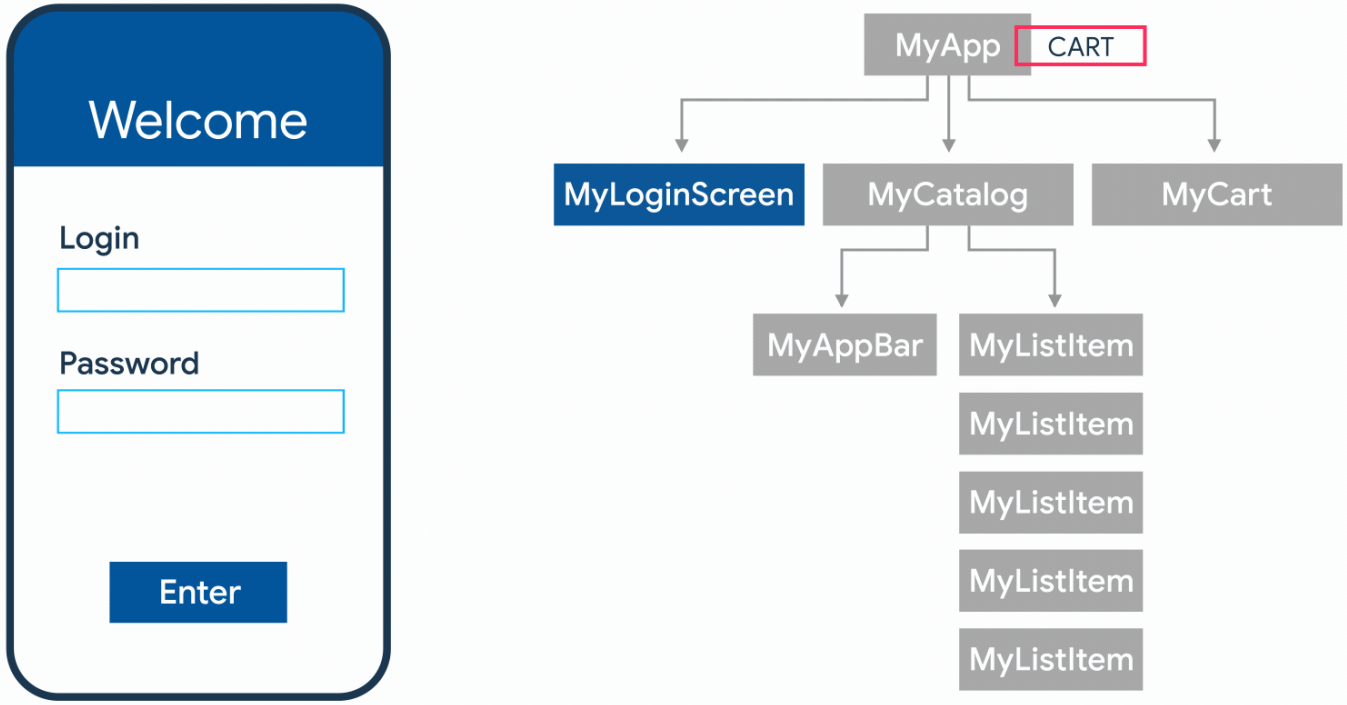
Resource Software Solution

State Management using Bloc

Introduction

- There comes a time when you need to **share application state between screens**, across your app.



Start thinking declaratively

- When you **change the state**, and that triggers a **redraw of the user interface**.

- The state that you *do* manage yourself can be separated into two conceptual types: **ephemeral state** and **app state**.

Ephemeral state

- Ephemeral state (sometimes called *UI state* or *local state*) is the state you can neatly contain in a **single widget**.
- There is no need to use state management techniques (Bloc, Redux, etc.) on this kind of state. All you need is a **StatefulWidget**.

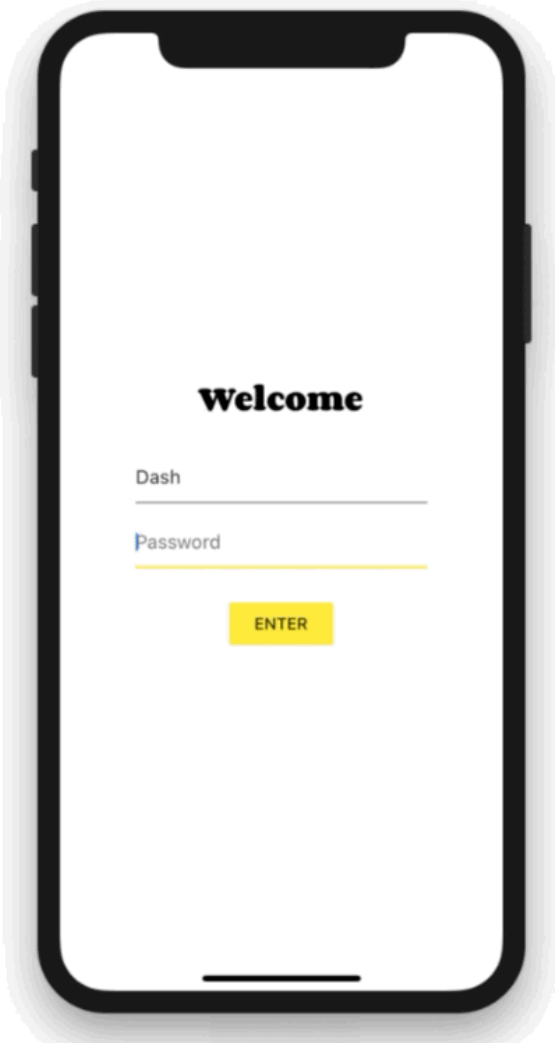
Ephemeral state

- You see how the currently selected item in a bottom navigation bar is held in the **_index** field of the _MyHomepageState class.
- **_index** is **ephemeral state**.
- The **_index** only changes inside the MyHomepage widget.

```
class MyHomepage extends StatefulWidget {
  const MyHomepage({super.key});

  @override
  State<MyHomepage> createState() => _MyHomepageState();
}

class _MyHomepageState extends State<MyHomepage> {
  int _index = 0;

  @override
  Widget build(BuildContext context) {
    return BottomNavigationBar(
      currentIndex: _index,
      onTap: (newIndex) {
        setState(() {
          _index = newIndex;
        });
      },
      // ... items ...
    );
  }
}
```

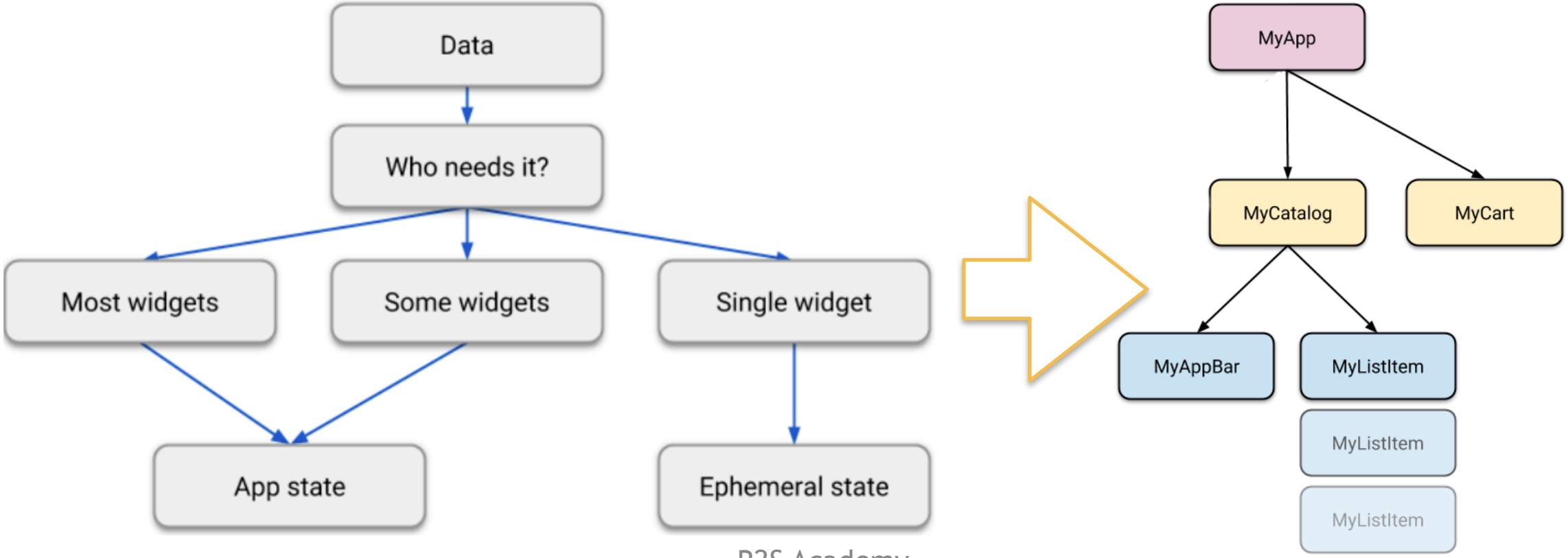
App state

- You want to share across many parts of your app, and that you want to keep between user sessions, is what we call application state (sometimes also called shared state).
- Examples of application state:
  - Notifications in a social networking app
  - The shopping cart in an e-commerce app
  - Read/unread state of articles in a news app
  - …



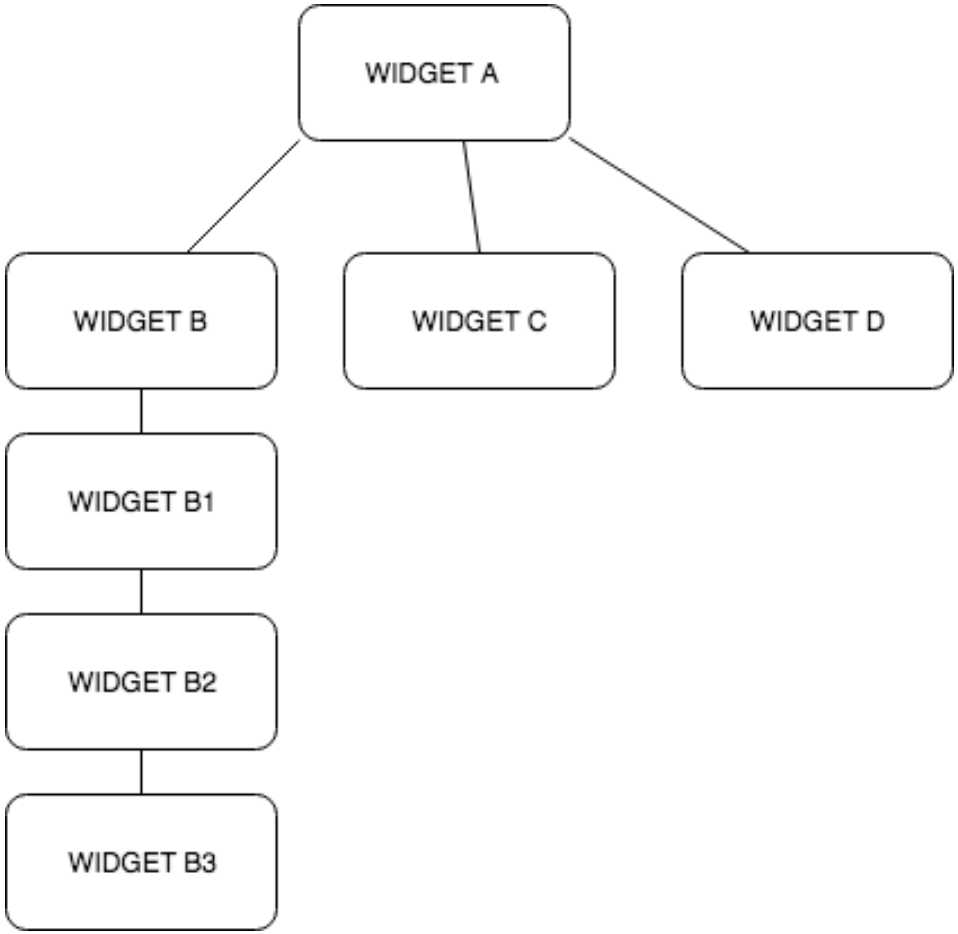
Ephemeral state vs App state

- **Ephemeral** state can be implemented using State and **setState**(), and is often local to a single widget.
- The rest is your **app state**. Both types have their place in any Flutter app, and the split between the two depends on the complexity of the app.



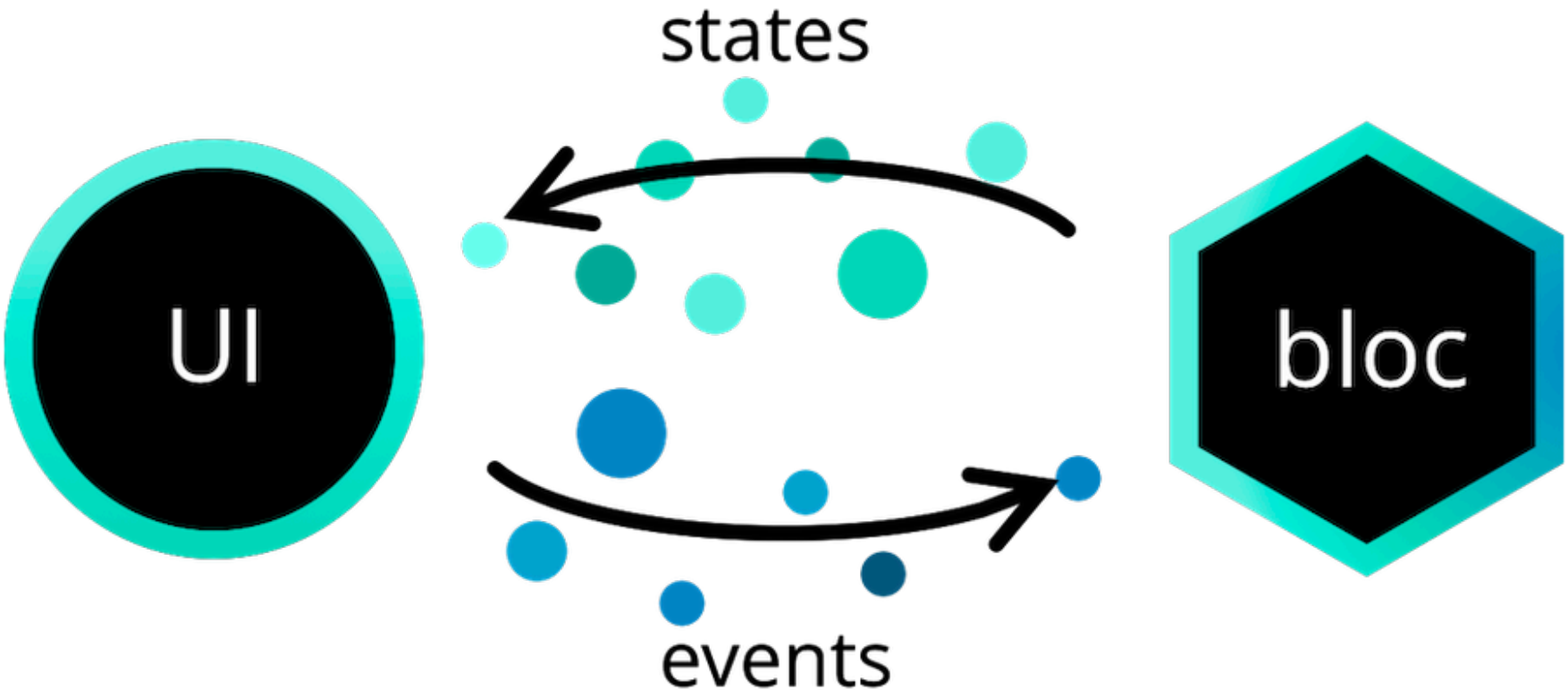
List of state management approaches

- setState
- Provider
- Riverpod
- InheritedWidget & InheritedModel
- Redux
- **BLoC / Rx**
- Reference: https://docs.flutter.dev/development/data-and-backend/state-mgmt/options



BloC pattern for Flutter

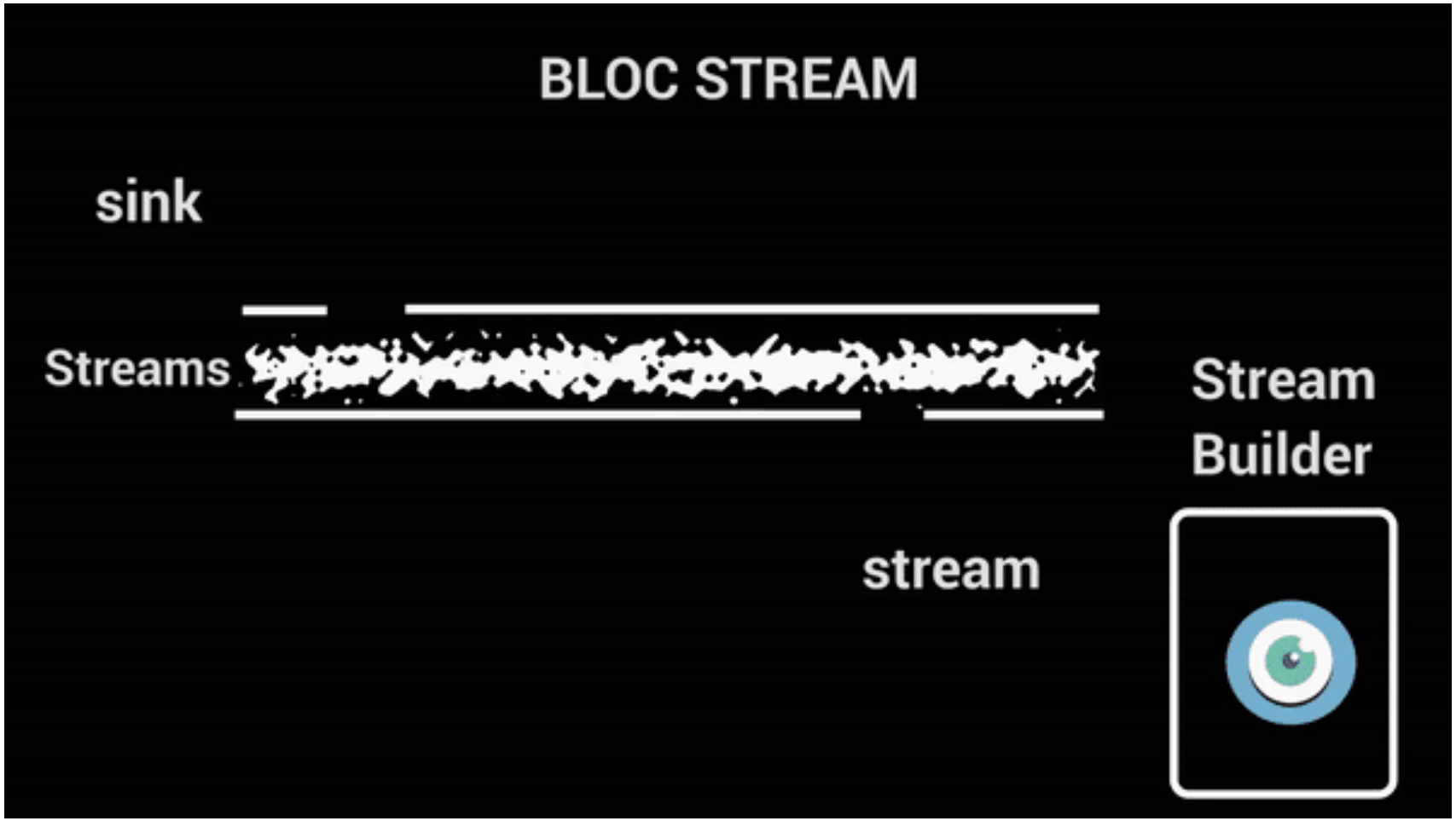
- Overview

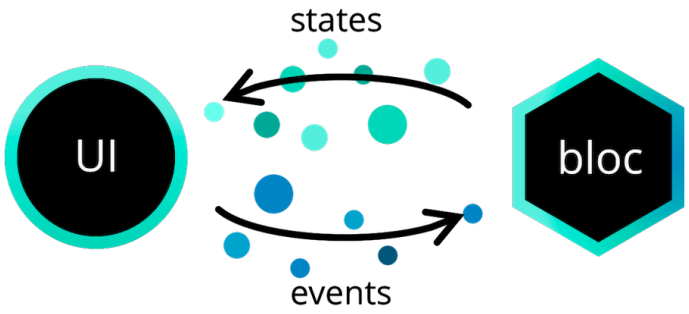


- The goal of this library is to make it easy to separate ***presentation*** from ***business logic***, facilitating **testability** and **reusability**.

BloC pattern for Flutter

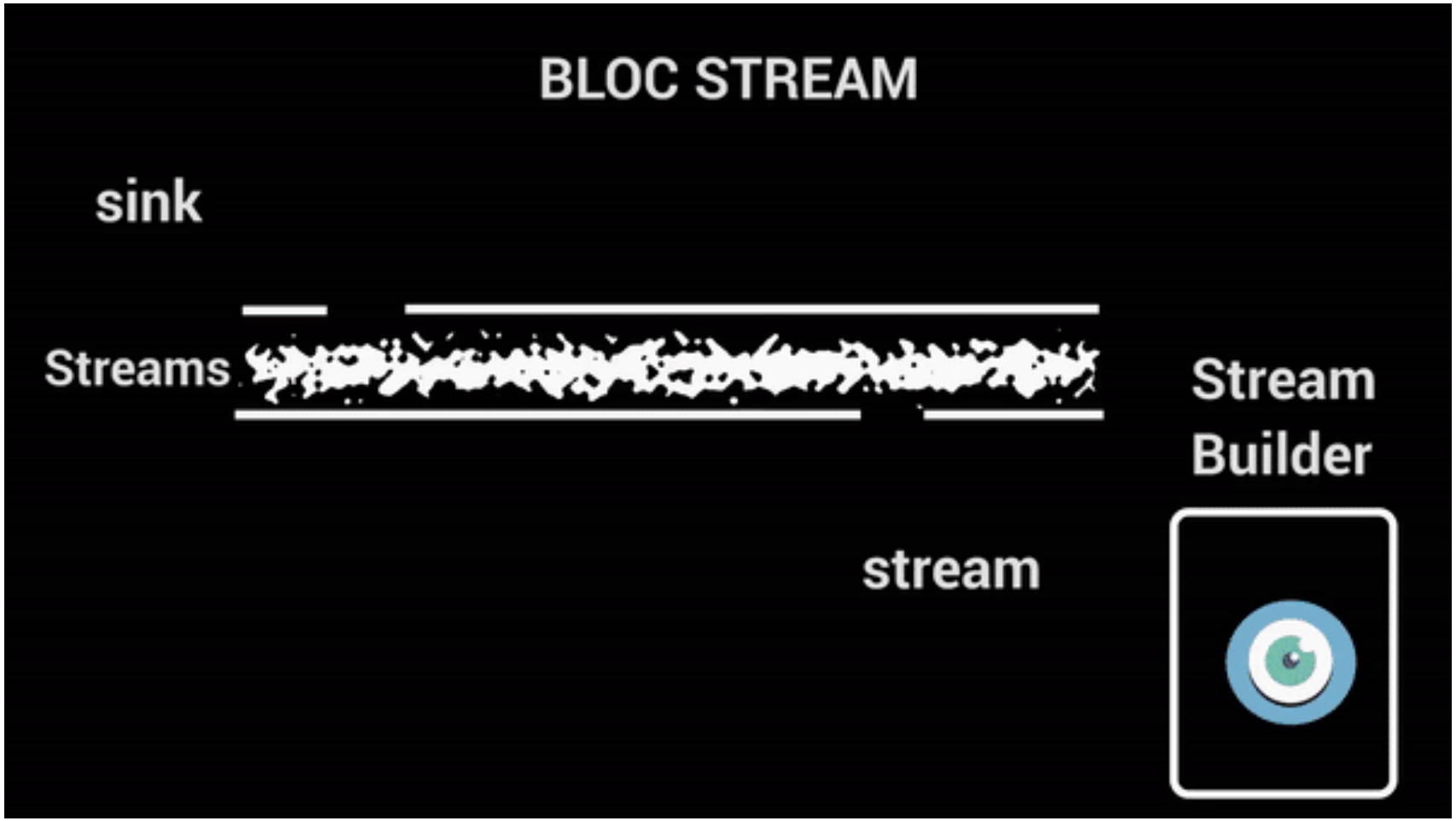
- BLoC stands for **B**usiness **Lo**gic **C**omponent. Before we dive into bloc we need you to understand the **Sink** and **Stream.**

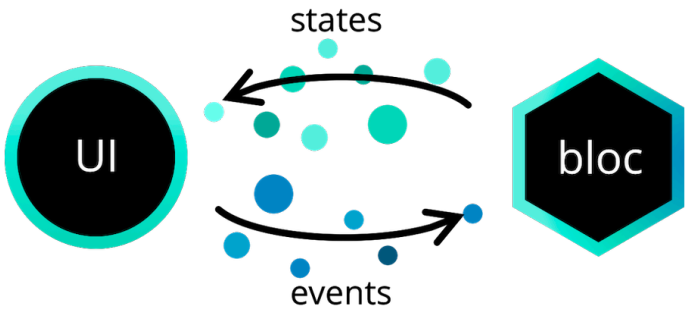




BloC pattern for Flutter

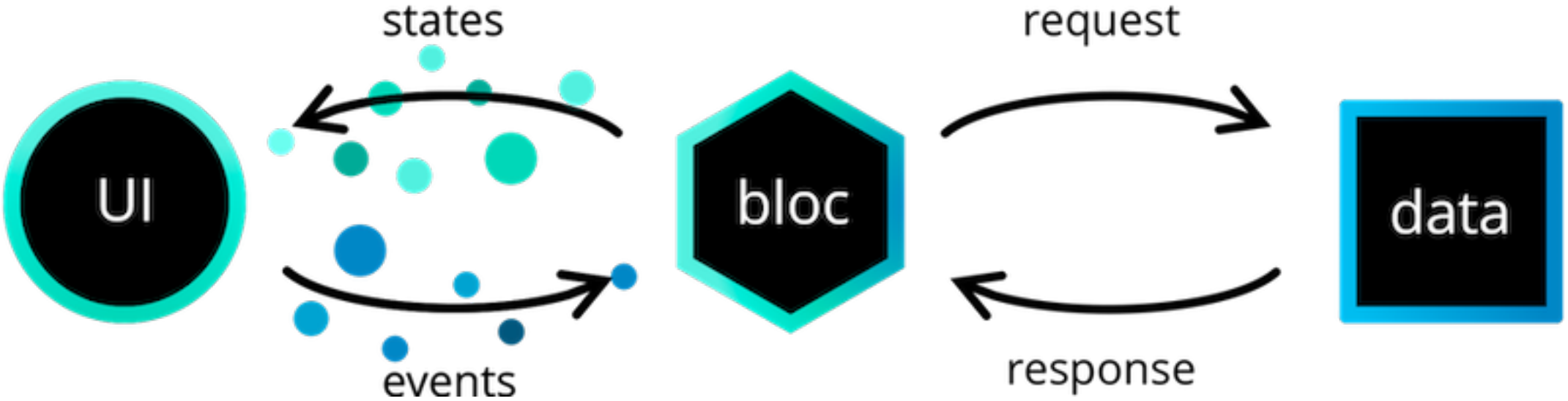
- We add the data into the **Sink** and listen to the streams of data through **Stream** and updated the UI using **StreamBuilder.**





BloC pattern for Getting Started

- **Overview**
  - **bloc** - Core bloc library
  - **flutter_bloc** - Powerful Flutter Widgets built to work with bloc in order to build fast, reactive mobile applications.
  - **angular_bloc** - Powerful Angular Components built to work with bloc in order to build fast, reactive web applications.



BloC pattern for Getting Started

- **Installation**
  - For a Flutter application, we need to add the flutter_bloc package to our pubspec.yaml as a dependency.

```
dependencies:
  flutter_bloc: ^8.0.0
```

- **Import**

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

BloC pattern for Getting Started

Resource Software Solution

- **Streams**
  - For A stream is a sequence of asynchronous data. In order to use the bloc library.
  - We can create a Stream in Dart by writing an **async*** (async generator) function.

```
Stream<int> countStream(int max) async* {
    for (int i = 0; i < max; i++) {
        yield i;
    }
}
```

BloC pattern for Getting Started

- **Creating a Bloc**
  - We must also define the event that the Bloc will be able to process.
  - Events are the **input** to a Bloc

```
abstract class CounterEvent {}

class CounterIncrementPressed extends CounterEvent {}

class CounterBloc extends Bloc<CounterEvent, int> {
  CounterBloc() : super(0);
}
```

BloC pattern for Getting Started

- **State Changes**
  - We can then update the EventHandler to handle the CounterIncrementPressed event

```
abstract class CounterEvent {}

class CounterIncrementPressed extends CounterEvent {}

class CounterBloc extends Bloc<CounterEvent, int> {
  CounterBloc() : super(0) {
    on<CounterIncrementPressed>((event, emit) {
      emit(state + 1);
    });
  }
}
```

BloC pattern for Getting Started

- **Using a Bloc**

```
return MaterialApp(
  theme: ThemeData(
    primarySwatch: Colors.blue,
  ),  // ThemeData
  home: BlocProvider<_CounterBloc>(
    create: (context) => _CounterBloc(),
    child: const _CounterPage(title: 'St
  ),  // BlocProvider
);  // MaterialApp
```

```
final counterBloc = BlocProvider.of<_CounterBloc>(context);
return Scaffold(
  appBar: AppBar(
    title: Text(title),
  ),  // AppBar
  body: Center(
    child: Column(
      mainAxisAlignment: MainAxisAlignment.center,
      children: [
        BlocBuilder<_CounterBloc, int>(builder: (context, count) {
          return Text('$count', style: Theme
              .of(context)
              .textTheme
              .headline1,);  // Text
        })  // BlocBuilder
      ],
    ),  // Column
  ),  // Center
```

setState vs BloC

- The setState(){}) is used to manage local state in the same StatefulWidget and it's child .
- BLoC pattern is used to manage global state.
- if you want to pass data from B2 to A?
  - Using StatefulWidget you should pass data from B2 to B1 to B to A.
  - Using BLoC pattern to manage global state you pass data from B2 to A directly.