Smart Trip Planner Flutter - High Level & Low Level Design

High-Level Architecture

- Presentation / UI Layer: Screens, UI widgets, forms, navigation flows.
- Application / Domain Layer: Business logic and use cases like generate trip suggestions, save trip, load trips.
- Data / Infrastructure Layer:
- - Remote (AI + Firebase): Handles Gemini AI queries and Firebase storage.
- - Local (Hive): Caches trips, chat history, and offline data.
- Cross-Cutting Services: AI provider, chat manager, data sync, connectivity utilities.
- Flow Example: User → UI → Use Case → AI/Firebase → Hive → UI.

Low-Level / Clean Architecture

- Domain Layer:
- - Entities: Trip, Suggestion, ChatMessage.
- - Repositories: TripRepository, AiProvider.
- Application Layer:
- - Use Cases: GenerateTripSuggestions, SaveTrip, GetLoggedTrips, LoadChatHistory.
- Data Layer:
- - HiveTripRepository, FirebaseTripRepository, GeminiAiProvider.
- - DTOs for mapping between local and domain entities.
- Presentation Layer:
- - State Management (Riverpod).
- - Widgets and Pages for UI states.
- Infrastructure: Connectivity detection, Hive adapters, Firebase clients.

Suggested Folder Structure

```
lib/
├── app/                    # App-level configurations
│   ├── app.bottomsheets.dart     # Bottom sheet configurations
│   ├── app.dart                  # Main app configuration
│   ├── app.dialogs.dart          # Dialog configurations
│   ├── app.locator.dart          # Dependency injection setup
│   └── app.router.dart           # Navigation routes setup
│
├── data/                   # Data layer
│   ├── models/               # Data models
│   │   ├── itinerary_model.dart
│   │   └── saved_conversation.dart
│
├── services/               # Business logic and services
│
├── ui/                     # User interface layer
│   ├── bottom_sheets/          # Bottom sheet UI components
│   ├── common/               # Shared UI components
│   │   ├── app_colors.dart     # Color constants
│   │   └── widgets/          # Reusable widgets
│   │       ├── ai_avatar.dart
│   │       └── user_avatar.dart
│   │
│   ├── dialogs/              # Dialog UI components
│   └── views/              # Main screen views
│       ├── followup_itinerarie/  # Follow-up itinerary screen
```

```
│   │   ├── followup_itinerarie_view.dart
│   │   └── followup_itinerarie_viewmodel.dart
│   │
│   ├── home/            # Home screen
│   │   ├── home_view.dart
│   │   └── home_viewmodel.dart
│   │
│   ├── itinerary/       # Itinerary screen
│   │   ├── itinerary_view.dart
│   │   └── itinerary_viewmodel.dart
│   │
│   ├── startup/         # App startup screen
│   │   ├── startup_view.dart
│   │   └── startup_viewmodel.dart
│   │
│   └── user_name/       # User name input screen
│       ├── user_name_view.dart
│       └── user_name_viewmodel.dart
│
└── main.dart            # Entry point of the application
```

Why This Matters

- Separation of Concerns: Independent UI, logic, and data layers.
- Testability: Use cases and repositories can be unit tested.
- Scalability: Easy to add features like offline sync or analytics.
- Maintainability: Cleaner code reduces complexity.