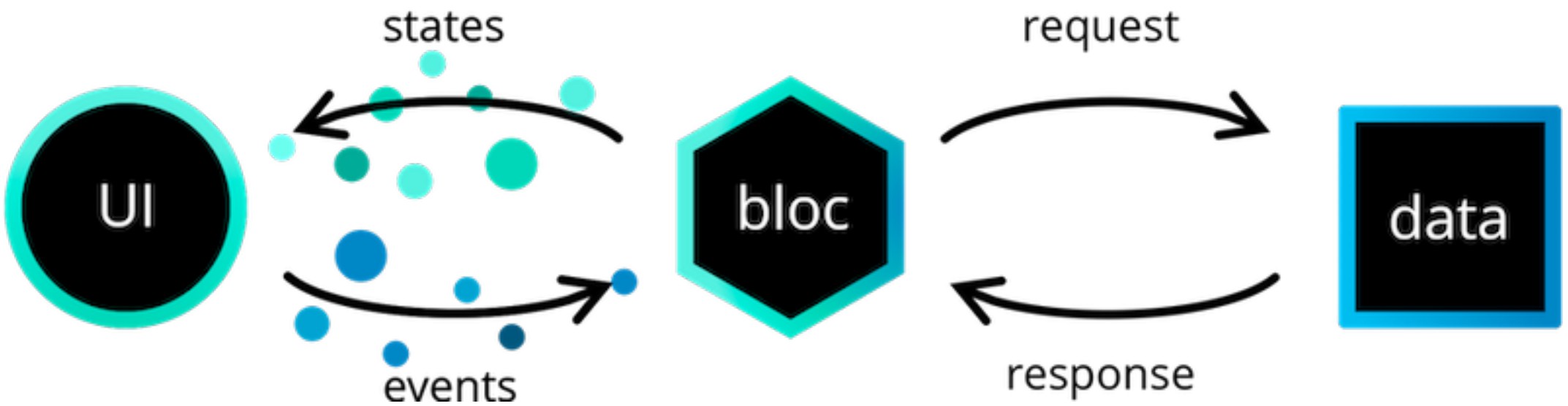
Resource Software Solution

# Lightweight Bloc with Cubit

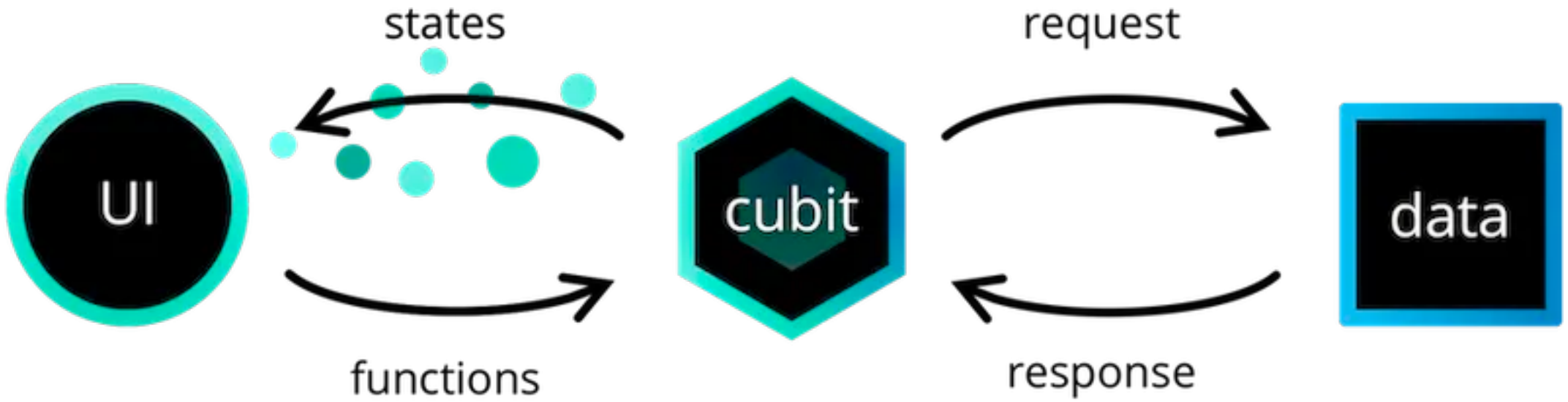
# Introduction

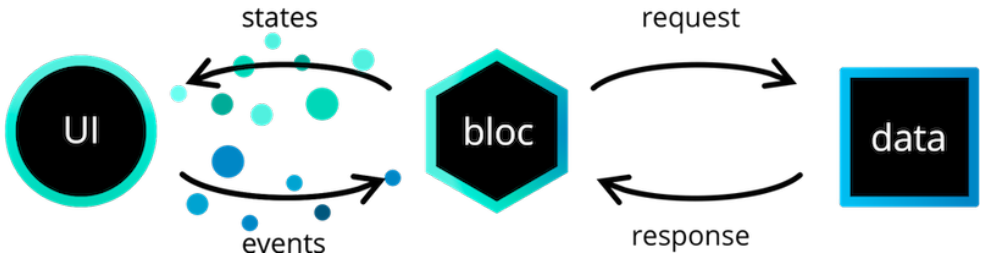
- BloC makes it easy to implement the Business Logic Component design pattern, which **separates presentation from business logic**.



# Introduction

- Cubit is a subset of the Bloc that **does not rely on events** and instead **uses methods to emit new states.**



# Introduction

```dart
abstract class CounterEvent {}

class Increment extends CounterEvent {}
class Decrement extends CounterEvent {}

class _CounterBloc extends Bloc<CounterEvent, int> {

  _CounterBloc() : super(0) {
    on<CounterEvent>(((event, emit) {
      if (event is Increment) {
        emit(state + 1);
      } else if (event is Decrement) {
        emit(state - 1);
      }
    }));
  }
}
```

**Bloc**

**VS**

```dart
class CounterCubit extends Cubit<int> {
  CounterCubit() : super(0);

  void increase() => emit(state + 1);
  void decrease() => emit(state - 1);
}
```

**Cubit**

# Cubit Example State Management

- We start by creating a cubit

Extending Cubit and specifying the type for our state

```dart
class CounterCubit extends Cubit<int> {
  CounterCubit() : super(0);

  void increase() => emit(state + 1);
  void decrease() => emit(state - 1);
}
```

Specifying the initial state

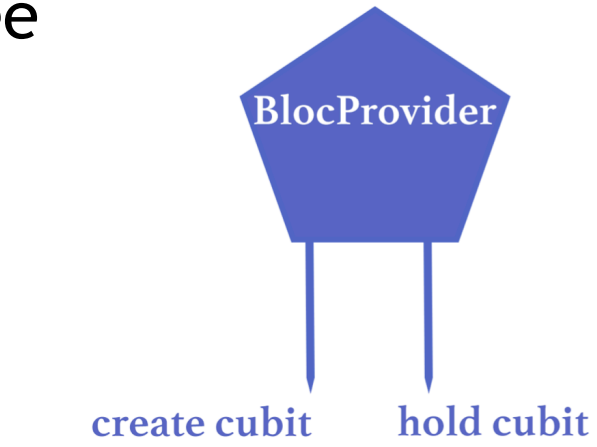
# Cubit Example State Management

- So putting creating and providing our Cubit in the MyApp Widget means every widget below it will have access to the Cubit.

```dart
class MyCounterApp extends StatelessWidget {
  const MyCounterApp({super.key});

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: BlocProvider(
        create: (_) => CounterCubit(),
        child: const _CounterPage(),
      ),  // BlocProvider
    );  // MaterialApp
  }
}
```

Using the BlocProvider as a dependency injector to create and provide to the child widget tree



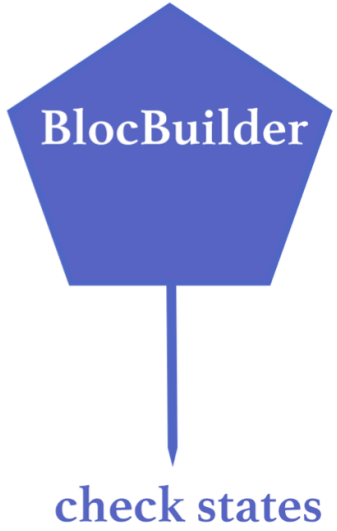

The create method, creates the cubit and makes it accessible in the context

# Cubit Example State Management

- After this, we proceed to use our cubit. We can use it either using the widget BlocBuilder as such

```
body: BlocBuilder<CounterCubit, int>(
  builder: (context, state) => Center(
    child: Text(
      '$state',
      style: Theme.of(context).textTheme.headline3,
    ),  // Text
  ),  // Center
),  // BlocBuilder
```

Using BlocBuilder to get access to our Cubit



# Cubit Example State Management

Resource Software Solution

- To call the method in Cubit

```dart
floatingActionButton: Column(
  mainAxisAlignment: MainAxisAlignment.end,
  crossAxisAlignment: CrossAxisAlignment.end,
  children: [
    FloatingActionButton(
      child: const Icon(Icons.add),
      onPressed: () => context.read<CounterCubit>().increase()),
    const SizedBox(height: 10,),
    FloatingActionButton(
      child: const Icon(Icons.remove),
      onPressed: () => context.read<CounterCubit>().decrease())
  ],
),  // Column
```

```dart
class CounterCubit extends Cubit<int> {
  CounterCubit() : super(0);

  void increase() => emit(state + 1);
  void decrease() => emit(state - 1);
}
```

# Summary - Bloc V7 vs Bloc V8

Resource Software Solution

BloC V7

```dart
enum CounterEvent { increment, decrement }

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

  @override
  Stream<int> mapEventToState(CounterEvent event) async* {
    switch (event) {
      case CounterEvent.decrement:
        yield state - 1;
        break;
      case CounterEvent.increment:
        yield state + 1;
        break;
    }
  }
}
```

BloC V8

```dart
class CounterIncrementPressed extends CounterEvent {}
class CounterDecrementPressed extends CounterEvent {}

class CounterBloc extends Bloc<CounterEvent, int> {
  /// {@macro counter_bloc}
  CounterBloc() : super(0) {
    on<CounterIncrementPressed>((event, emit) => emit(state + 1));
    on<CounterDecrementPressed>((event, emit) => emit(state - 1));
  }
}
```

# Summary - Bloc

- When we use BLoC, there's two important things: "Events" and "States".
- That means that when we send an "Event" we can receive one or more "States", and these "States" are sent in a **stream**.

```dart
// Here we map incoming events to states. In this case for 1 event we are sending
// two states: Loading and Profile.
@override
Stream<ProfileState> mapEventToState(
    ProfileEvent event, // incoming event
) async* { // async* gives you a stream
    if (event is CreateProfileEvent) {
        yield CreateProfileLoadingState(); // send loading state to the stream
        final Profile profile = await service.createProfile(); // do the API call
        yield ProfileState(profile); // send the new state with the profile data
    }
}
```

# Summary - Bloc

- If you send a **lot of events** for one BLoC could be **difficult to track** if you are not tracking them correctly.
- Sometimes we want to receive this “state” **synchronously**, but with BLoC this is **not possible**.

```dart
// Here we map incoming events to states. In this case for 1 event we are sending
// two states: Loading and Profile.
@override
Stream<ProfileState> mapEventToState(
    ProfileEvent event, // incoming event
) async* { // async* gives you a stream
    if (event is CreateProfileEvent) {
        yield CreateProfileLoadingState(); // send loading state to the stream
        final Profile profile = await service.createProfile(); // do the API call
        yield ProfileState(profile); // send the new state with the profile data
    }
}
```

# Summary - Cubit

- When we use cubit, we can **only send "states"** and to trigger these states we can do it by **calling a function** (like actions).
- We can decide if we want to send these "states" in a **synchronous** or **asynchronous** way.

```
// # Async
// create profile using cubit, because "service.createProfile" is a Future, we need to use await and async
// keep in mind that async is not the same that async*, async gives you a Future and async* gives you a Stream.
// This can be triggered with something like: contex.read<ProfileCubit>().createProfile();
void createProfile() async {
    emit(CreateProfileLoadingState()); // send loading state to the stream
    final Profile profile = await service.createProfile(); // do the API call operation and wait until the response
    emit(ProfileState(profile)); // send the new state with the profile data
}

// # Sync
// We can also do synchonous operations, like activate the profile when a button is pressed.
// This can be triggered with something like: contex.read<ProfileCubit>().setActive();
void setActive() => emit(ProfileActiveState());
```