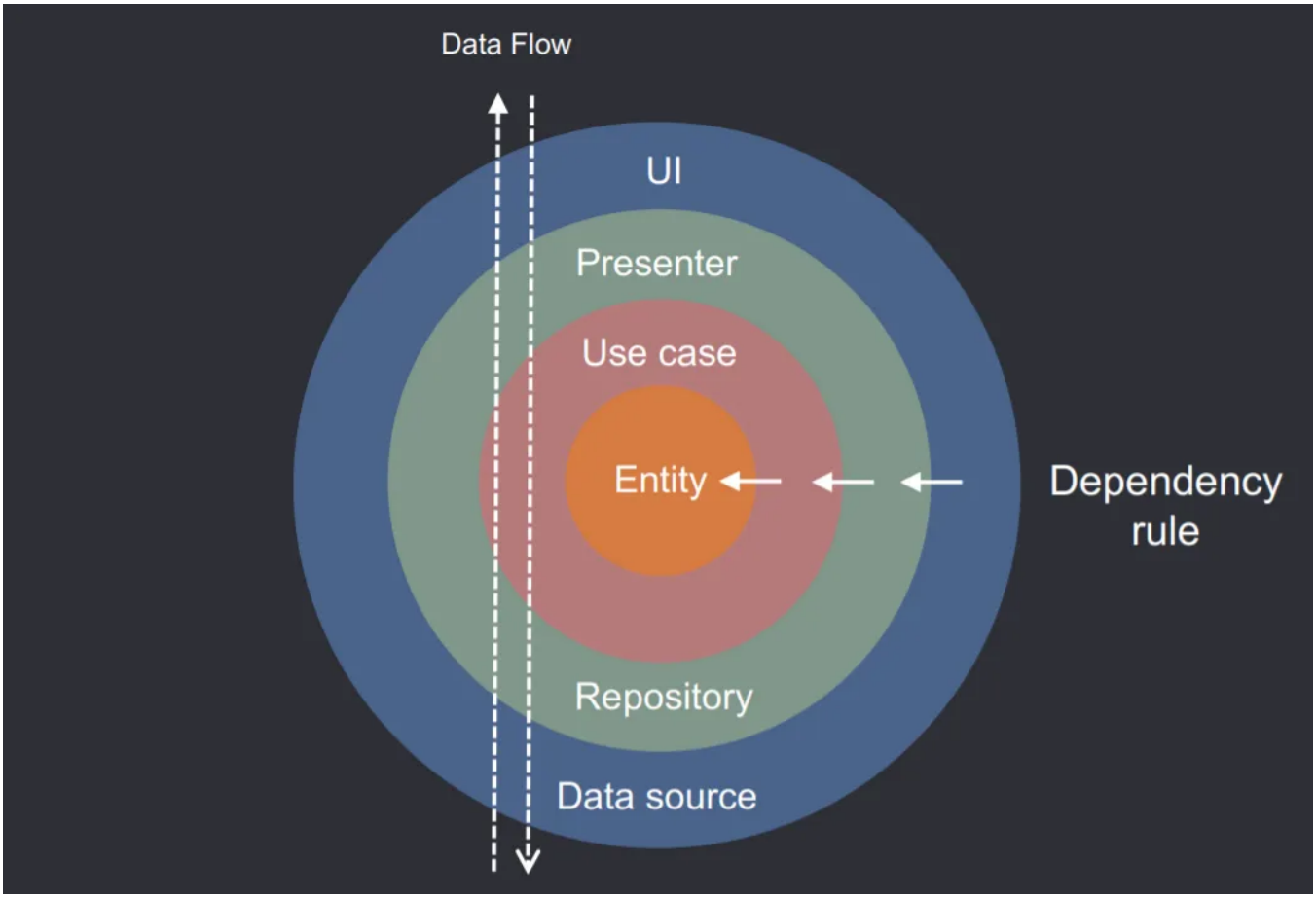
Clean Architecture Guide (with tested examples): Data Flow != Dependency Rule

Mario Sanoguera de Lorenzo · Follow
Published in ProAndroidDev
5 min read · Jul 26, 2018



Difference between Data Flow & Dependency Rule

Hello everyone! 👋 In this story I want to explain Clean Architecture (with tested examples) & talk about the most common mistake when adopting it (the difference between Data Flow and Dependency Rule).

This story is the follow up after my latest medium Intro to App Architecture. Make sure to check it out if you are interested in what makes a good Architecture & why + how to achieve a more maintainable codebase.

Data Flow

Let's start explaining Data Flow in Clean Architecture with an example.

Imagine opening an app that loads a list of posts which contains additional user information. The Data Flow would be:

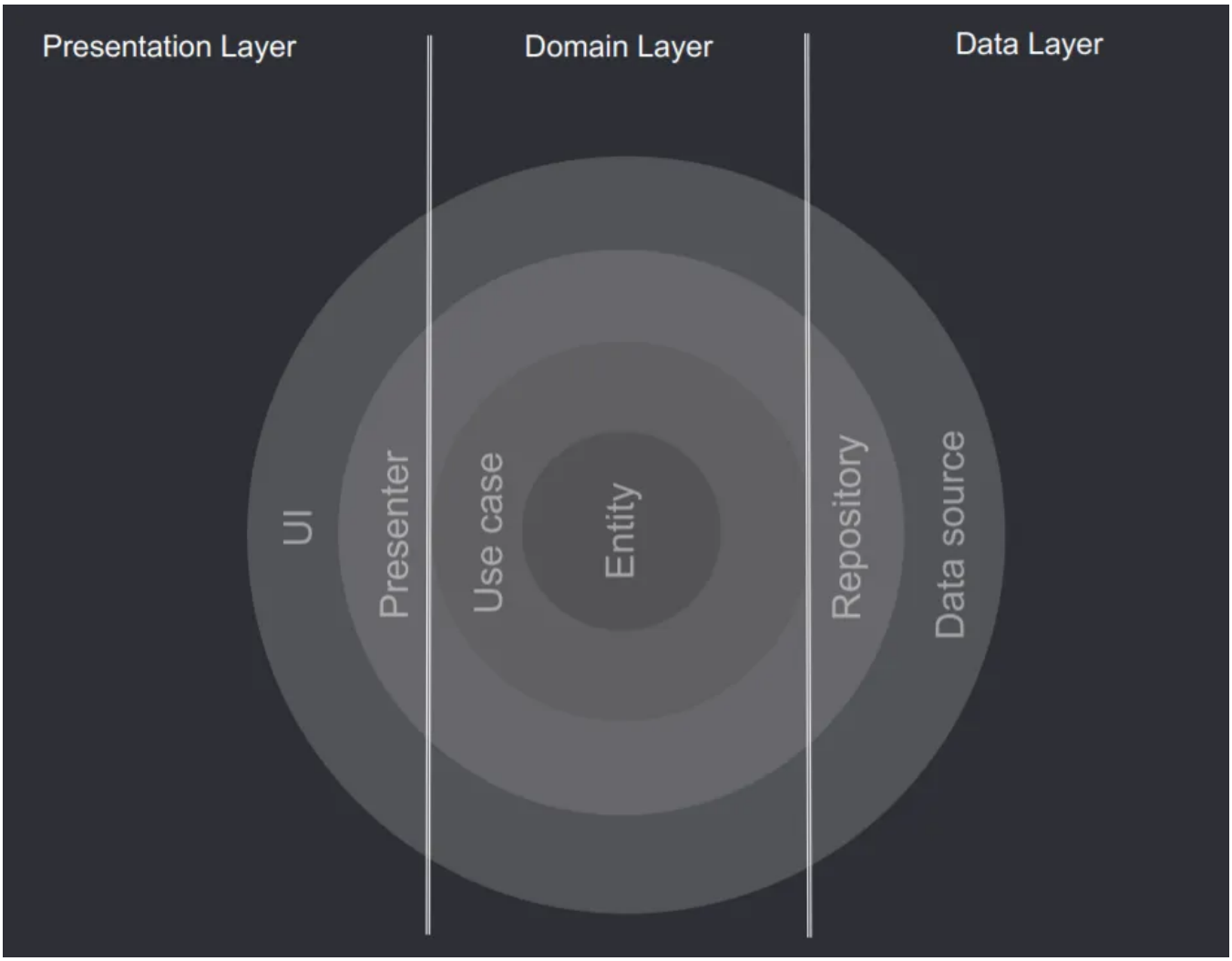
> 1. *UI calls method from Presenter/ViewModel.*
> 2. *Presenter/ViewModel executes Use case.*
> 3. *Use case combines data from User and Post Repositories.*

From the example above we can see how the user action flows from the UI all the way up to the Data Source and then flows back down. This Data Flow is not the same flow as the Dependency Rule.

. . .

Dependency Rule

Dependency Rule is the relationship that exists between the different layers. Before explaining the Dependency Rule in Clean Architecture lets rotate the onion 90 degrees. This helps to point out layers & boundaries. 🆒



Clean Architecture Layers

Let's identify the different layers & boundaries.

Presentation Layer contains *UI (Activities & Fragments)* that are coordinated by *Presenters/ViewModels which execute 1 or multiple Use cases.* Presentation Layer depends on Domain Layer.

Domain Layer is the most INNER part of the onion (no dependencies with other layers) and it contains *Entities, Use cases & Repository Interfaces.* Use cases combine data from 1 or multiple Repository Interfaces.

Data Layer contains *Repository Implementations and 1 or multiple Data Sources.* Repositories are responsible to coordinate data from the different Data Sources. Data Layer depends on Domain Layer.

• • •

Explaining the Domain Layer

> *Domain (with business rules) is the most important Layer.*

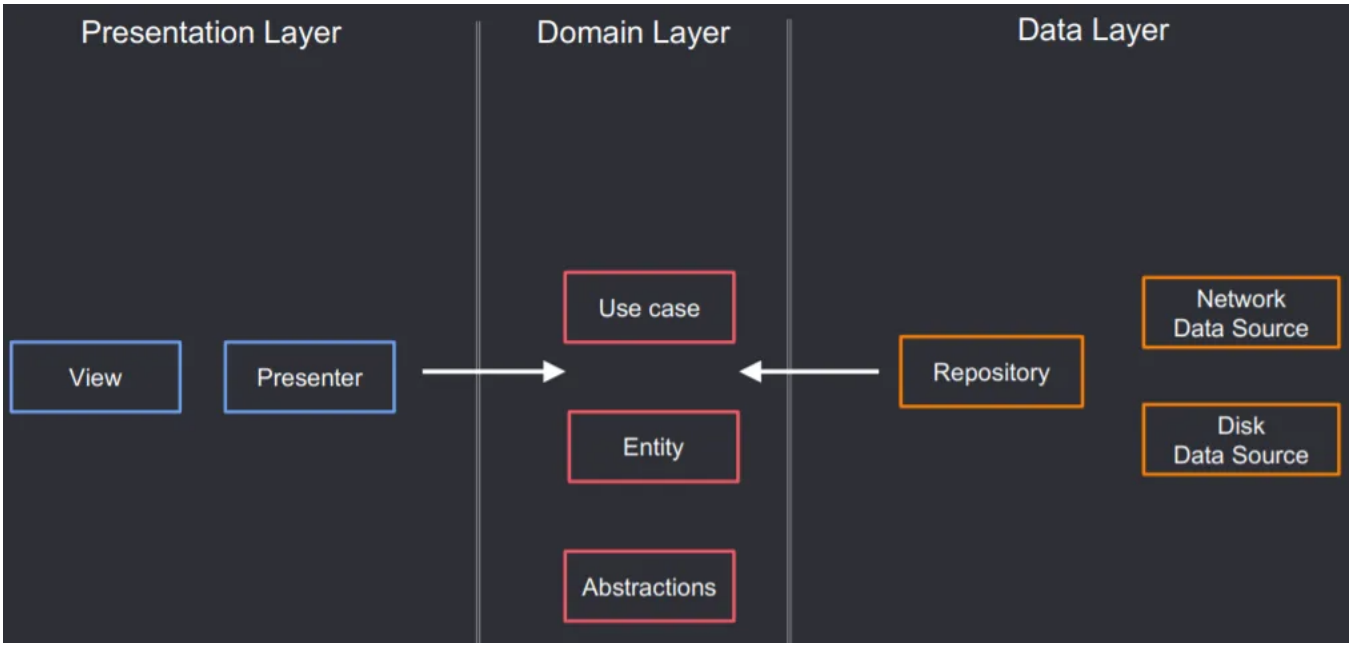
Domain is at the center of the onion which means it is the core of our program. This is one of the main reasons why it shouldn't have any dependencies with other layers.

Presentation and Data Layers are less important since they are only implementations that can be easily replaced. The list of posts could be displayed in Android, iOS, Web or even Terminal if your code is properly decoupled. The same happens with a Database or any kind of Data Source, it can be easily switched.

The outer you go on the onion the most likely things are prone to change. One of the most common mistakes is to have your app driven by your data layer/specific data system. Making it hard to replace or bridge with different data sources down the line.

> *Domain Layer does NOT depend on Data Layer.*

Having modules with the correct dependency rules means that our Domain doesn't have any dependency on any other layer. Due to no dependencies to any Android Library the Domain Layer should be a Kotlin Module. This is an extra boundary that will prevent polluting our most valuable layer with framework related classes. It also promotes reusability across platforms in case we switch over the Framework as our Domain Layer is completely agnostic.



Dependencies between Layers

Explaining the Domain Layer with an example!

What is the real problem that we need to solve?

“Load a list of posts with some user information for each post”

This is the core of our solution no matter where the data comes from or how we present it. This belongs to a Use case inside our **Domain Layer** which is the most inner layer of the architecture (business logic).

For those who haven’t tried Clean Architecture yet Use cases will avoid God Presenters/ViewModels since the Presentation Layer will only execute Use cases and notify the view (Separation of concerns + Single Responsibility Principle). This will also improve the **RUDT** points (**R**ead, **U**pdate, **D**ebug & **T**est) of your project.

```kotlin
data class CombinedUserPost(val user: User, val post: Post)

class UsersPostsUseCase @Inject constructor(
    private val userRepository: UserRepository,
    private val postRepository: PostRepository
) {

    fun get(refresh: Boolean): Single<List<CombinedUserPost>> =
        Single.zip(userRepository.get(refresh), postRepository.get(refresh),
            BiFunction { userList, postList -> map(userList, postList) })
}
```

UsersPostsUseCase.kt hosted with ❤️ by GitHub view raw

Tests here (omitted to make the story shorter).

This Use case is combining data from 2 repositories (UserRepository & PostRepository).

How does Domain NOT depend on Data?

```kotlin
interface PostRepository {

    fun get(refresh: Boolean): Single<List<Post>>
}
```

PostRepository.kt hosted with ❤️ by GitHub view raw

This is because Use cases in Domain are not using the actual implementation of the Repository that sits in the Data Layer. Instead, it is just using an abstraction/interface in the Domain Layer that acts as a contract for any repository who wants to provide the data.

In Clean Architecture it is the responsibility of the Data Layer to have 1 or multiple implementations of the Domain's interfaces and to bind the interface with the actual implementation.

Dagger2 binding example:

```kotlin
@Module
abstract class RepositoryModule {

    @Binds
    abstract fun bindPostRepository(repository: PostRepositoryImpl): PostRepository
}
```

RepositoryModule.kt hosted with ❤️ by GitHub view raw

Koin binding example:

```kotlin
val repositoryModule: Module = module {
    single { PostRepositoryImpl(cache = get(), remote = get()) as PostRepository }
}
```

RepositoryModule.kt hosted with ❤️ by GitHub view raw

This abstraction with the interface and its binding is the Dependency Inversion principle (D from SOLID) which is a way of decoupling modules.

High-level modules should not depend on low-level modules, both should depend on abstractions.

In simple terms, this means adding/depending on interfaces so we can easily switch the implementation and decouple our software modules.

Explaining the Data Layer: Repositories Implementation & Data Sources

The Repository Implementation implements the Domain Repository Interface and is in charge of combining 1 or multiple Data Sources. The most common ones are Memory, Cache & Remote.

```kotlin
@Singleton
class PostRepositoryImpl @Inject constructor(
    private val cacheDataSource: PostCacheDataSource,
    private val remoteDataSource: PostRemoteDataSource
) : PostRepository {

    override fun get(refresh: Boolean): Single<List<Post>> =
        when (refresh) {
            true -> remoteDataSource.get()
                .flatMap { cacheDataSource.set(it) }
            false -> cacheDataSource.get()
                .onErrorResumeNext { get(true) }
        }
}
```

PostRepositoryImpl.kt hosted with ❤️ by GitHub view raw

Tests here (omitted to make the story shorter).

This Repository is pulling from the cache and remote interfaces (each Data Source then binds to an implementation). This decoupling makes the Repository agnostic from its Data Sources, avoiding changes when switching a Data Source implementation.

Repositories expect from Data Sources the Domain Models already so it pushes the responsibility of mapping from Data to Domain to each individual Data Source implementation (or from Domain to Data in case you are pushing something to the Data Source).

Data Strategies.

Multiple Data Sources lead to different Data Strategies. My favorite is to only return Cache (unique source of truth) and to refresh Cache from Remote only when it is empty or there is a user action (swipe to refresh for ex.). This saves lots of data and was inspired after reading Build for the next billion users from Android Developers.

Final recap:

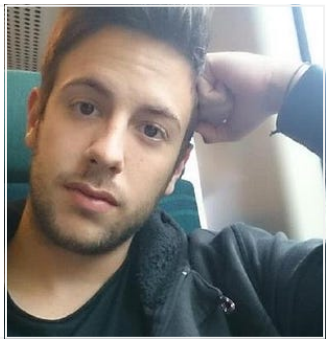
I've omitted the Presentation Layer as it only needs to execute use cases & display data. Remember that each layer has its own entities & mappers and that in order to keep our Domain Layer with no dependencies Data and Presentation are responsible to map to/from Domain Entities depending if you are pushing or pulling data.

Done! 👏 👏 👏

sanogueralorenzo/Android-Kotlin-Clean-Architecture

Android Sample Clean Architecture App written in Kotlin - sanogueralorenzo/Android-Kotlin-Clean-Architecture

github.com



You can check my sample app that was used as an ex. by clicking up here ☝️

Credits:

→ My friend Igor Wojda 👋🤖's Clean Architecture slides and talk.

→ Issue asking why Domain isn't depending on the Data layer.

Remember to follow, share & hit the 👏 button if you've liked it! (:

GitHub | LinkedIn | Twitter

Android | AndroidDev | Android App Development | Mobile App Development | Kotlin

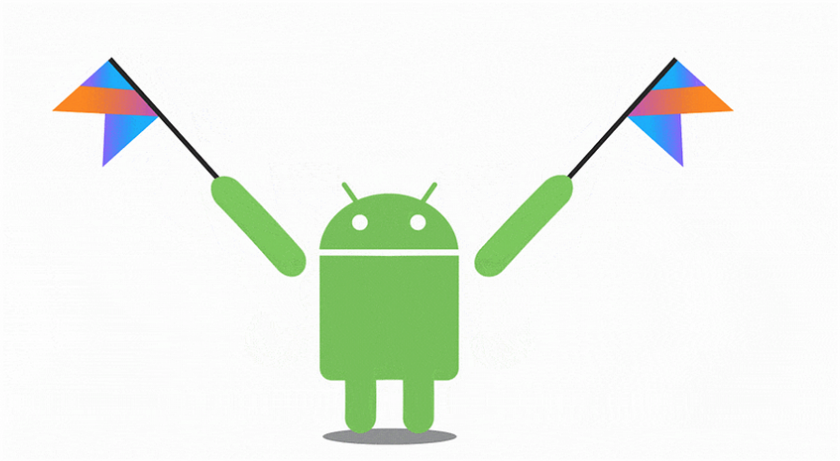
Follow

Written by Mario Sanoguera de Lorenzo

1.3K Followers · Editor for ProAndroidDev

Staff Engineer @Tonal

More from Mario Sanoguera de Lorenzo and ProAndroidDev



Mario Sanoguera de Lorenzo in ProAndroidDev

Moshi with Retrofit in Kotlin ❤️

Hello everyone! 👋 Today I want to show how to implement Moshi with Retrofit (new project or replacing Gson) in an Android project.

2 min read · Jan 11, 2018

1K 2

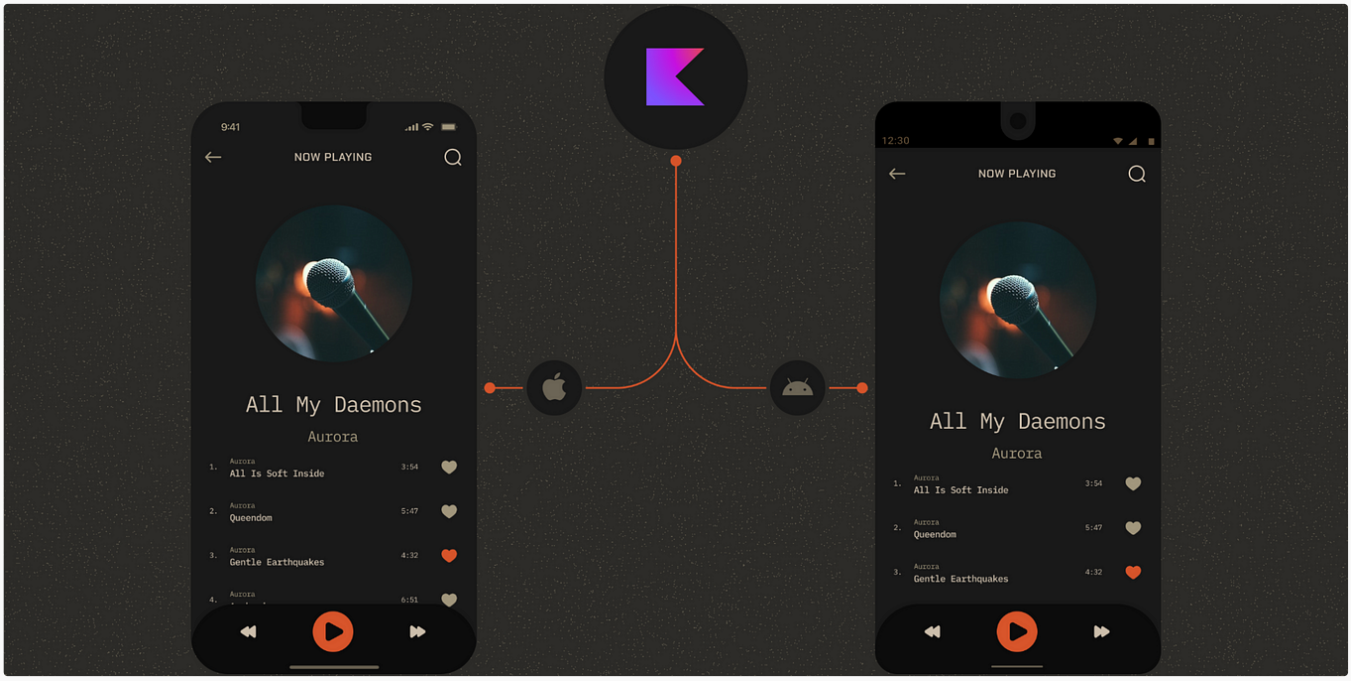
Tom Colvin in ProAndroidDev

A flexible, modern Android app architecture: complete step-by-step

Here we teach Android architecture by example. That means showing *how* various architecture decisions are made. We will encounter...

18 min read · Jul 5

512 4



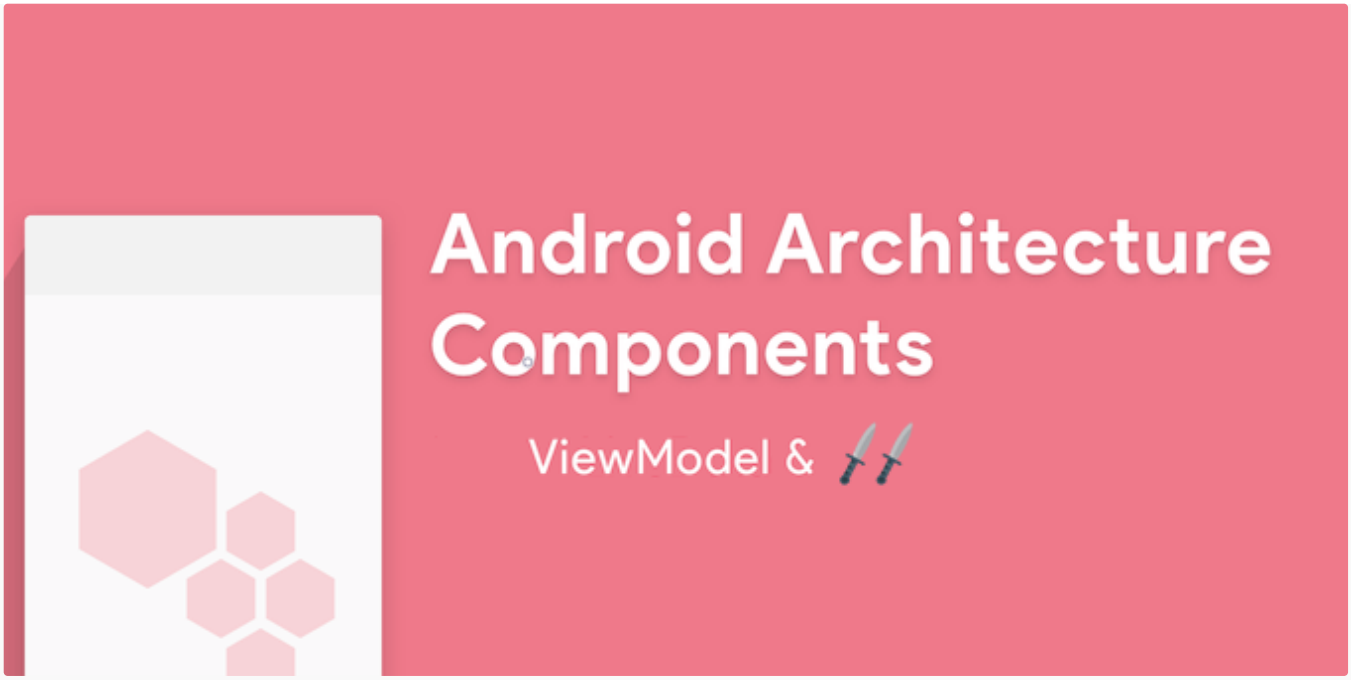
exyte in ProAndroidDev

Jetpack Compose Multiplatform Android & iOS

JetBrains and outside open-source contributors have been diligently working on Compose Multiplatform for several years and recently...

10 min read · Jul 7

247 7



Mario Sanoguera de Lorenzo in ProAndroidDev

ViewModel with Dagger2 (Android Architecture Components)

Hello everyone! 👋 In this story I want to share some light on how you can use ViewModel (Android Architecture Components) with Dagger 2...

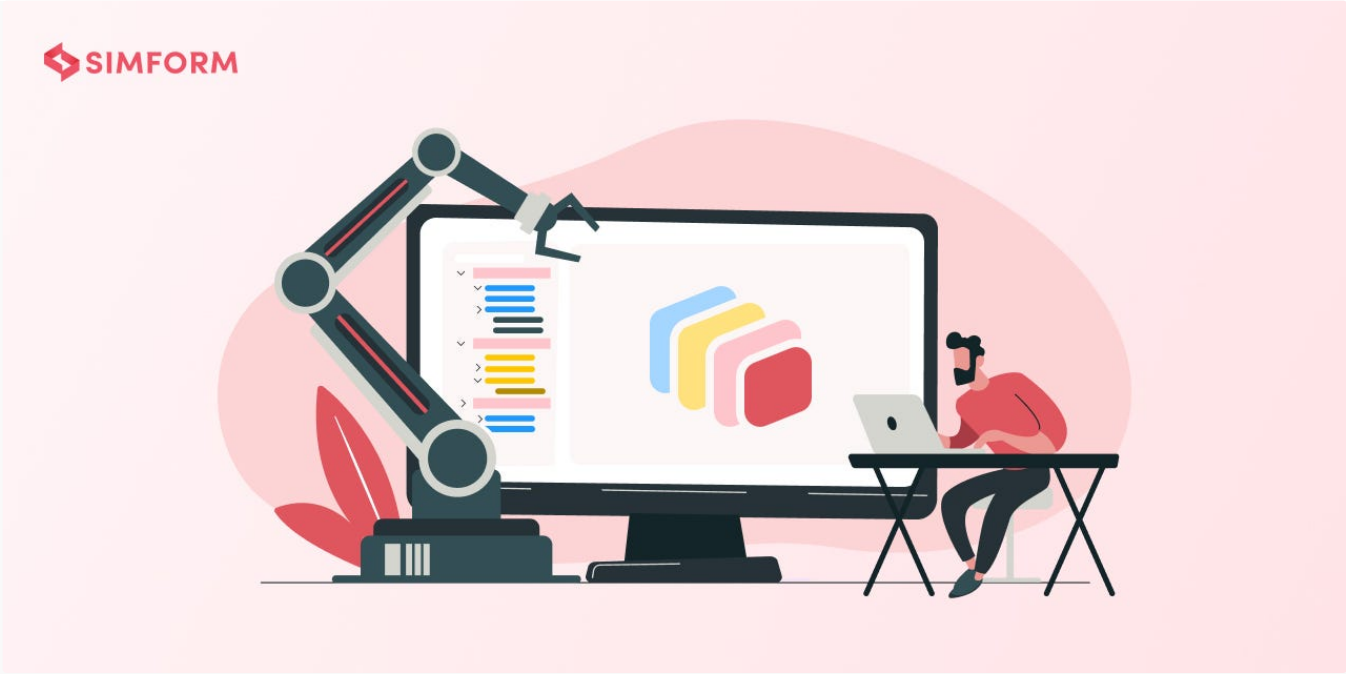
1 min read · Mar 28, 2018

2.2K 13

See all from Mario Sanoguera de Lorenzo

See all from ProAndroidDev

Recommended from Medium



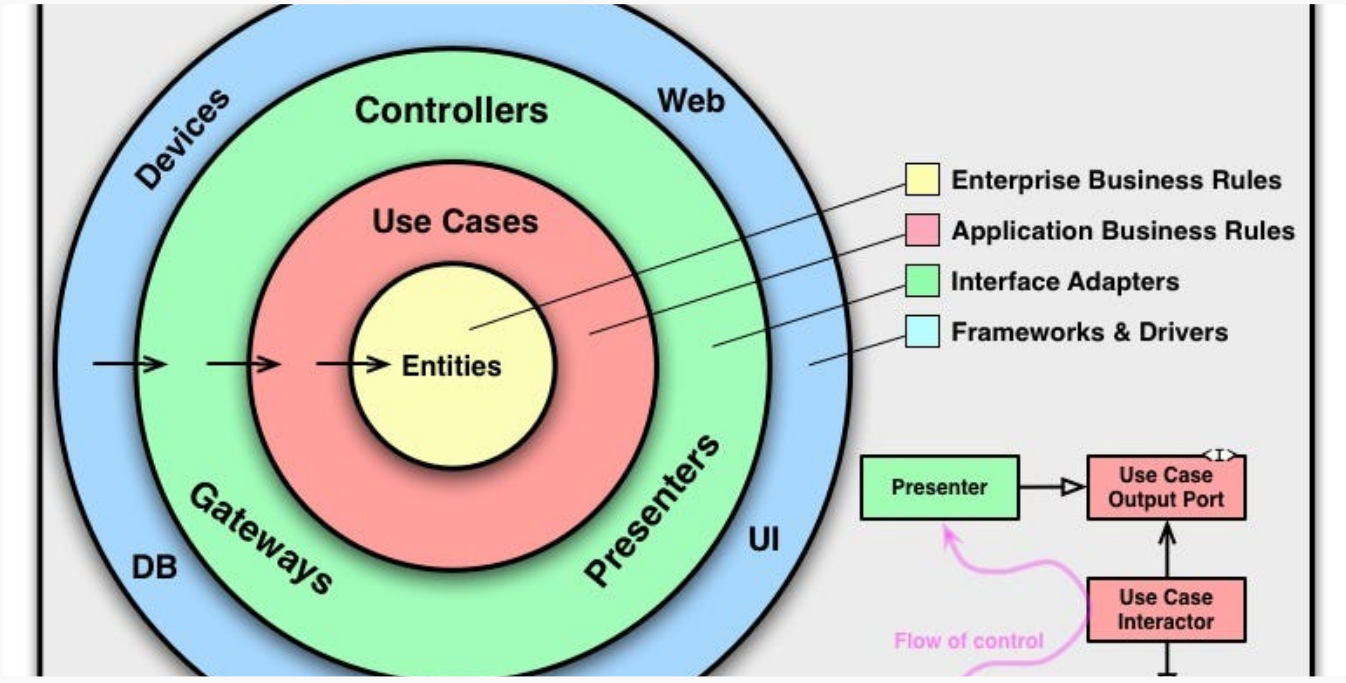
Nishchal Visavadiya in Simform Engineering

Clean Architecture in Android

Minimize the code complexity of your Android app with Clean Architecture!

6 min read · Jan 31

1.6K



Najibvenkitta

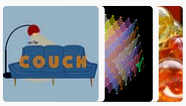
Clean Architecture With Getx

Clean architecture follows the design principle called separation of concerns.

2 min read · Mar 28

1

Lists



Medium Publications Accepting Story Submissions
145 stories · 262 saves

Now in AI: Handpicked by Better Programming
260 stories · 48 saves

Staff Picks
417 stories · 178 saves

Aslihan Gurkan in Dev Genius

MVVM Architectural Design Pattern in Swift

Model-View-ViewModel (MVVM) is a software architecture pattern used to separate the user interface (UI) logic from the business logic or...

5 min read · Apr 3

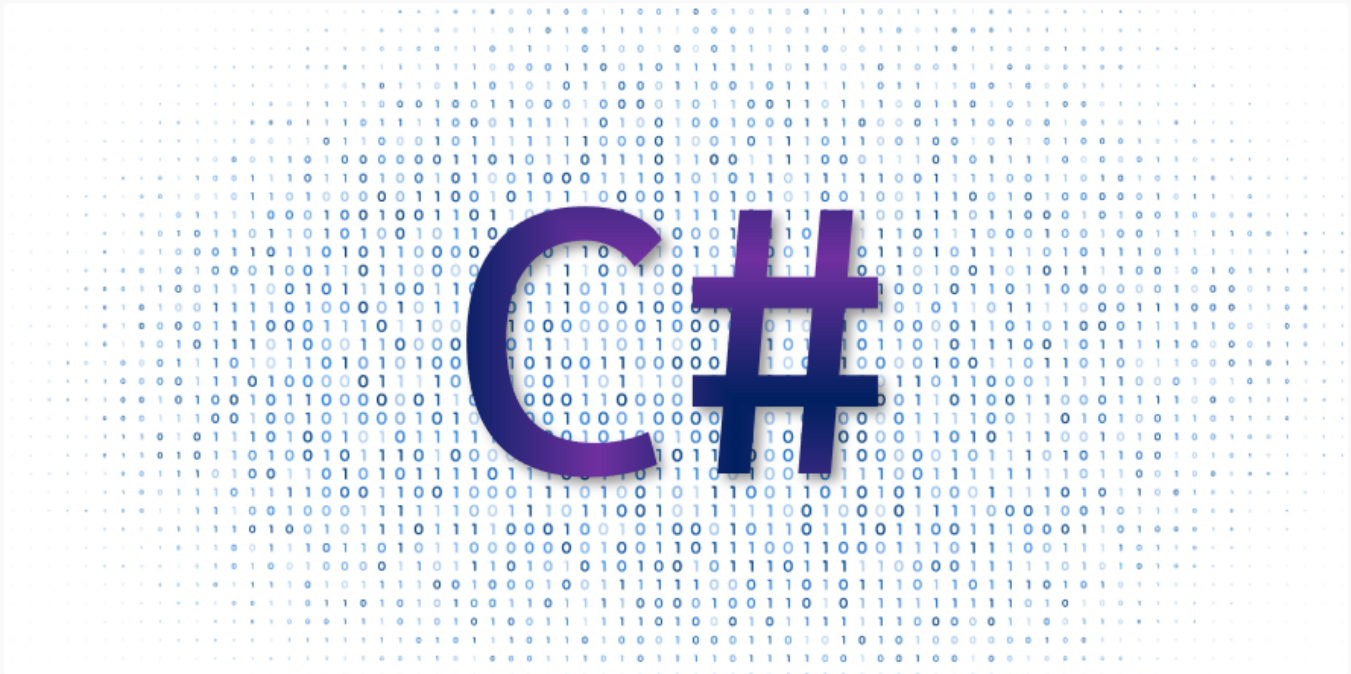
Aseem Wangoo in Better Programming

How To Use MVVM in Flutter

A step-by-step guide for Flutter developers

· 14 min read · Sep 7, 2022

4



Juldhais Hengkyawan

Clean Architecture in ASP .NET Core Web API

A Guide to Building Scalable, Maintainable Web API using ASP .NET Core

12 min read · Feb 3

14

Dheeraj Singh Bhadoria

Jetpack Compose Android App with MVVM Architecture and Retrofit - API Integration

API Integration in Jetpack Compose Android App with MVVM and Retrofit

3 min read · May 23

1

See more recommendations