Flutter Fundamentals
Bloc

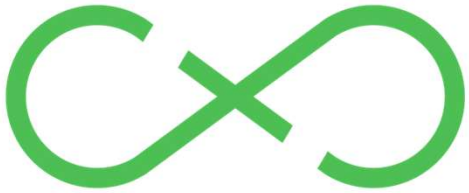
Peter Kassenaar –
info@kassenaar.com

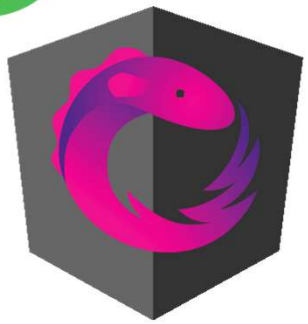
Managing State with Bloc

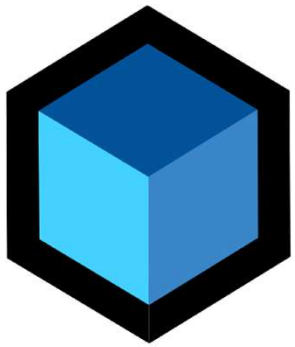
More ways of using *global* state in different screens in the app

What is State Management?

- Various design patterns, used for managing *state* (data in its broadest sense!) in your application.
- Multiple solutions possible – depends on application & framework

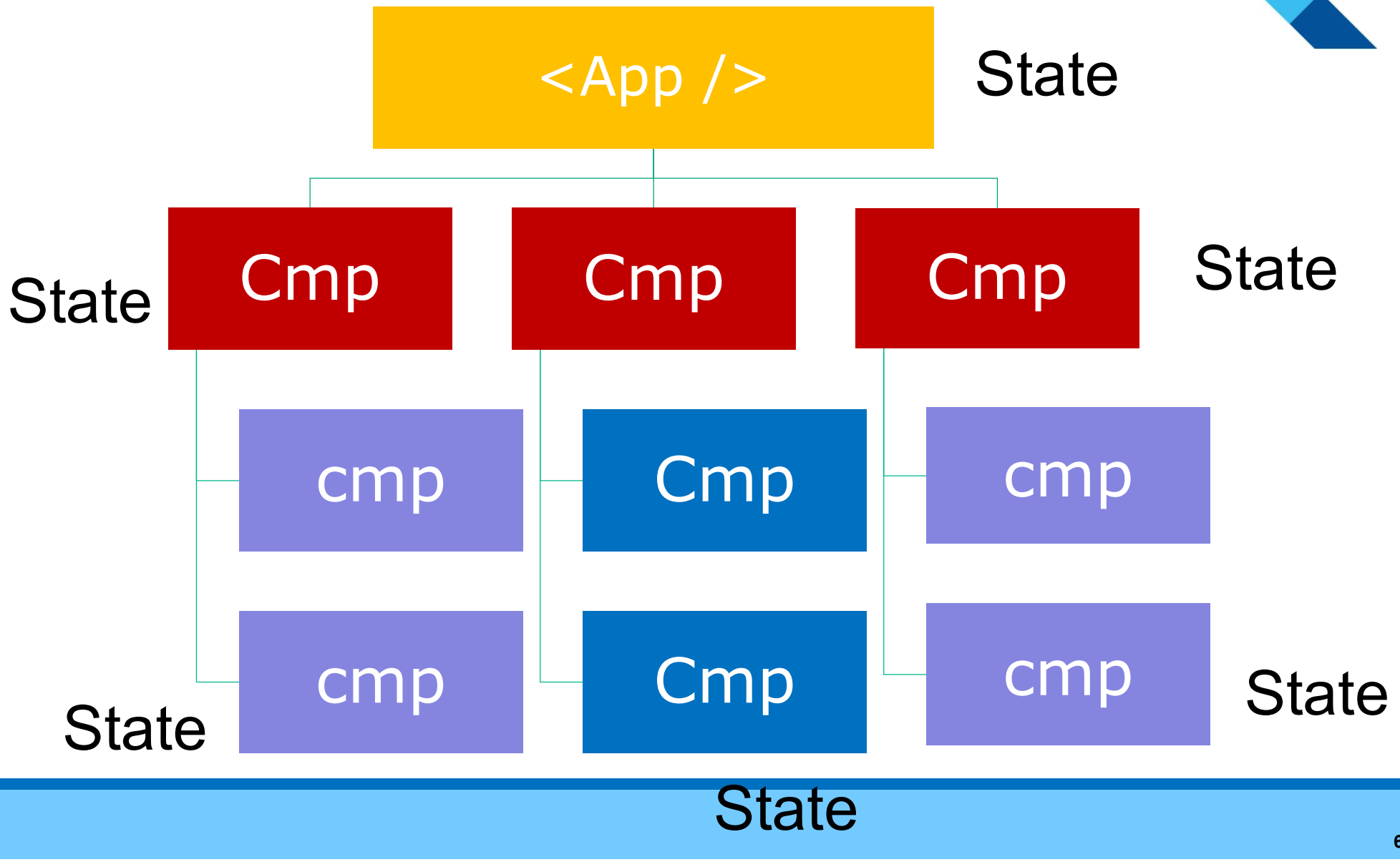






bloc

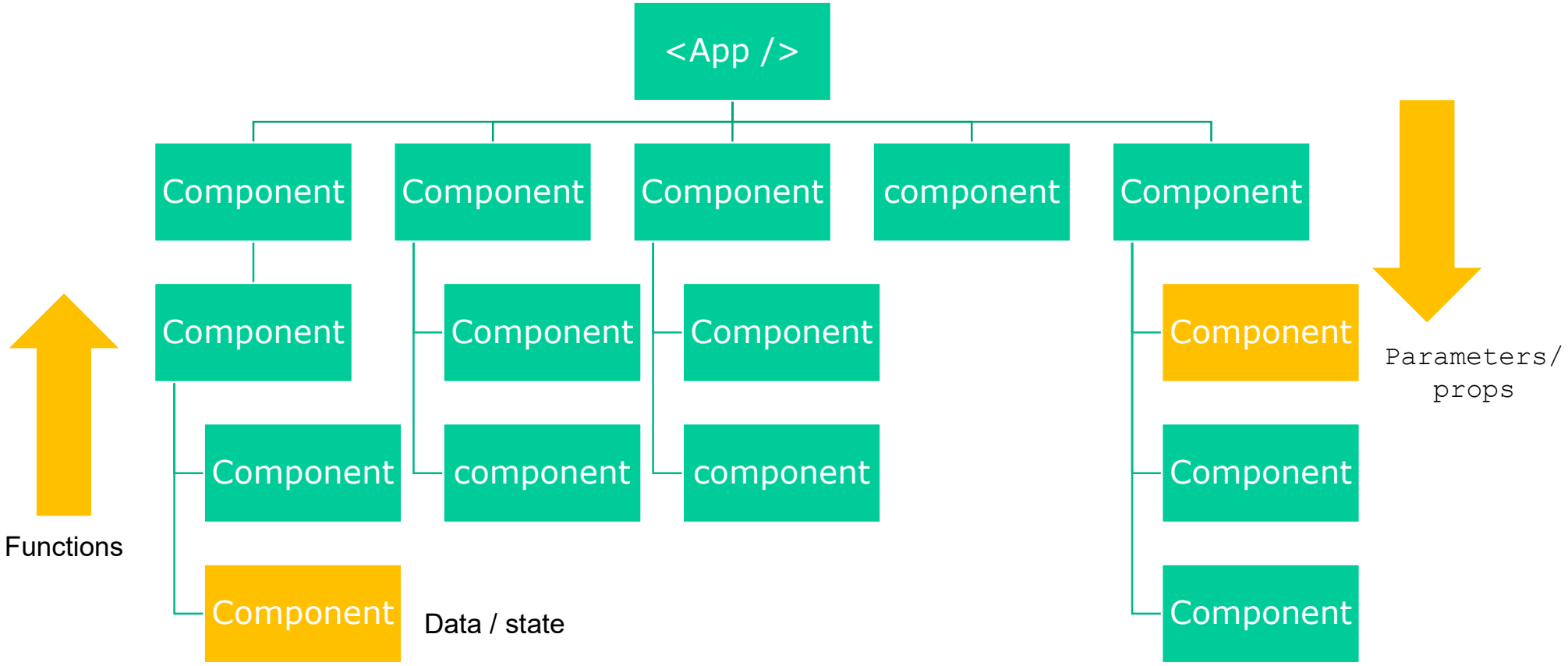
State management without a store



Without a State Management solution

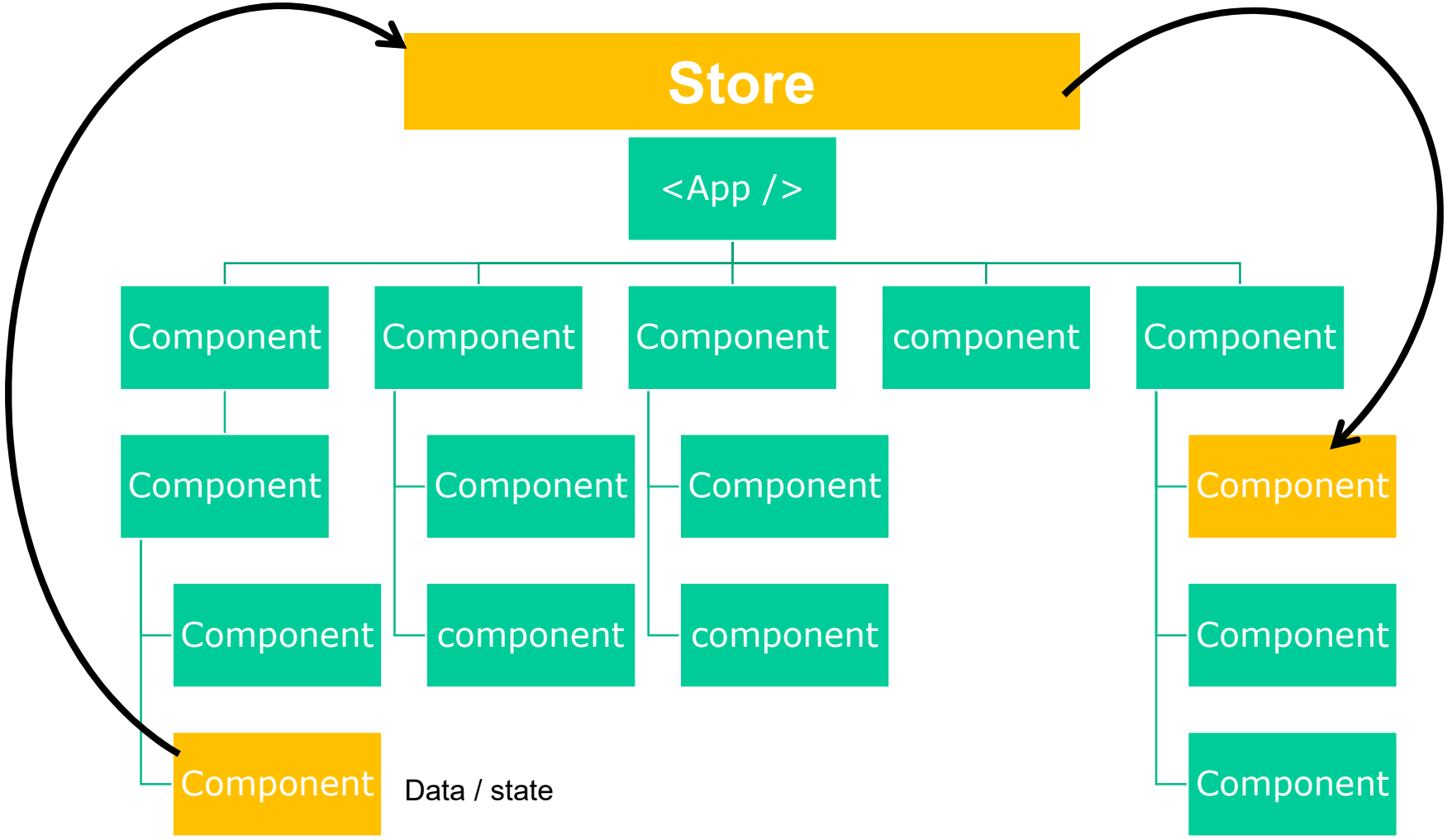
- This is not bad per se, but can lead to:
  - Confusion
  - Errors
  - Duplicating code
  - Maintenance nightmare
  - …

Data flow in complex applications

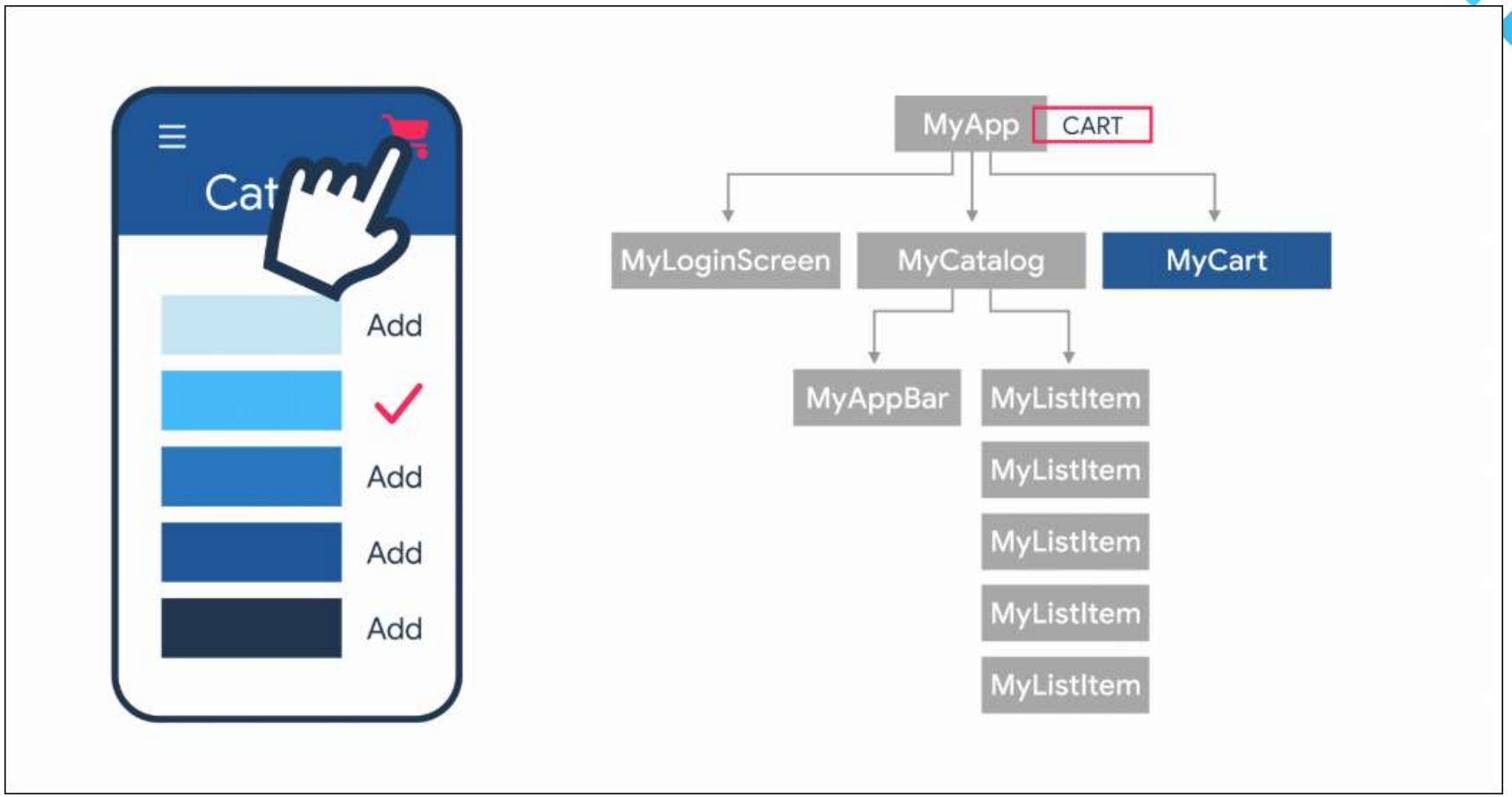


We don't want this…. Not very scalable

State management *with* a store



From the Flutter docs:



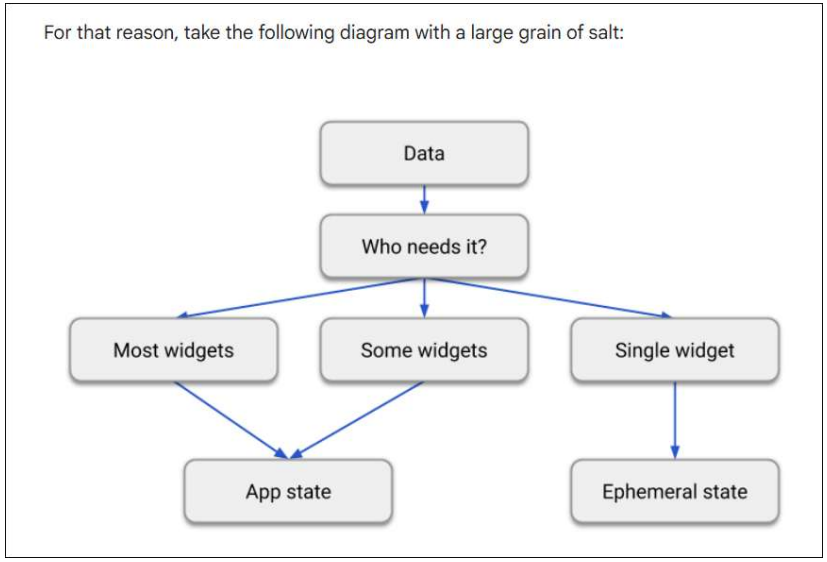
https://docs.flutter.dev/data-and-backend/state-mgmt/intro

Typically: two types of state

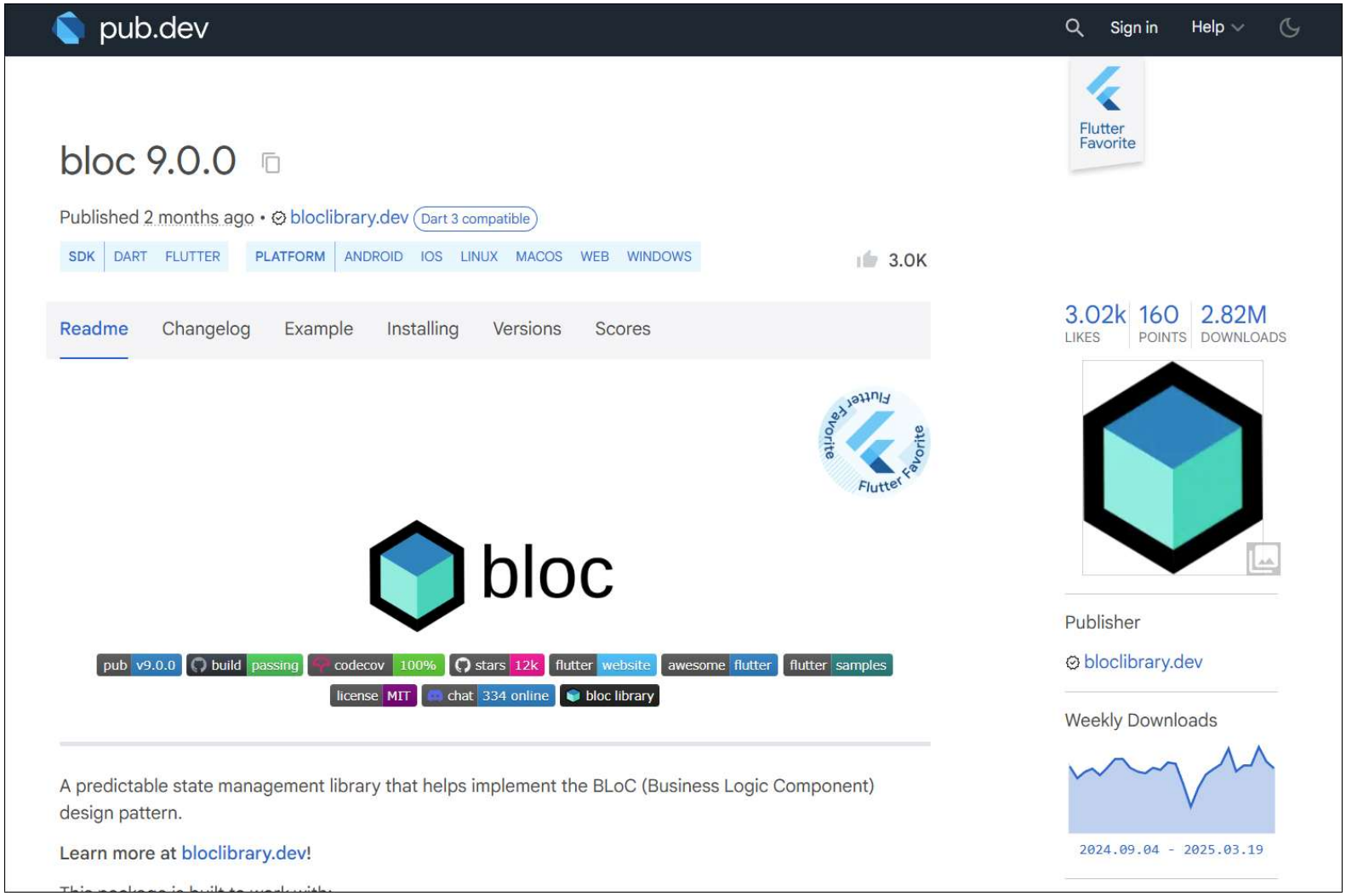
- Local state (= in components)
  - Also: “ephemeral state”, or “UI-state”
  - Usage: statefull widgets
  - Examples: current page, counter, selected tab in Navigation, etc.
- App State (= data in a state management solution)
  - Examples: user preferences, login state + info, shopping cart in your app, favorites/liked articles, etc.
- Various solutions possible in Flutter:
  - `Provider()`, `bloc`, `rxDart`, and (much!) more

More general info on state management:

- https://docs.flutter.dev/data-and-backend/state-mgmt/ephemeral-vs-app
- "Flutter State Management - The Grand Tour", https://www.youtube.com/watch?v=3tm-R7ymwhc

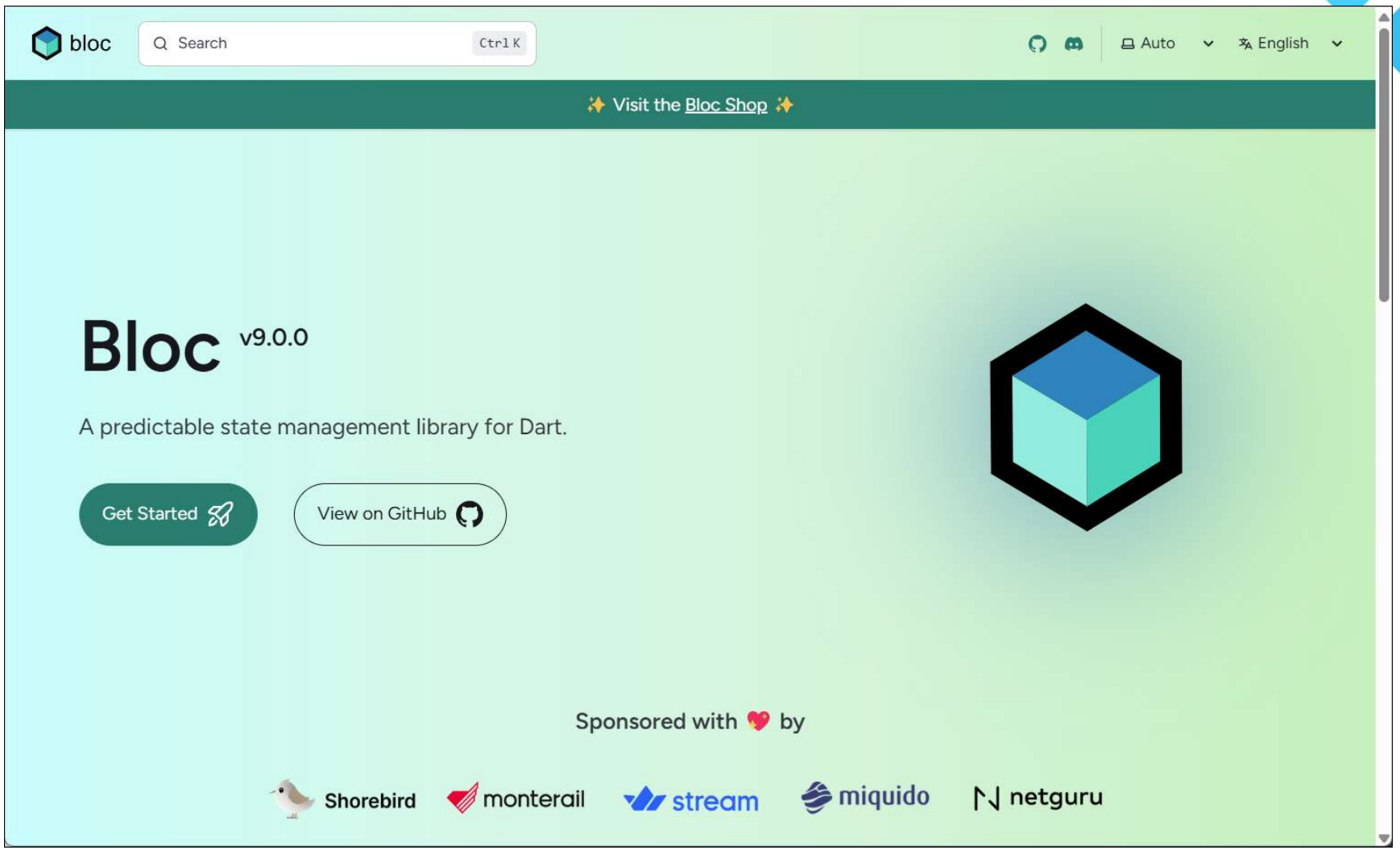


Our choice: `bloc`



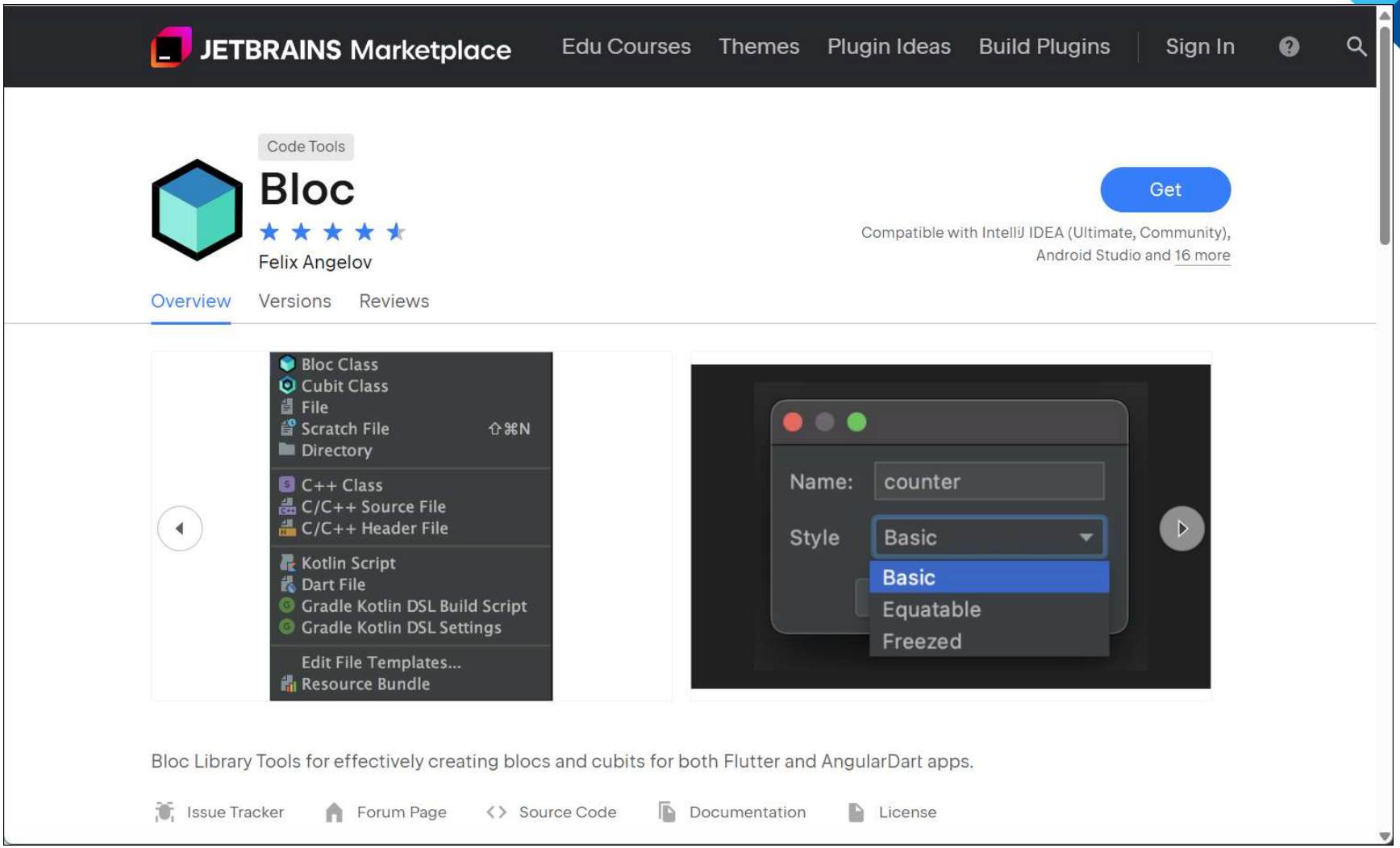
https://pub.dev/packages/bloc

General info on bloc



https://bloclibrary.dev/

Official bloc plug-in for IntelliJ



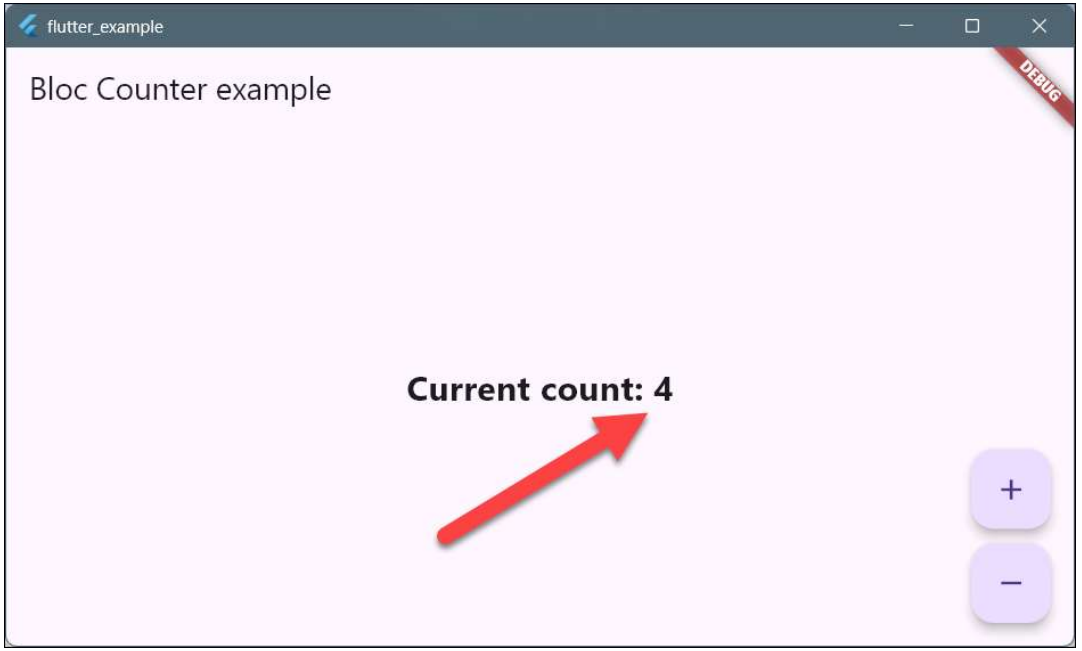
Disadvantage on using state management/bloc:

Warning:

bloc == complex stuff!

Simple example to get familiar with bloc

- Getting to know the terminology
- Simple Counter example
  - However, counter state is now in a `BlocProvider()` NOT in the widget anymore.



Lots of steps:

1. Install bloc in `pubspec.yaml`
2. Create a `(Multi)BlocProvider()`
3. Create `counter_bloc.dart`
4. Create `counter_state.dart`
5. Create `counter_event.dart`
6. Create `counter_page.dart`

Finally: show the actual state + content!

Best practice: `lowercase_plus_underscore`

example: `../examples/_400-bloc`

1. Installing `bloc`

- You only need to install `flutter_bloc x.x.x`
  - It will come with the default `bloc` and `cubit` packages
- `flutter pub add flutter_bloc`

```
#pubspec.yaml
# …
dependencies:
  flutter:
    sdk: flutter
  flutter_bloc: ^9.1.0
  http: ^1.3.0
```

Or: add manually and run `flutter pub get`

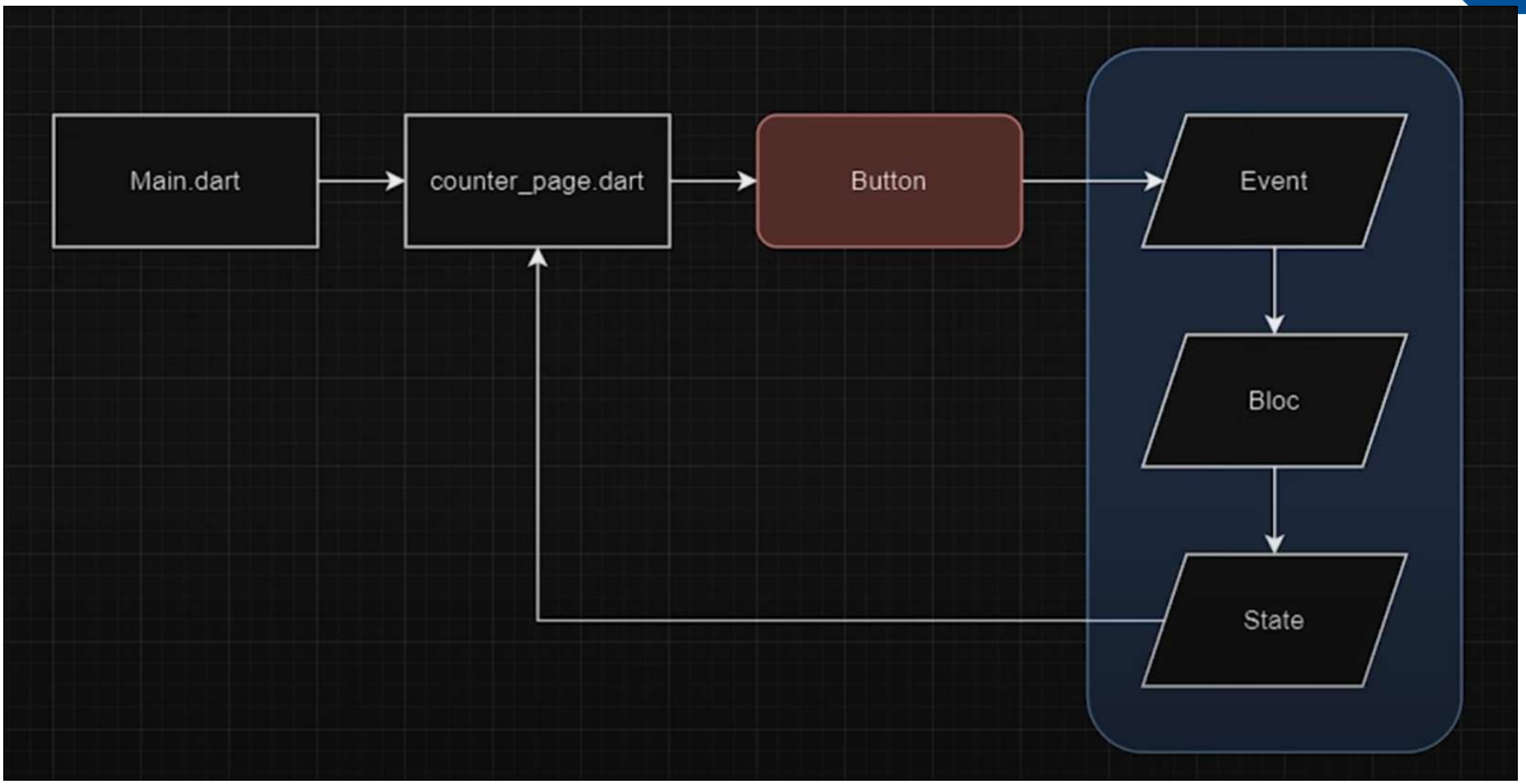
Create `(Multi)BlocProvider()`

- Inside `main.dart`, wrap your pages in a `BlocProvider()`
  - or `MultiBlocProvider()` if you have more providers – which is often the case

```dart
// main.dart
@override
Widget build(BuildContext context) {
  return MaterialApp(
    // 1. Using MultiBlocProvider().
    home: MultiBlocProvider(
      providers: [BlocProvider(create: (context) => CounterBloc())],
      child: const CounterPage(), // CounterPage now has acces to all state!
    ),
  );
}
```

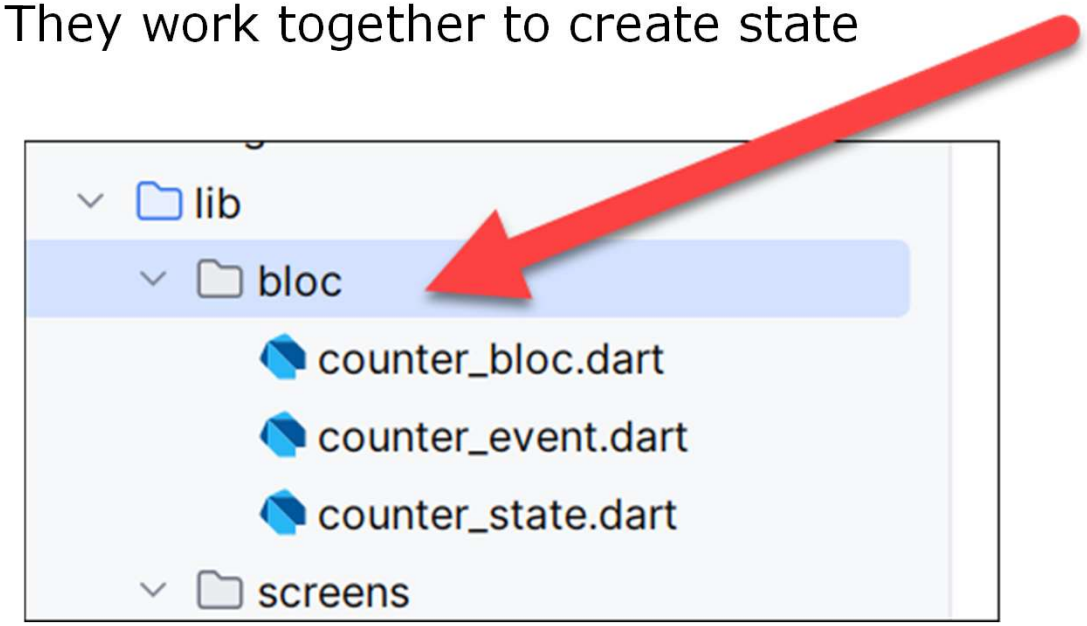
Will be created!

State management: one-way dataflow



Files in application

- Note that `bloc`, `event` and `state` are all inside the same rectangle
  - You need all of them in your app!
  - They work together to create state



2. Creating `counter_bloc.dart`

- Your `bloc`-page LISTENS to events and UPDATES the state
- It must therefore extend from these classes
  - They will be created in a minute!
- When an event listener kicks in, it emits an event with the new state.

Example `bloc` page

```dart
import 'package:flutter_bloc/flutt                    ;
import 'counter_event.dart';
import 'counter_state.dart';

class CounterBloc extends Bloc<CounterEvent, CounterState> {
CounterBloc() : super(CounterState(0)) {
    on<CounterIncrement>((event, emit) {
      emit(
        CounterState(state.count + 1),
      ); // increment the counter
    });
    on<CounterDecrement>((event, emit) {
      emit(
        CounterState(state.count - 1),
      ); // decrement the counter
    });
}
```

Will be created!

Will be created!

Event listeners, emit the new state

Creating `counter_state.dart`

- `CounterState` is initially very simple
- It holds variable(s) with the state
- Also: NO user interface!

```dart
class CounterState {
  // 1. properties in our state
  final int count;

  // 2. constructor
  CounterState(this.count);
}
```

When an event is triggered, this state isupdated

Creating `counter_event.dart`

- Also pretty simple, it holds the (names of) the events that can be fired
- They extend from an abstract (base) class, so we don't need to manually import them all

```dart
// using an abstract class here
abstract class CounterEvent{}

// All events extend from CounterEvent
class CounterIncrement extends CounterEvent{}
class CounterDecrement extends CounterEvent{}
```

Check for yourself: these classes are used inside `counter_bloc.dart`! To update the state.

Finally: `counter_page.dart`

- Use the state, by wrapping your UI in a `BlocBuilder()` widget
  - This has access to the state and can emit events
- Create a variable to emit the events.
  - NO local state in this widget
  - The `<CounterBloc>` is available, because in `main.dart` we wrapped the entire page in a `BlocProvider()`.

```
final counterBloc = context.read<CounterBloc>();
```

The `BlocBuilder<T>()` to use the state

```
// counter_page.dart
// …
child: BlocBuilder<CounterBloc, CounterState>(
  builder: (context, state) {
    return Column(
      mainAxisAlignment: MainAxisAlignment.center,
      children: [
        Text(
          'Current count: ${state.count}',
          style: TextStyle(…),
        ),
      ],
    );
  },
),
```

Show the current state

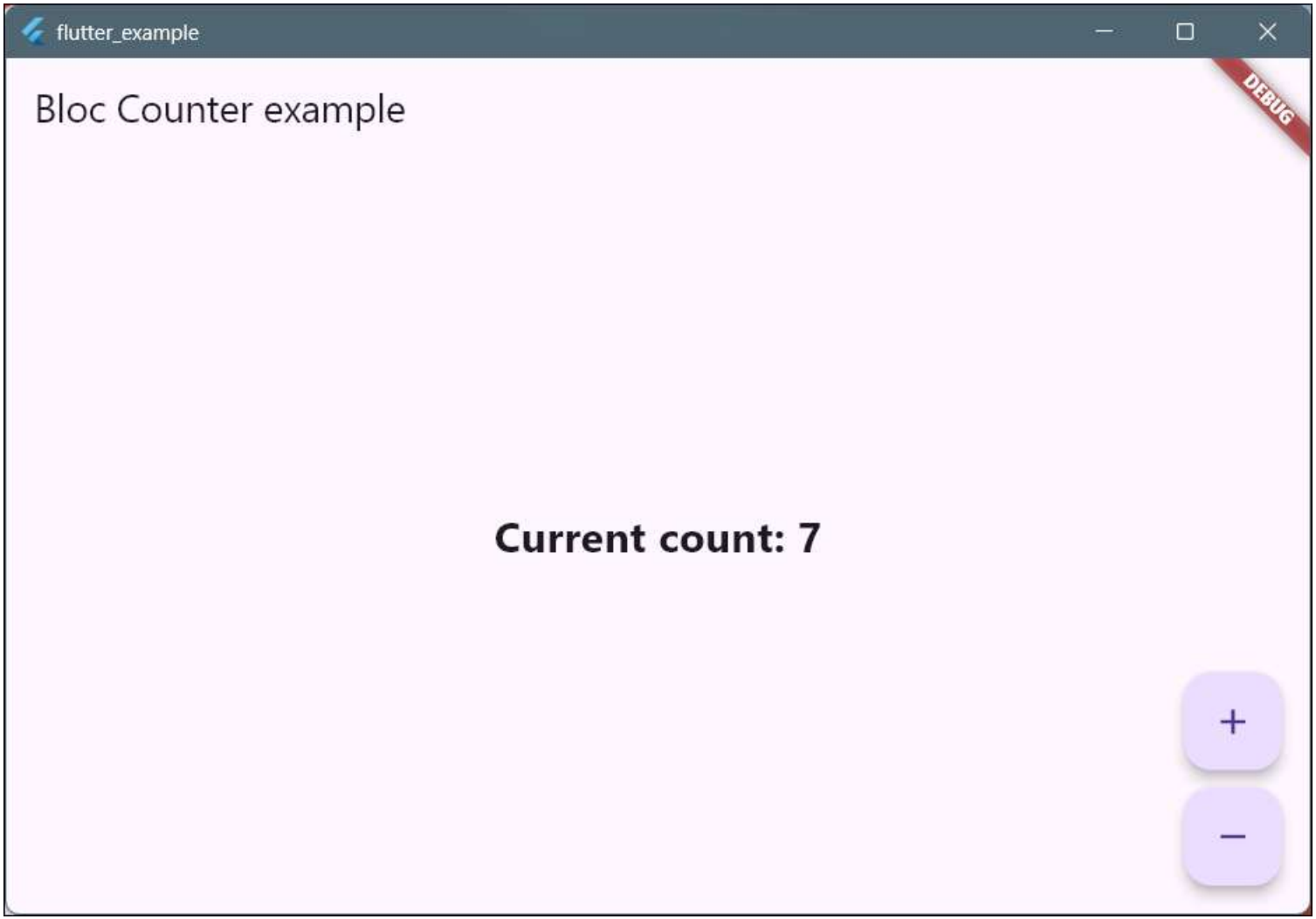
Updating the state by emitting events

- Create buttons that emit events, for instance:

```
floatingActionButton: Column(
  mainAxisSize: MainAxisSize.min,
  children: [
    FloatingActionButton(
      onPressed: () {
        counterBloc.add(CounterIncrement());
      },
      child: Icon(Icons.add),
    ),
    SizedBox(height: 10),
    FloatingActionButton(
      onPressed: () {
        counterBloc.add(CounterDecrement());
      },
      child: Icon(Icons.remove),
    ),
  ],
),
```

Using the variable created before, to emit events

Final Result

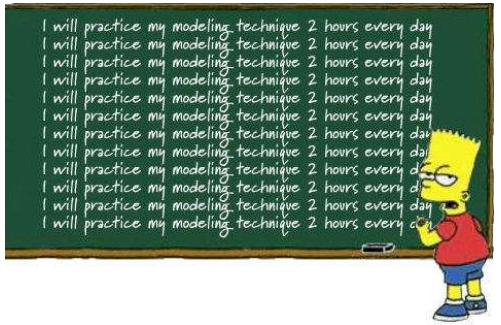


../examples/_400-bloc

Workshop

- Enhance the example with an extra button to reset the state
  - Tip: look at the diagram and follow it clockwise:
  - Add `event`, update `bloc`, update `state`, update `page`, etc.
- Optional: study the example and put variables of your own application in bloc state
- Optional: create a screen with a `TextField()`. Put text in the state, and read it in another screen/widget.
  - Tip: create completely new state/bloc/pages for it!

`../examples/_400-bloc`



Using state in other screens

Benefits of using the state: no more need to pass parameters around

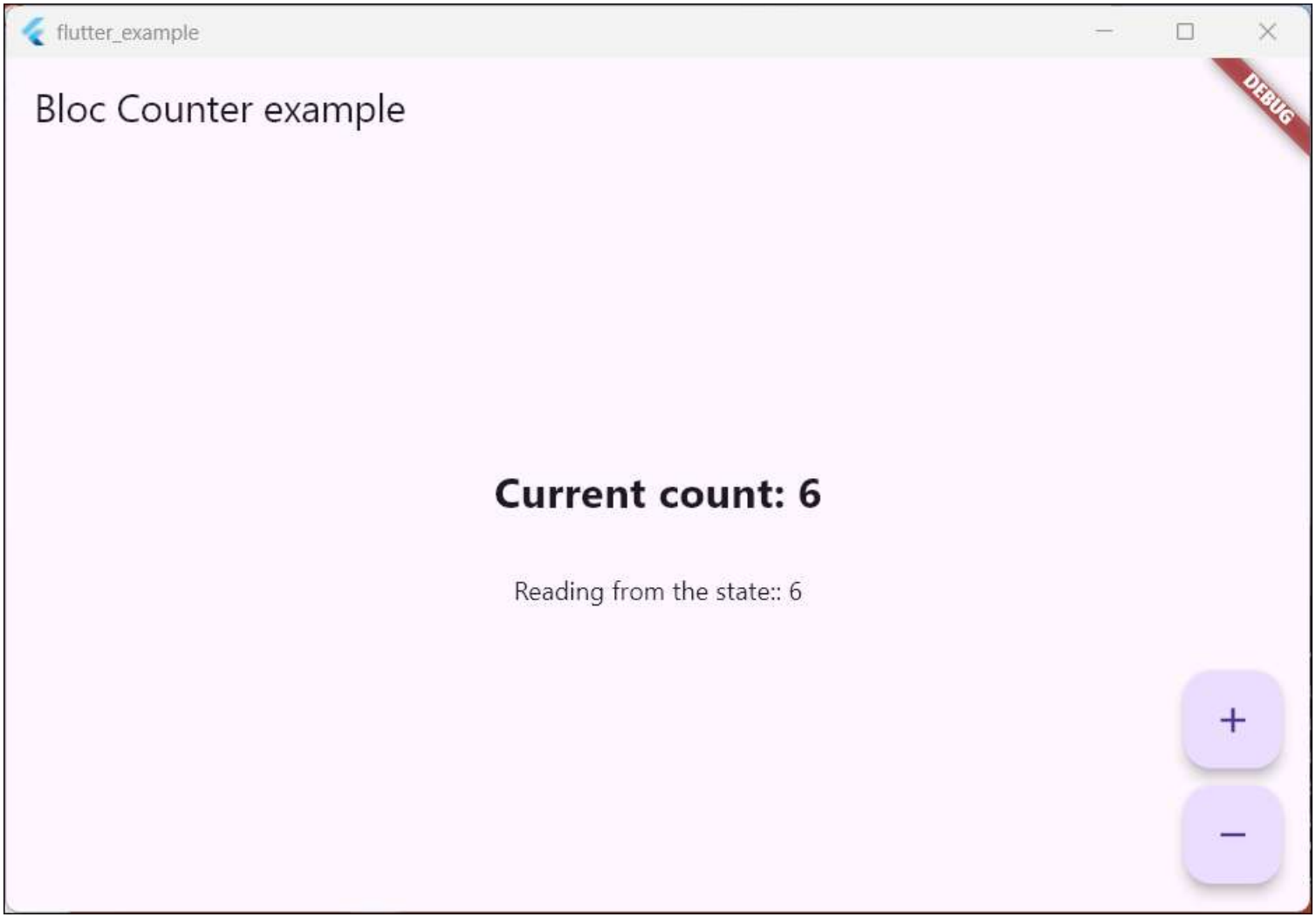
Retrieving state in other screens

- What if you want to read the state in other classes/widgets/screens?
- This is relatively easy:
  - `watch` a specific bloc and retrieve state

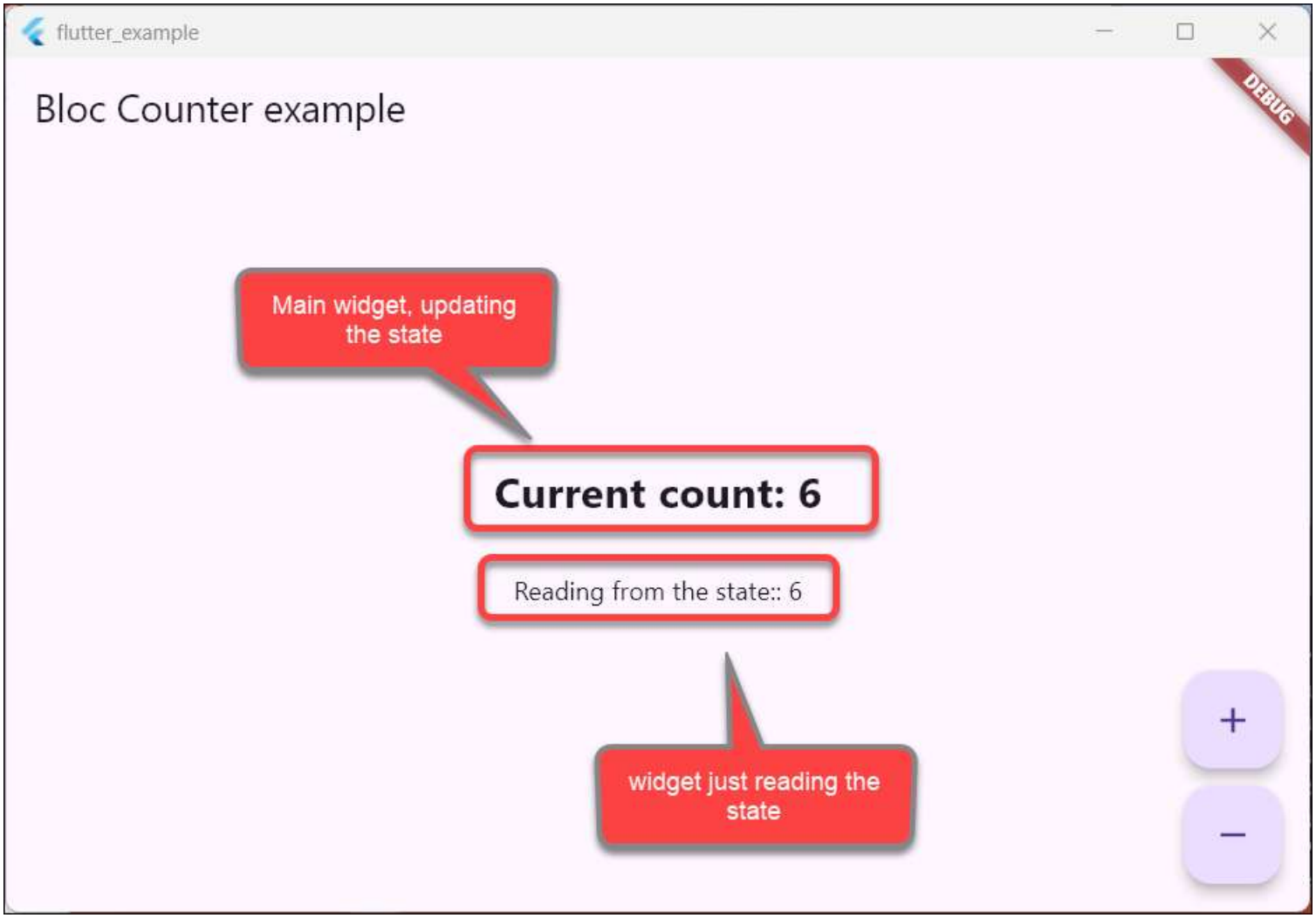
context.watch<T>

```
class _CounterPageReadState extends State<CounterPageRead> {
  @override
  Widget build(BuildContext context) {
    // We just want to *retrieve* the state in this widget
    final currentCounter = context.watch<CounterBloc>().state;

    return Text('Reading from the state:: ${currentCounter.count}');
  }
}
```

Second widget, reading from the state



Second widget, reading from the state



Background on `context.watch`

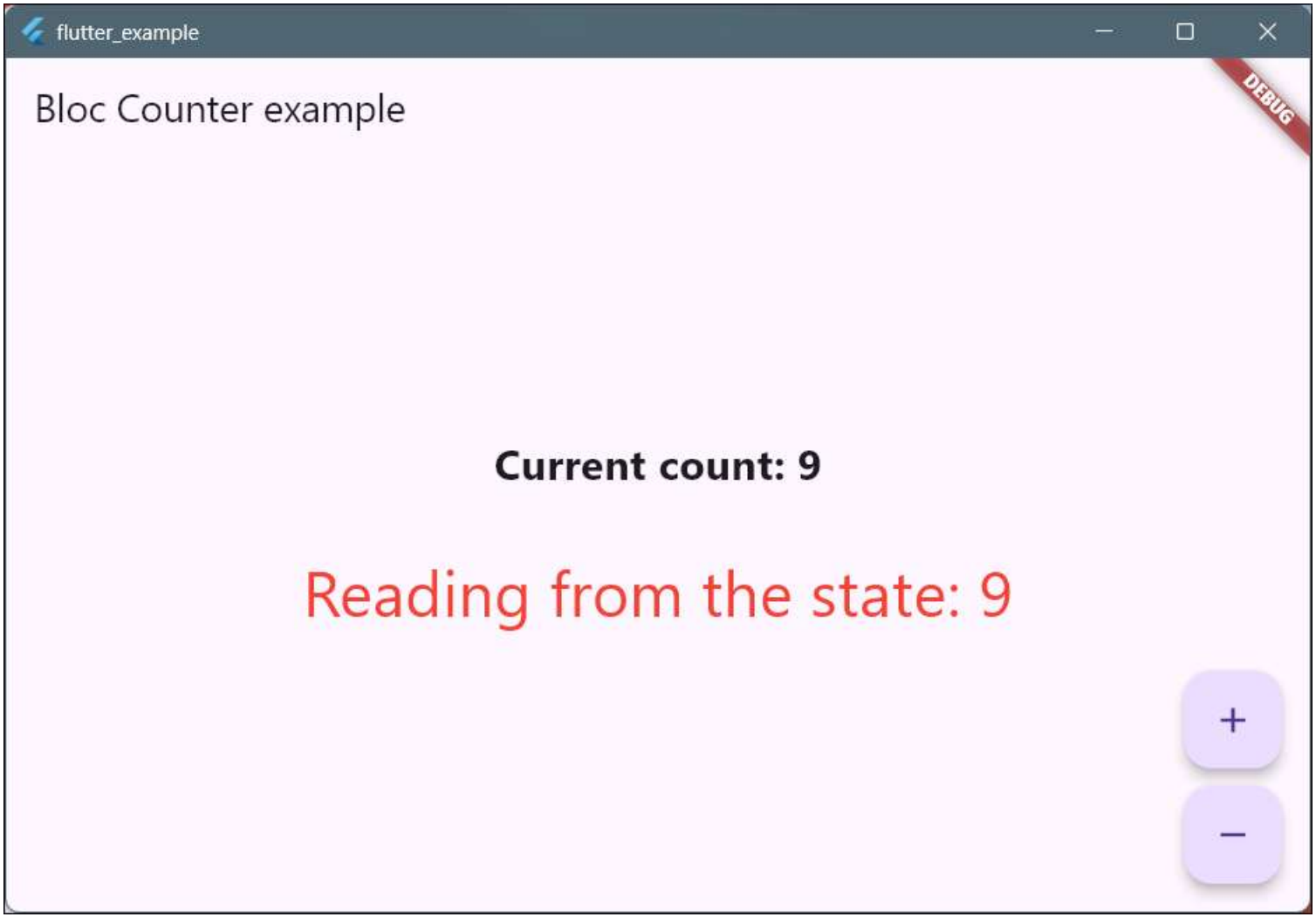
- Using `context.watch`
  - The `context.watch<CounterBloc>()` listens to `CounterBloc` state changes.
  - Anytime the state updates, `build()` will re-trigger, so the latest value of `state.count` is displayed.

Alternative: using `BlocBuilder()`

- Alternative: using `BlocBuilder()` to wrap `Text()`
  - Retrieve `CounterState` directly inside this bloc
- More flexible, but also more complex/boilerplate

```
class _CounterPageReadState extends State<CounterPageRead> {
  @override
  Widget build(BuildContext context) {
    return BlocBuilder<CounterBloc, CounterState>(
      builder: (context, state) {
        return Text(
          'Reading from the state: ${state.count}');
      },
    );
  }
}
```

Results: visually the same



Background on `BlocBuilder()`

- `BlocBuilder()` is designed to react to state changes for a specific bloc.
  - It rebuilds only when state of `CounterBloc` changes.
- Use `BlocBuilder()` instead of `context.watch`?
  - If you want finer control over widget rebuilding or:
  - restrict the part of the widget tree that rebuilds on state changes, `BlocBuilder()` is preferred.
- Both approaches are valid!
  - Choose based on preference and your widget's complexity.

Adding multiple properties to the state

Sometimes you want more than a single property on your bloc

Multiple state properties

- Let's say you also want to keep track of the total number of clicks.
- Expand the State with a property `numClicks`;
- Remember to now use the keyword `required`

```
class CounterState {
  //  Multiple properties in our state
  final int count;
  final int numClicks;

  CounterState({required this.count, required this.numClicks});
}
```

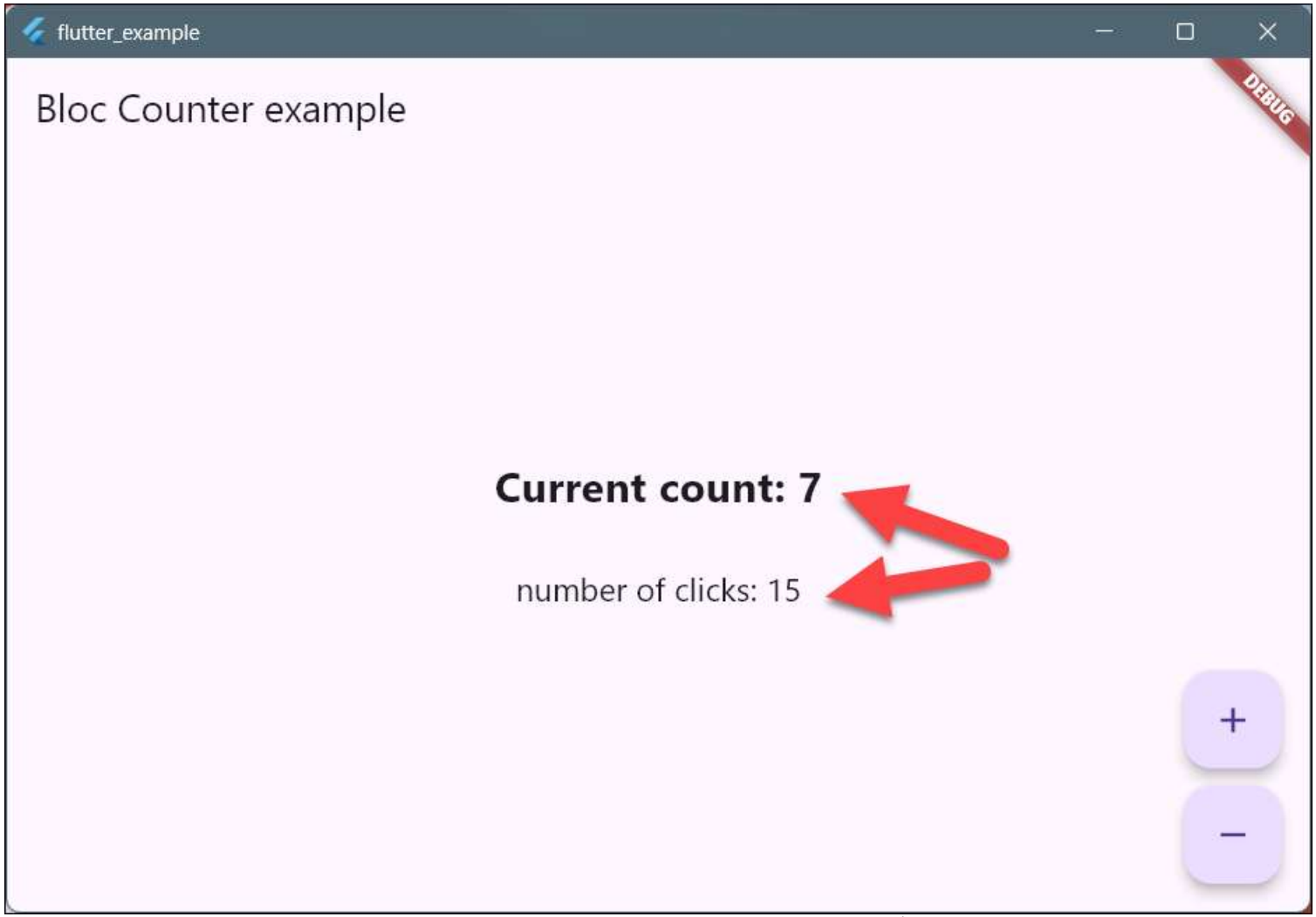
Using multiple state properties

- Update the `CounterBlock` with named properties to initialize and update the state
- Look at the `count` and `numClicks` properties.

```
class CounterBloc extends Bloc<CounterEvent, CounterState> {
  CounterBloc() : super(CounterState(count: 0, numClicks: 0)) {
    on<CounterIncrement>((event, emit) {
      emit(
        CounterState(count: state.count + 1, numClicks: state.numClicks + 1),
      ); // increment the counter
    });
    on<CounterDecrement>((event, emit) {
      emit(
        CounterState(count: state.count - 1, numClicks: state.numClicks + 1),
      ); // decrement the counter
    });
  }
}
```

Extra, named properties

Result

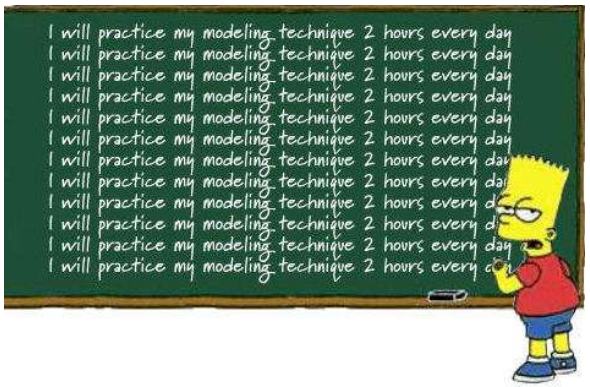


..examples/_420-multiple-properties

Workshop

- Add an extra property to the state (like `numClicks`) and show / update this property
- Study the example provided, or use your own app.
- Optional: create extra `BlocProvider()` with additional state

  (so the states are *independent* of each other) and use them inside `MultiBlocProvider()`



Using payload

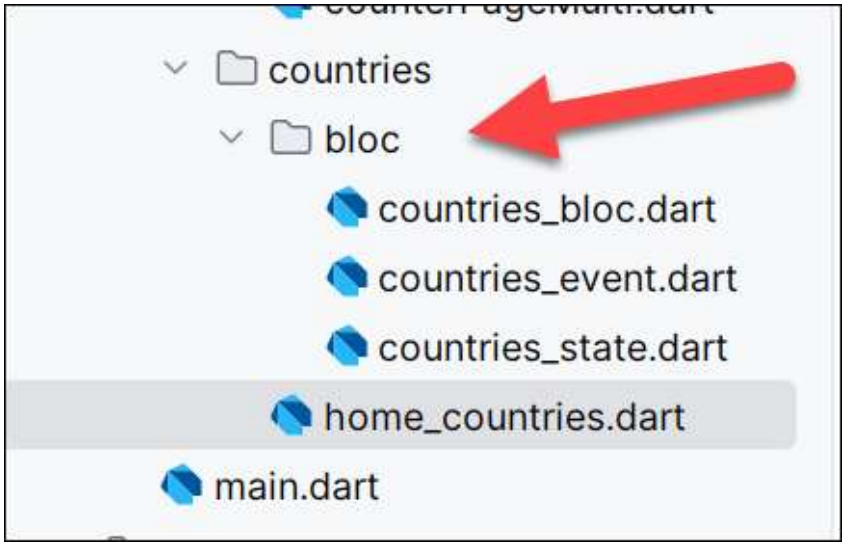
Sending parameters to set or update the state

What is `payload`?

- `payload` is just a term
- It is updated state in the store, often coming from another system, or user input.
  - For instance: let's say you want to update the counter with a variable number, typed in by the user (say: `5`). In this case, `5` is called the `payload`.
- Or, a realistic this scenario:
  1. On startup, we want to retrieve a list of data (e.g. countries)
  2. Other components or screens also need this data
  3. Instead of storing the data in local properties, we put them in the state upon retrieval, so other screens don't have to load the data again
  4. The countries (=data) are then called the `payload`

Again, multiple steps

- Create new `bloc`, `event` and `state` files.



countries_bloc

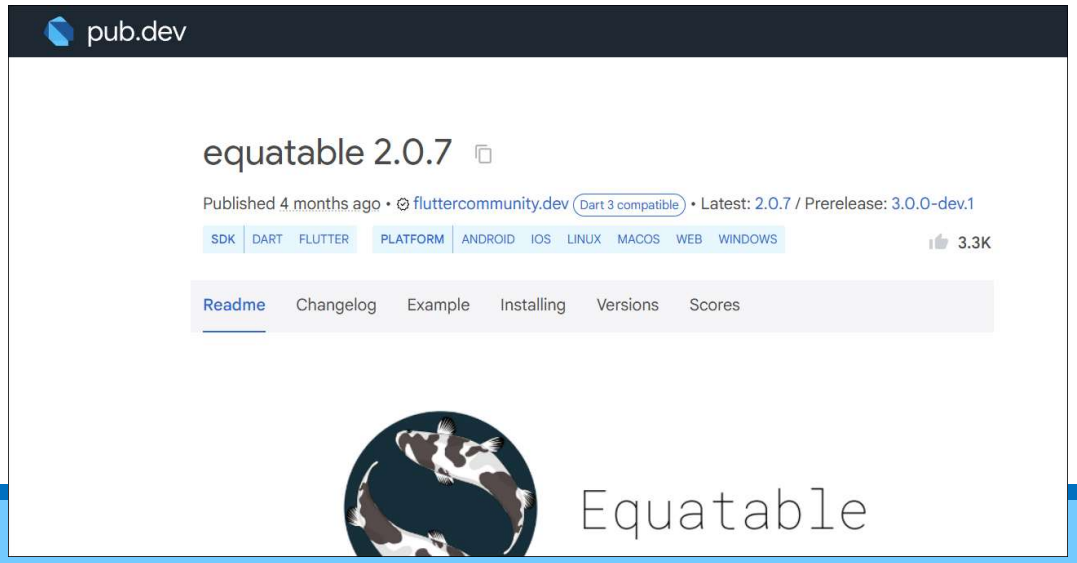
- Listen for event, execute function (called `FetchCountries`)
- Emit events on
  - starting loading (`CountriesLoading()`),
  - loading success (`CountriesLoaded()`)
  - Loading error (`CountriesError()`)
- Events will be created in the next step!
- Don’t glance at the next code, study it! ☺

1/3 countries_bloc.dart – multiple steps!

```dart
class CountriesBloc extends Bloc<CountriesEvent, CountriesState> {
  CountriesBloc() : super(CountriesInitial()) {
    // 1. Listen to the FetchCountries event
    on<FetchCountries>((event, emit) async {
      emit(CountriesLoading()); // emit the countries loading event.

      // 2. using the async/await notation here, therefore we can use try/catch
      try {
        final response = await http.get(Uri.parse(
            'https://restcountries.com/v3.1/all?fields=name,capital,flags'
        ));
        if(response.statusCode ==200){
          List countries = jsonDecode(response.body);
          emit(CountriesLoaded(countries)); // success. Emit CountriesLoaded()
        }
        else{
          emit(CountriesError('Failed to fetch countries')); // error. Emit error message
        }
      }
      catch(e) {
        emit(CountriesError('An error occurred: $e')); // General error: emit message
      }
    });
  }
}
```

2/3 `countries_event.dart`

- Notice the `extends Equatable` class
- The extra package `equatable` does deep comparison of objects
  - Not only if object is the same, but also if contents are the same!
  - https://pub.dev/packages/equatable



Class `countries_event.dart`

- This class overrides the `get() props` method

"In Dart, by default, the == operator compares object references, not their content. This means that two instances of the same class with identical fields are not considered equal unless you explicitly override the == operator and hashCode. The `equatable` package automates this process"

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

// CountriesEvent: our base class for events
abstract class CountriesEvent extends Equatable {
  @override
  List<Object?> get props => [];
}

// Event: fetching all countries from the API
class FetchCountries extends CountriesEvent {}
```

3/3 countries_state.dart

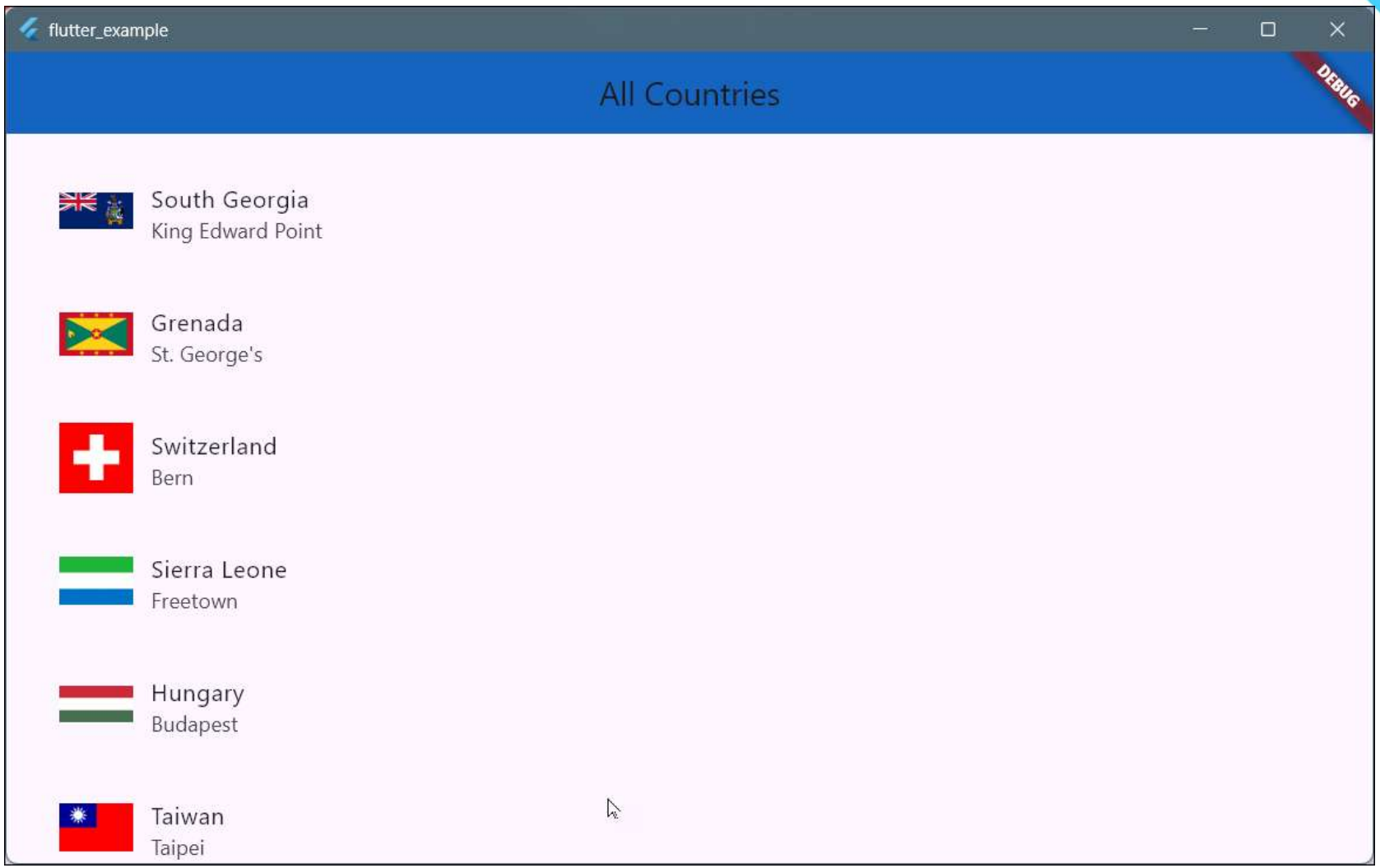
- Add possible events to the state
- Again, using `equatable` package, see `@override`

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

class CountriesState extends Equatable{
  @override
  List<Object?> get props => [];
}

// more states …

// State property to hold the successfully fetched list of countries
class CountriesLoaded extends CountriesState {
  final List countries;

  // constructor
  CountriesLoaded(this.countries);

  @override
  List<Object?> get props => [countries];
}
```

Showing results, calling Event on startup

- Create the UI as normal, but instead use `BlocProvider()` to read from the state
- Use different event types to show different content
  - Progress indicator on `CountriesLoading`
  - List of countries on `CountriesLoaded`
  - Message on `CountriesError`
- Results are visually the same!

Results


flutter_example
All Countries
DEBUG
South Georgia
King Edward Point
Grenada
St. George's
Switzerland
Bern
Sierra Leone
Freetown
Hungary
Budapest
Taiwan
Taipei

Sample code `home_countries.dart`

```
body: BlocProvider(
  create:
      (context) =>
          CountriesBloc()..add(
            FetchCountries(),
          ), // Automatically fetch countries on load,
  child: BlocBuilder<CountriesBloc, CountriesState>(
    builder: (context, state) {
      if (state is CountriesLoading) {
        return Center(child: CircularProgressIndicator());
      } else if (state is CountriesLoaded) {
        return ListView.builder(…)
      …
      …
  }
}
```

_430-payload/../home_countries.dart

Flow

- On startup, call `FetchCountries` using the cascade operator
  - https://dart.dev/language/operators#cascade-notation

```dart
var paint =
    Paint()
      ..color = Colors.black
      ..strokeCap = StrokeCap.round
      ..strokeWidth = 5.0;
```

```dart
(context) =>
          CountriesBloc()..add(
            FetchCountries(),
          ), // Automatically fetch, countries on load
```

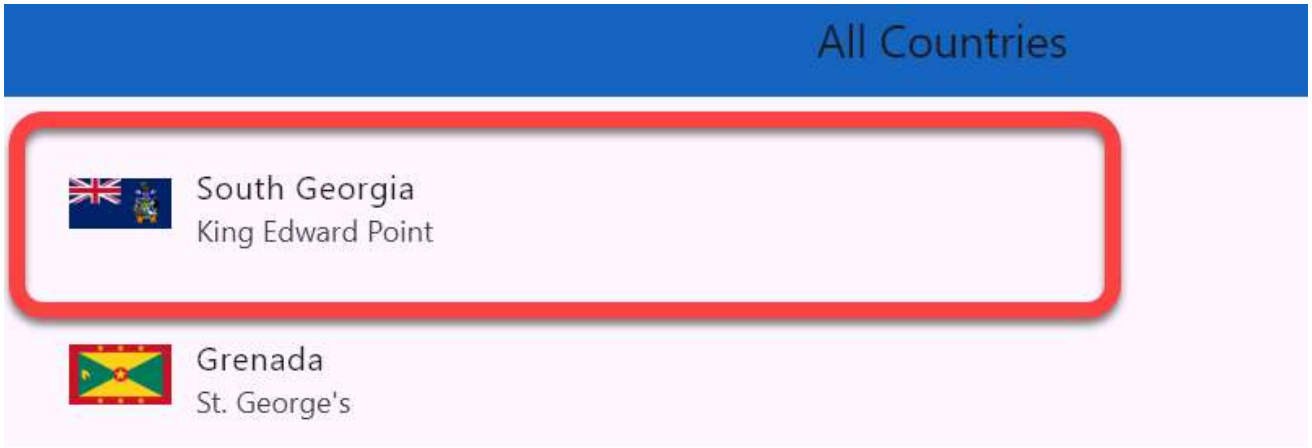
The constructor, `Paint()`, returns a `Paint` object. The code that follows the cascade notation operates on this object, ignoring any values that might be returned.

The previous example is equivalent to this code:

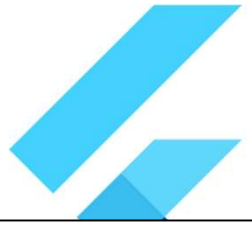
```dart
var paint = Paint();
paint.color = Colors.black;
paint.strokeCap = StrokeCap.round;
paint.strokeWidth = 5.0;
```

Fetch successful?

- When fetching countries successful, create a `ListView.builder()`
- Inside the `itemBuilder()`, loop over countries, create a `ListTile()` containing the requested data



Example `ListView.builder()`



```
else if (state is CountriesLoaded) {
  return ListView.builder(
    itemCount: state.countries.length,
    padding: EdgeInsets.all(10.0),
    // Function to build the items in the ListView
    // See https://api.flutter.dev/flutter/widgets/ListView-class.html for more info
    itemBuilder: (BuildContext context, int index) {
      final country = state.countries[index];
      return Padding(
        padding: EdgeInsets.all(10.0),
        child: Column(
          children: <Widget>[
            ListTile(
              leading: Image.network(country['flags']['png'],),
              title: Text(country['name']['common']),
              subtitle: Text(
                country['capital']?.first ?? 'No capital found.',
              ),), …
…
```

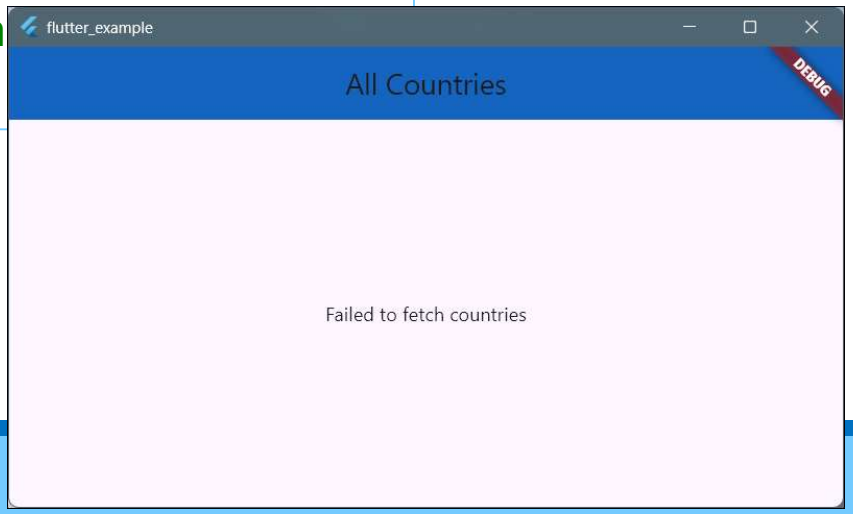
Fetch unsuccessful?

- If fetching was not successful,

  `emit(CountriesError(…))` was thrown

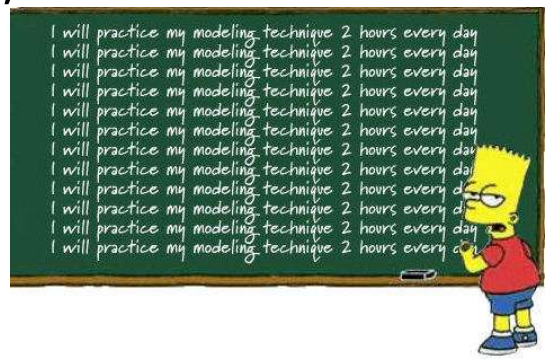
- Show the error on the page

```
…
} else if (state is CountriesError) {
  return Center(child: Text(state.message));
} else {
  return Center(child: Text('No data
}
```

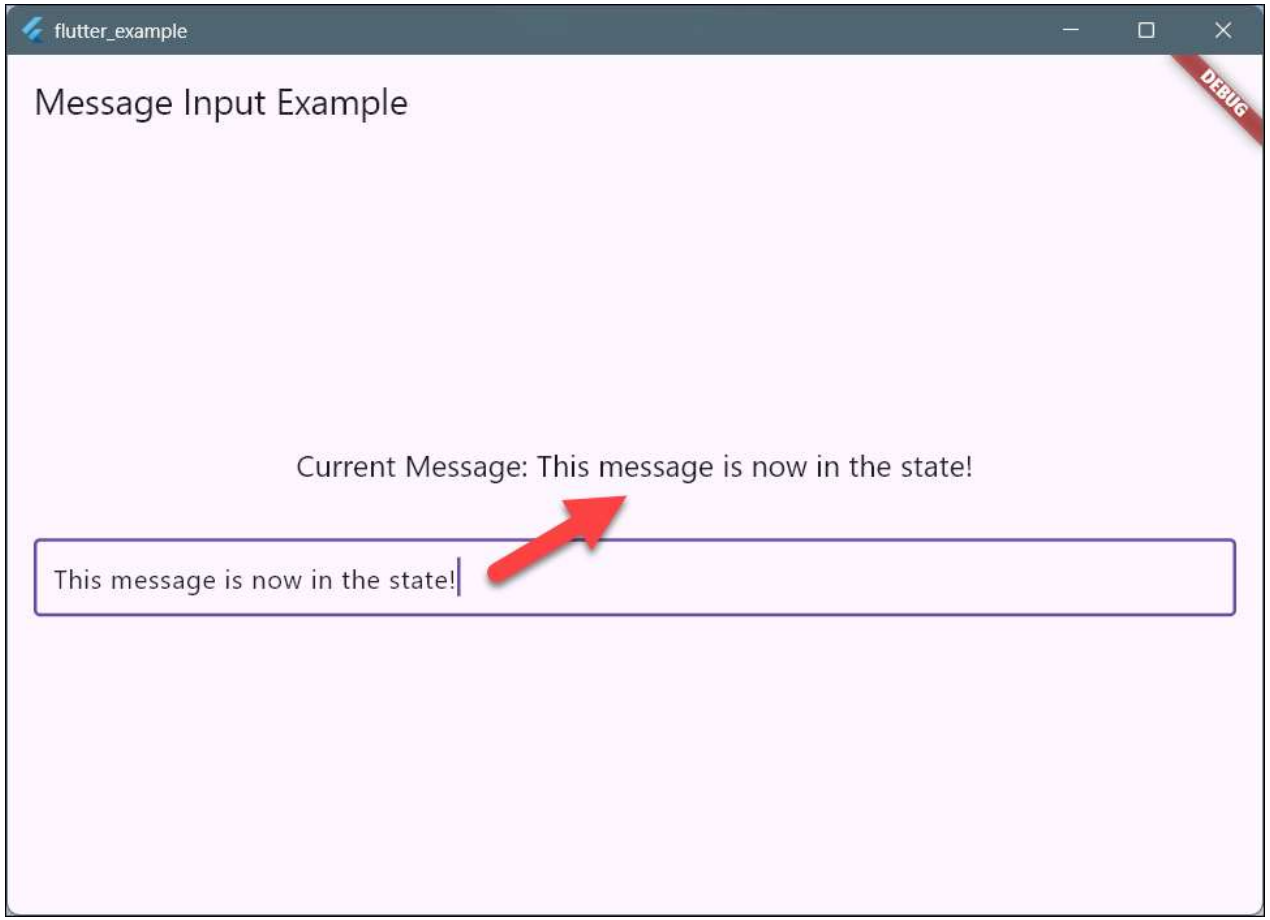


Workshop

- Create a new message page, with the following requirements:
  - A user can type in a message in a `TextField()`
  - The text typed in, is put in the state with a button press
  - The widget reads the text from the state
  - Another widget also reads the text from the state
- Use the structure with `page`, `event`, `state` and so on.
- A possible solution is already in ../`_430-payload`, but first try it yourself!



Example output:

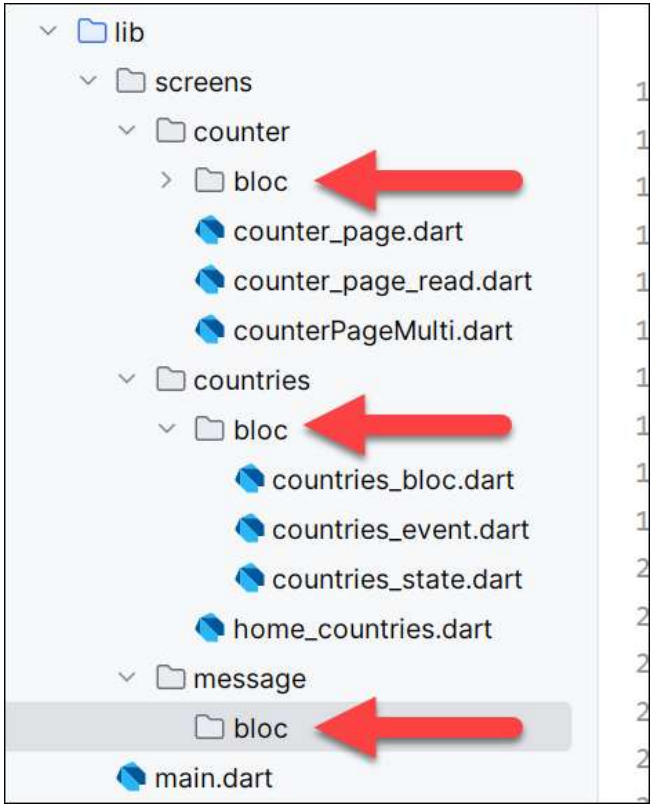


Tips on app structure

How do you structure your app using blocs?

Multiple actions – multiple bloc's

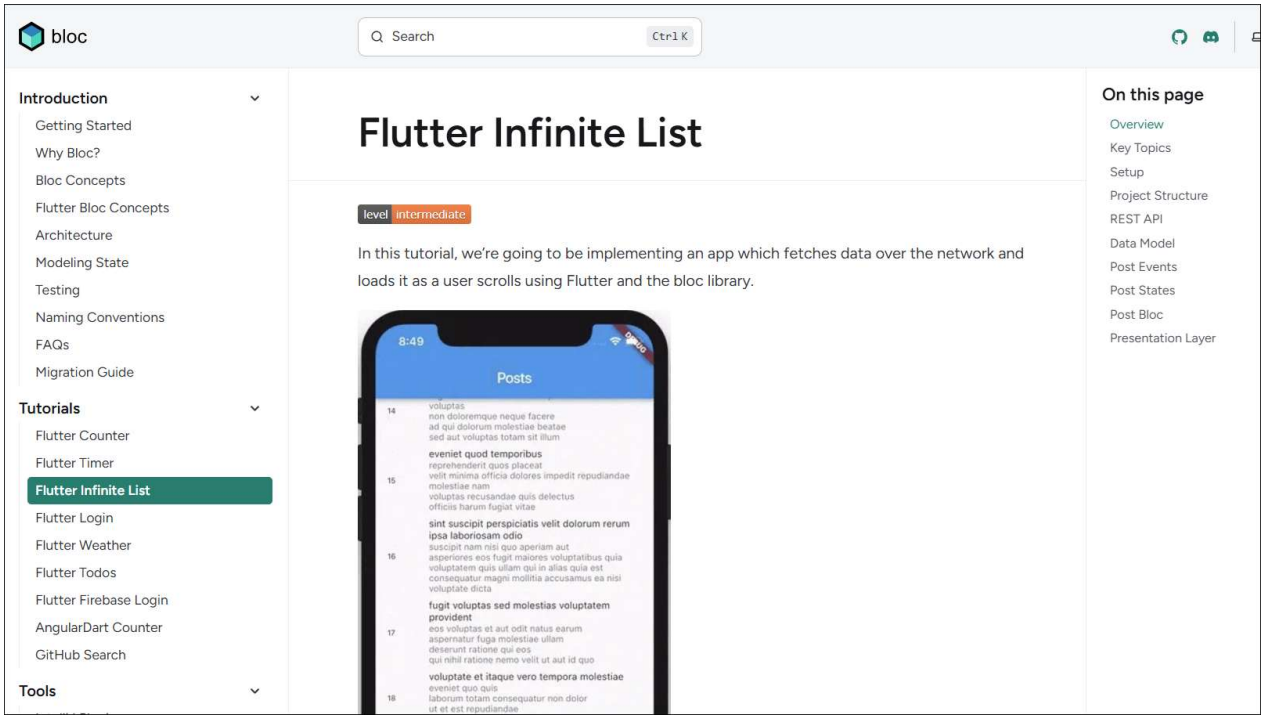
- When having multiple actions or screens, each action has its own associated bloc directory:



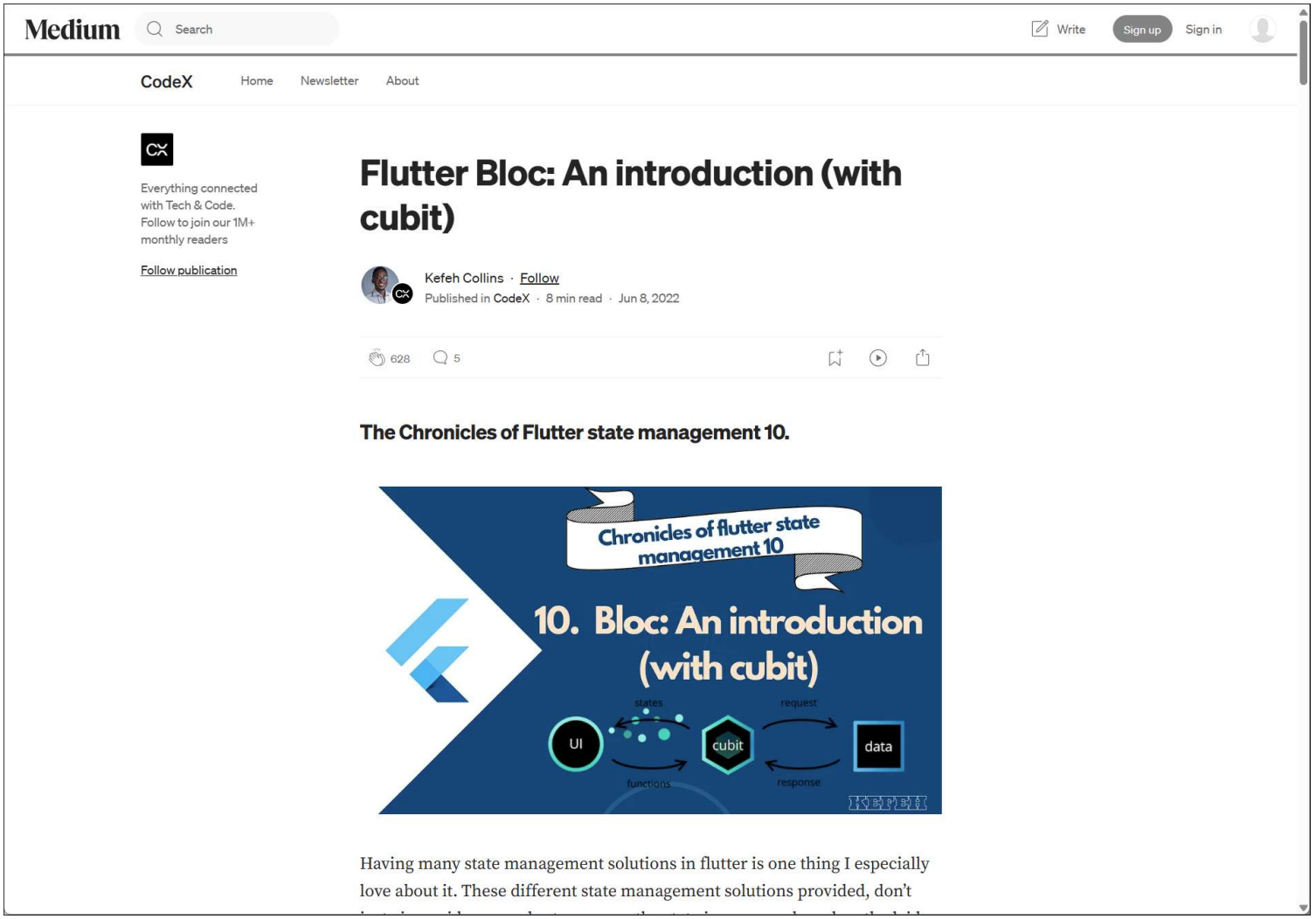
But: your mileage may vary! This is in no way mandatory. As long as Dart can find the imports, it is OK. Create a structure that makes sense to YOUR application.

More info on Bloc

- Tutorials:
  - For instance: https://bloclibrary.dev/tutorials/flutter-infinite-list/
  - And more! See list. Good for self studying



Article on Medium (might be behind paywall)



https://medium.com/codex/flutter-bloc-an-introduction-with-cubit-7eae1e740fd0

Cubit? Comparison with Bloc

Similarities:

- Both `Bloc` and `Cubit` are part of the `flutter_bloc` package.
- Both expose a stream of states and allow you to emit new ones.
- Both integrate seamlessly with `BlocBuilder`, `BlocListener`, etc.
- Both are used for state management in Flutter apps.

Differences:

Feature	Cubit	Bloc
Complexity	Simpler	More structured, handles complex flows
API Style	Method calls emit states	Event → transition → state
Boilerplate	Minimal	More (requires events and mapping logic)
Use Case	Straightforward state changes	Complex logic with many event types
Extensibility	Less (fewer lifecycle hooks)	More (e.g., `onTransition`, `onError`)

When to use which:

- **Use** `Cubit` for simple, linear state changes (like a counter, toggles, UI mode switching).
- **Use** `Bloc` when you have complex logic, multiple events per feature, or want full control over transitions and side effects.

Can they be used interchangeably?

- Short answer: no
  - `Cubit` is a simplified version of `Bloc`.
  - Every `Cubit` is a `BlocBase`, but not every `Bloc` is a `Cubit`.
  - You can start with `Cubit` and upgrade to `Bloc` later if needed.
- Beginnerst tip:
  - “Start with `Cubit`. Switch to `Bloc` if you feel constrained”
- But: Maritieme IT choose `Bloc`. So go with that.