# Wayfare Android Backend Architecture Documentation

**Version:** 1.0
**Date:** December 2024
**Architecture:** MVVM (Model-View-ViewModel)
**Language:** Kotlin
**Platform:** Android

---

## Table of Contents

---

## Project Overview

### Purpose

Wayfare is a comprehensive travel planning Android application that integrates with a FastAPI backend to provide users with personalized travel itineraries, place recommendations, and route management capabilities.

### Key Features

- **User Authentication**: JWT-based authentication with email verification
- **Route Planning**: Create, manage, and share travel routes
- **Place Discovery**: Search and explore travel destinations
- **Feedback System**: Rate and review places and routes
- **Location Services**: GPS integration for location-based features

- **Image Management**: Optimized image loading and caching

**Technology Stack**

- **Language**: Kotlin
- **Architecture**: MVVM with Clean Architecture principles
- **Networking**: Retrofit2 + OkHttp3
- **Image Loading**: Glide
- **Async Operations**: Kotlin Coroutines
- **Dependency Injection**: Manual DI with Container pattern
- **Local Storage**: SharedPreferences + Room (optional)

---

## Architecture Design

**MVVM + Clean Architecture**

```
              Presentation Layer

Activities/                   ViewModels
Fragments               (Business Logic)
(UI Components)           - State Management
                - User Interactions





               Domain Layer

Domain Models             Repository Interfaces
- User                    - UserRepository
- Route                   - RouteRepository
- Place                   - PlaceRepository





                Data Layer

Repository                   API Services
Implementations         - UserApiService
- UserRepoImpl            - RouteApiService
- RouteRepoImpl           - PlaceApiService
- PlaceRepoImpl




 Data Mappers                DTOs
- UserMapper              - UserDto
- RouteMapper             - RouteDto
- PlaceMapper             - PlaceDto
```

**Design Principles**

- **Separation of Concerns**: Each layer has distinct responsibilities
- **Dependency Inversion**: Higher layers depend on abstractions, not concretions
- **Single Responsibility**: Each class has one reason to change
- **Open/Closed**: Open for extension, closed for modification
- **Reactive Programming**: UI reacts to state changes automatically

---

## Project Structure Explained

The Wayfare Android application follows a well-organized directory structure that separates concerns and promotes maintainability. Here's what each directory contains and why it's important:

### data/ - The Data Access Layer

This directory contains all components responsible for fetching, processing, and managing data from external sources (like APIs) and local storage. It serves as the bridge between the raw data sources and the business logic of the application.

**data/api/** - Contains all network-related components: - **dto/** - Data Transfer Objects that represent the exact structure of data sent to and received from the API - **services/** - Retrofit interfaces that define all API endpoints and their HTTP methods - **interceptors/** - Components that automatically modify network requests (like adding authentication tokens) - **NetworkConfig.kt** - Central configuration for all network operations

**data/mappers/** - Contains mapper classes that convert data between different formats: - These classes translate API response data into app-friendly formats - They ensure the app's internal data structure is independent of the API structure - Makes it easy to change API structure without affecting the rest of the app

**data/repository/** - Implementation classes that coordinate data operations: - These classes implement the business rules defined in the domain layer - They decide whether to fetch data from the network or local storage - Handle error cases and data transformation

#### domain/ - The Business Logic Core

This directory defines what the application does, independent of how data is stored or presented. It contains the core business entities and rules.

**domain/model/** - Business entities that represent real-world concepts: - User, Route, Place, etc. - these are the main "things" the app works with - These models contain only business-relevant properties - They're independent of how data is stored or displayed

**domain/repository/** - Contracts that define how data should be accessed: - These are interfaces that specify what data operations are needed - They don't care about implementation details (API vs database) - Allow the business logic to be tested without real data sources

#### presentation/ - The User Interface Logic

Contains all components that handle user interactions and coordinate the display of information.

**presentation/auth/, user/, route/, etc.** - Feature-specific ViewModels: - Each ViewModel manages the logic for a specific screen or feature - They handle user input validation - Coordinate between the UI and the business logic - Manage the state of the user interface (loading, error, success states)

**ViewModelFactory.kt** - Creates ViewModels with their required dependencies

#### utils/ - Helper and Support Classes

Contains utility classes that provide common functionality used throughout the app.

- **Constants.kt** - Central location for all app-wide constant values
- **NetworkUtils.kt** - Helper functions for network operations and connectivity checks
- **ValidationUtils.kt** - Functions to validate user input (email, password, etc.)
- **SharedPreferencesManager.kt** - Manages local storage of user preferences and session data
- **ImageLoader.kt** - Handles loading and caching of images
- **LocationUtils.kt** - Utilities for GPS and location-related operations

### di/ - Dependency Injection

**AppContainer.kt** - Creates and manages all the app's dependencies in one place

### Root Level

**WayfareApplication.kt** - The main application class that initializes everything when the app starts

---

## Data Layer Explanation

The Data Layer is the foundation of our application's backend architecture. It handles all external data sources and provides a clean, consistent interface for the rest of the application to access data. Think of it as the "data warehouse" of your app.

### API Services - The Communication Bridge

API Services are specialized interfaces that define exactly how our Android app communicates with the backend server. Each service focuses on a specific area of functionality, making the code organized and easy to maintain.

**UserApiService - Managing User Accounts** This service handles everything related to user management and authentication. It provides the following capabilities:

- **User Registration**: Creates new user accounts with validation and email verification
- **User Login**: Authenticates users and provides access tokens for secure sessions
- **Profile Management**: Allows users to update their personal information and travel preferences
- **Password Management**: Enables secure password changes with proper validation
- **Account Deletion**: Provides a way for users to permanently delete their accounts
- **Email Verification**: Handles sending and verifying email confirmation codes

The service automatically handles authentication tokens, meaning once a user logs in, all subsequent requests include their credentials securely.

**RouteApiService - Travel Route Management** This service manages all operations related to travel routes and itineraries:

- **Route Creation**: Allows users to create detailed travel plans with multiple destinations
- **Personal Route Management**: Users can view, edit, and delete their own routes
- **Route Sharing**: Enables users to browse and discover routes created by other travelers
- **Advanced Filtering**: Supports searching routes by category (adventure, cultural, etc.), season, and budget
- **Route Details**: Provides comprehensive information about each route including must-visit places, daily schedules, and costs

**PlaceApiService - Location and Destination Data** Handles all operations related to places and points of interest:

- **Place Discovery**: Finds interesting locations and attractions in any city
- **Detailed Information**: Retrieves comprehensive details about specific places
- **Search Functionality**: Enables users to search for places by name, type, or location
- **Autocomplete Suggestions**: Provides real-time suggestions as users type place names

**LocationApiService - Geographic Information** Manages geographic data and location services:

- **City Information**: Provides data about cities worldwide including tourist information
- **Country Data**: Manages country-specific information and travel requirements
- **Regional Organization**: Groups locations by geographic regions for easier browsing

**FeedbackApiService - User Reviews and Ratings** Handles the review and rating system:

- **Place Reviews**: Users can rate and review specific places they've visited
- **Route Feedback**: Enables feedback on entire travel routes
- **Review Management**: Users can update or delete their own reviews
- **Community Insights**: Aggregates reviews to help other travelers make informed decisions

### Data Transfer Objects (DTOs) - The Data Containers

DTOs are special classes that define the exact structure of data exchanged with the server. They serve several important purposes:

- **Data Validation**: Ensure all required information is present before sending requests

- **Type Safety**: Prevent errors by defining exactly what data types are expected
- **API Independence**: Allow the app's internal data structure to be different from the server's
- **Documentation**: Serve as living documentation of what data the API expects and returns

#### Data Mappers - The Translation Layer

Data Mappers are specialized classes that convert data between different formats. They're essential because:

- **Format Translation**: Convert server data format to app-friendly format
- **Data Enrichment**: Add calculated fields or combine data from multiple sources
- **Backward Compatibility**: Handle changes in API structure without breaking the app
- **Data Cleaning**: Filter out unnecessary information and validate data integrity

#### Repository Implementations - The Data Coordinators

Repository implementations are the "smart" classes that make decisions about data operations:

- **Data Source Selection**: Decide whether to fetch fresh data from the server or use cached data
- **Error Handling**: Gracefully handle network failures, authentication errors, and server issues
- **Data Transformation**: Use mappers to convert data into the format needed by the app
- **Business Logic**: Implement data-related business rules (like caching strategies)
- **Session Management**: Handle user authentication and token refresh automatically

### Data Transfer Objects (DTOs)

### User DTOs

```
// Request DTOs
data class RegisterRequest(
    @SerializedName("username") val username: String,
    @SerializedName("password") val password: String,
    @SerializedName("email") val email: String,
    @SerializedName("first_name") val firstName: String,
    @SerializedName("last_name") val lastName: String
)
```

```kotlin
data class LoginRequest(
    @SerializedName("username") val username: String,
    @SerializedName("password") val password: String
)

// Response DTOs
data class LoginResponse(
    @SerializedName("message") val message: String,
    @SerializedName("success") val success: Boolean,
    @SerializedName("access_token") val accessToken: String
)

data class GetCurrentUserResponse(
    @SerializedName("user_id") val userId: String,
    @SerializedName("email") val email: String,
    @SerializedName("username") val username: String,
    @SerializedName("name") val name: String,
    @SerializedName("surname") val surname: String,
    @SerializedName("preferences") val preferences: UserPreferencesDto?,
    @SerializedName("home_city") val homeCity: String?
)
```

**Repository Implementations**

**UserRepositoryImpl**

```kotlin
class UserRepositoryImpl(
    private val userApiService: UserApiService,
    private val sharedPreferencesManager: SharedPreferencesManager
) : UserRepository {

    override suspend fun login(userLogin: UserLogin): ApiResult<AuthTokens> {
        return try {
            val request = UserMapper.mapToLoginRequest(userLogin)
            val response = userApiService.login(request)

            if (response.isSuccessful) {
                response.body()?.let { loginResponse ->
                    val authTokens = UserMapper.mapToAuthTokens(loginResponse)

                    // Store access token and login state
                    sharedPreferencesManager.saveAccessToken(authTokens.accessToken)
                    sharedPreferencesManager.saveUsername(userLogin.username)
                    sharedPreferencesManager.setLoggedIn(true)
                    ApiResult.Success(authTokens)
                } ?: ApiResult.Error("Login failed")
            } else {
                ApiResult.Error("Invalid credentials", response.code())
            }
        } catch (e: Exception) {
            NetworkUtils.handleApiError(e)
        }
    }

    // ... other methods
}
```

### Data Mappers

#### UserMapper

```
object UserMapper {
    fun mapToLoginRequest(userLogin: UserLogin): LoginRequest {
        return LoginRequest(
            username = userLogin.username,
            password = userLogin.password
        )
    }

    fun mapToAuthTokens(loginResponse: LoginResponse): AuthTokens {
        return AuthTokens(accessToken = loginResponse.accessToken)
    }

    fun mapToUser(userResponse: GetCurrentUserResponse): User {
        return User(
            userId = userResponse.userId,
            username = userResponse.username,
            email = userResponse.email,
            firstName = userResponse.name,
            lastName = userResponse.surname,
            preferences = userResponse.preferences?.let { mapToUserPreferences(it) },
            homeCity = userResponse.homeCity
        )
    }

    // ... other mapping methods
}
```

---

## Domain Layer Explanation

The Domain Layer represents the heart of our application - it defines what the app does and the core business rules, independent of how data is stored or how the user interface looks. Think of this layer as the "brain" of the application that contains all the business logic and rules.

### Domain Models - The Business Entities

Domain Models are classes that represent the real-world concepts that our travel planning app works with. They contain only the information that matters for the business logic, without any technical details about databases or user interfaces.

**User Model - The Traveler Profile** The User model represents a person who uses the Wayfare app. It contains:

- **Basic Identity Information**: Username, email, first name, and last name for identification
- **Travel Preferences**: Personal preferences that help customize the travel experience
  - **Interests**: What the user likes (museums, nightlife, nature, food, etc.)
  - **Budget Level**: How much the user typically wants to spend (low, medium, high)
  - **Travel Style**: How they prefer to travel (relaxed, moderate, accelerated pace)
- **Home Location**: The user's home city for better local recommendations

This model helps the app provide personalized travel suggestions and route recommendations based on individual preferences.

**Route Model - The Travel Plan** The Route model represents a complete travel itinerary or plan. It's the core concept of the entire application:

- **Basic Information**: Title, destination city and country, travel dates
- **Travel Characteristics**: Budget level, travel style, category (adventure, cultural, relaxation), and best season
- **Detailed Planning**:
  - **Must-Visit Places**: Key attractions and locations that shouldn't be missed
  - **Daily Schedule**: Day-by-day breakdown of activities and locations
  - **Route Statistics**: Estimated costs, duration, difficulty level
- **Metadata**: Creation and last update timestamps for tracking changes

Routes can be private (only visible to the creator) or public (shared with the community for others to discover and use).

**Place Model - Points of Interest** The Place model represents specific locations, attractions, restaurants, or points of interest:

- **Location Details**: Name, address, GPS coordinates for precise location
- **Practical Information**: Opening hours, contact information, accessibility details
- **Travel Context**: Why this place is interesting, best times to visit, estimated visit duration
- **Community Data**: Ratings, reviews, and photos from other travelers

**Feedback Model - Community Reviews** The Feedback model captures user experiences and reviews:

- **Rating System**: Numerical ratings for different aspects (overall, value, accessibility)
- **Written Reviews**: Detailed text feedback about experiences
- **Contextual Information**: When the visit occurred, travel style of the reviewer
- **Helpful Metrics**: How useful other users found this review

### Repository Interfaces - The Data Contracts

Repository interfaces define what data operations the application needs, without specifying how those operations should be implemented. They serve as contracts between the business logic and the data layer.

**UserRepository Interface - User Data Operations** This interface defines all the operations related to user management:

- **Authentication Operations**: Login, logout, registration, password changes
- **Profile Management**: Getting and updating user information and preferences
- **Session Management**: Checking if a user is logged in, storing and retrieving session data
- **Account Security**: Email verification, account deletion, password reset

The interface ensures that the business logic doesn't need to know whether user data is stored in a local database, retrieved from a web API, or cached in memory.

**RouteRepository Interface - Travel Plan Operations** Defines operations for managing travel routes:

- **Personal Route Management**: Creating, updating, deleting user's own routes
- **Route Discovery**: Finding and browsing public routes created by other users

- **Advanced Search**: Filtering routes by various criteria (location, budget, style, season)
- **Route Details**: Getting comprehensive information about specific routes
- **Social Features**: Sharing routes, copying routes from other users

**PlaceRepository Interface - Location Data Operations** Handles operations related to places and points of interest:

- **Place Discovery**: Finding places in specific cities or regions
- **Search Functionality**: Text-based search for places by name or type
- **Detailed Information**: Getting comprehensive details about specific places
- **Autocomplete**: Providing search suggestions as users type

**LocationRepository Interface - Geographic Data Operations** Manages geographic and administrative location data:

- **Geographic Organization**: Getting cities by country, countries by region
- **Location Search**: Finding cities and countries by name
- **Travel Information**: Getting travel-relevant information about destinations

**FeedbackRepository Interface - Review System Operations** Handles user reviews and ratings:

- **Review Management**: Creating, updating, deleting reviews for places and routes
- **Community Insights**: Retrieving reviews and ratings from other users
- **Review Analytics**: Getting aggregated rating information and statistics

### Why Repository Interfaces Matter

Repository interfaces are crucial because they:

- **Enable Testing**: Business logic can be tested with fake data sources
- **Support Flexibility**: Data sources can be changed without affecting business logic
- **Promote Clean Code**: Keep data access concerns separate from business rules
- **Enable Offline Support**: Can switch between online and offline data sources seamlessly
- **Simplify Development**: Different team members can work on data and business logic independently

---

## Presentation Layer Explanation

The Presentation Layer is responsible for managing the user interface logic and coordinating between what the user sees and the business logic of the application. This layer contains ViewModels that serve as the "traffic controllers" of the app, managing data flow and user interactions.

### ViewModels - The UI Logic Managers

ViewModels are intelligent classes that handle all the logic related to user interface operations. They serve as intermediaries between the UI (Activities/Fragments) and the business logic (Repositories). Think of them as the "smart assistants" that handle user requests and prepare data for display.

**LoginViewModel - Authentication Controller** The LoginViewModel manages all aspects of the user authentication process:

**Core Responsibilities:** - **Input Validation**: Checks that usernames and passwords meet requirements before sending to the server - **Authentication Flow**: Coordinates the login process with the backend server - **State Management**: Tracks whether the user is currently logging in, has succeeded, or encountered an error - **Error Handling**: Provides specific, user-friendly error messages for different failure scenarios - **Session Management**: Determines if a user is already logged in and handles logout processes

**User Experience Features:** - **Real-time Feedback**: Shows loading indicators while authentication is in progress - **Error Display**: Highlights specific input fields that have errors (username vs password) - **Form Management**: Remembers and clears form states appropriately - **Navigation Control**: Determines when to navigate to the main app after successful login

**RegisterViewModel - User Registration Controller** Handles the complex process of creating new user accounts:

**Registration Process Management:** - **Multi-step Validation**: Validates username availability, email format, password strength - **Email Verification**: Coordinates sending and verifying email confirmation codes - **Profile Setup**: Manages the additional information collection (travel preferences, home city) - **Error Prevention**: Prevents duplicate registrations and guides users through fixes

**User Guidance Features:** - **Progressive Validation**: Validates fields as users type, providing immediate feedback - **Helpful Error Messages**: Explains exactly what needs to be fixed and how - **Registration Progress**: Shows users how far they are in the registration process

**RouteListViewModel - Travel Plans Manager** This is one of the core ViewModels that manages the main functionality of displaying and organizing travel routes:

**Personal Route Management:** - **Route Display**: Loads and displays all routes created by the current user - **Route Organization**: Sorts routes by date, popularity, or other criteria - **Quick Actions**: Handles editing, deleting, and sharing routes directly from the list - **Route Status**: Shows whether routes are public, private, or draft status

**Discovery Features:** - **Public Route Browsing**: Loads routes shared by other users from around the world - **Advanced Filtering**: Allows filtering by destination, budget, season, travel style, or category - **Search Functionality**: Enables text-based search through route titles and descriptions - **Recommendation Engine**: Suggests routes based on user preferences and past activity

**Performance Optimization:** - **Lazy Loading**: Loads routes in batches to improve performance with large datasets - **Caching**: Remembers recently viewed routes to reduce network requests - **Background Updates**: Refreshes route data without interrupting user interaction

**RouteDetailViewModel - Individual Route Manager** Manages everything related to viewing and editing specific travel routes:

**Route Display Management:** - **Comprehensive Data Loading**: Loads all details including places, schedules, costs, and reviews - **Interactive Features**: Handles user interactions like expanding sections, viewing maps, or playing media - **Real-time Updates**: Reflects changes made by the user immediately in the interface

**Route Editing Capabilities:** - **Form Management**: Handles the complex forms for editing route details - **Place Management**: Adds, removes, and reorders places within the route - **Schedule Coordination**: Manages day-by-day itinerary planning and time allocation - **Validation Logic**: Ensures route data is complete and logical before saving

**PlaceViewModel - Location Information Manager** Handles all operations related to discovering and managing places of interest:

**Place Discovery:** - **Search Implementation**: Handles real-time search for places by name, type, or location - **Autocomplete**: Provides intelligent suggestions as users type - **Category Browsing**: Organizes places by types (restaurants, attractions, accommodations) - **Geographic Search**: Finds places near specific locations or within certain areas

**Place Details Management:** - **Information Display**: Shows comprehensive details including photos, reviews, and practical information - **User Interaction**: Handles favoriting places, adding to routes, and sharing - **Review Integration**: Displays community reviews and ratings in an organized manner

**UserProfileViewModel - Account Management** Manages all aspects of user account and profile management:

**Profile Information:** - **Data Display**: Shows current user information and travel preferences - **Profile Editing**: Handles updates to personal information and travel preferences - **Privacy Settings**: Manages what information is public vs private - **Account Settings**: Handles password changes, notification preferences, and account deletion

**Travel Analytics:** - **Activity Summary**: Shows user's travel statistics and route creation history - **Preference Learning**: Analyzes user behavior to improve recommendations - **Achievement System**: Tracks and displays travel-related achievements and milestones

### State Management Philosophy

All ViewModels follow a consistent pattern for managing user interface state:

**Loading States**: Show users when operations are in progress to prevent confusion **Success States**: Handle successful operations and update the UI appropriately **Error States**: Provide clear, actionable error messages that help users resolve issues **Empty States**: Guide users when there's no data to display, suggesting next actions

### Reactive User Interface

The ViewModels use LiveData to create a reactive user interface where: - **Automatic Updates**: UI automatically updates when data changes - **Lifecycle Awareness**: Prevents crashes by only updating active UI components - **Memory Efficiency**: Automatically cleans up when UI components are destroyed - **Consistent State**: Ensures the UI always reflects the current state of the data

---

## File-by-File Detailed Explanations

This section provides comprehensive English explanations for every file in the backend architecture, explaining what each file does, why it exists, and how it contributes to the overall application.

### Configuration Files

**app/build.gradle.kts - Project Dependencies and Build Configuration** This is the heart of the project setup. It defines: - **Dependencies**: All external libraries the app needs (Retrofit for networking, Glide for images, etc.) - **Build Features**: Enables ViewBinding for easier UI development - **Compile Settings**: Specifies which version of Android to target and support - **Plugin Configuration**: Sets up Kotlin compilation and annotation processing

**Why it matters**: This file determines what capabilities your app has and ensures all necessary libraries are available for the backend functionality.

**AndroidManifest.xml - App Permissions and Configuration** The manifest file is like the app's "ID card" that tells Android what the app can do: - **Permissions**: Declares that the app needs internet access, location services, and network monitoring - **Application Class**: Specifies that the app uses a custom Application class for initialization - **Security Configuration**: Points to network security settings that allow local development - **App Identity**: Sets the app name, icon, and other identifying information

**Why it matters**: Without proper manifest configuration, the app won't be able to access the internet or location services that are essential for travel planning.

**res/xml/network_security_config.xml - Development Network Settings** This file allows the app to communicate with local development servers: - **Local Development**: Permits HTTP (non-encrypted) connections to localhost - **Emulator Support**: Includes special localhost addresses used by Android emulators - **Security Balance**: Maintains security for production while enabling local testing

**Why it matters**: Essential for testing the app with a local backend server during development.

## Core Application Files

**WayfareApplication.kt - Application Initialization** This is the first class that runs when the app starts: - **Dependency Setup**: Creates the main dependency injection container - **Global Initialization**: Sets up app-wide configurations and managers - **Resource Management**: Ensures all necessary components are available throughout the app lifecycle

**Why it matters**: Provides the foundation that all other parts of the app depend on.

**di/AppContainer.kt - Dependency Injection Container** This class is like a "factory" that creates and manages all the app's major components: - **Component Creation**: Creates repositories, API services, and utility classes - **Dependency Resolution**: Ensures each component gets the dependencies it needs - **Singleton Management**: Ensures expensive objects are created only once and reused - **Configuration Management**: Centralizes how components are configured and connected

**Why it matters**: Keeps the app organized and makes it easy to test individual components by replacing them with mock versions.

**presentation/ViewModelFactory.kt - ViewModel Creation** A specialized factory that creates ViewModels with their required dependencies: - **ViewModel Instantiation**: Creates ViewModels with the correct repository dependencies - **Type Safety**: Ensures ViewModels are created with the correct types - **Dependency Injection**: Connects ViewModels to their data sources automatically

**Why it matters**: Enables the UI to get properly configured ViewModels without knowing about the complex dependency setup.

### Network and API Files

**data/api/NetworkConfig.kt - Network Infrastructure Setup** This file configures how the app communicates with the backend server: - **HTTP Client Setup**: Configures timeouts, connection pooling, and retry logic - **Authentication Integration**: Automatically adds user tokens to requests - **Logging Configuration**: Sets up request/response logging for debugging - **JSON Processing**: Configures how to convert between JSON and Kotlin objects

**Why it matters**: Ensures reliable, secure, and efficient communication with the backend server.

**data/api/interceptors/AuthInterceptor.kt - Automatic Authentication** An intelligent component that handles user authentication automatically: - **Token Injection**: Automatically adds authentication tokens to API requests - **Conditional Application**: Only adds tokens when needed (skips for login/register) - **Session Management**: Works with the session manager to get current user tokens

**Why it matters**: Users don't have to log in repeatedly, and developers don't have to manually handle authentication for every API call.

**API Service Files (UserApiService.kt, RouteApiService.kt, etc.)** These files define the "contract" between the Android app and the backend server: - **Endpoint Definition**: Specifies exactly which URLs to call for each operation - **Request/Response Structure**: Defines what data to send and expect back - **HTTP Method Specification**: Determines whether to use GET, POST, PUT, or DELETE - **Parameter Handling**: Manages URL parameters, request bodies, and headers

**Why it matters**: These serve as the official documentation and implementation of how the app talks to the server.

### Data Management Files

**Data Transfer Object (DTO) Files** These files in the `data/api/dto/` directories define the exact structure of data exchanged with the server: - **API Compatibility**: Match the exact format expected by the backend API - **Type**

**Safety**: Prevent runtime errors by defining expected data types - **Serialization**: Enable automatic conversion between JSON and Kotlin objects - **Validation**: Ensure required fields are present before sending requests

**Why they matter**: They serve as a protective barrier that ensures data integrity between the app and server.

**Domain Model Files (User.kt, Route.kt, etc.)** These files in `domain/model/` represent the core business concepts: - **Business Logic**: Contain only information relevant to travel planning business rules - **Platform Independence**: Can be used regardless of whether data comes from API, database, or cache - **Clean Structure**: Organized around what makes sense for users, not technical constraints - **Immutability**: Designed to prevent accidental data corruption

**Why they matter**: They represent the "truth" about what the app is supposed to do, independent of technical implementation details.

**Mapper Files (UserMapper.kt, RouteMapper.kt, etc.)** These files translate data between different formats: - **Format Translation**: Convert server data format to app-friendly format - **Data Enrichment**: Add calculated fields or default values - **Compatibility Layer**: Handle differences between API versions - **Data Cleaning**: Filter out unnecessary information

**Why they matter**: They keep the app's internal data structure independent of the server's data structure, making both easier to maintain.

**Repository Implementation Files** These files in `data/repository/` coordinate all data operations: - **Data Source Coordination**: Decide whether to use cached data or fetch fresh data - **Error Handling**: Convert technical errors into user-friendly messages - **Business Logic Implementation**: Implement data-related business rules - **Performance Optimization**: Manage caching and background data loading

**Why they matter**: They contain the "intelligence" about how to efficiently and reliably manage data.

### Utility and Helper Files

**utils/Constants.kt - Application-Wide Values** A central repository for all constant values used throughout the app: - **API Configuration**: Base URLs, timeout values, and endpoint paths - **Preference Keys**: Keys for storing user preferences and session data - **Error Messages**: Standard error messages used throughout the app - **Business Constants**: Travel categories, budget levels, and other business-related values

**Why it matters**: Having constants in one place makes the app easier to maintain and reduces the chance of typos.

**utils/NetworkUtils.kt - Network Operation Helpers** Utility functions for network-related operations: - **Connectivity Checking**: Determines if the device has internet connection - **Error Processing**: Converts network errors into user-friendly messages - **Response Handling**: Provides consistent handling of API responses - **Result Formatting**: Standardizes how success and error results are returned

**Why it matters**: Provides a consistent way to handle network operations throughout the app.

**utils/ValidationUtils.kt - Input Validation** Functions to validate user input before sending to the server: - **Email Validation**: Ensures email addresses are properly formatted - **Password Strength**: Checks password requirements for security - **Form Validation**: Validates complex forms like route creation - **Business Rule Validation**: Ensures data follows travel planning business rules

**Why it matters**: Prevents invalid data from being sent to the server and provides immediate feedback to users.

**utils/SharedPreferencesManager.kt - Local Data Storage** Manages storage of user preferences and session information: - **Session Management**: Stores and retrieves user login tokens - **Preference Storage**: Saves user settings and travel preferences - **Data Encryption**: Securely stores sensitive information - **Cache Management**: Manages locally cached data for offline access

**Why it matters**: Keeps users logged in and remembers their preferences between app sessions.

**utils/Extensions.kt - Convenience Functions** Kotlin extension functions that make common operations easier: - **UI Helpers**: Functions for showing toast messages and managing keyboard - **Data Formatting**: Helper functions for formatting dates, currencies, and other data - **Navigation Helpers**: Simplified functions for common navigation operations

**Why it matters**: Reduces code duplication and makes the codebase more readable.

**utils/ImageLoader.kt - Image Management** Specialized utility for loading and managing images: - **Performance Optimization**: Implements image caching and compression - **Loading States**: Handles placeholders and error images - **Memory Management**: Prevents memory leaks from image loading - **Format Support**: Handles different image formats and sources

**Why it matters**: Essential for a travel app that displays many images while maintaining good performance.

**utils/LocationUtils.kt - GPS and Location Services** Utilities for location-related operations: - **Permission Management**: Handles asking for and checking location permissions - **GPS Status**: Checks if location services are enabled - **Distance Calculations**: Calculates distances between locations - **Location Formatting**: Formats coordinates and addresses for display

**Why it matters**: Critical for a travel app that needs to work with user location and geographic data.

### Presentation Layer Files

**ViewModel Files (LoginViewModel.kt, RouteListViewModel.kt, etc.)** Each ViewModel manages the logic for specific screens or features: - **User Interaction Handling**: Processes user input and coordinates appropriate responses - **State Management**: Maintains the current state of the user interface - **Data Coordination**: Communicates with repositories to fetch and update data - **Error Handling**: Processes errors and formats them for user display - **Navigation Logic**: Determines when and how to navigate between screens

**Why they matter**: They contain all the "smart" behavior that makes the user interface responsive and intelligent.

This comprehensive file breakdown shows how each component contributes to creating a robust, maintainable, and user-friendly travel planning application. Every file has a specific purpose and works together with others to create the complete backend architecture.

## State Management

### ApiResult Pattern

```kotlin
sealed class ApiResult<out T> {
    data class Success<out T>(val data: T) : ApiResult<T>()
    data class Error(val message: String, val code: Int = -1) : ApiResult<Nothing>()
    data object Loading : ApiResult<Nothing>()
}

// Extension functions for easier 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
}
```

## Utility Classes

### Constants

```
object Constants {
    // API Configuration
    const val BASE_URL = "http://localhost:8000/"
    const val API_TIMEOUT = 30L // seconds

    // SharedPreferences Keys
    const val PREF_NAME = "wayfare_preferences"
    const val PREF_ACCESS_TOKEN = "access_token"
    const val PREF_USER_ID = "user_id"
    const val PREF_IS_LOGGED_IN = "is_logged_in"

    // Travel Preferences
    object TravelStyle {
        const val RELAXED = "relaxed"
        const val MODERATE = "moderate"
        const val ACCELERATED = "accelerated"
    }

    object Budget {
        const val LOW = "low"
        const val MEDIUM = "medium"
        const val HIGH = "high"
    }

    // Error Messages
    const val ERROR_NETWORK = "Network error. Please check your connection."
    const val ERROR_UNAUTHORIZED = "Session expired. Please log in again."
    const val ERROR_SERVER = "Server error. Please try again later."

    // Date Formats
    const val DATE_FORMAT_API = "yyyy-MM-dd"
    const val DATE_FORMAT_DISPLAY = "MMM dd, yyyy"
}
```

### NetworkUtils

```
object NetworkUtils {
    fun isNetworkAvailable(context: Context): Boolean {
        val connectivityManager = context.getSystemService(Context.CONNECTIVITY_SERVICE) as
        val network = connectivityManager.activeNetwork ?: return false
        val capabilities = connectivityManager.getNetworkCapabilities(network) ?: return fal

        return capabilities.hasTransport(NetworkCapabilities.TRANSPORT_WIFI) ||
                capabilities.hasTransport(NetworkCapabilities.TRANSPORT_CELLULAR) ||
                capabilities.hasTransport(NetworkCapabilities.TRANSPORT_ETHERNET)
    }

    fun getErrorMessage(throwable: Throwable): String {
        return when (throwable) {
            is IOException -> Constants.ERROR_NETWORK
            is SocketTimeoutException -> Constants.ERROR_NETWORK
            is HttpException -> {
                when (throwable.code()) {
                    401 -> Constants.ERROR_UNAUTHORIZED
                    400 -> Constants.ERROR_VALIDATION
                    in 500..599 -> Constants.ERROR_SERVER
                    else -> Constants.ERROR_UNKNOWN
                }
            }
            else -> Constants.ERROR_UNKNOWN
        }
    }

    fun handleApiError(throwable: Throwable): ApiResult.Error {
        val message = getErrorMessage(throwable)
        val code = if (throwable is HttpException) throwable.code() else -1
        return ApiResult.Error(message, code)
    }
}
```

### ValidationUtils

```
object ValidationUtils {
    fun isValidEmail(email: String): Boolean {
        return email.isNotEmpty() && Patterns.EMAIL_ADDRESS.matcher(email).matches()
    }

    fun isValidPassword(password: String): Boolean {
        return password.length >= 8
    }

    fun isValidUsername(username: String): Boolean {
        return username.length >= 3 && username.matches(Regex("^[a-zA-Z0-9_]+$"))
    }

    fun validateRegistration(
        username: String,
        email: String,
        password: String,
        firstName: String,
        lastName: String
    ): ValidationResult {
        return when {
            !isValidUsername(username) -> ValidationResult(false, "Username must be at least
            !isValidEmail(email) -> ValidationResult(false, "Please enter a valid email addr
            !isValidPassword(password) -> ValidationResult(false, "Password must be at least
            !isValidName(firstName) -> ValidationResult(false, "Please enter a valid first n
            !isValidName(lastName) -> ValidationResult(false, "Please enter a valid last nam
            else -> ValidationResult(true)
        }
    }

    data class ValidationResult(
        val isValid: Boolean,
        val errorMessage: String? = null
    )
}
```

**SharedPreferencesManager**

```
class SharedPreferencesManager(context: Context) {
    private val sharedPreferences: SharedPreferences =
        context.getSharedPreferences(Constants.PREF_NAME, Context.MODE_PRIVATE)

    fun saveAccessToken(token: String) {
        sharedPreferences.edit().putString(Constants.PREF_ACCESS_TOKEN, token).apply()
    }

    fun getAccessToken(): String? {
        return sharedPreferences.getString(Constants.PREF_ACCESS_TOKEN, null)
    }

    fun isLoggedIn(): Boolean {
        return sharedPreferences.getBoolean(Constants.PREF_IS_LOGGED_IN, false) &&
               getAccessToken() != null
    }

    fun saveUserSession(
        accessToken: String,
        userId: String,
        username: String,
        email: String
    ) {
        sharedPreferences.edit().apply {
            putString(Constants.PREF_ACCESS_TOKEN, accessToken)
            putString(Constants.PREF_USER_ID, userId)
            putString(Constants.PREF_USERNAME, username)
            putString(Constants.PREF_EMAIL, email)
            putBoolean(Constants.PREF_IS_LOGGED_IN, true)
            apply()
        }
    }

    fun clearUserSession() {
        sharedPreferences.edit().apply {
            remove(Constants.PREF_ACCESS_TOKEN)
            remove(Constants.PREF_USER_ID)
            remove(Constants.PREF_USERNAME)
            remove(Constants.PREF_EMAIL)
            putBoolean(Constants.PREF_IS_LOGGED_IN, false)
            apply()
        }
    }
}
```

---

## Dependency Injection

### AppContainer

```
class AppContainer(private val context: Context) {

    // Shared Preferences Manager
    val sharedPreferencesManager: SharedPreferencesManager by lazy {
        SharedPreferencesManager(context)
    }

    // Retrofit instance
    private val retrofit: Retrofit by lazy {
        NetworkConfig.createRetrofit(sharedPreferencesManager)
    }

    // API Services
    private val userApiService: UserApiService by lazy {
        NetworkConfig.createUserApiService(retrofit)
    }

    private val routeApiService: RouteApiService by lazy {
        NetworkConfig.createRouteApiService(retrofit)
    }
    // Repository implementations
    val userRepository: UserRepository by lazy {
        UserRepositoryImpl(userApiService, sharedPreferencesManager)
    }

    val routeRepository: RouteRepository by lazy {
        RouteRepositoryImpl(routeApiService, sharedPreferencesManager)
    }

    // ... other repositories
}
```

### Application Class

```
class WayfareApplication : Application() {
    lateinit var container: AppContainer

    override fun onCreate() {
        super.onCreate()
        container = AppContainer(this)
    }
}
```

### ViewModelFactory

```
class ViewModelFactory(
    private val userRepository: UserRepository,
    private val routeRepository: RouteRepository,
    private val placeRepository: PlaceRepository,
    private val locationRepository: LocationRepository,
    private val feedbackRepository: FeedbackRepository
) : ViewModelProvider.Factory {

    @Suppress("UNCHECKED_CAST")
    override fun <T : ViewModel> create(modelClass: Class<T>): T {
        return when (modelClass) {
            LoginViewModel::class.java -> LoginViewModel(userRepository) as T
            RegisterViewModel::class.java -> RegisterViewModel(userRepository) as T
            RouteListViewModel::class.java -> RouteListViewModel(routeRepository) as T
            RouteDetailViewModel::class.java -> RouteDetailViewModel(routeRepository) as T
            // ... other ViewModels
            else -> throw IllegalArgumentException("Unknown ViewModel class: ${modelClass.na
        }
    }
}
```

## Network Configuration

### NetworkConfig

```kotlin
object NetworkConfig {
    fun createRetrofit(sharedPreferencesManager: SharedPreferencesManager): Retrofit {
        val loggingInterceptor = HttpLoggingInterceptor().apply {
            level = HttpLoggingInterceptor.Level.BODY
        }

        val authInterceptor = AuthInterceptor(sharedPreferencesManager)

        val okHttpClient = OkHttpClient.Builder()
            .addInterceptor(loggingInterceptor)
            .addInterceptor(authInterceptor)
            .connectTimeout(Constants.API_TIMEOUT, TimeUnit.SECONDS)
            .readTimeout(Constants.API_TIMEOUT, TimeUnit.SECONDS)
            .writeTimeout(Constants.API_TIMEOUT, TimeUnit.SECONDS)
            .build()

        return Retrofit.Builder()
            .baseUrl(Constants.BASE_URL)
            .client(okHttpClient)
            .addConverterFactory(GsonConverterFactory.create())
            .build()
    }
}
```

### AuthInterceptor

```kotlin
class AuthInterceptor(
    private val sharedPreferencesManager: SharedPreferencesManager
) : Interceptor {

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

        // Skip adding authorization header if it's already present
        if (originalRequest.header("Authorization") != null) {
            return chain.proceed(originalRequest)
        }

        val accessToken = sharedPreferencesManager.getAccessToken()

        return if (accessToken != null) {
            val newRequest = originalRequest.newBuilder()
                .addHeader("Authorization", "Bearer $accessToken")
                .build()
            chain.proceed(newRequest)
        } else {
            chain.proceed(originalRequest)
        }
    }
}
```

---

## Image Loading System

### ImageLoader

```
object ImageLoader {
    fun loadImage(
        context: Context,
        imageUrl: String?,
        imageView: ImageView,
        placeholder: Int = R.drawable.ic_placeholder_image,
        error: Int = R.drawable.ic_error_image
    ) {
        Glide.with(context)
            .load(imageUrl)
            .apply(
                RequestOptions()
                    .placeholder(placeholder)
                    .error(error)
                    .diskCacheStrategy(DiskCacheStrategy.ALL)
                    .centerCrop()
            )
            .into(imageView)
    }

    fun loadCircularImage(
        context: Context,
        imageUrl: String?,
        imageView: ImageView,
        placeholder: Int = R.drawable.ic_profile_placeholder
    ) {
        Glide.with(context)
            .load(imageUrl)
            .apply(
                RequestOptions()
                    .placeholder(placeholder)
                    .error(placeholder)
                    .diskCacheStrategy(DiskCacheStrategy.ALL)
                    .circleCrop()
            )
            .into(imageView)
    }

    fun loadPlaceImage(
        context: Context,
        imageUrl: String?,
        imageView: ImageView,
        width: Int = 400,
        height: Int = 300
    ) {
        Glide.with(context)
            .load(imageUrl)
            .apply(
                RequestOptions()
                    .placeholder(R.drawable.ic_place_placeholder)
                    .error(R.drawable.ic_place_placeholder)
                    .diskCacheStrategy(DiskCacheStrategy.ALL)
                    .override(width, height)
                    .centerCrop()
            )
            .into(imageView)
    }
}
```

### Glide Configuration

```
@GlideModule
class WayfareGlideModule : AppGlideModule() {
    override fun registerComponents(context: Context, glide: Glide, registry: Registry) {
        val client = OkHttpClient.Builder()
            .connectTimeout(30, TimeUnit.SECONDS)
            .readTimeout(30, TimeUnit.SECONDS)
            .writeTimeout(30, TimeUnit.SECONDS)
            .build()

        registry.replace(
            GlideUrl::class.java,
            InputStream::class.java,
            OkHttpUrlLoader.Factory(client)
        )
    }

    override fun isManifestParsingEnabled(): Boolean {
        return false
    }
}
```

---

## Location Services

### LocationUtils

```
object LocationUtils {
    fun hasLocationPermissions(context: Context): Boolean {
        return ContextCompat.checkSelfPermission(
            context,
            Manifest.permission.ACCESS_FINE_LOCATION
        ) == PackageManager.PERMISSION_GRANTED &&
                ContextCompat.checkSelfPermission(
                    context,
                    Manifest.permission.ACCESS_COARSE_LOCATION
                ) == PackageManager.PERMISSION_GRANTED
    }

    fun isGpsEnabled(context: Context): Boolean {
        val locationManager = context.getSystemService(Context.LOCATION_SERVICE) as Locatio
        return locationManager.isProviderEnabled(LocationManager.GPS_PROVIDER) ||
                locationManager.isProviderEnabled(LocationManager.NETWORK_PROVIDER)
    }

    fun calculateDistance(
        lat1: Double, lng1: Double,
        lat2: Double, lng2: Double
    ): Float {
        val results = FloatArray(1)
        Location.distanceBetween(lat1, lng1, lat2, lng2, results)
        return results[0] / 1000 // Convert meters to kilometers
    }

    fun formatDistance(distanceKm: Float): String {
        return when {
            distanceKm < 1 -> "${(distanceKm * 1000).toInt()} m"
            distanceKm < 10 -> String.format("%.1f km", distanceKm)
            else -> "${distanceKm.toInt()} km"
        }
    }
}
```

### Location Permission Helper

```
class LocationPermissionHelper(
    private val activity: FragmentActivity,
    private val callback: LocationPermissionCallback
) {

    private val requestPermissionLauncher = activity.registerForActivityResult(
        ActivityResultContracts.RequestMultiplePermissions()
    ) { permissions ->
        val fineLocationGranted = permissions[Manifest.permission.ACCESS_FINE_LOCATION] ?: f
        val coarseLocationGranted = permissions[Manifest.permission.ACCESS_COARSE_LOCATION]

        if (fineLocationGranted && coarseLocationGranted) {
            callback.onPermissionGranted()
        } else {
            callback.onPermissionDenied()
        }
    }

    fun requestPermissions() {
        if (LocationUtils.hasLocationPermissions(activity)) {
            callback.onPermissionGranted()
        } else {
            requestPermissionLauncher.launch(
                arrayOf(
                    Manifest.permission.ACCESS_FINE_LOCATION,
                    Manifest.permission.ACCESS_COARSE_LOCATION
                )
            )
        }
    }
}

interface LocationPermissionCallback {
    fun onPermissionGranted()
    fun onPermissionDenied()
}
```

---

## Implementation Guide

### Step 1: Project Setup

1. Add dependencies to `build.gradle.kts`
2. Update `AndroidManifest.xml` with permissions and application class

3. Create network security configuration for localhost

### Step 2: Initialize Application

```kotlin
// In MainActivity or SplashActivity
class MainActivity : AppCompatActivity() {
    private val userRepository by lazy {
        getAppContainer().userRepository
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        // Check if user is logged in
        if (userRepository.isLoggedIn()) {
            // Navigate to main app
            navigateToMainApp()
        } else {
            // Show login screen
            navigateToLogin()
        }
    }
}
```

### Step 3: Create UI Components

```kotlin
// Example: LoginActivity
class LoginActivity : AppCompatActivity() {

    private val loginViewModel: LoginViewModel by lazy {
        val factory = getAppContainer().createViewModelFactory()
        ViewModelProvider(this, factory)[LoginViewModel::class.java]
    }

    private lateinit var binding: ActivityLoginBinding

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

        setupObservers()
        setupClickListeners()
    }

    private fun setupObservers() {
        loginViewModel.loginState.observe(this) { state ->
            when (state) {
                is LoginState.Success -> {
                    startActivity(Intent(this, MainActivity::class.java))
                    finish()
                }
                is LoginState.Error -> {
                    binding.root.showSnackbar(state.message)
                }
                LoginState.Idle -> {
                    // Initial state
                }
            }
        }

        loginViewModel.isLoading.observe(this) { isLoading ->
            binding.progressBar.visibility = if (isLoading) View.VISIBLE else View.GONE
            binding.loginButton.isEnabled = !isLoading
        }

        loginViewModel.usernameError.observe(this) { error ->
            binding.usernameInputLayout.error = error
        }

        loginViewModel.passwordError.observe(this) { error ->
            binding.passwordInputLayout.error = error
        }
    }

    private fun setupClickListeners() {
        binding.loginButton.setOnClickListener {
            val username = binding.usernameEditText.text.toString()
            val password = binding.passwordEditText.text.toString()

            loginViewModel.login(username, password)
        }

        binding.registerTextView.setOnClickListener {
            startActivity(Intent(this, RegisterActivity::class.java))
        }
    }
}
```

### Step 4: Handle RecyclerViews

```kotlin
// Example: RouteListAdapter
class RouteListAdapter(
    private val onRouteClick: (Route) -> Unit
) : RecyclerView.Adapter<RouteListAdapter.RouteViewHolder>() {

    private var routes = emptyList<Route>()

    fun submitList(newRoutes: List<Route>) {
        routes = newRoutes
        notifyDataSetChanged()
    }

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): RouteViewHolder {
        val binding = ItemRouteBinding.inflate(
            LayoutInflater.from(parent.context), parent, false
        )
        return RouteViewHolder(binding)
    }

    override fun onBindViewHolder(holder: RouteViewHolder, position: Int) {
        holder.bind(routes[position])
    }

    override fun getItemCount() = routes.size

    inner class RouteViewHolder(
        private val binding: ItemRouteBinding
    ) : RecyclerView.ViewHolder(binding.root) {

        fun bind(route: Route) {
            binding.titleTextView.text = route.title
            binding.cityTextView.text = route.city
            binding.datesTextView.text = DateUtils.formatDateRange(route.startDate, route.en

            // Category icon
            when (route.category) {
                Constants.Category.CITY_BREAK -> binding.categoryIcon.setImageResource(R.dra
                Constants.Category.BEACH -> binding.categoryIcon.setImageResource(R.drawable
                // ... other categories
            }

            binding.root.setOnClickListener {
                onRouteClick(route)
            }
        }
    }
}
```

---

## API Integration

### Supported Endpoints

### User Management

- `POST /user/register` - User registration
- `POST /user/login` - User authentication
- `GET /user/getCurrentUser` - Get user profile
- `POST /user/addInfo` - Update user preferences
- `POST /user/changePassword` - Change password
- `DELETE /user/delete` - Delete account
- `POST /user/sendVerification` - Send email verification
- `POST /user/sendVerification/verifyCode` - Verify email code

### Route Management

- `POST /route/create` - Create new route
- `GET /routes/user` - Get user's routes
- `GET /routes/{route_id}` - Get route details
- `PUT /routes/{route_id}` - Update route
- `DELETE /routes/{route_id}` - Delete route
- `GET /routes/public` - Browse public routes

### Place Management

- `GET /places/city` - Get places by city
- `POST /places/id` - Get places by IDs
- `POST /places/search` - Search places
- `POST /places/autocomplete` - Get autocomplete suggestions

### Location Management

- `GET /cities/all` - Get all cities
- `POST /cities/specific` - Get cities by country
- `GET /countries/all` - Get all countries
- `POST /countries/region` - Get countries by region
- `GET /countries/allRegions` - Get all regions

### Feedback Management

- `POST /feedback/place` - Submit place feedback
- `GET /feedback/place/{place_id}` - Get place feedback

- `PUT /feedback/place/{feedback_id}` - Update place feedback
- `DELETE /feedback/place/{feedback_id}` - Delete place feedback
- `POST /feedback/route` - Submit route feedback
- `GET /feedback/route/{route_id}` - Get route feedback

### Authentication Flow

```
// 1. User login
loginViewModel.login(username, password)

// 2. Token stored automatically by UserRepositoryImpl
// 3. AuthInterceptor adds Bearer token to all subsequent requests
// 4. Token validation handled automatically

// 5. On 401 response, user needs to login again
loginViewModel.loginState.observe(this) { state ->
    when (state) {
        is LoginState.Error -> {
            if (state.message.contains("unauthorized", ignoreCase = true)) {
                // Redirect to login
                startActivity(Intent(this, LoginActivity::class.java))
                finish()
            }
        }
    }
}
```

---

## Error Handling

### Error Handling Strategy

```
// 1. Network-level errors (NetworkUtils)
fun handleApiError(throwable: Throwable): ApiResult.Error {
    val message = when (throwable) {
        is IOException -> "Network error. Please check your connection."
        is HttpException -> {
            when (throwable.code()) {
                401 -> "Session expired. Please log in again."
                400 -> "Invalid input data."
                500 -> "Server error. Please try again later."
                else -> "An unexpected error occurred."
            }
        }
        else -> "An unexpected error occurred."
    }
    return ApiResult.Error(message, throwable.code())
}

// 2. Repository-level error handling
override suspend fun login(userLogin: UserLogin): ApiResult<AuthTokens> {
    return try {
        val response = userApiService.login(request)
        if (response.isSuccessful) {
            // Handle success
            ApiResult.Success(authTokens)
        } else {
            ApiResult.Error("Invalid credentials", response.code())
        }
    } catch (e: Exception) {
        NetworkUtils.handleApiError(e)
    }
}

// 3. ViewModel-level error handling
fun login(username: String, password: String) {
    viewModelScope.launch {
        when (val result = userRepository.login(userLogin)) {
            is ApiResult.Success -> {
                _loginState.value = LoginState.Success(result.data)
            }
            is ApiResult.Error -> {
                _loginState.value = LoginState.Error(result.message)
            }
        }
    }
}

// 4. UI-level error handling
loginViewModel.loginState.observe(this) { state ->
    when (state) {
        is LoginState.Error -> {
            binding.root.showSnackbar(state.message)
        }
    }
}
```

**Error Types**

- **Network Errors**: Connection issues, timeouts
- **Authentication Errors**: Invalid credentials, expired tokens
- **Validation Errors**: Invalid input data

- **Server Errors**: 5xx HTTP status codes
- **Unknown Errors**: Unexpected exceptions

---

## Testing Strategy

### Unit Testing

```kotlin
// ViewModel Testing
@RunWith(MockitoJUnitRunner::class)
class LoginViewModelTest {

    @Mock
    private lateinit var userRepository: UserRepository

    private lateinit var loginViewModel: LoginViewModel

    @Before
    fun setup() {
        loginViewModel = LoginViewModel(userRepository)
    }

    @Test
    fun `login with valid credentials should return success`() = runBlocking {
        // Given
        val userLogin = UserLogin("username", "password")
        val authTokens = AuthTokens("token")
        whenever(userRepository.login(userLogin)).thenReturn(ApiResult.Success(authTokens))

        // When
        loginViewModel.login("username", "password")

        // Then
        val state = loginViewModel.loginState.getOrAwaitValue()
        assertTrue(state is LoginState.Success)
        assertEquals(authTokens, (state as LoginState.Success).tokens)
    }

    @Test
    fun `login with invalid credentials should return error`() = runBlocking {
        // Given
        val userLogin = UserLogin("username", "wrong_password")
        whenever(userRepository.login(userLogin)).thenReturn(ApiResult.Error("Invalid creden

        // When
        loginViewModel.login("username", "wrong_password")
        // Then
        val state = loginViewModel.loginState.getOrAwaitValue()
        assertTrue(state is LoginState.Error)
        assertEquals("Invalid credentials", (state as LoginState.Error).message)
    }
}
```

**Repository Testing**

```kotlin
@RunWith(MockitoJUnitRunner::class)
class UserRepositoryImplTest {

    @Mock
    private lateinit var userApiService: UserApiService

    @Mock
    private lateinit var sharedPreferencesManager: SharedPreferencesManager

    private lateinit var userRepository: UserRepositoryImpl

    @Before
    fun setup() {
        userRepository = UserRepositoryImpl(userApiService, sharedPreferencesManager)
    }

    @Test
    fun `login should store token on success`() = runBlocking {
        // Given
        val loginRequest = LoginRequest("username", "password")
        val loginResponse = LoginResponse("Success", true, "access_token")
        val response = Response.success(loginResponse)

        whenever(userApiService.login(loginRequest)).thenReturn(response)

        // When
        val result = userRepository.login(UserLogin("username", "password"))

        // Then
        assertTrue(result is ApiResult.Success)
        verify(sharedPreferencesManager).saveAccessToken("access_token")
        verify(sharedPreferencesManager).setLoggedIn(true)
    }
}
```

### Integration Testing

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

    @get:Rule
    val activityRule = ActivityTestRule(LoginActivity::class.java)

    @Test
    fun loginFlow() {
        // Enter username and password
        onView(withId(R.id.usernameEditText))
            .perform(typeText("testuser"))

        onView(withId(R.id.passwordEditText))
            .perform(typeText("testpassword"))

        // Click login button
        onView(withId(R.id.loginButton))
            .perform(click())

        // Verify navigation to main activity
        intended(hasComponent(MainActivity::class.java.name))
    }
}
```

---

## Performance Optimization

### Network Optimization

- **Connection pooling** with OkHttp
- **Request/response caching** with Retrofit
- **Image caching** with Glide
- **Concurrent requests** with Kotlin Coroutines

### Memory Management

- **ViewBinding** instead of findViewById
- **Lazy initialization** for dependencies
- **Proper lifecycle management** for ViewModels
- **Image recycling** with Glide

### Background Processing

- **Coroutines** for asynchronous operations
- **Background threads** for heavy operations

- **Main thread** for UI updates only

---

## Security Considerations

### Authentication

- **JWT token storage** in encrypted SharedPreferences
- **Automatic token injection** via AuthInterceptor
- **Token expiration handling** with re-authentication
- **Secure communication** over HTTPS (production)

### Data Protection

- **Input validation** before API calls
- **SQL injection prevention** with parameterized queries
- **Network security config** for development
- **Certificate pinning** for production

### Privacy

- **Minimal permissions** requested
- **User consent** for location services
- **Data encryption** for sensitive information
- **Secure deletion** of user data

---

## Monitoring and Analytics

### Logging

```
// Network logging with OkHttp
val loggingInterceptor = HttpLoggingInterceptor().apply {
    level = if (BuildConfig.DEBUG) {
        HttpLoggingInterceptor.Level.BODY
    } else {
        HttpLoggingInterceptor.Level.NONE
    }
}
```

### Error Tracking

```
// Custom error reporting
fun reportError(error: Throwable, context: String) {
    if (BuildConfig.DEBUG) {
        Log.e("WayfareError", "$context: ${error.message}", error)
    } else {
        // Report to crash analytics service
        FirebaseCrashlytics.getInstance().recordException(error)
    }
}
```

### Performance Metrics

- **API response times** monitoring
- **Image loading performance** tracking
- **Memory usage** optimization
- **Battery consumption** monitoring

---

## Deployment

### Build Configuration

```
android {
    buildTypes {
        debug {
            buildConfigField("String", "BASE_URL", "\"http://localhost:8000/\"")
            isDebuggable = true
        }
        release {
            buildConfigField("String", "BASE_URL", "\"https://api.wayfare.com/\"")
            isMinifyEnabled = true
            proguardFiles(getDefaultProguardFile("proguard-android-optimize.txt"), "proguard
        }
    }
}
```

### Release Checklist

- ☐ Update base URL for production
- ☐ Enable ProGuard/R8 obfuscation
- ☐ Remove debug logging
- ☐ Test all API endpoints
- ☐ Verify authentication flow
- ☐ Test offline scenarios
- ☐ Performance testing
- ☐ Security audit

---

## How Everything Works Together

This section explains how all the components we've built work together to create a complete, functional travel planning application. Understanding these connections is crucial for maintaining and extending the application.

### The Complete Data Flow

When a user performs any action in the app, data flows through the architecture in a predictable, organized way:

1. **User Interaction**: User taps a button, types in a field, or swipes on the screen
2. **ViewModel Processing**: The ViewModel validates the input and determines what needs to happen
3. **Repository Coordination**: The ViewModel calls the appropriate Repository method
4. **Data Source Decision**: The Repository decides whether to use cached data or fetch from the server
5. **API Communication**: If needed, the Repository uses an API Service to communicate with the server
6. **Data Translation**: Mappers convert between server format and app format
7. **Result Processing**: The Repository processes the response and handles any errors
8. **UI Update**: The ViewModel updates its state, and the UI automatically updates via LiveData

### Architecture Integration Examples

**Example 1: User Login Process** Here's how all components work together when a user logs in:

1. **UI Layer**: User enters username and password in LoginActivity
2. **LoginViewModel**: Validates input, shows loading state
3. **UserRepository**: Coordinates the login process
4. **UserApiService**: Sends login request to server using AuthInterceptor
5. **UserMapper**: Converts server response to app-friendly format
6. **SharedPreferencesManager**: Stores authentication token
7. **LoginViewModel**: Updates UI state to show success
8. **Navigation**: User is taken to the main app screen

**Example 2: Browsing Travel Routes** When a user wants to see available travel routes:

1. **RouteListActivity**: User opens the route browsing screen
2. **RouteListViewModel**: Automatically starts loading user's personal routes

3. **RouteRepository**: Checks if cached routes are still valid
4. **RouteApiService**: Fetches fresh route data if needed (with automatic authentication)
5. **RouteMapper**: Converts server data to user-friendly route objects
6. **RouteListViewModel**: Updates the route list
7. **UI**: RecyclerView automatically displays the new routes
8. **ImageLoader**: Loads route thumbnail images in the background

**Example 3: Creating a New Route** When a user creates a travel plan:

1. **RouteCreationActivity**: User fills out route details form
2. **RouteDetailViewModel**: Validates each field as user types
3. **ValidationUtils**: Ensures dates, locations, and budget are valid
4. **PlaceRepository**: Provides autocomplete suggestions for locations
5. **RouteRepository**: Saves the new route to the server
6. **RouteMapper**: Converts app format to server format
7. **RouteApiService**: Sends route creation request
8. **NetworkUtils**: Handles any network errors gracefully
9. **UI**: Shows success message and navigates to the new route

### Security and Authentication Flow

The app maintains security through coordinated component interaction:

1. **Session Management**: SharedPreferencesManager stores encrypted tokens
2. **Automatic Authentication**: AuthInterceptor adds tokens to all requests
3. **Token Validation**: Server validates tokens and returns 401 if expired
4. **Error Handling**: NetworkUtils detects authentication failures
5. **Automatic Logout**: App redirects to login screen when tokens expire
6. **Re-authentication**: Users can log in again seamlessly

### Performance Optimization Strategy

Multiple components work together to ensure good performance:

1. **Image Caching**: ImageLoader caches images to reduce network usage
2. **Data Caching**: Repositories cache recent data to avoid redundant requests
3. **Background Loading**: ViewModels load data in background threads
4. **Lazy Loading**: Lists load data in batches as users scroll
5. **Memory Management**: Components clean up automatically when not needed

#### Error Handling Coordination

When errors occur, multiple layers work together to provide a good user experience:

1. **Network Errors**: NetworkUtils provides user-friendly error messages
2. **Validation Errors**: ValidationUtils prevents invalid data from being sent
3. **Server Errors**: Repositories translate server errors into actionable messages
4. **UI Errors**: ViewModels format errors for display and suggest solutions
5. **Recovery Options**: App provides ways for users to retry or take alternative actions

#### Reactive UI Coordination

The app provides a responsive user interface through coordinated reactive programming:

1. **LiveData Observation**: UI components automatically update when data changes
2. **State Synchronization**: All UI elements reflect the current state consistently
3. **Loading States**: Users always know when operations are in progress
4. **Error Display**: Errors are shown immediately and clearly
5. **User Feedback**: Actions provide immediate visual feedback

#### Offline and Connectivity Handling

The architecture supports various connectivity scenarios:

1. **Connectivity Detection**: NetworkUtils monitors internet connection
2. **Cached Data**: Repositories serve cached data when offline
3. **Graceful Degradation**: App continues working with available data
4. **Sync Management**: Data syncs automatically when connection returns
5. **User Notification**: Users are informed about connectivity status

#### Maintainability Benefits

The architecture design makes the app easy to maintain and extend:

1. **Separation of Concerns**: Each component has a clear, single responsibility
2. **Dependency Injection**: Components can be easily replaced or updated
3. **Interface Contracts**: Repository interfaces allow changing implementations
4. **Consistent Patterns**: All similar operations follow the same patterns
5. **Testing Support**: Each component can be tested independently

#### Scalability Considerations

The architecture supports future growth and changes:

1. **Modular Design**: New features can be added without affecting existing code
2. **Repository Pattern**: New data sources (database, cache) can be added easily
3. **ViewModel Pattern**: New screens follow the same proven patterns
4. **Utility Classes**: Common functionality can be reused throughout the app
5. **Configuration Management**: Settings can be changed without code changes

#### Business Logic Organization

The app organizes business rules logically:

1. **Domain Models**: Core business entities are clearly defined
2. **Repository Interfaces**: Business operations are explicitly documented
3. **ViewModel Logic**: UI-specific business rules are contained in ViewModels
4. **Validation**: Business rules are enforced at appropriate layers
5. **Mapper Logic**: Data transformation rules are centralized and reusable

This interconnected architecture creates a robust, maintainable, and user-friendly travel planning application where every component has a clear purpose and works harmoniously with others to deliver a seamless user experience.

---

## Conclusion

This comprehensive documentation covers the complete backend architecture implementation for the Wayfare Android application. Every component has been designed with clear purposes and detailed English explanations of:

### What We've Built:

- **Complete Data Layer**: API services, DTOs, mappers, and repository implementations for all travel planning features
- **Robust Domain Layer**: Clean business models and interfaces that define the app's core functionality
- **Intelligent Presentation Layer**: ViewModels that handle all UI logic and user interactions
- **Comprehensive Utilities**: Helper classes for validation, networking, image loading, and location services
- **Solid Foundation**: Dependency injection, error handling, and configuration management

**Key Architectural Benefits:**

- **Maintainability**: Each component has a single responsibility and clear boundaries
- **Testability**: Components can be tested independently with mock dependencies
- **Scalability**: New features can be added without affecting existing functionality
- **Reliability**: Comprehensive error handling and state management throughout
- **Performance**: Optimized data loading, caching, and image management
- **Security**: Proper authentication handling and secure data storage

**Ready for UI Development**

The backend architecture is complete and ready for UI implementation. The next step is to create:

1. **Activities and Fragments**: Screen implementations that use the ViewModels
2. **XML Layouts**: User interface designs using the provided utility functions
3. **RecyclerView Adapters**: List components that display data from ViewModels
4. **Navigation Components**: Screen transitions and user flow implementation

Every ViewModel is documented with clear explanations of how to use it, what data it provides, and how it handles user interactions. The utility classes provide all necessary helper functions for common UI operations.

This documentation serves as both a technical reference and a guide for building upon the solid foundation we've created together.

---

**Document Version**: 1.0
**Last Updated**: December 2024
**Architecture**: MVVM with Clean Architecture principles
**Project**: Wayfare Android Travel Planning Application