**Proposal: Backend Synchronization for Alarm Application**

**1. Data Architecture:-**

To extend the existing alarm application to support cloud synchronization, we need to design a scalable and robust data architecture that will work in sync with the current SQLite implementation. The cloud database should reflect the user’s alarm data in real-time while accounting for offline capabilities.

**Database Schema Design**

The existing SQLite database stores alarms with fields like time, repeat, and enabled status. To support cloud synchronization, we will maintain a similar schema in the cloud, but with additional fields that allow us to manage user-specific data and synchronization statuses. The database schema would look like:

1. **Alarms Table (Cloud)**
   * id (INTEGER): Primary key, auto-increment.
   * user\_id (INTEGER): Foreign key referencing the user. This allows linking alarms to specific users.
   * time (TEXT): Time of the alarm, stored in UTC format for global consistency.
   * repeat (TEXT): Repeat frequency of the alarm.
   * isEnabled (INTEGER): Boolean value to track if the alarm is enabled.
   * last\_synced (INTEGER): Timestamp when the alarm was last synced with the server.
   * sync\_status (TEXT): Stores whether the alarm is synced, pending, or conflict for tracking synchronization states.

**Handling User Data and Alarm Data**

* **User Data**: Each user will have a unique user ID assigned during authentication (using Google Sign-In). This ID will be associated with all alarm data, allowing us to identify and synchronize alarms per user.
* **Alarm Data**: Alarm details such as time, repeat frequency, and status will be stored both locally and on the server. The cloud database will store user-specific alarm data, while the local SQLite database will store alarms with synchronization flags to determine which alarms need to be synced.

**Handling Timezone Differences**

* To handle timezone differences effectively, all alarm times will be stored in **UTC** (Coordinated Universal Time) in both the local SQLite database and the cloud. The local time conversion will happen when displaying the alarm to the user. This will prevent issues when users travel to different timezones.
* The app will convert UTC time to the local timezone on display and handle conversions using Flutter’s timezone package or equivalent.

**2.Synchronization Strategy:-**

The goal is to keep alarm data synchronized between the client (device) and the server, while ensuring smooth offline functionality.

**Conflict Handling**

When syncing data between the server and the local SQLite database, conflicts may arise (e.g., a user modifies an alarm locally while offline, and the same alarm is modified on the server). To handle conflicts, we will use a **last-write-wins** strategy, but with a conflict resolution flow:

* **Conflict Detection**: If an alarm’s last\_synced timestamp is different on the server compared to the local version, it indicates a conflict.
* **Resolution Flow**: The app will prompt the user with a conflict resolution interface where they can choose to keep the local or server version, or merge both (for example, updating the repeat frequency from both versions).

**Offline-First Strategy**

To ensure the app works well offline, all alarm data will be stored locally in SQLite and will attempt to sync with the server only when a network connection is available. This approach ensures the application is fully functional offline while syncing the data once the user is online.

1. **Local Storage**: Alarm data will be stored locally when created or modified, with a sync\_status flag set to "pending."
2. **Synchronization Trigger**: The app will attempt to sync all "pending" alarms with the server when the device is online. If the sync fails, the data remains in the local database until it succeeds.

**Data Versioning Strategy**

To track and manage the synchronization of alarm data, each record will include a version number or a last\_synced timestamp. When a record is modified, the version or timestamp will be updated:

* The version number or timestamp helps identify which version of the data is the most recent.
* For each alarm, the app will check if the version or timestamp on the server is newer than the local copy and perform the necessary actions accordingly

**3.API Design.:-**

The cloud synchronization will require a set of RESTful API endpoints for alarm management, user authentication, and synchronization.

**Proposed Endpoints**

1. **POST /alarms**: Creates a new alarm on the server.
   * Request Body: { "user\_id": <user\_id>, "time": <time\_in\_utc>, "repeat": <repeat>, "isEnabled": <boolean> }
   * Response: { "status": "success", "alarm\_id": <alarm\_id> }
2. **GET /alarms**: Retrieves all alarms for a given user.
   * Query Parameters: user\_id=<user\_id>
   * Response: { "alarms": [ { "id": <id>, "time": <time\_in\_utc>, "repeat": <repeat>, "isEnabled": <boolean>, "last\_synced": <timestamp> } ] }
3. **PUT /alarms/{id}**: Updates an alarm by ID.
   * Request Body: { "time": <time\_in\_utc>, "repeat": <repeat>, "isEnabled": <boolean> }
   * Response: { "status": "success" }
4. **DELETE /alarms/{id}**: Deletes an alarm by ID.
   * Response: { "status": "success" }

**Data Format**

* **Requests and Responses**: All requests and responses will be in JSON format.
* **Time Format**: Times will be sent and received in ISO 8601 UTC format (e.g., "2024-12-13T15:30:00Z").
* **Authentication**: Requests will require an **OAuth 2.0 token** for authentication, which will be tied to the user’s Google account (via Google Sign-In).

**Authentication and Security Considerations**

* **OAuth 2.0**: The app will continue using **Google Sign-In** for user authentication. The OAuth token will be used to authenticate API requests.
* **Secure Connections**: All API endpoints will be secured using HTTPS to protect data in transit. Tokens will be sent in the Authorization header, and user passwords (if applicable) will be hashed before being stored.

**4.Error Handling:-**

Robust error handling is essential to ensure smooth synchronization and prevent data loss.

**Network Failures**

* **Network Detection**: The app will detect network connectivity issues. If no network is available, the app will queue synchronization operations and retry once the connection is restored.
* **Error Reporting**: Users will be notified of errors via in-app notifications or SnackBars, providing information on the type of error (e.g., "Unable to sync alarms due to network failure").

**Retry Mechanisms**

* **Exponential Backoff**: For failed sync attempts, an exponential backoff strategy will be used to retry the operation. This reduces the chances of overwhelming the server with requests and ensures more efficient synchronization when the connection is unstable.

**Data Consistency**

* **Conflict Resolution**: If conflicts occur during synchronization, a user interface will allow the user to resolve them manually, ensuring data consistency across devices and the server.
* **Sync Flags**: Every alarm will include a sync\_status flag to track whether it needs to be synced, preventing overwriting of alarms that haven’t been synchronized yet.