Guide to app architecture

Mobile app user experiences

- A typical Android app contains multiple app components, including activities, fragments, services, content providers, and broadcast receivers. You declare most of these app components in your app manifest.
- The Android OS then uses this file to decide how to integrate your app into the device's overall user experience.
- Given that a typical Android app might contain multiple components and that users often interact with multiple apps in a short period of time, apps need to adapt to different kinds of user-driven workflows and tasks.

- Keep in mind that mobile devices are also resource-constrained, so at any time, the operating system might kill some app processes to make room for new ones.

- Given the conditions of this environment, it's possible for your app components to be launched individually and out-of-order, and the operating system or user can destroy them at any time.

- Because these events aren't under your control, you shouldn't store or keep in memory any application data or state in your app components, and your app components shouldn't depend on each other.

Common architectural principles

- If you shouldn't use app components to store application data and state, how should you design your app instead?
- As Android apps grow in size, it's important to define an architecture that allows the app to scale, increases the app's robustness, and makes the app easier to test.
- An app architecture defines the boundaries between parts of the app and the responsibilities each part should have. In order to meet the needs mentioned above, you should design your app architecture to follow a few specific principles.

Separation of concerns

- The most important principle to follow is separation of concerns. It's a common mistake to write all your code in an Activity or a Fragment. These UI-based classes should only contain logic that handles UI and operating system interactions.

- By keeping these classes as lean as possible, you can avoid many problems related to the component lifecycle, and improve the testability of these classes.

- Keep in mind that you don't own implementations of Activity and Fragment; rather, these are just glue classes that represent the contract between the Android OS and your app.

- The OS can destroy them at any time based on user interactions or because of system conditions like low memory.

- To provide a satisfactory user experience and a more manageable app maintenance experience, it's best to minimize your dependency on them.

Drive UI from data models

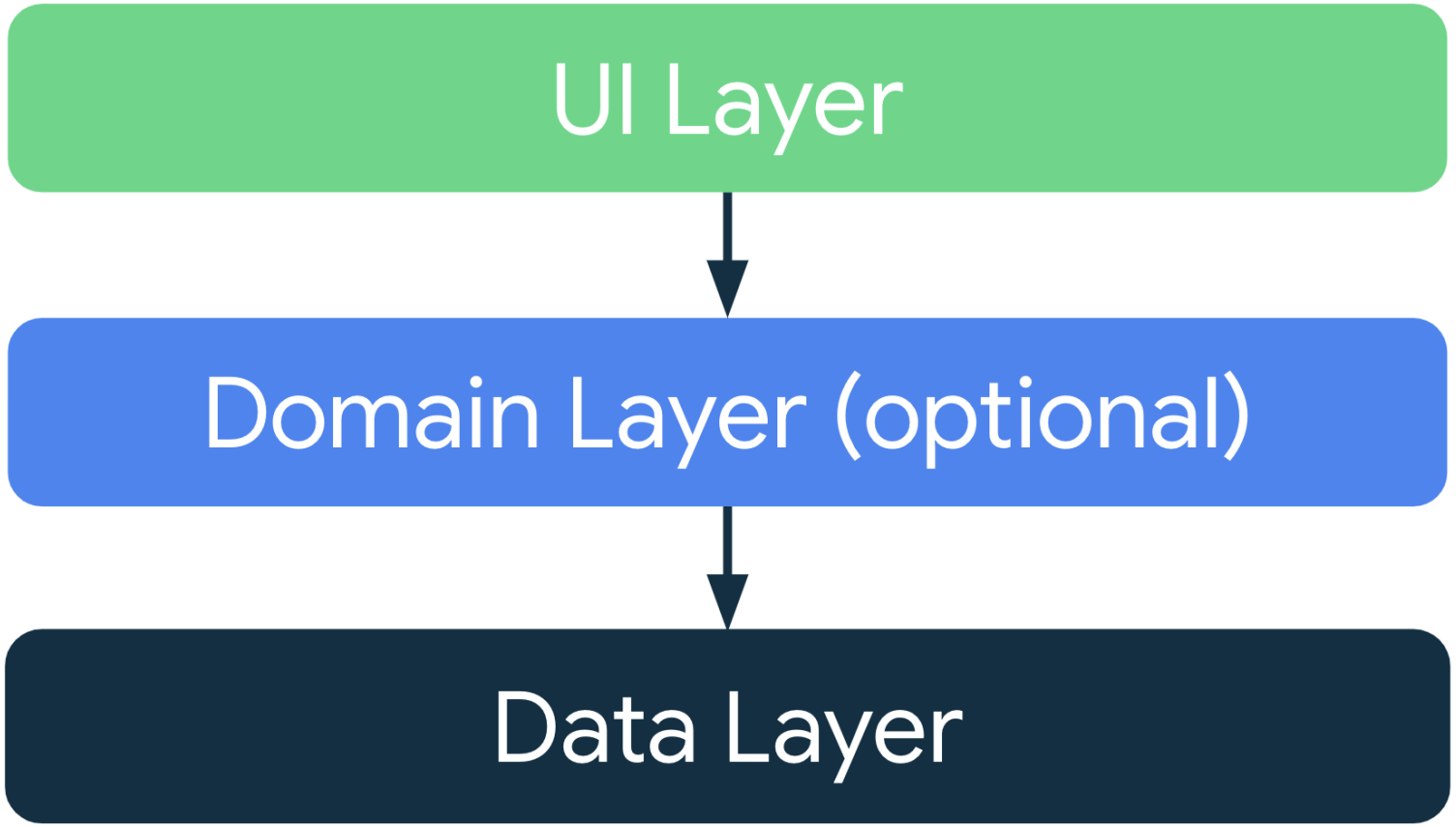
- Another important principle is that you should drive your UI from data models, preferably persistent models. Data models represent the data of an app. They're independent from the UI elements and other components in your app. This means that they are not tied to the UI and app component lifecycle but will still be destroyed when the OS decides to remove the app's process from memory.

- Persistent models are ideal for the following reasons:
  - Your users don't lose data if the Android OS destroys your app to free up resources.
  - Your app continues to work in cases when a network connection is flaky or not available.
  - If you base your app architecture on data model classes, you make your app more testable and robust.

Recommended app architecture

- Considering the common architectural principles mentioned in the previous section, each application should have at least two layers:
  - The *UI layer* that displays application data on the screen.
  - The *data layer* that contains the business logic of your app and exposes application data.
- You can add an additional layer called the *domain layer* to simplify and reuse the interactions between the UI and data layers.

Diagram of a typical app architecture.



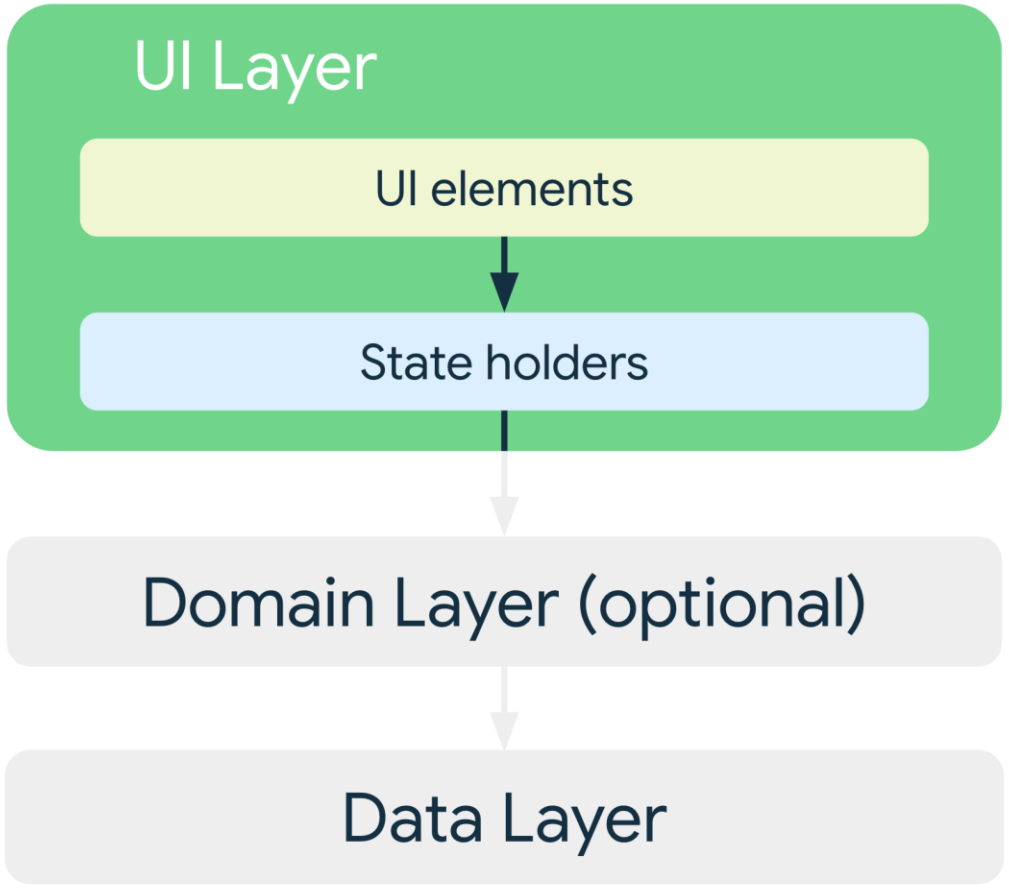
UI layer

- The role of the UI layer (or *presentation layer*) is to display the application data on the screen. Whenever the data changes, either due to user interaction (such as pressing a button) or external input (such as a network response), the UI should update to reflect the changes.

The UI layer is made up of two things:

- UI elements that render the data on the screen. You build these elements using Views or Jetpack Compose functions.
- State holders (such as ViewModel classes) that hold data, expose it to the UI, and handle logic.

The UI layer's role in app architecture.

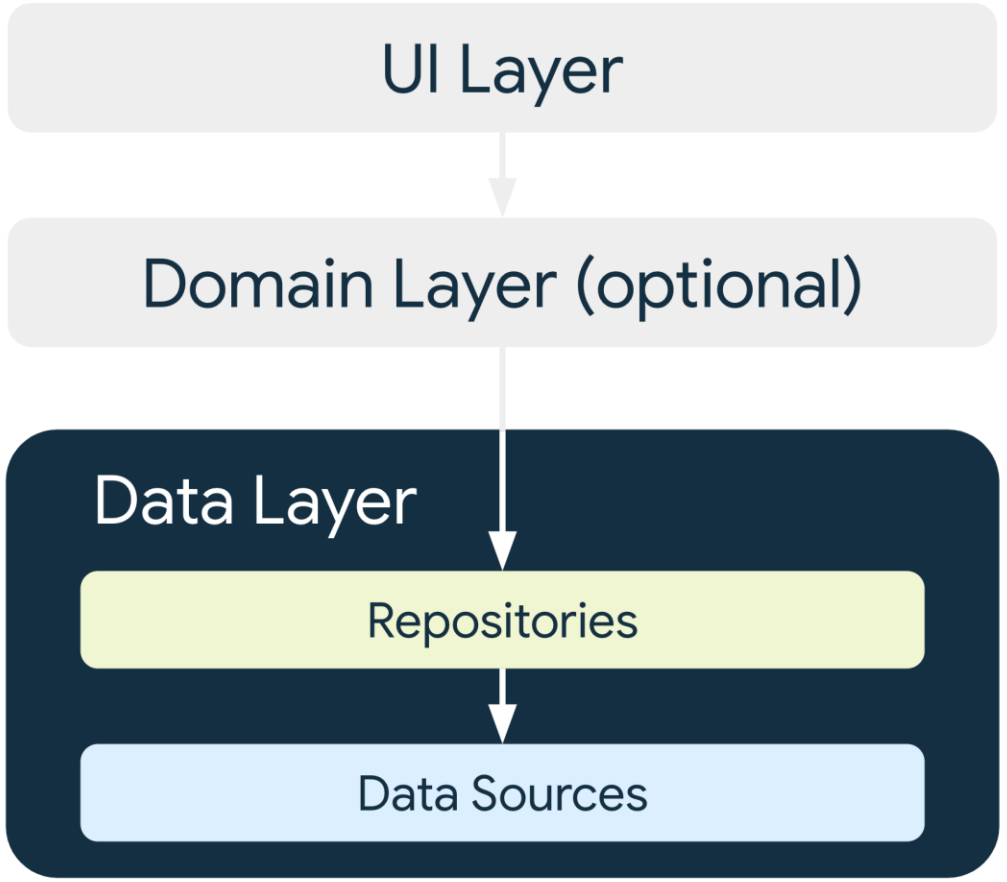


Data layer

- The data layer of an app contains the *business logic*. The business logic is what gives value to your app—it's made of rules that determine how your app creates, stores, and changes data.

- The data layer is made of *repositories* that each can contain zero to many *data sources*. You should create a repository class for each different type of data you handle in your app. For example, you might create a MoviesRepository class for data related to movies, or a PaymentsRepository class for data related to payments.

The data layer's role in app architecture.



Repository classes are responsible for the following tasks:

- Exposing data to the rest of the app.
- Centralizing changes to the data.
- Resolving conflicts between multiple data sources.
- Abstracting sources of data from the rest of the app.
- Containing business logic.

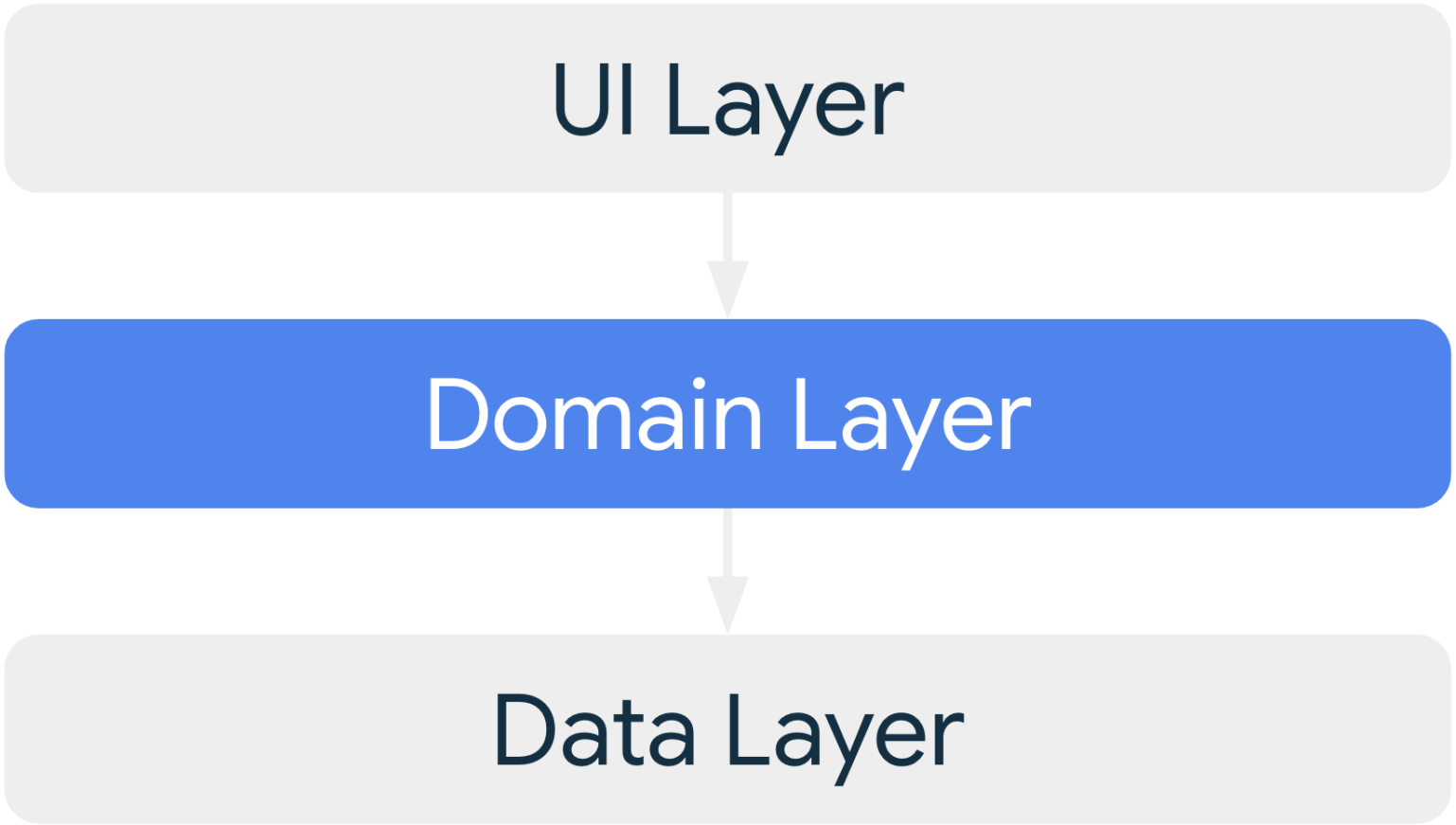
- Each data source class should have the responsibility of working with only one source of data, which can be a file, a network source, or a local database. Data source classes are the bridge between the application and the system for data operations.

Domain layer

- The domain layer is an optional layer that sits between the UI and data layers.

- The domain layer is responsible for encapsulating complex business logic, or simple business logic that is reused by multiple ViewModels. This layer is optional because not all apps will have these requirements. You should use it only when needed—for example, to handle complexity or favor reusability.

The domain layer's role in app architecture.



- Classes in this layer are commonly called *use cases* or *interactors*. Each use case should have responsibility over a *single* functionality.

- For example, your app could have a GetTimeZoneUseCase class if multiple ViewModels rely on time zones to display the proper message on the screen.

Manage dependencies between components

- Classes in your app depend on other classes in order to function properly. You can use either of the following design patterns to gather the dependencies of a particular class:
  - Dependency injection (DI): Dependency injection allows classes to define their dependencies without constructing them. At runtime, another class is responsible for providing these dependencies.
  - Service locator: The service locator pattern provides a registry where classes can obtain their dependencies instead of constructing them.
- These patterns allow you to scale your code because they provide clear patterns for managing dependencies without duplicating code or adding complexity. Furthermore, these patterns allow you to quickly switch between test and production implementations.

General best practices

- Programming is a creative field, and building Android apps isn't an exception. There are many ways to solve a problem; you might communicate data between multiple activities or fragments, retrieve remote data and persist it locally for offline mode, or handle any number of other common scenarios that nontrivial apps encounter.

- Although the following recommendations aren't mandatory, in most cases following them makes your code base more robust, testable, and maintainable in the long run:

Don't store data in app components.

- Avoid designating your app's entry points—such as activities, services, and broadcast receivers—as sources of data.

- Instead, they should only coordinate with other components to retrieve the subset of data that is relevant to that entry point.

- Each app component is rather short-lived, depending on the user's interaction with their device and the overall current health of the system.

Reduce dependencies on Android classes.

- Your app components should be the only classes that rely on Android framework SDK APIs such as Context, or Toast.

- Abstracting other classes in your app away from them helps with testability and reduces coupling within your app.

Create well-defined boundaries of responsibility between various modules in your app.

- For example, don't spread the code that loads data from the network across multiple classes or packages in your code base.

- Similarly, don't define multiple unrelated responsibilities—such as data caching and data binding—in the same class.

Expose as little as possible from each module.

- For example, don't be tempted to create a shortcut that exposes an internal implementation detail from a module.

- You might gain a bit of time in the short term, but you are then likely to incur technical debt many times over as your codebase evolves.

Focus on the unique core of your app so it stands out from other apps.

- Don't reinvent the wheel by writing the same boilerplate code again and again.

- Instead, focus your time and energy on what makes your app unique, and let the Jetpack libraries and other recommended libraries handle the repetitive boilerplate.

Consider how to make each part of your app testable in isolation.

- For example, having a well-defined API for fetching data from the network makes it easier to test the module that persists that data in a local database.

- If instead, you mix the logic from these two modules in one place, or distribute your networking code across your entire code base, it becomes much more difficult—if not impossible—to test effectively.

Persist as much relevant and fresh data as possible.

- That way, users can enjoy your app's functionality even when their device is in offline mode.

- Remember that not all of your users enjoy constant, high-speed connectivity—and even if they do, they can get bad reception in crowded places.

Model – View - ViewModel

???

Introduction to MVVM on Android

- Making an Android app in itself is not all that hard once you get the basics right. Making a **maintainable** app is a whole different story.

- You have to give your code a firm structure, prevent yourself from putting all the code inside an activity or fragment and make many smaller classes which have a single responsibility.

Structure

~~Code heavy activity~~

Single Responsibility

MVC

MVP

MVVM

- How can you achieve all of this? Architectural patterns! MVC, MVP, **MVVM**,… While anything is better than the dreaded “spaghetti code”, MVVM is one of the best options for Android development. It’s even fully supported and encouraged by Google with their first-party libraries.

Model – View - ViewModel

???

The meaning of Model-View-ViewModel

- Separation of concerns is a beautiful thing and every single design pattern tries to do the best that it can to achieve it.

- In case of MVVM, there are 3 inherent parts which help in accomplishing the separation of concerns: models, views and view models. You can also add a repository which acts as a **single source of truth** for all the data – more on that later.

Separation of concerns

Model

View

ViewModel

+Respository

View

ONLY User interface

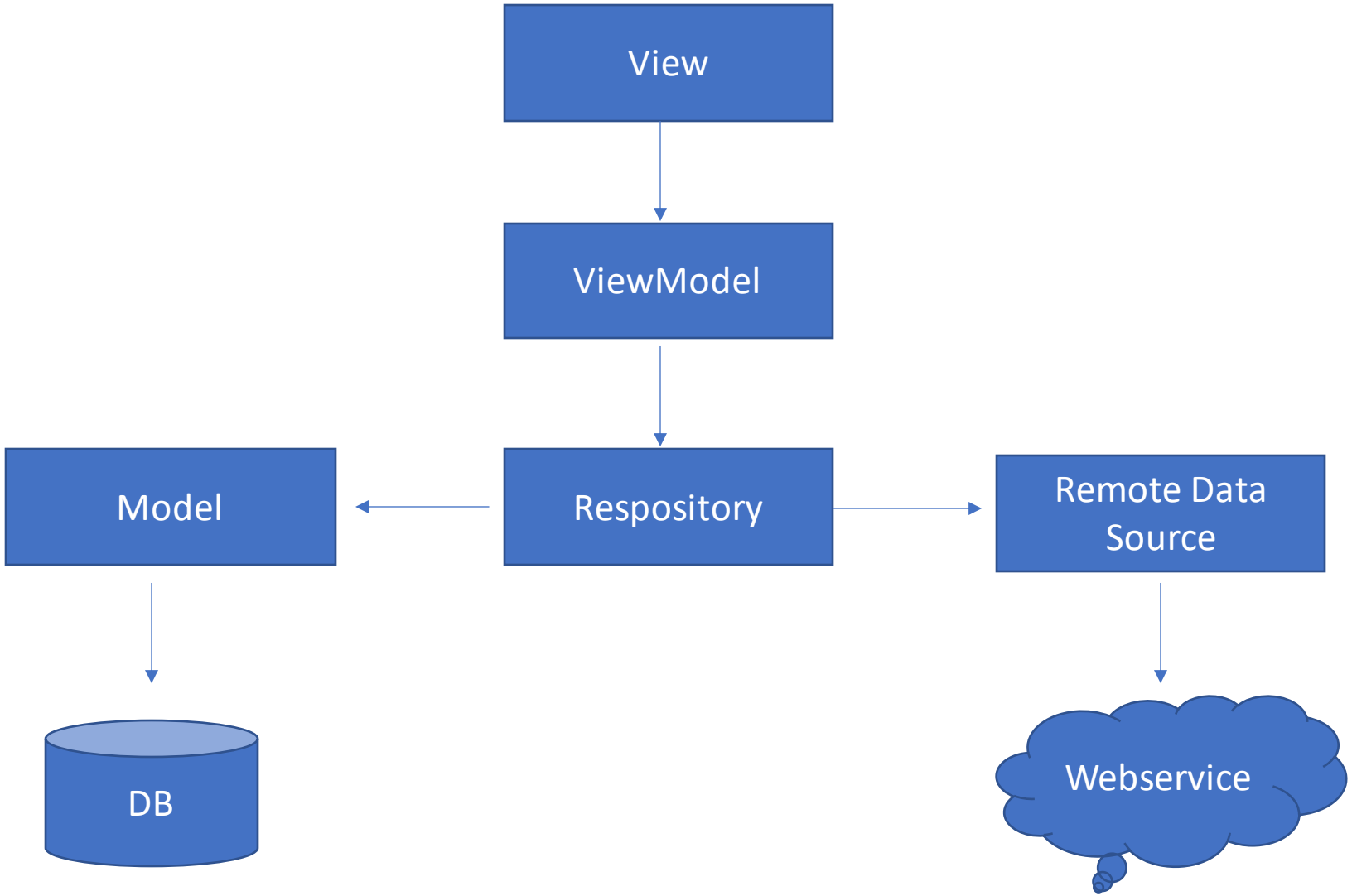
View

~~Business logic~~

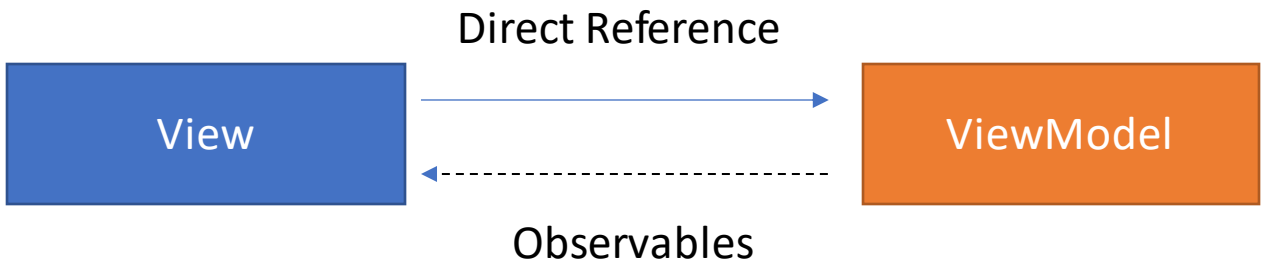
View

ViewModel

ViewModel


View
ViewModel
Respository
Model
Remote Data Source
DB
Webservice

ViewModel



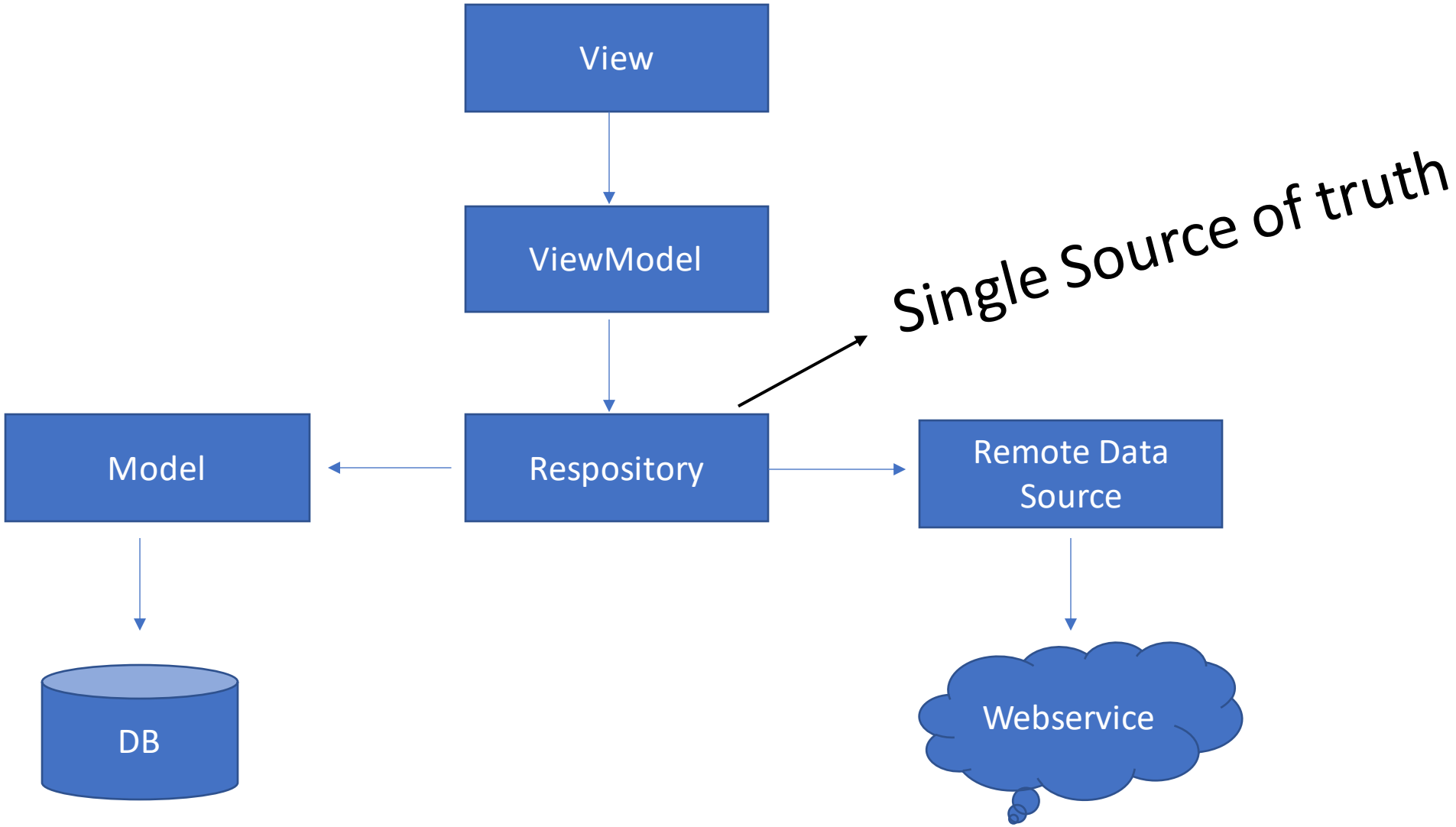
This can be done through LiveData which is a handy lifecycle aware library for creating observables. One of its advantages is that it automatically doesn't notify the observer if its activity or fragment is already destroyed, leaving you free from managing the lifecycle yourself.

Model

- Model is where you put all the business specific code. While technically there is an intermediate step between the ViewModel and the Model in the form of a Repository, you can kind of regard everything from Repository downwards as its own group of classes far away from the user interface. These operate on your app's data and fetch it from the local database or from the network.
- Repository has a special role of being a mediator between local storage and the server. This is where you check whether the remote data should be cached locally and so on. Repository is also the **single source of truth** for ViewModels. In other words, when ViewModel wants some data, it gets it from the Repository. Then it's up to the repository to decide what to do next. As far as the ViewModel is concerned, the data could be fetched from a toaster or from a supercomputer, it doesn't care – that's the business of the repository.

LiveData

ViewModel

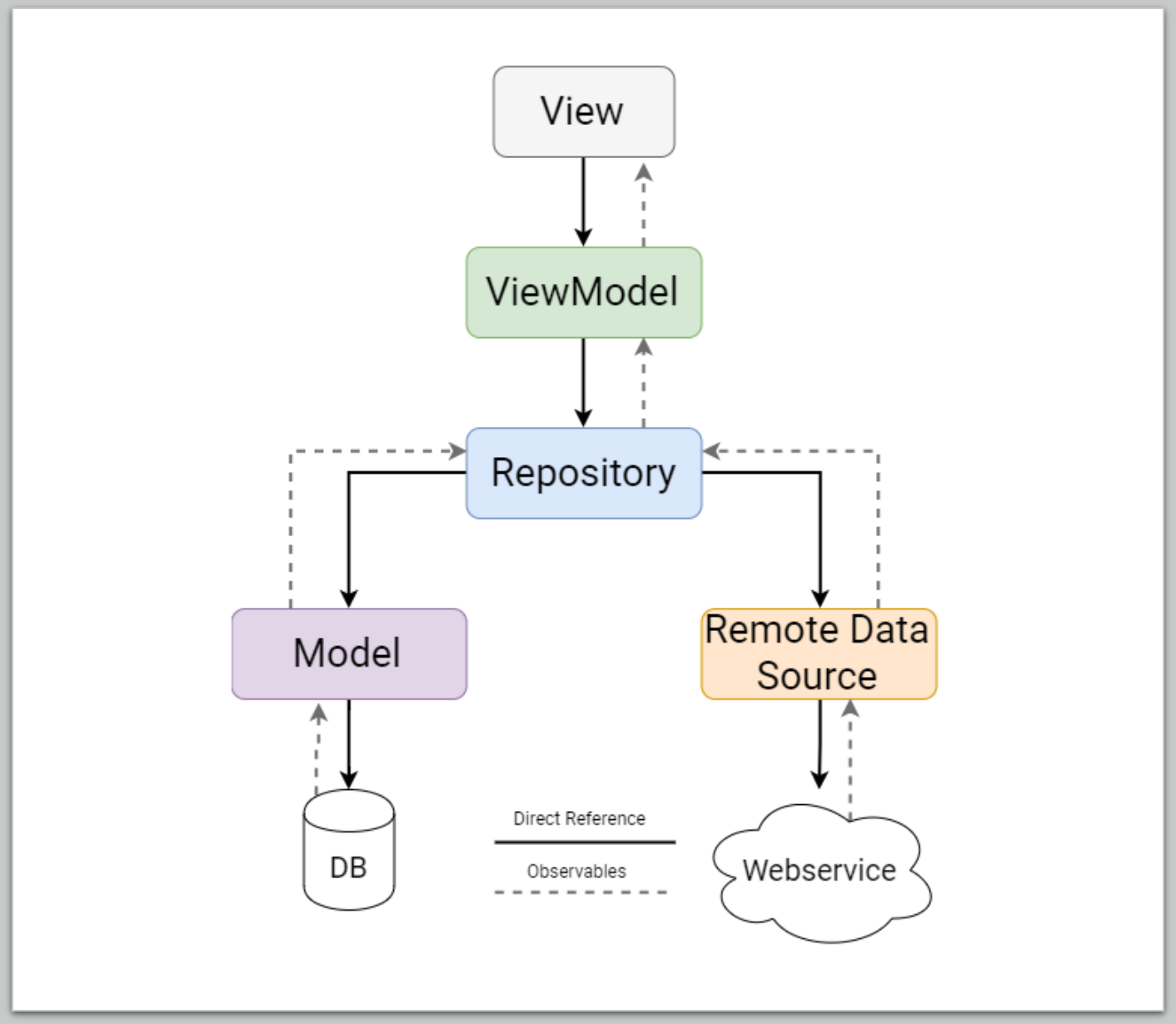

View
ViewModel
Single Source of truth
Model
Respository
Remote Data Source
DB
Webservice

Connectedness of MVVM components

- Not only that the View observes data in the ViewModel but also the ViewModel observes data in the Repository which in turn observes data coming from the local database and from the remote data source.
- To put this all into perspective, you can think of the connections between models, views, view models, repositories and other classes in the following manner.
- When traversing down the hierarchy, the **upper class has a direct reference to its child**. On the other hand, the child doesn't have a reference to its parent. **Children only expose some data by allowing it to be observed** through LiveData or any other library if you so desire.

ViewModel

- For a better imagination have a look at the arrow-cluttered diagram. I wanted to spare you of the non-necessary clutter at the start, that's why those observable arrows weren't present in the first diagram showing MVVM.



- The last important thing to mention here is that you should always **adhere to the reference tree above**.

- For example, don’t make your ViewModel get data from the database directly while bypassing the Repository! Everything has it’s purpose and makes the code modular, easy to maintain and nice to read.

- You will read your code many more times than you write it so make readability the number one priority. Don’t be lazy to create abstractions, you will thank yourself later.