Common architecture concepts

Contents

In this section, you'll find tried and true principles that guide architectural decisions in the larger world of app development, as well as information about how they fit into Flutter specifically. It's a gentle introduction to vocabulary and concepts related to the recommended architecture and best practices, so they can be explored in more detail throughout this guide.

Separation of concerns

Separation-of-concerns is a core principle in app development that promotes modularity and maintainability by dividing an application's functionality into distinct, self-contained units. From a high-level, this means separating your UI logic from your business logic. This is often described as *layered* architecture. Within each layer, you should further separate your application by feature or functionality. For example, your application's authentication logic should be in a different class than the search logic.

In Flutter, this applies to widgets in the UI layer as well. You should write reusable, lean widgets that hold as little logic as possible.

Layered architecture

Flutter applications should be written in *layers*. Layered architecture is a software design pattern that organizes an application into distinct layers, each with specific roles and responsibilities. Typically, applications are separated into 2 to 3 layers, depending on complexity.

- **UI layer** - Displays data to the user that is exposed by the business logic layer, and handles user interaction. This is also commonly referred to as the 'presentation layer'.
- **Logic layer** - Implements core business logic, and facilitates interaction between the data layer and UI layer. Commonly known as the 'domain layer'. The logic layer is optional, and only needs to be implemented if your application has complex business logic that happens on the client. Many apps are only concerned with presenting data to a user and allowing the user to change that data (colloquially known as CRUD apps). These apps might not need this optional layer.
- **Data layer** - Manages interactions with data sources, such as databases or platform plugins. Exposes data and methods to the business logic layer.

These are called 'layers' because each layer can only communicate with the layers directly below or above it. The UI layer shouldn't know that the data layer exists, and vice versa.

Single source of truth

Every data type in your app should have a single source of truth (SSOT). The source of truth is responsible for representing local or remote state. If the data can be modified in the app, the SSOT class should be the only class that can do so.

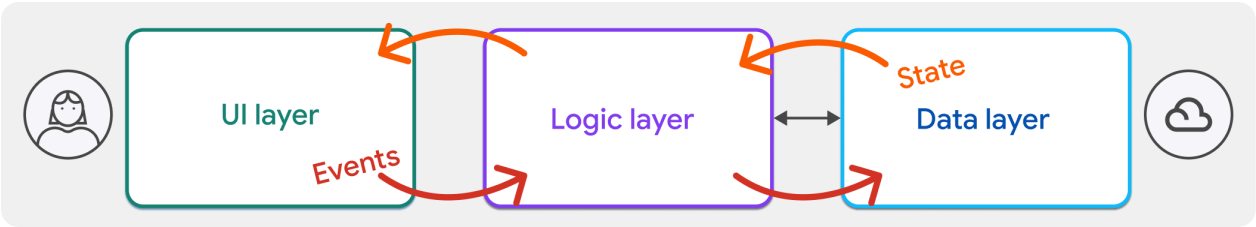
This can dramatically reduce the number of bugs in your application, and it can simply code because you'll only ever have one copy of the same data.

Generally, the source of truth for any given type of data in your application is held in a class called a **Repository**, which is part of the data layer. There is typically one repository class for each type of data in your app.

This principle can be applied across layers and components in your application as well as within individual classes. For example, a Dart class might use getters to derive values from an SSOT field (instead of having multiple fields that need to be updated independently) or a list of records to group related values (instead of parallel lists whose indices might get out of sync).

Unidirectional data flow

Unidirectional data flow (UDF) refers to a design pattern that helps decouple state from the UI that displays that state. In the simplest terms, state flows from the data layer through the logic layer and eventually to the widgets in the UI layer. Events from user-interaction flow the opposite direction, from the presentation layer back through the logic layer and to the data layer.



In UDF, the update loop from user interaction to re-rendering the UI looks like this:

1. [UI layer] An event occurs due to user interaction, such as a button being clicked. The widget's event handler callback invokes a method exposed by a class in the logic layer.
2. [Logic layer] The logic class calls methods exposed by a repository that know how to mutate the data.
3. [Data layer] The repository updates data (if necessary) and then provides the new data to the logic class.
4. [Logic layer] The logic class saves its new state, which it sends to the UI.
5. [UI layer] The UI displays the new state of the view model.

New data can also start at the data layer. For example, a repository might poll an HTTP server for new data. In this case, the data flow only makes the second half of the journey. The most important idea is that data changes always happen in the SSOT, which is the data layer. This makes your code easier to understand, less error prone, and prevents malformed or unexpected data from being created.

UI is a function of (immutable) state

Flutter is declarative, meaning that it builds its UI to reflect the current state of your app. When state changes, your app should trigger a rebuild of the UI that depends on that state. In Flutter, you'll often hear this described as "UI is a function of state".

$$UI = f(state)$$

It's crucial that your data drive your UI, and not the other way around. Data should be immutable and persistent, and views should contain as little logic as possible. This minimizes the possibility of data being lost when an app is closed, and makes your app more testable and resilient to bugs.

Extensibility

Each piece of architecture should have a well defined list of inputs and outputs. For example, a view model in the logic layer should only take in data sources as inputs, such as repositories, and should only expose commands and data formatted for views.

Using clean interfaces in this way allows you to swap out concrete implementations of your classes without needing to change any of the code that consumes the interface.

Testability

The principles that make software extensible also make software easier to test. For example, you can test the self-contained logic of a view model by mocking a repository. The view model tests don't require you to mock other parts of your application, and you can test your UI logic separate from Flutter widgets themselves.

Your app will also be more flexible. It will be straightforward and low risk to add new logic and new UI. For example, adding a new view model cannot break any logic from the data or business logic layers.

The next section explains the idea of inputs and outputs for any given component in your application's architecture.

Feedback

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

〈 Architecting Flutter apps

Guide to app architecture 〉