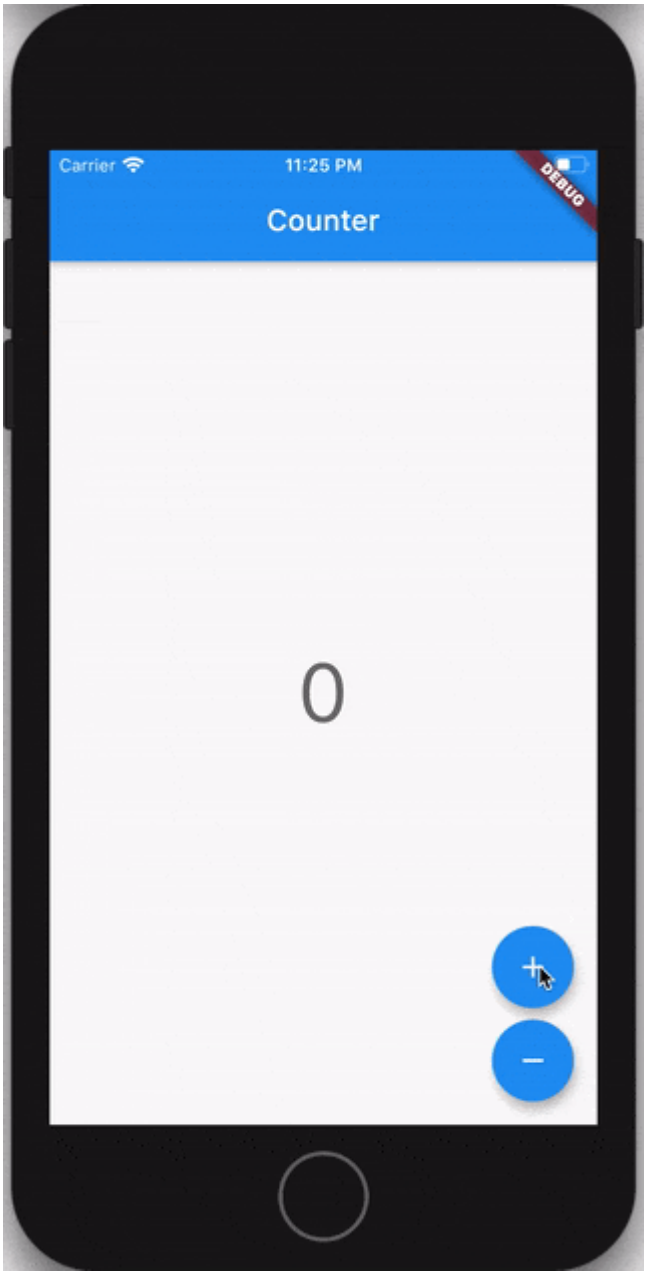
Flutter Counter Tutorial

level beginner

> In the following tutorial, we're going to build a Counter in Flutter using the Bloc library.



Key Topics

- Observe state changes with BlocObserver.
- BlocProvider, Flutter widget which provides a bloc to its children.
- BlocBuilder, Flutter widget that handles building the widget in response to new states.
- Using Cubit instead of Bloc. What's the difference?
- Adding events with context.read.⚡

Setup

We'll start off by creating a brand new Flutter project

```sh
flutter create flutter_counter
```

We can then go ahead and replace the contents of `pubspec.yaml` with

```yaml
name: flutter_counter
description: A new Flutter project.
version: 1.0.0+1
publish_to: none

environment:
  sdk: ">=2.12.0-0 <3.0.0"

dependencies:
  flutter:
    sdk: flutter
  bloc:
    path: ../../packages/bloc
  flutter_bloc:
    path: ../../packages/flutter_bloc

dependency_overrides:
  bloc:
    path: ../../packages/bloc
  flutter_bloc:
    path: ../../packages/flutter_bloc
```

```
  bloc_test:
    path: ../../packages/bloc_test

dev_dependencies:
  flutter_test:
    sdk: flutter
  bloc_test:
    path: ../../packages/bloc_test
  mocktail: ^0.1.0
  integration_test: ^1.0.0

flutter:
  uses-material-design: true
```

and then install all of our dependencies

```sh
flutter packages get
```

Project Structure

```
├── lib
|   ├── app.dart
|   ├── counter
|   |   ├── counter.dart
|   |   ├── cubit
|   |   |   └── counter_cubit.dart
|   |   └── view
|   |       ├── counter_page.dart
|   |       └── counter_view.dart
|   ├── counter_observer.dart
|   └── main.dart
├── pubspec.lock
├── pubspec.yaml
```

The application uses a feature-driven directory structure. This project structure enables us to scale the project by having self-contained features. In this example

we will only have a single feature (the counter itself) but in more complex applications we can have hundreds of different features.

BlocObserver

The first thing we're going to take a look at is how to create a `BlocObserver` which will help us observe all state changes in the application.

Let's create `lib/counter_observer.dart`:

```dart
import 'package:bloc/bloc.dart';

/// {@template counter_observer}
/// [BlocObserver] for the counter application which
/// observes all [Bloc] state changes.
/// {@endtemplate}
class CounterObserver extends BlocObserver {
  @override
  void onTransition(Bloc bloc, Transition transition) {
    super.onTransition(bloc, transition);
    print('${bloc.runtimeType} $transition');
  }
}
```

In this case, we're only overriding `onChange` to see all state changes that occur.

> **Note**: `onChange` works the same way for both `Bloc` and `Cubit` instances.

main.dart

Next, let's replace the contents of `main.dart` with:

```dart
import 'package:bloc/bloc.dart';
import 'package:flutter/material.dart';

import 'app.dart';
```

```dart
import 'counter_observer.dart';

void main() {
  Bloc.observer = CounterObserver();
  runApp(const CounterApp());
}
```

We're initializing the `CounterObserver` we just created and calling `runApp` with the `CounterApp` widget which we'll look at next.

Counter App

`CounterApp` will be a `MaterialApp` and is specifying the `home` as `CounterPage`.

```dart
import 'package:flutter/material.dart';

import 'counter/counter.dart';

/// {@template counter_app}
/// A [MaterialApp] which sets the `home` to [CounterPage].
/// {@endtemplate}
class CounterApp extends MaterialApp {
  /// {@macro counter_app}
  const CounterApp({Key? key}) : super(key: key, home: const CounterPage(
}
```

Note: We are extending `MaterialApp` because `CounterApp` *is* a `MaterialApp`. In most cases, we're going to be creating `StatelessWidget` or `StatefulWidget` instances and composing widgets in `build` but in this case there are no widgets to compose so it's simpler to just extend `MaterialApp`.

Let's take a look at `CounterPage` next!

Counter Page

The `CounterPage` widget is responsible for creating a `CounterCubit` (which we will look at next) and providing it to the `CounterView`.

```dart
import 'package:flutter/material.dart';
import 'package:flutter_bloc/flutter_bloc.dart';

import '../counter.dart';
import 'counter_view.dart';

/// {@template counter_page}
/// A [StatelessWidget] which is responsible for providing a
/// [CounterCubit] instance to the [CounterView].
/// {@endtemplate}
class CounterPage extends StatelessWidget {
  /// {@macro counter_page}
  const CounterPage({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return BlocProvider(
      create: (_) => CounterCubit(),
      child: CounterView(),
    );
  }
}
```

Note: It's important to separate or decouple the creation of a `Cubit` from the consumption of a `Cubit` in order to have code that is much more testable and reusable.

Counter Cubit

The `CounterCubit` class will expose two methods:

- `increment`: adds 1 to the current state
- `decrement`: subtracts 1 from the current state

The type of state the `CounterCubit` is managing is just an `int` and the initial state is `0`.

```dart
import 'package:bloc/bloc.dart';

/// {@template counter_cubit}
/// A [Cubit] which manages an [int] as its state.
/// {@endtemplate}
class CounterCubit extends Cubit<int> {
  /// {@macro counter_cubit}
  CounterCubit() : super(0);

  /// Add 1 to the current state.
  void increment() => emit(state + 1);

  /// Subtract 1 from the current state.
  void decrement() => emit(state - 1);
}
```

Tip: Use the VSCode Extension or IntelliJ Plugin to create new cubits automatically.

Next, let's take a look at the `CounterView` which will be responsible for consuming the state and interacting with the `CounterCubit`.

Counter View

The `CounterView` is responsible for rendering the current count and rendering two FloatingActionButtons to increment/decrement the counter.

```dart
import 'package:flutter/material.dart';
import 'package:flutter_bloc/flutter_bloc.dart';

import '../counter.dart';

/// {@template counter_view}
/// A [StatelessWidget] which reacts to the provided
```

```dart
/// [CounterCubit] state and notifies it in response to user input.
/// {@endtemplate}
class CounterView extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final textTheme = Theme.of(context).textTheme;
    return Scaffold(
      appBar: AppBar(title: const Text('Counter')),
      body: Center(
        child: BlocBuilder<CounterCubit, int>(
          builder: (context, state) {
            return Text('$state', style: textTheme.headline2);
          },
        ),
      ),
      floatingActionButton: Column(
        mainAxisAlignment: MainAxisAlignment.end,
        crossAxisAlignment: CrossAxisAlignment.end,
        children: <Widget>[
          FloatingActionButton(
            key: const Key('counterView_increment_floatingActionButton'),
            child: const Icon(Icons.add),
            onPressed: () => context.read<CounterCubit>().increment(),
          ),
          const SizedBox(height: 8),
          FloatingActionButton(
            key: const Key('counterView_decrement_floatingActionButton'),
            child: const Icon(Icons.remove),
            onPressed: () => context.read<CounterCubit>().decrement(),
          ),
        ],
      ),
    );
  }
}
```

A `BlocBuilder` is used to wrap the `Text` widget in order to update the text any time the `CounterCubit` state changes. In addition, `context.read<CounterCubit>()` is used to look-up the closest `CounterCubit` instance.

Note: Only the `Text` widget is wrapped in a `BlocBuilder` because that is the only widget that needs to be rebuilt in response to state changes in the `CounterCubit`. Avoid unnecessarily wrapping widgets that don't need to be rebuilt when a state changes.

Barrel

Add `counter.dart` to export all the public facing parts of the counter feature.

```dart
export 'cubit/counter_cubit.dart';
export 'view/counter_page.dart';
```

That's it! We've separated the presentation layer from the business logic layer. The `CounterView` has no idea what happens when a user presses a button; it just notifies the `CounterCubit`. Furthermore, the `CounterCubit` has no idea what is happening with the state (counter value); it's simply emitting new states in response to the methods being called.

We can run our app with `flutter run` and can view it on our device or simulator/emulator.

The full source (including unit and widget tests) for this example can be found here.

NEXT >
Timer