**Quizer App Documentation**

**Overview**

**The Quizer App is a quiz-based application that allows instructors to create questions on various topics. Students can attempt the quizzes, earning points based on correct answers (1 point per correct answer, 0 for incorrect). The app uses Clean Code and Clean Architecture principles for maintainability and scalability. The application includes features such as:**

* **Authentication (excluding Google and Facebook login)**
* **OTP-based password reset**
* **Local push notifications (scheduled)**
* **Scoring system for quizzes**

**Principles**

**The app adheres to the following key principles:**

**Clean Code:**

* **Separation of concerns: Features are divided into distinct modules and layers (as seen in the folder structure).**
* **Readability: Each component follows best practices to make the code easy to read and maintain.**

**Clean Architecture:**

* **Presentation Layer: Responsible for displaying UI elements and responding to user inputs.**
* **Domain Layer: Contains business logic, entities, and use cases.**
* **Data Layer: Handles data access, whether from APIs, databases, or other data sources.**

**Dependency Injection (DI):**

**The app uses dependency injection (see dependency\_injection.dart) to manage class dependencies and facilitate testing.**

**Folder Structure**

**The application is organized into well-defined folders as seen in the diagram:**

* **lib/config: Contains configuration elements such as:**
  + **notifications: Handles local push notifications.**
  + **routes: Defines the navigation routes of the app.**
  + **themes: Provides theme configurations.**
* **lib/core: Contains core functionalities such as:**
  + **constants: Defines app-wide constant values.**
  + **helper: Provides utility/helper classes and methods.**
  + **resources: Manages assets like images and localization files.**
* **lib/features:**
  + **data\_sources:**
    - **API: Handles communication with backend services.**
    - **local: Manages local data storage (e.g., SQLite, shared preferences).**
    - **models: Contains data models (e.g., question and user models).**
    - **repository\_impl: Implements data repositories.**
  + **domain:**
    - **entities: Defines core entities (e.g., Question, User).**
    - **repository: Interfaces for repositories.**
    - **usecases: Business logic use cases (e.g., fetching questions, validating OTPs).**
  + **presentation:**
    - **common: Shared UI elements and widgets.**
    - **cubit: Handles state management using the Cubit package.**
    - **pages: Contains all UI pages (e.g., quiz pages, login page).**
    - **state: Defines the different states of the UI (e.g., loading, error, success).**

**Authentication**

**The app supports authentication via email and password. It includes a feature for OTP-based password reset. The OTP is sent via email, and the user can reset their password by entering the correct OTP ,also user can signup with.**

**Quizzes**

**Instructors can create quizzes by defining questions. Each question carries one point for correct answers. The scoring system is simple, making it easy to track student progress.**

**Local Push Notifications**

**The app uses local push notifications to remind users about upcoming quizzes. Notifications are scheduled using a notification manager in lib/config/notifications.**

**API Integration**

**The app integrates with APIs for the following processes:**

* **User Authentication: Login and signup.**
* **Quiz Management: Fetching questions, submitting answers, and calculating scores.**
* **Password Reset: OTP generation for resetting passwords.**

**Key Features**

1. **Authentication:**
   * **Register and login functionality.**
   * **Password reset with OTP verification.**
2. **Quiz Creation and Management:**
   * **Instructors create quizzes.**
   * **Students attempt quizzes and earn points for correct answers.**
3. **Local Notifications:**
   * **Notifications for upcoming quizzes, reminders, etc.**
4. **Scoring System:**
   * **Each correct answer gives one point; incorrect answers give zero points.**
5. **Clean Architecture:**
   * **Each component is separated into layers (presentation, domain, data).**

**Example Use Case: Submitting a Quiz**

1. **User Input: The student submits their answers on the quiz page.**
2. **UI Interaction: The QuizCubit updates the state to show a loading indicator.**
3. **Business Logic: The SubmitQuizUseCase processes the answers and calculates the score.**
4. **Data Storage: The QuizRepositoryImpl stores the result in local storage or updates it on the server via the API.**
5. **Notification: A local notification is scheduled to notify the user of the next quiz.**

**Technologies Used**

* **Flutter: The UI framework for building the app.**
* **Cubit/Bloc: For state management.**
* **HTTP/REST API: For communication with backend services.**
* **Local Notifications: For sending reminders and updates to users.**
* **OTP Authentication: For secure password resets.**

**Future Enhancements**

* **Google and Facebook Authentication: To provide more login options.**
* **Advanced Quiz Statistics: Detailed statistics to help students track their performance.**
* **Leaderboard: A ranking system to encourage competition among students.**