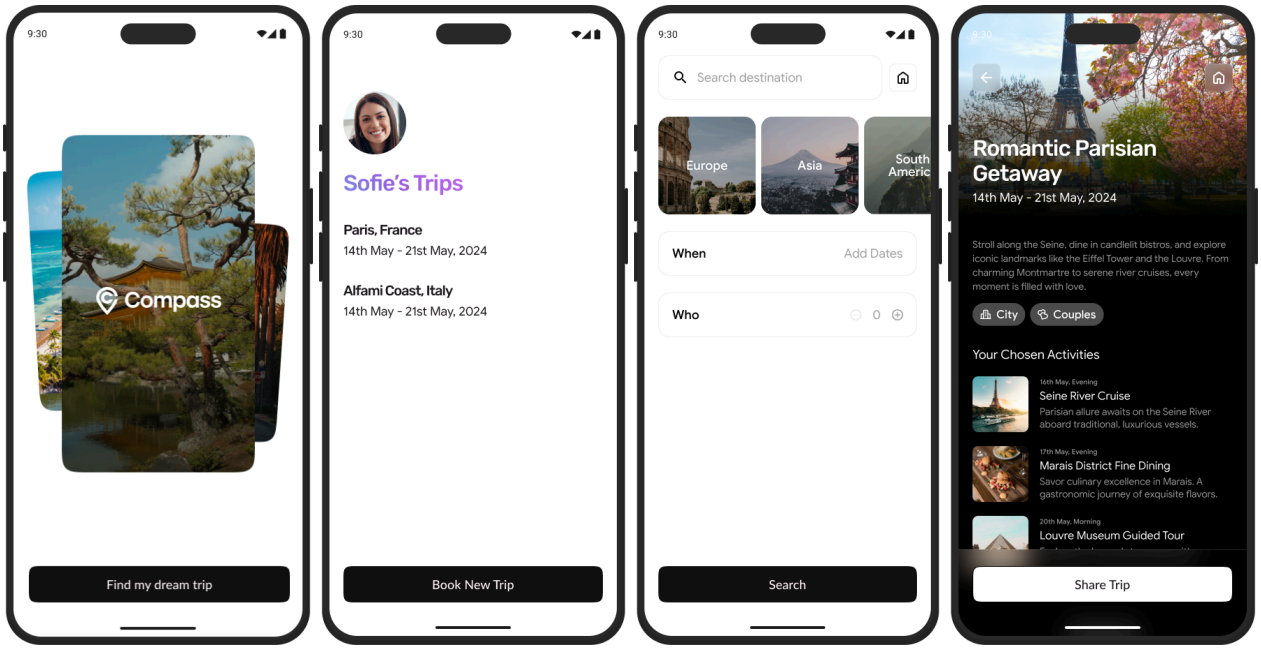
# Architecture case study



Contents



The code examples in this guide are from the Compass sample application, an app that helps users build and book itineraries for trips. It's a robust sample application with many features, routes, and screens. The app communicates with an HTTP server, has development and production environments, includes brand-specific styling, and contains high test coverage. In these ways and more, it simulates a real-world, feature-rich Flutter application.




The Compass app's architecture most resembles the MVVM design pattern as described in Flutter's app architecture guidelines. This architecture case study demonstrates how to implement those guidelines by walking through the "Home" feature of the compass app. If you aren't familiar with MVVM, you should read those guidelines first.

The Home screen of the Compass app displays user account information and a list of the user's saved trips. From this screen you can log out, open detailed trip pages, delete saved trips, and navigate to the first page of the core app flow, which allows the user to build a new itinerary.

In this case study, you'll learn the following:

- How to implement Flutter's app architecture guidelines using repositories and services in the data layer and the MVVM design pattern in the UI layer
- How to use the Command pattern to safely render UI as data changes
- How to use `ChangeNotifier` and `Listenable` objects to manage state
- How to implement Dependency Injection using `package:provider`
- How to set up tests when following the recommended architecture
- Effective package structure for large Flutter apps

This case-study was written to be read in order. Any given page might reference the previous pages.

The code examples in this case-study include all the details needed to understand the architecture, but they're not complete, runnable snippets. If you prefer to follow along with the full app, you can find it on GitHub.

# Package structure

Well-organized code is easier for multiple engineers to work on with minimal code conflicts and is easier for new engineers to navigate and understand. Code organization both benefits and benefits from well-defined architecture.

There are two popular means of organizing code:

1. By feature - The classes needed for each feature are grouped together. For example, you might have an `auth` directory, which would contain files like `auth_viewmodel.dart`, `login_usecase.dart`, `logout_usecase.dart`, `login_screen.dart`, `logout_button.dart`, etc.
2. By type - Each "type" of architecture is grouped together. For example, you might have directories such as `repositories`, `models`, `services`, and `viewmodels`.

The architecture recommended in this guide lends itself to a combination of the two. Data layer objects (repositories and services) aren't tied to a single feature, while UI layer objects (views and view models) are. The following is how the code is organized within the Compass application.

```
lib
|____ui
| |____core
| | |____ui
| | | |____<shared widgets>
| | |____themes
| |____<FEATURE NAME>
| | |____view_model
| | | |_____<view_model class>.dart
| | |____widgets
| | | |____<feature name>_screen.dart
| | | |____<other widgets>
|____domain
| |____models
| | |____<model name>.dart
|____data
| |____repositories
| | |____<repository class>.dart
| |____services
| | |____<service class>.dart
| |____model
| | |____<api model class>.dart
|____config
|____utils
|____routing
|____main_staging.dart
|____main_development.dart
|____main.dart

// The test folder contains unit and widget tests
test
|____data
|____domain
|____ui
|____utils

// The testing folder contains mocks other classes need to execute tests
testing
|____fakes
|____models
```

Most of the application code lives in the `data`, `domain`, and `ui` folders. The data folder organizes code by type, because repositories and services can be used across different features and by multiple view models. The ui folder organizes the code by feature, because each feature has exactly one view and exactly one view model.

Other notable features of this folder structure:

- The UI folder also contains a subdirectory named "core". Core contains widgets and theme logic that is shared by multiple views, such as buttons with your brand styling.
- The domain folder contains the application data types, because they're used by the data and ui layers.
- The app contains three "main" files, which act as different entry points to the application for development, staging, and production.
- There are two test-related directories at the same level as `lib`: `test/` has the test code, and its own structure matches `lib/`. `testing/` is a subpackage that contains mocks and other testing utilities which can be used in other packages' test code. The `testing/` folder could be described as a version of your app that you don't ship. It's the content that is tested.

There's additional code in the compass app that doesn't pertain to architecture. For the full package structure, view it on GitHub.

## Other architecture options

The example in this case-study demonstrates how one application abides by our recommended architectural rules, but there are many other example apps that could've been written. The UI of this app leans heavily on view models and `ChangeNotifier`, but it could've easily been written with streams, or with other libraries like provided by the `riverpod`, `flutter_bloc`, and `signals` packages. The communication between layers of this app handled everything with method calls, including polling for new data. It could've instead used streams to expose data from a repository to a view model and still abide by the rules covered in this guide.

Even if you do follow this guide exactly, and choose not to introduce additional libraries, you have decisions to make: Will you have a domain layer? If so, how will you manage data access? The answer depends so much on an individual team's needs that there isn't a single right answer. Regardless of how you answer these questions, the principles in this guide will help you write scalable Flutter apps.

And if you squint, aren't all architectures MVVM anyway?

## Feedback

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

〈 Guide to app architecture

UI Layer 〉