Software Design Document
for
WakeUP: Alarm Clock Reinvented

CS5337: Software Engineering
Prof. Dr. Jung Soo Lim

Prepared by Rizwan Islam, Jooeun Jeon, Daniel Macias,
Darpan Patel, Yug Kalubhai Patel

Table of Contents

Code Revision History

Name	Date	Reason for Changes	Version
Rizwan Islam, Jooeun Jeon, Daniel Macias, Darpan Patel, Yug Kalubhai Patel	10/15/2024	Initial Draft	1.0
Rizwan Islam	12/10/2024	Addition to app UI	1.1

1. Introduction

1.1 Overview

This document serves as the Software Design Document (SDD) for the "WakeUP" project, outlining the design and architectural decisions made to guide its development. It begins by defining the purpose and scope of the application, highlighting its core functionalities and target audience. Subsequent sections provide detailed descriptions of the system architecture, user interface design, and implementation strategy, including the use of Flutter, Android Studio, and GitHub for collaboration. The document also includes diagrams to illustrate the system's structure and interactions, ensuring clarity for all stakeholders. By detailing these aspects, the SDD aims to provide the team members and other stakeholders with a comprehensive understanding of the application's design and facilitate successful implementation and maintenance.

1.2 Purpose

The purpose of "WakeUP" is to revolutionize how users interact with alarm clocks by integrating engaging and mentally stimulating challenges into their morning routines. Traditional alarm clocks often fail to fully awaken users, leading to oversleeping or hitting the snooze button excessively. "WakeUP" addresses this issue by introducing an interactive and user-customizable solution that ensures users are both mentally and physically prepared to start their day. By requiring users to solve puzzles, math problems, or trivia questions to snooze or turn off their alarms, "WakeUP" minimizes the risk of users falling back to sleep and promotes an active wake-up process.

This project is designed to cater to users who struggle with grogginess or lack of motivation upon waking up. By incorporating a variety of engaging challenges and adjustable difficulty levels, "WakeUP" not only functions as a reliable alarm but also fosters cognitive engagement. Users can tailor their experience to suit their preferences, making the app adaptable for individuals with diverse lifestyles and schedules. Additionally, features like customizable snooze settings and prebuilt purpose-driven options (e.g., setting alarms for workouts or meetings) add convenience and efficiency to daily routines.

From a software engineering perspective, "WakeUP" serves as a platform to explore and implement advanced mobile application design principles. The project demonstrates how Flutter's cross-platform capabilities and Android Studio's robust environment can be leveraged to create a visually appealing and user-friendly application. Furthermore, the integration of GitHub for version control ensures seamless collaboration among team members while maintaining a clean, traceable codebase. Ultimately, "WakeUP" aims to deliver a unique and practical solution to a common problem, showcasing innovative design and thoughtful user-centric functionality.

1.3 Scope

The scope of the "WakeUP" project encompasses the design and development of a user-centric alarm clock application that combines traditional alarm functionalities with innovative features for an engaging wake-up experience. The app allows users to set and manage multiple alarms, customize snooze durations, and toggle alarms on or off with ease. Additionally, the app introduces unique puzzle-based challenges, including captcha puzzles, simple math problems, and trivia questions, which users must solve to snooze or turn off their alarms. These features aim to ensure users are mentally alert before disabling the alarm, reducing oversleeping and promoting productive mornings.

"WakeUP" is designed for a diverse range of users, with a primary focus on individuals who struggle to wake up on time or require structured routines to manage their day effectively. It caters to professionals with tight schedules, students with varying wake-up times, and anyone seeking a smarter alternative to traditional alarm clocks. By providing options to adjust puzzle difficulty levels and prebuilt settings based on alarm purposes (e.g., morning exercise or meetings), the app aims to offer a personalized and adaptive experience that aligns with each user's lifestyle and needs.

While "WakeUP" aims to deliver a robust and innovative alarm solution, certain functionalities are intentionally excluded from the project scope. These include integrations with wearable devices, cloud-based synchronization across multiple platforms, and voice-controlled alarm management. By narrowing the focus, the project ensures the delivery of a high-quality core application within the constraints of time and resources. Future iterations or updates may explore these advanced features as part of the app’s evolution.

1.4 Reference Material

Refer to Reference section 12

1.5 Definitions and Acronyms

Refer to Glossary section 11

2. System Overview

The "WakeUP" alarm app is designed using a modular and scalable system architecture that separates concerns into three primary components: Model, View, and Controller (MVC). This architecture ensures maintainability, flexibility, and ease of development.

2.1 System Architecture

2.1.1 Model

The Model represents the core data of the application, handling alarm settings, user preferences, and puzzle configurations. It stores the time set for each alarm, the type of puzzle selected (captcha, math, or trivia), and the difficulty level of the challenge (easy, medium, high). It also tracks the timeout settings, ensuring a seamless user experience by updating the system state as the user interacts with the app. The Model also interacts with local storage to save alarm data and user preferences persistently.

2.1.2 View

The View is responsible for presenting the user interface of the "WakeUP" app, displaying the alarm list, puzzle challenges, and interactive controls such as the toggle button to activate or deactivate alarms. The View is designed to be intuitive and visually engaging, making it easy for users to manage their alarms, set preferences, and solve puzzles. Flutter's widget-based framework allows for a consistent and responsive UI across multiple devices and screen sizes, ensuring that users have a smooth experience on both smartphones and tablets.

2.1.3 Controller

The Controller acts as the intermediary between the Model and the View. It processes user interactions, such as setting alarms, selecting puzzle types, or changing difficulty levels, and updates the Model accordingly. The Controller also handles business logic related to the alarm’s activation, snoozing, and turning off, ensuring that the correct puzzle challenge is presented when the alarm goes off. It is responsible for triggering notifications and managing the timing of alarms, including the timeout mechanism for when a user fails to solve the puzzle. The Controller ensures that all logic is efficiently executed without burdening the UI, allowing for a responsive experience.

2.2 Design Rationale

We chose the Model-View-Controller (MVC) architecture for the "WakeUP" alarm app due to its simplicity, scalability, and clear separation of concerns. MVC divides the app into three components: the Model (data handling), the View (UI), and the Controller (business logic). This separation ensures that each component can be developed and maintained independently, making the code easier to manage and scale as the app evolves.

We considered other architectures, such as MVVM (Model-View-ViewModel) and MVP (Model-View-Presenter). While MVVM offers strong data binding and better separation of UI logic, it introduces unnecessary complexity for the relatively simple "WakeUP" app. MVP, although offering additional testability, requires a Presenter layer that adds complexity for our needs. MVC was chosen as it is straightforward and well-suited for apps with simpler user interactions.

Key trade-offs included balancing simplicity and flexibility. While MVVM provides more decoupling, the overhead was not justified for this app. MVC offers enough maintainability and scalability while keeping the development process simple and efficient for our current requirements.

2.3 Technology Stack

The "WakeUP" app is built using a modern and powerful technology stack that ensures a seamless user experience and efficient development.

2.3.1 Flutter:

Flutter is chosen as the cross-platform framework for building the app's frontend, enabling the development of a visually consistent and responsive UI across both Android and iOS platforms. Flutter's widget-based approach allows for flexible and reusable UI components, significantly reducing the time and effort required for development.

2.3.2. Dart:

Dart, the programming language used with Flutter, is selected for its optimal performance, rich feature set, and tight integration with the Flutter framework. Dart's strong typing and asynchronous programming capabilities are ideal for building responsive applications, particularly when handling tasks such as managing alarms, user input, and background notifications.

2.3.3 Android Studio:

Android Studio serves as the primary integrated development environment (IDE) for the project. It provides robust support for Flutter and Dart, enabling efficient development, debugging, and testing of the app. With tools like real-time code analysis, device emulators, and seamless integration with GitHub, Android Studio ensures a streamlined development process and reliable deployment of the application.

This combination of technologies ensures that "WakeUP" is both performant and maintainable, providing a solid foundation for future enhancements and updates. The chosen stack also allows for flexibility in extending the app's functionality, making it adaptable to future user needs or feature requests.

3. Design Considerations

In designing the "WakeUP" alarm app, several important considerations were taken into account to ensure the application's effectiveness, usability, and long-term maintainability. These considerations span a variety of areas, including user experience, performance, flexibility, and the technical constraints of the chosen platform.

3.1 User Experience

The "WakeUP" app prioritizes a user-friendly interface for seamless interaction. Key design decisions include a clean layout, intuitive alarm management, and engaging puzzle features. To balance engagement

and usability, users can select puzzles with adjustable difficulty levels (easy, medium, high), ensuring a customizable experience that caters to various preferences.

3.2 Performance

Efficient alarm handling and puzzle loading are crucial. The app is optimized to manage multiple alarms and background processes while minimizing battery usage. Flutter was selected for its cross-platform performance, enabling smooth operation across a wide range of devices.

3.3 Flexibility and Scalability

The app is designed with scalability in mind. The MVC architecture allows easy integration of future features, such as new puzzle types or external device support. The modular structure ensures that updates or modifications can be implemented without affecting existing functionality.

3.4 Security

All user data, including alarm settings and preferences, is stored locally to prioritize privacy. Best practices for mobile security, such as secure data storage and permissions handling, have been followed to ensure the app remains secure against potential threats.

3.5 Accessibility

Accessibility features include high-contrast colors, large buttons, and support for screen readers to make the app usable for individuals with visual or motor impairments. These design choices aim to make the app inclusive for a diverse user base.

4. Functional Design

The Functional Design section outlines the core functionalities and workflows of the WakeUP app, detailing how different components of the system interact to deliver the intended user experience. This section also specifies the logic governing critical features like alarm scheduling, puzzle-solving mechanisms, and customization options.

4.1 Core Functionalities

The WakeUP app is designed to offer the following primary functionalities:

1. **Alarm Management:**
    a. Users can create, update, toggle, or delete alarms from a centralized interface.
    b. Alarms are stored with configurable options, such as time, puzzle type (Captcha, Simple Math), and difficulty level.

    c. An intuitive UI guides users in setting recurring or one-time alarms.
2. **Snooze and Turn-off Features:**
    a. To snooze or turn off an alarm, users must solve a puzzle based on their preselected difficulty level.
    b. A timeout mechanism limits the number of retries, prompting the user to take decisive action.
    c. Multiple attempts to solve a puzzle trigger customized messages to motivate users.
3. **Puzzle Integration:**
    a. Puzzles are dynamically generated based on user preferences.
    b. Difficulty settings impact the complexity of math problems, the type of Captcha, or the trivia questions presented.
    c. Immediate feedback is provided, guiding the user to either retry or proceed.
4. **Accessibility Features:**
    a. Text-to-speech functionality ensures alarms are inclusive for visually impaired users.
    b. Adjustable font sizes and high-contrast themes enhance usability for all users.

4.2 User Interaction Workflow

The app employs a streamlined workflow to guide users through alarm creation, activation, and deactivation. Below is a simplified breakdown:

1. **Alarm Creation:** Users navigate to the alarm setup page, input the desired time, choose puzzle type and difficulty, and save the alarm.
2. **Alarm Trigger:** At the set time, the alarm rings, displaying the snooze and stop options.
3. **Puzzle Resolution:** Users solve the presented puzzle to perform their desired action (snooze or stop). Successful resolution permits the action, while incorrect attempts prompt retries.
4. **Post-Alarm Feedback:** The app confirms successful actions and logs alarm activity for user review.

4.3 Functional Flowchart

The overall process flow of the WakeUP app is visually represented by the flowchart below:

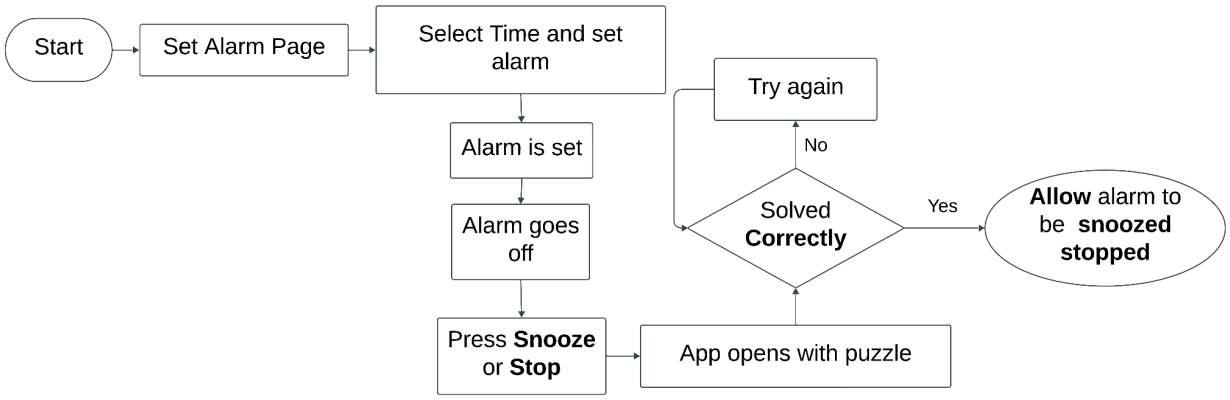


Figure: Flowchart demonstrating the process of the WakeUP app

4.4 Error Handling and Edge Cases

The app incorporates robust mechanisms to handle edge cases and minimize user frustration:

- **Missed Alarms:** Alarms that are not interacted with are logged as missed. The app provides reminders for critical alarms (e.g., for medication or appointments).

- **Puzzle Timeout:** If users fail to solve the puzzle within the set retries, the app escalates with louder alarm tones or motivational messages.

- **Data Validation:** During alarm setup, the app validates user inputs (e.g., ensuring alarm times are not overlapping).

4.5 Future Functional Enhancements

To enhance functionality in subsequent versions, the following features may be considered:

- Cloud synchronization for alarm settings across devices.
- Integration with wearable devices (e.g., smartwatches) to vibrate alongside the main alarm.
- Advanced analytics to provide insights into sleeping and waking patterns.
-

5. Non-functional Requirements

The Non-functional Requirements specify the quality attributes, constraints, and system behaviors that the WakeUP app must satisfy to ensure usability, reliability, and performance. These requirements focus on how the system operates rather than the specific tasks it performs.

5.1 Usability

5.1.1 User-Friendly Interface:

a. The app must have an intuitive and visually appealing UI that accommodates both novice and advanced users.
b. Clear labels, buttons, and navigation paths must be provided for all functionalities.

5.1.2 Accessibility:

c. Features such as adjustable font sizes, and high-contrast themes must be incorporated for users with visual impairments or other disabilities.
d. Haptic feedback for alarms ensures inclusivity for users with hearing impairments.

5.2 Performance

5.2.1 Response Time:

a. The app must execute user actions (e.g., setting alarms, snoozing, solving puzzles) with a response time of less than **2 seconds** under normal conditions.
b. Alarm triggers and puzzle displays must occur within **1 second** of the set time.

5.2.2 Efficiency:

c. The app should consume minimal system resources, ensuring smooth operation even on devices with limited processing power or memory.
d. Background processes, such as alarm triggers, must be lightweight to avoid impacting device performance.

5.3 Reliability

5.3.1 Alarm Accuracy:

a. The system must guarantee 100% reliability in alarm triggering at the specified time.
b. Time zone changes or device restarts must not affect the alarm schedule.

5.3.2 Fault Tolerance:

c. The app must handle unexpected failures, such as crashes, by preserving alarm configurations and ensuring alarms trigger as expected.

5.3.3 Data Integrity:

d. All user settings, preferences, and alarm logs must be securely stored and remain intact in the event of system updates or interruptions.

5.4 Security

5.4.1 User Privacy:

a. The app must comply with data protection regulations, such as **GDPR** and **CCPA**, by safeguarding user information and ensuring no data is shared without consent.
b. Puzzle-related data, such as trivia answers or user preferences, must remain encrypted and inaccessible to unauthorized parties.

5.4.2 Secure Data Storage:

c. All data, including alarm configurations and user settings, must be stored locally using industry-standard encryption techniques.
d. Sensitive information, if transmitted (e.g., for cloud backup in future versions), must utilize secure protocols like **HTTPS** or **TLS**.

5.5 Scalability

- The app must support an increasing number of alarms and puzzle configurations without degradation in performance.
- It must be designed to accommodate potential feature expansions, such as cloud synchronization and wearable integrations, without significant architectural changes.

5.6 Portability

5.6.1 Cross-Platform Compatibility:

a. The app must operate seamlessly on both Android and iOS devices, providing a consistent user experience across platforms.
b. Flutter's cross-platform capabilities ensure the same codebase works efficiently on different operating systems.

5.6.2 Device Support:

c. The app must support devices with a wide range of screen sizes and resolutions, including smartphones and tablets.

5.7 Maintainability

5.7.1 Code Quality:

a. The app's codebase must follow clean coding practices, including modular design, detailed comments, and adherence to naming conventions.
b. Comprehensive documentation must accompany the codebase to simplify future development and debugging.

5.7.2 Bug Fixes and Updates:

c. The system must allow seamless updates for new features or bug fixes without impacting existing functionalities.
d. An error reporting mechanism must capture crash logs for developers to diagnose and resolve issues effectively.

5.8 Extensibility

- The system should be designed with extensibility in mind, enabling easy addition of features such as new puzzle types, advanced alarm configurations, or integrations with other applications.

5.9 Ethical Considerations

- The app must not exploit user data or include intrusive advertisements that compromise the user experience.
- Gamification elements (e.g., puzzle-solving challenges) must avoid promoting unhealthy habits, such as over-reliance on snooze functionality.

6. System Design

The System Design section outlines the high-level architecture and components of the WakeUP app. It describes the organization of software components, interactions, and the design principles employed to achieve the functional and non-functional requirements.

6.1 High-Level Architecture

The app follows a Model-View-Controller (MVC) architecture to ensure modularity, scalability, and maintainability.

6.1.1 Model

A. Responsible for managing the data and business logic of the application.
B. Stores alarm configurations, puzzle data (questions, answers, and difficulty levels), and user preferences.
C. Interfaces with local storage to save and retrieve data persistently.

6.1.2 View

D. Comprises the user interface (UI) elements that allow users to interact with the system.
E. Displays pages for setting alarms, solving puzzles, and managing user preferences.
F. Designed using Flutter to provide a consistent and responsive UI across platforms.

6.1.3 Controller

G. Acts as the intermediary between the model and view.
H. Handles user inputs (e.g., setting alarms or solving puzzles) and updates the model and view accordingly.
I. Implements core functionalities like scheduling alarms, triggering puzzles, and validating user responses.

6.2 Key Components

The WakeUP app consists of several interdependent components to deliver the desired functionality:

1. **Alarm Management Module**
    a. Enables users to create, update, delete, and toggle alarms.
    b. Triggers alarms at the specified time, incorporating logic for time zone adjustments and device restarts.

2. **Puzzle Module**
    a. Provides three puzzle options (Captcha, Math Problems, Trivia) to turn off or snooze the alarm.

b. Dynamically adjusts puzzle difficulty based on user preferences and snooze patterns.

3. **Notification Module**
    a. Integrates with the device's notification system for timely alerts.

4. **Settings Module**
    a. Allows users to configure app behavior, including default snooze durations, puzzle preferences, and notification settings.
    b. Saves user settings persistently using local storage.

5. **Persistence Module**
    a. Ensures alarm data, user preferences, and puzzle configurations are securely stored in the device's local storage.
    b. Leverages Flutter's built-in libraries for data serialization and storage management.

6.3 Data Flow

The system ensures smooth communication between components through defined data flows:

6.3.1 Alarm Trigger Flow:

a. User sets an alarm → Data is stored in the **Model** → Alarm triggers at the set time → **Controller** fetches the puzzle and updates the **View** to display it.

6.3.2 Puzzle Solving Flow:

b. Alarm triggers puzzle → User inputs solution → **Controller** validates the solution → Updates **Model** with results and adjusts difficulty.

6.3.3 Settings Update Flow:

c. User modifies preferences in the settings menu → **Controller** updates the **Model** with new preferences → **View** reflects updated settings.

6.4 System Interactions

The interactions between users and the system are managed as follows:

1. **User Interaction**:
    a. Users set alarms and customize settings through the intuitive UI.
    b. Puzzle interactions are designed to be engaging yet straightforward, ensuring quick user responses.
2. **System Processing**:
    a. The **Controller** handles user inputs, triggers alarms, and processes puzzle solutions.

b. The **Model** maintains data consistency, while the **View** ensures a seamless visual experience.

3. **External Interactions**:
    a. The app interacts with the device's operating system to access notifications, alarms, and system resources (e.g., vibration for alarm alerts).

6.5 System Constraints

The design considers the following constraints to ensure optimal performance:

1. **Hardware Constraints**:
    a. Limited processing power and memory on older devices may affect performance.

2. **Platform Constraints**:
    a. Differences in alarm and notification APIs across Android and iOS must be accounted for to maintain cross-platform functionality.

3. **User Constraints**:
    a. The app must accommodate diverse user preferences and accessibility requirements.

6.6 Trade-offs and Design Choices

6.6.1 Technology Stack:

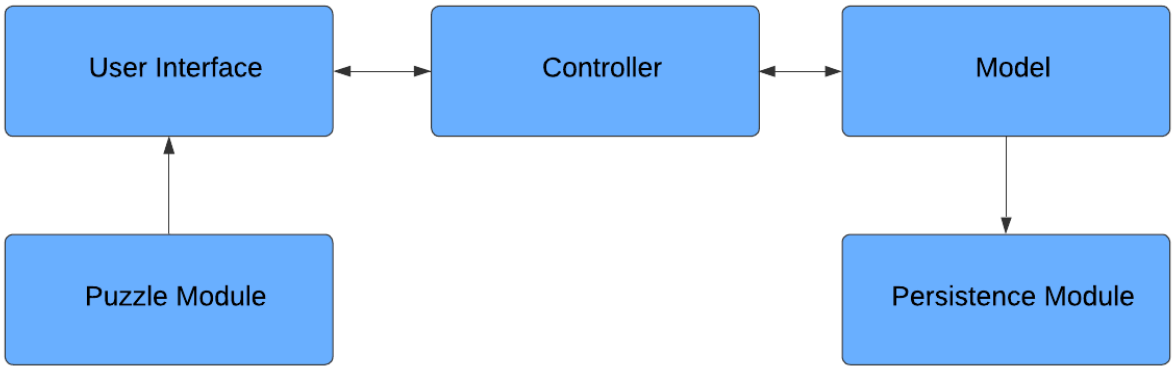
a. Flutter was chosen for its cross-platform capabilities, despite the steeper learning curve for new developers.
b. Local storage is used instead of cloud storage to simplify initial implementation, trading off multi-device synchronization.

6.6.2 Puzzle Functionality:

c. The app prioritizes simplicity and engagement for puzzles, trading off advanced gamification features to reduce development complexity.

6.7 System Design Diagram

To better understand the architecture, the following diagram visualizes the relationships between the key components:



7. Interface Design

The **Interface Design** section of the *WakeUP* app emphasizes usability, aesthetics, and the intuitiveness of user interactions, ensuring that users can easily navigate and utilize the app's features.

7.1 Key Elements of Interface Design

1. Main Clock Page

a. **Description**: The main clock interface prominently displays the current time, date, and timezone. A clean, minimalist design is adopted for easy readability and a modern feel.

b. **Features**:

i. A digital clock with large fonts for enhanced visibility.

ii. Sidebar navigation to quickly access the "Alarm," "Timer," and "Stopwatch" functionalities.

2. Alarm Setup Page

a. **Description**: The alarm setup page provides a simple and direct way for users to set and manage alarms.

b. **Features**:

i. A floating action button (FAB) with a plus sign (+) to add a new alarm.

ii. When adding an alarm, users are prompted with a modal form to:

1. **Enter Alarm Title**: Users can name the alarm (e.g., "Morning Exercise").
2. **Set Date and Time**: A calendar date picker and an intuitive time picker (with an analog clock interface for time selection).

iii. Alarms are displayed as cards, each showing:

1. Alarm title.
2. Time and date.
3. Toggle switch to enable or disable the alarm.

3. **Notification System**
    a. **Description**: When the alarm goes off, a notification bar displays the scheduled alarm title and body.
    b. **Features**:
        i. Quick access to snooze or stop the alarm via the notification.

4. **Puzzle Challenge Page**
    a. **Description**: To stop or snooze the alarm, the user must solve a puzzle. This unique feature ensures that the user is alert.
    b. **Features**:
        i. Difficulty Level Selection: Easy, Medium, or Hard.
        ii. Puzzle Prompt: Users are presented with a question (e.g., "What is 6 x 7?").
        iii. Answer Input: A text field for users to type their answer.
        iv. Feedback:
            1. Correct Answer: Displays a confirmation message ("The alarm has been stopped").
            2. Incorrect Answer: Prompts the user to "Try again" with the same or a new question.

5. **Styling and Aesthetic Choices**
    a. **Color Scheme**:
        i. A dark theme with shades of purple and white for a sleek, modern look.
        ii. High contrast between text and background to enhance readability.
    b. **Typography**:
        i. Large and legible fonts for time and alarm information.
        ii. Clear labels for buttons and actions.
    c. **Icons**:
        i. Intuitive icons (e.g., clock, alarm, FAB) for better navigation.

7.2 Key User Scenarios

1. **Setting an Alarm**
    a. Users navigate to the "Alarm" section from the sidebar, tap the "Add Alarm" button, and input the desired time, date, and alarm title.
    b. Upon saving, the new alarm appears in the alarm list with a toggle switch.

2. **Responding to an Alarm**
    a. When the alarm goes off, the user receives a notification and must solve a puzzle within the app to snooze or stop the alarm.
    b. The interface ensures clarity by showing the puzzle question and providing instant feedback.

3. **Customizing Alarm**
    a. Users can easily manage alarms by toggling them on/off or editing details.

4. **Navigating Between Features**
    a. The sidebar allows quick switching between the main clock, alarm, timer, and stopwatch functionalities.

7.3 Improvements Based on Feedback

The current design emphasizes clarity and simplicity; however, additional refinements could include:

- Enhanced animations for transitions between pages.
- Tooltips to guide new users.
- The ability to customize puzzle categories (e.g., Trivia, Math, Captcha) directly from the settings page.

7.4 Flowchart Representation

The flow of user interactions aligns with the app's workflow, as depicted in the earlier-provided flowchart. Each interaction reflects an intuitive sequence, ensuring users can achieve their goals without confusion.

7.5 Screenshots of the current interface

The provided screenshots effectively illustrate the current interface and serve as visual aids to explain the app's functionality.

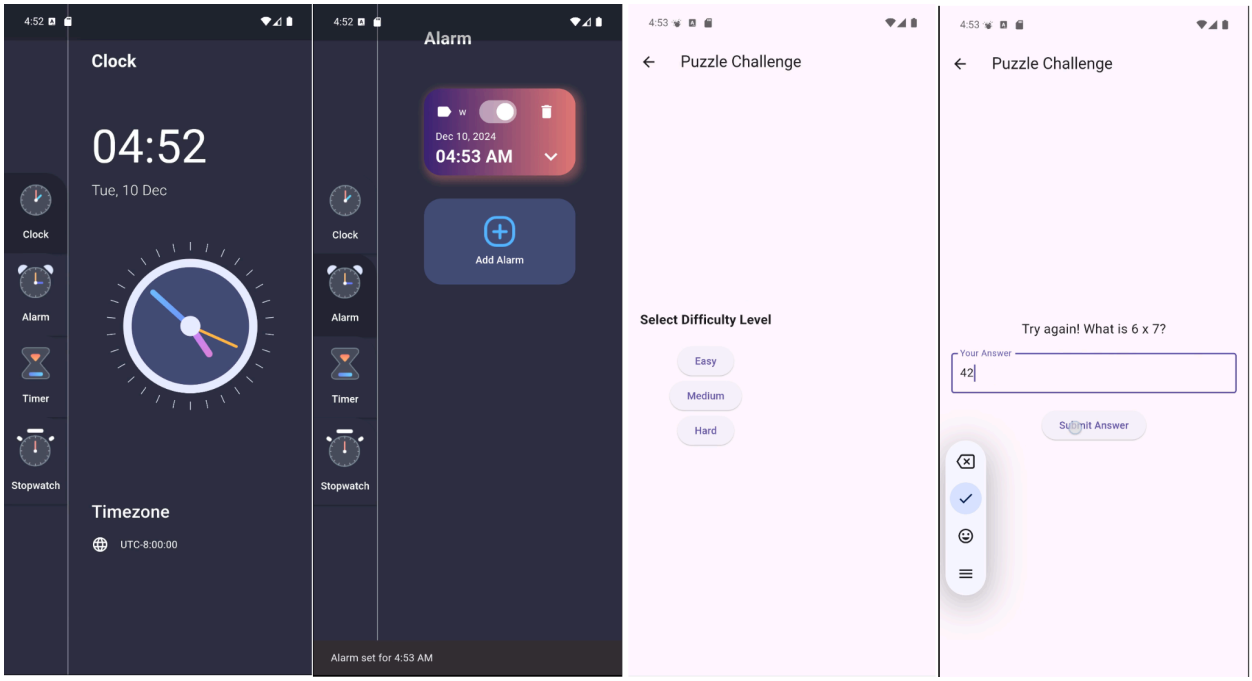


Figure: WakeUP app interface

2.4 Development Methods

The **WakeUP** project utilized a hybrid **Agile Development** approach tailored to meet the project's requirements and timeline constraints.

Chosen Method: Agile Development

- **Description**: Agile methodology emphasizes iterative development, continuous feedback, and flexibility to adapt to changing requirements.
- **Application**:
    - The project was divided into sprints, with each sprint focusing on specific deliverables, such as UI development, puzzle integration, and alarm functionality.
    - Regular team meetings facilitated collaboration, identified blockers, and allowed for iterative refinement of features.
    - Continuous testing was conducted at the end of each sprint to ensure progress aligned with requirements.

Reasons for Choosing Agile

- **Iterative Refinement**: Agile allowed the team to develop and refine features incrementally, ensuring a functional product by the delivery deadline.
- **Collaboration**: Frequent check-ins and code reviews using GitHub enhanced team collaboration and maintained code quality.
- **Flexibility**: Agile accommodated changes in scope, such as refining puzzle logic or adjusting UI elements based on stakeholder feedback.

Alternative Methods Considered

1. **Waterfall Development**
    - **Reason for Rejection**: The linear and rigid structure of Waterfall would not have allowed flexibility for scope changes or iterative improvements, which were critical in this project.
2. **Unplanned Mad Scramble Development**
    - **Reason for Rejection**: A disorganized approach could have jeopardized the delivery deadline and compromised quality.

3. Architectural Strategies

3.1 Use of Specific Products

- **Programming Language**: **Dart** was chosen for its tight integration with **Flutter**, ensuring optimal performance and faster development cycles.
    - **Reason**: Dart's asynchronous programming model is ideal for managing tasks like alarm scheduling and user input handling.
    - **Alternative Considered**: Java/Kotlin was considered but rejected due to the added complexity of platform-specific development. Flutter's cross-platform capabilities simplified development for both Android and iOS.
- **Database**: **Local storage with SQLite** was selected for storing alarm configurations and user preferences.

    - **Reason**: SQLite provides lightweight, efficient, and reliable local storage without requiring complex database management.
    - **Alternative Considered**: Cloud storage was considered but deferred to future versions due to time constraints and offline usability requirements.

3.2 Reuse of Existing Software Components

- **Component Reuse**: Flutter widgets and libraries were leveraged to streamline UI development and improve maintainability.
    - **Reason**: Flutter's widget library allowed rapid prototyping of the user interface while ensuring responsiveness across devices.
    - **Trade-off**: Dependency on Flutter-specific components may require additional effort to adapt if switching to a different framework in the future.

3.3 Future Plans for Enhancements

- **Cloud Synchronization**: Plans include extending the system to support cloud-based alarm synchronization for multi-device access.
    - **Impact**: The current architecture includes modular storage handling to simplify future integration of external APIs for cloud services.
- **Wearable Device Integration**: Adding support for alarm notifications on wearables (e.g., smartwatches).
    - **Impact**: Modular design ensures the alarm trigger logic can be adapted for external devices without significant codebase changes.

3.4 User Interface Paradigms

- **Paradigm**: Intuitive, widget-based UI using Flutter, with material design elements.
    - **Reason**: Ensures a familiar and visually appealing experience across Android and iOS platforms.
    - **Alternative Considered**: Custom-built UI frameworks were rejected as they would increase development time and reduce maintainability.

3.5 Error Detection and Recovery

- **Strategy**: Robust error handling mechanisms were implemented to address edge cases such as failed alarm triggers or invalid user inputs.
    - **Examples**:
        - Alarms that fail due to time zone changes or device restarts are logged and rescheduled.
        - Users receive clear feedback on invalid input during alarm setup.
    - **Trade-off**: Increased development time for implementing fallback mechanisms but improved reliability.

3.6 Memory Management Policies

- **Strategy**: Efficient use of resources by loading puzzles and alarm configurations on demand.
    - **Reason**: Minimizes memory usage on lower-end devices, ensuring smooth performance.

3.7 Data Storage Management

- **Approach**: Persistent local storage via SQLite, ensuring user settings and alarm data are retained across sessions.
    - **Reason**: Eliminates dependency on network availability and improves app reliability.
    - **Trade-off**: Cloud synchronization capabilities were deprioritized for this version due to development constraints.

3.8 Concurrency and Synchronization

- **Strategy**: Dart's asynchronous programming model was used for alarm triggers and puzzle handling to ensure non-blocking operations.
    - **Reason**: Improves responsiveness and ensures smooth user interactions even during background processes.

3.9 Communication Mechanisms

- **Approach**: Native APIs for managing alarms and notifications on Android and iOS.
    - **Reason**: Native integration ensures reliable and timely alarm triggers across platforms.

3.10 Generalized Approaches to Control

- **Architecture**: **Model-View-Controller (MVC)** ensures modularity and clear separation of concerns.
    - **Reason**: Facilitates maintainability and scalability by isolating business logic, data management, and UI components.

5. Policies and Tactics

5.1 Choice of which specific products used

- **Policy**: Use of **Flutter** with **Dart** as the core development framework and language.
    - **Reason**:
        - Cross-platform development with Flutter ensures consistent UI and behavior across Android and iOS.
        - Dart's asynchronous capabilities facilitate responsive alarm management and puzzle interaction.
    - **Alternative Considered**: Native development using Java/Kotlin (Android) and Swift (iOS).
        - **Rejected**: Native development would require separate codebases, doubling the development effort and maintenance.
- **Policy**: Use of **SQLite** for local data storage.
    - **Reason**:
        - SQLite offers lightweight and persistent storage for alarm configurations and user preferences, ensuring offline functionality.
    - **Alternative Considered**: Firebase Realtime Database or Cloud Firestore.
        - **Rejected**: Cloud storage was deemed unnecessary for the initial version and would complicate offline usability.

5.2 Plans for ensuring requirements traceability

Policy: Maintain comprehensive documentation and use version control for all requirements and changes.

- **Implementation**:
  - Requirements are tracked using a shared document on a collaboration platform (e.g., GitHub).
  - Commits in the version control system reference specific requirements, ensuring traceability from implementation to design goals.
- **Reason**:
  - Ensures that all features and bug fixes align with the specified requirements.
- **Alternative Considered**: Manual tracking through meeting notes or spreadsheets.
  - **Rejected**: Prone to human error and lack of automation in linking requirements to implementation.

5.3 Plans for testing the software

Policy: Employ automated and manual testing for all critical features.

- **Automated Testing**:
  - Unit tests for individual modules, such as alarm scheduling and puzzle validation.
  - Integration tests to ensure seamless communication between Model, View, and Controller.
- **Manual Testing**:
  - Usability testing for the UI to validate responsiveness and accessibility.
  - Edge case testing, such as device restarts or time zone changes, to ensure alarm reliability.
- **Reason**:
  - A combination of testing methods ensures both functional correctness and user satisfaction.
- **Alternative Considered**: Relying solely on manual testing.
  - **Rejected**: Inefficient and less reliable for covering all potential edge cases.

5.4 Coding Guidelines and Conventions

- **Policy**: Adhere to Dart's best practices and style guide.
  - **Reason**:
    - Improves code readability and maintainability.
    - Facilitates collaboration among team members by enforcing a consistent coding style.
  - **Alternative Considered**: Allowing developers to use personal coding styles.
    - **Rejected**: Could lead to inconsistent code that is harder to review and maintain.

5.5 Error Detection and Recovery

- **Policy**: Implement robust error-handling mechanisms for common issues.
  - **Examples**:
    - Retry mechanisms for puzzle solutions to ensure smooth user experience.
    - Logs for missed alarms or invalid user inputs to identify and address issues.

    - **Reason**:
        - Enhances app reliability and user satisfaction.
    - **Alternative Considered**: Minimal error handling with simple error messages.
        - **Rejected**: Lacked user guidance and reduced app reliability.

5.6 User Interface Considerations

- **Policy**: Design a responsive, accessible UI with Flutter's widget-based approach.
    - **Reason**:
        - Ensures compatibility across devices with varying screen sizes and resolutions.
        - Meets accessibility standards with high-contrast themes, adjustable fonts, and screen reader support.
    - **Alternative Considered**: Static UI design with fixed layouts.
        - **Rejected**: Fails to adapt to different devices and screen sizes.

5.7 Plans for Maintaining the Software

- **Policy**: Modular design with clear separation of concerns (MVC architecture).
    - **Reason**:
        - Simplifies future updates and bug fixes by isolating data, logic, and presentation layers.
    - **Alternative Considered**: Monolithic design with tightly coupled components.
        - **Rejected**: Increased complexity for maintaining and scaling the system.

5.8 Hierarchical Organization of Source Code

- **Policy**: Organize code into directories corresponding to MVC layers.
    - **Reason**:
        - Facilitates easy navigation and management of source code.
    - **Structure**:
        - /models: Contains data handling and business logic.
        - /views: Houses all UI-related components.
        - /controllers: Implements logic for coordinating between models and views.

6. Detailed System Design

6.1 Name of Component (Module)

6.1.1 Responsibilities

- Handle creation, storage, updating, toggling, and deletion of alarms.
- Schedule alarms based on user-specified configurations (e.g., time, repetition).
- Trigger alarms at the designated time and initiate the corresponding puzzle challenges.
- Ensure alarms persist across sessions, even after device restarts or time zone changes.

6.1.2 Constraints

- **Timing Constraints**: Alarms must trigger within 1 second of the set time.
- **Storage Constraints**: Limited local storage requires efficient handling of alarm data.
- **Platform Constraints**: Different APIs for Android and iOS necessitate platform-specific adaptations.

6.1.3 Composition

- **Alarm Scheduler**: Handles timing and triggering of alarms.
- **Data Persistence**: Stores alarm configurations using SQLite.
- **Notification Handler**: Sends notifications when alarms are triggered.

6.1.4 Uses/Interactions

- Collaborates with the **Puzzle Module** to fetch and display the puzzle challenge upon alarm trigger.
- Interacts with the **Settings Module** to apply user preferences (e.g., snooze duration).
- Interfaces with device notification systems for alert delivery.

6.1.5 Resources

Memory: Stores active alarms in memory for quick access.

SQLite Database: Persists alarm configurations and user settings.

Notification API: Manages system-level alarm and notification behaviors.

6.1.6 Interface/Export

- **createAlarm(time, config)**: Adds a new alarm with specified time and configuration.

- **deleteAlarm(id)**: Removes an alarm by its identifier.
- **triggerAlarm(id)**: Activates the alarm and initiates the puzzle challenge

6.2 Puzzle Module

6.2.1 Responsibilities

- Generate and display puzzles (e.g., Captcha, math problems, trivia) based on user settings.
- Validate user responses and communicate results to the Alarm Management Module.
- Adjust difficulty based on user performance or preferences.

6.2.2 Constraints

- **Dynamic Puzzle Generation**: Puzzles must load instantly when the alarm triggers.
- **Validation**: Must accurately and efficiently verify user solutions.

6.2.3 Composition

- **Puzzle Generator**: Creates puzzles dynamically based on type and difficulty.
- **Validation Engine**: Checks user input against correct answers.
- **UI Renderer**: Displays puzzles within the app interface.

6.2.4 Uses/Interactions

- Receives puzzle type and difficulty from the **Settings Module**.
- Sends validation results to the **Alarm Management Module** to determine snooze or stop actions.

6.2.5 Resources

- **CPU**: For generating and validating puzzles.
- **Local Storage**: Stores predefined trivia questions and answers.
- **UI Framework**: Renders puzzles on the user interface.

6.2.6 Interface/Exports

- **generatePuzzle(type, difficulty)**: Produces a puzzle of the specified type and difficulty.
- **validateAnswer(puzzleId, userInput)**: Validates the user's response to the puzzle.

6.3 Settings Module

6.3.1 Responsibilities

- Allow users to configure app behavior, including alarm preferences, default snooze duration, and puzzle difficulty.
- Persist user preferences across app sessions.
- Provide a centralized interface for managing settings.

6.3.2 Constraints

- **Platform Differences**: Must ensure consistent behavior across Android and iOS.
- **Storage Efficiency**: Optimize storage for user preferences to avoid bloating local databases.

6.3.3 Composition

- **Settings Manager**: Manages user preferences.
- **UI Renderer**: Displays settings options to the user.

6.3.4 Uses/Interactions

- Interacts with the **Alarm Management Module** to apply default snooze durations.
- Sends puzzle type and difficulty to the **Puzzle Module**.

6.3.5 Resources

- **SQLite Database**: Stores user preferences.
- **UI Framework**: Provides an interface for configuring settings.

6.3.6 Interface/Exports

- **getPreference(key)**: Retrieves the value of a specific preference.
- **setPreference(key, value)**: Updates a specific user preference.

7. Detailed Lower level Component Design

7.1 Alarm Class

7.1.1 Classification

- **Type:** Data Model Class

- **Purpose:** Represents an individual alarm with properties and methods to manage its lifecycle.

7.1.2 Processing Narrative (PSPEC)

- Stores alarm attributes (e.g., time, label, repeat settings).
- Validates and updates alarm properties.
- Interacts with the Alarm Scheduler to trigger alarms.

7.1.3 Interface Description

- **Attributes:**
  - **id:** Unique identifier for the alarm.
  - **time:** Alarm trigger time.
  - **repeat:** Boolean or array for repeat settings.
  - **label:** User-defined label for the alarm.
  - **puzzleType:** Type of puzzle associated with the alarm.
- **Methods:**
  - **validate():** Ensures alarm attributes are correctly set.
  - **schedule():** Sends the alarm to the scheduler for activation.
  - **cancel():** Removes the alarm from the schedule.

7.1.4 Processing Detail

- **7.1.4.1 Design Class Hierarchy:**
  - **Parent Class:** None (base model).
  - **Child Classes:** None.
- **7.1.4.2 Restrictions/Limitations:**
  - Must handle time zone changes gracefully.
  - Requires unique identifiers for alarms.
- **7.1.4.3 Performance Issues:**
  - Efficient scheduling for alarms with frequent repetitions.
- **7.1.4.4 Design Constraints:**
  - Must persist data locally using SQLite.
- **7.1.4.5 Processing Detail For Each Operation:**
  - **validate():** Checks for valid time format and non-empty label**.**
  - **schedule():** Uses system APIs to register alarms.

7.2 Puzzle Class

7.2.1 Classification

- **Type:** Functional Class
- **Purpose:** Generates and validates puzzles presented to the user.

7.2.2 Processing Narrative (PSPEC)

- Dynamically creates puzzles based on user preferences and difficulty.
- Validates user inputs against correct answers.
- Adjusts difficulty for subsequent puzzles.

7.2.3 Interface Description

- **Attributes:**
  - **id:** Unique identifier for the puzzle.
  - **type:** Type of puzzle (e.g., Captcha, Math).
  - **difficulty:** Puzzle difficulty level.
  - **question:** The generated puzzle question.
  - **answer:** Correct answer to the puzzle.
- **Methods:**
  - **generate():** Creates a puzzle question based on type and difficulty**.**
  - **validate(input):** Checks if the user's input matches the answer.

7.2.4 Processing Detail

- **7.2.4.1 Design Class Hierarchy:**
  - **Parent Class:** None (base functional class).
  - **Child Classes:**
    - **CaptchaPuzzle:** Subclass for Captcha puzzles.
    - **MathPuzzle:** Subclass for math problems.
    - **TriviaPuzzle:** Subclass for trivia questions.
- **7.2.4.2 Restrictions/Limitations:**
  - Captcha puzzles must be accessible for visually impaired users.
- **7.2.4.3 Performance Issues:**
  - Dynamic puzzle generation must occur within 1 second.
- **7.2.4.4 Design Constraints:**
  - Must support easy integration of new puzzle types.
- **7.2.4.5 Processing Detail For Each Operation:**
  - **generate():** Uses random number generation or predefined data (for trivia).
  - **validate(input):** Compares user input to the correct answer.

7.3 Settings File

7.3.1 Classification

- **Type:** Configuration File
- **Purpose:** Stores user preferences and app configurations.

7.3.2 Processing Narrative (PSPEC)

- Manages persistent storage of user preferences, such as default snooze duration and puzzle difficulty.

7.3.3 Interface Description

- **Attributes:**
  - **snoozeDuration:** Default duration for snooze (in minutes).
  - **defaultPuzzleType:** Default puzzle type for new alarms.
  - **difficultyLevel:** Default puzzle difficulty.
- **Methods:**
  - **loadSettings():** Loads settings from local storage.
  - **saveSettings():** Updates settings in local storage.

7.3.4 Processing Detail

- **7.3.4.1 Design Class Hierarchy:**
  - **Parent Class:** None (standalone utility).
  - **Child Classes:** None.
- **7.3.4.2 Restrictions/Limitations:**
  - Must handle corrupted or missing configuration files.
- **7.3.4.3 Performance Issues:**
  - Minimal as file access occurs infrequently.
- **7.3.4.4 Design Constraints:**
  - Must use encrypted storage for sensitive preferences.
- **7.3.4.5 Processing Detail For Each Operation:**
  - **loadSettings():** Reads JSON configuration file.
  - **saveSettings():** Serializes preferences to JSON

7.4 Notification Handler

7.4.1 Classification

- **Type:** Functional Component
- **Purpose:** Interfaces with the system to deliver alarms and notifications.

7.4.2 Processing Narrative (PSPEC)

- Schedules, triggers, and manages system-level notifications for alarms.

7.4.3 Interface Description

- **Attributes:**
  - **notificationId:** Unique ID for notifications.
  - **sound:** Customizable alarm sound.
  - **vibrationPattern:** Vibration settings for notifications.
- **Methods:**
  - **triggerNotification(id):** Activates the notification.
  - **cancelNotification(id):** Cancels a scheduled notification.

7.4.4 Processing Detail

- **7.4.4.1 Design Class Hierarchy:**
  - **Parent Class:** None (standalone functional class).
  - **Child Classes:** None**.**
- **7.4.4.2 Restrictions/Limitations:**
  - Platform differences in notification APIs must be abstracted.
- **7.4.4.3 Performance Issues:**
  - Must avoid delays in triggering notifications.
- **7.4.4.4 Design Constraints:**
  - Must integrate seamlessly with native Android and iOS notification systems.
- **7.4.4.5 Processing Detail For Each Operation:**
  - **triggerNotification(id):** Uses platform APIs to create system alerts.
  - **cancelNotification(id):** Removes the notification from the system.

8. Database Design

8.1 Tables and Descriptions

8.1.1 Alarms Table

- **Purpose**: Stores alarm configurations and details.
- **Columns**:
  - **id** (INTEGER, Primary Key, Auto Increment): Unique identifier for each alarm.
  - **time** (TEXT): The time the alarm is set to trigger (e.g., "07:00 AM").
  - **repeat** (TEXT): Repeat settings (e.g., "Daily", "Weekdays").
  - **label** (TEXT): User-defined label for the alarm (e.g., "Morning Workout").
  - **puzzleType** (TEXT): Type of puzzle associated with the alarm (e.g., "Captcha", "Math").
  - **difficulty** (TEXT): Difficulty level of the puzzle (e.g., "Easy", "Medium").
  - **isActive** (INTEGER): Status of the alarm (1 for active, 0 for inactive).

8.1.2 Settings Table

- **Purpose**: Stores user preferences and global app configurations.
- **Columns**:
  - **id** (INTEGER, Primary Key, Auto Increment): Unique identifier for each settings entry.
  - **snoozeDuration** (INTEGER): Default snooze duration in minutes.
  - **defaultPuzzleType** (TEXT): Default puzzle type for newly created alarms.

- **difficultyLevel** (TEXT): Default difficulty level for puzzles.
- **theme** (TEXT): User-selected theme (e.g., "Light", "Dark").
- **notificationsEnabled** (INTEGER): Indicates whether notifications are enabled (1 for yes, 0 for no).

8.1.3 Puzzles Table

- **Purpose**: Stores predefined puzzle data for trivia questions or Captcha challenges.
- **Columns**:
  - **id** (INTEGER, Primary Key, Auto Increment): Unique identifier for each puzzle.
  - **type** (TEXT): Puzzle type (e.g., "Trivia", "Captcha").
  - **question** (TEXT): The puzzle question (e.g., "What is 7 + 5?").
  - **answer** (TEXT): Correct answer to the puzzle.
  - **difficulty** (TEXT): Difficulty level of the puzzle.

8.2 Relationships Between Tables

- **Alarms and Puzzles**:
  - The puzzletable field in the alarms table corresponds to the type field in the puzzles table.
  - Example: If an alarm specifies a puzzle type of "Trivia," the app fetches a trivia question from the puzzles table.
- **Settings and Alarms**:
  - Default values from the settings table (e.g., default puzzle type) are applied when creating new alarms.

9. Requirements Validation and Verification

Requirement	Component Module	UI Element	Testing Method
1. Create and manage alarms	Alarm Management Module	Alarm List, Add/Edit Alarm	- Unit tests for createAlarm and deleteAlarm methods.
			- Manual UI testing to add, update, and delete alarms and verify functionality.
2. Trigger alarms at the specified time	Alarm Management Module	Notification System	- Integration tests to verify timely alarm triggers.
			- Manual testing with

			alarms set across different times and dates.
3. Display puzzle challenges	Puzzle Module	Puzzle Screen	- Unit tests for generatePuzzle and validateAnswer methods.
			- Manual testing for each puzzle type and difficulty level.
4. Validate puzzle solutions	Puzzle Module	Puzzle Screen	- Unit tests for the validateAnswer function to ensure correct validation logic.
			- Edge case testing (e.g., incorrect answers, timeouts).
5. Snooze or stop alarms	Alarm Management Module	Puzzle Screen	- Integration tests to ensure correct snooze or stop functionality after solving puzzles.
			- Manual testing to verify snooze duration and alarm reactivation.
6. Allow customization of puzzle type	Settings Module	Settings Screen	- Unit tests for setPreference and getPreference functions.
			- Manual testing to ensure selected puzzle type is reflected in alarms.
7. Support default snooze duration	Settings Module	Settings Screen	- Unit tests for storing and retrieving default snooze settings.
			- Manual testing to verify default snooze duration is applied to new alarms.
8. Provide accessibility	Settings Module	Settings Screen	- Manual testing for

options			theme changes (light/dark mode) and text resizing.
			- Accessibility testing with screen readers and high-contrast mode.
9. Persist alarms and settings locally	Alarm Management Module,	N/A	- Unit tests for SQLite integration to verify data is stored and retrieved correctly.
	Settings Module		- Manual testing by restarting the app and verifying alarms and settings remain intact.
10. Notify users through system alerts	Notification Module	Alarm Notification	- Unit tests for triggerNotification and cancelNotification functions.
			- Manual testing to verify notifications trigger at the correct time.
11. Provide responsive UI	View Module	All Screens	- Performance testing to measure UI response times for different user actions.
			- Manual testing to ensure smooth navigation and interaction on various devices and screen sizes.
12. Log missed alarms	Alarm Management Module	Alarm List	- Unit tests to verify logging functionality for missed alarms.
			- Manual testing by setting alarms and letting them expire.
13. Handle edge cases (e.g., time zones)	Alarm Management Module	Alarm List	- Unit tests for alarms set across time zone changes.

			- Manual testing by setting alarms in different time zones and verifying correct triggers.
14. Ensure offline functionality	All Modules	All Screens	- Unit tests to simulate no-network conditions and ensure alarms, puzzles, and settings remain functional.
			- Manual testing by enabling airplane mode and verifying core functionalities.

10. Glossary

Alarm: A notification triggered at a specific time to alert the user, typically in the form of sound or vibration, prompting the user to wake up or take action.

Snooze: A feature that temporarily silences the alarm for a specified period, allowing the user to rest for a short time before the alarm rings again.

Timeout: A specified period during which the user is not allowed to turn off or snooze the alarm after failing to solve the puzzle challenge. This prevents the user from easily bypassing the wake-up process.

Captcha: A type of challenge-response test used to verify that the user is human, often involving recognizing distorted text, selecting images, or completing simple puzzles.

Trivia: A form of quiz challenge that asks users general knowledge questions, which they must answer correctly to snooze or turn off the alarm.

Puzzle Challenge: A cognitive task that the user must solve to disable or snooze the alarm, which may include captcha puzzles, math equations, or trivia questions. These challenges help ensure that the user is awake and alert before turning off the alarm.

Dart: A programming language developed by Google, used primarily for building mobile and web applications with Flutter. Dart is known for its performance and scalability, making it ideal for creating high-performance apps.

Flutter: An open-source UI framework developed by Google, used to build natively compiled applications for mobile, web, and desktop from a single codebase. It is used in this project to create a cross-platform mobile app.

Android Studio: An integrated development environment (IDE) designed for building Android applications. It provides tools for coding, testing, and debugging, and supports development with Flutter for cross-platform applications.

GitHub: A web-based platform for version control and collaboration, using Git. It allows developers to share code, track changes, and work on projects together, making it ideal for team-based projects.

System Architecture: The structural design of an application, including the components (Model, View, Controller) and their interactions. It defines how the app's different parts work together to deliver the desired functionality.

MVC (Model-View-Controller): A software architectural pattern that separates the application into three interconnected components: Model (data), View (UI), and Controller (business logic). This separation improves modularity and facilitates easier maintenance.

Model: In the Model-View-Controller (MVC) architecture, the Model represents the data and business logic of the application. It handles the storage, retrieval, and manipulation of data such as alarms and user preferences.

View: In the MVC architecture, the View is responsible for displaying the user interface and presenting the data to the user. It receives input from the user and communicates with the Controller to update the Model.

Controller: In the Model-View-Controller (MVC) architecture, the Controller is responsible for processing user inputs and managing the communication between the Model and View components. It implements business logic and ensures that user interactions are properly handled.

11. References

Android Developers. *Android Developer Documentation*. Google, https://developer.android.com. Accessed 10 Dec. 2024.

Flutter Team. *Flutter Documentation*. Google, https://docs.flutter.dev. Accessed 10 Dec. 2024.

Macia, Daniel. *Software Requirements Document (SRD)*. GitHub, https://github.com/dmacia49/CS_5781_Project/blob/main/Software_Requirements_Document_(SRD).pdf. Accessed 10 Dec. 2024.

SQLite Consortium. *SQLite Documentation*. SQLite, https://sqlite.org. Accessed 10 Dec. 2024.