TymexTest Project Technical Document

This document provides a technical overview of the Android project structure. It outlines the purpose and responsibilities of each package and key components within the project.

Project Overview

The project follows the **Clean Architecture** pattern, dividing the codebase into distinct layers:

1. **Data Layer**
2. **Domain Layer**
3. **Presentation Layer**

The structure is designed to ensure separation of concerns, scalability, and testability.

Package Structure and Component Details

1. `data`

Responsibilities:

- Handles data operations, including fetching data from remote APIs or local storage.
- They are as a bridge between the domain layer and external data sources.

Sub-Packages:

- `remote`: Contains classes and methods for interacting with remote APIs.
- `repository`: Implements the `UserRepository` interface from the domain layer, providing the actual data logic.

Key Classes:

- `UserRepository`: Implements user data fetching logic.

---

2. `domain`

Responsibilities:

- Contains the business logic of the application.
- Defines models, use cases, and repository interfaces.

Sub-Packages:

- `model`: Includes data models used across the application.
    - `User`: Represents user data.
    - `UserDetail`: Represents detailed information about a user.
- `repository`: Defines the `UserRepository` interface, ensuring the data layer adheres to a consistent contract.
- `usecase`: Contains application-specific business logic encapsulated in use cases.
    - `GetUsersUseCase`: Fetches a list of users.
    - `GetUserDetailUseCase`: Fetches details of a specific user.

---

3. `presentation`

Responsibilities:

- Handles UI and user interactions.
- Includes Activities, ViewModels, and Adapters for managing views and data binding.

Sub-Packages:

- `activities`:
  - `BaseActivity`: A base class for shared functionality across activities.
  - `MainActivity`: The main entry point of the application.
  - `UserDetailActivity`: Displays detailed information about a user.
- `adapters`:
  - `UserAdapter`: Provides a binding mechanism between a dataset and RecyclerView.
- `viewmodel`:
  - `UserViewModel`: Manages UI-related data for user lists.
  - `UserDetailViewModel`: Manages UI-related data for user details.

---

4. `di`

Responsibilities:

- Handles dependency injection (DI) using Hilt frameworks.

Key Classes:

- `AppModule`: Provides application-wide dependencies, such as repositories, network clients, application context, etc.

---

5. `utils`

Responsibilities:

- Contains utility classes and constants used throughout the application.

Key Classes:

- `Constants`: Stores static constant values for global use.

6. `Unittest`

Responsibilities:

- Contains Unittest classes used throughout the application.

Key Classes:

- `GetUsersUseCaseTest`: Testing class for UserUseCase.
- `UserViewModelTest`: Testing class for UserViewModel.

Root Files:

- `TymeXTestApp`: The application class, serving as the entry point for app initialization.

Development Considerations

1. **Scalability**: The modular structure allows for easy extension of features and addition of new components.

2. **Testability**: Use cases and repository interfaces enable unit testing of the business logic.

3. **Maintainability**: Clear separation of concerns ensures that changes in one layer have minimal impact on others.

Tools & Libraries

- Dependency Injection: Hilt.
- Other libraries : Retrofit, Glide, Gson, Mokito.
- Architecture: Clean Architecture pattern.

Build and Testing

Build

1. Open the project in **Android Studio**.
2. Sync the Gradle files to ensure all dependencies are resolved.
3. Build the project using the **Build > Make Project** option or the shortcut `Ctrl+F9` (Windows/Linux) or `Cmd+F9` (Mac).
4. Connect you Android device or start Simunator device to run the application by click the Run icon on Android Studio.

Testing

Unit Testing

1. Write unit tests for the `usecase` and `repository` classes in the `domain` and `data` layers respectively.
2. Use **JUnit** and **Mockito** for writing and mocking dependencies.
3. Run unit tests using the command:
    - **Run > Run Tests** in Android Studio.