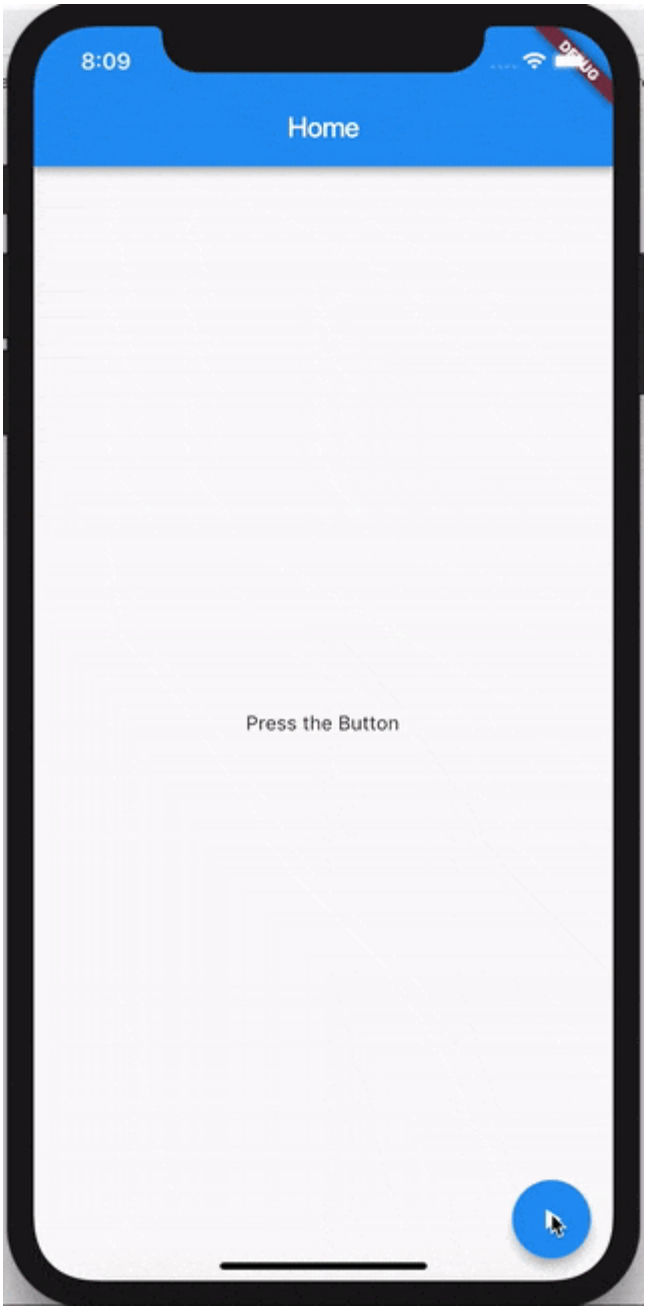
Recipes: Show SnackBar with BlocListener

> In this recipe, we're going to take a look at how to use `BlocListener` to show a `SnackBar` in response to a state change in a bloc.



Bloc

Let's build a basic `DataBloc` which will handle `DataEvents` and output `DataStates`.

DataEvent

For simplicity, our `DataBloc` will only respond to a single `DataEvent` called `FetchData`.

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

@immutable
abstract class DataEvent {}

class FetchData extends DataEvent {}
```

DataState

Our `DataBloc` can have one of three different `DataStates`:

- `Initial` - the initial state before any events are added
- `Loading` - the state of the bloc while it is asynchronously "fetching data"
- `Success` - the state of the bloc when it has successfully "fetched data"

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

@immutable
abstract class DataState {}

class Initial extends DataState {}

class Loading extends DataState {}

class Success extends DataState {}
```

DataBloc

Our `DataBloc` should look something like this:

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

class DataBloc extends Bloc<DataEvent, DataState> {
  DataBloc() : super(Initial());

  @override
  Stream<DataState> mapEventToState(
    DataEvent event,
  ) async* {
    if (event is FetchData) {
      yield Loading();
      await Future.delayed(Duration(seconds: 2));
      yield Success();
    }
  }
}
```

Note: We're using `Future.delayed` to simulate latency.

UI Layer

Now let's take a look at how to hook up our `DataBloc` to a widget and show a `SnackBar` in response to a success state.

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

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return BlocProvider(
      create: (context) => DataBloc(),
      child: MaterialApp(
```

```
        home: Home(),
      ),
    );
  }
}

class Home extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final dataBloc = BlocProvider.of<DataBloc>(context);
    return Scaffold(
      appBar: AppBar(title: Text('Home')),
      body: BlocListener<DataBloc, DataState>(
        listener: (context, state) {
          if (state is Success) {
            Scaffold.of(context).showSnackBar(
              SnackBar(
                backgroundColor: Colors.green,
                content: Text('Success'),
              ),
            );
          }
        },
        child: BlocBuilder<DataBloc, DataState>(
          builder: (context, state) {
            if (state is Initial) {
              return Center(child: Text('Press the Button'));
            }
            if (state is Loading) {
              return Center(child: CircularProgressIndicator());
            }
            if (state is Success) {
              return Center(child: Text('Success'));
            }
          },
        ),
      ),
      floatingActionButton: Column(
        crossAxisAlignment: CrossAxisAlignment.end,
        mainAxisAlignment: MainAxisAlignment.end,
        children: <Widget>[
          FloatingActionButton(
            child: Icon(Icons.play_arrow),
```

```
              onPressed: () {
                dataBloc.add(FetchData());
              },
            ),
          ],
        ),
      );
    }
  }
```

We use the `BlocListener` widget in order to **DO THINGS** in response to state changes in our `DataBloc`.

We use the `BlocBuilder` widget in order to **RENDER WIDGETS** in response to state changes in our `DataBloc`.

> We should **NEVER** "do things" in response to state changes in the `builder` method of `BlocBuilder` because that method can be called many times by the Flutter framework. The `builder` method should be a pure function that just returns a widget in response to the state of the bloc.

The full source for this recipe can be found here.

NEXT >
Navigation