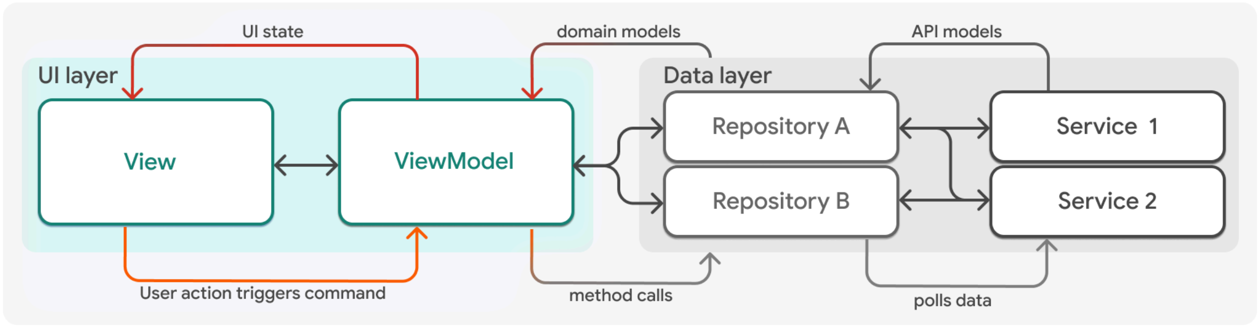
UI layer case study

Contents

Define a view model
    UI state
    Updating UI state
Define a view
    Display UI data in a view
    Update the UI
    Handling user events
    Command objects
    Ensuring views can render before data exists
Feedback

The UI layer of each feature in your Flutter application should be made up of two components: a **`View`** and a **`ViewModel`**.



In the most general sense, view models manage UI state, and views display UI state. Views and view models have a one-to-one relationship; for each view, there's exactly one corresponding view model that manages that view's state. Each pair of view and view model make up the UI for a single feature. For example, an app might have classes called `LogOutView` and a `LogOutViewModel`.

Define a view model

A view model is a Dart class responsible for handling UI logic. View models take domain data models as input and expose that data as UI state to their corresponding views. They encapsulate logic that the view can attach to event handlers, like button presses, and manage sending these events to the data layer of the app, where data changes happen.

The following code snippet is a class declaration for a view model class called the `HomeViewModel`. Its inputs are the repositories that provide its data. In this case, the view model is dependent on the `BookingRepository`and `UserRepository` as arguments.

home_viewmodel.dart

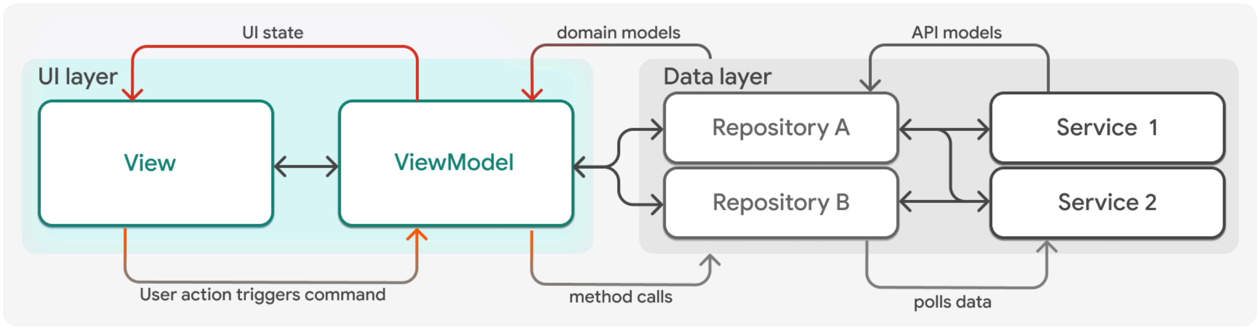
```dart
class HomeViewModel {
  HomeViewModel({
    required BookingRepository bookingRepository,
    required UserRepository userRepository,
  }) :
    // Repositories are manually assigned because they're private members.
    _bookingRepository = bookingRepository,
    _userRepository = userRepository;

  final BookingRepository _bookingRepository;
  final UserRepository _userRepository;
  // ...
}
```

View models are always dependent on data repositories, which are provided as arguments to the view model's constructor. view models and repositories have a many-to-many relationship, and most view models will depend on multiple repositories.

As in the earlier `HomeViewModel` example declaration, repositories should be private members on the view model, otherwise views would have direct access to the data layer of the application.

UI state

The output of a view model is data that a view needs to render, generally referred to as **UI State**, or just state. UI state is an immutable snapshot of data that is required to fully render a view.



The view model exposes state as public members. On the view model in the following code example, the exposed data is a `User` object, as well as the user's saved itineraries which are exposed as an object of type `List<TripSummary>`.

home_viewmodel.dart

dart

```dart
class HomeViewModel {
  HomeViewModel({
   required BookingRepository bookingRepository,
   required UserRepository userRepository,
  }) : _bookingRepository = bookingRepository,
      _userRepository = userRepository;

  final BookingRepository _bookingRepository;
  final UserRepository _userRepository;

  User? _user;
  User? get user => _user;

  List<BookingSummary> _bookings = [];

  /// Items in an [UnmodifiableListView] can't be directly modified,
  /// but changes in the source list can be modified. Since _bookings
  /// is private and bookings is not, the view has no way to modify the
  /// list directly.
  UnmodifiableListView<BookingSummary> get bookings => UnmodifiableListView(_bookings);

  // ...
}
```

As mentioned, the UI state should be immutable. This is a crucial part of bug-free software.

The compass app uses the [`package:freezed`] to enforce immutability on data classes. For example, the following code shows the `User` class definition. `freezed` provides deep immutability, and generates the implementation for useful methods like `copyWith` and `toJson`.

user.dart

dart

```dart
@freezed
class User with _$User {
  const factory User({
    /// The user's name.
    required String name,

    /// The user's picture URL.
    required String picture,
  }) = _User;

  factory User.fromJson(Map<String, Object?> json) => _$UserFromJson(json);
}
```

ⓘ **Note**

In the view model example, two objects are needed to render the view. As the UI state for any given model grows in complexity, a view model might have many more pieces of data from many more repositories exposed to the view. In some cases, you might want to create objects that specifically represent the UI state. For example, you could create a class named `HomeUiState`.

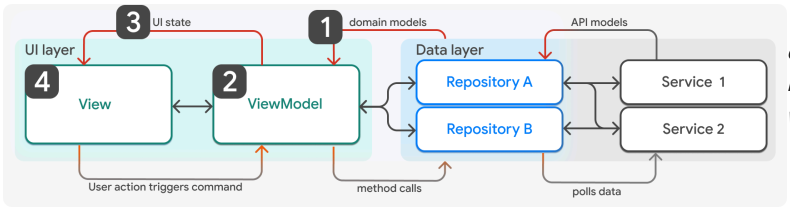
Updating UI state

In addition to storing state, view models need to tell Flutter to re-render views when the data layer provides a new state. In the Compass app, view models extend [`ChangeNotifier`] to achieve this.

home_viewmodel.dart

```dart
class HomeViewModel extends ChangeNotifier {
  HomeViewModel({
   required BookingRepository bookingRepository,
   required UserRepository userRepository,
  }) : _bookingRepository = bookingRepository,
      _userRepository = userRepository;
  final BookingRepository _bookingRepository;
  final UserRepository _userRepository;

  User? _user;
  User? get user => _user;

  List<BookingSummary> _bookings = [];
  List<BookingSummary> get bookings => _bookings;

  // ...
}
```

`HomeViewModel.user` is a public member that the view depends on. When new data flows from the data layer and new state needs to be emitted, [`notifyListeners`]() is called.



This figure shows from a high-level how new data in the repository propagates up to the UI layer and triggers a re-build of your Flutter widgets.

1. New state is provided to the view model from a Repository.
2. The view model updates its UI state to reflect the new data.
3. `ViewModel.notifyListeners` is called, alerting the View of new UI State.
4. The view (widget) re-renders.

For example, when the user navigates to the Home screen and the view model is created, the `_load` method is called. Until this method completes, the UI state is empty, the view displays a loading indicator. When the `_load` method completes, if it's successful, there's new data in the view model, and it must notify the view that new data is available.

home_viewmodel.dart

```dart
class HomeViewModel extends ChangeNotifier {
  // ...

 Future<Result> _load() async {
    try {
      final userResult = await _userRepository.getUser();
      switch (userResult) {
        case Ok<User>():
          _user = userResult.value;
          _log.fine('Loaded user');
        case Error<User>():
          _log.warning('Failed to load user', userResult.error);
      }

      // ...

      return userResult;
    } finally {
      notifyListeners();
    }
  }
}
```

Note

`ChangeNotifier` and [`ListenableBuilder`] (discussed later on this page) are part of the Flutter SDK, and provide a good solution for updating the UI when state changes. You can also use a robust third-party state management solution, such as package:riverpod, package:flutter_bloc, or package:signals. These libraries offer different tools for handling UI updates. Read more about using `ChangeNotifier` in our state-management documentation.

Define a view

A view is a widget within your app. Often, a view represents one screen in your app that has its own route and includes a `Scaffold` at the top of the widget subtree, such as the `HomeScreen`, but this isn't always the case.

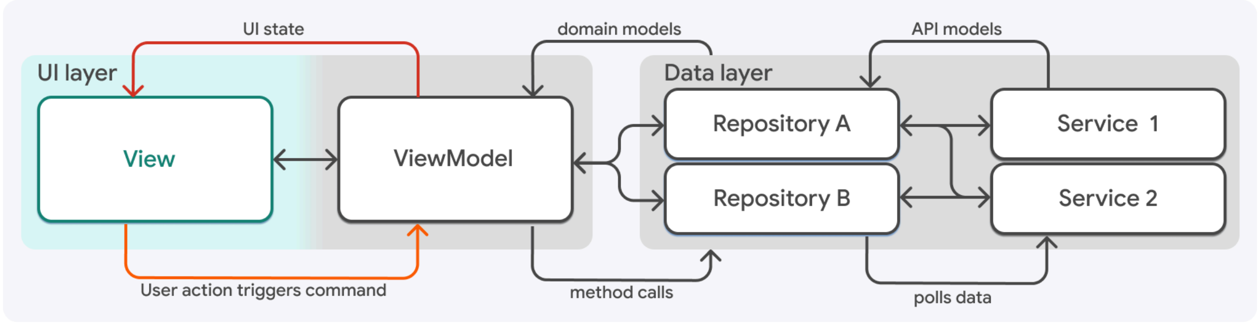
Sometimes a view is a single UI element that encapsulates functionality that needs to be re-used throughout the app. For example, the Compass app has a view called `LogoutButton`, which can be dropped anywhere in the widget tree that a user might expect to find a logout button. The `LogoutButton` view has its own view model called `LogoutViewModel`. And on larger screens, there might be multiple views on screen that would take up the full screen on mobile.

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.

The widgets within a view have three responsibilities:

- They display the data properties from the view model.
- They listen for updates from the view model and re-render when new data is available.
- They attach callbacks from the view model to event handlers, if applicable.



Continuing the Home feature example, the following code shows the definition of the `HomeScreen` view.

home_screen.dart

```dart
class HomeScreen extends StatelessWidget {
  const HomeScreen({super.key, required this.viewModel});

  final HomeViewModel viewModel;

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      // ...
    );
  }
}
```

Most of the time, a view's only inputs should be a `key`, which all Flutter widgets take as an optional argument, and the view's corresponding view model.

Display UI data in a view

A view depends on a view model for its state. In the Compass app, the view model is passed in as an argument in the view's constructor. The following example code snippet is from the `HomeScreen` widget.

home_screen.dart

```dart
class HomeScreen extends StatelessWidget {
  const HomeScreen({super.key, required this.viewModel});

  final HomeViewModel viewModel;

  @override
  Widget build(BuildContext context) {
    // ...
  }
}
```

Within the widget, you can access the passed-in bookings from the `viewModel`. In the following code, the `booking` property is being provided to a sub-widget.

home_screen.dart

```dart
@override
  Widget build(BuildContext context) {
    return Scaffold(
      // Some code was removed for brevity.
      body: SafeArea(
        child: ListenableBuilder(
          listenable: viewModel,
          builder: (context, _) {
            return CustomScrollView(
              slivers: [
                SliverToBoxAdapter(...),
                SliverList.builder(
                   itemCount: viewModel.bookings.length,
                    itemBuilder: (_, index) => _Booking(
                      key: ValueKey(viewModel.bookings[index].id),
                      booking:viewModel.bookings[index],
                      onTap: () => context.push(Routes.bookingWithId(
                         viewModel.bookings[index].id)),
                      onDismissed: (_) => viewModel.deleteBooking.execute(
                           viewModel.bookings[index].id,
                         ),
                    ),
                ),
              ],
            );
          },
        ),
      ),
```

Update the UI

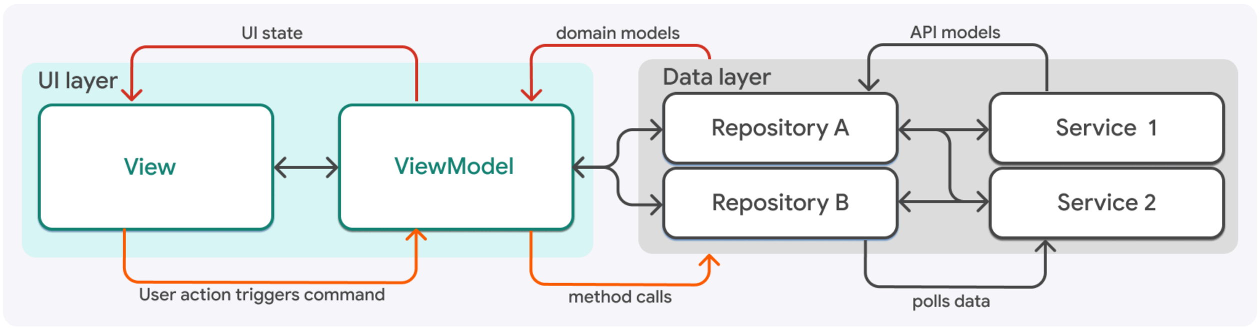
The `HomeScreen` widget listens for updates from the view model with the [`ListenableBuilder`] widget. Everything in the widget subtree under the `ListenableBuilder` widget re-renders when the provided [`Listenable`] changes. In this case, the provided `Listenable` is the view model. Recall that the view model is of type [`ChangeNotifier`] which is a subtype of the `Listenable` type.

home_screen.dart

```dart
@override
Widget build(BuildContext context) {
  return Scaffold(
    // Some code was removed for brevity.
      body: SafeArea(
        child: ListenableBuilder(
          listenable: viewModel,
          builder: (context, _) {
            return CustomScrollView(
              slivers: [
                SliverToBoxAdapter(),
                SliverList.builder(
                  itemCount: viewModel.bookings.length,
                  itemBuilder: (_, index) =>
                      _Booking(
                        key: ValueKey(viewModel.bookings[index].id),
                        booking: viewModel.bookings[index],
                        onTap: () =>
                            context.push(Routes.bookingWithId(
                                viewModel.bookings[index].id)
                            ),
                        onDismissed: (_) =>
                            viewModel.deleteBooking.execute(
                              viewModel.bookings[index].id,
                            ),
                      ),
                ),
              ],
            );
          }
        )
      )
  );
}
```

Handling user events

Finally, a view needs to listen for *events* from users, so the view model can handle those events. This is achieved by exposing a callback method on the view model class which encapsulates all the logic.

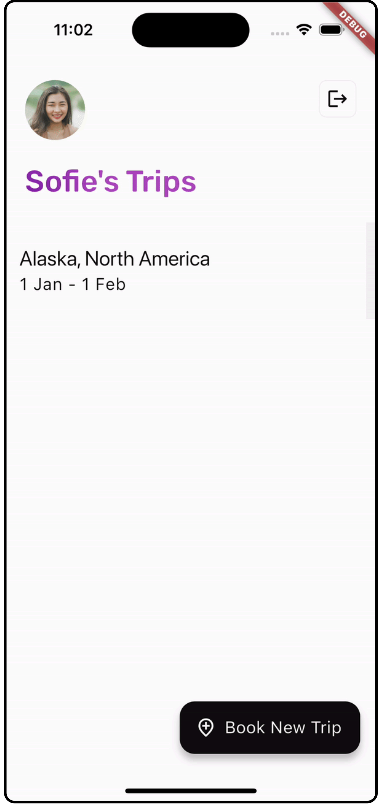


On the `HomeScreen`, users can delete previously booked events by swiping a [`Dismissible`](Dismissible) widget.

Recall this code from the previous snippet:

home_screen.dart

```dart
SliverList.builder(
  itemCount: widget.viewModel.bookings.length,
  itemBuilder: (_, index) => _Booking(
    key: ValueKey(viewModel.bookings[index].id),
    booking: viewModel.bookings[index],
    onTap: () => context.push(
      Routes.bookingWithId(viewModel.bookings[index].id)
    ),
    onDismissed: (_) =>

viewModel.deleteBooking.execute(widget.viewModel.bookings[index].
  ),
),
```



On the HomeScreen, a user's saved trip is represented by the _Booking widget. When a _Booking is dismissed, the viewModel.deleteBooking method is executed.

A saved booking is application state that persists beyond a session or the lifetime of a view, and only repositories should modify such application state. So, the HomeViewModel.deleteBooking method turns around and calls a method exposed by a repository in the data layer, as shown in the following code snippet.

home_viewmodel.dart

```dart
Future<Result<void>> _deleteBooking(int id) async {
  try {
    final resultDelete = await _bookingRepository.delete(id);
    switch (resultDelete) {
      case Ok<void>():
        _log.fine('Deleted booking $id');
      case Error<void>():
        _log.warning('Failed to delete booking $id', resultDelete.error);
        return resultDelete;
    }

    // Some code was omitted for brevity.
    // final  resultLoadBookings = ...;

    return resultLoadBookings;
  } finally {
    notifyListeners();
  }
}
```

In the Compass app, these methods that handle user events are called **commands**.

Command objects

Commands are responsible for the interaction that starts in the UI layer and flows back to the data layer. In this app specifically, a Command is also a type that helps update the UI safely, regardless of the response time or contents.

The `Command` class wraps a method and helps handle the different states of that method, such as `running`, `complete`, and `error`. These states make it easy to display different UI, like loading indicators when `Command.running` is true.

The following is code from the `Command` class. Some code has been omitted for demo purposes.

command.dart

```dart
abstract class Command<T> extends ChangeNotifier {
  Command();
  bool running = false;
  Result<T>? _result;

  /// true if action completed with error
  bool get error => _result is Error;

  /// true if action completed successfully
  bool get completed => _result is Ok;

  /// Internal execute implementation
  Future<void> _execute(action) async {
    if (_running) return;

    // Emit running state - e.g. button shows loading state
    _running = true;
    _result = null;
    notifyListeners();

    try {
      _result = await action();
    } finally {
      _running = false;
      notifyListeners();
    }
  }
}
```

The `Command` class itself extends `ChangeNotifier`, and within the method `Command.execute`, `notifyListeners` is called multiple times. This allows the view to handle different states with very little logic, which you'll see an example of later on this page.

You may have also noticed that `Command` is an abstract class. It's implemented by concrete classes such as `Command0` `Command1`. The integer in the class name refers to the number of arguments that the underlying method expects. You can see examples of these implementation classes in the Compass app's `utils` directory.

Package recommendation

Instead of writing your own `Command` class, consider using the `flutter_command` package, which is a robust library that implements classes like these.

Ensuring views can render before data exists

In view model classes, commands are created in the constructor.

home_viewmodel.dart

dart

```dart
class HomeViewModel extends ChangeNotifier {
  HomeViewModel({
   required BookingRepository bookingRepository,
   required UserRepository userRepository,
  }) : _bookingRepository = bookingRepository,
      _userRepository = userRepository {
    // Load required data when this screen is built.
    load = Command0(_load)..execute();
    deleteBooking = Command1(_deleteBooking);
  }

  final BookingRepository _bookingRepository;
  final UserRepository _userRepository;

  late Command0 load;
  late Command1<void, int> deleteBooking;

  User? _user;
  User? get user => _user;

  List<BookingSummary> _bookings = [];
  List<BookingSummary> get bookings => _bookings;

  Future<Result> _load() async {
    // ...
  }

  Future<Result<void>> _deleteBooking(int id) async {
    // ...
  }

  // ...
}
```

The `Command.execute` method is asynchronous, so it can't guarantee that the data will be available when the view wants to render. This gets at *why* the Compass app uses `Commands`. In the view's `Widget.build` method, the command is used to conditionally render different widgets.

home_screen.dart

```dart
// ...
child: ListenableBuilder(
  listenable: viewModel.load,
  builder: (context, child) {
    if (viewModel.load.running) {
      return const Center(child: CircularProgressIndicator());
    }

    if (viewModel.load.error) {
      return ErrorIndicator(
        title: AppLocalization.of(context).errorWhileLoadingHome,
        label: AppLocalization.of(context).tryAgain,
          onPressed: viewModel.load.execute,
        );
     }

    // The command has completed without error.
    // Return the main view widget.
    return child!;
  },
),

// ...
```

Because the `load` command is a property that exists on the view model rather than something ephemeral, it doesn't matter when the `load` method is called or when it resolves. For example, if the load command resolves before the `HomeScreen` widget was even created, it isn't a problem because the `Command` object still exists, and exposes the correct state.

This pattern standardizes how common UI problems are solved in the app, making your codebase less error-prone and more scalable, but it's not a pattern that every app will want to implement. Whether you want to use it is highly dependent on other architectural choices you make. Many libraries that help you manage state have their own tools to solve these problems. For example, if you were to use streams and `StreamBuilders` in your app, the `AsyncSnapshot` classes provided by Flutter have this functionality built in.

Real world example

While building the Compass app, we found a bug that was solved by using the Command pattern. Read about it on GitHub.

Feedback

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

〈 Case study overview　　　Data Layer 〉