# Wayfare Android Application - Comprehensive Documentation

**Document Version:** 2.1
**Last Updated:** June 2025
**Architecture:** MVVM with Clean Architecture principles
**Project:** Wayfare Android Travel Planning Application
**Package:** com.zeynekurtulus.wayfare
**Min SDK:** 26 (Android 8.0)
**Target SDK:** 36 (Android 14)
**Version:** 1.0

---

## Table of Contents

---

## Project Overview

### What is Wayfare?

Wayfare is a modern Android travel planning application that serves as a comprehensive platform for travelers to discover destinations, plan detailed itineraries, and share their travel experiences with a community of fellow travelers. The app combines intelligent trip planning, destination discovery, and social features to create a complete travel ecosystem.

### Primary Goals

The application aims to: - **Simplify Travel Planning**: Provide users with an intuitive, step-by-step trip creation process - **Enable Discovery**: Help users find interesting destinations and experiences - **Foster Community**: Allow travelers to share experiences and learn from others - **Ensure Reliability**: Deliver a stable, performant, and secure travel planning experience

### Target Users

- **Individual Travelers**: People planning personal trips and vacations
- **Travel Enthusiasts**: Users interested in discovering new destinations
- **Community Members**: Travelers who want to share experiences and read reviews
- **Trip Planners**: Users who enjoy creating detailed travel itineraries

---

## Architecture & Design Patterns

### MVVM (Model-View-ViewModel) Pattern

The application follows the MVVM architectural pattern, which provides clear separation of concerns:

#### Model Layer

- **Domain Models**: Core business objects (Route, Place, User, Feedback)
- **Data Transfer Objects (DTOs)**: API request/response structures
- **Repository Interfaces**: Define data operations contracts

#### View Layer

- **Activities**: Full-screen UI containers (MainActivity, LoginActivity, etc.)
- **Fragments**: Reusable UI components (HomeFragment, SearchFragment, etc.)
- **Adapters**: RecyclerView adapters for displaying lists
- **Custom Views**: Specialized UI components

#### ViewModel Layer

- **ViewModels**: Manage UI state and business logic
- **LiveData**: Observable data holders for UI updates
- **Coroutines**: Handle asynchronous operations

### Clean Architecture Principles

The app implements Clean Architecture principles:

1. **Dependency Inversion**: High-level modules don't depend on low-level modules
2. **Single Responsibility**: Each class has one reason to change
3. **Interface Segregation**: Clients depend only on interfaces they use
4. **Open/Closed Principle**: Open for extension, closed for modification

### Key Design Patterns

**Repository Pattern** Centralizes data access logic and provides a unified interface for data operations: - **UserRepository**: Handles user authentication and profile management - **RouteRepository**: Manages trip creation, retrieval, and modification - **PlaceRepository**: Handles destination data and search operations - **FeedbackRepository**: Manages user reviews and ratings

**Dependency Injection (Manual)** Uses a custom AppContainer for dependency management:

```
class AppContainer(private val context: Context) {
    // Provides all necessary dependencies
    val userRepository: UserRepository by lazy { UserRepositoryImpl(userApiService, userMapp
    val routeRepository: RouteRepository by lazy { RouteRepositoryImpl(routeApiService, rout
    // ... other dependencies
}
```

**Observer Pattern** Implements LiveData for reactive UI updates:

```
viewModel.routeList.observe(viewLifecycleOwner) { routes ->
    adapter.updateRoutes(routes)
}
```

---

## Project Structure

The project follows a clean, organized structure based on architectural layers:

```
app/src/main/java/com/zeynekurtulus/wayfare/
+-- data/                                     # Data Layer
|   +-- api/                                 # API Communication
|   |   +-- dto/                            # Data Transfer Objects
|   |   |   +-- auth/                       # Authentication DTOs
|   |   |   +-- route/                      # Route/Trip DTOs
|   |   |   +-- place/                      # Place/Destination DTOs
|   |   |   +-- feedback/                   # Review/Rating DTOs
|   |   |   +-- user/                       # User Management DTOs
|   |   +-- services/                       # Retrofit API Services
|   |   |   +-- UserApiService.kt           # User operations
|   |   |   +-- RouteApiService.kt          # Trip operations
|   |   |   +-- PlaceApiService.kt          # Place operations
|   |   |   +-- FeedbackApiService.kt       # Feedback operations
|   |   |   +-- LocationApiService.kt       # Geographic data
|   |   |   +-- MustVisitApiService.kt      # Must-visit places
|   |   +-- NetworkConfig.kt                # Network configuration
|   +-- mappers/                            # Data Mapping Layer
|   |   +-- UserMapper.kt                  # User data transformations
|   |   +-- RouteMapper.kt                 # Route data transformations
|   |   +-- PlaceMapper.kt                 # Place data transformations
|   |   +-- FeedbackMapper.kt              # Feedback data transformations
|   +-- repository/                        # Repository Implementations
|       +-- UserRepositoryImpl.kt          # User data operations
|       +-- RouteRepositoryImpl.kt         # Route data operations
|       +-- PlaceRepositoryImpl.kt         # Place data operations
|       +-- FeedbackRepositoryImpl.kt      # Feedback data operations
+-- domain/                                # Domain Layer
|   +-- model/                            # Core Business Models
|   |   +-- User.kt                       # User domain model
|   |   +-- Route.kt                      # Trip/Route domain model
|   |   +-- Place.kt                      # Place/Destination domain model
|   |   +-- Feedback.kt                   # Review/Rating domain model
|   |   +-- City.kt                       # City and trip creation models
|   |   +-- MustVisitPlaceSearch.kt       # Must-visit place search model
|   +-- repository/                       # Repository Interfaces
|       +-- UserRepository.kt             # User operations contract
|       +-- RouteRepository.kt            # Route operations contract
|       +-- PlaceRepository.kt            # Place operations contract
|       +-- FeedbackRepository.kt         # Feedback operations contract
+-- presentation/                         # Presentation Layer
|   +-- activities/                       # Android Activities
|   |   +-- MainActivity.kt               # Main app container
|   |   +-- SplashActivity.kt            # App launch screen
|   |   +-- LoginActivity.kt             # User authentication
|   |   +-- SignUpActivity.kt            # User registration
|   |   +-- OtpVerificationActivity.kt   # Email verification
|   +-- fragments/                       # UI Fragments
|   |   +-- HomeFragment.kt              # Home dashboard
|   |   +-- SearchFragment.kt            # Search container with tabs
|   |   +-- SearchRoutesFragment.kt      # Route search functionality
|   |   +-- SearchPlacesFragment.kt      # Place search functionality
|   |   +-- TripMakerFragment.kt         # Trip creation wizard
|   |   +-- CalendarFragment.kt          # Calendar view of trips
|   |   +-- ProfileFragment.kt           # User profile management
|   |   +-- MyTripsFragment.kt           # User's trip list
|   |   +-- AllDestinationsFragment.kt   # Top destinations list
|   |   +-- TripDetailsFragment.kt       # Detailed trip view
|   |   +-- DestinationDetailsFragment.kt # Detailed destination view
|   |   +-- PlaceDetailsFragment.kt      # Detailed place information
|   |   +-- GiveFeedbackFragment.kt      # Route feedback submission
|   |   +-- GivePlaceFeedbackFragment.kt # Place feedback submission
|   |   +-- ViewFeedbackFragment.kt      # Route feedback display
|   |   +-- ViewPlaceFeedbackFragment.kt # Place feedback display
|   +-- adapters/                         # RecyclerView Adapters
|   |   +-- MyTripsAdapter.kt             # Trip list display
|   |   +-- AllDestinationsAdapter.kt     # Destination cards
|   |   +-- FeedbackAdapter.kt            # Feedback list display
|   |   +-- CalendarTripsAdapter.kt       # Calendar trip display
|   |   +-- PlaceSearchAdapter.kt         # Place search results
|   |   +-- CitySuggestionsAdapter.kt     # City autocomplete
|   |   +-- MustVisitPlacesAdapter.kt     # Must-visit places
|   |   +-- SelectedPlacesAdapter.kt      # Selected places in trip maker
|   +-- navigation/                       # Navigation Management
|   |   +-- BottomNavigationHandler.kt    # Bottom navigation controller
|   +-- viewmodels/                       # ViewModels
|   |   +-- UserViewModel.kt              # User state management
|   |   +-- RouteListViewModel.kt         # Route list state
|   |   +-- PlaceViewModel.kt             # Place data state
|   |   +-- FeedbackViewModel.kt          # Feedback state
|   |   +-- TripMakerViewModel.kt         # Trip creation state
|   |   +-- LocationViewModel.kt          # Location data state
|   +-- utils/                            # UI Utilities
|       +-- ViewModelFactory.kt           # ViewModel creation
+-- di/                                   # Dependency Injection
|   +-- AppContainer.kt                   # Manual DI container
+-- utils/                                # General Utilities
|   +-- Constants.kt                      # App-wide constants
|   +-- SharedPreferencesManager.kt     # Local storage
|   +-- ApiResult.kt                      # API response wrapper
|   +-- NetworkUtils.kt                   # Network utilities
|   +-- ValidationUtils.kt                # Input validation
+-- WayfareApplication.kt               # Application class
```

### Key Directories Explained

**Data Layer (`data/`)** Handles all data operations including API communication, data transformation, and repository implementations. This layer is responsible for: - Making HTTP requests to the backend API - Converting between API DTOs and domain models - Implementing data access patterns - Managing network configuration and error handling

**Domain Layer (`domain/`)** Contains the core business logic and models. This layer defines: - Business entities and their relationships - Repository contracts (interfaces) - Domain-specific rules and validations - Core application functionality independent of frameworks

**Presentation Layer (`presentation/`)** Manages the user interface and user interactions. This layer includes: - UI components (Activities, Fragments) -

ViewModels for state management - Adapters for displaying data - Navigation logic and user interaction handlers

**Dependency Injection (`di/`)** Provides centralized dependency management through a manual DI container that creates and manages object lifecycles.

**Utils (`utils/`)** Contains utility classes and functions used throughout the application for common operations like validation, network handling, and configuration management.

---

## Core Features

### 1. User Authentication & Management

The app provides a comprehensive user management system with secure authentication:

#### Authentication Features

- **User Registration**: Create new accounts with email verification
- **Secure Login**: Email/password authentication with JWT tokens
- **Password Management**: Change password functionality for existing users
- **Email Verification**: OTP-based email verification system
- **Account Management**: Update user preferences and profile information
- **Account Deletion**: Secure account removal with data cleanup

#### Implementation Details

```kotlin
// User Authentication Flow
class UserViewModel {
    fun login(email: String, password: String) {
        viewModelScope.launch {
            val result = userRepository.login(email, password)
            when (result) {
                is ApiResult.Success -> {
                    // Store auth token and redirect to main app
                    sharedPreferencesManager.saveAuthToken(result.data.accessToken)
                    navigateToMainApp()
                }
                is ApiResult.Error -> {
                    // Display error message to user
                    _loginState.value = LoginState.Error(result.message)
                }
            }
        }
    }
}
```

**Security Features**

- JWT token-based authentication
- Secure token storage using SharedPreferences
- Automatic token refresh handling
- Session timeout management
- Input validation and sanitization

### 2. Trip Planning & Creation

The core feature that allows users to create detailed travel itineraries through a guided process:

**Trip Creation Wizard** The app uses a step-by-step approach for trip creation:

1. **Welcome & Setup**: Introduction to the trip creation process
2. **Destination Selection**: Choose travel destination with autocomplete search
3. **Date Selection**: Pick start and end dates using date pickers
4. **Interest Categories**: Select travel interests (museums, food, outdoors, etc.)
5. **Season Selection**: Choose travel season (spring, summer, autumn, winter)
6. **Budget Selection**: Pick budget level (low, medium, high)
7. **Travel Style**: Choose pace (relaxed, moderate, accelerated)
8. **Must-Visit Places**: Optional selection of specific places to include
9. **Privacy Settings**: Choose whether to make the trip public
10. **Generation**: AI-powered itinerary creation
11. **Review & Save**: Final review and trip saving

**Trip Management Features**

- **My Trips**: View all created trips in a organized list
- **Trip Details**: Comprehensive view of trip information including:
  - Day-by-day itinerary
  - Activity details and timing
  - Place information with images
  - Budget breakdown
  - Travel statistics
- **Trip Actions**: Edit, duplicate, delete, share, and export trips
- **Privacy Control**: Toggle trip visibility (public/private)
- **Community Feedback**: Rate and review trips, view others' feedback

#### Advanced Features

- **Smart Suggestions**: AI-powered place recommendations
- **Calendar Integration**: View trips in calendar format
- **Sharing Capabilities**: Share trips via social media and messaging
- **Export Options**: Export trip details for offline use

### 3. Search & Discovery

Comprehensive search functionality for both routes and places:

**Dual Search Interface** The app features a tabbed search interface:

**Routes Search Tab**: - Text-based search for trip titles and descriptions - Advanced filtering by: - Location (city, country) - Category (cultural, adventure, beach, etc.) - Budget level - Travel style - Season - Sort options (popularity, rating, recent, alphabetical)

**Places Search Tab**: - Place name and keyword search - City-based filtering - Category filtering - Rating-based filtering - Budget-appropriate place filtering

#### Search Implementation

```
// Route Search with Multiple Filters
@GET("routes/search")
suspend fun searchRoutes(
    @Query("q") query: String? = null,
    @Query("city") city: String? = null,
    @Query("country") country: String? = null,
    @Query("category") category: String? = null,
    @Query("budget") budget: String? = null,
    @Query("travel_style") travelStyle: String? = null,
    @Query("season") season: String? = null,
    @Query("sort_by") sortBy: String? = null,
    @Query("limit") limit: Int = 20
): Response<SearchRoutesResponse>
```

#### Search Features

- **Real-time Search**: Results update as user types
- **Autocomplete**: Smart suggestions for places and cities
- **Filter Persistence**: Remember user's filter preferences
- **Result Cards**: Rich card display with images and key information
- **Clickable Results**: Navigate to detailed views from search results

### 4. Community Feedback System

Robust feedback system allowing users to share experiences:

**Feedback Types** **Route Feedback**: - Star ratings (1-5 stars) - Written comments and reviews - Visit date tracking - User identification (username display) - Feedback viewing and management

**Place Feedback**: - Similar rating and comment system - Place-specific feedback - Date-based reviews - Community-driven quality assessment

#### Feedback Implementation

```kotlin
// Feedback Submission
class FeedbackViewModel {
    fun submitRouteFeedback(routeId: String, rating: Int, comment: String, visitDate: String
        viewModelScope.launch {
            val request = SubmitRouteFeedbackRequest(
                routeId = routeId,
                rating = rating,
                comment = comment,
                visitedOn = visitDate
            )
            val result = feedbackRepository.submitRouteFeedback(request)
            // Handle result...
        }
    }
}
```

#### Features

- **Anonymous or Named Reviews**: Users can choose visibility
- **Review Management**: Edit and delete own reviews
- **Review Display**: View all community feedback
- **Rating Aggregation**: Average ratings for routes and places
- **Feedback Quality**: Report inappropriate content

### 5. Navigation & User Experience

Advanced navigation system with user-friendly features:

#### Bottom Navigation

- **Home**: Dashboard with quick access to main features
- **Calendar**: Calendar view of planned trips
- **Search**: Dual-tab search for routes and places
- **Trip Maker**: Guided trip creation process
- **Profile**: User account and settings management

#### Navigation Features

- **Back Button Handling**: Smart back navigation throughout the app

- **Fragment Management**: Proper fragment lifecycle management
- **State Preservation**: Maintain user's progress during navigation
- **Deep Linking**: Direct access to specific app sections
- **Tab Switching**: Smooth transitions between main sections

#### User Experience Enhancements

- **Loading States**: Clear loading indicators during operations
- **Error Handling**: User-friendly error messages and recovery options
- **Offline Handling**: Graceful degradation when network is unavailable
- **Responsive Design**: Adaptation to different screen sizes
- **Accessibility**: Support for accessibility features

### 6. Data Persistence & Synchronization

Comprehensive data management system:

#### Local Storage

- **SharedPreferences**: User authentication tokens and preferences
- **Cache Management**: Temporary storage for frequently accessed data
- **Offline Support**: Basic functionality without internet connection

#### Data Synchronization

- **Real-time Updates**: Automatic data refresh when app is active
- **Conflict Resolution**: Handle data conflicts during synchronization
- **Background Sync**: Update data when app is in background
- **Delta Sync**: Only sync changed data to improve performance

#### Data Models

```kotlin
// Core Trip Model
@Parcelize
data class Route(
    val routeId: String,
    val userId: String,
    val title: String,
    val city: String,
    val country: String,
    val startDate: String,
    val endDate: String,
    val budget: String,
    val travelStyle: String,
    val category: String,
    val season: String,
    val stats: RouteStats,
    val mustVisit: List<MustVisitPlace>,
    val days: List<RouteDay>,
    val isPublic: Boolean = false
) : Parcelable
```

---

## API Integration

### Overview

The Wayfare Android app integrates with a comprehensive REST API backend that provides all necessary functionality for travel planning, user management, and community features. The API uses JWT authentication and follows RESTful principles.

### Base Configuration

```
object Constants {
    const val BASE_URL = "http://10.0.2.2:8000/"  // Android Emulator
    const val API_TIMEOUT = 30L // seconds
}
```

### Authentication

All authenticated endpoints require a Bearer token in the Authorization header:

```
@Header("Authorization") authorization: String  // Format: "Bearer {token}"
```

### Supported API Endpoints

#### User Management Endpoints

#### User Registration

```
@POST("user/register")
suspend fun register(@Body request: RegisterRequest): Response<RegisterResponse>
```

- Creates new user accounts with email verification
- Validates email uniqueness and password strength
- Returns user ID and access token upon successful registration

#### User Authentication

```
@POST("user/login")
suspend fun login(@Body request: LoginRequest): Response<LoginResponse>
```

- Authenticates users with email and password
- Returns JWT access token for subsequent API calls
- Handles invalid credentials with appropriate error messages

#### Get Current User

```
@GET("user/getCurrentUser")
suspend fun getCurrentUser(@Header("Authorization") authorization: String): Response<GetCurr
```

- Retrieves current user's profile information
- Includes travel preferences and account settings
- Used for profile display and preference pre-filling

**Update User Information**

```
@POST("user/addInfo")
suspend fun addInfo(@Header("Authorization") authorization: String, @Body request: AddInfoRe
```

- Updates user travel preferences and profile data
- Includes interests, budget preferences, and personal information
- Validates data before updating database

**Change Password**

```
@POST("user/changePassword")
suspend fun changePassword(@Header("Authorization") authorization: String, @Body request: Ch
```

- Allows users to change their account password
- Requires current password verification
- Enforces password strength requirements

**Email Verification**

```
@POST("user/sendVerification")
suspend fun sendVerification(@Body request: SendVerificationRequest): Response<VerificationR

@POST("user/sendVerification/verifyCode")
suspend fun verifyCode(@Body request: VerifyCodeRequest): Response<VerificationResponse>
```

- Sends OTP codes to user's email address
- Verifies codes for account activation
- Handles code expiration and retry logic

**Route Management Endpoints**   **Create Route**

```
@POST("route/create")
suspend fun createRoute(@Header("Authorization") authorization: String, @Body request: Creat
```

- Creates new travel itineraries based on user preferences
- Uses AI to generate day-by-day activities and recommendations
- Returns complete route with places, timing, and details

**Get User Routes**

```
@GET("routes/user")
suspend fun getUserRoutes(@Header("Authorization") authorization: String): Response<UserRout
```

- Retrieves all routes created by the authenticated user
- Returns route summaries with key information

- Supports pagination for large route collections

**Get Route Details**

```
@GET("routes/{route_id}")
suspend fun getRoute(@Header("Authorization") authorization: String, @Path("route_id") route
```

- Fetches detailed information for a specific route
- Includes day-by-day itinerary and place details
- Used for route viewing and editing

**Update Route**

```
@PUT("routes/{route_id}")
suspend fun updateRoute(@Header("Authorization") authorization: String, @Path("route_id") ro
```

- Modifies existing route information
- Allows changes to route details, dates, and preferences
- Validates ownership before allowing updates

**Delete Route**

```
@DELETE("routes/{route_id}")
suspend fun deleteRoute(@Header("Authorization") authorization: String, @Path("route_id") ro
```

- Permanently removes a route from user's collection
- Validates ownership before deletion
- Cannot be undone

**Get Public Routes**

```
@GET("routes/public")
suspend fun getPublicRoutes(@Header("Authorization") authorization: String, @Query("category
```

- Retrieves publicly shared routes from other users
- Supports filtering by category, season, and budget
- Used for inspiration and discovery

**Search Routes**

```
@GET("routes/search")
suspend fun searchRoutes(@Query("q") query: String?, @Query("city") city: String?, @Query("c
```

- Advanced search functionality with multiple filter options
- Supports text search across route titles and descriptions
- Provides sorting options for search results

**Place Management Endpoints** **Get Places by City**

```
@GET("places/city")
suspend fun getPlacesByCity(@Query("city") city: String, @Query("limit") limit: Int = 20): R
```

- Retrieves all available places in a specific city
- Used for destination discovery and trip planning

- Returns place details including ratings and categories

**Search Places**

```
@POST("places/search")
suspend fun searchPlaces(@Body request: SearchPlacesRequest): Response<SearchPlacesResponse>
```

- Comprehensive place search with multiple criteria:
  - City name (required)
  - Category filtering
  - Budget level filtering
  - Rating requirements
  - Name/keyword search
  - Country filtering

```
// Search Request Structure
data class SearchPlacesRequest(
    val city: String,                         // REQUIRED
    val category: String? = null,             // OPTIONAL
    val budget: String? = null,               // "low", "medium", "high"
    val rating: Double? = null,               // Exact rating match
    val name: String? = null,                 // Partial name search
    val country: String? = null,              // Country filter
    val min_rating: Double? = null,           // Minimum rating filter
    val keywords: String? = null,             // Description search
    val limit: Int = 20                       // Max results
)
```

**Autocomplete Places**

```
@POST("places/autocomplete")
suspend fun getAutocompleteSuggestions(@Body request: AutocompleteRequest): Response<Autocom
```

- Provides real-time suggestions as users type
- Used in search bars and input fields
- Improves user experience with smart suggestions

**Location Data Endpoints  Get All Cities**

```
@GET("cities/all")
suspend fun getAllCities(): Response<GetAllCitiesResponse>
```

- Returns list of all available cities in the system
- Used for destination selection and filtering
- Includes city coordinates and country information

**Get Cities by Country**

```
@POST("cities/specific")
suspend fun getCitiesByCountry(@Body request: GetCitiesByCountryRequest): Response<GetCities
```

- Filters cities by specific country
- Used for country-based destination browsing
- Supports multiple country selection

### Get All Countries

```
@GET("countries/all")
suspend fun getAllCountries(): Response<GetAllCountriesResponse>
```

- Provides complete list of available countries
- Used for country selection and filtering
- Includes country codes and region information

### Feedback Management Endpoints   Submit Route Feedback

```
@POST("feedback/route")
suspend fun submitRouteFeedback(@Header("Authorization") token: String, @Body request: Submi
```

- Allows users to rate and review routes
- Includes star rating (1-5) and written comments
- Tracks visit date for authentic reviews

### Get Route Feedback

```
@GET("feedback/route/{route_id}")
suspend fun getRouteFeedback(@Path("route_id") routeId: String): Response<GetRouteFeedbackRe
```

- Retrieves all feedback for a specific route
- Public endpoint (no authentication required)
- Displays community ratings and reviews

### Submit Place Feedback

```
@POST("feedback/place")
suspend fun submitPlaceFeedback(@Header("Authorization") token: String, @Body request: Submi
```

- Community rating system for individual places
- Similar structure to route feedback
- Helps improve place recommendations

### Get Place Feedback

```
@GET("feedback/place/{place_id}")
suspend fun getPlaceFeedback(@Path("place_id") placeId: String): Response<GetPlaceFeedbackRe
```

- Public access to place reviews and ratings
- Used for informed decision making
- Displays aggregated rating information

### API Response Format

All API responses follow a consistent structure:

```kotlin
data class ApiResponse<T>(
    val success: Boolean,           // Operation success status
    val message: String,            // Human-readable message
    val status_code: Int,           // HTTP status code
    val data: T?                    // Response data (if successful)
)
```

**Success Response Example:**

```json
{
    "success": true,
    "message": "Route created successfully",
    "status_code": 201,
    "data": {
        "route_id": "64f8a1b2c3d4e5f6789abc02",
        "title": "Rome Adventure",
        "created_at": "2025-06-01T10:30:00Z"
    }
}
```

**Error Response Example:**

```json
{
    "success": false,
    "message": "Invalid authentication token",
    "status_code": 401,
    "data": null
}
```

### Error Handling

The app implements comprehensive error handling for all API interactions:

```kotlin
sealed class ApiResult<out T> {
    data class Success<T>(val data: T) : ApiResult<T>()
    data class Error(val message: String, val code: Int? = null) : ApiResult<Nothing>()
    object Loading : ApiResult<Nothing>()
}

// Usage in Repository
class RouteRepositoryImpl {
    suspend fun createRoute(request: CreateRouteRequest): ApiResult<Route> {
        return try {
            val response = routeApiService.createRoute("Bearer $token", request)
            if (response.isSuccessful && response.body()?.success == true) {
                ApiResult.Success(routeMapper.mapToDomain(response.body()!!.data))
            } else {
                ApiResult.Error(response.body()?.message ?: "Unknown error")
            }
        } catch (e: Exception) {
            ApiResult.Error(NetworkUtils.getErrorMessage(e))
        }
    }
}
```

### Network Configuration

```kotlin
object NetworkConfig {
    fun provideOkHttpClient(): OkHttpClient {
        return OkHttpClient.Builder()
            .connectTimeout(Constants.API_TIMEOUT, TimeUnit.SECONDS)
            .readTimeout(Constants.API_TIMEOUT, TimeUnit.SECONDS)
            .writeTimeout(Constants.API_TIMEOUT, TimeUnit.SECONDS)
            .addInterceptor(HttpLoggingInterceptor().apply {
                level = HttpLoggingInterceptor.Level.BODY
            })
            .build()
    }

    fun provideRetrofit(okHttpClient: OkHttpClient): Retrofit {
        return Retrofit.Builder()
            .baseUrl(Constants.BASE_URL)
            .client(okHttpClient)
            .addConverterFactory(GsonConverterFactory.create())
            .build()
    }
}
```

---

## Data Models

### Domain Models

The application uses clean, well-defined domain models that represent core business entities:

#### User Model

```kotlin
@Parcelize
data class User(
    val userId: String,
    val username: String,
    val email: String,
    val firstName: String,
    val lastName: String,
    val isVerified: Boolean,
    val travelPreferences: TravelPreferences?,
    val createdAt: String,
    val updatedAt: String
) : Parcelable

@Parcelize
data class TravelPreferences(
    val budget: String,              // "low", "medium", "high"
    val travelStyle: String,         // "relaxed", "moderate", "accelerated"
    val interests: List<String>,     // User's travel interests
    val preferredSeasons: List<String>
) : Parcelable
```

### Route Model

```
@Parcelize
data class Route(
    val routeId: String,
    val userId: String,
    val title: String,
    val city: String,
    val cityId: String?,
    val country: String,
    val countryId: String?,
    val startDate: String,           // Format: "yyyy-MM-dd"
    val endDate: String,
    val budget: String,              // "low", "medium", "high"
    val travelStyle: String,         // "relaxed", "moderate", "accelerated"
    val category: String,            // Trip category
    val season: String,              // "spring", "summer", "autumn", "winter"
    val stats: RouteStats,
    val mustVisit: List<MustVisitPlace>,
    val days: List<RouteDay>,
    val createdAt: String?,
    val updatedAt: String?,
    val isPublic: Boolean = false    // Privacy setting
) : Parcelable

@Parcelize
data class RouteStats(
    val viewsCount: Int,
    val copiesCount: Int,
    val likesCount: Int
) : Parcelable
```

```kotlin
@Parcelize
data class RouteDay(
    val date: String,                // "yyyy-MM-dd"
    val activities: List<Activity>
) : Parcelable

@Parcelize
data class Activity(
    val placeId: String?,
    val placeName: String,
    val time: String,                // "HH:mm"
    val notes: String?,
    val image: String?
) : Parcelable
```

### Place Model

```kotlin
@Parcelize
data class Place(
    val placeId: String,
    val name: String,
    val address: String?,
    val coordinates: Coordinates?,
    val category: String?,           // Place category
    val rating: Double?,             // Overall rating
    val priceLevel: Int?,           // 1-4 price scale
    val openingHours: Map<String, String>?,
    val image: String?,
    val detailUrl: String?,
    val duration: Int?               // Recommended visit duration in minutes
) : Parcelable

@Parcelize
data class TopRatedPlace(
    val placeId: String,
    val name: String,
    val city: String,
    val category: String?,
    val wayfareCategory: String,     // Wayfare-specific categorization
    val price: String?,
    val rating: Double?,
    val wayfareRating: Double?,      // Wayfare community rating
    val totalFeedbackCount: Int,
    val image: String?,
    val detailUrl: String?,
    val openingHours: Map<String, String>?,
    val coordinates: Coordinates?,
    val address: String?,
    val source: String?,
    val country: String?,
    val countryId: String?,
    val cityId: String?,
    val priceLevel: Int?
) : Parcelable

@Parcelize
data class MustVisitPlace(
    val placeId: String?,
    val placeName: String,
    val address: String?,
    val coordinates: Coordinates?,
    val notes: String?,
    val source: String,              // Data source identifier
    val openingHours: Map<String, String>?,
    val image: String?
) : Parcelable

@Parcelize
data class Coordinates(
    val lat: Double,
    val lng: Double
) : Parcelable
```

### City and Location Models

```kotlin
@Parcelize
data class City(
    val cityId: String,
    val name: String,
    val country: String,
    val countryId: String,
    val displayText: String,         // Formatted display name
    val coordinates: CityCoordinates
) : Parcelable

@Parcelize
data class CityCoordinates(
    val lat: Double,
    val lng: Double
) : Parcelable
```

```kotlin
@Parcelize
data class Country(
    val countryId: String,
    val name: String,
    val code: String,                  // ISO country code
    val region: String
) : Parcelable
```

### Feedback Models

```kotlin
@Parcelize
data class RouteFeedback(
    val feedbackId: String,
    val userId: String,
    val username: String,              // Display name for feedback
    val routeId: String,
    val rating: Int,                   // 1-5 star rating
    val comment: String,
    val visitedOn: String,             // "yyyy-MM-dd"
    val createdAt: String,
    val updatedAt: String
) : Parcelable

@Parcelize
data class PlaceFeedback(
    val feedbackId: String,
    val userId: String,
    val username: String,
    val placeId: String,
    val rating: Int,                   // 1-5 star rating
    val comment: String,
    val visitedOn: String,
    val createdAt: String,
    val updatedAt: String
) : Parcelable
```

### Trip Creation Models

```kotlin
@Parcelize
data class TripCreationData(
    var selectedCity: City? = null,
    var startDate: String? = null,
    var endDate: String? = null,
    var category: String? = null,
    var season: String? = null,
    var interests: List<String> = emptyList(),
    var budget: String? = null,
    var travelStyle: String? = null,
    var title: String? = null,
    var isPublic: Boolean = false    // Privacy setting for new trips
) : Parcelable {
    // Non-parcelable field for must-visit places
    var selectedMustVisitPlaces: List<MustVisitPlaceSearch> = emptyList()
}

data class MustVisitPlaceSearch(
    val placeId: String,
    val name: String,
    val category: String?,
    val image: String?,
    val rating: Double?,
    val coordinates: Coordinates?
)
```

**Search Models**

```
data class SearchPlaces(
    val city: String,                    // REQUIRED - supports partial search
    val category: String? = null,        // OPTIONAL - searches 'category' and 'wayfare_cat
    val budget: String? = null,          // OPTIONAL - "low", "medium", "high"
    val rating: Double? = null,          // OPTIONAL - exact rating match
    val name: String? = null,            // OPTIONAL - partial name search
    val country: String? = null,         // OPTIONAL - partial country search
    val minRating: Double? = null,       // OPTIONAL - minimum rating filter
    val keywords: String? = null,        // OPTIONAL - description search
    val limit: Int = 10                  // OPTIONAL - max results
)

data class RouteSearchParams(
    val query: String? = null,           // Text search in titles/descriptions
    val city: String? = null,
    val country: String? = null,
    val category: String? = null,
    val budget: String? = null,
    val travelStyle: String? = null,
    val season: String? = null,
    val sortBy: String? = null,          // "popularity", "rating", "recent", "title"
    val limit: Int = 20
)

@Parcelize
data class AutocompletePlace(
    val placeId: String,
    val name: String,
    val category: String?
) : Parcelable
```

### Data Transfer Objects (DTOs)

DTOs are used for API communication and are mapped to domain models:

#### User DTOs

```
data class RegisterRequest(
    val username: String,
    val email: String,
    val password: String,
    val firstName: String,
    val lastName: String
)

data class RegisterResponse(
    val success: Boolean,
    val message: String,
    val data: RegisterData?
)

data class RegisterData(
    val userId: String,
    val accessToken: String
)

data class LoginRequest(
    val email: String,
    val password: String
)

data class LoginResponse(
    val success: Boolean,
    val message: String,
    val data: LoginData?
)

data class LoginData(
    val accessToken: String,
    val user: UserDto
)
```

### Route DTOs

```
data class CreateRouteRequest(
    val city: String,
    val startDate: String,
    val endDate: String,
    val budget: String,
    val category: String,
    val season: String,
    val interests: List<String>,
    val travelStyle: String,
    val mustVisit: List<MustVisitPlaceDto>,
    val title: String?,
    val isPublic: Boolean = false
)

data class CreateRouteResponse(
    val success: Boolean,
    val message: String,
    val data: RouteDto?
)

data class RouteDto(
    @SerializedName("route_id") val routeId: String,
    @SerializedName("user_id") val userId: String,
    val title: String,
    val city: String,
    @SerializedName("city_id") val cityId: String?,
    val country: String,
    @SerializedName("country_id") val countryId: String?,
    @SerializedName("start_date") val startDate: String,
    @SerializedName("end_date") val endDate: String,
    val budget: String,
    @SerializedName("travel_style") val travelStyle: String,
    val category: String,
    val season: String,
    val stats: RouteStatsDto,
    @SerializedName("must_visit") val mustVisit: List<MustVisitPlaceDto>,
    val days: List<RouteDayDto>,
    @SerializedName("created_at") val createdAt: String?,
    @SerializedName("updated_at") val updatedAt: String?,
    @SerializedName("is_public") val isPublic: Boolean?
)
```

### Place DTOs

```
data class SearchPlacesRequest(
    val city: String,
    val category: String? = null,
    val budget: String? = null,
    val rating: Double? = null,
    val name: String? = null,
    val country: String? = null,
    val min_rating: Double? = null,
    val keywords: String? = null,
    val limit: Int = 20
)

data class PlaceDto(
    @SerializedName("place_id") val placeId: String,
    val name: String,
    val address: String?,
    val coordinates: CoordinatesDto?,
    val category: String?,
    val rating: Double?,
    @SerializedName("price_level") val priceLevel: Int?,
    @SerializedName("opening_hours") val openingHours: Map<String, String>?,
    val image: String?,
    @SerializedName("detail_url") val detailUrl: String?,
    val duration: Int?
)
```

**Feedback DTOs**

```
data class SubmitRouteFeedbackRequest(
    @SerializedName("route_id") val routeId: String,
    val rating: Int,
    val comment: String,
    @SerializedName("visited_on") val visitedOn: String
)

data class RouteFeedbackDto(
    @SerializedName("feedback_id") val feedbackId: String,
    @SerializedName("user_id") val userId: String,
    @SerializedName("username") val username: String,
    @SerializedName("route_id") val routeId: String,
    val rating: Int,
    val comment: String,
    @SerializedName("visited_on") val visitedOn: String,
    @SerializedName("created_at") val createdAt: String,
    @SerializedName("updated_at") val updatedAt: String
)
```

### Data Mapping

The app uses mapper classes to convert between DTOs and domain models:

```kotlin
object RouteMapper {
    fun mapToDomain(dto: RouteDto): Route {
        return Route(
            routeId = dto.routeId,
            userId = dto.userId,
            title = dto.title,
            city = dto.city,
            cityId = dto.cityId,
            country = dto.country,
            countryId = dto.countryId,
            startDate = dto.startDate,
            endDate = dto.endDate,
            budget = dto.budget,
            travelStyle = dto.travelStyle,
            category = dto.category,
            season = dto.season,
            stats = mapStatsToDomain(dto.stats),
            mustVisit = dto.mustVisit.map { mapMustVisitToDomain(it) },
            days = dto.days.map { mapDayToDomain(it) },
            createdAt = dto.createdAt,
            updatedAt = dto.updatedAt,
            isPublic = dto.isPublic ?: false
        )
    }

    fun mapToCreateRequest(data: TripCreationData, mustVisitPlaces: List<MustVisitPlaceSearc
        return CreateRouteRequest(
            city = data.selectedCity?.name ?: "",
            startDate = data.startDate ?: "",
            endDate = data.endDate ?: "",
            budget = data.budget ?: "",
            category = data.category ?: "",
            season = data.season ?: "",
            interests = data.interests,
            travelStyle = data.travelStyle ?: "",
            mustVisit = mustVisitPlaces.map { mapMustVisitToDto(it) },
            title = data.title,
            isPublic = data.isPublic
        )
    }
}
```

### Constants and Enumerations

```kotlin
object Constants {
    object TravelStyle {
        const val RELAXED = "relaxed"         // Slow-paced, flexible schedule
        const val MODERATE = "moderate"       // Balanced pace with some flexibility
        const val ACCELERATED = "accelerated" // Fast-paced, packed schedule
    }

    object Budget {
        const val LOW = "low"       // Budget-friendly options
        const val MEDIUM = "medium" // Mid-range spending
        const val HIGH = "high"     // Luxury/premium options
    }

    object Season {
        const val SPRING = "spring"
        const val SUMMER = "summer"
        const val AUTUMN = "autumn"
        const val WINTER = "winter"
    }

    object Interests {
        const val MUSEUMS = "Museums and Art Galleries"
        const val FOOD_DRINKS = "Food & Drinks"
        const val OUTDOORS = "Outdoors"
        const val HIDDEN_GEMS = "Hidden Gems"
        const val FAMILY_FRIENDLY = "Family Friendly"
        const val ARCHITECTURE = "architecture"
        const val NIGHTLIFE = "nightlife"
        const val SHOPPING = "shopping"
        const val HISTORICAL = "historical"
        const val NATURE = "nature"
    }
}
```

### API Result Wrapper

```kotlin
sealed class ApiResult<out T> {
    data class Success<T>(val data: T) : ApiResult<T>()
    data class Error(val message: String, val code: Int? = null) : ApiResult<Nothing>()
    object Loading : ApiResult<Nothing>()
}

// Extension function for easy handling
inline fun <T> ApiResult<T>.onSuccess(action: (T) -> Unit): ApiResult<T> {
    if (this is ApiResult.Success) action(data)
    return this
}

inline fun <T> ApiResult<T>.onError(action: (String, Int?) -> Unit): ApiResult<T> {
    if (this is ApiResult.Error) action(message, code)
    return this
}
```

---

## UI Components

### Activities

**MainActivity** The main container activity that hosts all app fragments and manages bottom navigation:

```
class MainActivity : AppCompatActivity() {
    private lateinit var binding: ActivityMainBinding
    private lateinit var bottomNavigationHandler: BottomNavigationHandler

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)

        setupBottomNavigation()
        handleBackNavigation()

        // Clear fragment references to prevent ViewPager2 crashes
        if (savedInstanceState != null) {
            bottomNavigationHandler.clearFragmentReferences()
        }
    }

    // Public method for switching to Trip Maker from other fragments
    fun switchToTripMaker() {
        bottomNavigationHandler.switchToTab(BottomNavigationHandler.NavigationTab.TRIP_MAKER
    }
}
```

**Key Features:** - Bottom navigation management - Fragment lifecycle handling - Back navigation coordination - ViewPager2 crash prevention - Public trip maker access

**SplashActivity** App entry point with loading and authentication checks:

```
class SplashActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        // Check authentication status
        val isLoggedIn = sharedPreferencesManager.isLoggedIn()

        Handler(Looper.getMainLooper()).postDelayed({
            if (isLoggedIn) {
                startActivity(Intent(this, MainActivity::class.java))
            } else {
                startActivity(Intent(this, LoginActivity::class.java))
            }
            finish()
        }, SPLASH_DELAY)
    }
}
```

#### Authentication Activities

- **LoginActivity**: User sign-in with email/password
- **SignUpActivity**: New user registration
- **OtpVerificationActivity**: Email verification with OTP codes

### Fragments

**Navigation Fragments** **HomeFragment**: Main dashboard with quick access to features - Welcome message and user greeting - Quick action buttons (Plan Trip, Browse Destinations) - Recent trips display - Featured destinations carousel

**SearchFragment**: Tabbed search interface - TabLayout with ViewPager2 for Routes and Places tabs - Coordinated search functionality - State management across tabs - Search result persistence

```
class SearchFragment : Fragment() {
    private var _binding: FragmentSearchBinding? = null
    private val binding get() = _binding!!

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)
        setupViewPager()
    }

    private fun setupViewPager() {
        val adapter = SearchPagerAdapter(this)
        binding.viewPager.adapter = adapter
        binding.viewPager.offscreenPageLimit = 1
        binding.viewPager.isSaveEnabled = false // Prevent ViewPager2 crashes

        TabLayoutMediator(binding.tabLayout, binding.viewPager) { tab, position ->
            tab.text = when (position) {
                0 -> "Routes"
                1 -> "Places"
                else -> ""
            }
        }.attach()
    }

    // Reset search when leaving fragment
    override fun onPause() {
        super.onPause()
        // Reset search results in child fragments
    }
}
```

**Search Sub-Fragments** **SearchRoutesFragment**: Route search with filtering - Text search input with real-time results - Advanced filtering options (location, budget, category, style) - RecyclerView with route cards - Navigation to route details

**SearchPlacesFragment**: Place search and discovery - Place name and city search - Category and rating filters - Place cards with images and ratings - Navigation to place details

**Trip Management Fragments** **TripMakerFragment**: Multi-step trip creation wizard

```
class TripMakerFragment : Fragment() {
    private val viewModel: TripMakerViewModel by viewModels()

    // Unsaved changes detection
    fun hasUnsavedChanges(): Boolean {
        return viewModel.hasUnsavedChanges()
    }

    // Handle navigation warnings
    override fun onPause() {
        super.onPause()
        if (hasUnsavedChanges() && !isNavigatingBack) {
            showUnsavedChangesDialog()
        }
    }
    private fun showUnsavedChangesDialog() {
        val dialogView = LayoutInflater.from(requireContext())
            .inflate(R.layout.dialog_unsaved_changes, null)

        val dialog = AlertDialog.Builder(requireContext())
            .setView(dialogView)
            .setCancelable(false)
            .create()

        // Apply white background and dim overlay
        dialog.window?.setBackgroundDrawable(
            ContextCompat.getDrawable(requireContext(), R.drawable.bg_dialog_white)
        )
        dialog.window?.setDimAmount(0.6f)

        dialog.show()
    }
}
```

**MyTripsFragment**: User's trip collection - Grid/List view of created trips - Trip action menu (edit, duplicate, delete, share) - Empty state handling - Navigation to trip details

**TripDetailsFragment**: Comprehensive trip view - Day-by-day itinerary display - Activity details with timing - Place information and images - Trip actions (share, edit, delete) - Community feedback integration

**Content Fragments** **AllDestinationsFragment**: Top destinations discovery - Grid layout with destination cards - Category filtering matching step_interests - Filter by wayfare_category field - Image loading with Glide - Navigation to destination details

**CalendarFragment**: Calendar view of trips - Month/week calendar display - Trip indicators on dates - Quick trip access from calendar - Date-based trip filtering

**ProfileFragment**: User account management - Profile information display - Travel preferences editing - Account settings - Logout functionality

**Detail Fragments** **DestinationDetailsFragment**: Destination information - Comprehensive destination details - Image gallery - Planning integration (removed heart and map buttons) - Trip planning button with proper navigation

**PlaceDetailsFragment**: Individual place details - Place information and ratings - Opening hours display - Image and location details - Community feedback access

**Feedback Fragments** **GiveFeedbackFragment**: Route feedback submission

```
class GiveFeedbackFragment : Fragment() {
    private val feedbackViewModel: FeedbackViewModel by viewModels()

    fun hasUnsavedChanges(): Boolean {
        return binding.ratingBar.rating > 0 ||
               binding.commentEditText.text.toString().isNotBlank() ||
               binding.visitDateEditText.text.toString().isNotBlank()
    }

    private fun submitFeedback() {
        val rating = binding.ratingBar.rating.toInt()
        val comment = binding.commentEditText.text.toString()
        val visitDate = binding.visitDateEditText.text.toString()

        feedbackViewModel.submitRouteFeedback(routeId, rating, comment, visitDate)
    }
}
```

**GivePlaceFeedbackFragment**: Place feedback submission - Star rating input - Date picker for visit date - Comment text area - Submission handling with validation

**ViewFeedbackFragment**: Route feedback display - RecyclerView of community reviews - Rating aggregation - User feedback display - Empty state for no reviews

**ViewPlaceFeedbackFragment**: Place feedback display - Similar to route feedback - Place-specific review display - Community rating information

### RecyclerView Adapters

**MyTripsAdapter** Displays user's trips with action menus:

```
class MyTripsAdapter(
    private val onTripClick: (Route) -> Unit,
    private val onMenuClick: (Route, View) -> Unit  // Anchor view for PopupMenu positionin
) : RecyclerView.Adapter<MyTripsAdapter.TripViewHolder>() {

    class TripViewHolder(private val binding: ItemTripCardBinding) : RecyclerView.ViewHolde
        fun bind(route: Route, onTripClick: (Route) -> Unit, onMenuClick: (Route, View) ->
            binding.tripTitle.text = route.title
            binding.tripDestination.text = "${route.city}, ${route.country}"
            binding.tripDates.text = "${route.startDate} - ${route.endDate}"

            // Load trip image
            Glide.with(binding.root.context)
                .load(route.image)
                .placeholder(R.drawable.placeholder_destination)
                .into(binding.tripImage)

            // Click listeners
            binding.root.setOnClickListener { onTripClick(route) }
            binding.menuButton.setOnClickListener { onMenuClick(route, it) }
        }
    }
}
```

**AllDestinationsAdapter** Displays destination cards with filtering:

```
class AllDestinationsAdapter(
    private val onDestinationClick: (TopRatedPlace) -> Unit
) : RecyclerView.Adapter<AllDestinationsAdapter.DestinationViewHolder>() {

    fun updateDestinations(newDestinations: List<TopRatedPlace>) {
        destinations = newDestinations
        notifyDataSetChanged()
    }

    fun filterDestinations(category: String?) {
        val filtered = if (category.isNullOrEmpty() || category == "All") {
            originalDestinations
        } else {
            originalDestinations.filter { destination ->
                destination.wayfareCategory.equals(category, ignoreCase = true)
            }
        }
        updateDestinations(filtered)
    }
}
```

**FeedbackAdapter** Displays community reviews and ratings:

```
class FeedbackAdapter : RecyclerView.Adapter<FeedbackAdapter.FeedbackViewHolder>() {

    class FeedbackViewHolder(private val binding: ItemFeedbackBinding) : RecyclerView.ViewHo
        fun bind(feedback: RouteFeedback) {
            binding.userName.text = feedback.username  // Display actual username
            binding.ratingBar.rating = feedback.rating.toFloat()
            binding.comment.text = feedback.comment
            binding.visitDate.text = "Visited on ${feedback.visitedOn}"
            binding.reviewDate.text = formatDate(feedback.createdAt)
        }
    }
}
```

### Custom Dialogs and UI Components

#### Unsaved Changes Dialog

```xml
<!-- dialog_unsaved_changes.xml -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    android:padding="24dp">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Unsaved Changes"
        android:textSize="20sp"
        android:textStyle="bold" />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="16dp"
        android:text="You have unsaved changes. Are you sure you want to leave?" />

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="24dp"
        android:orientation="horizontal">

        <Button
            android:id="@+id/cancelButton"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:text="Cancel" />

        <Button
            android:id="@+id/discardButton"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:layout_marginStart="16dp"
            android:text="Discard" />
    </LinearLayout>
</LinearLayout>
```

**Delete Confirmation Dialog**

```xml
<!-- dialog_delete_route.xml -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    android:padding="24dp">

    <ImageView
        android:layout_width="72dp"
        android:layout_height="72dp"
        android:layout_gravity="center"
        android:src="@drawable/ic_delete_large"
        android:tint="@color/red_500" />

    <TextView
        android:id="@+id/dialogTitle"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:layout_marginTop="16dp"
        android:text="Delete Trip"
        android:textSize="20sp"
        android:textStyle="bold" />

    <TextView
        android:id="@+id/tripTitle"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:layout_marginTop="8dp"
        android:textSize="16sp"
        android:textStyle="bold" />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:layout_marginTop="16dp"
        android:gravity="center"
        android:text="This action cannot be undone. Are you sure you want to delete this tri

    <!-- Action buttons -->
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="24dp"
        android:orientation="horizontal">

        <Button
            android:id="@+id/cancelButton"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:text="Cancel" />

        <Button
            android:id="@+id/deleteButton"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:layout_marginStart="16dp"
            android:backgroundTint="@color/red_500"
            android:text="Delete"
            android:textColor="@android:color/white" />
    </LinearLayout>
</LinearLayout>
```

### ViewModels

**TripMakerViewModel** Manages trip creation state and validation:

```
class TripMakerViewModel(
    private val routeRepository: RouteRepository,
    private val locationRepository: LocationRepository
) : ViewModel() {

    private val _tripCreationData = MutableLiveData(TripCreationData())
    val tripCreationData: LiveData<TripCreationData> = _tripCreationData

    private val _currentStep = MutableLiveData(0)
    val currentStep: LiveData<Int> = _currentStep

    fun hasUnsavedChanges(): Boolean {
        val data = _tripCreationData.value ?: return false
        return data.selectedCity != null ||
               data.startDate != null ||
               data.endDate != null ||
               data.category != null ||
               data.interests.isNotEmpty() ||
               data.budget != null ||
            data.selectedMustVisitPlaces.isNotEmpty()
    }

    fun resetTripMakerFromNavigation() {
        _tripCreationData.value = TripCreationData()
        _currentStep.value = 0
    }
}
```

**FeedbackViewModel** Manages feedback submission and retrieval:

```
class FeedbackViewModel(
    private val feedbackRepository: FeedbackRepository
) : ViewModel() {

    private val _submitState = MutableLiveData<SubmitFeedbackState>(SubmitFeedbackState.Idle
    val submitState: LiveData<SubmitFeedbackState> = _submitState

    fun submitRouteFeedback(routeId: String, rating: Int, comment: String, visitDate: String
        viewModelScope.launch {
            _submitState.value = SubmitFeedbackState.Loading

            val result = feedbackRepository.submitRouteFeedback(
                SubmitRouteFeedbackRequest(routeId, rating, comment, visitDate)
            )

            _submitState.value = when (result) {
                is ApiResult.Success -> SubmitFeedbackState.Success
                is ApiResult.Error -> SubmitFeedbackState.Error(result.message)
            }
        }
    }
}

sealed class SubmitFeedbackState {
    object Idle : SubmitFeedbackState()
    object Loading : SubmitFeedbackState()
    object Success : SubmitFeedbackState()
    data class Error(val message: String) : SubmitFeedbackState()
}
```

## Navigation System

### Bottom Navigation Architecture

The app uses a centralized navigation system managed by `BottomNavigationHandler` to ensure consistent navigation behavior and prevent common fragment management issues.

### BottomNavigationHandler

```kotlin
class BottomNavigationHandler(
    private val activity: MainActivity,
    private val fragmentManager: FragmentManager,
    private val containerId: Int,
    private val bottomNavigationView: BottomNavigationView
) {

    enum class NavigationTab {
        HOME, CALENDAR, SEARCH, TRIP_MAKER, PROFILE
    }

    // Fragment instances - managed to prevent ViewPager2 crashes
    private var homeFragment: HomeFragment? = null
    private var calendarFragment: CalendarFragment? = null
    private var searchFragment: SearchFragment? = null
    private var tripMakerFragment: TripMakerFragment? = null
    private var profileFragment: ProfileFragment? = null

    fun switchToTab(tab: NavigationTab) {
        // Check for unsaved changes before switching
        val currentFragment = fragmentManager.findFragmentById(containerId)
        if (currentFragment != null && hasUnsavedChanges(currentFragment)) {
            showUnsavedChangesDialog(tab)
            return
        }

        // Special handling for SearchFragment to prevent ViewPager2 crashes
        if (tab == NavigationTab.SEARCH) {
            searchFragment = null  // Always create fresh instance
        }

        val fragment = getOrCreateFragment(tab)

        fragmentManager.beginTransaction()
            .replace(containerId, fragment)
            .commit()
        updateBottomNavigationSelection(tab)
    }

    private fun getOrCreateFragment(tab: NavigationTab): Fragment {
        return when (tab) {
            NavigationTab.HOME -> {
                if (homeFragment == null || homeFragment?.isDetached == true) {
                    homeFragment = HomeFragment()
                }
                homeFragment!!
            }
            NavigationTab.CALENDAR -> {
                if (calendarFragment == null || calendarFragment?.isDetached == true) {
                    calendarFragment = CalendarFragment()
                }
                calendarFragment!!
            }
            NavigationTab.SEARCH -> {
                // Always create new SearchFragment to avoid ViewPager2 issues
                searchFragment = SearchFragment()
                searchFragment!!
            }
            NavigationTab.TRIP_MAKER -> {
                if (tripMakerFragment == null || tripMakerFragment?.isDetached == true) {
                    tripMakerFragment = TripMakerFragment()
                }
                tripMakerFragment!!
            }
            NavigationTab.PROFILE -> {
                if (profileFragment == null || profileFragment?.isDetached == true) {
                    profileFragment = ProfileFragment()
                }
                profileFragment!!
            }
        }
    }

    // Use reflection to check for unsaved changes
    private fun hasUnsavedChanges(fragment: Fragment): Boolean {
        return try {
            val method = fragment.javaClass.getMethod("hasUnsavedChanges")
            method.invoke(fragment) as? Boolean ?: false
        } catch (e: Exception) {
            false
        }
    }
```

```
    fun handleBackPress(): Boolean {
        val currentFragment = fragmentManager.findFragmentById(containerId)

        // Handle fragment back stack first
        if (fragmentManager.backStackEntryCount > 0) {
            fragmentManager.popBackStack()
            return true
        }

        // Handle tab switching for non-home tabs
        if (currentFragment !is HomeFragment) {
            switchToTab(NavigationTab.HOME)
            return true
        }

        return false  // Let activity handle (exit app)
    }

    // Clear fragment references to prevent ViewPager2 crashes on recreation
    fun clearFragmentReferences() {
        homeFragment = null
        calendarFragment = null
        searchFragment = null
        tripMakerFragment = null
        profileFragment = null
    }
}
```

**Key Navigation Features** **Unsaved Changes Detection**: The navigation system automatically detects unsaved changes in fragments like TripMaker and feedback forms, showing appropriate warning dialogs.

**ViewPager2 Crash Prevention**: Special handling for SearchFragment to prevent common ViewPager2 state restoration crashes by always creating fresh instances.

**Fragment Lifecycle Management**: Proper fragment instance management to prevent memory leaks and ensure smooth navigation.

**Back Navigation**: Smart back button handling that prioritizes fragment back stack over tab switching.

### Fragment Navigation Patterns

**Detail Navigation** Navigation to detail screens uses the activity's fragment manager to ensure proper back stack management:

```kotlin
// Correct navigation pattern
private fun navigateToTripDetails(route: Route) {
    val fragment = TripDetailsFragment().apply {
        arguments = Bundle().apply {
            putParcelable("route", route)
        }
    }

    requireActivity().supportFragmentManager.beginTransaction()
        .replace(R.id.fragmentContainer, fragment)
        .addToBackStack(null)
        .commit()
}
```

**Tab Switching with State Preservation** When navigating between bottom navigation tabs, the system preserves fragment state and handles cleanup:

```kotlin
// Example: Navigate to Trip Maker from empty trips state
private fun navigateToTripMaker() {
    (activity as? MainActivity)?.switchToTripMaker()
}
```

### Search Navigation Architecture

The search system uses a tabbed interface with ViewPager2, implementing special crash prevention measures:

```kotlin
class SearchFragment : Fragment() {

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)
        setupViewPager()
    }

    private fun setupViewPager() {
        val adapter = SearchPagerAdapter(this)
        binding.viewPager.adapter = adapter
        binding.viewPager.offscreenPageLimit = 1
        binding.viewPager.isSaveEnabled = false  // Prevent state saving

        TabLayoutMediator(binding.tabLayout, binding.viewPager) { tab, position ->
            tab.text = when (position) {
                0 -> "Routes"
                1 -> "Places"
                else -> ""
            }
        }.attach()
    }

    // Prevent ViewPager2 crashes by not saving instance state
    override fun onSaveInstanceState(outState: Bundle) {
        // Intentionally don't call super to prevent state saving
    }

    override fun onDestroyView() {
        super.onDestroyView()
        binding.viewPager.adapter = null  // Clear adapter reference
        _binding = null
    }

    // Reset search results when leaving fragment
    override fun onPause() {
        super.onPause()
        resetSearchResults()
    }

    private fun resetSearchResults() {
        // Reset search in child fragments
        val adapter = binding.viewPager.adapter as? SearchPagerAdapter
        adapter?.resetAllSearches()
    }
}
```

---

## Key Implementations

### 1. ViewPager2 Crash Prevention

One of the major challenges solved in this project was preventing ViewPager2-related crashes when switching between tabs. The solution involves multiple layers of protection:

#### Problem

```
FATAL EXCEPTION: Fragment no longer exists for key f#0
at androidx.fragment.app.FragmentManager.getFragment(FragmentManager.java:1281)
at androidx.viewpager2.adapter.FragmentStateAdapter.restoreState(FragmentStateAdapter.java:5
```

#### Solution Implementation 1. Always Create Fresh SearchFragment Instances

```
// In BottomNavigationHandler
NavigationTab.SEARCH -> {
    // Always create new SearchFragment to avoid ViewPager2 issues
    searchFragment = SearchFragment()
    searchFragment!!
}
```

**2. Disable ViewPager2 State Saving**

```
// In SearchFragment
private fun setupViewPager() {
    binding.viewPager.isSaveEnabled = false
    binding.viewPager.offscreenPageLimit = 1
}

override fun onSaveInstanceState(outState: Bundle) {
    // Don't save instance state to prevent restoration issues
}
```

**3. Clear Fragment References on Recreation**

```
// In MainActivity
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)

    if (savedInstanceState != null) {
        bottomNavigationHandler.clearFragmentReferences()
    }
}
```

**4. Proper Fragment Lifecycle Management**

```
// Check for detached fragments before reuse
if (homeFragment == null || homeFragment?.isDetached == true) {
    homeFragment = HomeFragment()
}
```

### 2. Unsaved Changes Warning System

The app implements a comprehensive system to warn users about unsaved changes:

#### Implementation

**1. Fragment Interface**

```
// Fragments implement this method
fun hasUnsavedChanges(): Boolean {
    return viewModel.hasUnsavedChanges()
}
```

**2. Detection in Navigation Handler**

```kotlin
private fun hasUnsavedChanges(fragment: Fragment): Boolean {
    return try {
        val method = fragment.javaClass.getMethod("hasUnsavedChanges")
        method.invoke(fragment) as? Boolean ?: false
    } catch (e: Exception) {
        false
    }
}
```

### 3. Warning Dialog Display

```kotlin
private fun showUnsavedChangesDialog(targetTab: NavigationTab) {
    val dialogView = LayoutInflater.from(activity)
        .inflate(R.layout.dialog_unsaved_changes, null)

    val dialog = AlertDialog.Builder(activity)
        .setView(dialogView)
        .setCancelable(false)
        .create()

    // Apply styling
    dialog.window?.setBackgroundDrawable(
        ContextCompat.getDrawable(activity, R.drawable.bg_dialog_white)
    )
    dialog.window?.setDimAmount(0.6f)

    // Handle user choices
    dialogView.findViewById<Button>(R.id.cancelButton).setOnClickListener {
        dialog.dismiss()
    }

    dialogView.findViewById<Button>(R.id.discardButton).setOnClickListener {
        dialog.dismiss()
        resetFragment()
        switchToTab(targetTab)
    }

    dialog.show()
}
```

### 3. Feedback System Implementation

The feedback system allows users to rate and review both routes and places:

**Route Feedback Flow** **1. Feedback Submission**

```
class GiveFeedbackFragment : Fragment() {

    private fun submitFeedback() {
        val rating = binding.ratingBar.rating.toInt()
        val comment = binding.commentEditText.text.toString()
        val visitDate = binding.visitDateEditText.text.toString()

        if (validateInput(rating, comment, visitDate)) {
            feedbackViewModel.submitRouteFeedback(routeId, rating, comment, visitDate)
        }
    }

    private fun observeSubmissionState() {
        feedbackViewModel.submitState.observe(viewLifecycleOwner) { state ->
            when (state) {
                is SubmitFeedbackState.Loading -> showLoading()
                is SubmitFeedbackState.Success -> {
                    showSuccess()
                    navigateBack()
                }
                is SubmitFeedbackState.Error -> showError(state.message)
                SubmitFeedbackState.Idle -> hideLoading()
            }
        }
    }
}
```

**2. Feedback Display**

```
class ViewFeedbackFragment : Fragment() {

    private fun setupObservers() {
        feedbackViewModel.feedbackList.observe(viewLifecycleOwner) { result ->
            when (result) {
                is ApiResult.Success -> {
                    if (result.data.isEmpty()) {
                        showEmptyState()
                    } else {
                        showFeedbackList(result.data)
                    }
                }
                is ApiResult.Error -> {
                    if (result.code == 404) {
                        showEmptyState()  // No feedback yet
                    } else {
                        showErrorState(result.message)
                    }
                }
                is ApiResult.Loading -> showLoadingState()
            }
        }
    }
}
```

**4. Search Implementation**

The search system provides comprehensive filtering for both routes and places:

### Advanced Route Search

```
class SearchRoutesFragment : Fragment() {

    private fun setupSearchWithDebounce() {
        binding.searchEditText.addTextChangedListener(object : TextWatcher {
            private var searchJob: Job? = null

            override fun onTextChanged(s: CharSequence?, start: Int, before: Int, count: Int
                searchJob?.cancel()
                searchJob = viewLifecycleOwner.lifecycleScope.launch {
                    delay(300) // Debounce delay
                    performSearch(s.toString())
                }
            }
        })
    }

    private fun performSearch(query: String) {
        val searchParams = RouteSearchParams(
            query = query.takeIf { it.isNotBlank() },
            city = getSelectedCity(),
            country = getSelectedCountry(),
            category = getSelectedCategory(),
            budget = getSelectedBudget(),
            travelStyle = getSelectedTravelStyle(),
            season = getSelectedSeason(),
            sortBy = getSelectedSortOption(),
            limit = 20
        )

        routeListViewModel.searchRoutes(searchParams)
    }
}
```

### Place Search with Multiple Criteria

```
class SearchPlacesFragment : Fragment() {

    private fun searchPlaces() {
        val query = binding.searchEditText.text.toString()
        val city = binding.cityEditText.text.toString()

        val searchParams = SearchPlaces(
            city = city,
            name = query.takeIf { it.isNotBlank() },
            keywords = query.takeIf { it.isNotBlank() },
            category = getSelectedCategory(),
            budget = getSelectedBudget(),
            minRating = getSelectedMinRating(),
            country = getSelectedCountry(),
            limit = 20
        )

        placeViewModel.searchPlaces(searchParams)
    }
}
```

## 5. Trip Creation Wizard

The trip creation process uses a step-by-step wizard approach:

### State Management

```
class TripMakerViewModel : ViewModel() {

    private val _currentStep = MutableLiveData(0)
    val currentStep: LiveData<Int> = _currentStep

    private val _tripCreationData = MutableLiveData(TripCreationData())
    val tripCreationData: LiveData<TripCreationData> = _tripCreationData

    fun nextStep() {
        val current = _currentStep.value ?: 0
        if (current < TOTAL_STEPS - 1) {
            _currentStep.value = current + 1
        }
    }

    fun previousStep() {
        val current = _currentStep.value ?: 0
        if (current > 0) {
            _currentStep.value = current - 1
        }
    }

    fun updateCity(city: City) {
        val currentData = _tripCreationData.value ?: TripCreationData()
        _tripCreationData.value = currentData.copy(selectedCity = city)
    }

    fun hasUnsavedChanges(): Boolean {
        val data = _tripCreationData.value ?: return false
        return data.selectedCity != null ||
               data.startDate != null ||
               data.endDate != null ||
               data.category != null ||
               data.interests.isNotEmpty() ||
               data.budget != null ||
               data.selectedMustVisitPlaces.isNotEmpty()
    }
}
```

### Step Implementation

```
private fun setupStepFlow() {
    viewModel.currentStep.observe(viewLifecycleOwner) { step ->
        when (step) {
            0 -> showWelcomeStep()
            1 -> showDestinationStep()
            2 -> showDateStep()
            3 -> showCategoryStep()
            4 -> showSeasonStep()
            5 -> showInterestsStep()
            6 -> showBudgetStep()
            7 -> showTravelStyleStep()
            8 -> showMustVisitStep()
            9 -> showLoadingStep()
            10 -> showResultsStep()
        }
    }
}
```

### 6. Image Loading and Caching

The app uses Glide for efficient image loading with proper error handling:

```
// Image loading with fallbacks
Glide.with(this)
```

```kotlin
    .load(imageUrl)
    .placeholder(R.drawable.placeholder_destination)
    .error(R.drawable.error_image)
    .centerCrop()
    .into(imageView)

// Circular image loading for profile pictures
Glide.with(this)
    .load(profileImageUrl)
    .placeholder(R.drawable.default_avatar)
    .circleCrop()
    .into(profileImageView)
```

**7. Data Persistence and Caching**

**SharedPreferences Management**

```kotlin
class SharedPreferencesManager(context: Context) {

    private val preferences = context.getSharedPreferences(PREF_NAME, Context.MODE_PRIVATE)

    fun saveAuthToken(token: String) {
        preferences.edit()
            .putString(PREF_ACCESS_TOKEN, token)
            .putBoolean(PREF_IS_LOGGED_IN, true)
            .apply()
    }

    fun getAuthToken(): String? {
        return preferences.getString(PREF_ACCESS_TOKEN, null)
    }

    fun isLoggedIn(): Boolean {
        return preferences.getBoolean(PREF_IS_LOGGED_IN, false) &&
               getAuthToken() != null
    }

    fun clearUserData() {
        preferences.edit()
            .remove(PREF_ACCESS_TOKEN)
            .remove(PREF_USER_ID)
            .remove(PREF_USERNAME)
            .remove(PREF_EMAIL)
            .putBoolean(PREF_IS_LOGGED_IN, false)
            .apply()
    }
}
```

### Repository Caching Strategy

```
class RouteRepositoryImpl : RouteRepository {

    private var cachedRoutes: List<Route>? = null
    private var cacheTimestamp: Long = 0

    override suspend fun getUserRoutes(forceRefresh: Boolean): ApiResult<List<Route>> {
        // Check cache validity
        if (!forceRefresh && isCacheValid()) {
            cachedRoutes?.let { return ApiResult.Success(it) }
        }

        // Fetch from API
        return try {
            val response = routeApiService.getUserRoutes("Bearer ${getToken()}")
            if (response.isSuccessful && response.body()?.success == true) {
                val routes = response.body()!!.data.map { routeMapper.mapToDomain(it) }

                // Update cache
                cachedRoutes = routes
                cacheTimestamp = System.currentTimeMillis()

                ApiResult.Success(routes)
            } else {
                ApiResult.Error(response.body()?.message ?: "Failed to fetch routes")
            }
        } catch (e: Exception) {
            ApiResult.Error(NetworkUtils.getErrorMessage(e))
        }
    }

    private fun isCacheValid(): Boolean {
        return System.currentTimeMillis() - cacheTimestamp < CACHE_DURATION_ROUTES
    }
}
```

---

## Performance & Security

### Performance Optimizations

#### 1. Memory Management  Fragment Lifecycle Optimization

```kotlin
class BaseFragment : Fragment() {

    private var _binding: ViewBinding? = null
    protected val binding get() = _binding!!

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null  // Prevent memory leaks
    }
}
```

**Image Loading Optimization**

```kotlin
// Efficient image loading with proper sizing
Glide.with(context)
    .load(imageUrl)
    .override(Target.SIZE_ORIGINAL, 300)  // Limit height to reduce memory usage
    .diskCacheStrategy(DiskCacheStrategy.ALL)
    .placeholder(R.drawable.placeholder_shimmer)
    .into(imageView)
```

**RecyclerView Performance**

```kotlin
class OptimizedAdapter : RecyclerView.Adapter<ViewHolder>() {

    init {
        setHasStableIds(true)  // Enable stable IDs for better performance
    }

    override fun getItemId(position: Int): Long {
        return items[position].id.hashCode().toLong()
    }

    override fun onViewRecycled(holder: ViewHolder) {
        super.onViewRecycled(holder)
        // Clear Glide requests to prevent memory leaks
        Glide.with(holder.itemView.context).clear(holder.imageView)
    }
}
```

### 2. Network Optimization   Request Caching

```kotlin
// OkHttp cache configuration
private fun provideCache(context: Context): Cache {
    val cacheSize = 10 * 1024 * 1024  // 10 MB
    return Cache(File(context.cacheDir, "http_cache"), cacheSize.toLong())
}
```

```kotlin
private fun provideOkHttpClient(context: Context): OkHttpClient {
    return OkHttpClient.Builder()
        .cache(provideCache(context))
        .addNetworkInterceptor(CacheInterceptor())
        .connectTimeout(30, TimeUnit.SECONDS)
        .readTimeout(30, TimeUnit.SECONDS)
        .build()
}
```

**Response Compression**

```kotlin
class CompressionInterceptor : Interceptor {
    override fun intercept(chain: Interceptor.Chain): Response {
        val request = chain.request().newBuilder()
            .addHeader("Accept-Encoding", "gzip")
            .build()
        return chain.proceed(request)
    }
}
```

**Pagination Implementation**

```kotlin
class RouteListViewModel : ViewModel() {

    private var currentPage = 0
    private var isLoading = false
    private var hasMoreData = true

    fun loadMoreRoutes() {
        if (isLoading || !hasMoreData) return

        isLoading = true
        viewModelScope.launch {
            val result = routeRepository.getPublicRoutes(
                limit = Constants.DEFAULT_PAGE_SIZE,
                offset = currentPage * Constants.DEFAULT_PAGE_SIZE
            )

            when (result) {
                is ApiResult.Success -> {
                    val newRoutes = result.data
                    if (newRoutes.size < Constants.DEFAULT_PAGE_SIZE) {
                        hasMoreData = false
                    }

                    currentPage++
                    appendRoutes(newRoutes)
                }
                is ApiResult.Error -> handleError(result.message)
            }

            isLoading = false
        }
    }
}
```

**3. UI Performance   Lazy Loading with Shimmer Effects**

```
// Shimmer placeholder during loading
class ShimmerViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
    fun showShimmer() {
        itemView.findViewById<ShimmerFrameLayout>(R.id.shimmerLayout).apply {
            startShimmer()
            visibility = View.VISIBLE
        }
    }

    fun hideShimmer() {
        itemView.findViewById<ShimmerFrameLayout>(R.id.shimmerLayout).apply {
            stopShimmer()
            visibility = View.GONE
        }
    }
}
```

**ViewHolder Pattern Optimization**

```
class TripViewHolder(private val binding: ItemTripCardBinding) : RecyclerView.ViewHolder(bi

    fun bind(route: Route) {
        // Bind text data immediately
        binding.tripTitle.text = route.title
        binding.tripDates.text = "${route.startDate} - ${route.endDate}"

        // Load images asynchronously
        loadTripImage(route.image)
    }

    private fun loadTripImage(imageUrl: String?) {
        binding.tripImage.post {
            Glide.with(binding.root.context)
                .load(imageUrl)
                .placeholder(R.drawable.placeholder_trip)
                .into(binding.tripImage)
        }
    }
}
```

**Security Implementation**

**1. Authentication Security JWT Token Management**

```
class AuthTokenManager(private val sharedPreferencesManager: SharedPreferencesManager) {

    fun saveToken(token: String) {
        // Store token securely
        sharedPreferencesManager.saveAuthToken(token)
    }

    fun getAuthHeader(): String? {
        val token = sharedPreferencesManager.getAuthToken()
        return if (token != null) "Bearer $token" else null
    }

    fun isTokenValid(): Boolean {
        val token = sharedPreferencesManager.getAuthToken()
        if (token.isNullOrEmpty()) return false

        // Check token expiration (basic validation)
        return try {
            val payload = decodeJWTPayload(token)
            val expiration = payload.getLong("exp") * 1000
            System.currentTimeMillis() < expiration
        } catch (e: Exception) {
            false
        }
    }

    private fun decodeJWTPayload(token: String): JSONObject {
        val parts = token.split(".")
        if (parts.size != 3) throw IllegalArgumentException("Invalid JWT token")

        val payload = String(Base64.decode(parts[1], Base64.URL_SAFE))
        return JSONObject(payload)
    }
}
```

**API Security Headers**

```
class AuthInterceptor(private val authTokenManager: AuthTokenManager) : Interceptor {

    override fun intercept(chain: Interceptor.Chain): Response {
        val originalRequest = chain.request()

        // Add authentication header if available
        val authHeader = authTokenManager.getAuthHeader()
        val requestBuilder = originalRequest.newBuilder()

        if (authHeader != null) {
            requestBuilder.addHeader("Authorization", authHeader)
        }

        // Add security headers
        requestBuilder
            .addHeader("X-Requested-With", "XMLHttpRequest")
            .addHeader("Accept", "application/json")

        return chain.proceed(requestBuilder.build())
    }
}
```

**2. Input Validation and Sanitization** **Comprehensive Input Validation**

```
object ValidationUtils {

    fun isValidEmail(email: String): Boolean {
        return email.isNotEmpty() &&
               Patterns.EMAIL_ADDRESS.matcher(email).matches() &&
               email.length <= 320  // RFC 5321 limit
    }

    fun isValidPassword(password: String): Boolean {
        return password.length >= 8 &&
               password.any { it.isDigit() } &&
               password.any { it.isLetter() } &&
               password.any { !it.isLetterOrDigit() }
    }

    fun sanitizeInput(input: String): String {
        return input.trim()
            .replace(Regex("[<>\"'&]"), "")  // Remove potentially dangerous characters
            .take(1000)  // Limit length
    }

    fun validateRouteTitle(title: String): ValidationResult {
        val sanitized = sanitizeInput(title)
        return when {
            sanitized.isBlank() -> ValidationResult(false, "Title cannot be empty")
            sanitized.length < 3 -> ValidationResult(false, "Title must be at least 3 charac
            sanitized.length > 100 -> ValidationResult(false, "Title cannot exceed 100 chara
            else -> ValidationResult(true)
        }
    }
}
```

**SQL Injection Prevention**

```
// Using parameterized queries (if using Room)
@Query("SELECT * FROM routes WHERE user_id = :userId AND title LIKE :searchTerm")
suspend fun searchUserRoutes(userId: String, searchTerm: String): List<RouteEntity>
```

**3. Network Security   Certificate Pinning** (Production Configuration)

```
class NetworkSecurityConfig {

    fun createSecureOkHttpClient(): OkHttpClient {
        val certificatePinner = CertificatePinner.Builder()
            .add("api.wayfare.com", "sha256/AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=")
            .build()

        return OkHttpClient.Builder()
            .certificatePinner(certificatePinner)
            .connectionSpecs(listOf(ConnectionSpec.MODERN_TLS))
            .build()
    }
}
```

**Network Security Configuration** (res/xml/network_security_config.xml)

```
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
    <domain-config cleartextTrafficPermitted="false">
        <domain includeSubdomains="true">api.wayfare.com</domain>
        <pin-set expiration="2026-01-01">
            <pin digest="SHA-256">AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=</pin>
        </pin-set>
    </domain-config>

    <!-- Allow localhost for development -->
    <domain-config cleartextTrafficPermitted="true">
        <domain includeSubdomains="true">localhost</domain>
        <domain includeSubdomains="true">10.0.2.2</domain>
    </domain-config>
</network-security-config>
```

**4. Data Protection   Sensitive Data Handling**

```kotlin
class SecureDataManager(context: Context) {

    private val keyAlias = "wayfare_secret_key"

    init {
        generateSecretKey()
    }

    private fun generateSecretKey() {
        val keyGenerator = KeyGenerator.getInstance(KeyProperties.KEY_ALGORITHM_AES, "Andro
        val keyGenParameterSpec = KeyGenParameterSpec.Builder(
            keyAlias,
            KeyProperties.PURPOSE_ENCRYPT or KeyProperties.PURPOSE_DECRYPT
        )
            .setBlockModes(KeyProperties.BLOCK_MODE_GCM)
            .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_NONE)
            .build()

        keyGenerator.init(keyGenParameterSpec)
        keyGenerator.generateKey()
    }

    fun encryptData(data: String): String {
        val keyStore = KeyStore.getInstance("AndroidKeyStore")
        keyStore.load(null)

        val secretKey = keyStore.getKey(keyAlias, null) as SecretKey
        val cipher = Cipher.getInstance("AES/GCM/NoPadding")
        cipher.init(Cipher.ENCRYPT_MODE, secretKey)

        val encryptedData = cipher.doFinal(data.toByteArray())
        val iv = cipher.iv

        return Base64.encodeToString(iv + encryptedData, Base64.DEFAULT)
    }
}
```

---

## Testing

### Testing Strategy

The application employs a comprehensive testing strategy covering unit tests, integration tests, and UI tests.

#### 1. Unit Testing ViewModel Testing

```kotlin
@ExperimentalCoroutinesApi
class TripMakerViewModelTest {

    @get:Rule
    val instantExecutorRule = InstantTaskExecutorRule()

    @get:Rule
    val mainDispatcherRule = MainDispatcherRule()

    private lateinit var viewModel: TripMakerViewModel
    private lateinit var mockRouteRepository: RouteRepository
    private lateinit var mockLocationRepository: LocationRepository

    @Before
    fun setup() {
        mockRouteRepository = mockk()
        mockLocationRepository = mockk()
        viewModel = TripMakerViewModel(mockRouteRepository, mockLocationRepository)
    }

    @Test
    fun `when city is selected, trip creation data is updated`() {
        // Given
        val testCity = City("1", "Paris", "France", "FR", "Paris, France", CityCoordinates(

        // When
        viewModel.updateCity(testCity)

        // Then
        val tripData = viewModel.tripCreationData.getOrAwaitValue()
        assertEquals(testCity, tripData.selectedCity)
    }

    @Test
    fun `hasUnsavedChanges returns true when data is present`() {
        // Given
        val testCity = City("1", "Paris", "France", "FR", "Paris, France", CityCoordinates(
        viewModel.updateCity(testCity)

        // When
        val hasChanges = viewModel.hasUnsavedChanges()

        // Then
        assertTrue(hasChanges)
    }
}
```

### Repository Testing

```kotlin
@ExperimentalCoroutinesApi
class RouteRepositoryImplTest {

    @get:Rule
    val mainDispatcherRule = MainDispatcherRule()

    private lateinit var repository: RouteRepositoryImpl
    private lateinit var mockApiService: RouteApiService
    private lateinit var mockMapper: RouteMapper

    @Before
    fun setup() {
        mockApiService = mockk()
        mockMapper = mockk()
        repository = RouteRepositoryImpl(mockApiService, mockMapper)
    }

    @Test
    fun `getUserRoutes returns success when API call succeeds`() = runTest {
        // Given
        val mockResponse = mockk<Response<UserRoutesResponse>>()
        val mockRouteDto = mockk<RouteDto>()
        val mockRoute = mockk<Route>()

        every { mockResponse.isSuccessful } returns true
        every { mockResponse.body()?.success } returns true
        every { mockResponse.body()?.data } returns listOf(mockRouteDto)
        every { mockMapper.mapToDomain(mockRouteDto) } returns mockRoute

        coEvery { mockApiService.getUserRoutes(any()) } returns mockResponse

        // When
        val result = repository.getUserRoutes()

        // Then
        assertTrue(result is ApiResult.Success)
        assertEquals(listOf(mockRoute), (result as ApiResult.Success).data)
    }
}
```

### Utility Testing

```kotlin
class ValidationUtilsTest {

    @Test
    fun `isValidEmail returns true for valid email`() {
        assertTrue(ValidationUtils.isValidEmail("test@example.com"))
        assertTrue(ValidationUtils.isValidEmail("user.name+tag@domain.co.uk"))
    }

    @Test
    fun `isValidEmail returns false for invalid email`() {
        assertFalse(ValidationUtils.isValidEmail(""))
        assertFalse(ValidationUtils.isValidEmail("invalid-email"))
        assertFalse(ValidationUtils.isValidEmail("@domain.com"))
        assertFalse(ValidationUtils.isValidEmail("user@"))
    }

    @Test
    fun `isValidPassword returns true for strong password`() {
        assertTrue(ValidationUtils.isValidPassword("StrongPass123!"))
        assertTrue(ValidationUtils.isValidPassword("MyP@ssw0rd"))
    }

    @Test
    fun `isValidPassword returns false for weak password`() {
        assertFalse(ValidationUtils.isValidPassword("weak"))
        assertFalse(ValidationUtils.isValidPassword("password"))
        assertFalse(ValidationUtils.isValidPassword("12345678"))
    }
}
```

**2. Integration Testing   Fragment Testing**

```
@RunWith(AndroidJUnit4::class)
class SearchFragmentTest {

    @get:Rule
    val activityRule = ActivityScenarioRule(MainActivity::class.java)

    @Test
    fun searchFragment_displaysTabsCorrectly() {
        // Navigate to search fragment
        onView(withId(R.id.navigation_search)).perform(click())

        // Verify tabs are displayed
        onView(withText("Routes")).check(matches(isDisplayed()))
        onView(withText("Places")).check(matches(isDisplayed()))
    }

    @Test
    fun searchRoutes_performsSearchWithQuery() {
        // Navigate to search fragment
        onView(withId(R.id.navigation_search)).perform(click())

        // Enter search query
        onView(withId(R.id.searchEditText))
            .perform(typeText("Rome"), closeSoftKeyboard())

        // Verify search results are displayed
        onView(withId(R.id.searchResultsRecyclerView))
            .check(matches(isDisplayed()))
    }
}
```

**API Integration Testing**

```
@RunWith(AndroidJUnit4::class)
class ApiIntegrationTest {

    private lateinit var mockWebServer: MockWebServer
    private lateinit var apiService: RouteApiService

    @Before
    fun setup() {
        mockWebServer = MockWebServer()
        mockWebServer.start()

        val retrofit = Retrofit.Builder()
            .baseUrl(mockWebServer.url("/"))
            .addConverterFactory(GsonConverterFactory.create())
            .build()

        apiService = retrofit.create(RouteApiService::class.java)
    }

    @After
    fun teardown() {
        mockWebServer.shutdown()
    }

    @Test
    fun getUserRoutes_returnsSuccessResponse() = runTest {
        // Given
        val mockResponse = """
            {
                "success": true,
                "message": "Routes retrieved successfully",
                "status_code": 200,
                "data": []
            }
        """.trimIndent()

        mockWebServer.enqueue(
            MockResponse()
                .setResponseCode(200)
                .setBody(mockResponse)
        )

        // When
        val response = apiService.getUserRoutes("Bearer test-token")

        // Then
        assertTrue(response.isSuccessful)
        assertEquals(true, response.body()?.success)
    }
}
```

**3. UI Testing  End-to-End User Flows**

```
@RunWith(AndroidJUnit4::class)
@LargeTest
class TripCreationFlowTest {

    @get:Rule
    val activityRule = ActivityScenarioRule(MainActivity::class.java)

    @Test
    fun completetripCreationFlow() {
        // Navigate to trip maker
        onView(withId(R.id.navigation_trip_maker)).perform(click())

        // Step 1: Select destination
        onView(withId(R.id.citySearchEditText))
            .perform(typeText("Paris"), closeSoftKeyboard())

        onView(withText("Paris, France")).perform(click())
        onView(withId(R.id.nextButton)).perform(click())

        // Step 2: Select dates
        onView(withId(R.id.startDateButton)).perform(click())
        // Select date in date picker
        onView(withText("OK")).perform(click())
        onView(withId(R.id.endDateButton)).perform(click())
        // Select end date
        onView(withText("OK")).perform(click())

        onView(withId(R.id.nextButton)).perform(click())

        // Continue through remaining steps...

        // Verify trip creation success
        onView(withText("Trip created successfully!"))
            .check(matches(isDisplayed()))
    }
}
```

---

## Build Configuration

### Gradle Configuration

#### App-level build.gradle.kts

```
plugins {
    alias(libs.plugins.android.application)
    alias(libs.plugins.kotlin.android)
    id("kotlin-kapt")
    id("kotlin-parcelize")
}

android {
    namespace = "com.zeynekurtulus.wayfare"
    compileSdk = 36

    defaultConfig {
        applicationId = "com.zeynekurtulus.wayfare"
        minSdk = 26
        targetSdk = 36
        versionCode = 1
        versionName = "1.0"

        testInstrumentationRunner = "androidx.test.runner.AndroidJUnitRunner"

        // Build config fields for different environments
        buildConfigField("String", "API_BASE_URL", "\"http://10.0.2.2:8000/\"")
        buildConfigField("boolean", "DEBUG_MODE", "true")
    }
    buildTypes {
        debug {
            isDebuggable = true
            applicationIdSuffix = ".debug"
            versionNameSuffix = "-debug"

            buildConfigField("String", "API_BASE_URL", "\"http://10.0.2.2:8000/\"")
            buildConfigField("boolean", "DEBUG_MODE", "true")
        }

        release {
            isMinifyEnabled = true
            isShrinkResources = true

            proguardFiles(
                getDefaultProguardFile("proguard-android-optimize.txt"),
                "proguard-rules.pro"
            )

            buildConfigField("String", "API_BASE_URL", "\"https://api.wayfare.com/\"")
            buildConfigField("boolean", "DEBUG_MODE", "false")
        }
    }

    compileOptions {
        sourceCompatibility = JavaVersion.VERSION_11
        targetCompatibility = JavaVersion.VERSION_11
    }

    kotlinOptions {
        jvmTarget = "11"
    }

    buildFeatures {
        viewBinding = true
        buildConfig = true
    }

    testOptions {
        unitTests {
            isIncludeAndroidResources = true
        }
    }
}

dependencies {
```

```
// Core Android
implementation("androidx.core:core-ktx:1.13.1")
implementation("androidx.appcompat:appcompat:1.7.0")
implementation("com.google.android.material:material:1.12.0")
implementation("androidx.constraintlayout:constraintlayout:2.1.4")

// Architecture Components
implementation("androidx.lifecycle:lifecycle-viewmodel-ktx:2.8.7")
implementation("androidx.lifecycle:lifecycle-livedata-ktx:2.8.7")
implementation("androidx.lifecycle:lifecycle-runtime-ktx:2.8.7")
implementation("androidx.activity:activity-ktx:1.9.3")
implementation("androidx.fragment:fragment-ktx:1.8.5")

// Navigation
implementation("androidx.navigation:navigation-fragment-ktx:2.8.4")
implementation("androidx.navigation:navigation-ui-ktx:2.8.4")

// Coroutines
implementation("org.jetbrains.kotlinx:kotlinx-coroutines-android:1.7.3")
implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.7.3")

// Networking
implementation("com.squareup.retrofit2:retrofit:2.9.0")
implementation("com.squareup.retrofit2:converter-gson:2.9.0")
implementation("com.squareup.okhttp3:okhttp:4.12.0")
implementation("com.squareup.okhttp3:logging-interceptor:4.12.0")

// JSON parsing
implementation("com.google.code.gson:gson:2.10.1")

// Image loading
implementation("com.github.bumptech.glide:glide:4.16.0")
implementation("com.github.bumptech.glide:okhttp3-integration:4.16.0")
implementation("de.hdodenhof:circleimageview:3.1.0")
kapt("com.github.bumptech.glide:compiler:4.16.0")

// UI Components
implementation("androidx.recyclerview:recyclerview:1.3.2")
implementation("androidx.swiperefreshlayout:swiperefreshlayout:1.1.0")
implementation("androidx.preference:preference-ktx:1.2.1")

// Local Database (Room)
implementation("androidx.room:room-runtime:2.6.1")
implementation("androidx.room:room-ktx:2.6.1")
kapt("androidx.room:room-compiler:2.6.1")
```

```
    // Testing
    testImplementation("junit:junit:4.13.2")
    testImplementation("org.mockito:mockito-core:5.1.1")
    testImplementation("io.mockk:mockk:1.13.5")
    testImplementation("org.jetbrains.kotlinx:kotlinx-coroutines-test:1.7.3")
    testImplementation("androidx.arch.core:core-testing:2.2.0")

    androidTestImplementation("androidx.test.ext:junit:1.2.1")
    androidTestImplementation("androidx.test.espresso:espresso-core:3.6.1")
    androidTestImplementation("androidx.test.espresso:espresso-contrib:3.6.1")
    androidTestImplementation("androidx.test:runner:1.6.2")
    androidTestImplementation("androidx.test:rules:1.6.1")
    androidTestImplementation("com.squareup.okhttp3:mockwebserver:4.12.0")
}
```

## ProGuard Configuration

### proguard-rules.pro

```
# Add project specific ProGuard rules here.

# Keep all model classes
-keep class com.zeynekurtulus.wayfare.domain.model.** { *; }
-keep class com.zeynekurtulus.wayfare.data.api.dto.** { *; }

# Gson rules
-keepattributes Signature
-keepattributes *Annotation*
-dontwarn sun.misc.**
-keep class com.google.gson.** { *; }
-keep class * implements com.google.gson.TypeAdapter
-keep class * implements com.google.gson.TypeAdapterFactory
-keep class * implements com.google.gson.JsonSerializer
-keep class * implements com.google.gson.JsonDeserializer

# Retrofit rules
-keepattributes Signature, InnerClasses, EnclosingMethod
-keepattributes RuntimeVisibleAnnotations, RuntimeVisibleParameterAnnotations
-keepclassmembers,allowshrinking,allowobfuscation interface * {
    @retrofit2.http.* <methods>;
}
-dontwarn org.codehaus.mojo.animal_sniffer.IgnoreJRERequirement
-dontwarn javax.annotation.**
-dontwarn kotlin.Unit
-dontwarn retrofit2.KotlinExtensions
```

```
# OkHttp rules
-dontwarn okhttp3.**
-dontwarn okio.**
-dontwarn javax.annotation.**

# Glide rules
-keep public class * implements com.bumptech.glide.module.GlideModule
-keep class * extends com.bumptech.glide.module.AppGlideModule {
 <init>(...);
}
-keep public enum com.bumptech.glide.load.ImageHeaderParser$** {
  **[] $VALUES;
  public *;
}

# Parcelize rules
-keepclassmembers class * implements android.os.Parcelable {
  public static final android.os.Parcelable$Creator CREATOR;
}

# Keep ViewBinding classes
-keep class com.zeynekurtulus.wayfare.databinding.** { *; }

# Keep enum classes
-keepclassmembers enum * {
    public static **[] values();
    public static ** valueOf(java.lang.String);
}
```

**Version Catalog (libs.versions.toml)**

```toml
[versions]
kotlin = "1.9.22"
android-gradle-plugin = "8.2.2"
core-ktx = "1.13.1"
appcompat = "1.7.0"
material = "1.12.0"
constraintlayout = "2.1.4"
lifecycle = "2.8.7"
navigation = "2.8.4"
coroutines = "1.7.3"
retrofit = "2.9.0"
okhttp = "4.12.0"
gson = "2.10.1"
glide = "4.16.0"
room = "2.6.1"
```

```toml
junit = "4.13.2"
espresso = "3.6.1"
test-ext = "1.2.1"

[libraries]
androidx-core-ktx = { module = "androidx.core:core-ktx", version.ref = "core-ktx" }
androidx-appcompat = { module = "androidx.appcompat:appcompat", version.ref = "appcompat" }
material = { module = "com.google.android.material:material", version.ref = "material" }
androidx-constraintlayout = { module = "androidx.constraintlayout:constraintlayout", versio

# Lifecycle
androidx-lifecycle-viewmodel = { module = "androidx.lifecycle:lifecycle-viewmodel-ktx", vers
androidx-lifecycle-livedata = { module = "androidx.lifecycle:lifecycle-livedata-ktx", versi

# Navigation
androidx-navigation-fragment = { module = "androidx.navigation:navigation-fragment-ktx", ve
androidx-navigation-ui = { module = "androidx.navigation:navigation-ui-ktx", version.ref = '

# Networking
retrofit = { module = "com.squareup.retrofit2:retrofit", version.ref = "retrofit" }
retrofit-gson = { module = "com.squareup.retrofit2:converter-gson", version.ref = "retrofit'
okhttp = { module = "com.squareup.okhttp3:okhttp", version.ref = "okhttp" }
okhttp-logging = { module = "com.squareup.okhttp3:logging-interceptor", version.ref = "okht

# Testing
junit = { module = "junit:junit", version.ref = "junit" }
androidx-test-ext-junit = { module = "androidx.test.ext:junit", version.ref = "test-ext" }
androidx-espresso-core = { module = "androidx.test.espresso:espresso-core", version.ref = "

[plugins]
android-application = { id = "com.android.application", version.ref = "android-gradle-plugi
kotlin-android = { id = "org.jetbrains.kotlin.android", version.ref = "kotlin" }
```

### CI/CD Configuration

#### GitHub Actions Workflow (.github/workflows/android.yml)

```yaml
name: Android CI

on:
  push:
    branches: [ main, develop ]
  pull_request:
    branches: [ main ]

jobs:
```

```yaml
test:
  runs-on: ubuntu-latest

  steps:
  - uses: actions/checkout@v3

  - name: Set up JDK 11
    uses: actions/setup-java@v3
    with:
      java-version: '11'
      distribution: 'temurin'

  - name: Cache Gradle packages
    uses: actions/cache@v3
    with:
      path: |
        ~/.gradle/caches
        ~/.gradle/wrapper
      key: ${{ runner.os }}-gradle-${{ hashFiles('**/*.gradle*', '**/gradle-wrapper.proper
      restore-keys: |
        ${{ runner.os }}-gradle-

  - name: Grant execute permission for gradlew
    run: chmod +x gradlew

  - name: Run unit tests
    run: ./gradlew testDebugUnitTest

  - name: Run instrumented tests
    uses: reactivecircus/android-emulator-runner@v2
    with:
      api-level: 29
      script: ./gradlew connectedDebugAndroidTest

  - name: Generate test report
    uses: dorny/test-reporter@v1
    if: success() || failure()
    with:
      name: 'Test Results'
      path: '**/TEST-*.xml'
      reporter: java-junit

build:
  runs-on: ubuntu-latest
  needs: test
```

```
steps:
- uses: actions/checkout@v3

- name: Set up JDK 11
  uses: actions/setup-java@v3
  with:
    java-version: '11'
    distribution: 'temurin'

- name: Build debug APK
  run: ./gradlew assembleDebug

- name: Upload APK
  uses: actions/upload-artifact@v3
  with:
    name: debug-apk
    path: app/build/outputs/apk/debug/app-debug.apk
```

---

## Conclusion

This comprehensive documentation covers all aspects of the Wayfare Android application as of June 2025. The app represents a modern, well-architected travel planning platform that successfully implements:

- **Clean Architecture**: Clear separation of concerns with MVVM pattern
- **Robust API Integration**: Comprehensive backend communication with proper error handling
- **Advanced UI Features**: Sophisticated navigation, search, and feedback systems
- **Performance Optimization**: Efficient memory management and network operations
- **Security Implementation**: Authentication, input validation, and data protection
- **Comprehensive Testing**: Unit, integration, and UI testing strategies
- **Production-Ready Build**: Optimized release configuration with proper security measures

The application successfully addresses the complex challenges of mobile travel planning while maintaining code quality, user experience, and system reliability.

### Key Achievements

1. **ViewPager2 Crash Resolution**: Implemented multiple layers of protection against fragment state restoration issues
2. **Unsaved Changes System**: Comprehensive warning system for data loss prevention

3. **Advanced Search**: Dual-tab search with extensive filtering capabilities
4. **Community Features**: Complete feedback and rating system for routes and places
5. **Navigation Excellence**: Smooth, intuitive navigation with proper back stack management
6. **Performance Optimization**: Efficient image loading, caching, and memory management
7. **Security Implementation**: JWT authentication, input validation, and secure data handling

### Version Information

- **Document Version**: 2.1
- **Last Updated**: June 2025
- **App Version**: 1.0
- **Target SDK**: 36 (Android 14)
- **Minimum SDK**: 26 (Android 8.0)

This documentation serves as a complete reference for developers, maintainers, and stakeholders working with the Wayfare Android application.