Smart Trip Planner Flutter - Architecture Documentation

1. Architecture Overview

The application follows the MVVM (Model-View-ViewModel) architecture pattern using the Stacked framework. Here's a detailed breakdown:

```
Architecture Layers:

┌─────────────────┐
│    UI Layer     │ ← Views, Widgets, UI Components
├─────────────────┤
│ ViewModel Layer │ ← Business Logic, State Management
├─────────────────┤
│  Service Layer  │ ← Data Operations, API Calls
├─────────────────┤
│   Data Layer    │ ← Models, DTOs, Local Storage
└─────────────────┘
```

2. complete folder structure of the lib directory:

- 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

- **This structure follows the MVVM (Model-View-ViewModel) architecture pattern with Stacked framework:**
  1. app/ - Contains app-level configurations
     - Navigation setup
     - Dependency injection
     - Dialog and bottom sheet configurations
  2. data/ - Contains all data-related code
     - Data models
     - DTOs (Data Transfer Objects)
     - Repository implementations
  3. services/ - Contains business logic
     - API services
     - Local storage services
     - Other business services
  4. ui/ - Contains all UI-related code
     - views/ - Screen implementations (each with its view and viewmodel)
     - common/ - Shared UI components and styles
     - bottom_sheets/ - Bottom sheet implementations
     - dialogs/ - Dialog implementations
- Each view follows the MVVM pattern with:
  - *_view.dart - UI implementation (View)
  - *_viewmodel.dart - Business logic for the view (ViewModel)
- The project uses the Stacked framework which provides:
  - Dependency injection (via app.locator.dart)
  - Navigation (via app.router.dart)
  - State management (via ViewModels)
- This structure makes the code:
  - Modular and maintainable
  - Easy to test
  - Scalable
  - Clear separation of concerns
  - Easy to navigate and understand

2. Core Components

2.1 App Configuration (app)

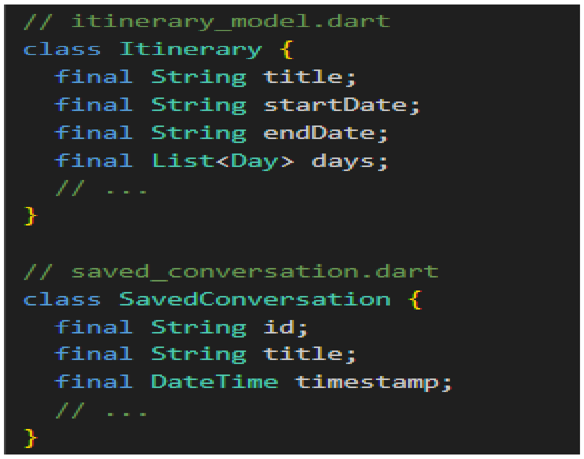
```
// app.dart - Main app configuration
class App extends StatelessWidget {
  // Configures the application theme, routes, and initial setup
}

// app.locator.dart - Dependency injection setup
@StackedApp(
  routes: [...],
  dependencies: [...],
)
class App { }
```

Key Files:

- app.dart - Application entry point and configuration
- app.router.dart - Navigation route definitions
- app.locator.dart - Dependency injection container
- app.dialogs.dart - Dialog service configuration
- app.bottomsheets.dart - Bottom sheet service configuration

2.2 Data Layer (data)

Models and Data Structures:



```
// itinerary_model.dart
class Itinerary {
  final String title;
  final String startDate;
  final String endDate;
  final List<Day> days;
  // ...
}

// saved_conversation.dart
class SavedConversation {
  final String id;
  final String title;
  final DateTime timestamp;
  // ...
}
```

2.3 Services Layer (services)

Business Logic and Data Operations:

```
class NavigationService {
  Future<void> navigateTo(String route);
  Future<void> back();
  // ...
}

class DialogService {
  Future<DialogResponse> showDialog();
  // ...
}
```

2.4 UI Layer (ui)

Views Structure

Each view follows the MVVM pattern:

```
// Example: user_name_view.dart
class UserNameView extends StackedView<UserNameViewModel> {
  @override
  Widget builder(context, viewModel, child) {
    // UI implementation
  }
}

// Example: user_name_viewmodel.dart
class UserNameViewModel extends BaseViewModel {
  // Business logic and state management
}
```

Common UI Components (common)

```
// app_colors.dart
class AppColors {
  static const Color primary = Color(0xFF00584D);
  static const Color background = Color(0xFFF8F9FA);
  // ...
}

// widgets/user_avatar.dart
class UserAvatar extends StatelessWidget {
  // Reusable avatar widget
}
```

3. Key Features Implementation

3.1 Navigation Flow

```
graph LR
    A[Startup] --> B[User Name]
    B --> C[Home]
    C --> D[Itinerary]
    D --> E[Followup Itinerary]
```

3.2 State Management

```
// Using Stacked for state management
class HomeViewModel extends BaseViewModel {
  // Reactive state management
  String _userName = '';
  String get userName => _userName;

  void setUserName(String name) {
    _userName = name;
    notifyListeners();
  }
}
```

3.3 Data Persistence

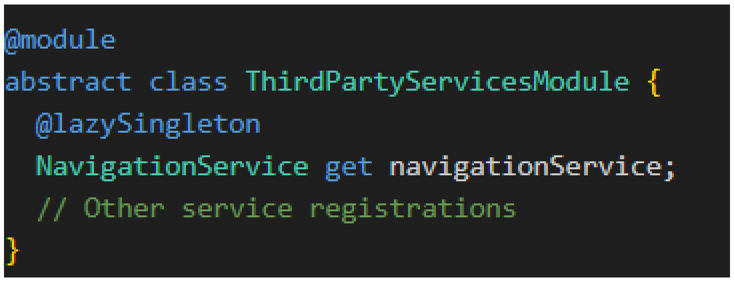
```
// Using Hive for local storage
class StorageService {
  Future<void> saveData(String key, dynamic value) async {
    final box = await Hive.openBox('appBox');
    await box.put(key, value);
  }
}
```

4. Design Patterns Used

4.1 MVVM Pattern

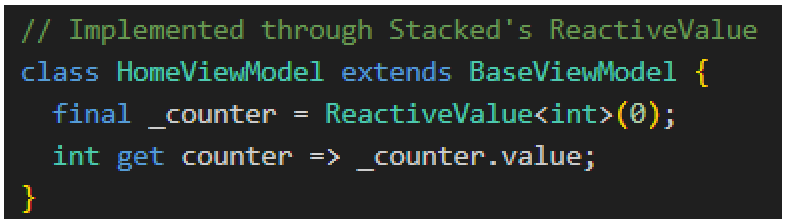
- **View**: UI implementation (*_view.dart)
- **ViewModel**: Business logic (*_viewmodel.dart)
- **Model**: Data structures (models)

4.2 Dependency Injection



```
@module
abstract class ThirdPartyServicesModule {
  @lazySingleton
  NavigationService get navigationService;
  // Other service registrations
}
```

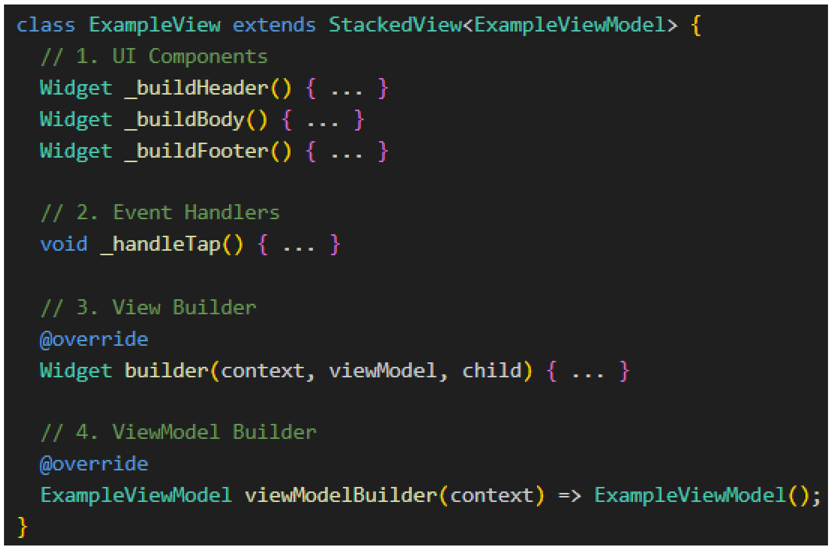
4.3 Observer Pattern



```
// Implemented through Stacked's ReactiveValue
class HomeViewModel extends BaseViewModel {
  final _counter = ReactiveValue<int>(0);
  int get counter => _counter.value;
}
```

5. View Structure

Each view follows this structure:

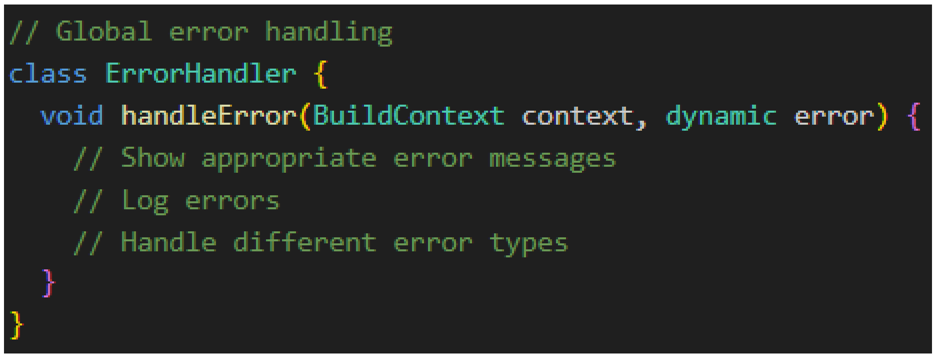


```
class ExampleView extends StackedView<ExampleViewModel> {
  // 1. UI Components
  Widget _buildHeader() { ... }
  Widget _buildBody() { ... }
  Widget _buildFooter() { ... }

  // 2. Event Handlers
  void _handleTap() { ... }

  // 3. View Builder
  @override
  Widget builder(context, viewModel, child) { ... }

  // 4. ViewModel Builder
  @override
  ExampleViewModel viewModelBuilder(context) => ExampleViewModel();
}
```

6. Application Flow

1. **Startup**
   - App initialization
   - Check user authentication
   - Load saved preferences
2. **User Name Entry**
   - Collect user information
   - Store in local storage
   - Navigate to home
3. **Home Screen**
   - Display saved itineraries
   - Option to create new itinerary
   - Profile management
4. **Itinerary Creation**
   - AI-powered trip planning
   - Save and share options
   - Follow-up modifications

7. Error Handling



```dart
// Global error handling
class ErrorHandler {
  void handleError(BuildContext context, dynamic error) {
    // Show appropriate error messages
    // Log errors
    // Handle different error types
  }
}
```

8. Testing Structure

```dart
// Unit Tests
void main() {
  group('HomeViewModel Tests', () {
    test('should update user name', () {
      // Test implementation
    });
  });
}

// Widget Tests
void main() {
  testWidgets('HomeView displays user name', (tester) async {
    // Widget test implementation
  });
}
```

9. Performance Considerations

1. **Widget Optimization**
   - Use const constructors
   - Implement shouldRebuild in custom widgets
   - Minimize rebuilds using ValueNotifier
2. **Memory Management**
   - Proper disposal of controllers and listeners
   - Image caching and optimization
   - Stream subscription management

//END////////////////////////////////////