ChatGPT

Plan for Sri Lanka Travel Mobile App Development

Project Overview and Requirements

This project involves developing a **cross-platform mobile app** (iOS and Android) using Flutter for travelers in Sri Lanka. The app will serve as a travel guide with features for finding nearby attractions, planning tours/trips, exploring places on a map, viewing travel tips videos, saving favorite items, and accessing emergency information. We will use a **Flutter** front-end (to build one app for both iOS and Android from a single codebase [1]) and a **Supabase** back-end (which provides a hosted PostgreSQL database and API) for data storage. **Clerk** will handle user authentication (Google sign-up/sign-in) and manage user accounts and subscription billing as needed. Below is a breakdown of the app's key features and the step-by-step plan to design and implement it from system architecture to database design and development.

Key Features and Functionality

- **User Authentication:** Allow users to sign up/sign in with Google via Clerk integration. This provides a quick OAuth login and secure account management.
- **Side Menu (Hamburger Navigation):** A slide-out menu to navigate between major sections of the app (Home, Tours, Explore, Videos, Saved, Emergency, etc.).
- **Home Page (Nearby Places):** Displays nearby attractions, hotels, and restaurants relative to the user's current location. (Requires obtaining the device's GPS location).
- **Tours Page:** Provides curated **Recommended Tours**, user-created itineraries (**My Trips**), and bookmarked tours (**Saved Tours**).
- **Explore Page (Map):** Interactive Google Map showing points of interest (hotels, attractions, hidden gems, beaches, current events). Also includes a section (5.1) for **Recommended Travel Apps** for Sri Lanka.
- **Travel Tips & Videos:** A section to watch tips, tricks, advice videos for traveling in Sri Lanka (likely embedded YouTube videos or stored video links).
- **Saved Items:** A personalized list where users can save/bookmark attractions, places, or content for later reference.
- **Emergency Info:** Quick access to emergency contacts (tourist police, ambulance, etc.) and safety information for travelers.

These features address the user's core needs: finding interesting places, planning and saving trips, getting helpful advice, and ensuring safety information is at hand. The app will initially be in English (with potential to add other languages later, as the user noted "for now English is fine"). Both iOS and Android platforms will be targeted (the user specified **"1. both"**, meaning both platforms).

Technology Stack Selection

- **Flutter:** Chosen for front-end development to build a high-quality native interface for both Android and iOS using a single codebase [2] . Flutter's cross-platform efficiency will reduce development time and ensure a uniform user experience.

- **Supabase:** Chosen for the back-end. Supabase provides an open-source Firebase alternative with a robust PostgreSQL database at its core [3]. It offers a ready-made RESTful API for the database via PostgREST [4] and real-time capabilities, which will simplify building the app's data services. Using Supabase means we get a hosted Postgres database, authentication (though we will use Clerk instead), file storage, and potentially cloud functions if needed, all integrated seamlessly.
- **Clerk (Auth & Billing):** Chosen for user authentication and account management. Clerk provides streamlined user management – with easy sign-up/sign-in flows, social login options (Google OAuth in our case), and user profile management [5]. We will integrate Clerk's Flutter SDK for handling authentication in-app. Clerk also has a billing feature (integrating with Stripe) that can manage subscriptions/payments if we decide to offer premium content or paid features in the future.
- **Google Maps API:** We will integrate the Google Maps Flutter plugin to display interactive maps on the Explore page [6]. This requires obtaining API keys and adding the `google_maps_flutter` package to the Flutter project. Google Maps will allow us to show map tiles, markers for locations, and user location on the map.
- **Device Geolocation:** To get the user's current location (for nearby attractions on the Home page), we will use Flutter's geolocation capabilities. For example, the `geolocator` plugin provides a cross-platform API to get GPS location in Flutter [7]. This will let us determine what attractions/hotels are "nearby" the user.
- **External APIs (Optional):** We might use Google Places API or a custom dataset for detailed place information (ratings, hours, etc.) if needed. However, we can also store the necessary place details in our own database if we have a predefined list of attractions and hotels for Sri Lanka. For events (happenings), an external source or a manually updated list might be needed unless we maintain that data ourselves.
- **Supabase Storage:** If we need to store images or videos (e.g., photos of attractions or the travel tip videos) we can use Supabase's built-in storage service which is an S3-compatible object storage [8]. This allows storing media files and serving them to the app. In our case, attraction images or thumbnails for videos could be stored here, or we might embed YouTube for video content to avoid heavy storage.

With this tech stack, we leverage Flutter for a smooth UI and Supabase for a scalable backend. The use of Clerk for auth ensures secure and easy login (especially via Google) and offloads user management complexities. Next, we design the system architecture and data model around these technologies.

System Architecture Design

The system will follow a client-server architecture with the following components:

- **Flutter Mobile App (Client):** The Flutter app will handle all user interactions, UI rendering, and make network requests. It will include the UI screens (pages) for each feature: authentication screens, home, tours, explore (map), video list, saved items, and emergency info. The app will use Flutter's widgets for layout and controllers for state management (we might choose a state management approach like Provider, Riverpod, or BLoC for maintainability).
- **Clerk Authentication Service:** The app will integrate with Clerk's authentication. We will use the Clerk Flutter SDK or API to initiate Google OAuth flows. On sign-in, the user will authenticate through Google and Clerk will return an auth token/session. Clerk manages users and can issue JWTs or session tokens we can use. The Flutter app will store the user session securely (possibly using Clerk's SDK which handles session persistence).

- **Supabase Backend (BaaS):** Supabase will host the PostgreSQL database that stores our application data. Supabase provides an API (REST and optional GraphQL) for the Flutter app to read/write data. We'll configure Row-Level Security (RLS) policies in Supabase for any tables containing user-specific data (like saved items or user-created trips) to ensure only the owning user can read/write their data. The app will communicate with Supabase either using the Supabase Dart client library or via HTTP requests. Since we are using Clerk for auth, we might need to integrate Clerk with Supabase's auth if we want to enforce RLS; one approach is to use a service role key or create a JWT from Clerk that Supabase can validate. Alternatively, since public data (like attractions) can be read without auth, we can make those tables publicly readable (with an RLS policy that allows anon read [9]). For private data, the app could call Supabase with an authenticated context (e.g., we might use Supabase's JWT if the user also exists in Supabase, or simply use the user's Clerk ID in our tables and filter in queries).
- **External APIs and Services:**
- **Google Maps SDK:** Runs within the Flutter app to display maps. The map widget will be embedded in the Explore page; it will load map tiles from Google's servers and we can overlay markers for locations from our database.
- **Location Services:** The Flutter app (client) will access device GPS hardware (through the geolocator plugin or similar) to obtain the user's coordinates when needed (e.g., when the Home page loads or when user taps "find nearby").
- **Video Platform/Storage:** For travel tip videos, if they are hosted on YouTube, we will use the YouTube embed player or a WebView to show them. If they are custom videos and we use Supabase Storage or another CDN, the app will stream them from there.
- **Clerk & Stripe for Billing (Future):** If the app will offer premium content or subscriptions down the line, Clerk's billing (integrated with Stripe) can be used to manage subscriptions/payments [10]. This isn't a day-one feature but the architecture can accommodate it (the Clerk backend would talk to Stripe; from the app perspective, we'd just show paywall or upgrade options and let Clerk handle the heavy lifting via its SDK or API).

Workflow: The overall flow is that the user launches the Flutter app, which checks if the user is logged in (using Clerk). If not, the user is presented with a sign-up/sign-in page (Clerk's UI components can be used for a quick start, or custom Flutter forms that use Clerk's API). After authentication, the app stores the session. The main app UI (home, etc.) loads. The Flutter app then fetches data from Supabase: for example, on Home page it will query the "Places" data (attractions, restaurants, hotels) and filter by proximity to the user's location; on the Tours page it will fetch tours data, etc. These queries go over HTTPS to Supabase's endpoints (with appropriate auth headers if needed). Supabase executes SQL behind the scenes (since it's a Postgres DB) and returns JSON data that the Flutter app parses. The Flutter app then displays the data in a user-friendly format (lists, maps, etc.). Any user actions that modify data (like saving a place or creating a trip) will trigger writes to Supabase (e.g., inserting a row in a "Saved" table or "Trips" table). We will also integrate the Google Map, which is mostly on the client side: when the Explore page opens, the Flutter app initializes the Google Map widget (with our API key), and then we can add markers by pulling location data from Supabase.

This architecture is cloud-based: most data lives in the cloud (Supabase DB), and the app requires internet connectivity to fetch the latest info (although we could cache some data locally for performance/offline usage as a future improvement). The security is handled by Clerk (for user auth) and Supabase's RLS for data. The next step is to plan the database schema (entities, attributes, and relations) in detail.

Data Model and Database Design

Using a relational model (PostgreSQL on Supabase) is ideal for structured data with clear relationships. We will define tables (entities) for each major data object in the app and establish relationships (foreign keys) where necessary. Below are the key entities with their attributes and relationships:

- **User:** Represents the app's users. Since we use Clerk, basic user information (email, name, etc.) is stored in Clerk. We may still have a `users` table in Supabase to store app-specific profile info or to link to other tables. Attributes might include:
- `user_id` (UUID or Clerk's user ID, primary key)
- `name` (full name or username – could be fetched from Clerk OAuth profile)
- `email` (for reference, though Clerk manages credentials)
- Any profile preferences (e.g., preferred language or settings)

Relationship: Users can have many saved items and many trips, so there will be one-to-many relations from User to SavedItem, and User to Trip (via foreign key user_id).

- **Place:** Represents a point of interest in Sri Lanka. This can include attractions (e.g., Sigiriya Rock Fortress), hotels, restaurants, beaches, hidden gems, etc. We can have one unified table `places` with a column indicating the category/type, since these all share common attributes (name, location, description). Attributes:
- `place_id` (primary key)
- `name`
- `type` (e.g., `attraction`, `hotel`, `restaurant`, `beach`, `hidden_gem`, `event`)
- `description`
- `address` or area name
- `latitude` and `longitude` (for mapping and "near me" functionality)
- `image_url` (optional link to a photo)
- Other details like contact info or rating if relevant

This design treats all points of interest uniformly, distinguished by a type field (similar to how some travel apps use a common structure for points of interest like hotels or landmarks [11]). We might also create separate tables for certain types if their attributes differ greatly (for example, events might have a date/time), but initially a single table with type is simpler.

- **Tour:** Represents a curated tour or itinerary of multiple places. We will have a `tours` table for recommended tours (curated by the app developers/admin). Attributes:
- `tour_id` (primary key)
- `title` (e.g., "Cultural Triangle 3-Day Tour")
- `description` (summary of the tour or itinerary)
- `duration` or `days` (how long the tour is, if applicable)
- Possibly a `places_list` or we create a related table `tour_places` (see below)
- `created_by` (this could be null or an admin user ID if we want to track who created the tour; recommended tours by admin vs user-generated trips)

For the relationship of tours to places: a tour consists of multiple places in sequence. We will create a join table **TourPlace** (or itinerary stops) to model a many-to-many relationship between Tour and Place: -

`tour_id` (foreign key references Tour) - `place_id` (foreign key references Place) - `order` (an integer indicating the order of the place in the tour itinerary) This way, each tour can have an ordered list of places. For example, Tour "Best Beaches Day Trip" might have places [PlaceA, PlaceB, PlaceC] with order 1,2,3.

- **Trip (User Trip):** This represents the **"My Trips"** feature where a logged-in user can create their own custom trip/itinerary. We can use a structure similar to Tour for user-created trips:
- `trip_id` (primary key)
- `user_id` (foreign key references User – the owner of the trip)
- `title` (trip name given by user, e.g., "My Colombo Day 1")
- `description` (optional notes)
- We'll also need a join table between Trip and Place (similar to TourPlace above) to list which places the trip includes and in what order.
- Perhaps `date` or start date if the user plans on a specific date (optional).

Each user can have many trips. We enforce that only the owning user can view/edit their trips via user_id foreign key and RLS policy (or by filtering queries by user_id).

- **SavedItem (Favorites):** Represents the "saved content" feature where users bookmark items (places, tours, or videos). We can design this as a single table to handle all types of saved content:
- `user_id` (foreign key to User)
- `item_type` (e.g., `place`, `tour`, or `video`)
- `item_id` (the primary key of the item in its respective table, e.g., place_id if item_type is place)
- `saved_at` (timestamp when the user saved it)

The primary key could be composite (user_id + item_type + item_id) to prevent duplicates. This table allows a user to have many saved items. We will query it to display the user's saved places, saved tours, etc. (We might also split this into separate tables like SavedPlace, SavedTour for simplicity, but a unified table is efficient).

- **Video (Travel Tips Videos):** If we plan to manage a list of videos (tips & tricks) ourselves, we can have a `videos` table:
- `video_id` (primary key)
- `title`
- `description`
- `video_url` or `youtube_id` (to embed or link to the video)
- Perhaps `thumbnail_url` for a preview image
- Possibly a `category` if we have different types of videos (tips, cultural info, etc.)

These videos are mostly read-only content provided by us or partners. Users might save videos to their favorites (then SavedItem will reference video_id).

- **EmergencyContact:** We can have a table for emergency info if we want to store it structurally:
- `contact_id`
- `name` (e.g., "Tourist Police", "Ambulance", "Emergency Hotline")
- `phone_number`
- `description` or notes (e.g., operating hours or coverage area)

- Possibly `location` if we list nearest police stations (or we could include police stations in the Place table as a type of place).

However, since emergency info is fairly static and small, we might also just hard-code it in the app UI. A table is useful if we want to easily update it or have multiple contacts.

- **AppRecommendations:** For feature 5.1 (recommended apps for traveling Sri Lanka), we might create a simple table:
- `app_id`
- `name` (e.g., "Google Translate", "PickMe Taxi App")
- `description` (how it helps the traveler)
- `url` (link to App Store/Play Store)
- `icon_image` (icon link if we want to display logo)

These entries can be shown in a list on the Explore page as recommended third-party apps. This can also be static content, but a table allows updating the list without app updates.

Relationships Summary: Users relate to many trips, many saved items. Trips and Tours relate to Places (many-to-many via join tables). A user's saved item can reference any place, tour, or video. We will enforce referential integrity via foreign keys in Supabase (since it's PostgreSQL) so that, for example, a SavedItem referencing a place must have a valid place_id from the places table. This relational approach ensures consistency (Supabase being a Postgres-based system naturally supports these relations [3]).

We will also index certain columns (like latitude/longitude in the places table, possibly using PostGIS for geoqueries) to enable efficient querying of "nearby" locations. If enabled, Supabase's Postgres can use the earth distance or cube extension to query nearby coordinates, or we can do a bounding box query (latitude and longitude range). This will help implement the Home page's location-based query efficiently.

With the data model in place, the next steps involve designing the user interface and then implementing the features using the chosen tech stack.

UI/UX Design and App Structure

Before diving into coding, we will create a basic UI/UX design (wireframes or mockups) for the major screens. This helps ensure a good user experience and clarify the navigation structure. The main screens and their content will be:

- **Login/Signup Screen:** If the user is not logged in, present a welcome screen with options to **"Sign in with Google"**. Clerk provides pre-built UI components for sign-in we can possibly use, or we design a simple login page that triggers Clerk's OAuth flow. For the MVP, using Clerk's default UI for OAuth might speed things up.
- **Home Screen:** After login, the Home page is the default screen. It might show a greeting and a feed of nearby attractions, hotels, and restaurants. We can organize this as sections (e.g., "Attractions near you", "Hotels near you", etc.) or as a combined list sorted by distance. Each item might show name, category, distance away, and maybe a thumbnail image. The Home screen will likely use the device location to query places from the database. It should also include a search bar or filter to

change location or search for other places (optional for later). A common header or icon to open the side menu will be present.

- **Side Menu (Navigation Drawer):** A hamburger icon (≡) on the app bar can open the side drawer. The drawer will list navigation options: Home, Tours, Explore, Videos, Saved, Emergency, Settings/Profile, and Logout. This allows the user to jump to any main section quickly. We will implement this using Flutter's `Drawer` widget with `ListTile` items for each section.
- **Tours Screen:** Likely structured with tabs or segments for *Recommended*, *My Trips*, and *Saved*. For example, a TabBar can be used to switch between these three lists:
- **Recommended Tours:** Shows tours from the `tours` table (those curated by us). Each tour item shows a title, brief description or highlights, duration, etc., and maybe a representative image. Tapping a tour opens a detail page with the full itinerary (list of places included on that tour, possibly with a small map or route).
- **My Trips:** Shows the trips created by the logged-in user (from the `trips` table). Each item shows the trip name, maybe the number of places in it or a date. Users can also have an option here to "Create a new trip". Creating a trip would involve selecting places from a list or map and saving an itinerary – this can be a more advanced UI (possibly use a multi-select map interface or a form where user picks places).
- **Saved Tours:** Shows any tours (from the Recommended list) that the user has saved/bookmarked.
- **Explore Screen (Map):** This screen contains a full-screen Google Map widget. We will show the user's current location on the map (blue dot) and various markers for points of interest:
- We can categorize markers by type (use different marker colors or icons for attractions vs hotels vs beaches vs events). A legend or filter button can allow toggling categories.
- When the user taps on a marker, we show a small info card (bottom sheet or popup) with the place name and maybe an image, and options to view details or save it.
- There could also be a list view that the user can slide up from the bottom to see a list of places (optional).
- The Explore screen might have an overlay UI with a button or menu for selecting what to display (e.g., "Show: [✔] Attractions [✔] Hotels [✔] Events").
- For **Recommended Apps (5.1)**: we can include a section either overlaying the map or as a separate tab/section on the Explore page. Perhaps a floating panel or a bottom section listing some recommended travel apps (with icons and descriptions), since this is more informational. Alternatively, we could put this list in the side menu or Settings, but the user specifically listed it under Explore, so likely it will be seen on the Explore page, maybe accessible via a button like "Travel Apps".
- **Videos Screen (Tips & Tricks):** A scrollable list of travel tip videos. Each item shows a thumbnail, title, and maybe a short description or duration. Tapping an item opens the video player page. The video can play within the app (embedding a YouTube player or a video widget). We should ensure the videos open in landscape/fullscreen mode if needed for better viewing. This section is mostly static content that we provide, but we might update it over time by adding new videos to the `videos` table.
- **Saved Items Screen:** This page compiles all of the user's saved content. One UI approach is to have tabs for *Places*, *Tours*, and *Videos* within Saved, or a segmented list labeling each item's type. For example, it might show a list: if an item is a place, display it with a location icon; if it's a tour, maybe a route icon; if video, a play icon. The user can tap any item to view its details (place detail page, tour detail, or play video). This screen basically queries the SavedItem table for the user and then fetches details for each item (we might do a join or multiple queries).

- **Emergency Screen:** A simple but crucial page. It can list emergency phone numbers and information. We'll present items like "Tourist Police: 119" with a phone icon to tap and call (Flutter can initiate a phone dialer intent). We can also include general advice for emergencies. If we have multiple contacts (police, ambulance, hotline, embassy contacts), they can be listed here. Additionally, we might include a map of nearby police stations or hospitals – that could be a future enhancement; for now, a static info page suffices. We ensure this page is accessible quickly (maybe an SOS icon somewhere or at least via the menu) for user safety.
- **Settings/Profile:** Not explicitly listed by the user, but typically an app would have a profile or settings page (language switch, app info, etc.). Since currently only English is needed, a language setting is not required now. We may include a simple profile page showing the user's name/email (fetched from Clerk) and perhaps an option to log out. This could be accessible from the side menu.

In terms of style, we will ensure a clean, travel-themed UI. Maybe use a pleasant color scheme with imagery of Sri Lanka (beaches, cultural sites) as header images or icons. We should also design app icon and logo to give it a unique branding (maybe an icon with Sri Lanka map outline or a famous landmark).

Now that the design and structure are clear, we proceed with the development steps.

Development Plan (Step-by-Step)

Now we outline the steps to go from concept to a deployed app, including setting up the system architecture components, implementing features, and testing.

1. **Requirement Analysis & Planning:** Finalize the feature set, gather any content data (list of attractions, hotels, emergency contacts, etc.), and define scope for the first version. This includes confirming that we will support both Android and iOS (yes, using Flutter covers both), confirming use of English language initially (as noted), and identifying any external data sources needed (e.g., for events or places). Create a development timeline and task breakdown.

2. **Set Up Project Environment:** Initialize a Flutter project (e.g., using `flutter create`). Set up version control (Git repository) to track our code. Configure the app's package name, and set up Flutter dependencies for our tech stack:

3. Add the Supabase Flutter SDK (`supabase_flutter`) to `pubspec.yaml` [12].
4. Add the Clerk Flutter SDK (`clerk_flutter`) to `pubspec.yaml`.
5. Add Google Maps Flutter plugin (`google_maps_flutter`) [6].
6. Add geolocation plugin (`geolocator`) for location features.

7. Any other needed plugins (e.g., `url_launcher` to open external links for recommended apps or to dial phone numbers on the emergency page). After adding, run `flutter pub get` to install packages. On the native side, we'll also follow platform-specific setup: for Google Maps, update the Android `AndroidManifest.xml` and iOS `AppDelegate` with the API key as per Google's instructions [13]. For geolocation, we may need to request permissions (so include the location permission in AndroidManifest and Info.plist).

8. **Design the Database (Supabase) Schema:** Create a new Supabase project (through Supabase web console). In the Supabase dashboard, define the tables as per our data model:

9. Create tables: users (if needed), places, tours, tour_places, trips, trip_places, saved_items, videos, emergency_contacts, app_recommendations.
10. Define primary keys and foreign keys (for example, `tour_places.tour_id -> tours.id`, `tour_places.place_id -> places.id`, etc.).
11. Insert initial data: Since our app content is curated, we will need to populate the `places` table with data for Sri Lanka's attractions, hotels, etc. (This could be a sizable task – possibly hundreds of entries – but we can start with a subset for testing). Also populate emergency contacts, a few recommended tours, some videos, and recommended apps entries. Supabase's Table Editor or SQL insert scripts can be used for this.
12. Enable Row Level Security (RLS) on tables that contain user-specific data (users, trips, saved_items). Then create RLS policies to allow appropriate access:
    - For `trips` and `saved_items`, allow the owning user (where user_id matches the logged-in user's UID) to select/modify their records, and perhaps allow no anonymous access (since those require login). Public data like `places`, `tours`, `videos` can be readable by anonymous users [9].
    - Since we use Clerk for auth, one approach is to create a service role for the Flutter app to interact with Supabase securely. Alternatively, we might use a JWT from Clerk: we'd then need to configure Supabase to trust a JWT (maybe not trivial out-of-the-box; possibly easier is to use Supabase's own auth for users and map Clerk users to it, but to keep things simpler initially, we might use the service role or open RLS for reading public data).

13. Test the database by using the Supabase Query editor or a simple script to ensure data can be retrieved as expected (e.g., a sample query to get places near a certain lat/long if using PostGIS).

14. **Integrate Authentication (Clerk) in Flutter:** Configure Clerk for our app:

15. Create a Clerk application via the Clerk dashboard and obtain the Publishable Key for the Flutter app.
16. In Flutter, initialize the Clerk SDK with our app's publishable key (as shown in Clerk's example code) [14]. This is typically done in the `main.dart` before runApp.
17. Implement the authentication UI. The simplest way is to use `ClerkAuthentication()` widget (from the SDK) which provides a full pre-built sign-in interface [15]. This will handle Google OAuth internally if configured. We have to ensure Google OAuth is enabled in the Clerk dashboard for our app (and possibly set up Google API credentials if moving to production [16]).
18. Alternatively, use a WebView or redirect for Google Sign-In if needed (Clerk's docs note that Google might block OAuth via WebViews [17], so Clerk likely uses an external browser or Native flow).
19. Once the user signs in, Clerk will provide a user object or session token. We may store basic user info in our `users` table at this point (e.g., on first login, create an entry if not exists). Clerk's backend can also trigger webhooks on new user sign-up if we want to sync to Supabase, or we can do it client-side: e.g., after successful sign-up, call a Supabase RPC or function to insert the user.
20. Ensure that the app can detect the auth state: show the login screen if not authenticated, or proceed to main app if authenticated (Clerk SDK likely has a method to check if a user is signed in).

21. Test the Google sign-up flow on a device/emulator to confirm it works (Clerk provides Google OAuth in dev mode without extra setup [18], but in production we'd put proper keys).

22. **Implement App Navigation (Side Menu and Routes):** Set up the basic navigation structure of the app:

23. Use a **Drawer** widget in Scaffold for the side hamburger menu. Define the menu items corresponding to the main screens. Each menu item onPressed will navigate to the respective screen.
24. Use Flutter's named routes or a navigator to handle screen transitions. We might create a `HomePage`, `ToursPage`, `ExplorePage`, `VideosPage`, `SavedPage`, `EmergencyPage`, etc. For simplicity, we can use a `Drawer` in a single `Scaffold` that swaps the body based on selection, or use a Navigator push for each page. A common approach is to have a main Scaffold with the drawer and use a `PageView` or conditional to show the selected page widget.
25. Ensure the AppBar has the menu icon to open the drawer on each main screen. Also, add a title that changes based on which section is active (e.g., "Home", "Explore").
26. If needed, also include a bottom navigation bar for quick switch between main sections (this is optional if using side menu exclusively, but some apps use both). Given the side menu covers it, we might stick to just the drawer to avoid redundancy.

27. Test navigation: clicking each drawer item should correctly navigate to the right page. Also handle back button (Android) behavior gracefully – if using push navigation, pressing back from a main page could exit the app (which is okay if Home is the root).

28. **Home Page – Location and Nearby Places:** Develop the functionality to show nearby attractions/ hotels on Home:

29. **Location Permission:** Use geolocator to request location permission from the user the first time. Handle the permission grant or denial. If denied, we might show a message that location is needed for nearby feature, and handle gracefully.
30. **Get Current Position:** Once permission is granted, get the user's current latitude/longitude using `Geolocator.getCurrentPosition()` [19]. This yields coordinates.
31. **Query Nearby Places:** Call Supabase to retrieve places near that location. If we have PostGIS and a function, we could do something like `SELECT * FROM places WHERE earth_distance(ll_to_earth(lat, lon), ll_to_earth($userLat, $userLon)) < some_radius`. Alternatively, if not using PostGIS, we can fetch all or a subset of places and compute distances in Dart. A simpler approach: store each place with a region or city field, and query by region if we can infer user region by lat/long. But ideally, implement at least a basic radius filter. For MVP, we might query all places and filter the nearest N in the app for simplicity if performance is okay (maybe not too many places).
32. **Display Results:** Use a ListView.builder to show the list of nearby places. Sort them by distance (we can calculate distance using haversine formula in Dart if not done in SQL). Each list item shows the place name, type (maybe as a colored tag or icon), and distance (in km or meters). Possibly also show a thumbnail image if available.
33. **Interaction:** Tapping a place could open a Place Details page (which can show more info and maybe a small map focused on that place, plus a Save button). We should implement a simple place detail screen that pulls all info of that place from the DB and offers "Save to Favorites" (which inserts into SavedItem) and maybe "Add to Trip" (if user wants to add it to a trip).
34. This page should refresh or update if the user's location changes significantly (or provide a pull-to-refresh to re-check location and update list).

35. Test by simulating different locations (if possible) or using known coordinates to see that the filtering works.

36. **Tours Page – Recommended & My Trips:** Implement the Tours section:

37. Use a TabBar (Flutter's DefaultTabController) to create three tabs: Recommended, My Trips, Saved.
38. **Recommended Tours:** Query Supabase for all entries in `tours` table. Display them in a list with cards. Each card might show the tour title, a representative image (maybe image of one of the places or a generic image for the tour), and short description. Also maybe the number of days or stops.
    - Implement the Tour Detail page: when a tour is tapped, navigate to a detail screen. On this screen, fetch the list of places in that tour (join `tour_places` with `places`). Display them in order (could be a numbered list of stops, with each item showing place name, maybe an icon or thumbnail). Possibly provide a map overview of the tour route by showing the path between the places on a small map widget (advanced, can skip initially). Also include a button "Save Tour" which if tapped will add an entry in SavedItem (item_type=tour).
39. **My Trips:** This tab will show trips created by the logged-in user. Initially, this list might be empty for new users. Provide a prominent **"Create Trip"** button or card. For listing existing trips, similar UI to tours (title, maybe how many places or a cover image if we assign one).
    - Creating a Trip: We need a UI flow for this. It could be a form where the user enters a trip name and optionally a description or date. After creating the trip entry, we allow the user to add places. One way: show a list of all places (or a searchable list) with checkboxes to add to trip, or allow user to pick via the map (select markers and add to trip). A simpler approach is a multi-select list: for MVP, we might let the user add places by searching names. This is a complex feature; to keep it simpler, we could postpone a fancy UI and just allow adding by search.
    - Editing a Trip: Once a trip is created, users might want to reorder or remove places. We can implement dragging to reorder if time permits (using Flutter's ReorderableListView).
    - Trip Detail View: similar to Tour Detail, showing all places in the trip. Also allow "Delete Trip" or "Edit Trip" actions for the owner.
40. **Saved Tours:** List tours that the user saved. We can fetch from SavedItem where user_id = current and item_type = 'tour', then join with tours table to get tour details. UI is similar to Recommended list, possibly with a label or badge indicating it's saved. The user can tap to view details (same Tour Detail page). Also, allow "Unsave" (which would delete the SavedItem entry) perhaps via a swipe action or an icon.

Test the Tours page thoroughly: ensure data loads correctly in each tab, and that creating a trip and saving tours works (verify the database entries are created/removed properly). This page involves both read and write operations to the DB (creating trips, saving/unsaving), so it's critical to handle errors (e.g., no duplicate saving, etc.) gracefully.

1. **Explore Page – Map and Discover:** Implement the Explore page with Google Maps integration:
2. **Google Map Setup:** Initialize the GoogleMap widget in the Explore screen. Provide it an `initialCameraPosition` (perhaps centered on Sri Lanka at a reasonable zoom level, or on the user's location if available). Ensure the map renders correctly by running on both Android and iOS (requires the API key setup done earlier).
3. **Markers for Places:** Fetch place data from Supabase to display on the map. We might load all places for the map (and possibly filter by type). If there are very many points, consider clustering or limiting to a certain area, but Sri Lanka is not huge and if we have, say, a few hundred points, it should be manageable. Use the Google Maps plugin's Marker API to add markers. We can iterate through the list of places and create a Marker for each with its position (lat, lng) and a custom icon based on category (we can use different colored pins).

4. **Marker Interactivity:** Implement `onTap` for markers to show an info window or, better, use Flutter overlays (like showModalBottomSheet) to display a small panel with the place details. The panel can show name, type, maybe image, and buttons for “View Details” and “Save”. If info windows (the built-in ones) are easier, we can start with that – but customizing them might be limited, so a bottom sheet is more flexible.
5. **Filter Controls:** Add UI controls to filter which markers are shown. For example, have a floating Action Button or an expandable chips/buttons to toggle categories (Attractions, Hotels, Restaurants, etc.). When a filter is toggled, update the markers to show/hide accordingly. This can be as simple as maintaining a set of active types and when building markers, skip those not active.
6. **User Location on Map:** If permission was granted earlier, we can enable the `myLocationEnabled` property on GoogleMap to show the blue dot for user location. Also a button to center map on current location would be useful.
7. **Recommended Apps Section:** We need to incorporate the list of recommended travel apps (5.1) on this screen. Since it’s unrelated to the map directly, one approach is to have a bottom sheet or panel (maybe scrollable) that contains the list of apps. For instance, a section below the map (if screen space allows) titled "Recommended Apps for Traveling Sri Lanka". Each item can display the app name, an icon (we may use a predefined image or fetch from URL), and possibly a short note like “Useful for taxi bookings” etc. Tapping it could use `url_launcher` to open the app’s Play Store/App Store page.
8. Alternatively, a separate screen for recommended apps could be triggered by a button. But to follow the spec list, including it on the Explore page is likely expected. We will ensure it’s visible – maybe the Explore page is a scrollable Column where the top half is the map and below that is the list of apps.

9. Test the map thoroughly: verify markers appear at correct locations, filters work, tapping a marker shows the detail popup, and that performance is acceptable (especially on older devices). Also test that the recommended app links work (on a device, opening the store or browser).

10. **Videos Page – Tips & Tricks:** Develop the video listing and playback:

11. Query the `videos` table from Supabase to get the list of travel tip videos. Alternatively, if we have a static list of YouTube links, we could hardcode them, but using the database allows easy content updates. Ensure that for each video we have either a direct URL or an identifier.
12. Display the list in a ListView. Each list item shows a thumbnail image (if we have a URL, or we could use YouTube API to fetch thumbnail by ID), title, and maybe a subtitle.
13. When a user taps a video, navigate to a Video Player screen. If using YouTube links, the simplest way is to open the YouTube app or a WebView to that video URL. However, a better in-app experience is to use a Flutter video player widget or a YouTube player plugin. For instance, `youtube_player_flutter` package allows embedding YouTube players in the app. We must handle the orientation changes if full-screen, etc.
14. Provide controls to play/pause, and an option to save the video to favorites (maybe a heart icon that adds to SavedItem).
15. Ensure to handle network connectivity (videos should show a loading indicator if they buffer). Also, possibly allow users to rotate to landscape for full-screen.

16. Test video playback on both platforms, as well as the transition back and forth between the list and player screen.

17. **Saved Items Page – Favorites:** Implement the Saved page to show all user-saved content:

    - Query the `saved_items` table for the current user. We can either:
    - Do separate queries for each type (e.g., get all saved places with a join on places, get all saved tours with join on tours, etc.), then combine, or
    - Fetch all saved_items and then for each, fetch the details of the referenced item. This could be done in parallel using multiple futures.
    - An elegant approach is to do it in sections: e.g., show saved places in one section (if any), saved tours in another, saved videos in another. Use headers to label each section.
    - Or use TabBar inside Saved page similar to Tours: one tab for Places, one for Tours, one for Videos.
    - For each saved item, display it similar to how it appears in its original context (for places, maybe a small map thumbnail or icon; for tours, a route icon; for videos, a play icon).
    - Allow unsaving: Each item can have a remove button (like a trash or filled heart icon to toggle). When tapped, remove that entry from saved_items and refresh the list.
    - Tapping an item should navigate the user to the appropriate detail page (we can actually reuse the same detail pages we made: place detail, tour detail, video player).
    - Test saving and unsaving from various parts of the app (Home, Tour detail, Video, etc.) and see that Saved page reflects the changes accordingly. Also verify that if an item is removed, it disappears from Saved page and possibly the original pages should update their saved icon state.

18. **Emergency Page – Contacts and Info:** Implement the Emergency section:

    - This page might not require dynamic data fetch if we choose to hardcode the info. However, to be consistent, we could fetch from `emergency_contacts` table.
    - Display each contact with a large icon (like a phone or warning icon), name of service, and phone number. Use `url_launcher` to dial the number when tapped. For example, tapping "Tourist Police – 119" should open the phone dialer with 119 pre-filled.
    - If we include addresses or maps (for police stations, hospitals), we could list them and allow tapping to open in Google Maps app via a URL (`geo:` intent or Google Maps URL scheme).
    - Also provide any brief advice (maybe as text at bottom, e.g., “For any emergency, you can also call 112 for general emergency services.”).
    - Since safety is critical, perhaps also include a one-tap dial for the main emergency line (like an SOS button).
    - Test the phone dialing on a real device because emulator might not support call, but ensure the intent is formed correctly.

19. **Testing and Quality Assurance:** Perform thorough testing after implementing all features:

    - **Unit Tests:** Write unit tests for critical utility functions (if any, e.g., distance calculation). Since much of our app is UI with backend integration, unit testing might be limited. Focus on testing any custom logic like filtering or data parsing.
    - **Integration Tests:** If possible, write integration tests using Flutter’s integration test framework to simulate user flows (e.g., login, view home list, save an item, see it in saved list).
    - **Manual Testing:** Go through each feature on both Android and iOS (use emulators and actual devices if available). Test various scenarios:
    - New user signup and returning user login.

    - Permissions: deny and grant location permission to see how the app reacts.
    - No internet scenario: the app should handle gracefully (show a message or cached data if offline).
    - Data correctness: verify that the data shown (places, tours, etc.) matches the database content.
    - Performance: scroll through lists, move the map, ensure the app remains responsive. Optimize any slow parts (e.g., maybe add pagination or lazy-loading if the list of places is very long).
    - Security: Ensure that a user cannot see another user's trips or saved items (try manipulating requests if possible, or observe that filtering by user_id works).
    - Fix any bugs encountered. Pay attention to platform-specific issues (like differences in Google Maps behavior on iOS vs Android, or any layout issues).

20. **Deployment Preparations:** Before publishing the app:

    - Register developer accounts on Google Play Store and Apple App Store if not already.
    - Prepare app metadata (name, description, screenshots). An app name could be something like "Travel Lanka" or "SriLanka Explorer" – ensure it's unique enough.
    - Set up app icons and branding. Flutter's `pubspec.yaml` and native config will need the new app icon (we can use Flutter's `flutter_launcher_icons` package to generate icons for all platforms easily).
    - Build release versions: For Android (generate an APK/AAB, sign it with a keystore), for iOS (Xcode archive and upload to TestFlight).
    - On the backend side, ensure the Supabase project's API keys are properly used:
    - In Flutter, we used an anon public key for Supabase (which is fine for client-side with RLS). Make sure that is configured and not exposing any unintended data.
    - For Clerk, ensure we use production publishable key and domain for production environment.
    - Perform final sanity checks on release builds (especially OAuth redirection on mobile might need certain app scheme configuration if using web redirect; Clerk's docs should cover mobile OAuth setup).
    - Deploy the app to beta testers if possible, get feedback, then proceed to release in the app stores following their guidelines.

21. **Maintenance and Future Enhancements:** After initial release, plan for maintaining and improving the app:

    - **User Feedback:** Gather feedback from users and fix any usability issues or bugs. Monitor crashes or errors (integrating a tool like Sentry for Flutter can help get runtime crash reports).
    - **Content Updates:** Continue to update the places database (Supabase) with new attractions or events. Perhaps build an admin interface (could be a simple web interface or even using Supabase Studio) to manage content.
    - **Multi-language Support:** Since currently English is used, plan how to add other languages (Sinhalese, Tamil, etc.) if targeting a broader audience, especially if local users might use it. Flutter's internationalization features can be used to provide translations for UI text. Content like descriptions of places might need translation as well, which could be stored in the database or handled via a service.

- ◦ **Offline Mode:** Consider implementing offline capabilities for key data. For example, using cached data or a local database (SQLite or Hive). Supabase doesn't automatically cache offline, but we can fetch and store certain data (like saved places or downloaded map areas) so the app remains useful with poor connectivity.
- ◦ **Enhanced Features:** Potential future features include: in-app booking links for hotels or tours, user reviews and ratings for places, push notifications for events or travel advisories, social features (sharing trips or connecting with other travelers), etc. We should ensure the architecture (especially the database schema) can extend to these if needed (for instance, adding a reviews table).
- ◦ **Performance Tuning:** If usage grows, ensure the database is indexed properly (e.g., on place coordinates, on foreign keys for faster joins). Supabase can scale the DB and we might use caching strategies for heavy read operations (like Cloudflare caching for Supabase REST if open).
- ◦ **Security Audits:** Regularly review security – Clerk will handle auth security, but we must ensure our implementation of data access is secure (no leaky endpoints, etc.). Also keep dependencies up to date for security patches.
- ◦ **Monitoring & Analytics:** Integrate analytics to see which features are used (e.g., Firebase Analytics or others) to understand user behavior. Also monitor Supabase logs or use logging to see if any errors occur in API calls.

By following this detailed plan – from initial design through development steps and testing – we can systematically build the Sri Lanka travel app. We leveraged a modern tech stack (Flutter, Supabase, Clerk) to reduce boilerplate and focus on features. Flutter gives us a smooth cross-platform UI [1], Supabase provides a powerful Postgres backend with minimal setup [3], and Clerk handles the critical auth flows [5] including Google OAuth. Throughout development, keeping the user experience in mind (quick access to information, easy saving of favorites, and reliable offline-ready functionality) will ensure the app is well-received by travelers.

Finally, we will document everything (code, architecture decisions, how to operate the content management for the app) so that the app can be maintained and scaled by the development team. With this comprehensive approach, we can confidently proceed to create the travel app for Sri Lanka, meeting all the listed requirements and setting a strong foundation for future growth of the product.

Sources:

- Flutter cross-platform capabilities [1] [2]
- Clerk Flutter SDK for authentication [5]
- Supabase architecture (PostgreSQL core and REST API) [3] [4]
- Google Maps Flutter integration steps [6]
- Flutter Geolocator plugin for GPS location [7]
- Couchbase travel sample data model (point-of-interest types) [11]

[1] [2] Top Reasons to Choose Flutter for Travel App Development
https://kodytechnolab.com/blog/flutter-for-travel-app-development/

[3] [4] [8] Architecture | Supabase Docs
https://supabase.com/docs/guides/getting-started/architecture

[5] [14] [15] clerk_flutter | Flutter package
https://pub.dev/packages/clerk_flutter

[6] [13] Adding Google Maps to Flutter. This article will show you step-by-step... | by Kenzie Davisson | Flutter | Medium
https://medium.com/flutter/google-maps-and-flutter-cfb330f9a245

[7] geolocator | Flutter package - Pub.dev
https://pub.dev/packages/geolocator

[9] [12] Use Supabase with Flutter | Supabase Docs
https://supabase.com/docs/guides/getting-started/quickstarts/flutter

[10] Instant, zero-integration SaaS billing with Clerk + Stripe
https://stripe.com/sessions/2025/instant-zero-integration-saas-billing-with-clerk-stripe

[11] Travel App Data Model | Couchbase Docs
https://docs.couchbase.com/c-sdk/current/ref/travel-app-data-model.html

[16] [17] [18] Sign-up & Sign-in: Add Google as a social connection
https://clerk.com/docs/authentication/social-connections/google

[19] A Comprehensive Guide to Implementing Flutter Geolocator
https://www.dhiwise.com/post/maximizing-user-experience-integrating-flutter-geolocator