Software Design Document: TheReminder

### **1. System Overview**

This Software Design Document describes the implementation of the 'Create/Save Task Data' feature of TheReminder app. This feature is essential for ensuring task persistence, enabling the application to reliably save newly created or edited tasks in a local SQLite database. By doing so, users' tasks remain available across application restarts and device usage sessions. The system uses the Singleton design pattern to manage data consistency, prevent redundant database connections, and provide a centralized mechanism for data operations.

### **2. System Context**

TheReminder is a mobile task management app developed using Flutter, primarily targeting Android platforms. All task data is stored locally on the device using SQLite, which is accessed through the sqflite plugin.

The system architecture is functionally divided into:

* **UI Layer**, where users interact with tasks through screens and forms
* **Data Layer**, managed by the DatabaseHelper singleton, which handles all SQLite operations

The DatabaseHelper class is responsible for database initialization, schema creation, and CRUD operations, ensuring consistent and exclusive access throughout the app. This follows the Singleton design pattern.

There is no backend or cloud integration; the app is fully operational offline, and all data persistence is handled locally on the device.

### **3. Key Features and Functionality**

- Create new tasks with title, description, due date, and priority.

- Save tasks persistently into a local SQLite database.

- Ensure database schema is initialized at app launch.

- Handle updates to existing task data through the same manager class.

- Allow only one instance of the database access layer.

### **4. Assumptions and Dependencies**

* The user owns an Android device with support for local storage and necessary accessibility hardware (e.g., vibration motor, screen reader support).
* The application is developed using the Flutter framework and is compatible with Android SDK version 21 or higher.
* The local database layer is implemented using the sqflite plugin and relies on additional packages such as path.
* The SQLite integration is assumed to be functional and initialized at application startup before any data-related operations.
* The application is designed to work fully offline; it does not rely on any external backend server or network connection.
* To ensure data consistency, a Singleton pattern is used to manage all database interactions through a single point of access.

### **5. Architectural Design**

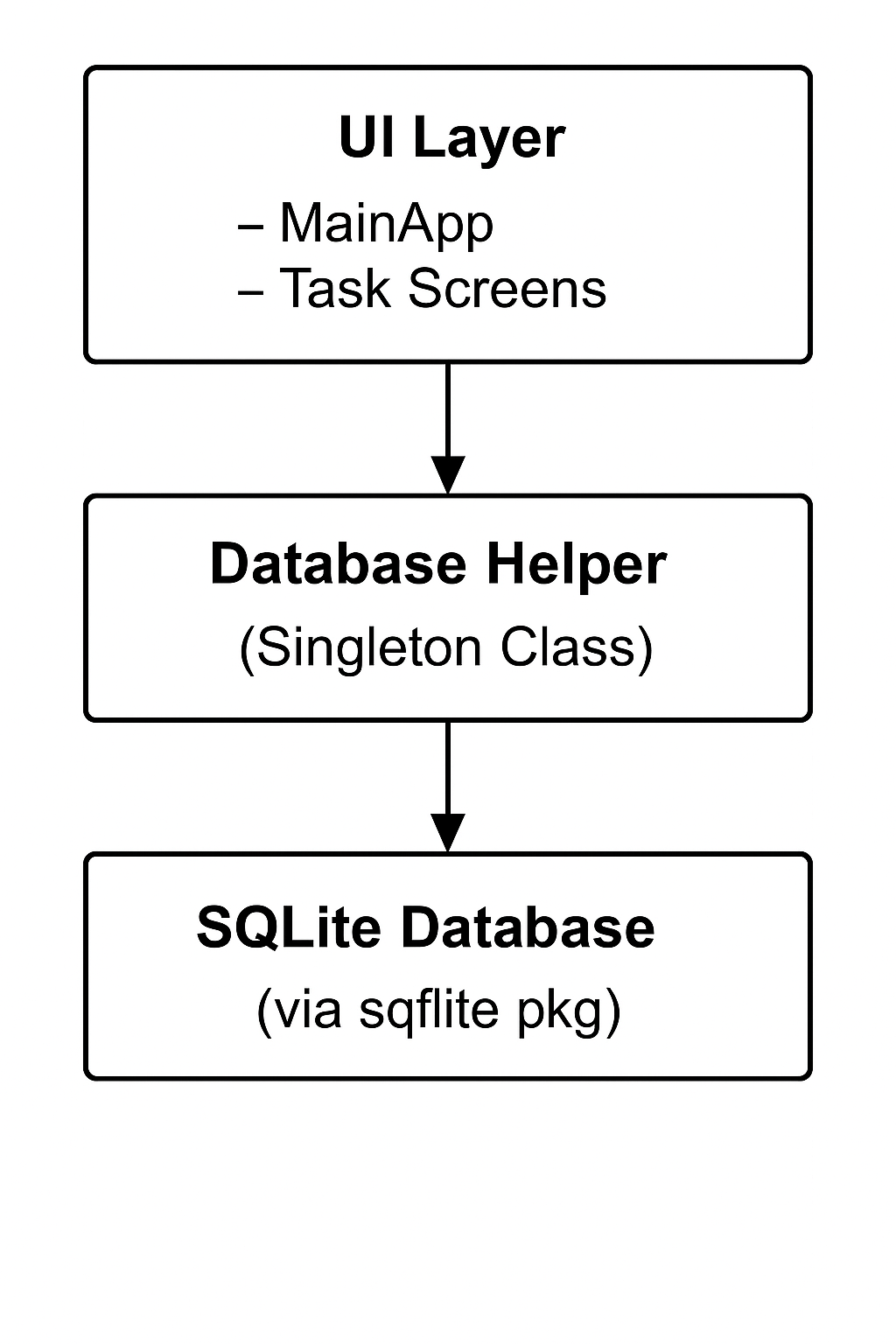
### **a. System Architecture Diagram (High-Level)**

The architecture of the Reminder application is organized into three main layers:

**UI Layer** contains the user-facing components, including MainApp, task creation, and list screens.

**DatabaseHelper** acts as a service layer, managing data access through a Singleton pattern.

**SQLite Database** stores all task and reminder data locally, accessed via the sqflite Flutter plugin.



### **b. Architectural Patterns and Styles**

* **Layered Architecture**:  
   The application follows a layered structure with:  
  + **Presentation Layer** (Flutter widgets)
  + **Model Layer** (Task, Reminder, and Enums)
  + **Data Access Layer** (DatabaseHelper)
* **Singleton Pattern**:  
   DatabaseHelper is implemented as a Singleton to ensure a single instance of the database is used app-wide. It encapsulates all interactions with the database.

### **c. Rationale for Architectural Decisions**

* **Offline-first** functionality is prioritized, so SQLite is used as the local data store with no backend dependency.
* The **sqflite** package is lightweight, performant, and integrates seamlessly with Flutter.
* A **Singleton** architecture is used to ensure:  
  + One consistent database connection
  + Lazy initialization
  + Simplified and centralized database logic
* **Layer separation** improves maintainability, readability, and future extensibility of the application.

### **6. Component Design**

#### **a. Subsystems and Modules**

There are 2 subsystems currently that work together to provide basic CRUD operations and a way to view the data saved using said CRUD operations:

* **User Interface (UI) Subsystem:**

Handles rendering data and user interactions.

**UI subsystem modules:**

* + HomeScreen
  + CreateTaskScreen
  + SettingsScreen
* **Data Persistence Subsystem:**

Handles all data related tasks including saving and retrieving data from SQLite database.

**Data Persistence Subsystem Modules:**

* Reminder Model
* Task Model
* Database Helper

#### **b. Responsibilities of the Component**

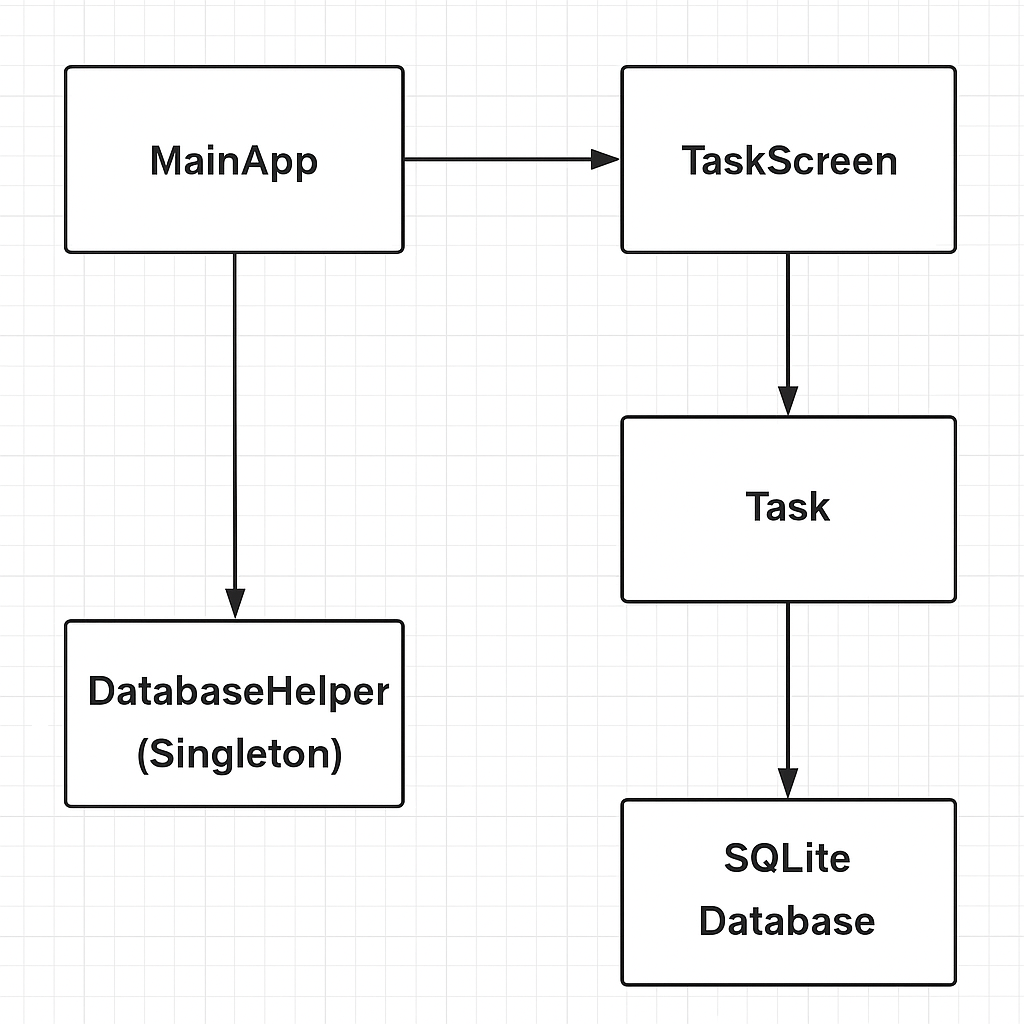
The DatabaseHelper class:

* Initializes and opens the SQLite database
* Creates required tables (Task, Reminder)
* Inserts new tasks and their reminders
* Deletes tasks and associated reminders
* Fetches all task data for rendering

#### **c. Interfaces Between Components**

* main.dart (UI) calls DatabaseHelper.instance.tasks to load data
* Task creation screen calls addTask(Task) to save user input
* Task and Reminder classes are passed into the database logic via toMap() and retrieved with fromMap()
* Returns Future-based results (Future<List<Task>>) so the UI can await and reactively update

#### **d. Component Diagram**



### **7. Data Design**

#### **Data Model / ER Diagram**

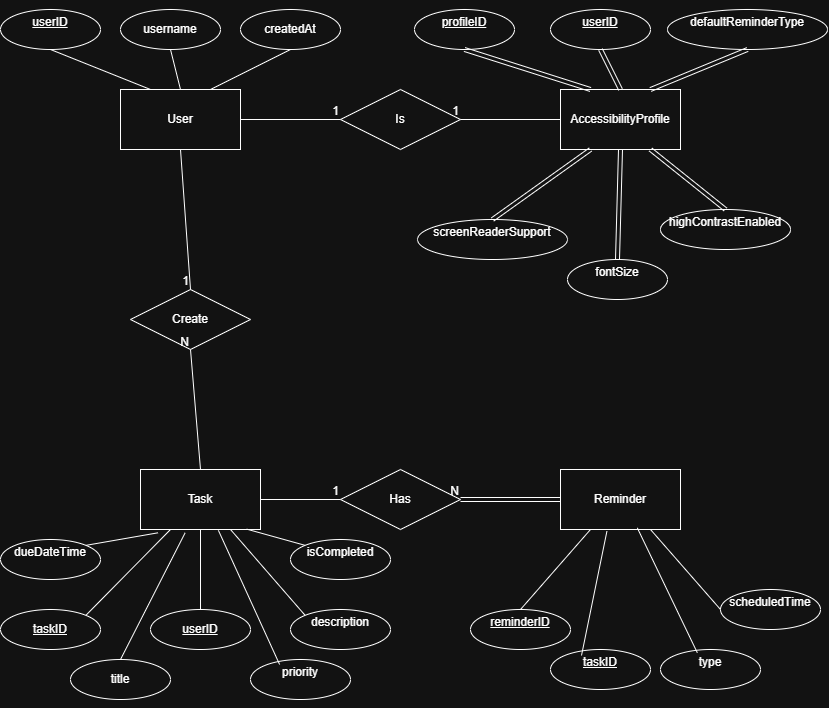
The application's data model includes four primary entities:

#### **Entities & Relationships:**

1. User  
   * Attributes: userID, username, createdAt
   * Relationships:  
     + 1:N User → Creates → Task
     + 1:1 User → Is → AccessibilityProfile
2. AccessibilityProfile  
   * Attributes: profileID, userID, defaultReminderType, screenReaderSupport, fontSize, highContrastEnabled
   * Purpose: Defines accessibility settings for each user (font size, reminder defaults, etc.)
3. Task  
   * Attributes: taskID, title, description, dueDateTime, isCompleted, priority, userID
   * Relationships:  
     + 1:N Task → Has → Reminder
4. Reminder  
   * Attributes: reminderID, taskID, type, scheduledTime
   * Purpose: Stores the type and time for notifying a user of an upcoming task.

Each task belongs to a user and may have one or more reminders. Each user has one accessibility profile.

The diagram above visually represents these entities and relationships.



### **b. Data Storage (Database or File Structure)**

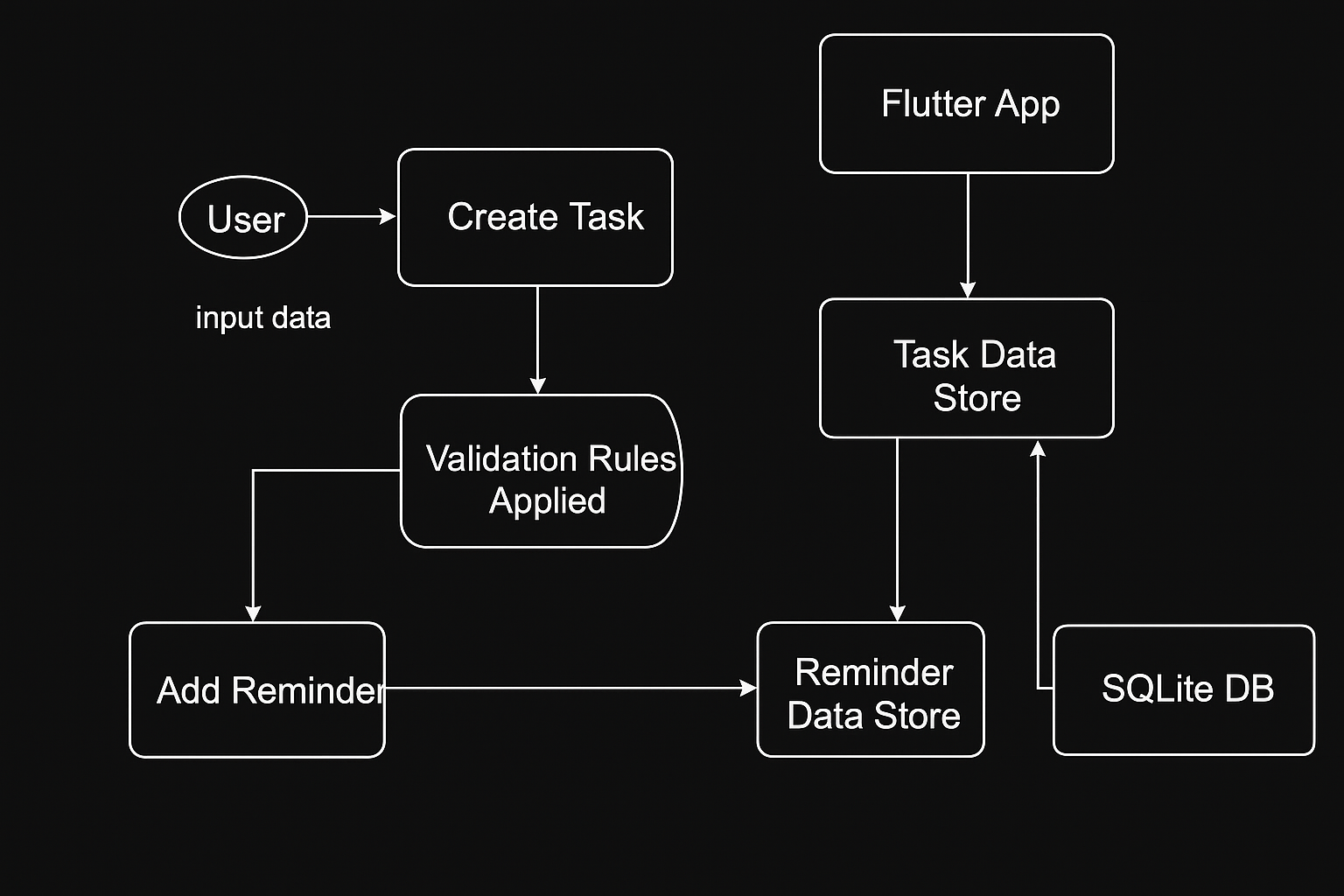
The application uses SQLite as the primary data storage engine, integrated via the sqflite Flutter plugin. The structure is defined through two main tables:

#### **Tables:**

1. Task  
   * Columns: taskID, title, description, dueDateTime, isCompleted, priority
   * Indexed: taskID (PK), userID (FK)
2. Reminder  
   * Columns: reminderID, taskID, type, scheduledTime
   * Foreign key: taskID → Task(taskID)

Note: The User and AccessibilityProfile entities are currently only represented in the ER model for extensibility and accessibility mockup; they are not yet implemented in the SQLite schema.

### **c. Data Flow Diagrams**



* When a user creates a task, it is passed from the UI to the DatabaseHelper.addTask() method and inserted into SQLite.
* When the app loads, DatabaseHelper.tasks retrieves all tasks and their reminders for display.
* Data flows one-directionally from user input to storage, and then from storage to UI rendering.

### **d. Data Validation Rules**

Validation occurs primarily in the UI before data reaches the database. Key rules include:

| **Field** | **Rule** |
| --- | --- |
| title | Required, must be non-empty |
| dueDateTime | Must be a valid future date/time string  (is currently string but will be changed) |
| reminderType | Must match one of the allowed enum values |
| priority | Must be one of: High, Medium, Low |
| isCompleted | Boolean, default false |
| fontSize | Integer or enum; must match available UI sizes |
| screenReaderSupport | Boolean, UI-toggled |

Additional rules may be enforced later in the model or service layer (e.g., checking if a reminder’s time precedes the task’s due date).

### **8. Design Patterns**

In the current implementation of *TheReminder*, **Singleton Design Pattern** has been applied to manage the local data storage system. While other design patterns such as Strategy, Observer, and Decorator were planned and mapped to other use cases, only Singleton has been fully implemented as part of the **"Create & Save Task Data"** use case.

### **Singleton Pattern**

* **Purpose**: To ensure that only one instance of the database access class exists and is shared throughout the application.
* **Implementation Context**: Applied in the DatabaseHelper class, which manages all database-related operations including task creation, retrieval, update, and deletion.
* **Problem Solved**: In mobile apps, opening multiple SQLite connections can lead to inconsistent data, crashes, or resource leaks. Singleton provides a single point of control for all task-related data operations.

### **How It Works in TheReminder**

* A private static instance of DatabaseHelper is declared inside the class.
* The constructor is made private using a named constructor (DatabaseHelper.\_internal()).
* A factory constructor (DatabaseHelper()) always returns the same instance.
* The database is lazily initialized the first time it's accessed through the database getter.

This ensures that all screens—such as the task creation form or task list—interact with a shared and consistent database state, avoiding duplicate connections and maintaining data integrity throughout the app lifecycle.

### **9. Implementation Notes**

TheReminder application integrates with several external interfaces to fulfill its core features, particularly around local data storage, notifications, and accessibility enhancements. But most of the accessibility and notification features will be implemented in the future. Currently TheReminder integrates with technology given below.

#### **9.1 Flutter SDK**

* **Purpose**: Core framework used for UI design, logic implementation, and cross-platform capabilities.
* **Technology**: Dart programming language within Flutter SDK (version 3.8.1).
* **Platform**: Targeted only for Android, although Flutter supports cross-platform development.

#### **9.2 SQLite (via sqflite plugin)**

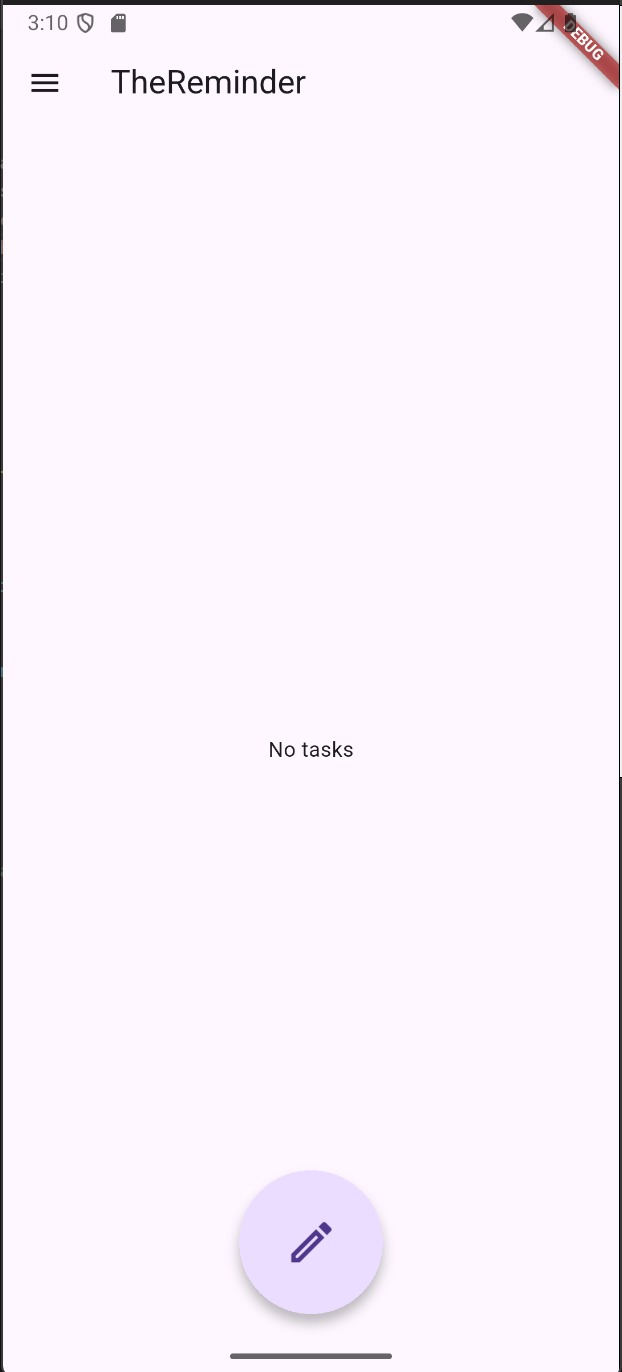
* **Purpose**: Manages persistent local storage for tasks, reminders, and user settings.
* **Plugin**: sqflite – provides a Flutter-compatible interface to Android’s native SQLite API.
* **Interaction**: Task data is serialized via Dart model classes and stored in SQLite using structured tables.

#### **9.3 Path Provider (via path plugin)**

* **Purpose**: Used to determine device-specific paths for SQLite database file storage.
* **Plugin**: path – assists in constructing valid file paths across Android devices.

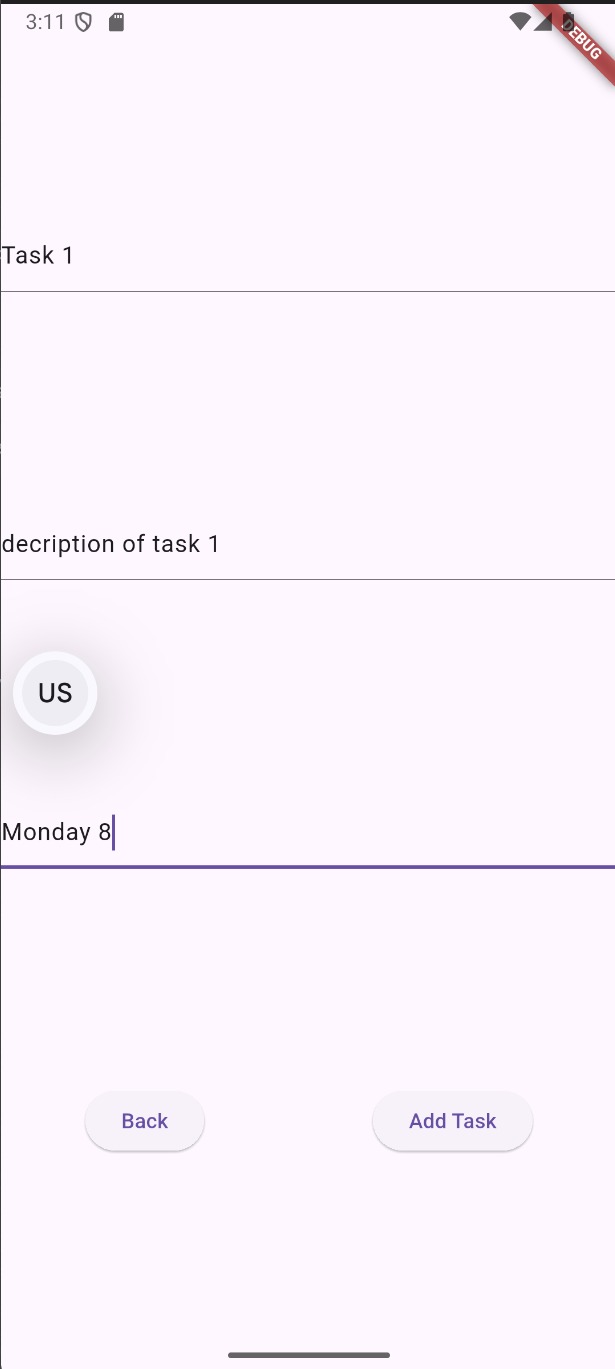
### **10. User Interface Design**

TheReminder features a simple and clean user interface that supports essential task management operations like creating, viewing, and deleting tasks. The UI is built using Flutter and follows Material Design principles to ensure a smooth and accessible user

**Initial Home Screen (No Tasks)**

* When no tasks exist, the app displays a centered message: **"No tasks"**.
* A **Floating Action Button (FAB)** is located at the bottom center for adding a new task.
* The top app bar contains a **menu (drawer)** icon and the title **“TheReminder”**.

#### **Create Task Screen**

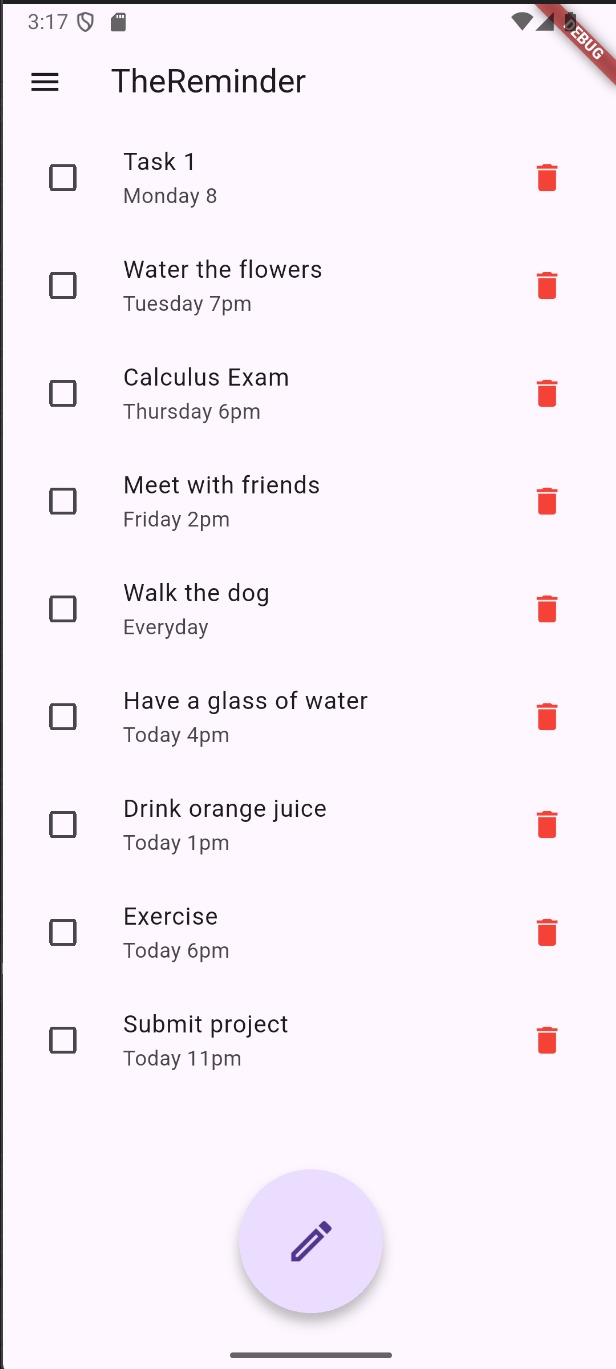


* Users can input:  
  + **Task Title**
  + **Description**
  + **Due Date/Time**
* Two buttons at the bottom:  
  + **"Back"** – returns to the home screen
  + **"Add Task"** – saves the task to the database
* Input fields are clearly separated and easy to interact with.

**Home Screen with One Task**

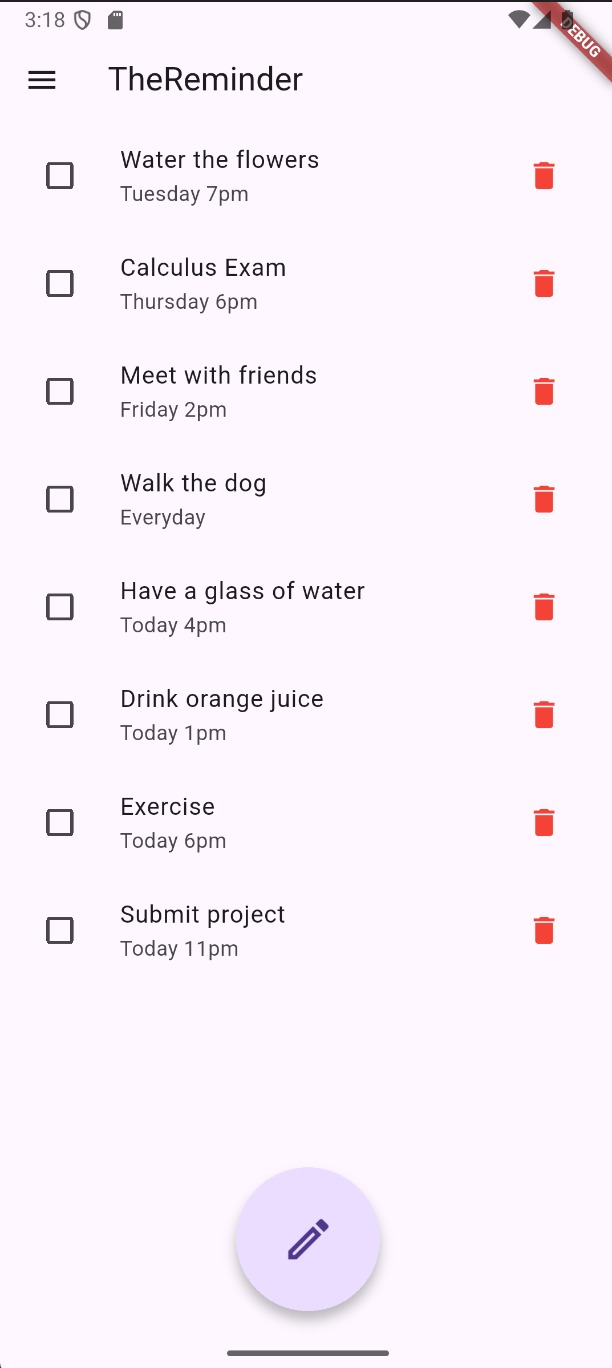
* The screen shows the added task with:  
  + A **checkbox** to mark it as completed
  + The **due date** displayed beneath the task title
  + A **red trash icon** to delete the task

#### **Home Screen with Multiple Tasks Including "Task 1"**



* "Task 1" appears at the top, followed by several other tasks (e.g., “Water the flowers”, “Calculus Exam”).
* Layout ensures:  
  + Clear task visibility
  + Each task has its own checkbox and delete icon
* The list is vertically scrollable, and the FAB remains accessible.

#### **Updated Task List After Deletion**



* Confirms that the delete function works as expected.
* The remaining tasks shift upward in the list.
* The interface updates dynamically without needing a page reload or restart.
* Reinforces the app’s support for real-time task management.

### **11. External Interfaces**

TheReminder interacts with several third-party libraries and platform APIs to deliver its full functionality.

| **External Library / Interface** | **Purpose** |
| --- | --- |
| **sqflite** | Local data storage using SQLite |
| **path** | Manages file paths across platforms |
| **Flutter SDK** | Core framework for UI and logic implementation |
| **dart:developer** | Used for lightweight logging and debugging during development |

These external libraries are managed via pubspec.yaml and automatically resolved during development

### **12. Performance Considerations**

The application is optimized for mobile use and built with performance best practices in mind:

* **Efficient Local Storage**: All task-related data is stored in SQLite, ensuring fast read/write operations without needing a network connection.
* **Startup Speed**: The database is initialized during the initState() lifecycle of the app. This avoids delays in user interaction.
* **Reminder Handling**: Reminder logic is lightweight, using background timers or scheduling through flutter\_local\_notifications.
* **UI Responsiveness**: Flutter’s reactive rendering ensures smooth transitions and low input latency across different Android devices.
* **Scalability**: While designed for single-user offline use, the database structure and component separation allow future extension to multi-user or cloud-based systems.
* **Optimization Strategy**:  
  + Lazy loading of task data (only when needed)
  + Reuse of widget states to reduce rebuild costs
  + Enum and constant-based logic to avoid runtime overhead

### **13. Error Handling and Logging**

TheReminder includes basic yet essential mechanisms for error handling and logging to ensure application robustness:

* **Database Initialization Errors**: If the database fails to load during startup (e.g., due to missing permissions or file corruption), the system ensures the UI does not crash. Logging mechanisms can help trace such errors during development.
* **Silent Failures in Task Fetching**: Task retrieval during app launch or refresh is handled asynchronously. While there is no explicit error message to the user, logging should be used to capture any failures in loading data.
* **User Interaction Logs**: Certain interactions, such as opening the settings drawer, are logged using Dart's built-in logging tools (log() function). This provides traceability of user actions during testing.
* **Future Improvement**: For production readiness, implementing a global error handler and integrating with a structured logging system (e.g., Firebase Crashlytics) would enhance visibility and maintainability.

### **14. Design for Testability**

The application is designed with modularity in mind, allowing individual components to be tested effectively:

* **Layered Design**: By separating data operations (DatabaseHelper), UI (Flutter widgets), and models (Task), it becomes easier to write unit tests or widget tests for each part.
* **Model Serialization**: The Task class handles its own mapping to and from database records, making it predictable and easy to mock or test.
* **Database Abstraction**: All database operations are abstracted under one class (DatabaseHelper), which simplifies mocking during test runs.
* **Expandable Test Coverage**: The structure enables test automation for key flows like adding tasks, updating reminders, and marking completion, without tightly coupling with UI events.

### **15. Deployment and Installation Design**

Deployment is tailored for Android devices and intended to be smooth and developer-friendly:

* **Build Environment**: The project is built using Flutter and compiled into APKs via Android Studio. Developers can use either real devices or emulators to test installations.
* **Installation Process**: The APK can be distributed manually or uploaded to testing services like Firebase App Distribution.
* **Dependencies and Setup**: All project dependencies (e.g., sqflite, path etc.) are listed in pubspec.yaml, and are automatically resolved using flutter pub get.
* **File Structure and Storage**: The SQLite database is stored locally on the device, and created automatically on the first launch.

### **16. Change Log**

| **Date** | **Description** | **Contributor** |
| --- | --- | --- |
| 2025-06-15 | Initial setup of database schema and tables | Feyza Coşkun |
| 2025-06-20 | Integrated reminder logic into the task model | Taha Mert Ağım |
| 2025-06-25 | Finalized enum and task serialization logic | Talha Akbulut |
| 2025-06-28 | Connected database operations to home screen | Beyzanur Zeybek |
| 2025-07-01 | Polished UI, added drawer, logging, cleanup | Özge Doğan |

This log ensures transparency of changes and helps track progress over the final phases of the assignment.

### **17. Future Work / Open Issues**

Although the current implementation meets the basic requirements, several enhancements and open issues are identified for future development:

* **Reminder Storage Refactor**: Currently, reminders are stored as a string. A dedicated table or structure for reminders would improve scalability and flexibility.
* **Settings Functionality**: The settings screen is present but not yet functional. Integrating real settings (e.g., font size, default reminder type) is planned.
* **Accessibility Feedback**: While visual and audio cues are implemented, further feedback mechanisms for accessibility (e.g., vibration patterns, voice prompts) could be added.
* **User Management**: Multi-profile support is outlined in the database schema but not yet active in the UI.
* **Reminder Engine Robustness**: Error recovery mechanisms (e.g., retries if a reminder fails to fire) are not yet implemented.
* **Cloud Backup or Sync**: For long-term use, cloud synchronization of tasks and preferences would be valuable.

These improvements would enhance the reliability, usability, and inclusiveness of the application in real-world use.