



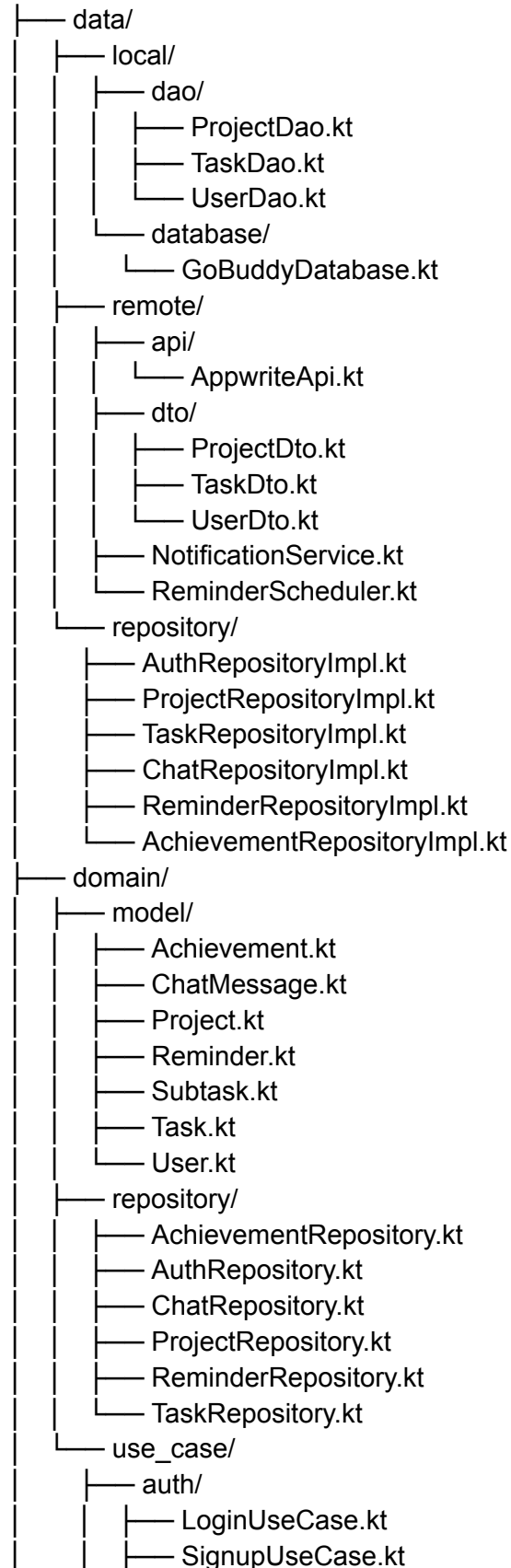
GO Buddy App Architecture Manual

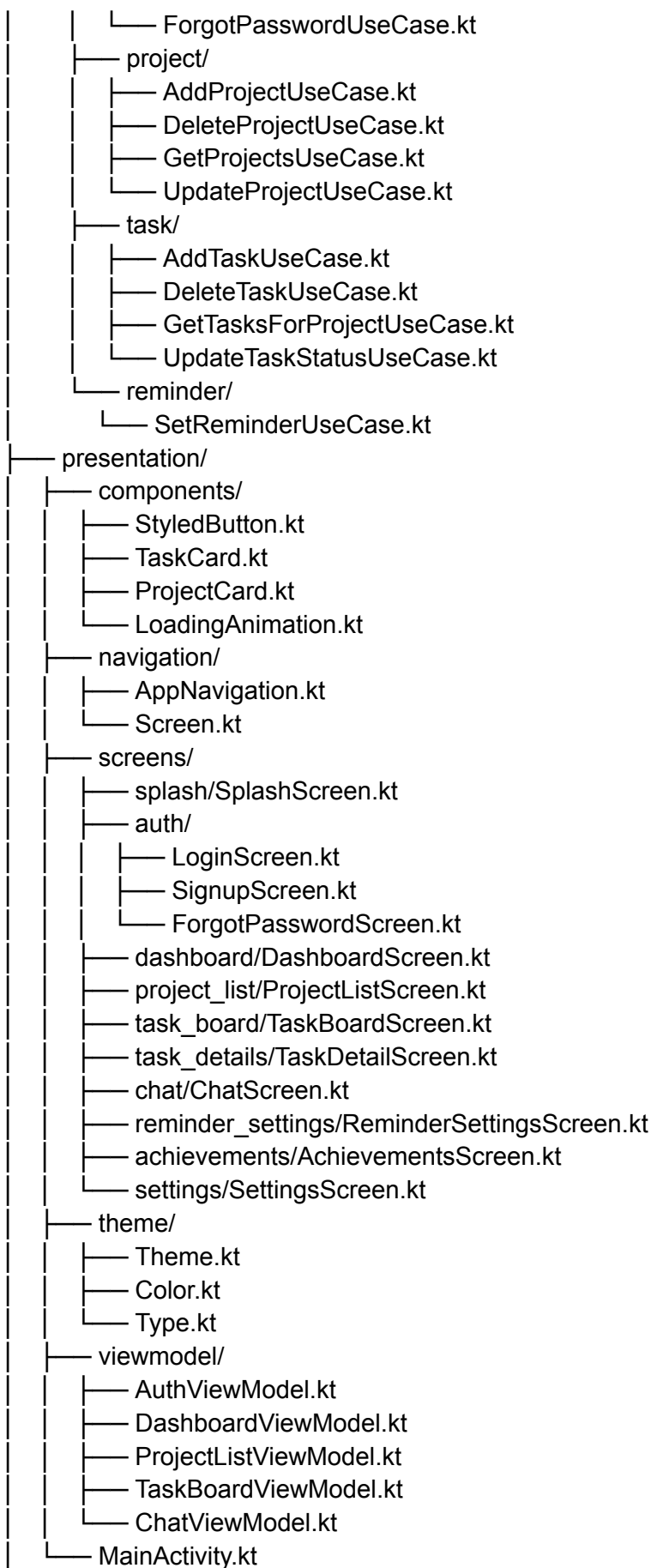
Firestore-Free Clean Architecture Guide



Full Project Structure Overview

mursalin.companion.gobuddy/





◆ File Naming Conventions

- `*ViewModel.kt` → ViewModel classes (UI logic)

- `*UseCase.kt` → Business logic files
 - `*RepositoryImpl.kt` → Data access implementation
 - `*Repository.kt` → Abstract interfaces
 - `*.kt` under `screens/` → Jetpack Compose screens
 - `*.kt` under `model/` → Core data classes
-

Section-by-Section Manual (with Concepts + What to Learn)

Section 1: `data/local/`

Handles local database logic via Room

- 💡 DAO interfaces (`ProjectDao.kt`, etc.)
 - 🧠 Uses: SQL queries, Kotlin Coroutines, OOP
 - 📖 Learn: Room, KAPT, DAO Patterns, SQL basics
-

Section 2: `data/remote/`

Handles API calls and notifications

- 🧠 Uses: REST abstraction, DTOs, local notification
 - 📖 Learn: Appwrite SDK, AlarmManager, Serialization, NotificationCompat
-

Section 3: `data/repository/`

Connects Room + Remote into clean data layer

- 🧠 Uses: Repository pattern, clean separation, mapper pattern
 - 📖 Learn: Dependency Inversion, Domain ↔ DTO Mapping, async programming
-

Section 4: `domain/model/`

Pure data models

- 🧠 Uses: DDD, immutability, clean data structures
 - 📖 Learn: Kotlin data class, Enums, Sealed Classes
-

📦 Section 5: **domain/repository/**

Interfaces for each repository

- 🧠 Uses: Interface-driven design
 - 📖 Learn: OOP, Dependency Inversion, Interface contracts
-

📦 Section 6: **domain/use_case/**

Each use case is a single responsibility class

- 🧠 Uses: Command pattern, testable pure logic
 - 📖 Learn: Clean Architecture, Unit Testing
-

📦 Section 7: **presentation/components/**

Reusable UI building blocks

- 🧠 Uses: Declarative UI, parameterized Composables
 - 📖 Learn: Jetpack Compose, UI design principles
-

📦 Section 8: **presentation/screens/**

Full-featured screens with state management

- 🧠 Uses: MVVM, UDF (Unidirectional Data Flow)
 - 📖 Learn: Navigation Compose, StateFlow, Lifecycle
-

📦 Section 9: **presentation/viewmodel/**



Manages screen state and actions

- 🧠 Uses: Flow, ViewModelScope, UI state exposure



-  Learn: Kotlin Flow, Clean ViewModel setup, Hilt (optional)
-

Section 10: `MainActivity.kt`

App entry point

-  Uses: `setContent`, theme application
 -  Learn: Compose lifecycle, themes, root navigation setup
-

Key Technologies You Should Learn

 Topic	 Why
Kotlin Coroutines + Flow	Async data, UI state
Jetpack Compose	UI framework
Room Database	Local storage
Appwrite SDK	Backend (Auth, DB, Functions)
AlarmManager/WorkManager	Reminder system
Clean Architecture	Maintainable code
MVVM	Best architecture pattern
DTO Mapping	Clean data flow
Hilt	Dependency injection
State Management	Jetpack best practices

◆ Summary for `data/local/`

The `data/local/` package is responsible for managing local data storage using **Room Database**. It includes DAO interfaces to interact with the database and an abstract Room class to configure entities. This section ensures offline support and fast data retrieval using local SQLite-based persistence. You'll use annotations, Kotlin Coroutines, and SQL queries to define and access your app's internal data.

◆ Summary for `data/remote/`

The `data/remote/` package handles all communication with external services like Appwrite and system-level features like notifications. It wraps Appwrite SDK calls into your custom API class, structures remote data with DTOs, and manages reminders using native Android schedulers like `AlarmManager`. This allows full backend integration without Firebase.

◆ Summary for **data/repository/**

The **data/repository/** package acts as a **bridge** between your app's domain logic and the actual data sources (local or remote). It implements interfaces declared in the **domain** layer and decides where the data comes from. This abstraction enables testability, separation of concerns, and aligns with Clean Architecture best practices.

◆ Summary for **domain/model/**

domain/model/ defines the **core data structures** of your app in a platform-independent and pure Kotlin way. These models have no dependency on Android SDK or any external library, which keeps them portable and reusable. This layer reflects how your app thinks about and processes data internally.

◆ Summary for **domain/repository/**

This layer outlines **contracts** or interfaces that the **data** layer must fulfill. By abstracting data sources, you enforce loose coupling between business logic and data handling. This pattern promotes scalability, flexibility, and makes it easy to swap data sources in the future (e.g., replace Appwrite or Room with another backend).

◆ Summary for **domain/use_case/**

Each use case represents a **specific feature action**—like logging in, updating a task, or deleting a project. These classes encapsulate business rules and ensure that application logic is organized, testable, and easy to understand. Use cases should be pure, with no dependency on UI or Android framework components.

◆ Summary for **presentation/components/**

This package includes small, reusable **UI components** used throughout the app's Compose screens. Whether it's a styled button or a card layout, these are built using **@Composable** functions and promote a consistent UI design and code reuse across the app.

◆ Summary for **presentation/screens/**

The **screens** package contains full Jetpack Compose UI screens grouped by features like login, task board, or dashboard. Each screen observes data from its ViewModel and updates accordingly. These are your **user-facing pages**, where state, layout, and interactivity come together.

◆ Summary for `presentation/viewmodel/`

ViewModels in this layer serve as the **controller** between UI and domain logic. They manage UI state, call use cases, and expose results to the Compose layer using `StateFlow`. This follows the MVVM pattern and ensures unidirectional, reactive UI updates.

◆ Summary for `MainActivity.kt`

`MainActivity.kt` is the **entry point** of your app. It initializes Jetpack Compose using `setContent {}` and hosts the app-wide navigation graph and theming logic. It's responsible for bootstrapping your app's layout and navigation.

◆ Summary for Technologies To Learn

This section maps the technologies, design patterns, and programming concepts you'll need to master for building GO Buddy. It includes Android system APIs, Jetpack Compose, architectural patterns like Clean Architecture, and backend integration with Appwrite. This serves as your personal roadmap to becoming a full-stack Android engineer.