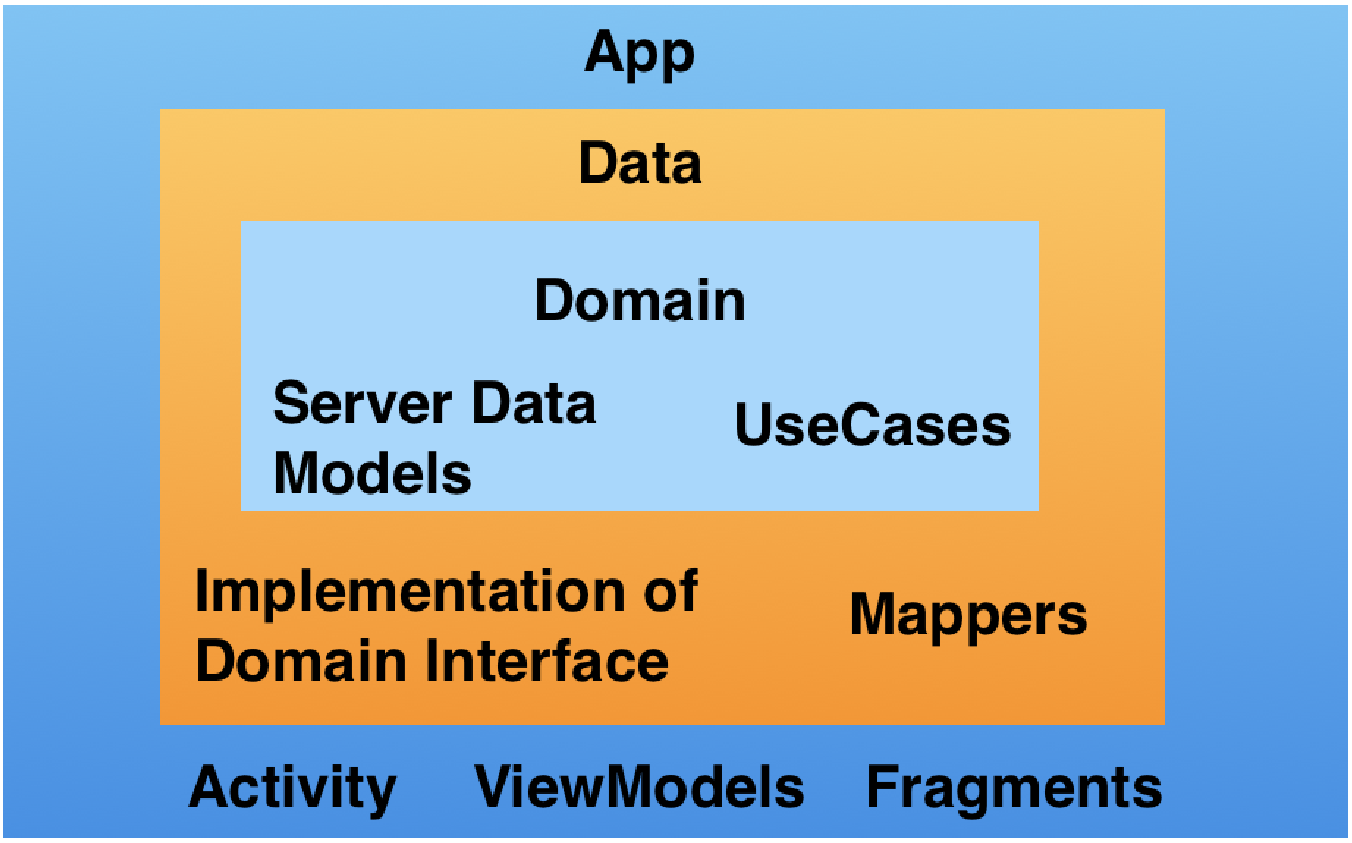


To implement the clean architecture in Android we need to first understand a few things.

UseCases

Repositories

Mappers

UseCases

These are also known as interactors. Each individual functionality or business logic unit can be called a use case. Like fetching data from a network or reading data from database, preferences, etc can be referred to as use cases. Usecases are the business logic executors that fetch data from data source either remote or local and gives it back to the requester in our case it would be the app layer.

940 | 8

Repository

The repository is an interface that we create in our domain module. And Repository Implementation will be in the data module. Here use cases acts as a mediator between our Repository interface and app module.

Mappers

Mappers by the name itself are suggesting that it maps from one type to another type. Actually, in our apps, we use the same object which we receive from the server to set details to UI.

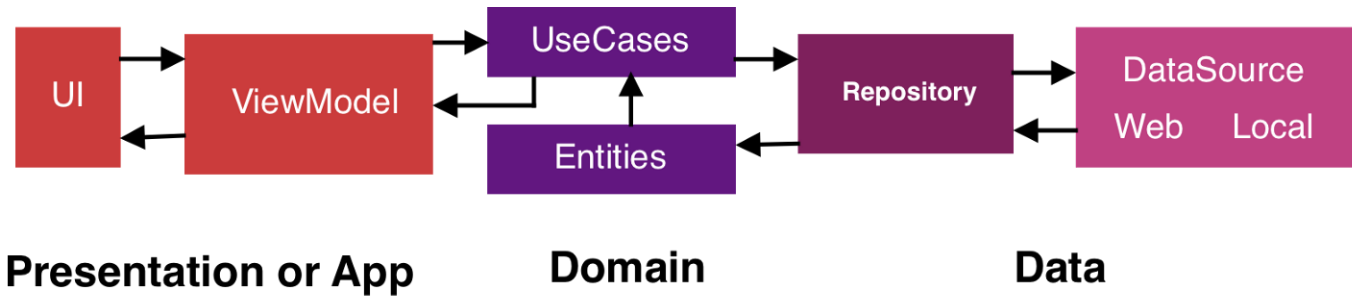
However, it would be a good practice to map the server model to the app model, where the app model contains only the data required to be consumed nothing else. Do not get confused POJO(Plain Old Java Object) or Data Model or Object class or Entity refers to the same term.

. . .

Data Flow

Let's get started with Data flow. If a user event is triggered in UI then we communicate it with **ViewModel** or **presenter** to take necessary action. Then ViewModel connects with the **use case** to get the result for the action.

The use case then interacts with the **repository** class to get the solution from appropriate **data sources** like network or database or preference. Following is the image of data flow we discussed



Before proceeding further into sample its best to have an idea of following frameworks because the example is completely based on these frameworks

- **Dagger 2** -A fully static, compile-time dependency injection framework
- RxJava 2 -is a library for asynchronous and event-based programs using observable sequences.
- **Retrofit 2** -A type-safe HTTP client for Android for Network calls
- **ViewModel** -is a class that is responsible for preparing and managing the data for an Activity or a Fragment.

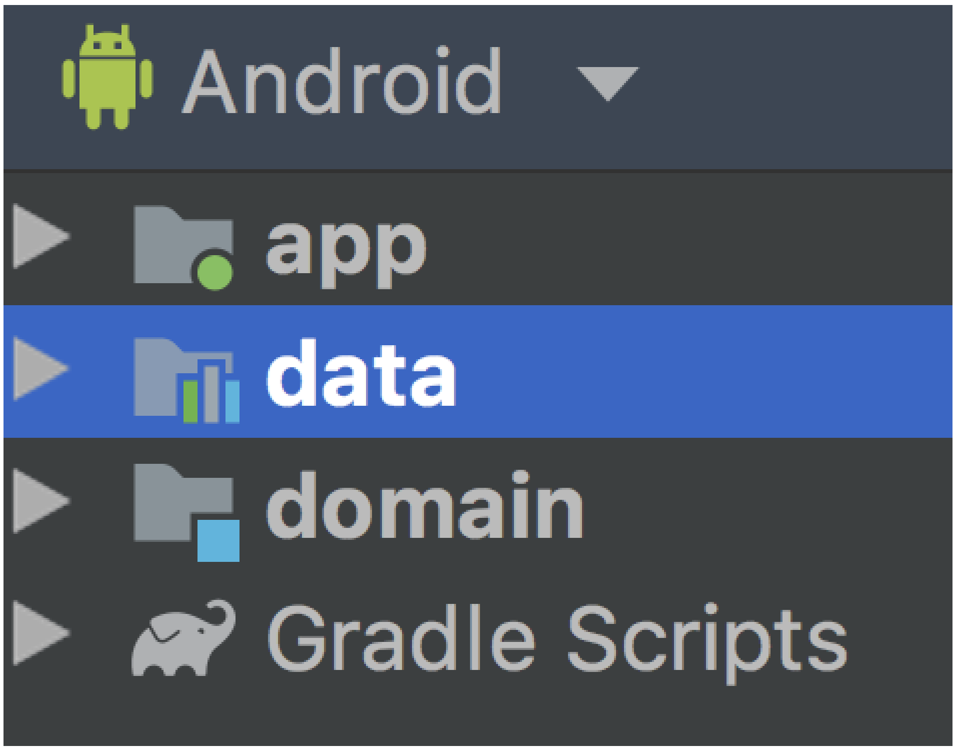
- **Kotlin** -Kotlin is a cross-platform, statically typed, general-purpose programming language with type inference

. . .

Example

Let's practice Clean Architecture with an example of making a network call to fetch share details on the click of a button.

To start, we need to create three modules: **app**, **data**, and **domain**. The **b** module is a complete Java, there will be no Android-related things. So we can create a Java Library module for the domain. The folder structure will be looking as below.

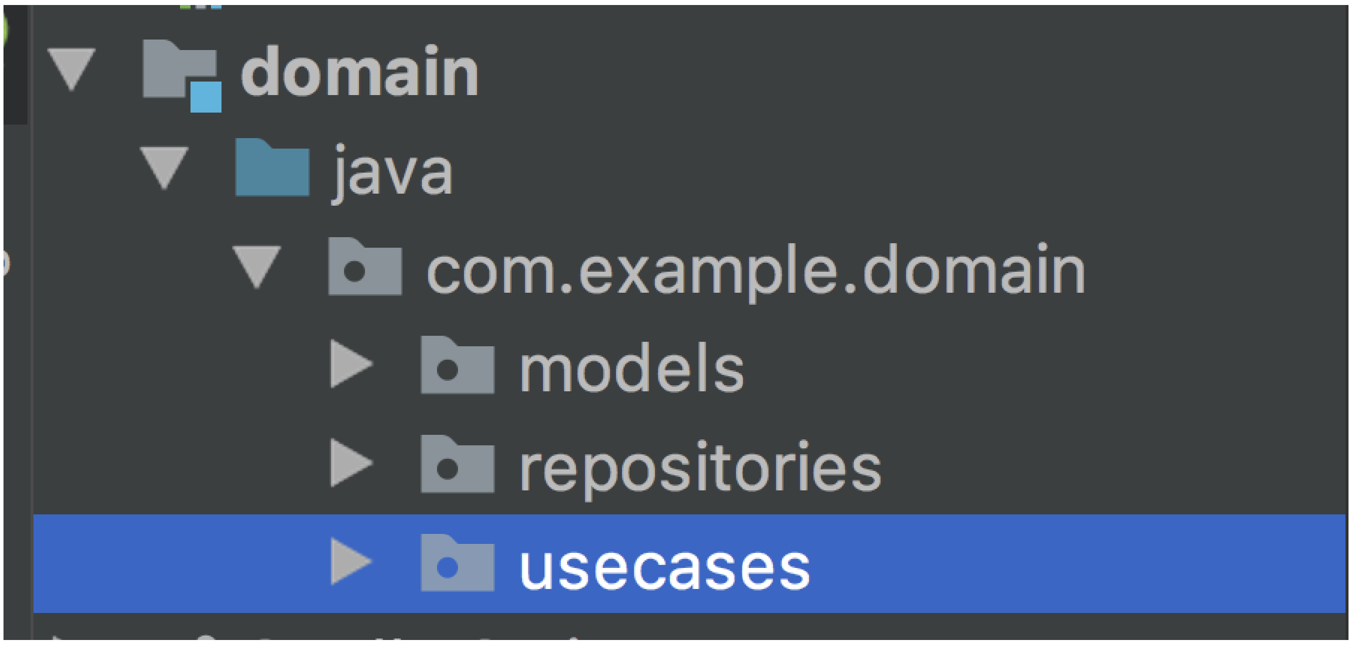


Here domain is included in all the modules and data is included in the app. The approach is bottom-up, which means we will start with domain and move to the app

. . .

Domain Layer

As we know the domain layer consists of models, use cases, repositories create separate directories for each of them. The domain layer would like below



Now let us first start creating model objects or entities or POJO classes in the models' directory.

```kotlin
package com.example.domain.models

data class ShareDetailsModel (
    var shareMessage: String? = null,
    var shareUrl:String ?=null,
    var source: String
)
```

ShareDetailsModel.kt hosted with ❤ by GitHub view raw

Now Let's create Repository, nothing but an interface with name RemoteRepo

```kotlin
package com.example.domain.repositories

import com.example.domain.models.ShareDetailsModel
import io.reactivex.Single

interface RemoteRepo {

    fun getShareDetails(): Single<ShareDetailsModel>
}
```

Repo.kt hosted with ❤ by GitHub view raw

Let's create a use case to fetch data from the data source. As it was a common procedure to remove the redundant code which needs to be used at multiple places