Guide to app architecture

Contents

The following pages demonstrate how to build an app using best practices. The recommendations in this guide can be applied to most apps, making them easier to scale, test, and maintain. However, they're guidelines, not steadfast rules, and you should adapt them to your unique requirements.

This section provides a high-level overview of how Flutter applications can be architected. It explains the layers of an application, along with the classes that exist within each layer. The section after this provides concrete code samples and walks through a Flutter application that's implemented these recommendations.

Overview of project structure

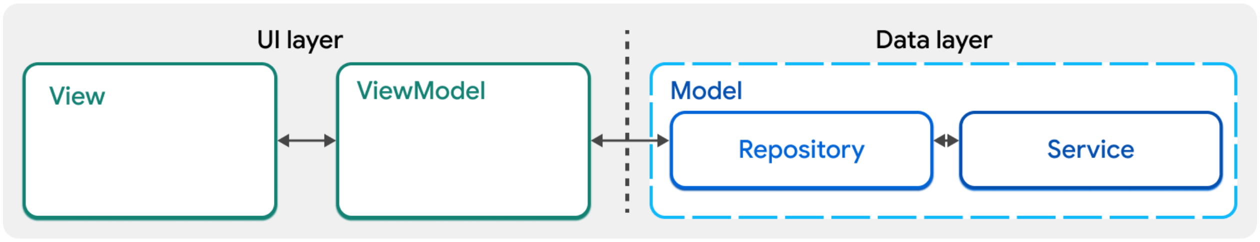
Separation-of-concerns is the most important principle to follow when designing your Flutter app. Your Flutter application should split into two broad layers, the UI layer and the Data layer.

Each layer is further split into different components, each of which has distinct responsibilities, a well-defined interface, boundaries and dependencies. This guide recommends you split your application into the following components:

- Views
- View models
- Repositories
- Services

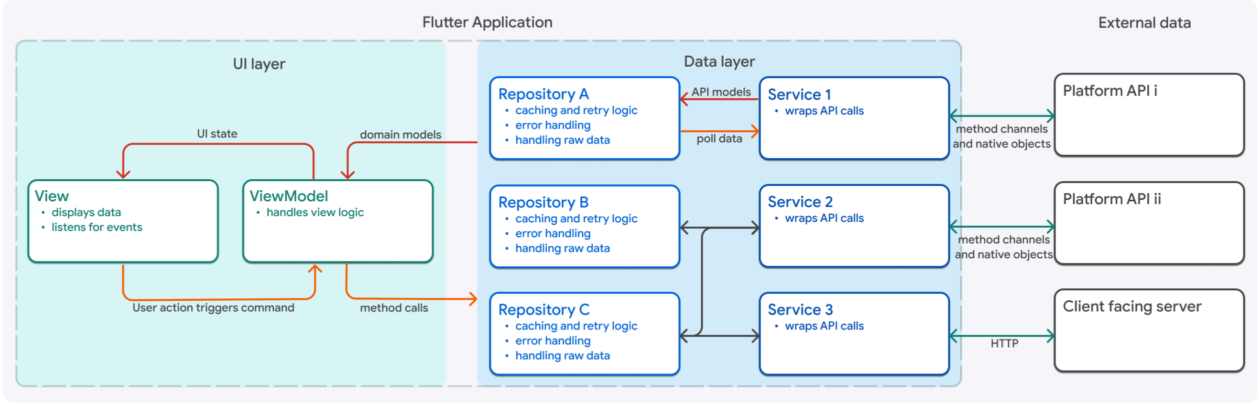
MVVM

If you've encountered the Model-View-ViewModel design pattern (MVVM), this will be familiar. MVVM is a design pattern that separates a feature of an application into three parts: the `Model`, the `ViewModel` and the `View`. Views and view models make up the UI layer of an application. Repositories and services represent the data of an application, or the model layer of MVVM. Each of these components is defined in the next section.

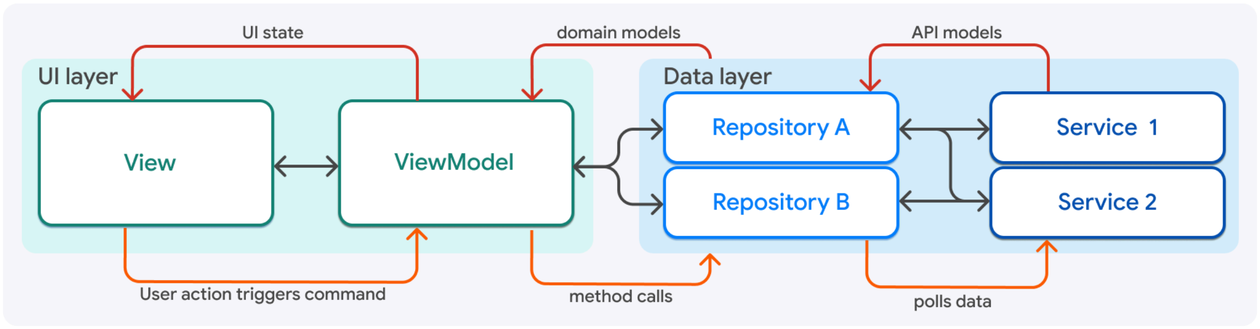


Every feature in an application will contain one view to describe the UI and one view model to handle logic, one or more repositories as the sources of truth for your application data, and zero or more services that interact with external APIs, like client servers and platform plugins.

A single feature of an application might require all of the following objects:



Each of these objects and the arrows that connect them will be explained thoroughly by the end of this page. Throughout this guide, the following simplified version of that diagram will be used as an anchor.



> **Note**
>
> Apps with complex logic might also have a logic layer that sits in between the UI layer and data layer. This logic layer is commonly called the *domain layer.* The domain layer contains additional components called often interactors or use-cases. The domain layer is covered later in this guide.

UI layer

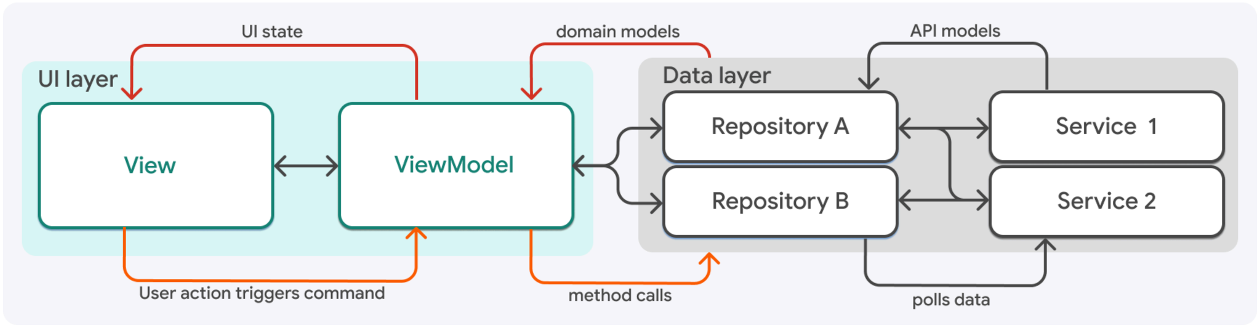
An application's UI layer is responsible for interacting with the user. It displays an application's data to the user and receives user input, such as tap events and form inputs.

The UI reacts to data changes or user input. When the UI receives new data from a Repository, it should re-render to display that new data. When the user interacts with the UI, it should change to reflect that interaction.

The UI layer is made up of two architectural components, based on the MVVM design pattern:

- **Views** describe how to present application data to the user. Specifically, it refers to a *composition of widgets *that make a feature. For instance, a view is often (but not always) a screen that has a `Scaffold` widget, along with all of the widgets below it in the widget tree. Views are also responsible for passing events to the view model in response to user interaction.
- **View models** contain the logic that converts app data into *UI State*, because data from repositories is often formatted differently from the data that needs to be displayed. For example, you might need to combine data from multiple repositories, or you might want to filter a list of data records.

Views and view models should have a 1:1 relationship.



In the simplest terms, a view model manages the UI state and the view displays that state. Using views and view models, your UI layer can maintain state during configuration changes (such as screen rotations), and you can test the logic of your UI independently of Flutter widgets.

ⓘ Note

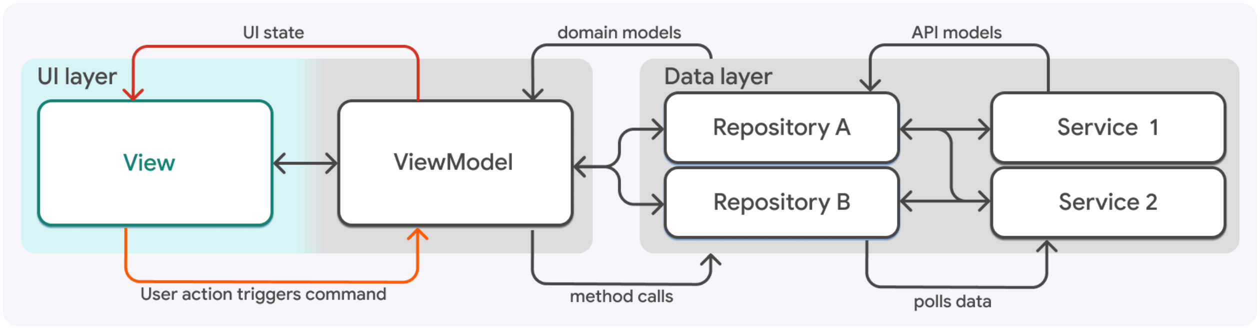
'View' is an abstract term, and one view doesn't equal one widget. Widgets are composable, and several can be combined to create one view. Therefore, view models don't have a 1-to-1 relationship with widgets, but rather a 1-to-1 relation with a *collection* of widgets.

A feature of an application is user centric, and therefore defined by the UI layer. Every instance of a pair of view and view model defines one feature in your app. This is often a screen in your app, but it doesn't have to be. For example, consider logging in and out. Logging in is generally done on a specific screen whose only purpose is to provide the user with a way to log in. In the application code, the login screen would be made up of a `LoginViewModel` class and a `LoginView` class.

On the other hand, logging out of an app is generally not done on a dedicated screen. The ability to log out is generally presented to the user as a button in a menu, a user account screen, or any number of different locations. It's often presented in multiple locations. In that scenario, you might have a `LogoutViewModel` and a `LogoutView` which only contains a single button that can be dropped into other widgets.

Views

In Flutter, views are the widget classes of your application. Views are the primary method of rendering UI, and shouldn't contain any business logic. They should be passed all data they need to render from the view model.



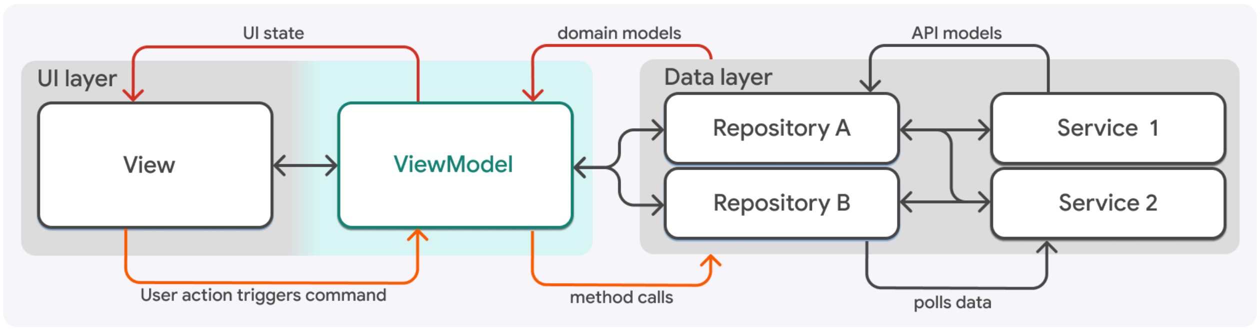
The only logic a view should contain is:

- Simple if-statements to show and hide widgets based on a flag or nullable field in the view model
- Animation logic
- Layout logic based on device information, like screen size or orientation.
- Simple routing logic

All logic related to data should be handled in the view model.

View models

A view model exposes the application data necessary to render a view. In the architecture design described on this page, most of the logic in your Flutter application lives in view models.



A view model's main responsibilities include:

- Retrieving application data from repositories and transforming it into a format suitable for presentation in the view. For example, it might filter, sort or aggregate data.
- Maintaining the current state needed in the view, so that the view can rebuild without losing data. For example, it might contain boolean flags to conditionally render widgets in the view, or a field that tracks which section of a carousel is active on screen.
- Exposes callbacks (called **commands**) to the view that can be attached to an event handler, like a button press or form submission.

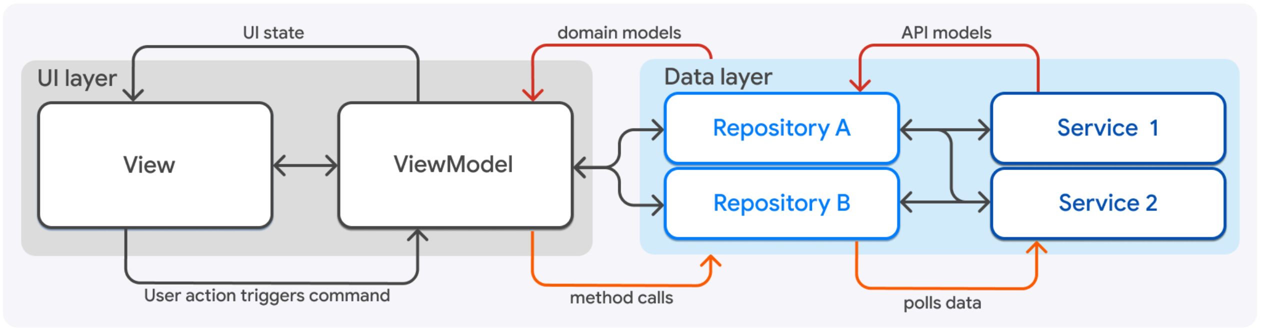
Commands are named for the command pattern, and are Dart functions that allow the views to execute complex logic without knowledge of its implementation. Commands are written as members of the view model class to be called by the gesture handlers in the view class.

You can find examples of views, view models, and commands on the UI layer portion of the App architecture case study.

For a gentle introduction to MVVM in Flutter, check out the state management fundamentals.

Data layer

The data layer of an app handles your business data and logic. Two pieces of architecture make up the data layer: services and repositories. These pieces should have well defined inputs and outputs to simplify their reusability and testability.



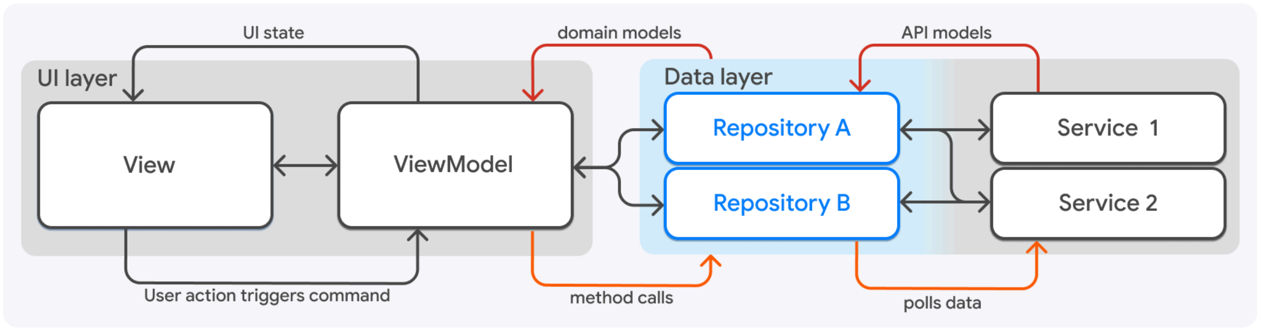
Using MVVM language, services and repositories make up your *model layer*.

Repositories

Repository classes are the source of truth for your model data. They're responsible for polling data from services, and transforming that raw data into **domain models**. Domain models represent the data that the application needs, formatted in a way that your view model classes can consume. There should be a repository class for each different type of data handled in your app.

Repositories handle the business logic associated with services, such as:

- Caching
- Error handling
- Retry logic
- Refreshing data
- Polling services for new data
- Refreshing data based on user actions



Repositories output application data as domain models. For example, a social media app might have a `UserProfileRepository` class that exposes a `Stream<UserProfile?>`, which emits a new value whenever the user signs in or out.

The models output by repositories are consumed by view models. Repositories and view models have a many-to-many relationship. A view model can use many repositories to get the data it needs, and a repository can be used by many view models.

Repositories should never be aware of each other. If your application has business logic that needs data from two repositories, you should combine the data in the view model or in the domain layer, especially if your repository-to-view-model relationship is complex.

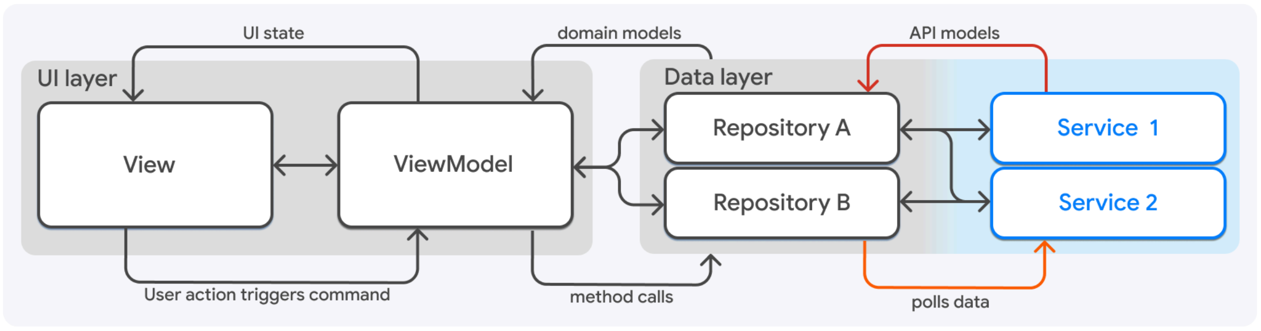
Services

Services are in the lowest layer of your application. They wrap API endpoints and expose asynchronous response objects, such as `Future` and `Stream` objects. They're only used to isolate data-loading, and they hold no state. Your app should have one service class per data source. Examples of endpoints that services might wrap include:

- The underlying platform, like iOS and Android APIs
- REST endpoints
- Local files

As a rule of thumb, services are most helpful when the necessary data lives outside of your application's Dart code - which is true of each of the preceding examples.

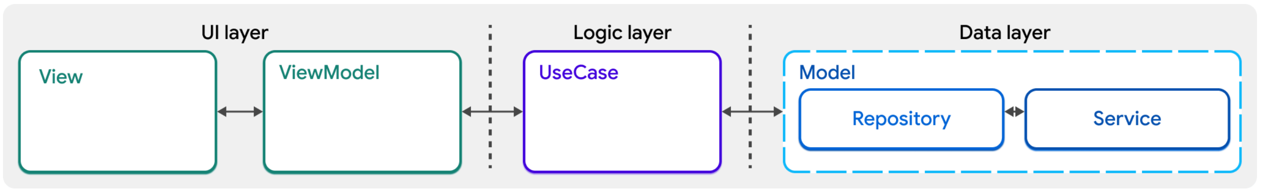
Services and repositories have a many-to-many relationship. A single Repository can use several services, and a service can be used by multiple repositories.



Optional: Domain layer

As your app grows and adds features, you may need to abstract away logic that adds too much complexity to your view models. These classes are often called interactors or **use-cases**.

Use-cases are responsible for making interactions between the UI and Data layers simpler and more reusable. They take data from repositories and make it suitable for the UI layer.



Use-cases are primarily used to encapsulate business logic that would otherwise live in the view model and meets one or more of the following conditions:

1. Requires merging data from multiple repositories
2. Is exceedingly complex
3. The logic will be reused by different view models

This layer is optional because not all applications or features within an application have these requirements. If you suspect your application would benefit from this additional layer, consider the pros and cons:

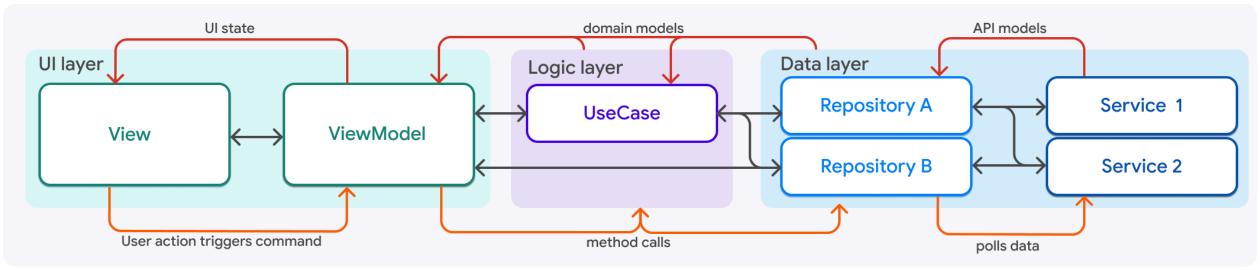
Pros	Cons
✅ Avoid code duplication in view models	✖ Increases complexity of your architecture, adding more classes and higher cognitive load
✅ Improve testability by separating complex business logic from UI logic	✖ Testing requires additional mocks
✅ Improve code readability in view models	✖ Adds additional boilerplate to your code

Data access with use-cases

Another consideration when adding a Domain layer is whether view models will continue to have access to repository data directly, or if you'll enforce view models to go through use-cases to get their data. Put another way, will you add use-cases as you need them? Perhaps when you notice repeated logic in your view models? Or, will you create a use-case each time a view model needs data, even if the logic in the use-case is simple?

If you choose to do the latter, it intensifies the earlier outlined pros and cons. Your application code will be extremely modular and testable, but it also adds a significant amount of unnecessary overhead.

A good approach is to add use-cases only when needed. If you find that your view models are accessing data through use-cases most of the time, you can always refactor your code to utilize use-cases exclusively. The example app used later in this guide uses use-cases for some features, but also has view models that interact with repositories directly. A complex feature may ultimately end up looking like this:



This method of adding use-cases is defined by the following rules:

- Use-cases depend on repositories
- Use-cases and repositories have a many-to-many relationship
- View models depend on one or more use-cases *and* one or more repositories

This method of using use-cases ends up looking less like a layered lasagna, and more like a plated dinner with two mains (UI and data layers) and a side (domain layer). Use-cases are just utility classes that have well-defined inputs and outputs. This approach is flexible and extendable, but it requires greater diligence to maintain order.

Feedback

As this section of the website is evolving, we welcome your feedback!

〈 Common architecture concepts

Architecture case study 〉