. We created a table to distribute the workload that listed the roles, responsibilities and members (See table 1 below). Roles were distributed based on interests and strength, informed by a simple SWOT analysis.

We first set up a SQL dummy database and the Flask API. We then aligned with the team on the database to integrate with the backend development. We then developed the client-side features and the core logic: sentiment analysis, affirmations, keyword detection, and hashtag recommendations. Throughout this process, we tested and refined the functions to keep the project aligned.

Role	**Tasks/Responsibilities**	**Team members involved**
Project Manager	GitHub workflow, set meetings, oversee task board	(Alisha, Sara, Yousr)
Backend Developers	Main API integration, keyword logic developer	(Alisha, Sara, Siham) (Chimna) (Priscilla, Siham)
Database Developer	Design and implement DB schema	(Mia, Yousr)
Tester/QA	Unit tests, local and manual integration tests	(Sara, Siham) (All team members)
Documentation	Writing and drafting the project document	(Mia, Alisha, Sara, Yousr, Siham, Chimna)
Presentation	Prepare slides	(Mia, Yousr)

Table. 1. Table showing roles and team members involved

Tools and libraries

Tool/library	**Description**
mysql.connector	Enables connection to the MySQL database, which stores all user data
flask	A lightweight web application framework that allows for uncomplicated web development. We have used this module to define API routes and expose database content, including the public feed endpoint. For example, to see our public feed, you will be able to do so on: http://127.0.0.1:5001/feed
requests	Handles HTTP GET and POST requests, used to retrieve affirmations from an external API and communicate with Flask routes
itertools	A module that supports efficient iteration when formatting and displaying user posts in the terminal
string	A module that provides string utilities, used to remove punctuation from posts during keyword detection
nltk	This natural language processing library was used in our hashtags class, and is incorporated into the wider "create post" functionality of the application.

vaderSentiment	A sentiment analysis tool, designed to analyse text to decipher a user's sentiment. We used this as part of the wider "create post" functionality of the application, to decipher the mood of a user's post, to produce a negative, neutral or positive label. This label is then used to produce a personalised affirmation to the user upon them creating a post
random	A module used to select affirmation categories from curated lists to personalise user feedback

Implementation process and challenges

1. **Rapid Iteration Led to Frequent Refactoring:** as new features were added, existing code often required restructuring to accommodate changes. This reinforced Agile principles and highlighted that clean code evolves through iteration.
2. **Inconsistent Sprint Velocities:** varying team availability led to over-or underestimation of sprint capacity, resulting in partially completed features. This improved our ability to plan more realistically and manage sprint scope.
3. **Dependencies Between Features Slowed Progress:** some functions depended on others being finished first; for example, database schema decisions impacted API routes, which then affected client-side functions. This emphasised the importance of identifying dependencies early during sprint planning.
4. **Managing Our Multi-File Architecture:** working across multiple files increased the risk of unintended side effects from small changes. This experience highlighted the need for clear structure, consistent naming and strong communication.
5. **SQL Relationships and Constraints:** we ran into issues where data didn't update properly because constraints weren't defined clearly enough. These issues taught us the importance of thorough database planning before coding.
6. **Merging Code Collaboratively:** merging code from six people meant we faced some conflicts and fragile branches. Our backend engineers had to communicate when they would push or alter code, to reduce conflict. Improved coordination and GitHub practices strengthened the team's confidence in branching, pull requests and conflict resolution.
7. **Debugging API + Database Interactions:** a few bugs came from the client, server and database not agreeing on formatting or unexpected responses. Layer-by-layer debugging reinforced the value of testing components independently before integration.
8. **Agile Development:** the project was managed using Agile and Scrum principles, with work divided into short, task-focused sprints. Progress and priorities of each section were tracked using a Kanban board in Notion, before integrating the app together.

   We had regular meetings and communication via Slack, WhatsApp and Zoom to review tasks, set deadlines and address challenges. When attendance was limited, meetings were recorded or notes shared to maintain alignment.

   At development stage, the team would review progress and discuss improvements for the next sprint to encourage progress. We used screen-sharing sessions to review codes and check the GitHub workflow, particularly around debugging SQL queries, Flask API requests and client-side functions.

TESTING & EVALUATION

Testing Strategy

1. Unit Testing of Core Functions: core database utilities and helper functions were manually unit-tested, including post retrieval, insertion, likes handling, hashtag generation, keyword detection and sentiment analysis.

API Route Testing Through Postman and Browser Checks: Flask routes were tested individually using Postman and browser-based GET requests to ensure: JSON responses matched expected structures, the API rejected invalid or incomplete POST requests, redirect behaviour for liking posts worked consistently, support Hub and Wellness Hub endpoints returned correctly formatted data.

3. End-to-End Testing Through the Client-side Application: we tested the complete workflow from start to finish, such as writing and submitting a post through to receiving personalised affirmations. This ensured the entire system behaved consistently when all components interacted together.

4. Collaborative Team Testing (User Simulation): each team member created their own journal entry using the live app. This collaborative simulation effectively acted as diverse user testing, helping us identify issues tied to real usage rather than controlled examples.

6. Regression Testing During Final Integration: due to multiple contributors, major features were re-tested following integration and code merges to prevent regression.

7. Functional and user testing: testing was regularly carried out. Below demonstrates the functional tests performed with the expected results versus the actual results (See table 2 below).

Functional test case	**Expected results**	**Actual results**
Create a public post	Post should be saved to SQL database, show in the user's profile and public feed	Post was created, stored and displayed correctly in user profile and public feed
View public feed	Show all public feeds for all users	All post entries displayed all posts made public. The app correctly displayed "There are no public posts yet" when no posts
User's personal profile	User should be able to see both public and private posts in order of most recent	App showed all correct post entries saved, most recent post first
Like a public post	Likes count should increase and username added to the 'user_likes' field database. Otherwise, prevents multiple likes from the user.	System incremented like counts 0 --> 1and stored username successfully. System displayed "You have already liked this post." If user liked again
Generating hashtags	App should recommend hashtags based on extracted keywords	App correctly showed list of suggested hashtags and accepted user input
Keyword detection trigger (e.g. "anxiety", "homeless", "abuse")	App should display support message and offer to redirect to the support hub	For sensitive posts, support message successfully appeared, and redirection worked as expected

Sentiment analysis and Affirmation generation	Posts should generate affirmation category based on sentiment level	Affirmation successfully appeared after user post entry
Support hub / Wellness hub	User can access categories and resources clearly and allow user to select one	Categories displayed correctly and selected category showed full resource list

Table 2. Each functional feature tested showing expected and actual results.

Summary of Unit Tests

1. Affirmations class test:
    a. *class TestSentimentAnalysis(TestCase)* - tests ability of the sentiment analysis to produce a positive, neutral and negative label;
    b. *class TestAffirmations(TestCase)* - tests the affirmation class's ability to produce a random category of affirmation from the curated lists of categories.
2. *test_hashtag_recommendations()* - tests the hashtag_generation helper function's ability to produce hashtags. The hashtag_generation function can be found in the clientside_main file.
3. *Keyword detection class test* - tests if the presence of a keyword will trigger a true response and for normal or unrelated posts without sensitive words return as false.
4. Print helper function tests:
    a. *test_print_helper_public_feed* - tests the print_helper_func function's ability to iterate over two lists (one being user's posts (the feed) and the other being titles for the elements in those posts e.g. ["Post id: ", "Username", etc.]), combine them and print them in a cohesive way; and
    b. *test_print_helper_username* - tests the print_helper_func function's ability to iterate over two lists (one being a particular user's posts and the other being titles for the elements in those posts e.g. ["Post id: ", "Username", etc.]), combine them and print them in a cohesive way.
5. *TestGetPublicFeed* - tests the ability of the api to return post data correctly. It includes a mock post for testing purposes.
6. *Support_hub_test* - tests whether the support_hub_helper function in the clientside file can return resources as specified by the user.
7. *TestGetUserPosts* - tests the ability of the application to return a user's public post data correctly. It includes a mock post for testing purposes.
8. *Wellness_career_helper_test* - tests whether the wellness_career_helper function in the clientside file can return resources as specified by the user.

System Limitations (not exhaustive)

1. **Simplified Data Model for Likes:** duplicate likes are prevented using a string-based userlikes column. While functional, this approach is not scalable and would require a relational table for high-volume use.
2. **Absence of Comments or Interaction Features:** the platform does not support comments or messaging, which limits community building. This aligns with the project scope, but restricts direct social support.

3. **No Real-Time Updating or Session Management:** the system relies on manual refreshes and repeated API requests, with no real-time updates, background tasks, or user sessions, resulting in a more static user experience (compared to modern applications).
4. **Database Structure Needs Normalisation for Scalability:** some values (hashtags, userlikes) are stored in single text fields rather than relational tables. This works for a small, educational-scale application, but limits scalability, performance and data analysis.
5. **No Automated Testing Framework:** although the team conducted extensive manual and functional testing, the system lacks unit tests using frameworks like PyTest.

CONCLUSION

Summary

OurVoice successfully achieves its goal of providing a safe, supportive and empowering digital journal platform for women and non-binary individuals. Through journaling, sentiment-aware affirmations, and curated resources, the system encourages meaningful self-reflection and personal growth.

Suggested Improvements / Future Work (not exhaustive)

1. **Username/password functionality:** a huge drawback is that the application currently lacks password protection.The existing users table in our SQL database could be extended to support password storage and stronger account security.
2. **Hashtag search functionality:** adding a feature to allow users to search posts by hashtags would allow users to search through hashtags to find posts related to any chosen topic of interest.
3. **Graphical User Interface:** we had long-term goals to create a graphical user interface for our application, for a more fun and interactive environment. A web or mobile version would make it more accessible and easier to use, allowing a much more modern experience like social media app beyond the terminal.
4. **Pagination for the feed:** future improvements could be made to manage many posts, such as showing posts smaller sections or pages, making it easier to browse and improve overall performance.
5. **Saving favourite resources:** allowing users to save/download resources would improve long-term accessibility and user engagement.
6. **Improved Natural Language Processing (NLP):** more advanced NLP methods would improve accuracy, enhance hashtag recommendations, refine affirmations and trigger support resources more reliably when a user may need help.