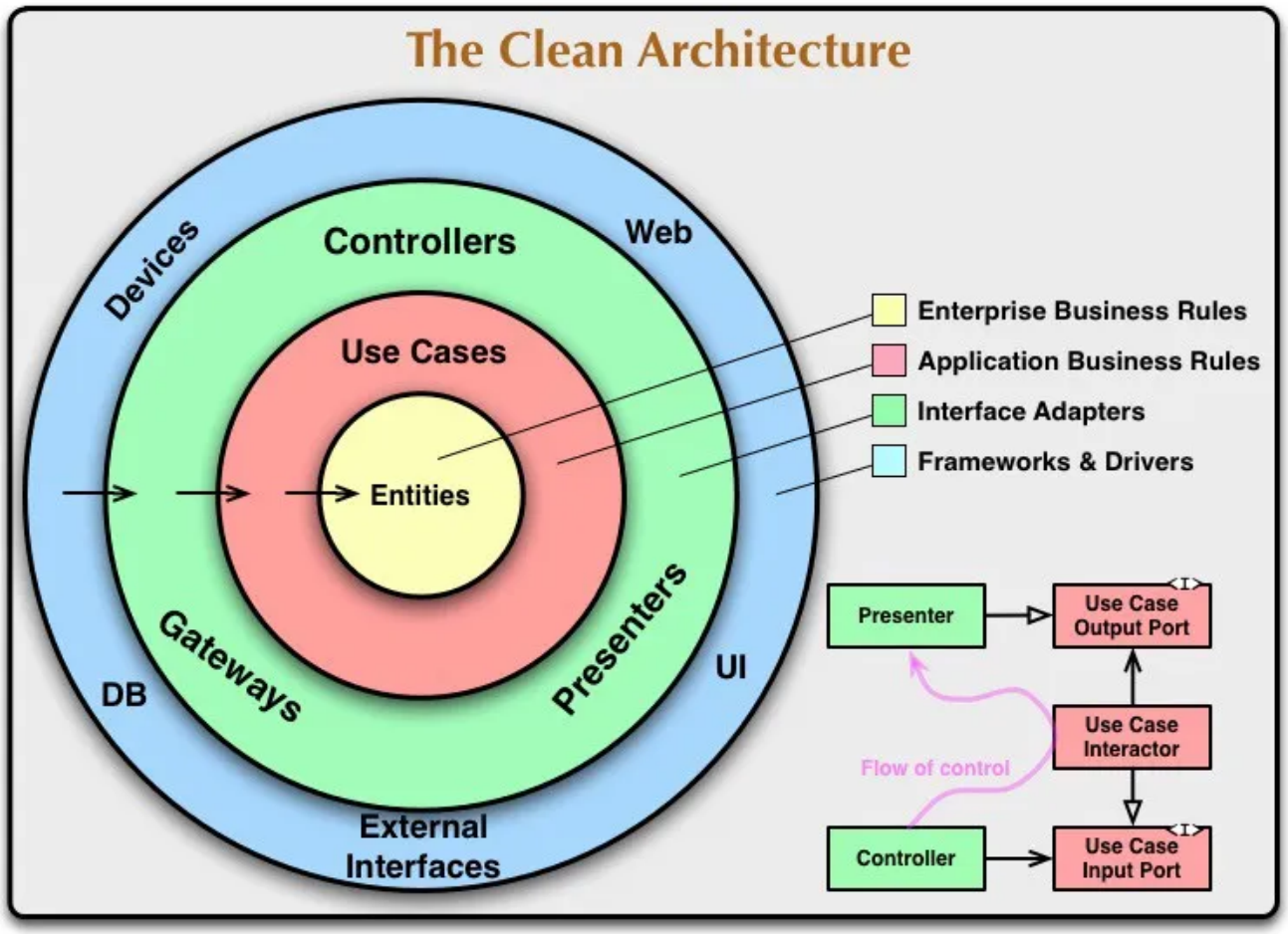
Member-only story

Detailed Guide on Android Clean Architecture

Best way to write Android Apps

Satya Pavan Kantamani · Follow
Published in Programming Geeks
7 min read · Dec 5, 2019

Listen Share More



Every developer starts from scratch with eager to learn and write fast, many of us lack the idea of writing clean code. At the initial stage, we fight to learn fast and make things look better without knowing their results in the future. I was not exceptional I to had done the same at the initial stage of my career as an Android Developer.

But after spending years on development I understood that getting results by doing something is not a good way of development. It took me years to understand the importance of **architecture** in development. However, if you are smart it may take less time for you.

. . .

Is there a need that we should fallow architecture for an Android App development?

The answer would be obviously **YES** because fallowing good architecture helps in many ways like reducing the development time, easy understanding, low maintenance, etc.

Writing clean and quality code for applications requires effort and experience. An application should not only meet the UI and UX requirements but also should be easy to understand, flexible, testable, and maintainable.

Previously there is no much importance of architectures for app development but the scenario has changed drastically. Because as the application grows the complexity also increases. Most of the apps out there are following one or the other architectures.

There are many architectures out there like **MVC**, **MVP**, **MVVM, MVI,** etc extending with **clean code.** They may sound odd but once you know the inner thing that would be very easy to understand and follow. There are many resources out there explaining about MVC, MVP, MVVM, etc.

. . .

Robert C. Martin, also known as **Uncle Bob,** came up with the Clean Architecture concept in the year 2012. Checkout clean code blog for better understanding. In this

article, we will review the concept of **Clean Architecture** in Android applications and an example.

Why Clean Architecture?

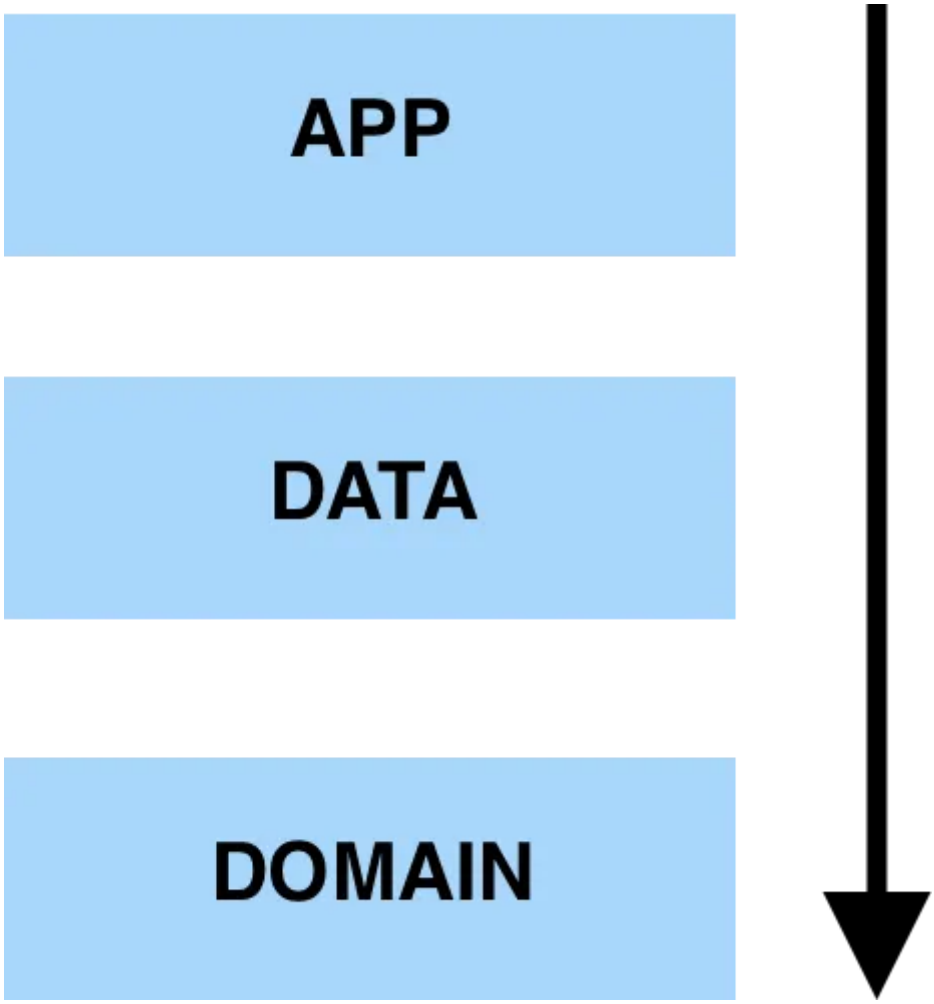
1. **Separation of Concerns** — Separation of code in different modules or sections with specific responsibilities making it easier for maintenance and further modification.
2. **Loose coupling —** flexible code anything can be easily be changed without changing the system
3. **Easily Testable**

. . .

Layers of Clean Architecture

Clean architecture is also referred to as **Onion** architecture as it has different layers. As per our requirement, we need to define the layers, however, architecture doesn't specify the number of layers.

For a basic idea, let us consider there are three layers for now

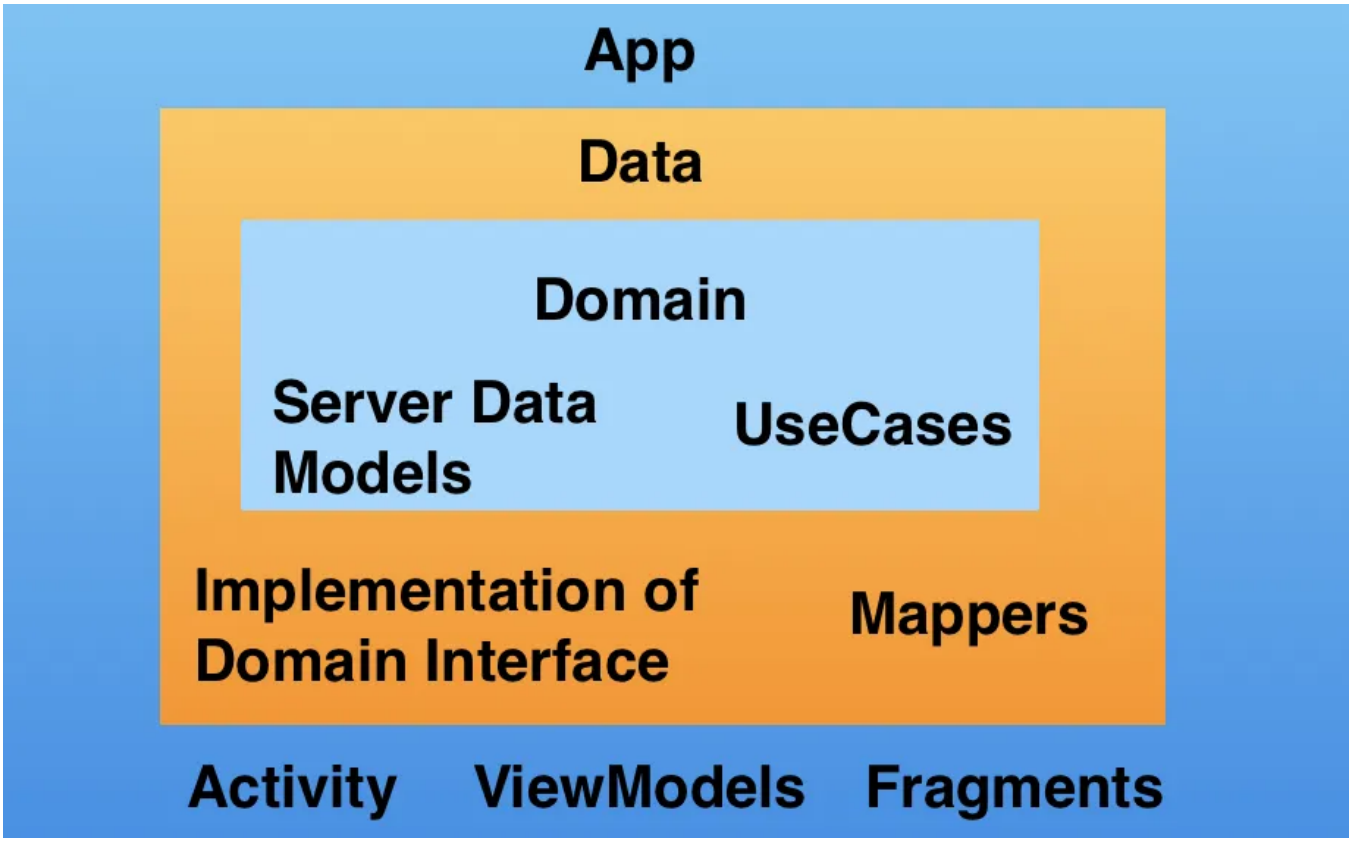


- **Presentation or APP:** A layer that interacts with the **UI,** mainly Android Stuff like Activities, Fragments, ViewModel, etc. It would include both **domain** and **data** layers.
- **Domain**: Contains the **business logic** of the application. It is the **individual** and **innermost** module. It's a complete java module.
- **Data**: It includes the **domain** layer**.** It would implement the **interface** exposed by **domain** layer and **dispenses** data to **app**

. . .

How it applies to Android?

In Android App, we need to have different modules to make things work out of the box. Let's have a look at how the clean architecture applies to Android with the different section as below



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.

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.

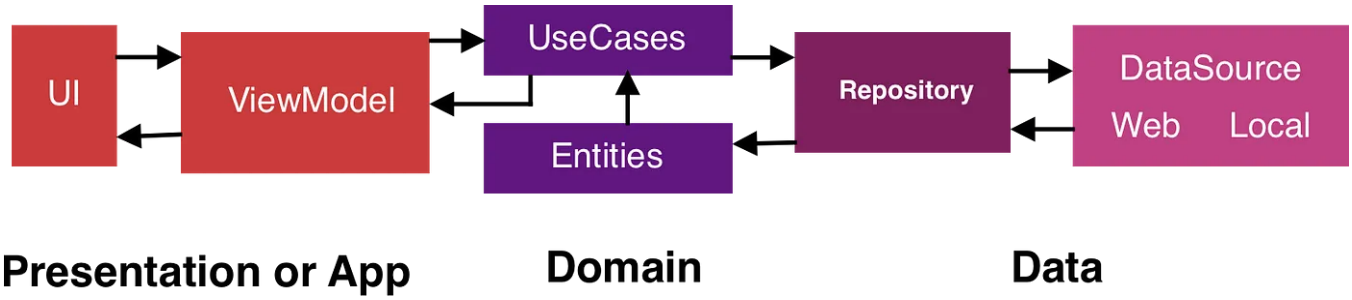
However, it would be a good practice to map the server model to the app model, where the app model contains only the data required to be consumed nothing else. Do not get confused POJO(Plain Old Java Object) or Data Model or Object class or Entity refers to the same term.

. . .

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.

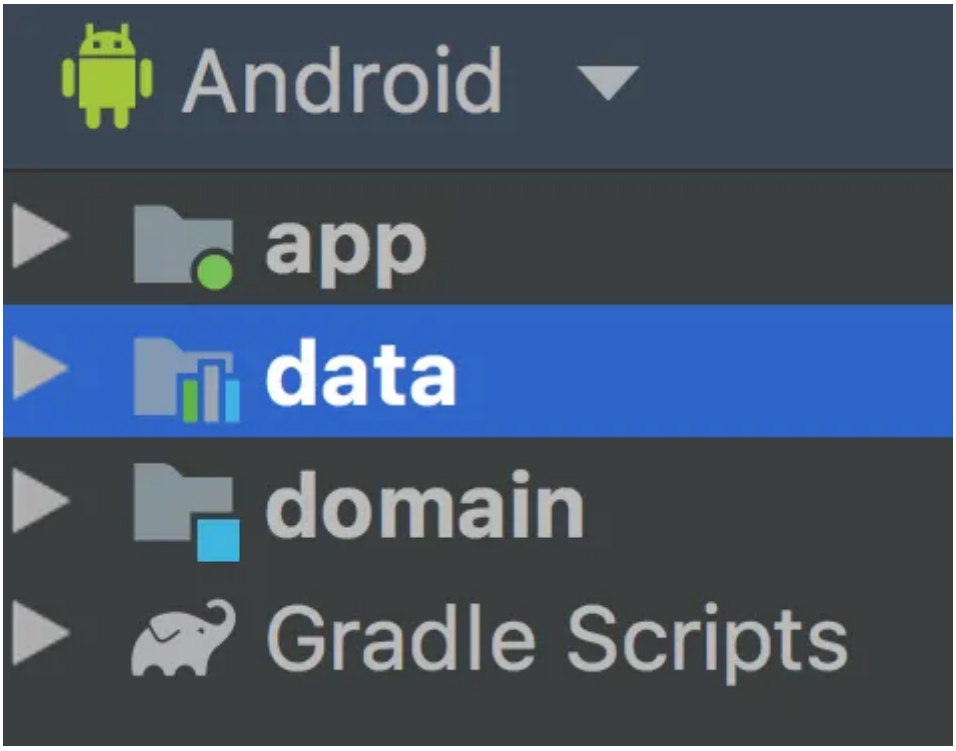
- **Kotlin** -Kotlin is a cross-platform, statically typed, general-purpose programming language with type inference

. . .

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.

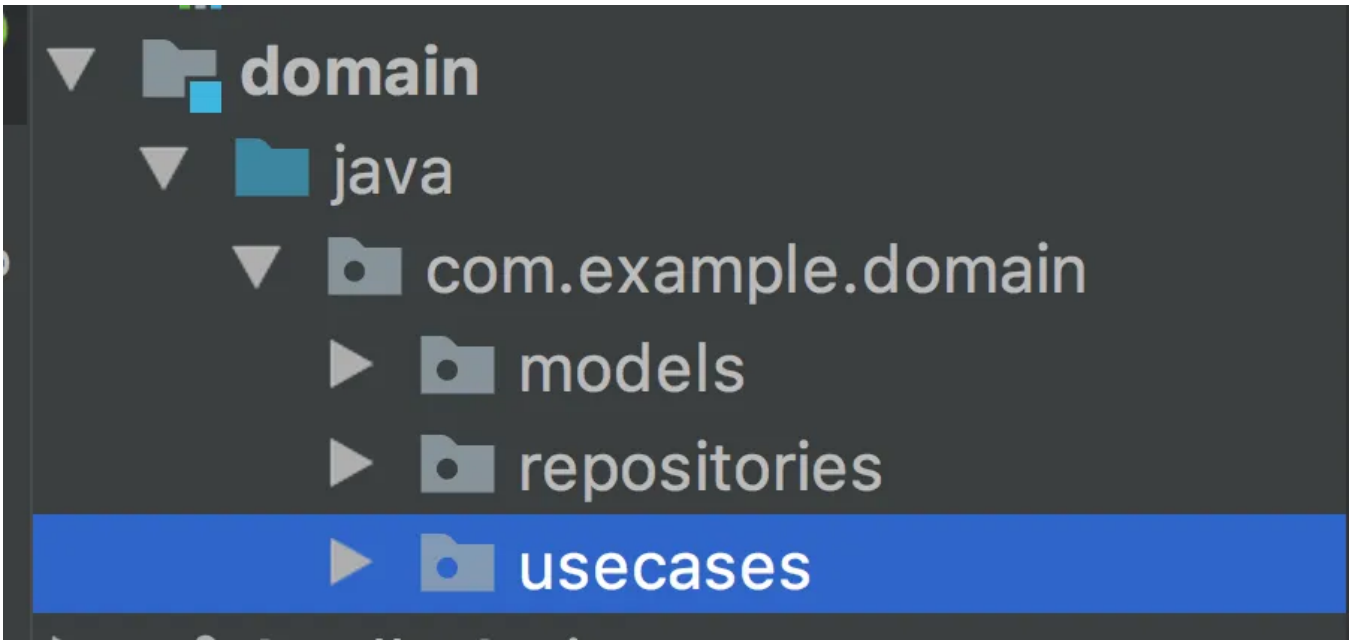


Here domain is included in all the modules and data is included in the app. The approach is bottom-up, which means we will start with domain and move to the app

. . .

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

firstly, we create a base class for use cases for easy extension and usage in future

```
package  com.example.domain.usecases

import io.reactivex.Single


interface SingleUseCase<R> {
    fun execute(): Single<R>
}
```

SingleUseCase.kt hosted with ❤ by GitHub view raw

Later extend this class where ever required. We need to inject the repository class using the Dagger framework and call the repo method to get data from the destination.

```
package com.example.domain.usecases

import com.example.domain.models.ShareDetailsModel
import com.example.domain.repositories.RemoteRepo
import io.reactivex.Single
import javax.inject.Inject

class GetShareDetailsUseCase @Inject constructor(val apiRepo: RemoteRepo):
    SingleUseCase<ShareDetailsModel> {

    override fun execute(): Single<ShareDetailsModel> {
        return apiRepo.getShareDetails()

    }
}
```

GetShareDetailsUseCase.kt hosted with ❤ by GitHub view raw

For reference, have a look at the Gradle file of the domain module

```
apply plugin: 'java-library'

apply plugin: 'kotlin'

apply plugin: 'kotlin-kapt'

repositories {
    mavenCentral()
    maven { url 'https://jitpack.io' }
}


dependencies {
implementation fileTree(dir: 'libs', include: ['*.jar'])
api "io.reactivex.rxjava2:rxkotlin:2.3.0"
api "io.reactivex.rxjava2:rxjava:2.2.12"
api 'io.reactivex.rxjava2:rxandroid:2.1.1'
api "com.jakewharton.rxbinding2:rxbinding-kotlin:2.2.0"
api "com.google.code.gson:gson:2.8.5"
api "com.google.dagger:dagger:2.25.2"
api "com.google.dagger:dagger-android-support:2.25.2"
}

sourceCompatibility = "7"
targetCompatibility = "7"
```

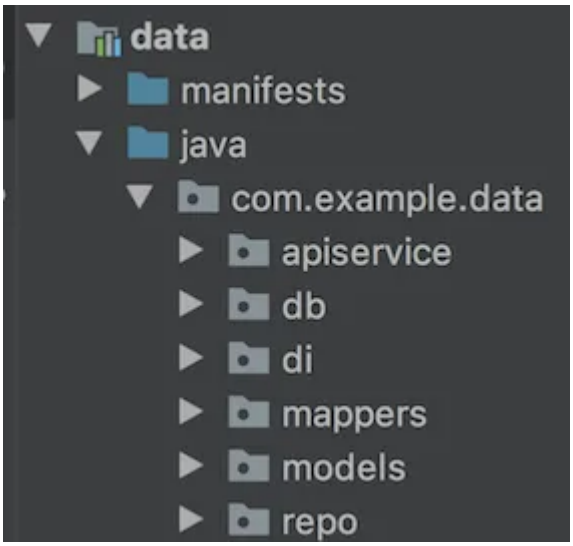
CleanDomainGradle.gradle hosted with ❤️ by GitHub view raw

As we created everything required at the domain layer we will move to the data layer.

. . .

Data Layer

The data module consists of server models, mappers, API services, and repo implementations. The Data module directory would look like below



Initially create the class ShareDataModel to read data from the server response.

```kotlin
package com.example.data.models

import com.google.gson.annotations.SerializedName

data class ShareDataModel(
    @SerializedName("status_code")
    val statusCode: Int = -1,
    @SerializedName("message")
    val shareMessage: String? = null,
    @SerializedName("source")
    val source: String? = null,
    @SerializedName("share_url")
    val shareUrl: String? = null,
    @SerializedName("campaign")
    val campaign: String? = null,
    @SerializedName("user_id")
    val userdId: String? = null
)
```

ShareDataModel.kt hosted with ❤️ by GitHub view raw

Secondly, let's create ApiService

```
package com.example.data.apiservice

import com.example.data.models.ShareDataModel
import io.reactivex.Single
import retrofit2.http.GET


interface ApiService {

    @GET("/share/")
    fun getSharingDetails(): Single<ShareDataModel>

}
```

ApiService.kt hosted with ❤ by GitHub view raw

Now we need a mapper class that maps the server data to the data model required by the app.

```
package com.example.data.mappers

import com.example.data.models.ShareDataModel
import com.example.domain.models.ShareDetailsModel
import javax.inject.Inject


/**
 * A mapper to map the ShareDataModel from server to ShareDetailsModel in a presentable
 */
class ShareMappper @Inject constructor() {

    fun toShareDetails(shareModelServer: ShareDataModel): ShareDetailsModel {
        return ShareDetailsModel(
            shareModelServer.shareMessage ?: "",
            shareModelServer.shareUrl ?: "",
            shareModelServer.source ?: ""
        )
    }
}
```

ShareMappper.kt hosted with ❤ by GitHub view raw

Now we need to create a Repository Implementation class for the repository interface in the domain module which handles the data fetching

```kotlin
package com.example.data.repo

import com.example.data.apiservice.ApiService
import com.example.data.mappers.ShareMappper
import com.example.domain.models.ShareDetailsModel
import com.example.domain.repositories.RemoteRepo
import io.reactivex.Single
import javax.inject.Inject

class RemoteRepoImpl @Inject constructor(
    private val apiService: ApiService,
    private val shareMapper: dagger.Lazy<ShareMappper>) : RemoteRepo {

    override fun getShareDetails(): Single<ShareDetailsModel> {
        return  apiService.getSharingDetails()
            .map {
                shareMapper.get().toShareDetails(it)
            }
    }
}
```

RemoteRepoImpl.kt hosted with ❤️ by GitHub view raw

Finally, lets set up the **di**(Dependency Injection) part. There will be two classes one **NetworkModule** which contains provide methods for retrofit, Okhttp, and second **ApiModule** which has the API service endpoint to fetch the data.

NetworkModule class

```kotlin
package com.example.data.di

import com.google.gson.Gson
import com.google.gson.GsonBuilder
import dagger.Module
import dagger.Provides
import okhttp3.OkHttpClient
import okhttp3.logging.HttpLoggingInterceptor
import retrofit2.Retrofit
import retrofit2.adapter.rxjava2.RxJava2CallAdapterFactory
import retrofit2.converter.gson.GsonConverterFactory
import javax.inject.Named
import javax.inject.Singleton

@Module
class NetworkModule {

    @Provides
    fun provideOkHttpClient(): OkHttpClient {
        val okHttpBuilder = OkHttpClient.Builder()
        okHttpBuilder.addInterceptor(HttpLoggingInterceptor())
        return okHttpBuilder.build()
    }

    @Singleton
    @Provides
    fun provideGson(): Gson {
        return GsonBuilder()
            .setLenient()
            .create()
    }

    @Provides
    @Named("auth_retrofit")
    fun provideRetrofit(okHttpClient: OkHttpClient): Retrofit {
        return Retrofit.Builder()
            .baseUrl("www.androidapps.com")
            .addConverterFactory(GsonConverterFactory.create())
            .addCallAdapterFactory(RxJava2CallAdapterFactory.create())
            .client(okHttpClient)
            .build()
    }

}
```

NetwrokModule.gradle hosted with ❤️ by GitHub view raw

ApiModule class

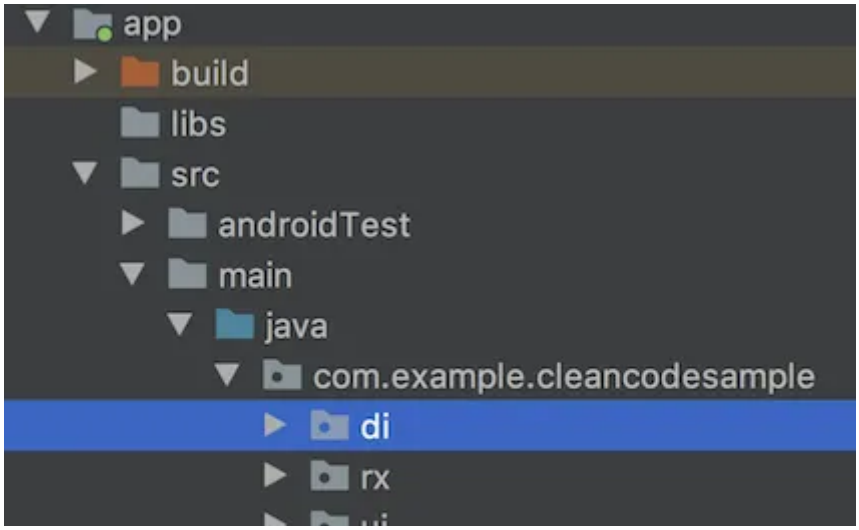
```kotlin
package com.example.data.di

import com.example.data.apiservice.ApiService
import dagger.Module
import dagger.Provides
import retrofit2.Retrofit

@Module(includes = [NetworkModule::class])
class ApiModule {

    @Provides
    fun bindApiService(retrofit: Retrofit): ApiService {
        return retrofit.create(ApiService::class.java)
    }

}
```

ApiModule.kt hosted with ❤️ by GitHub view raw

That's it we are done with the data module now its time to move back to the app or presentation layer.

App or Presentation Layer

The directory structure would be as below.



Lets set up the base di module

The di directory has the ActivityBindingModule, AppComponent, AppModule, ViewModelFactory, ViewmodelKey, ViewModelModule classes. If you are familiar with dagger2 you would understand the importance of each class.

ViewModelFactory

```kotlin
package com.example.cleancodesample.di

import androidx.fragment.app.FragmentActivity
import androidx.lifecycle.ViewModel
import androidx.lifecycle.ViewModelProvider
import androidx.lifecycle.ViewModelProviders
import javax.inject.Inject
import javax.inject.Provider
import javax.inject.Singleton

@Singleton
class ViewModelFactory @Inject constructor(private val viewModels: MutableMap<Class<out
        @JvmSuppressWildcards Provider<ViewModel>>): ViewModelProvider.Factory {
    override fun <T : ViewModel?> create(modelClass: Class<T>): T =
        viewModels[modelClass]?.get() as T
}

inline fun <reified VM: ViewModel> ViewModelProvider.Factory.obtainViewModel(activity:
    return ViewModelProviders.of(activity, this)[VM::class.java]
}
```

ViewModelFactory.kt hosted with ❤️ by GitHub view raw

ViewModelKey

```kotlin
package com.example.cleancodesample.di

import androidx.lifecycle.ViewModel
import dagger.MapKey
import kotlin.reflect.KClass

@Target(AnnotationTarget.FUNCTION, AnnotationTarget.PROPERTY_GETTER, AnnotationTarget.P
@kotlin.annotation.Retention(AnnotationRetention.RUNTIME)
@MapKey
internal annotation class ViewModelKey(val value: KClass<out ViewModel>)
```

ViewModelKey.kt hosted with ❤️ by GitHub view raw

ViewModelModule: Here we declare all our view models present in the app so that dagger takes care of the rest of the things

```kotlin
package com.example.cleancodesample.di

import androidx.lifecycle.ViewModel
import androidx.lifecycle.ViewModelProvider
import com.example.cleancodesample.viewmodel.MainViewModel
import dagger.Binds
import dagger.Module
import dagger.multibindings.IntoMap

@Module
abstract class ViewModelModule {

    @Binds
    abstract fun bindViewModelFactory(factory: ViewModelFactory): ViewModelProvider.Fac

    @Binds
    @IntoMap
    @ViewModelKey(MainViewModel::class)
    abstract fun bindMainViewModel(viewModel: MainViewModel): ViewModel

}
```

ViewModelModule.kt hosted with ❤️ by GitHub view raw

ActivityBindingModule: Need to define all the Views related stuff like Activities and Fragments so that dagger can inject things at those places.

```kotlin
package com.example.cleancodesample.di

import com.example.cleancodesample.ui.MainActivity
import dagger.Module
import dagger.android.ContributesAndroidInjector

@Module(includes = [ViewModelModule::class])
abstract class ActivityBindingModule {

    @ContributesAndroidInjector
    abstract fun bindMainScreenActivity(): MainActivity

}
```

ActivityBindingModule.kt hosted with ❤️ by GitHub view raw

AppModule

```
package com.example.cleancodesample.di

import android.app.Application
import android.content.Context
import com.example.cleancodesample.rx.SchedulersFacade
import com.example.cleancodesample.rx.SchedulersProvider
import dagger.Binds
import dagger.Module

@Module
abstract class AppModule {

    @Binds
    abstract fun bindContext(application: Application): Context

    @Binds
    abstract fun providerScheduler(schedulersFacade: SchedulersFacade): SchedulersProvi
}
```

AppModule.kt hosted with ❤️ by GitHub view raw

AppComponent

```
package com.example.cleancodesample.di

import android.app.Application
import com.example.cleancodesample.CleanCodeSampleApplication
import com.example.data.di.ApiModule
import com.example.data.di.NetworkModule
import dagger.BindsInstance
import dagger.Component
import dagger.android.AndroidInjectionModule
import dagger.android.AndroidInjector
import dagger.android.support.AndroidSupportInjectionModule
import javax.inject.Singleton

@Singleton
@Component(
    modules = [
        NetworkModule::class,
        ApiModule::class,
        AppModule::class,
        AndroidSupportInjectionModule::class,
        AndroidInjectionModule::class,
        ActivityBindingModule::class]
)
interface AppComponent : AndroidInjector<CleanCodeSampleApplication> {

    @Component.Builder
    interface Builder {

        @BindsInstance
        fun application(application: Application): Builder

        fun build(): AppComponent
    }
}
```

AppComponent.kt hosted with ❤️ by GitHub view raw

For Rx related stuff we have SchedulersProvider Interface and SchedulersFacade class which implements that interface

SchedulerProvider

```
package com.example.cleancodesample.rx

import io.reactivex.Scheduler
```

```
interface SchedulersProvider {
    fun ui(): Scheduler
    fun io(): Scheduler
    fun computation(): Scheduler
}
```

SchedulersFacade

```
package com.example.cleancodesample.rx

import io.reactivex.Scheduler
import io.reactivex.android.schedulers.AndroidSchedulers
import io.reactivex.schedulers.Schedulers
import javax.inject.Inject
import javax.inject.Singleton

@Singleton
class SchedulersFacade @Inject constructor() : SchedulersProvider {

    override fun io(): Scheduler {
        return Schedulers.io()
    }

    override fun computation(): Scheduler {
        return Schedulers.computation()
    }

    override fun ui(): Scheduler {
        return AndroidSchedulers.mainThread()
    }
}
```

SchedulersFacade.kt hosted with ❤️ by GitHub view raw

. . .

Now let's move to our UI part that is regarding activity and ViewModel

XML

```xml
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.androi
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".ui.MainActivity">


    <Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Button"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintLeft_toLeftOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

sample_layout.xml hosted with ❤️ by GitHub view raw

MainViewModel

```
package com.example.cleancodesample.viewmodel

import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.ViewModel
import com.example.cleancodesample.rx.SchedulersProvider
import com.example.domain.models.ShareDetailsModel
import com.example.domain.usecases.GetShareDetailsUseCase
import io.reactivex.disposables.CompositeDisposable
import javax.inject.Inject

class MainViewModel @Inject constructor(
    val shareUseCase: GetShareDetailsUseCase,
    val schedulers: SchedulersProvider) : ViewModel() {

    val shareLiveData = MutableLiveData<ShareDetailsModel>()
    protected val compositeDisposable = CompositeDisposable()

    fun getShareData() {
        shareUseCase.execute()
            .subscribeOn(schedulers.io())
            .subscribe({
                it?.let {
                    shareLiveData.postValue(it)
                }
            },{

            }).let {
                compositeDisposable.add(it)
            }
    }

    override fun onCleared() {
        compositeDisposable.clear()
    }

}
```

MainViewModel.gradle hosted with ❤️ by GitHub view raw

MainActivity

```
package com.example.cleancodesample.ui

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.widget.Toast
import androidx.lifecycle.Observer
import androidx.lifecycle.ViewModelProvider
import com.example.cleancodesample.R
import com.example.cleancodesample.di.obtainViewModel
import com.example.cleancodesample.viewmodel.MainViewModel
import kotlinx.android.synthetic.main.activity_main.*
import javax.inject.Inject

class MainActivity : AppCompatActivity() {

    @Inject
    lateinit var viewModelFactory: ViewModelProvider.Factory

    private lateinit var viewmodel : MainViewModel
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        viewmodel =   viewModelFactory.obtainViewModel(this)
        button?.setOnClickListener {
            viewmodel.getShareData()
        }
        viewmodel.shareLiveData.observe(this, Observer {
            it?.let {
               Toast.makeText(this,"Data fetched ${it.shareMessage}",Toast.LENGTH_LONG)
            }
        })
    }

}
```

MainActivityClean.gradle hosted with ❤️ by GitHub view raw

Now finally Applications class

```
package com.example.cleancodesample

import android.content.Context
import androidx.multidex.MultiDex
import com.example.cleancodesample.di.AppComponent
import com.example.cleancodesample.di.DaggerAppComponent
import dagger.android.AndroidInjector
import dagger.android.DaggerApplication

class CleanCodeSampleApplication : DaggerApplication() {

    private lateinit var appComponent: AppComponent

    override fun attachBaseContext(base: Context) {
        super.attachBaseContext(base)
        MultiDex.install(this)
    }

    override fun applicationInjector(): AndroidInjector<out DaggerApplication> {
        appComponent = DaggerAppComponent.builder()
            .application(this)
            .build()
        return appComponent
    }
}
```

Application.gradle hosted with ❤️ by GitHub view raw

Many of us have different views on placing the classes in different directories it's up to you, the above was just a sample of my View on Clean architecture.

. . .

Please let me know your suggestions and comments.

You can find me on **Medium** and **Linkedin**!

Thank you for reading.

Android | Android App Development | Architecture | Programming | Software Development

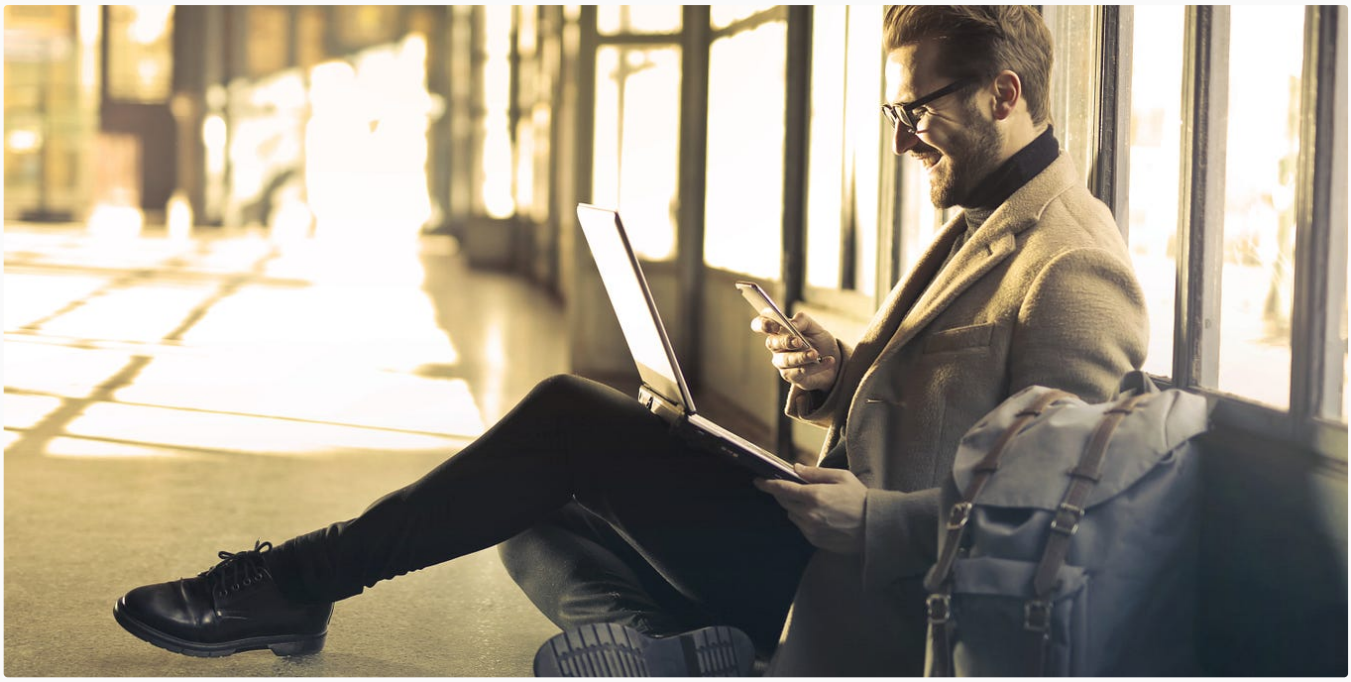
Follow

Written by Satya Pavan Kantamani

2.8K Followers · Writer for Programming Geeks

Android Dev, Interested in Traveling, App development. Based in Hyderabad, India. Catch me at https://about.me/satyapavankumar

More from Satya Pavan Kantamani and Programming Geeks



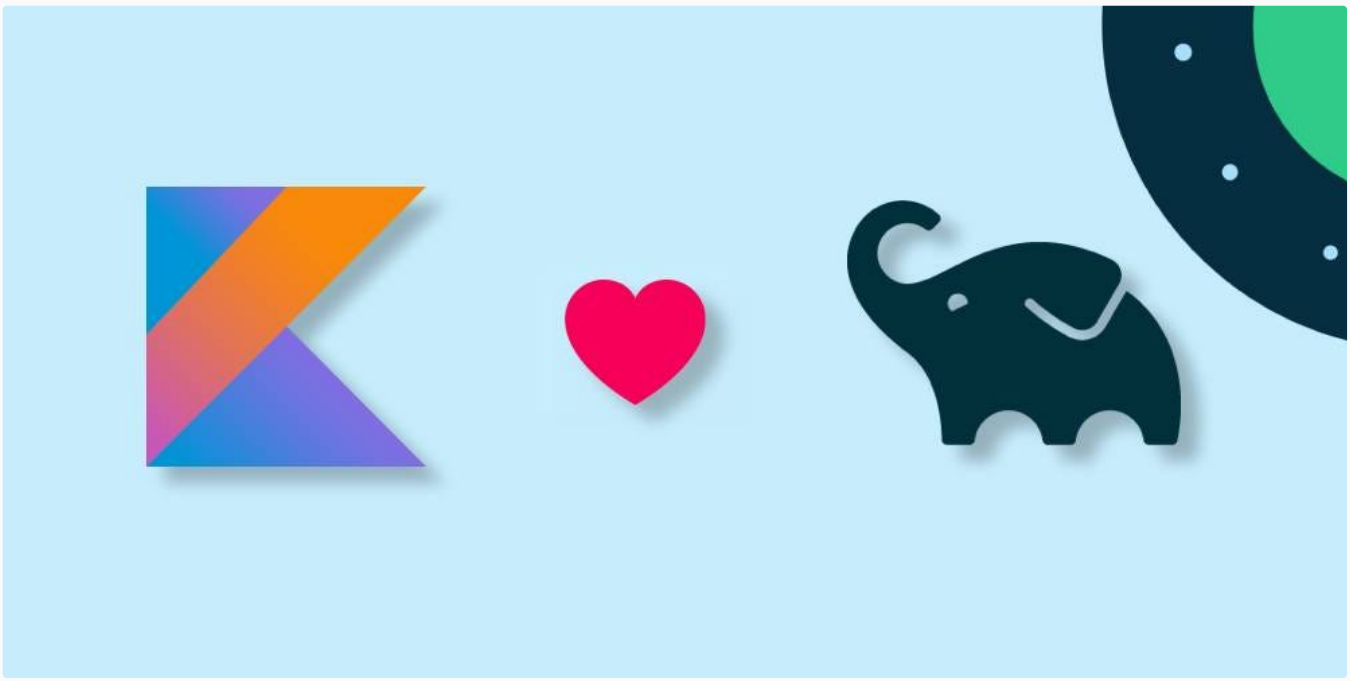
Satya Pavan Kantamani

Use Android Knowledge to build side hustles

The easiest way to reach direct customers or clients to earn additional income

· 5 min read · Jun 2

108 1



Vikas Kumar in Programming Geeks

Kotlin DSL: Gradle scripts in Android made easy

If you are scared of Gradle scripts with unfamiliar groovy syntax files, then Kotlin DSL is made for you. Its time to face the Gradle...

6 min read · Sep 1, 2020

Siva Ganesh Kantamani in Programming Geeks

Integrating Google Drive API in Android Applications

Take away from this article

· 4 min read · Jun 24, 2021

62 4

Satya Pavan Kantamani in Better Programming

How To Implement a Foreground Service in Android

Know what a foreground service is and integrate one into your Android app

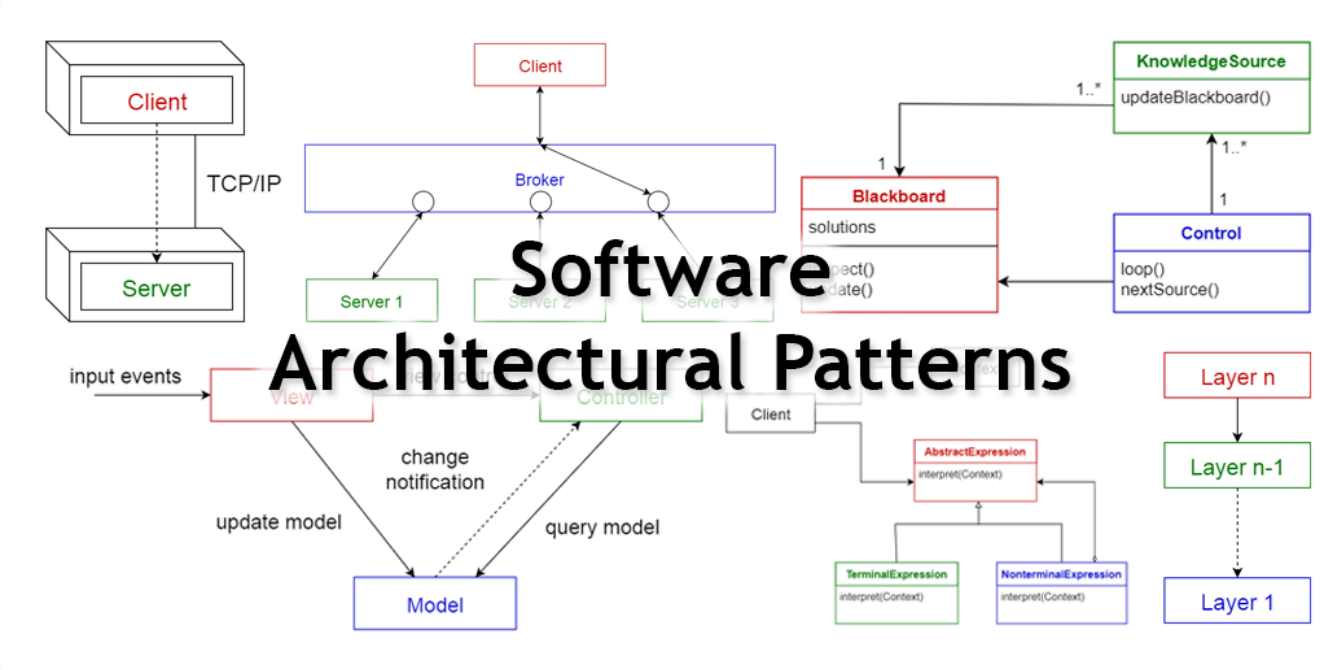
· 5 min read · Jun 30, 2021

222 4

See all from Satya Pavan Kantamani

See all from Programming Geeks

Recommended from Medium



Vijini Mallawaarachchi in Towards Data Science

10 Common Software Architectural Patterns in a nutshell

Ever wondered how large enterprise scale systems are designed? Before major software development starts, we have to choose a suitable...

✨ · 5 min read · Sep 4, 2017

38K 126

Barros

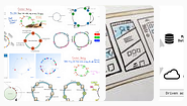
Android: Single Source Of Truth Strategy (Offline-First)

As suggested by the name, with single source of truth strategy we are defining a place where to find and store our data, being sure that...

· 4 min read · Feb 15

15

Lists

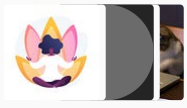


General Coding Knowledge

20 stories · 61 saves

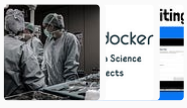
It's never too late or early to start something

10 stories · 21 saves



Stories to Help You Grow as a Software Developer

19 stories · 164 saves



Coding & Development

11 stories · 41 saves

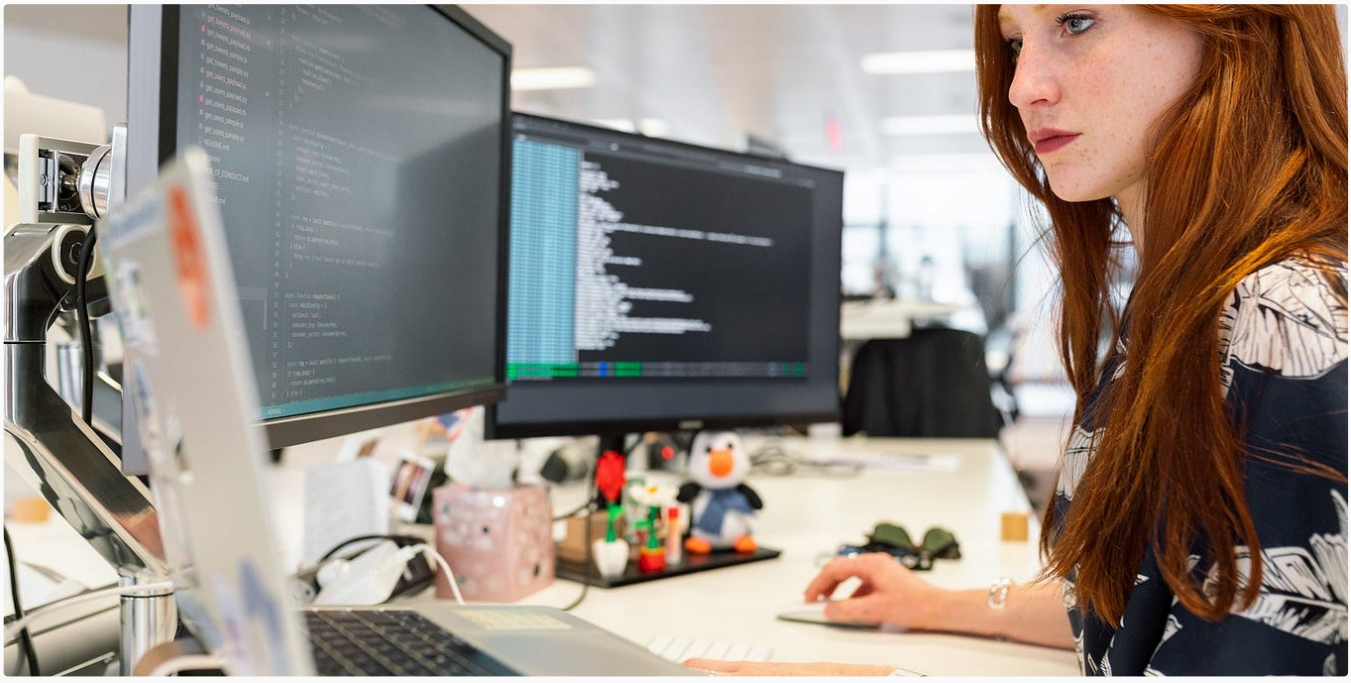
Elye in Mobile App Development Publication

Learning Android's Room Database Made Easy

Android's Room definitely much easier than vanilla SQLite, LoaderManager, and Cursor

· 9 min read · Jan 21

82



The Coding Diaries in The Coding Diaries

Why Experienced Programmers Fail Coding Interviews

A friend of mine recently joined a FAANG company as an engineering manager, and found themselves in the position of recruiting for...

5 min read · Nov 2, 2022

4.8K 108

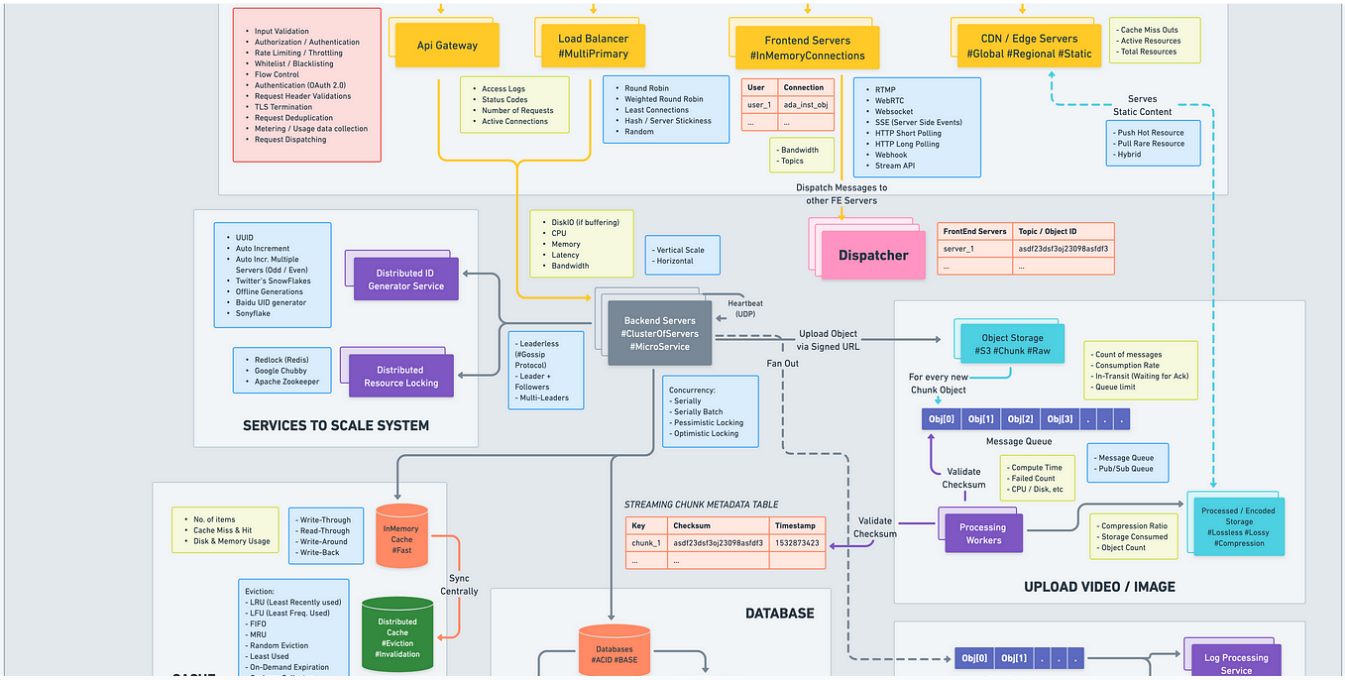
Gerardo Fernández

The SOLID principles

The SOLID principles explained in detail for the development of maintainable and robust applications

9 min read · Jan 10

13



Love Sharma in ByteByteGo System Design Alliance

System Design Blueprint: The Ultimate Guide

Developing a robust, scalable, and efficient system can be daunting. However, understanding the key concepts and components can make the...

· 9 min read · Apr 20

5.6K 47

See more recommendations