Architecture recommendations and resources

Contents

This page presents architecture best practices, why they matter, and whether we recommend them for your Flutter application. You should treat these recommendations as recommendations, and not steadfast rules, and you should adapt them to your app's unique requirements.

The best practices on this page have a priority, which reflects how strongly the Flutter team recommends it.

- **Strongly recommend:** You should always implement this recommendation if you're starting to build a new application. You should strongly consider refactoring an existing app to implement this practice unless doing so would fundamentally clash with your current approach.
- **Recommend**: This practice will likely improve your app.
- **Conditional**: This practice can improve your app in certain circumstances.

Separation of concerns

You should separate your app into a UI layer and a data layer. Within those layers, you should further separate logic into classes by responsibility.

Recommendation	Description
Use clearly defined data and UI layers. Strongly recommend	Separation of concerns is the most important architectural principle. The data layer exposes application data to the rest of the app, and contains most of the business logic in your application. The UI layer displays application data and listens for user events from users. The UI layer contains separate classes for UI logic and widgets.
Use the repository pattern in the data layer. Strongly recommend	The repository pattern is a software design pattern that isolates the data access logic from the rest of the application. It creates an abstraction layer between the application's business logic and the underlying data storage mechanisms (databases, APIs, file systems, etc.). In practice, this means creating Repository classes and Service classes.
Use ViewModels and Views in the UI layer. (MVVM) Strongly recommend	Separation of concerns is the most important architectural principle. This particular separation makes your code much less error prone because your widgets remain "dumb".
Use `ChangeNotifiers` and `Listenables` to handle widget updates. Conditional	The `ChangeNotifier` API is part of the Flutter SDK, and is a convenient way to have your widgets observe changes in your ViewModels. There are many options to handle state-management, and ultimately the decision comes down to personal preference. Read about our ChangeNotifier recommendation or other popular options.

Do not put logic in widgets. Strongly recommend	Logic should be encapsulated in methods on the ViewModel. 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 ViewModel • Animation logic that relies on the widget to calculate • Layout logic based on device information, like screen size or orientation. • Simple routing logic
Use a domain layer. Conditional	A domain layer is only needed if your application has exceeding complex logic that crowds your ViewModels, or if you find yourself repeating logic in ViewModels. In very large apps, use-cases are useful, but in most apps they add unnecessary overhead. Use in apps with complex logic requirements.

Handling data

Handling data with care makes your code easier to understand, less error prone, and prevents malformed or unexpected data from being created.

Recommendation	Description
Use unidirectional data flow. Strongly recommend	Data updates should only flow from the data layer to the UI layer. Interactions in the UI layer are sent to the data layer where they're processed.
Use `Commands` to handle events from user interaction. Recommend	Commands prevent rendering errors in your app, and standardize how the UI layer sends events to the data layer. Read about commands in the architecture case study.
Use immutable data models. Strongly recommend	Immutable data is crucial in making sure that data only updates in the model.
Use freezed or built_value to generate immutable data models. Recommend	You can use packages to help generate useful functionality in your data models, freezed or built_value. These can generate common model methods like JSON ser/des, deep equality checking and copy methods. These code generation packages can add significant build time to your applications if you have a lot of models.
Create separate API models and domain models. Conditional	Using separate models adds verbosity, but prevents complexity in ViewModels and use-cases. Use in large apps.

App structure

Well organized code benefits both the health of the app itself, and the team working on the code.

Recommendation	Description

Use dependency injection. Strongly recommend	Dependency injection prevents your app from having globally accessible objects, which makes your code less error prone. We recommend you use the provider package to handle dependency injection.
Use go_router for navigation. Recommend	Go_router is the preferred way to write 90% of Flutter applications. There are some specific use-cases that go_router doesn't solve, in which case you can use the Flutter Navigator API directly or try other packages found on pub.dev.
Use standardized naming conventions for classes, files and directories. Recommend	We recommend naming classes for the architectural component they represent. For example, you may have the following classes: • HomeViewModel • HomeScreen • UserRepository • ClientApiService For clarity, we do not recommend using names that can be confused with objects from the Flutter SDK. For example, you should put your shared widgets in a directory called `ui/core/`, rather than a directory called `/widgets`.
Use abstract repository classes Strongly recommend	Repository classes are the sources of truth for all data in your app, and facilitate communication with external APIs. Creating abstract repository classes allows you to create different implementations, which can be used for different app environments, such as "development" and "staging".

Testing

Good testing practices makes your app flexible. It also makes it straightforward and low risk to add new logic and new UI.

Recommendation	Description
Test architectural components separately, and together. Strongly recommend	* Write unit tests for every service, repository and ViewModel class. These tests should test the logic of every method individually. • Write widget tests for views. Testing routing and dependency injection are particularly important.
Make fakes for testing (and write code that takes advantage of fakes.) Strongly recommend	Fakes aren't concerned with the inner workings of any given method as much as they're concerned with inputs and outputs. If you have this in mind while writing application code, you're forced to write modular, lightweight functions and classes with well defined inputs and outputs.

Recommended resources

- Code and templates
  - Compass app source code - Source code of a full-featured, robust Flutter application that implements many of these recommendations.
  - Flutter skeleton - A Flutter application template that includes many of these recommendations.
  - very_good_cli - A Flutter application template made by the Flutter experts Very Good Ventures. This template generates a similar app structure.
- Documentation
  - Very Good Engineering architecture documentation - Very Good Engineering is a documentation site by VGV that has technical articles, demos, and open-sourced projects. It includes documentation on architecting Flutter applications.
  - State Management with ChangeNotifier walkthrough - A gentle introduction into using the primitives in the Flutter SDK for your state management.

- Tooling
  - Flutter developer tools - DevTools is a suite of performance and debugging tools for Dart and Flutter.
  - flutter_lints - A package that contains the lints for Flutter apps recommended by the Flutter team. Use this package to encourage good coding practices across a team.

Feedback

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

〈 Architecture case study

Design patterns 〉