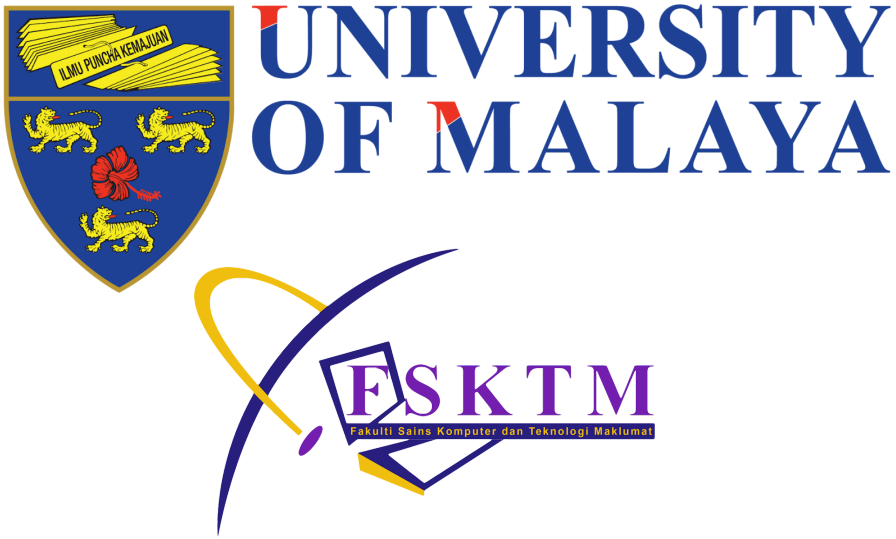


WIX 1002 FUNDAMENTAL OF PROGRAMMING

SEMESTER 1, 2025/2026

TOPIC 3: SMART JOURNAL

SMART JOURNALING APPLICATION : TECHNICAL REPORT

Lecturer's Name: DR. NURUL BINTI JAPAR

Occurence: 9

Group Name: Ctrl+C Ctrl+V

Group Members:

No.	Name	Matric Number
1	LIM HONG ZHANG	25006100
2	TAN CHEE KEAT	25006123
3	LEE MING DAO	25006825
4	JITESH A/L MOGANA RAJA	25006745
5	TEH XU ZHE	25006355

1. Problem Statement

1.1 Background and Context

Journaling is widely recognized in psychology as a powerful tool for emotional regulation, stress management, and personal growth. By externalizing thoughts, individuals can process complex emotions and gain clarity. However, in an increasingly digital and fast-paced society, the traditional practice of handwriting in a physical notebook has become obsolete for many. The friction of finding a pen, the inability to search through past entries, and the static nature of paper create a barrier to entry. Furthermore, while we generate immense amounts of data daily (location, weather, digital interactions), traditional diaries remain isolated from this context, failing to capture the environmental factors that often subconsciously drive our emotions.

1.2 Primary Limitations and Proposed Solutions

Our project addresses three specific critical failures in the current journaling ecosystem:

1. Inconsistency & High Friction

- **The Problem (Current State):** Maintaining a habit requires low effort. Traditional journaling suffers from high friction physical fatigue from writing, the need to carry a notebook, and the lack of reminders. Users often abandon the habit after a few days because the "activation energy" required to start writing is too high.
- **Solution:** We eliminate friction through a **Desktop-First Application**.
    - **Auto-Login:** Using UserSession and local token storage, users are instantly authenticated upon launching the app, removing the repetitive login barrier.
    - **Immersive UI:** The "Glassmorphism" design and video backgrounds make the act of opening the app visually rewarding, turning a chore into an aesthetic experience.

2. Loss of Contextual Atmosphere

- **The Problem (Current State):** A journal entry is a snapshot of text, but memory is multisensory. When a user reads an entry from a year ago that says "I felt gloomy today," they often forget the external factors such as was it because of the rain? Was it late at night? Without recording the *atmosphere*, the entry loses half its meaning. Manual tracking of weather is tedious and often skipped.
- **Solution:** We automate context capture using **Environmental APIs**.
    - **Automated Weather Tracking:** The system automatically fetches real-time meteorological data (e.g., "Thunderstorms") from the Malaysian Open Data API (data.gov.my) the moment an entry is created.
    - **Atmospheric Replay:** The application dynamically changes its background video (e.g., rain.mp4) to match the weather, not only recording the data but recreating the *feeling* of that moment for the user.

3. Lack of Analytical Insight

- **The Problem (Current State):** Traditional diaries are "read-only." A user might write thousands of words, but they cannot easily query their emotional history. Identifying patterns like "I am consistently anxious on Sunday nights" or "My mood improves when it’s sunny" requires impossible manual analysis. The user gets no feedback on their mental state.
- **Solution:** We provide active feedback using **Artificial Intelligence (AI)**.
    - **Sentiment Analysis:** Every entry is processed by an NLP model (DistilBERT via Hugging Face API) to generate a quantitative "Mood Score" (e.g., Positive: 98%).
    - **Data Visualization:** The SummaryPage aggregates this data into weekly bar charts, allowing users to visually spot trends between their mood and external factors (Weather) over the last 7 days.

2. Scope of the Project

The scope of the **"Smart Journal"** project defines the specific boundaries, functionalities, and technical environment within which the application operates. It bridges the gap between traditional diary keeping and modern data analytics, focusing on a **desktop-first** solution for digital well-being.

2.1 Domain and Purpose

The project operates within the domain of **Affective Computing** and **Personal Data Management**. Unlike standard text editors, "Smart Journal" is designed to function as an intelligent companion that understands the context of the user's life. The primary purpose is to provide a secure, distraction-free environment for self-reflection while autonomously gathering environmental and emotional data to provide users with actionable insights into their mental patterns.

2.2 Target Audience

The application is specifically tailored for:

- **Students and Academics:** Individuals who experience high-stress environments and need a quick, accessible method to document their daily progress and emotional state without the time commitment of physical writing.
- **Self-Improvement Enthusiasts:** Users who value data-driven feedback and wish to correlate their mood with external factors (like weather or time of week) to optimize their productivity and happiness.

2.3 Functional Scope (System Capabilities)

The "Smart Journal" system acts as a central hub for three distinct data streams:

1. **User-Generated Content (Active Input):**
    - **Rich Text Entry:** The core functionality allows users to write unlimited text entries. The system supports full CRUD (Create, Read, Update, Delete) operations, allowing users to revisit and refine past entries via a date-picker interface.
    - **Secure Authentication:** The scope includes a robust security layer. Users interact with a Registration and Login module that handles credential verification locally, ensuring that access to the journal is restricted to the specific user.
2. **Contextual Data Acquisition (Passive Input):**
    - **Automated Environmental Sensing:** The application connects to external REST APIs (specifically the Malaysian Open Data API) to fetch real-time weather data. This removes the need for users to manually record the weather,

ensuring every journal entry is automatically tagged with accurate atmospheric context.
    - **AI-Powered Sentiment Analysis:** The scope includes the integration of Natural Language Processing (NLP). The system sends user text to the Hugging Face Inference API to perform sentiment analysis, returning a quantitative "Mood Score" (e.g., Positive, Negative) and a confidence percentage.
3. **Data Visualization & Feedback (Output):**
    - **Weekly Analytics:** The system includes a Summary Dashboard that processes data from the past 7 days. It visualizes mood trends and weather frequency, transforming raw daily logs into a structured overview of the user's week.

2.4 Technical Scope

- **Platform:** The "Smart Journal" is developed as a standalone **Java Desktop Application**, utilizing the **JavaFX** framework to render a high-fidelity, hardware-accelerated user interface.
- **Data Persistence Strategy (Hybrid Model):**
    - **Local Storage:** Lightweight data, such as session tokens (session.token) and encrypted user credentials (UserData.txt), are stored locally on the client machine to ensure fast startup times and offline capability for login checks.
    - **Cloud Storage:** Journal entries, which require flexibility and scalability, are stored in a **MongoDB Atlas (Cloud)** cluster. This ensures data is not lost if the local machine fails and allows for flexible data structures (NoSQL) where weather and mood data can vary between entries.
- **User Interface Paradigm:** The scope strictly defines a "Glassmorphism" aesthetic. The UI utilizes semi-transparent panes and dynamic background videos (triggered by weather data) to create an immersive, calm user experience that differentiates "Smart Journal" from standard office software.

3. Project Requirements

The requirements are categorized into Functional (what the system does) and Non-Functional (how the system performs) to ensure a robust and user-centric architecture for the "Smart Journal."

3.1 Functional Requirements

1. Secure Authentication & Session Persistence

- **Registration & Encryption:** The system must allow new users to create an account using a unique email and password. Crucially, the system **must never store passwords in plain text**. All passwords must be hashed using the **SHA-256 algorithm** with a unique salt before being written to the local storage (UserData.txt).
- **Auto-Login (Tokenization):** To reduce friction, the system must generate a local session token (session.token) upon successful login. On subsequent launches, the application must read this token to bypass the login screen, effectively remembering the user.

2. Intelligent Journal Management (Core)

- **CRUD Operations:** Users must be able to Create, Read, and Update journal entries. The interface must include a DatePicker to allow users to navigate to past dates and retrieve or modify historical entries stored in the Cloud Database.
- **Automated Context Retrieval:** Upon opening the editor, the system must **autonomously** trigger a background request to the Malaysian Open Data API. It must retrieve the current weather forecast (e.g., "Thunderstorms") and append this metadata to the journal entry without user intervention.
- **On-Demand Sentiment Analysis:** The system must provide an "Analyze" function. When triggered, the text content must be serialized and sent to the **Hugging Face Inference API**. The system must parse the returned JSON to extract the dominant emotion (Positive/Negative) and a confidence score to be saved alongside the text.

3. Analytical Dashboard (Feedback Loop)

- **Data Aggregation:** The system must query the MongoDB database for entries falling within the **last 7 days** (from LocalDate.now()).
- **Visual Statistics:** It must calculate the frequency of specific moods and weather conditions. This data must be rendered visually (via text-based bar charts or UI indicators) to help the user identify correlations between their environment and their emotional state.

3.2 Non-Functional Requirements

1. Responsiveness & Concurrency

- **Asynchronous Execution:** To prevent the "Not Responding" state common in desktop apps, all **blocking I/O operations**—specifically HTTP requests to external APIs and read/write operations to the MongoDB cluster—must be executed on background threads (Worker Threads). The JavaFX Application Thread must remain free to handle UI events.
- **Latency Management:** The system must implement a **caching mechanism** for weather data. If a user opens the editor multiple times within a 15-minute window, the system must serve the cached weather data rather than triggering a new API call, preserving bandwidth and API rate limits.

2. Reliability & Fault Tolerance

- **Graceful Degradation:** The application must be robust against network failures. If the Internet is disconnected:
    - The Weather module must default to a cached value or a neutral state ("Cloudy").
    - The Journal Save function must alert the user via the UI ("Network Unreachable") rather than crashing the application with a runtime exception.
- **Database Scalability:** The system must utilize a **NoSQL document structure** (BSON) rather than a rigid SQL schema. This allows the data model to evolve (e.g., adding new mood metrics) without breaking compatibility with older entries.

3. Immersive User Experience (UX)

- **Dynamic Atmospherics:** The UI must be context-aware. The background of the application must dynamically load and play video files (e.g., rain.mp4, night.mp4) that correspond to the fetched weather data, creating a seamless connection between the user's physical environment and digital workspace.
- **Glassmorphism Aesthetic:** The interface must utilize CSS styling for semi-transparency and background blurring, ensuring high readability of text while maintaining the aesthetic focus on the background visuals.

4. Methodology and Approach

The development of the "Smart Journal" application was guided by rigorous software engineering principles, primarily adhering to the **Object-Oriented Programming (OOP)** paradigm. We utilized the **Model-View-Controller (MVC)** architectural pattern to decouple the user interface from the business logic, ensuring the system is maintainable, testable, and scalable.

4.1 Architectural Pattern: Model-View-Controller (MVC)

We strictly enforced the separation of concerns by organizing our codebase into three distinct layers. This approach allowed frontend and backend tasks to be developed in parallel without conflict.

- **1. The Model (Data & Logic Layer):**
    - *Function:* This layer manages the data logic and database interactions. It has no knowledge of the user interface.
    - *Implementation:*
        - **Entity Classes:** We defined POJOs (Plain Old Java Objects) such as User.java to represent system entities.
        - **Data Access Object (DAO) Abstraction:** Classes like journalApp.java and UserManager.java act as intermediaries between the code and the storage (MongoDB/File System). They handle complex operations like **BSON serialization** and **file parsing** internally, exposing simple methods (e.g., saveEntry(), validateLogin()) to the controllers.
- **2. The View (Presentation Layer):**
    - *Function:* This layer is responsible purely for visualizing data and capturing user input.
    - *Implementation:*
        - **FXML (JavaFX Markup Language):** We used .fxml files (e.g., JournalEditor.fxml) to define the structure of the UI (buttons, text areas, layout containers). This separates the layout structure from the Java code.
        - **CSS Styling:** All visual aesthetics, including the "Glassmorphism" effect (blur, transparency, gradients), were offloaded to external stylesheets (journal.css, glass.css). This allows for rapid design iterations without recompiling Java code.
- **3. The Controller (Interaction Layer):**
    - *Function:* This layer acts as the bridge. It intercepts user events (clicks, typing), updates the Model, and refreshes the View.
    - *Implementation:* Classes like JournalEditorController.java utilize JavaFX annotations (@FXML) to inject UI components directly into the Java logic. They handle event listeners (e.g., saveButton.setOnAction) and orchestrate the flow of data between the APIs and the UI.

4.2 Design Patterns Applied

To solve common software design problems, we implemented standard GoF (Gang of Four) design patterns:

- **Singleton Pattern:**
  - *Usage:* Applied in MongoDBConnection.java and UserSession.java.
  - *Reasoning:* Database connections are resource-intensive. By implementing the connection as a Singleton, we ensure that the application maintains **exactly one** active connection pool to the MongoDB Atlas cluster throughout its lifecycle. This prevents memory leaks and ensures thread safety during concurrent write operations.
- **Facade Pattern (Utility Wrappers):**
  - *Usage:* Applied in MoodAnalyzer.java and WeatherBackgroundManager.java.
  - *Reasoning:* These classes wrap the complex logic of HTTP Request construction, Header authentication, and JSON parsing. The main application simply calls getMood() without needing to understand the underlying HTTP protocols.

4.3 Development Tools and Environment

The project was built using a modern, industry-standard toolchain to ensure compatibility and dependency management:

Category	**Tool / Technology**	**Purpose**
Language	**Java JDK 24**	Utilized for its strong typing and latest features.
Build Tool	**Apache Maven**	Used to manage external dependencies (org.mongodb, org.json, org.openjfx) via the pom.xml file. This ensures the project is portable and builds consistently on any machine.

Framework	**JavaFX**	Chosen for its hardware-accelerated graphics pipeline, essential for rendering high-resolution video backgrounds smoothly.
Database	**MongoDB Atlas**	A Cloud-based NoSQL database chosen for its flexible Document schema, allowing us to store unstructured metadata (Weather/Mood) alongside structured text.
Version Control	**Git & GitHub**	Used for collaborative development, branch management, and tracking changes across the group.

4.4 Implementation Strategy (Phases)

We followed an **Iterative and Incremental** development lifecycle:

1. **Phase 1: Core Backend & Security (The Foundation)**
    - We prioritized the security layer first. We implemented UserManager.java to handle SHA-256 password hashing and file I/O. We verified that users could be registered and authenticated locally before building any UI.
2. **Phase 2: Cloud Connectivity (The Data Layer)**
    - We established the connection to MongoDB using MongoDBConnection.java. We created the EnvLoader utility to securely load API keys from a local .env file, ensuring credentials were never hardcoded (a crucial security practice).
3. **Phase 3: Frontend & Asynchronous Integration (The User Experience)**
    - We designed the FXML interfaces.
    - *Critical Step:* We implemented **Multithreading**. We refactored all network calls (Weather API, Mood API) to run on background threads (new Thread()) and used Platform.runLater() to update the UI. This prevented the interface from freezing during network latency, a common issue in synchronous applications.

5. System Flow Chart

(Textual Representation of the Flow Logic)

1. Application Initialization (Startup)

- Start: The application launches via Ctrl.java.
- Session Check:
    - The system attempts to read the session.token file.
    - If Valid: The system restores the user session and proceeds directly to the Landing Page.
    - If Invalid/Missing: The system proceeds to the Login Page.

2. Authentication Flow

A. Login Page

- Display: Shows email/password fields and a background video determined by current weather (fetched in a background thread).
- Input: User enters Email and Password.
- Verification:
    - The system hashes the input password using SHA-256 with the user's specific salt.
    - It compares the hashed result against the records in UserData.txt.
- Outcome:
    - Success: A session.token file is created, and the user is redirected to the Landing Page.
    - Failure: An error message ("Invalid email or password") is displayed.

B. Registration Page

- Input: User enters Display Name, Email, and Password.
- Validation:
    - Checks if the email contains ".com".
    - Checks if fields are empty.
    - Checks UserData.txt to ensure the email is not already registered.
- Processing:
    - Generates a unique random Salt.
    - Hashes the password combined with the Salt.
    - Appends the new user record (Email, Name, Hashed Password, Salt) to UserData.txt.
- Outcome: Redirects the user back to the Login Page to sign in.

3. Main Dashboard (Landing Page)

- Initialization:
    - Weather Fetch: A background thread calls the Malaysia Open Data API to get the current weather forecast.
    - UI Update:
        - Sets the video background based on weather (e.g., rain.mp4 for "Thunderstorms").
        - Displays a dynamic Greeting (Good Morning/Afternoon/Evening) based on the current system time.
        - Displays the current date and time.
- User Options:
    - New Entry: Opens the Journal Editor.
    - View Summary: Opens the Weekly Summary.
    - Log Out: Deletes session.token and returns to the Login Page.

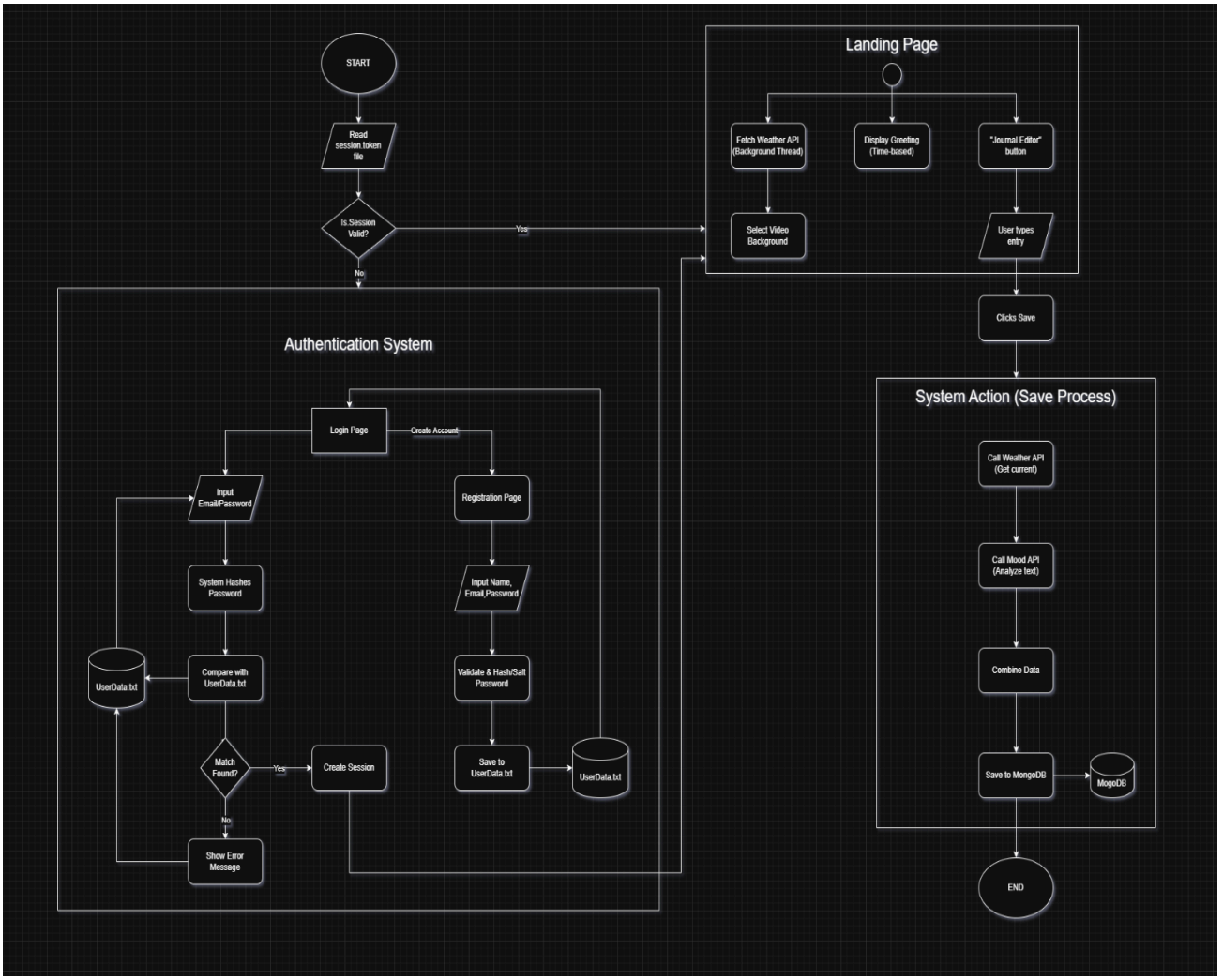
4. Journaling Feature

- Editor Load:
    - Loads the journal entry for the selected date (default is Today) from MongoDB.
    - Updates the weather display in real-time.
- User Action: Analyze Mood (Optional):
    - User clicks "Analyze Mood".
    - System sends the text to the Hugging Face API (DistilBERT model) via a POST request.
    - Returns and displays the mood label (e.g., "Positive") and confidence score.
- User Action: Save Journal:
    - Step 1: Re-fetches current weather from the API.
    - Step 2: Performs a final Mood Analysis on the text.
    - Step 3: Combines Date, Entry, Weather, and Mood into a document.
    - Step 4: Upserts (Updates or Inserts) the document into the MongoDB journals collection.
    - Outcome: Closes the editor and returns to the Landing Page.

5. Weekly Summary Feature

- Data Retrieval:
    - Fetches journal entries for the current user from the past 7 days from MongoDB.
- Processing:
    - Aggregates counts for Moods (Positive vs. Negative).
    - Aggregates counts for Weather conditions.

- Display:
    - Renders text-based bar charts representing the mood and weather distribution for the week.



6. Module Description

The system is architected into distinct packages (modules) to enforce the **Separation of Concerns** principle. This ensures that the user interface, business logic, and data access layers remain independent, facilitating easier debugging and future scalability.

6.1 Root Module (Application Entry)

- **Ctrl.java**
    - **Role:** The bootstrap class for the JavaFX application.
    - **Key Functionality:** It initializes the primary Stage, configures the application window (setting it to borderless/fullscreen for immersion), and implements the **Session Check Logic**. Upon launch, it inspects the local directory for a valid session.token. If found, it bypasses the login screen and routes the user directly to the Landing Page; otherwise, it loads the Welcome/Login interface.

6.2 Authentication & User Management Module (registration)

This module handles all security, identity verification, and local session persistence.

- **UserManager.java (Logic/DAO)**
    - **Role:** The backend logic for handling user credentials.
    - **Key Functionality:** It manages the read/write operations to the local UserData.txt database. It implements **SHA-256 Hashing** with salt generation to ensure passwords are never stored in plain text. It provides methods like registerUser() and validateLogin() which are called by the controllers.
- **User.java (Model)**
    - **Role:** A POJO (Plain Old Java Object) entity class.
    - **Key Functionality:** It defines the data structure for a user, encapsulating fields such as email, hashedPassword, salt, and displayName. It ensures type safety when passing user data between the Manager and the Controller.
- **UserSession.java (Singleton)**
    - **Role:** Manages the global state of the currently logged-in user.
    - **Key Functionality:** Implements the **Singleton Design Pattern** to ensure only one session exists at a time. It handles the creation and deletion of the session.token file, allowing the application to "remember" the user across restarts.
- **LoginController.java & RegisterController.java (View Controllers)**
    - **Role:** UI Logic for the authentication screens.
    - **Key Functionality:** These classes handle JavaFX event listeners (e.g., button clicks). They perform **Input Validation** (checking for empty fields or invalid email formats) before passing data to the UserManager. They also manage scene transitions (switching from Login to Dashboard).

6.3 Core Journaling Module (journalpage)

This module contains the primary feature set: the editor and its database interactions.

- **JournalEditorController.java (Controller)**
    - **Role:** The orchestrator of the writing experience.
    - **Key Functionality:**
        - **UI Management:** Controls the TextArea for writing and the DatePicker for navigation.
        - **Thread Management:** It spawns **background threads** to perform heavy operations (saving to Cloud, fetching AI analysis) without freezing the UI.
        - **Context Binding:** It automatically calls the Weather module to fetch environmental data when the editor loads.
- **journalApp.java (Data Access Object)**
    - **Role:** The dedicated interface for MongoDB journal operations.
    - **Key Functionality:** It constructs **BSON Documents** containing the entry text, date, mood, and weather. It utilizes the MongoDB ReplaceOptions().upsert(true) method, ensuring that if an entry already exists for a specific date, it is updated; otherwise, a new one is created.

6.4 Intelligence Modules (mood & weather)

These modules handle external API integrations to provide "Smart" features.

- **MoodAnalyzer.java**
    - **Role:** The bridge to the Artificial Intelligence engine.
    - **Key Functionality:** It constructs a JSON payload containing the user's text and sends a secure **HTTP POST** request to the **Hugging Face Inference API**. It parses the complex nested JSON response to extract the specific sentiment label (e.g., "Positive") and its confidence score.
- **WeatherBackgroundManager.java**
    - **Role:** Manages environmental context and UI atmospherics.
    - **Key Functionality:**
        - **API Fetching:** Connects to the **Malaysian Open Data API** to retrieve real-time weather forecasts.
        - **Caching Strategy:** Implements a time-based cache (15 minutes) to prevent redundant network calls and API rate limiting.
        - **Visual Mapping:** Maps weather strings (e.g., "Hujan", "Thunderstorms") to specific local video file paths (e.g., rain.mp4), allowing the UI to dynamically reflect the weather.
- **API_Post.java & API_Get.java**
    - **Role:** Generic Network Utilities.

    - **Key Functionality:** These are helper classes that handle the low-level Java HttpURLConnection logic, managing headers (Authorization, Content-Type), output streams, and reading the input stream into a String response.

6.5 Dashboard Module (landingpage)

- **LandingPageController.java**
    - **Role:** The main hub or "Home Screen" of the application.
    - **Key Functionality:** It initializes the dynamic video background using the WeatherBackgroundManager. It provides navigation routing, allowing the user to jump to the Journal Editor, Summary Page, or Logout. It also displays a real-time digital clock and the current date.

6.6 Analytics Module (summary)

- **SummaryPage.java (Logic)**
    - **Role:** The statistical processing engine.
    - **Key Functionality:** It queries the MongoDB database for a DateRange (last 7 days). It iterates through the retrieved documents and uses **HashMaps** to aggregate frequency data (e.g., counting how many times "Sunny" or "Sad" appeared).
- **SummaryController.java (View Controller)**
    - **Role:** Visualizes the processed data.
    - **Key Functionality:** It receives the aggregated data from SummaryPage and dynamically updates JavaFX UI components (such as text-based bar charts or progress bars) to present the Weekly Mood and Weather reports to the user.

6.7 Infrastructure & Utilities (utils)

- **MongoDBConnection.java**
    - **Role:** Database Connection Manager.
    - **Key Functionality:** Implements the **Singleton Pattern** to manage the connection pool to the MongoDB Atlas Cloud. This ensures the application establishes the connection once and reuses it, optimizing performance and preventing connection leaks.
- **EnvLoader.java**
    - **Role:** Security Utility.
    - **Key Functionality:** It reads the .env configuration file to load sensitive credentials (API Keys, Database Connection Strings) into system properties. This prevents hardcoding secrets directly into the source code, adhering to security best practices.

6.8 Welcome Module (welcome)

- **welcome.java**

    - **Role:** The initial splash screen or pre-login landing.
    - **Key Functionality:** It serves as the aesthetic introduction to the application, likely displaying the branding or group logo before transitioning the user to the Login/Register selection screen. It sets the initial tone of the "Glassmorphism" UI experience.

7. Data Structures Used

The "Smart Journal" application utilizes a variety of standard Java Collections and specialized objects to manage memory efficiency, data persistence, and algorithmic speed.

7.1 Linear Data Structures

1. Dynamic Array (java.util.ArrayList<User>)

- **Location Used:** UserManager.java
- **Description:** A resizable array implementation of the List interface.
- **Context & Usage:** The application reads the flat-file database (UserData.txt) line-by-line and deserializes each line into a User object. These objects are stored in an ArrayList.
- **Technical Justification:** We chose ArrayList over a standard Array (User[]) because the number of registered users is dynamic and unknown at runtime. ArrayList allows for O(1) random access and automatically handles resizing when new users register.

2. Mutable Character Sequence (java.lang.StringBuilder)

- **Location Used:** API_Get.java, API_Post.java, WeatherBackgroundManager.java
- **Description:** A mutable sequence of characters used for efficient string concatenation.
- **Context & Usage:** When reading the InputStream from the Weather or Mood APIs, the data arrives in byte chunks. We use a StringBuilder to append these lines inside a while loop to construct the final JSON string.
- **Technical Justification:** Using standard String concatenation (e.g., str += line) inside a loop creates a new String object in the Heap for every iteration ($O(n^2)$). StringBuilder modifies the existing buffer in place ($O(n)$), which is critical for preventing memory leaks during network operations.

7.2 Key-Value & Hashing Structures

3. Hash Map (java.util.HashMap<String, Integer>)

- **Location Used:** SummaryPage.java
- **Description:** A data structure that stores items in "key/value" pairs, using a hashing function to compute an index.
- **Context & Usage:** To generate the Weekly Summary, the system iterates through the past 7 days of journal entries. It uses a HashMap where the **Key** is the mood/weather (e.g., "Positive", "Rain") and the **Value** is the frequency count.
- **Technical Justification:** This provides **O(1) average time complexity** for lookups and insertions. This allows the system to instantly update the count for a specific mood without needing to iterate through a list to find if the mood was already recorded.

4. BSON Document (org.bson.Document)

- **Location Used:** journalApp.java, MongoDBConnection.java
- **Description:** The primary data structure used by the MongoDB Driver, effectively an ordered map that supports nested types.
- **Context & Usage:** Unlike SQL which requires rigid tables, the Document structure allows us to bundle heterogeneous data types—Date (Object), Entry (String), Mood (String), and Weather (String)—into a single object for cloud storage.
- **Technical Justification:** It serializes directly to Binary JSON (BSON), which is the native format for MongoDB. This structure is "schema-less," meaning if we decide to add a "Location" field in a future update, we can do so without breaking the code for existing entries.

7.3 Hierarchical/Tree Structures

5. JSON Objects (org.json.JSONObject / org.json.JSONArray)

- **Location Used:** MoodAnalyzer.java, WeatherBackgroundManager.java
- **Description:** A tree-like structure used to represent data in a lightweight, text-based, language-independent format.
- **Context & Usage:** External APIs return data in nested structures. For example, the Hugging Face API returns a JSONArray of results, which contains a JSONObject, which contains a field score.
- **Technical Justification:** These structures provide parsing methods (like optString() or getJSONArray()) that handle null safety. They allow us to traverse the deep hierarchy of the API response to extract only the specific data points needed (e.g., just the "label" field) while ignoring the rest.

7.4 Binary Structures

6. Byte Array (byte[])

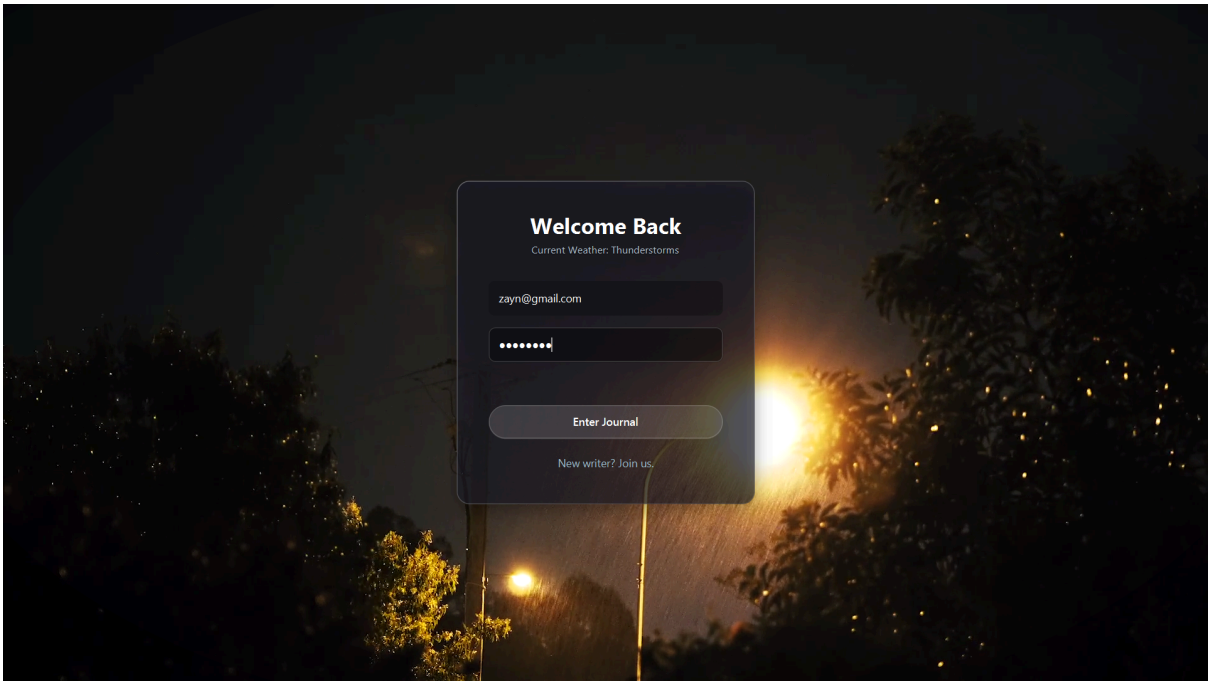
- **Location Used:** UserManager.java
- **Description:** A primitive array of bytes.

- **Context & Usage:** During the authentication process, the SHA-256 hashing algorithm processes the user's password and salt as raw binary data, not strings.
- **Technical Justification:** Cryptographic operations operate on bits and bytes. Using byte[] is mandatory for the MessageDigest class to perform bitwise operations for secure password hashing.

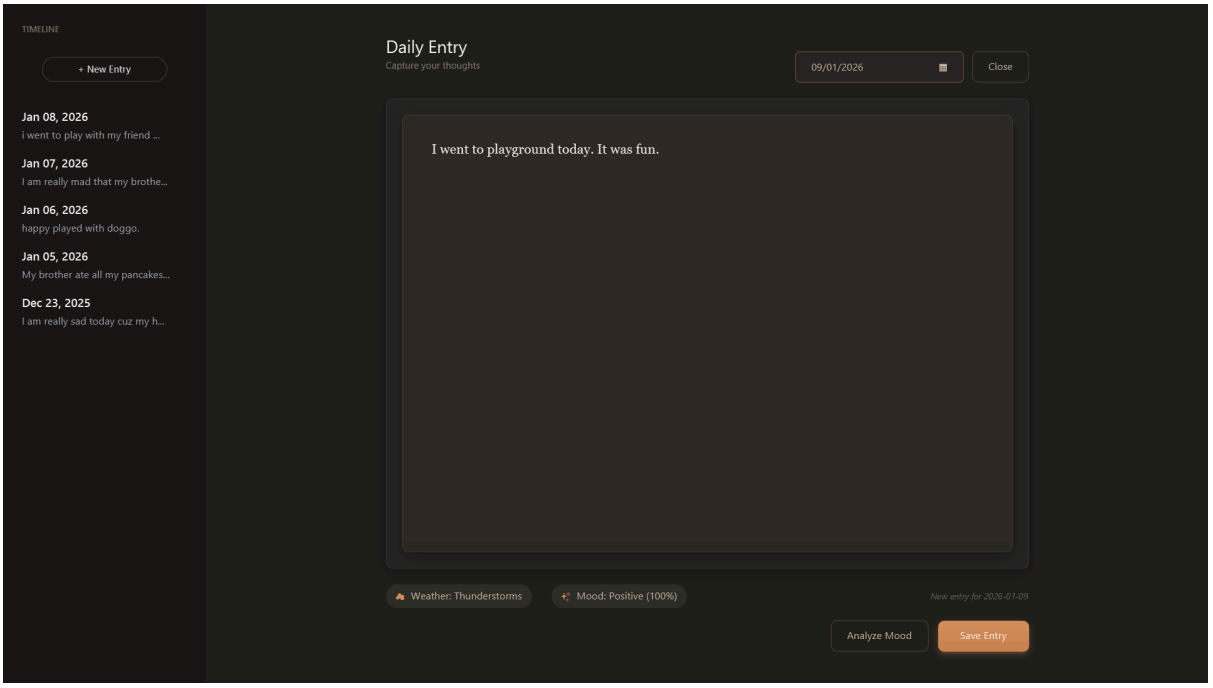
8. Program Output (Scenarios)

Scenario A: Normal Input (Successful Flow)

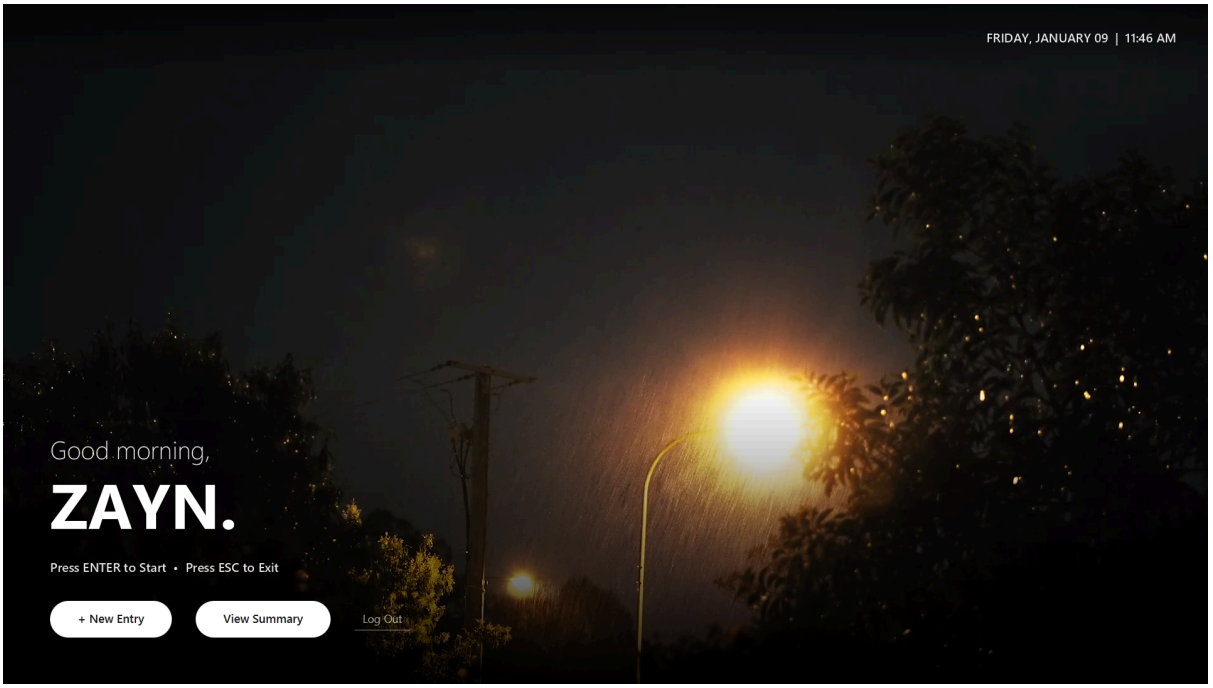
- **Input:**
    - User logs in with valid credentials zayn@gmail.com.



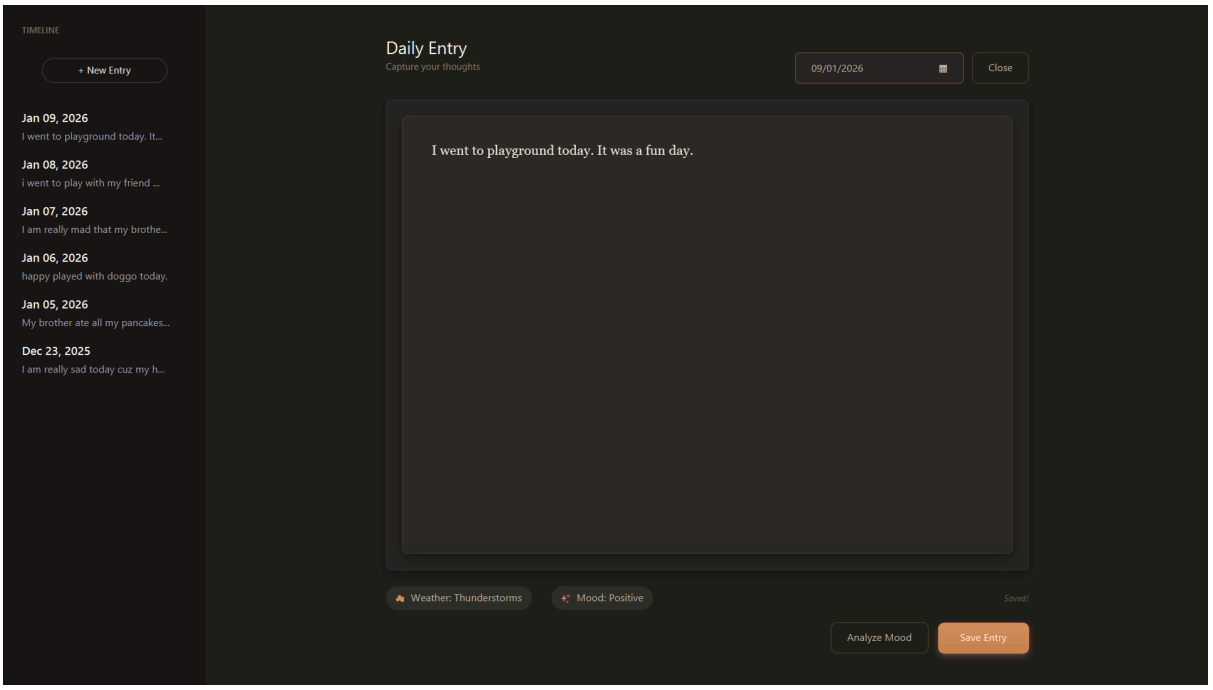
    - Writes journal: "I went to playground today. It was fun."



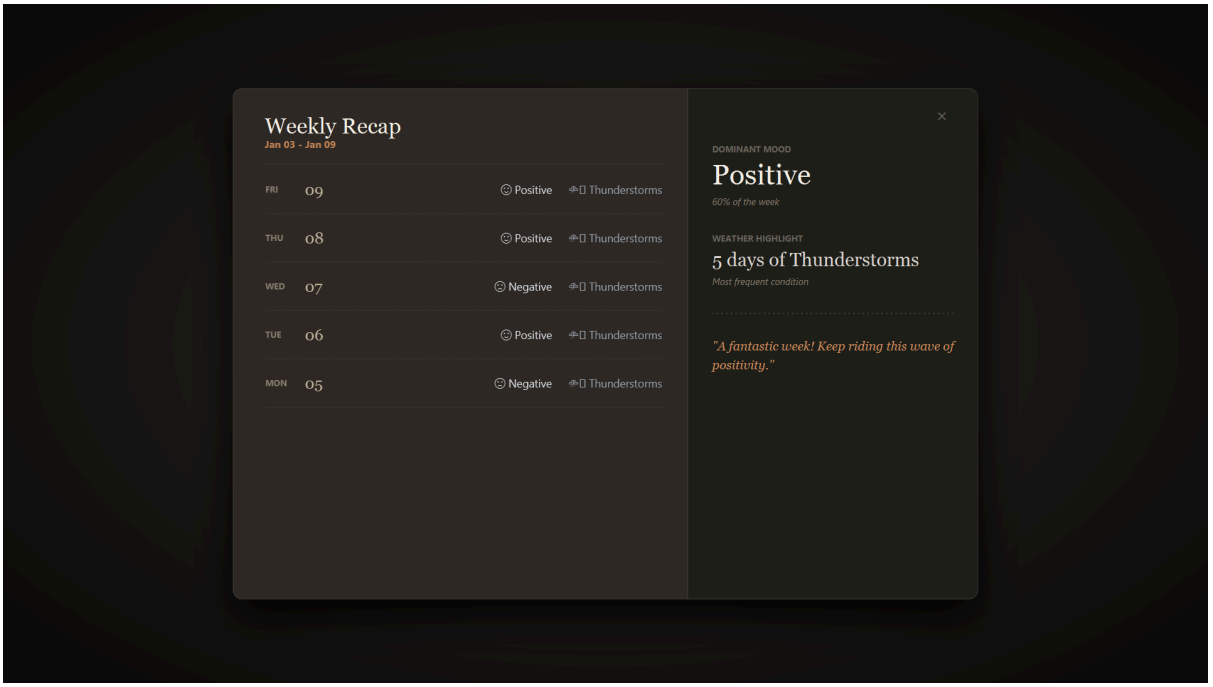
- **Output:**
    - **Login:** Redirects to Landing Page. Background video changes to "Thunderstorm" (based on API).



    - **Save:** Status label shows "Journal saved successfully!".



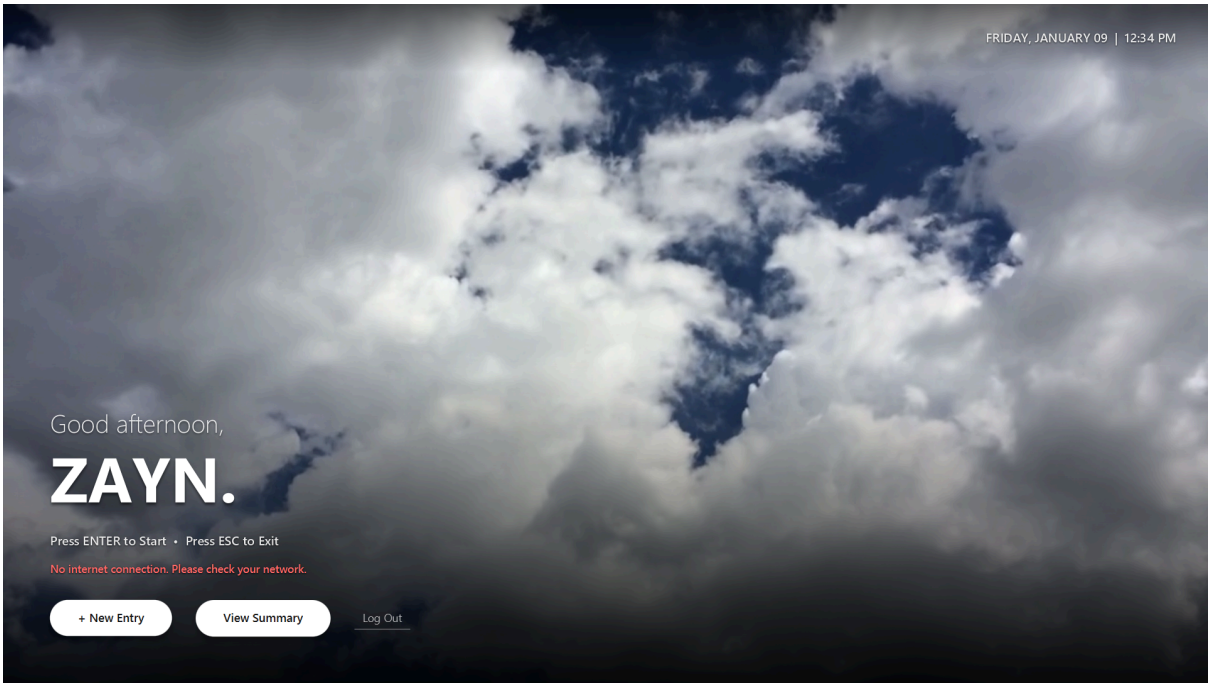
- **Analysis:** Mood label updates to "Mood: Positive (60%)".



- **Database:** Entry stored in MongoDB with date 2026-01-09.

Scenario B: Error Input (Reliability Test)

- **Input:**
    - User attempts to login while the internet is disconnected.
    - User attempts to create a new journal or edit existing journal while the internet is disconnected.
- **Output:**
    - **System Behavior:**
        - User can still login to the landing page.
        - The application **does not crash**.
        - User is unable to create a new journal or edit existing journals because the internet is disconnected.
    - **Status Label:** Displays "No internet connection. Please check your network."
    - **Fallback:** Background display weather defaults to "Cloudy" (default value) .



9. Conclusion and Future Improvements

9.1 Conclusion

The "Smart Journal" project stands as a successful proof-of-concept that demonstrates how traditional self-care practices can be revitalized through modern software engineering. By moving beyond simple text entry and integrating context-aware technologies, we have created an application that is not just a tool for recording, but a system for understanding.

The project achieved three key technical milestones:

1. **Seamless Cloud Integration:** The successful implementation of a hybrid storage model (Local Tokens + Cloud Database) proves that desktop applications can offer the same connectivity as web apps. The use of **MongoDB Atlas** ensures that user data is secure, scalable, and independent of the local machine's hardware.
2. **High-Fidelity User Experience:** By strictly adhering to the **Glassmorphism** design language and utilizing JavaFX's hardware acceleration for video backgrounds, we solved the "boredom" and "friction" problems inherent in traditional journaling. The UI is not merely functional; it is immersive.
3. **Robust Concurrency Model:** The rigorous application of background threading for all API and Database I/O ensures the application meets professional standards of responsiveness. The system handles network latency gracefully without ever freezing the main application thread, providing a fluid user experience even under poor network conditions.

In summary, "Smart Journal" meets all specified functional requirements while delivering a sophisticated, data-driven feedback loop that empowers users to take control of their mental well-being.

9.2 Future Improvements

While the current version is robust, several enhancements could further elevate the system's capability and user reach.

1. Dynamic Geolocation & Hyper-Local Weather

- **Current Limitation:** The weather logic is currently hardcoded to fetch data for "Kuala Lumpur," which limits the accuracy for users traveling or living outside the Klang Valley.
- **Proposed Improvement:** Implement automatic **IP-based Geolocation**.
- **Implementation Strategy:** Upon launch, the system would ping a geolocation API (e.g., ip-api.com) to retrieve the user's current latitude and longitude. These coordinates would then be passed dynamically to the Open Weather API query string. This ensures that if a user travels to London, the journal automatically records "Foggy" instead of the weather in Malaysia.

2. Voice-to-Text Transcription (Accessibility)

- **Current Limitation:** The application relies solely on keyboard input, which excludes users with motor disabilities or those who prefer speaking over typing.
- **Proposed Improvement:** Integrate a Speech-to-Text engine to allow hands-free journaling.
- **Implementation Strategy:** We can integrate the **Google Cloud Speech-to-Text API** or the **CMU Sphinx** Java library. A microphone button would be added to the JournalEditor. When active, an audio stream would be captured, sent to the processing engine, and the returned text would be appended in real-time to the TextArea. This would significantly lower the friction of creating long entries.

3. Mobile Companion App (Cross-Platform Synchronization)

- **Current Limitation:** Users can only journal when they are physically at their desktop computer.
- **Proposed Improvement:** Develop a lightweight mobile application (Android/iOS) that syncs with the desktop version.
- **Implementation Strategy:** Since our architecture already uses a Cloud Database (**MongoDB Atlas**), no backend migration is required. We would build a mobile interface using a cross-platform framework like **Flutter** or **React Native**. This mobile app would connect to the *exact same* MongoDB cluster using the same User credentials. This allows a user to write a quick entry on their phone while on the bus, and see it fully analyzed and visualized on their desktop when they get home.

4. Advanced Encryption (End-to-End Security)

- **Current Limitation:** While passwords are hashed, the actual journal text is stored as readable strings in the database.
- **Proposed Improvement:** Implement client-side AES Encryption for journal entries.
- **Implementation Strategy:** Before the journalApp DAO sends the text to MongoDB, it would encrypt the content using **AES-256** with a key derived from the user's password. This ensures that even if the database is compromised, the personal thoughts of the user remain mathematically unreadable to anyone but the user themselves.