The command pattern

Contents

Model-View-ViewModel (MVVM) is a design pattern that separates a feature of an application into three parts: the model, the view model and the view. Views and view models make up the UI layer of an application. Repositories and services represent the data layer of an application, or the model layer of MVVM.

A command is a class that wraps a method and helps to handle the different states of that method, such as running, complete, and error.

View models can use commands to handle interaction and run actions. As well, they can be used to display different UI states, like loading indicators when an action is running, or an error dialog when an action failed.

View models can become very complex as an application grows and features become bigger. Commands can help to simplify view models and reuse code.

In this guide, you will learn how to use the command pattern to improve your view models.

Challenges when implementing view models

View model classes in Flutter are typically implemented by extending the `ChangeNotifier` class. This allows view models to call `notifyListeners()` to refresh views when data is updated.

```dart
class HomeViewModel extends ChangeNotifier {
  // ···
}
```

View models contain a representation of the UI state, including the data being displayed. For example, this `HomeViewModel` exposes the `User` instance to the view.

```dart
class HomeViewModel extends ChangeNotifier {

  User? get user => // ...
  // ···
}
```

View models also contain actions typically triggered by the view; for example, a `load` action in charge of loading the `user`.

```dart
class HomeViewModel extends ChangeNotifier {

  User? get user => // ...
  // ···
  void load() {
    // load user
  }
  // ···
}
```

UI state in view models

A view model also contains UI state besides data, such as whether the view is running or has experienced an error. This allows the app to tell the user if the action has completed successfully.

```dart
class HomeViewModel extends ChangeNotifier {

  User? get user => // ...

  bool get running => // ...

  Exception? get error => // ...

  void load() {
    // load user
  }
  // ···
}
```

You can use the running state to display a progress indicator in the view:

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

Or use the running state to avoid executing the action multiple times:

```dart
void load() {
  if (running) {
    return;
  }
  // load user
}
```

Managing the state of an action can get complicated if the view model contains multiple actions. For example, adding an `edit()` action to the `HomeViewModel` can lead the following outcome:

```dart
class HomeViewModel extends ChangeNotifier {
  User? get user => // ...

  bool get runningLoad => // ...

  Exception? get errorLoad => // ...

  bool get runningEdit => // ...

  Exception? get errorEdit => // ...

  void load() {
    // load user
  }

  void edit(String name) {
    // edit user
  }
}
```

Sharing the running state between the `load()` and `edit()` actions might not always work, because you might want to show a different UI component when the `load()` action runs than when the `edit()` action runs, and you'll have the same problem with the `error` state.

Triggering UI actions from view models

View model classes can run into problems when executing UI actions and the view model's state changes.

For example, you might want to show a `SnackBar` when an error occurs, or navigate to a different screen when an action completes. To implement this, listen for changes in the view model, and perform the action depending on the state.

In the view:

```dart
@override
void initState() {
  super.initState();
  widget.viewModel.addListener(_onViewModelChanged);
}

@override
void dispose() {
  widget.viewModel.removeListener(_onViewModelChanged);
  super.dispose();
}
```

```dart
void _onViewModelChanged() {
  if (widget.viewModel.error != null) {
    // Show Snackbar
  }
}
```

You need to clear the error state each time you execute this action, otherwise this action happens each time `notifyListeners()` is called.

```dart
void _onViewModelChanged() {
  if (widget.viewModel.error != null) {
    widget.viewModel.clearError();
    // Show Snackbar
  }
}
```

Command pattern

You might find yourself repeating the above code over and over, implementing a different running state for each action in every view model. At that point, it makes sense to extract this code into a reusable pattern: a command.

A command is a class that encapsulates a view model action, and exposes the different states that an action can have.

```dart
class Command extends ChangeNotifier {
  Command(this._action);

  bool get running => // ...

  Exception? get error => // ...

  bool get completed => // ...

  void Function() _action;

  void execute() {
    // run _action
  }

  void clear() {
    // clear state
  }
}
```

In the view model, instead of defining an action directly with a method, you create a command object:

```dart
class HomeViewModel extends ChangeNotifier {
  HomeViewModel() {
    load = Command(_load)..execute();
  }

  User? get user => // ...

  late final Command load;

  void _load() {
    // load user
  }
}
```

The previous `load()` method becomes `_load()`, and instead the command `load` gets exposed to the View. The previous `running` and `error` states can be removed, as they are now part of the command.

Executing a command

Instead of calling `viewModel.load()` to run the load action, now you call `viewModel.load.execute()`.

The `execute()` method can also be called from within the view model. The following line of code runs the `load` command when the view model is created.

```dart
HomeViewModel() {
  load = Command(_load)..execute();
}
```

The `execute()` method sets the running state to `true` and resets the `error` and `completed` states. When the action finishes, the `running` state changes to `false` and the `completed` state to `true`.

If the `running` state is `true`, the command cannot begin executing again. This prevents users from triggering a command multiple times by pressing a button rapidly.

The command's `execute()` method captures any thrown `Exceptions` automatically and exposes them in the `error` state.

The following code shows a sample `Command` class that has been simplified for demo purposes. You can see a full implementation at the end of this page.

```dart
class Command extends ChangeNotifier {
  Command(this._action);

  bool _running = false;
  bool get running => _running;

  Exception? _error;
  Exception? get error => _error;

  bool _completed = false;
  bool get completed => _completed;

  final Future<void> Function() _action;

  Future<void> execute() async {
    if (_running) {
      return;
    }

    _running = true;
    _completed = false;
    _error = null;
    notifyListeners();

    try {
      await _action();
      _completed = true;
    } on Exception catch (error) {
      _error = error;
    } finally {
      _running = false;
      notifyListeners();
    }
  }

  void clear() {
    _running = false;
    _error = null;
    _completed = false;
  }
}
```

Listening to the command state

The `Command` class extends from `ChangeNotifier`, allowing Views to listen to its states.

In the `ListenableBuilder`, instead of passing the view model to `ListenableBuilder.listenable`, pass the command:

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

And listen to changes in the command state in order to run UI actions:

```dart
@override
void initState() {
  super.initState();
  widget.viewModel.addListener(_onViewModelChanged);
}

@override
void dispose() {
  widget.viewModel.removeListener(_onViewModelChanged);
  super.dispose();
}
```

```dart
void _onViewModelChanged() {
  if (widget.viewModel.load.error != null) {
    widget.viewModel.load.clear();
    // Show Snackbar
  }
}
```

Combining command and ViewModel

You can stack multiple `ListenableBuilder` widgets to listen to `running` and `error` states before showing the view model data.

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

    if (widget.viewModel.load.error != null) {
      return Center(
        child: Text('Error: ${widget.viewModel.load.error}'),
      );
    }

    return child!;
  },
  child: ListenableBuilder(
    listenable: widget.viewModel,
    builder: (context, _) {
      // ...
    },
  ),
),
```

You can define multiple commands classes in a single view model, simplifying its implementation and minimizing the amount of repeated code.

```dart
class HomeViewModel2 extends ChangeNotifier {
  HomeViewModel2() {
    load = Command(_load)..execute();
    delete = Command(_delete);
  }

  User? get user => // ...

  late final Command load;

  late final Command delete;

  Future<void> _load() async {
    // load user
  }

  Future<void> _delete() async {
    // delete user
  }
}
```

Extending the command pattern

The command pattern can be extended in multiple ways. For example, to support a different number of arguments.

```dart
class HomeViewModel extends ChangeNotifier {
  HomeViewModel() {
    load = Command0(_load)..execute();
    edit = Command1<String>(_edit);
  }

  User? get user => // ...

  // Command0 accepts 0 arguments
  late final Command0 load;

  // Command1 accepts 1 argument
  late final Command1 edit;

  Future<void> _load() async {
    // load user
  }

  Future<void> _edit(String name) async {
    // edit user
  }
}
```

Putting it all together

In this guide, you learned how to use the command design pattern to improve the implementation of view models when using the MVVM design pattern.

Below, you can find the full `Command` class as implemented in the Compass App example for the Flutter architecture guidelines. It also uses the `Result` class to determine if the action completed successfuly or with an error.

This implementation also includes two types of commands, a `Command0`, for actions without parameters, and a `Command1`, for actions that take one parameter.

> **Note**
>
> Check pub.dev for other ready-to-use implementations of the command pattern, such as the `flutter_command` package.

```dart
// Copyright 2024 The Flutter team. All rights reserved.
// Use of this source code is governed by a BSD-style license that can be
// found in the LICENSE file.

import 'dart:async';

import 'package:flutter/foundation.dart';

import 'result.dart';

/// Defines a command action that returns a [Result] of type [T].
/// Used by [Command0] for actions without arguments.
typedef CommandAction0<T> = Future<Result<T>> Function();

/// Defines a command action that returns a [Result] of type [T].
/// Takes an argument of type [A].
/// Used by [Command1] for actions with one argument.
typedef CommandAction1<T, A> = Future<Result<T>> Function(A);

/// Facilitates interaction with a view model.
///
/// Encapsulates an action,
/// exposes its running and error states,
/// and ensures that it can't be launched again until it finishes.
///
/// Use [Command0] for actions without arguments.
/// Use [Command1] for actions with one argument.
///
/// Actions must return a [Result] of type [T].
///
/// Consume the action result by listening to changes,
/// then call to [clearResult] when the state is consumed.
abstract class Command<T> extends ChangeNotifier {
  bool _running = false;

  /// Whether the action is running.
  bool get running => _running;

  Result<T>? _result;

  /// Whether the action completed with an error.
  bool get error => _result is Error;

  /// Whether the action completed successfully.
  bool get completed => _result is Ok;

  /// The result of the most recent action.
  ///
  /// Returns `null` if the action is running or completed with an error.
  Result<T>? get result => _result;

  /// Clears the most recent action's result.
  void clearResult() {
    _result = null;
    notifyListeners();
  }

  /// Execute the provided [action], notifying listeners and
```

```dart
  /// setting the running and result states as necessary.
  Future<void> _execute(CommandAction0<T> action) async {
    // Ensure the action can't launch multiple times.
    // e.g. avoid multiple taps on button
    if (_running) return;

    // Notify listeners.
    // e.g. button shows loading state
    _running = true;
    _result = null;
    notifyListeners();

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

/// A [Command] that accepts no arguments.
final class Command0<T> extends Command<T> {
  /// Creates a [Command0] with the provided [CommandAction0].
  Command0(this._action);

  final CommandAction0<T> _action;

  /// Executes the action.
  Future<void> execute() async {
    await _execute(() => _action());
  }
}

/// A [Command] that accepts one argument.
final class Command1<T, A> extends Command<T> {
  /// Creates a [Command1] with the provided [CommandAction1].
  Command1(this._action);

  final CommandAction1<T, A> _action;

  /// Executes the action with the specified [argument].
  Future<void> execute(A argument) async {
    await _execute(() => _action(argument));
  }
}
```