Guide to app architecture

This guide encompasses best practices and recommended architecture (#recommended-app-arch) for building robust, high-quality apps.

Note: This page assumes a basic familiarity with the Android Framework. If you are new to Android app development, check out the Android Basics course (/courses/android-basics-kotlin/course) to get started and learn more about the concepts mentioned in this guide.

Mobile app user experiences

A typical Android app contains multiple app components (/guide/components/fundamentals#components), including activities (/guide/components/activities/intro-activities), fragments (/guide/fragments), services (/guide/components/services), content providers (/guide/topics/providers/content-providers), and broadcast receivers (/guide/components/broadcasts). You declare most of these app components in your app manifest (/guide/topics/manifest/manifest-intro). 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 (https://en.wikipedia.org/wiki/Separation_of_concerns). It's a common mistake to write all your code in an `Activity` (/reference/android/app/Activity) or a `Fragment` (/reference/android/app/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.

Single source of truth

When a new data type is defined in your app, you should assign a Single Source of Truth (SSOT) to it. The SSOT is the *owner* of that data, and only the SSOT can modify or mutate it. To achieve this, the SSOT exposes the data using an immutable type, and to modify the data, the SSOT exposes functions or receive events that other types can call.

This pattern brings multiple benefits:

- It centralizes all the changes to a particular type of data in one place.
- It protects the data so that other types cannot tamper with it.
- It makes changes to the data more traceable. Thus, bugs are easier to spot.

In an offline-first application, the source of truth for application data is typically a database. In some other cases, the source of truth can be a ViewModel or even the UI.

Unidirectional Data Flow

The single source of truth principle (#single-source-of-truth) is often used in our guides with the Unidirectional Data Flow (UDF) pattern. In UDF, **state** flows in only one direction. The **events** that modify the data flow in the opposite direction.

In Android, state or data usually flow from the higher-scoped types of the hierarchy to the lower-scoped ones. Events are usually triggered from the lower-scoped types until they reach the SSOT for the corresponding data type. For example, application data usually flows from data sources to the UI. User events such as button presses flow from the UI to the SSOT where the application data is modified and exposed in an immutable type.

This pattern better guarantees data consistency, is less prone to errors, is easier to debug and brings all the benefits of the SSOT pattern.

Recommended app architecture

This section demonstrates how to structure your app following recommended best practices.

Note: The recommendations and best practices present in this page can be applied to a broad spectrum of apps to allow them to scale, improve quality and robustness, and make them easier to test. However, you should treat them as guidelines and adapt them to your requirements as needed.

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.

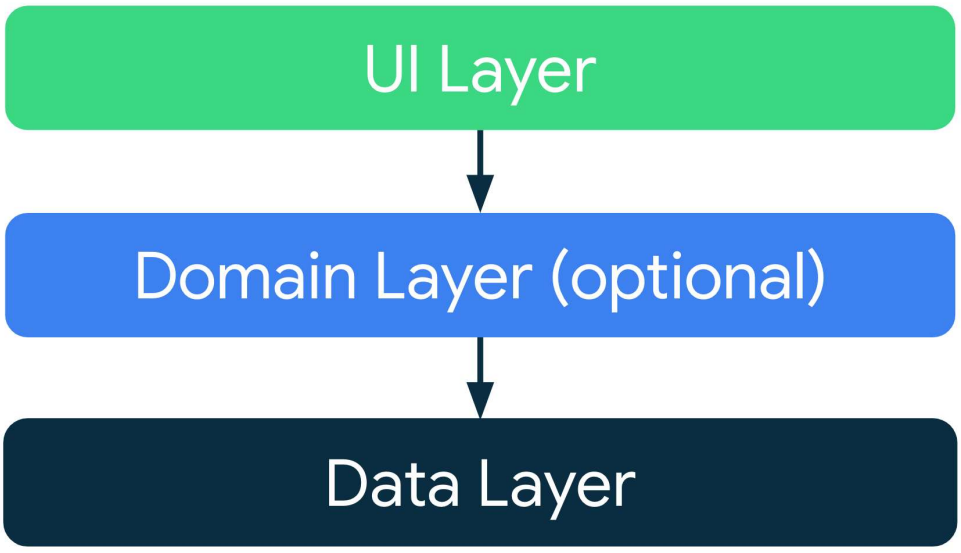


Figure 1. *Diagram of a typical app architecture.*

Note: The arrows in the diagrams in this guide represent dependencies between classes. For example, the domain layer depends on data layer classes.

Modern App Architecture

This *Modern App Architecture* encourages using the following techniques, among others:

- A reactive and layered architecture.
- Unidirectional Data Flow (UDF) in all layers of the app.
- A UI layer with state holders to manage the complexity of the UI.
- Coroutines and flows.
- Dependency injection best practices.

For more information, see the following sections, the other Architecture pages in the table of contents, and the recommendations page (/topic/architecture/recommendations) that contains a summary of the most important best practices.

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 (/jetpack/compose) functions.
- State holders (such as ViewModel (/topic/libraries/architecture/viewmodel) classes) that hold data, expose it to the UI, and handle logic.

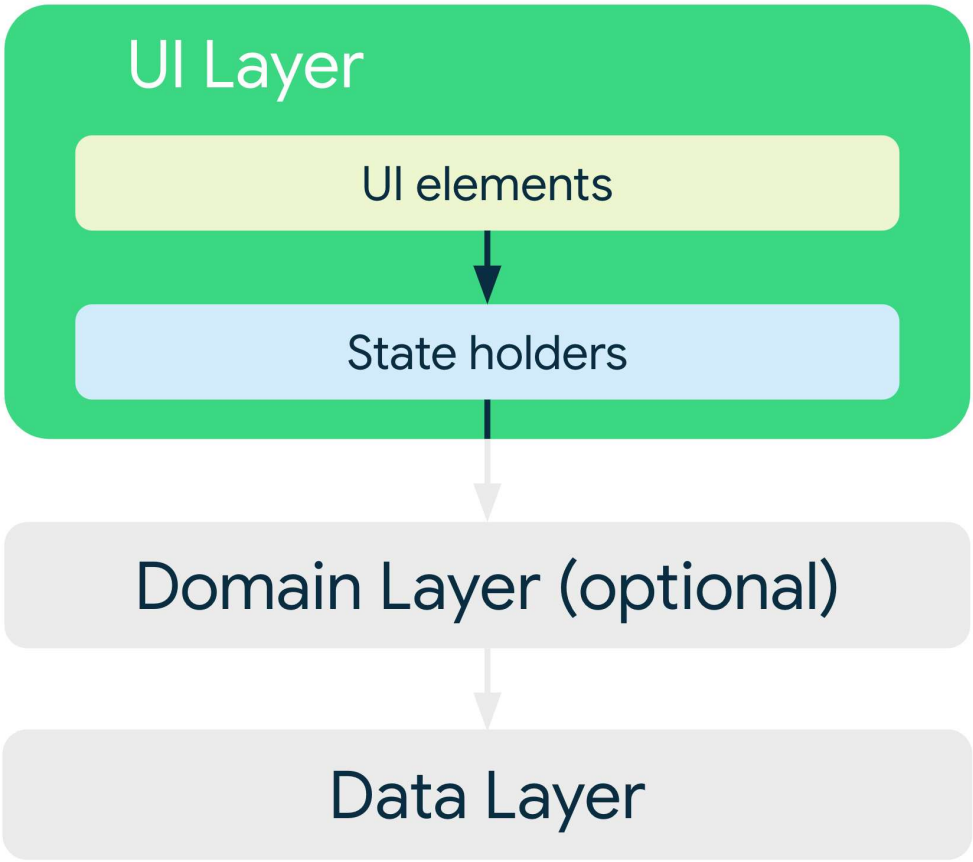


Figure 2. *The UI layer's role in app architecture.*

To learn more about this layer, see the UI layer page (/jetpack/guide/ui-layer).

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.

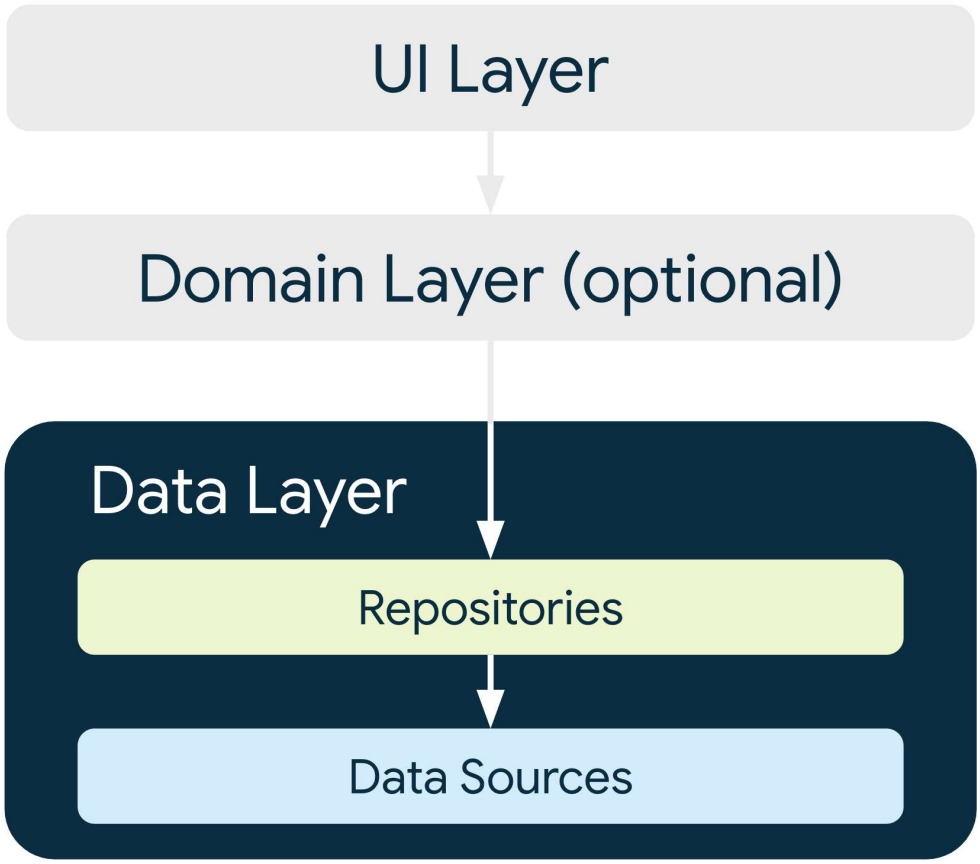


Figure 3. *The data layer's role in app architecture.*

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.

To learn more about this layer, see the data layer page (/jetpack/guide/data-layer).

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.

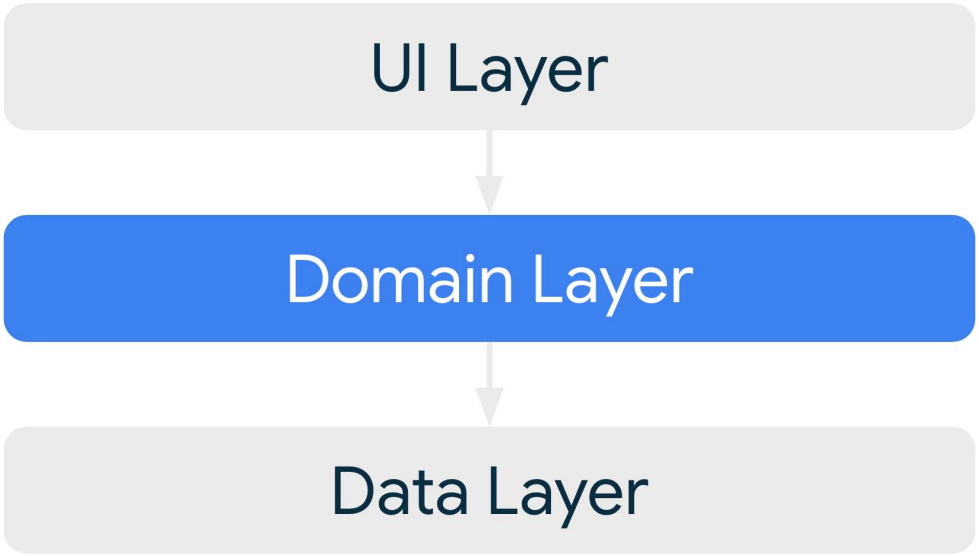


Figure 4. *The domain layer's role in app architecture.*

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.

To learn more about this layer, see the domain layer page (/jetpack/guide/domain-layer).

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) (/training/dependency-injection): Dependency injection allows classes to define their dependencies without constructing them. At runtime, another class is responsible for providing these dependencies.

- Service locator (https://en.wikipedia.org/wiki/Service_locator_pattern): 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.

We recommend following dependency injection patterns and using the Hilt library (/training/dependency-injection/hilt-android) **in Android apps.** Hilt automatically constructs objects by walking the dependency tree, provides compile-time guarantees on dependencies, and creates dependency containers for Android framework classes.

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` (/reference/android/content/Context), or `Toast` (/guide/topics/ui/notifiers/toasts). Abstracting other classes in your app away from them helps with testability and reduces coupling (https://en.wikipedia.org/wiki/Coupling_(computer_programming)) 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. Following the recommended app architecture (#recommended-app-arch) will help you with this.

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.

Types are responsible for their concurrency policy.

If a type is performing long-running blocking work, it should be responsible for moving that computation to the right thread. That particular type knows the type of computation that it is doing and in which thread it should be executed. Types should be main-safe, meaning they're safe to call from the main thread without blocking it.

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.

Benefits of Architecture

Having a good Architecture implemented in your app brings a lot of benefits to the project and engineering teams:

- It improves the maintainability, quality and robustness of the overall app.
- It allows the app to scale. More people and more teams can contribute to the same codebase with minimal code conflicts.
- It helps with onboarding. As Architecture brings consistency to your project, new members of the team can quickly get up to speed and be more efficient in less amount of time.
- It is easier to test. A good Architecture encourages simpler types which are generally easier to test.
- Bugs can be investigated methodically with well defined processes.

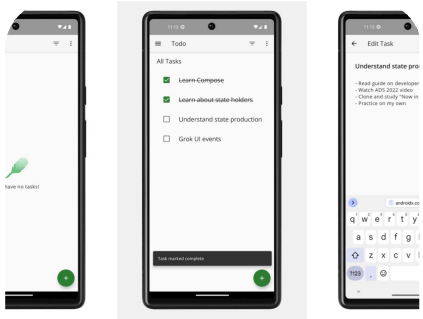
Investing in Architecture also has a direct impact in your users. They benefit from a more stable application, and more features due to a more productive engineering team. However, Architecture also requires an up-front time investment. To help you justify this time to the rest of your company, take a look at these case studies (/quality) where other companies share their success stories when having a good architecture in their app.

Samples

The following Google samples demonstrate good app architecture. Go explore them to see this guidance in practice:

ttps://github.com/android/com
pose-
samples/tree/main/Jetcaster)

(https://github.com/android/arch
itecture-
templates/tree/multimodule)



GITHUB

Jetcaster sample

(https://github.com/android/compose-samples/tree/main/Jetcaster)

Jetcaster is a sample podcast app, built with

GITHUB

Architecture starter template (multi-module)

(https://github.com/android/architecture-templates/tree/multimodule)

This template is compatible

GITHUB

Architecture

(https://github.com/android/architecture-samples/tree/main)

These samples showcase different architectural approaches to developing Android apps. In its differe...

More

Last updated 2025-02-10 UTC.