Clean architecture for Android with Kotlin: a pragmatic approach for starters

by Antonio Leiva | Blog, Kotlin | 66 comments

Clean architecture is a topic that never gets old in the Android world, and from the comments and questions I receive, I feel it's still not very clear.

I know there are tens (or probably hundreds) of articles related to clean architecture, but here I wanted to give **a more pragmatic/simplistic approach** that can help in the first incursion to the clean architecture. That's why I'll be omitting concepts that may feel unavoidable to architecture purists.

My only goal here is that you understand what I consider the main (and most complicated) topic in clean architecture: **the dependency inversion**. Once you get that, you can go to other articles to fill in the little gaps that I may have left outside.

Clean architecture: why should I care?

Even if you decide not to use architectures in your Apps, I think that learning them is really interesting, because they will help you understand important programming and OOP concepts.

Architectures allow decoupling different units of your code in an organized manner. That way the code gets easier to understand, modify and test.

But complex architectures, like the pure clean architecture, **can also bring the opposite effect**: decoupling your code also means creating lots of boundaries, models, data transformations… that may end up increasing the learning curve of your code to a point where it wouldn't be worth it.

So, as you should do with everything you learn, try it in the real world and decide what level of complexity you want to introduce. It will depend on the team, the size of the App, the kind of problems it solves…

So let's start! First, let's define the layers that our App will use.

The layers for a clean architecture

You can see different approaches from different people. But for simplicity, we're sticking to 5 layers (it's complex enough anyway 😂):

1. Presentation

it's the layer that interacts with the UI. You will probably see it divided into two layers in other examples, because you could technically extract everything but the framework classes to another layer. But in practice, it's hardly ever useful and it complicates things.

This presentation layer usually consists of Android UI (activities, fragments, views) and presenters or view models, depending on the presentation pattern you decide to use. If you go for MVP, I have an article where I explain it in deep (and I would like to write one about MVVM soon).

2. Use cases

It's usually called interactors too. These are mainly **the actions that the user can trigger**. Those can be active actions (the user clicks on a button) or implicit actions (the App navigates to a screen).

If you want to be extra-pragmatic, you can even avoid this layer. I like it because it's usually the point where I switch threads. From this point, I can run everything else on a background thread and forget about being careful

with the UI thread. I like it because I don't need to wonder anymore if something is running in the UI thread or a background thread.

3. Domain

Also known as business logic. **These are the rules of your business**.

It contains all the business models. For instance, in a movies App, it could be the `Movie` class, the `Subtitle` class, etc.

Ideally, it should be the biggest layer, though it's true that Android Apps usually tend to just draw an API in the screen of a phone, so most of the core logic will just consist of requesting and persisting data.

4. Data

In this layer, you have **an abstract definition of the different data sources**, and how they should be used. Here, you will normally use a repository pattern that, for a given request, it's able to decide where to find the information.

In a typical App, you would save your data locally and recover it from the network. So this layer can check whether the data is in a local database. If it's there and it's not expired, return it as a result, and otherwise ask the API for it and save it locally.

But data not only comes from a request. You may, for instance, need data from the device sensors, or from a `BroadcastReceiver` (though **the data layer should never know about this concept**! We'll see it later)

5. Framework

You can find this layer called in many different ways. It basically encapsulates the **interaction with the framework**, so that the rest of the code can be agnostic and reusable in case you want to implement the same App in another platform (a real option nowadays with Kotlin multi-platform projects!). With *framework* I'm not only referring to the Android framework here, but to any external libraries that we want to able to change easily in the future.

For instance, if the data layer needs to persist something, here you could use Room to do it. Or if it needs to do a request, you would use Retrofit. Or

it can access the sensors to request some info. Whatever you need!

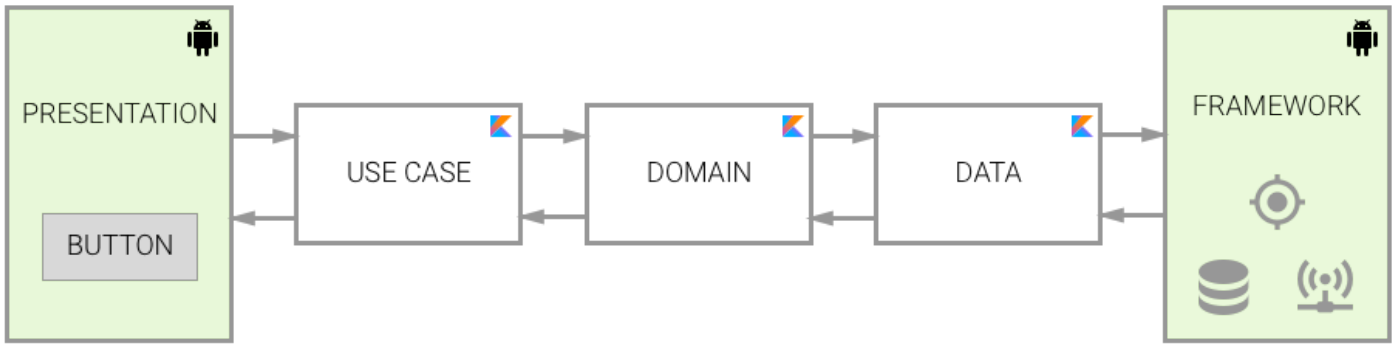
This layer should be as simple as possible, as all the logic should be abstracted into the data layer.

> *Remember! These are the suggested layers, but some of them can be merged. You could even just have three layers: presentation – domain – framework. This probably can't be strictly called clean architecture, but I honestly don't care about namings. I'll leave 5 layers because it helps me explain the next point, which is the important one.*

Interaction between layers

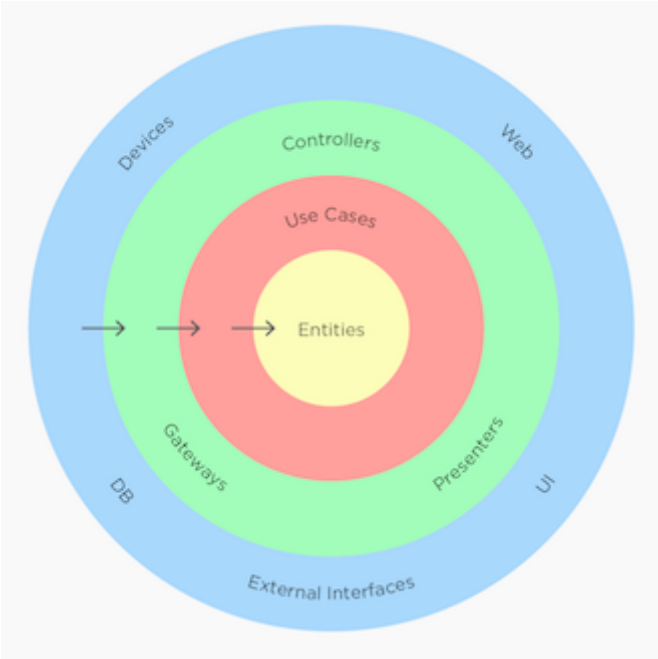
So this is the hardest part to explain and understand. I'll try to be as clear as possible because I think this is also the most important point if you want to understand the clean architecture. But feel free to write me if you don't understand anything, and I'll update this text.

When you think of a logical way of interaction, you'd say that the presentation uses the use cases layer, which then will use the domain to access the data layer, which will finally use the framework to get access to the requested data. Then this data flies back to the layer structure until it reaches the presentation layer, which updates the UI. This would be a simple graph of what's happening:



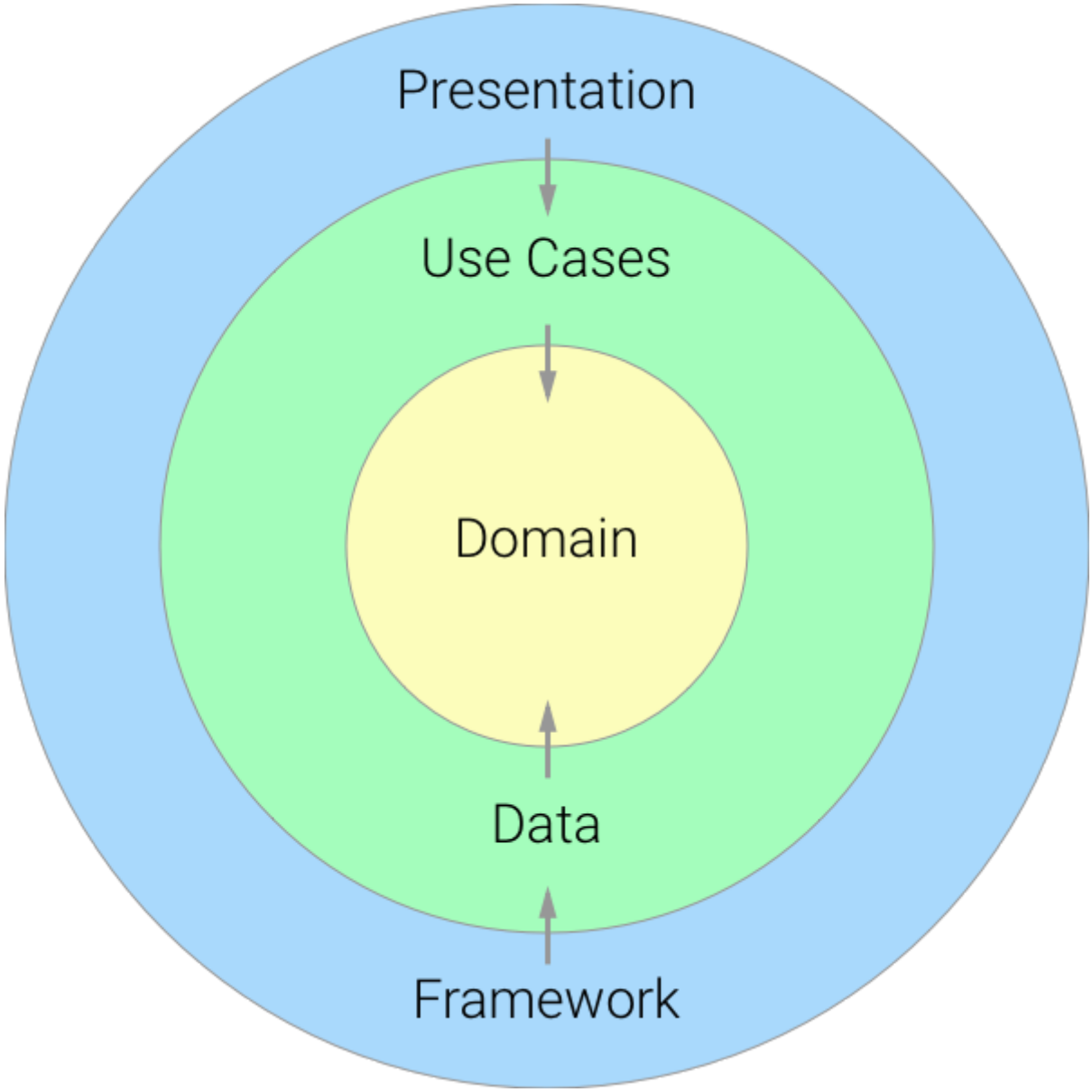
As you can see, the two limits of the flow depend on the framework, so they require using the Android dependency, while the rest of the layers only require Kotlin. This is really interesting if you want to divide each layer into a separate submodule. **If you were to reuse the same code for (let's say) a Web App, you'd just need to reimplement the presentation and framework layers**.

But don’t mix the flow of the App with the direction of the dependencies between layers. If you’ve read about clean architecture before, you probably saw this graph:



Which is a bit different from the previous image. Namings are also different, but I reformulate this in a minute. Basically, the clean architecture says that we have outer and inner layers, and that the inner layers shouldn’t know about the outer ones. This means that an outer class can have an explicit dependency from an inner class, but not the other way round.

Let’s recreate the above graph with our own layers:



So from the UI to the domain, everything is quite simple, right? The presentation layer has a Use Case dependency, and it's able to call to start the flow. Then the Use Case has a dependency to the domain.

But the problems come we go from the inside to the outside. For instance, when the data layer needs something from the framework. As it's an inner layer, the Data layer doesn't know anything about the external layers, so how can it communicate with them?

Pay attention, here it comes an important concept.

Dependency Inversion Principle

If you have learned about SOLID principles, you may have read about Dependency Inversion. But, as with many of these concepts, it's possible that you didn't understand how to apply it. The dependency inversion is the "D" of SOLID, and this is what it states:

> *A. High-level modules should not depend on low-level modules. Both should depend on abstractions.*
> *B. Abstractions should not depend on details. Details should depend on abstractions.*

So, honestly, this doesn't say too much to me in my day to day work. But with our example, it's easier to understand.

An example for dependency inversion

Let's say we have a `DataRepository` in the data layer that requires a `RoomDatabase` to recover some persisted data. The first approach would be to have something like this. This is our `RoomDatabase`:

```
class RoomDatabase {
    fun requestItems(): List<Item> { ... }
}
```

And the `DataRepository` would instantiate it and use it:

```
class DataRepository {
    private val roomDatabase = RoomDatabase()
    fun requestItems(): List<Item> {
        val items = roomDatabase.requestItems()
        ...
        return result
    }
}
```

But this is not possible! **The Data layer doesn't know about the classes in Framework because it's an inner layer**.

So the first step is to do an **inversion of control** (don't mix with dependency inversion, they're not the same), which means that instead of instantiating the class ourselves, we let it be provided from the outside (through the constructor):

```
class DataRepository(private val roomDatabase: RoomDatabase) {
    fun requestItems(): List<Item> {
        val items = roomDatabase.requestItems()
        ...
        return result
    }
```

```
}
```

Easy right? But here it is where we need to do the dependency inversion. Instead of depending on the specific implementation, we're going to depend on an abstraction (an interface). So the data module will have the following interface:

```
interface DataPersistence {
    fun requestItems(): List<Item>
}
```

Now the `DataRepository` can just use the interface (which is in its same layer):

```
class DataRepository(private val dataPersistence: DataPersistence) {
    fun requestItems(): List<Item> {
        val items = dataPersistence.requestItems()
        ...
        return result
    }
}
```

And as the framework layer can use the data layer, it can implement that interface:

```
class RoomDatabase : DataPersistence {
    override fun requestItems(): List<Item> { ... }
}
```

The only remaining point would be how to provide the dependency to the `DataRepository`. **That's done using dependency injection**. The external layers will take care of it.

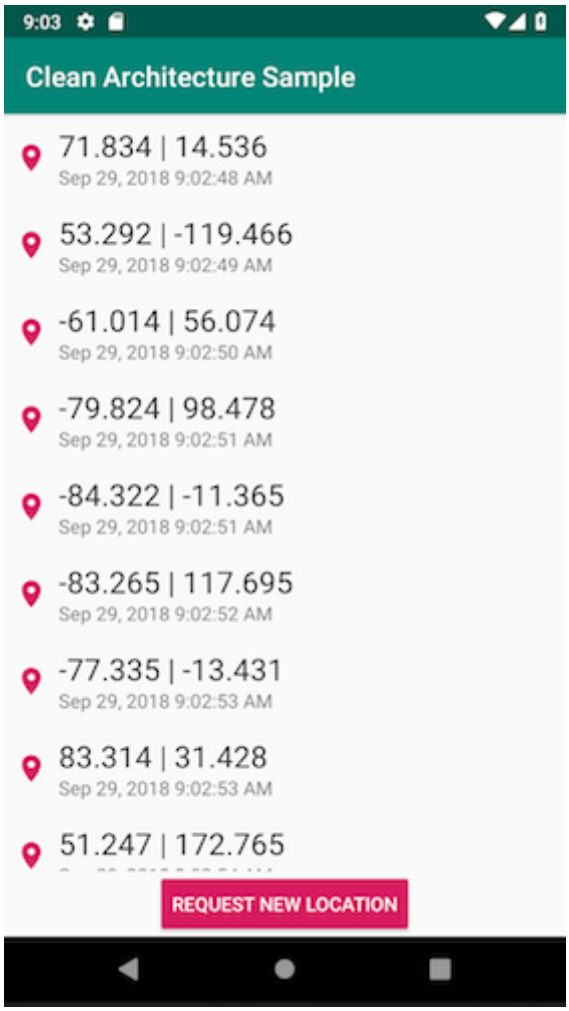
I won't dive deep into dependency injection in this article, as I don't want to add many complex concepts. I have a set of articles talking about it and about Dagger if you want to know more.

Building an example

First of all, you can find the whole example in this repository. I'll be skipping some details, so be sure to go there, fork it and play with it.

So all this is really good, but you need to put it into practice if you want to understand it.

For that, we're creating an App that will allow to request the current geolocation thanks to a button and keep a record of the previously requested locations, by showing them on a list.



Building a sample project

The project will consist of a set of 5 modules.

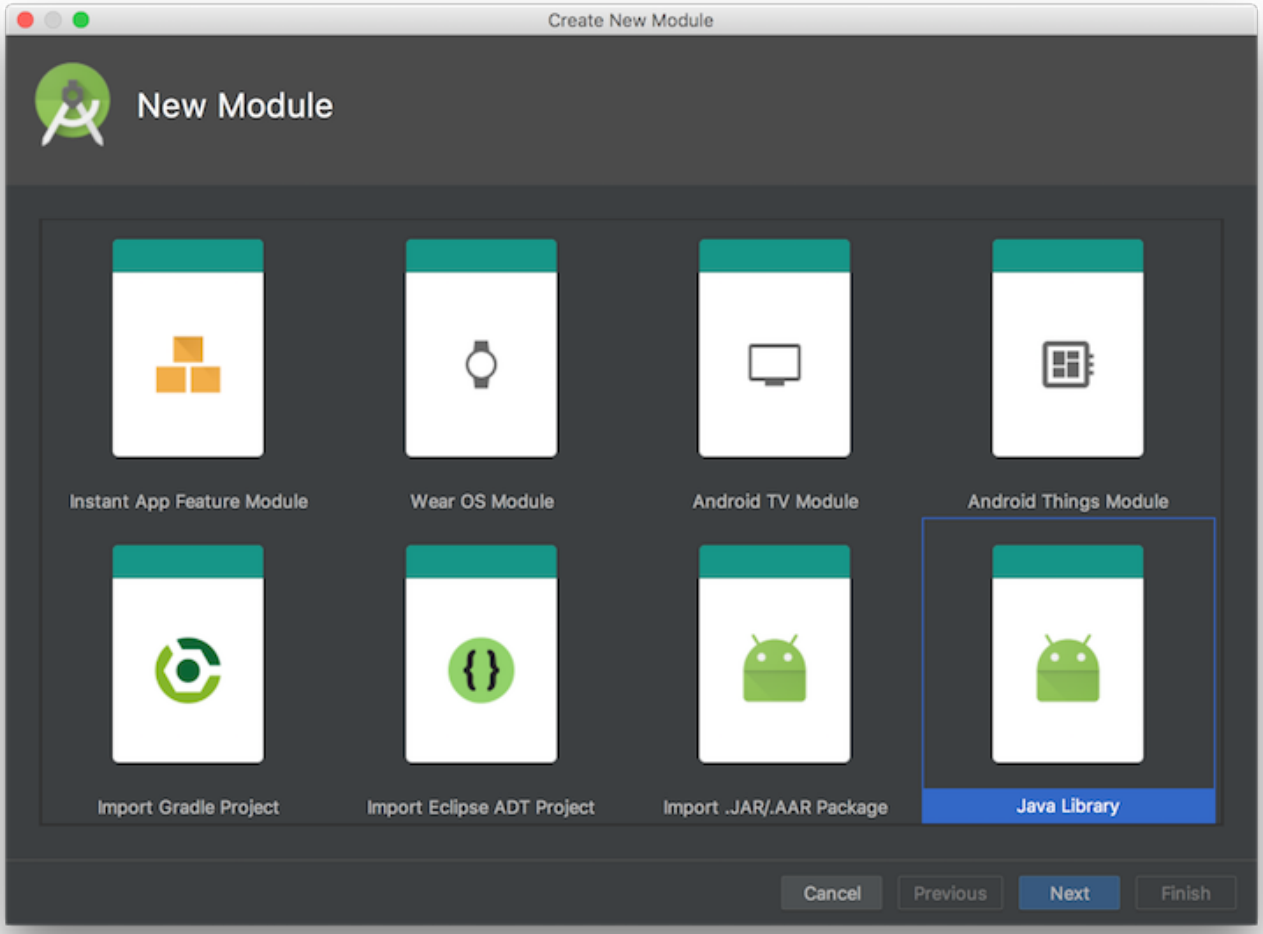
For simplicity, I'm only creating 4:

- **app**: it will be the only project that uses the Android framework, which will include the Presentation and Framework layers.
- **usecases**: it will be a Kotlin module (doesn't need the Android framework).
- **domain**: another Kotlin module.

- **data**: a Kotlin module too.

You could perfectly do everything in the same module, and use packages. But it's easier not to violate the dependency flow if you do it like that. So I recommend it if you're starting.

Create a new project, which will automatically create the `app` module, and then create the extra Java modules:



There is not a "Kotlin Library" option, so you will need to add the Kotlin support later.

The domain layer

We need a class that represents the location. On Android, we already have a Location class. But remember that we want to leave the implementation details in the outer layers.

Imagine that you want to use this code tomorrow for a web App written in KotlinJS. You won't have access to the Android classes anymore.

So here it is the class:

```
data class Location(val latitude: Double, val longitude: Double, val date: Date)
```

> *If you read about this before, pure clean architecture would have one model representation per layer, which in our case would involve having a `Location` class on each layer. Then, you would use data transformations to convert it through different layers. That makes layers less coupled, but also everything more complex. In this example, I'll just do it when it's strictly required.*

That's probably all you need in this layer for this simple example.

The data layer

The data layer is usually modeled by using repositories that have access to the data we need. We can have something like this:

```
class LocationsRepository {
    fun getSavedLocations(): List<Location> { ... }
    fun requestNewLocation(): List<Location> { ... }
}
```

It has a function to get the old requested locations, and another function to request a new one.

As you can see, this layer is using the domain layer. An outer layer can use the inner layers (but not the other way round). To do that, you need to add a new dependency to the module `build.gradle`:

```
dependencies {
    implementation project(':domain')
    ...
}
```

The repository is going to use a couple of sources:

```
class LocationsRepository(
    private val locationPersistenceSource: LocationPersistenceSource,
    private val deviceLocationSource: DeviceLocationSource
```

```
)
```

One of them has access to the persisted locations, and the other to the device location.

And here it is where the dependency inversion magic happens. These two sources are interfaces:

```
interface LocationPersistenceSource {
    fun getPersistedLocations(): List<Location>
    fun saveNewLocation(location: Location)
}
interface DeviceLocationSource {
    fun getDeviceLocation(): Location
}
```

And the data layer doesn't know (and doesn't need to know) what's the real implementation of these interfaces. The persistence and device locations need to be managed by the specific device framework. Again, getting back to the KotlinJS example, a web App would implement this very differently from an Android App.

Now, the `LocationsRepository` can use these sources without knowing about the final implementation:

```
fun getSavedLocations(): List<Location> =
locationPersistenceSource.getPersistedLocations()
fun requestNewLocation(): List<Location> {
    val newLocation = deviceLocationSource.getDeviceLocation()
    locationPersistenceSource.saveNewLocation(newLocation)
    return getSavedLocations()
}
```

The Use Cases layer

This is usually a very simple layer, that just converts user actions into interactions with the rest of inner layers. In our case, we're having a couple of use cases:

- `GetLocations`: it returns the locations that have already been recorded by the App.

- `RequestNewLocation`: it will tell the `LocationsRepository` to look for the current location

These use cases will get a dependency to the `LocationRepository`:

```
class GetLocations(private val locationsRepository: LocationsRepository) {
    operator fun invoke(): List<Location> =
locationsRepository.getSavedLocations()
}
```

```
class RequestNewLocation(private val locationsRepository:
LocationsRepository) {
    operator fun invoke(): List<Location> =
locationsRepository.requestNewLocation()
}
```

The framework layer

This one will be part of the `app` module, and will mainly implement the dependencies that we provide to the rest of layers. In our particular case, it will be `LocationPersistenceSource` and `DeviceLocationSource`.

The first could be implemented with Room for instance, and the second with the `LocationManager`. But, for the purpose of making this explanation simpler, I'll be using fake implementations. I might do the real implementation at the end, but it would only add complexity to the explanation so I prefer you to forget about it for now.

For the persistence, I'm using a simple in-memory implementation:

```
class InMemoryLocationPersistenceSource : LocationPersistenceSource {
    private var locations: List<Location> = emptyList()
    override fun getPersistedLocations(): List<Location> = locations
    override fun saveNewLocation(location: Location) {
        locations += location
    }
}
```

And a random generator for the other one:

```
class FakeLocationSource : DeviceLocationSource {
    private val random = Random(System.currentTimeMillis())
    override fun getDeviceLocation(): Location =
        Location(random.nextDouble() * 180 - 90,
```

```
            random.nextDouble() * 360 - 180, Date())
}
```

Think about this in a real project. Thanks to the interfaces, **during the implementation of a new feature, you can provide fake dependencies** while working on the rest of the flow, and forget about the implementation details until the end.

This also proves that **these implementation details are easily interchangeable**. So maybe your App can work initially with an in-memory persistence, and then move to stored persistence. That can be implemented as we want, and then replaced.

Imagine that a new shiny library appears (like Room 😂) and you want to test it and consider about migrating. You just need to implement the interface using the new library, replace the dependency, and that's it!

And of course, **this also helps on tests, where we can replace those components by fake or mocked ones**.

The presentation layer

And now we can write the UI. For this example, I'm using MVP, because this article was originated by the questions in my original article about MVP, and because I think that it's easier to understand that using MVVM with architecture components. But both approaches are very similar.

First, we need to write the presenter, which will receive a `View` dependency (the presenter interface to interact with its view) and the two use cases:

```
class MainPresenter(
    private var view: View?,
    private val getLocations: GetLocations,
    private val requestNewLocation: RequestNewLocation
) {
    interface View {
        fun renderLocations(locations: List<Location>)
    }
    fun onCreate() = launch(UI) {
        val locations = bg { getLocations() }.await()
        view?.renderLocations(locations)
    }
    fun newLocationClicked() = launch(UI) {
        val locations = bg { requestNewLocation() }.await()
        view?.renderLocations(locations)
```

```
    }
    fun onDestroy() {
        view = null
    }
}
```

All quite simple here, apart from the way to do the background tasks. I'm using coroutines. I'm not sure if it's the best decision, because I wanted to keep this example as easy as possible. So let me know in the comments if you don't understand it. I already talked about Kotlin 1.3 coroutines on this blog if you are interested.

Finally, the `MainActivity`. In order to avoid using a dependency injector, I declared the dependencies here:

```
private val presenter: MainPresenter
init {
    // This would be done by a dependency injector in a complex App
    //
    val persistence = InMemoryLocationPersistenceSource()
    val deviceLocation = FakeLocationSource()
    val locationsRepository = LocationsRepository(persistence,
deviceLocation)
    presenter = MainPresenter(
        this,
        GetLocations(locationsRepository),
        RequestNewLocation(locationsRepository)
    )
}
```

I wouldn't recommend this for a big App, because you wouldn't be able to replace the dependencies in UI tests for instance, but it's enough for this example.

And the rest of the code doesn't need much explanation. It has a `RecyclerView` and a `Button`. When the button is clicked, it calls to the presenter so that it requests a new location:

```
newLocationBtn.setOnClickListener { presenter.newLocationClicked() }
```

And when the presenter finishes, it calls the `View` method. This interface is implemented by the activity this way:

```
override fun renderLocations(locations: List<Location>) {
    locationsAdapter.items = locations
}
```

Layer models and data transformations

As I mentioned above, I don't like creating models for all layers just by default. But to me, the presentation can be a good place where we could make use of this concept.

As you saw in the adapter code, I did some complex calculations to convert the domain model into what we want to see in the screen. That can be simplified by using a specific model:

```
data class Location(val coordinates: String, val date: String)
```

And then, we can convert the domain model into the presentation model. In this case, I'm using an extension function that applies to the domain `Location`. Remember that, thanks to named imports, we can change the name of the classes when we have two that are called the same:

```
import com.antonioleiva.domain.Location as DomainLocation
...
fun DomainLocation.toPresentationModel(): Location = Location(
    "${latitude.toPrettifiedString()} | ${longitude.toPrettifiedString()}",
    date.toPrettifiedString()
)
```

After this, the transformation in the presenter is really easy:

```
view?.renderLocations(locations.map(DomainLocation::toPresentationModel))
```

And finally, the adapter looks much simpler:

```
fun bind(location: Location) {
    with(location) {
        locationCoordinates.text = coordinates
        locationDate.text = date
    }
}
```

That way, we prevent from doing data transformations in the UI classes, which reduces the complexity.

Conclusion

So that's it! The basics of clean architecture are in fact quite simple.

You only need to understand **how the dependency inversion works, and then link the layers properly**. Just remember not to add dependencies to the outer modules from the inner ones, and you will have an excellent help from the IDE to do things right.

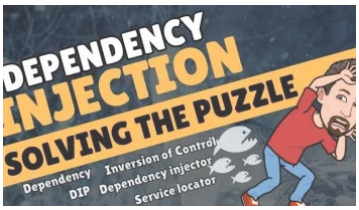
I know this is a lot of information. But my goal is that this serves as an entry point for people that never saw clean architecture before. So if you still have doubts, please let me know in the comments and I'll rewrite this article as many times as necessary.

And also remember that in order to make this article simple, I've omitted some complexities that you would find in a regular clean architecture. Once you have this settled, I suggest you read other more complete examples. I always like to recommend this one from Fernando Cejas.

The link to the Github repository is here. Go there to review the details, and if you like it, please show it with a star ☺

Happy coding!

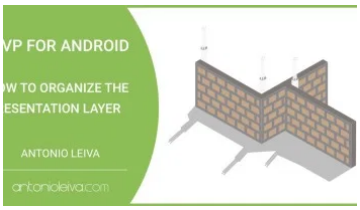
Related



Dependency Injection - All the Must-Know Concepts to start using it

June 26, 2019

In "Blog"



MVP for Android: how to organize the presentation layer

July 4, 2018

In "Blog"

MVVM with architecture components: a step by step guideline for MVP lovers

December 12, 2018

In "Blog"

66 Comments

1.

Omer Zak on October 1, 2018 at 00:58

Reply

I dislike the concept of layers, in which a layer is allowed to depend upon a layer below it but not vice versa.

I prefer to have those “layers” independent from each other. A layer would connect to another layer only via interfaces, both directions.

Such a design would allow one to replace any layer without touching the other layers. Useful for unit tests and for creating a line of similar products.

- 

  Antonio Leiva on October 1, 2018 at 08:49

  That’s another approach, yeah. What I’ve found is that the inner layers usually don’t need interfaces because you can replace them with mocks in tests. It’s just a practical decision, and from a practical point of view, I’ve never required modifying them. I usually don’t need to change the classes in inner classes because they are the business core, so if it changes, it doesn’t make sense to keep the old one. It’s usually not the same with external dependencies. But that doesn’t mean that there aren’t any other great

architectures. I'd love to see some code following your approach if you have something published.