Day 5:
State Management

State Management - Overview

State Management is how you handle and maintain the state of your application.

- State = data that changes over time (user input, counters, etc.)
- Good state management → clean, maintainable code
- Common libraries: `provider`, `flutter_bloc`, `Redux`, `MobX`

State Management - Approaches

Scenario: A counter app with three buttons for incrementing, decrementing, and resetting the counter.

Three main approaches to manage state in Flutter:

1. `setState` – Simple but limited
2. **Provider** – Recommended for most apps
3. **BLoC (Business Logic Component)** – Best for complex state

`setState` - Overview

`setState` is the built-in way to manage state in Flutter using StatefulWidgets.

- Easiest way to manage state
- Mixes UI and business logic
- Rebuilds entire subtree → performance issues

Example:

```
setState(() => _counter++);
```

Simple but Limited

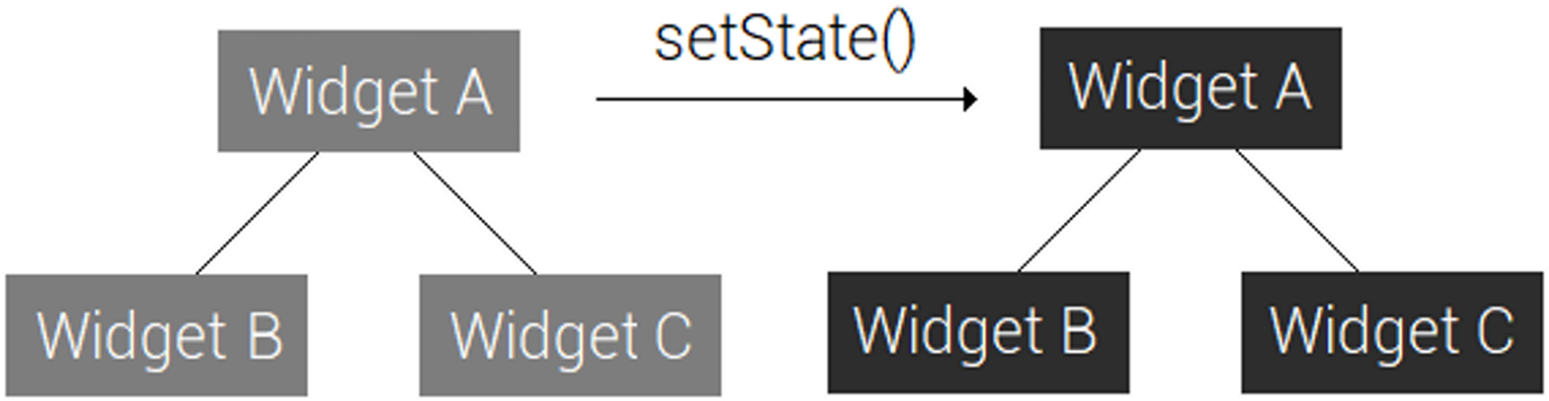
`setState` - Code Structure

```
class DemoPage extends StatefulWidget {
    @override
    _DemoPageState createState() => _DemoPageState();
}

class _DemoPageState extends State<DemoPage> {
    int _counter = 0;
    void _increment() => setState(() => _counter++);
}
```

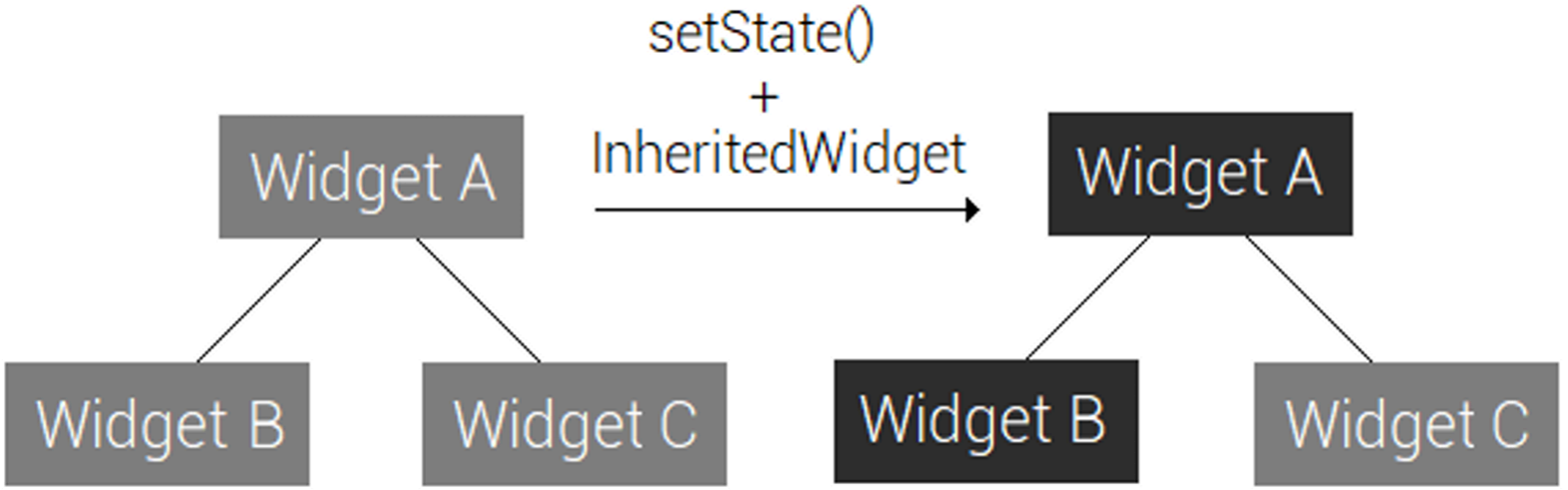
`setState` - Problems

- Rebuilds **all children**, even unrelated widgets
- Breaks **Single Responsibility Principle (SRP)**
- UI logic + business logic in one class
- Hard to maintain in large apps



`setState` + `InheritedWidget`

- Optimizes rebuilds with `InheritedWidget`
- But → boilerplate code ↑
- Not recommended for manual use
- Use libraries like `provider` instead

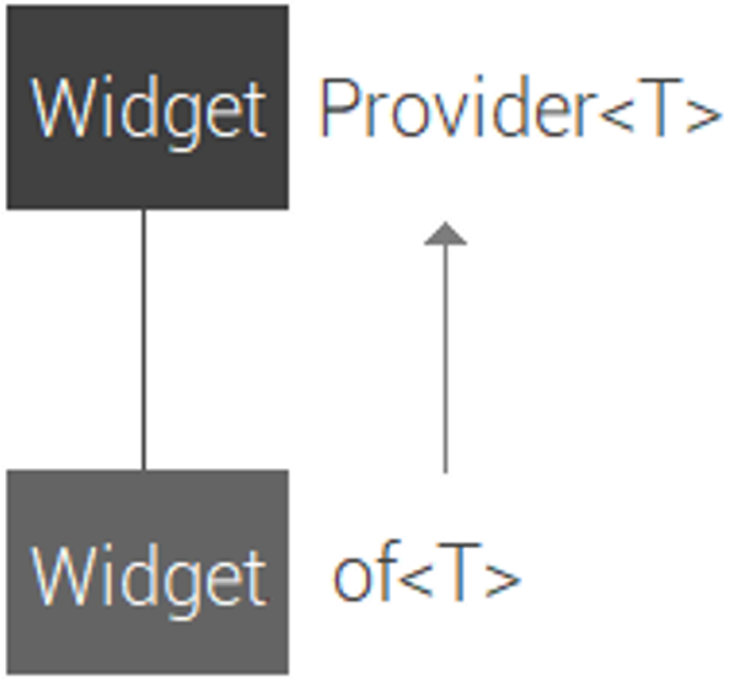


Provider - Overview

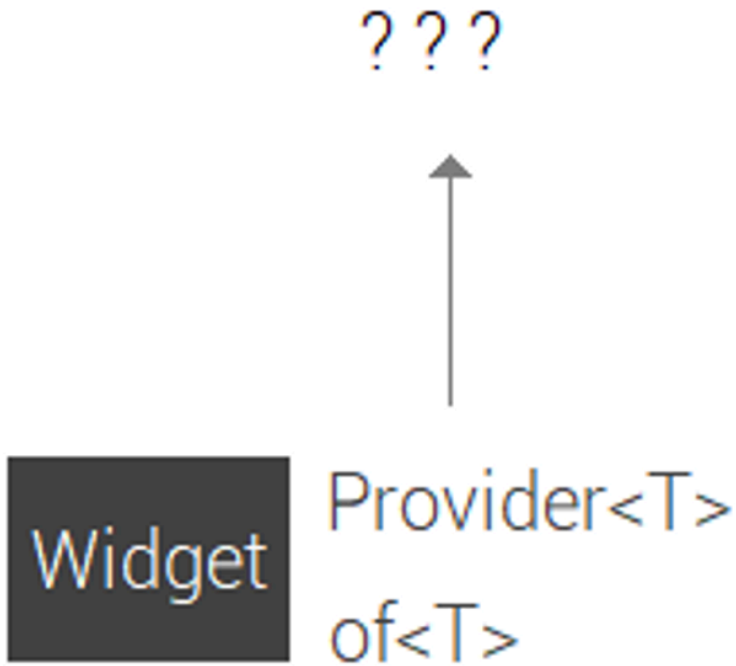
Provider is a popular state management library in Flutter, created by Rémi Rousselet.

Advantages:

- Uses `ChangeNotifier` for state changes
- Separates UI and business logic
- Less boilerplate than `setState` + `InheritedWidget`



1.



2.

Provider - Model Example

```
class CounterModel with ChangeNotifier {
    int _counter = 0;
    void increment() {
        _counter++;
        notifyListeners();
    }
    int get currentCount => _counter;
}
```

Provider - UI Example

```
ChangeNotifierProvider(
    create: (_) => CounterModel(),
    child: DemoPage(),
);

// In widget:
final counter = Provider.of<CounterModel>(context);
```

Provider - Advantages

- `Consumer<T>`: Rebuilds only needed widgets
- `Selector<T, S>`: Fine-grained rebuild control
- Optimizes performance
- Reduces unnecessary rebuilds

BLoC - Overview

BLoC (Business Logic Component) is a design pattern for managing state in Flutter applications.

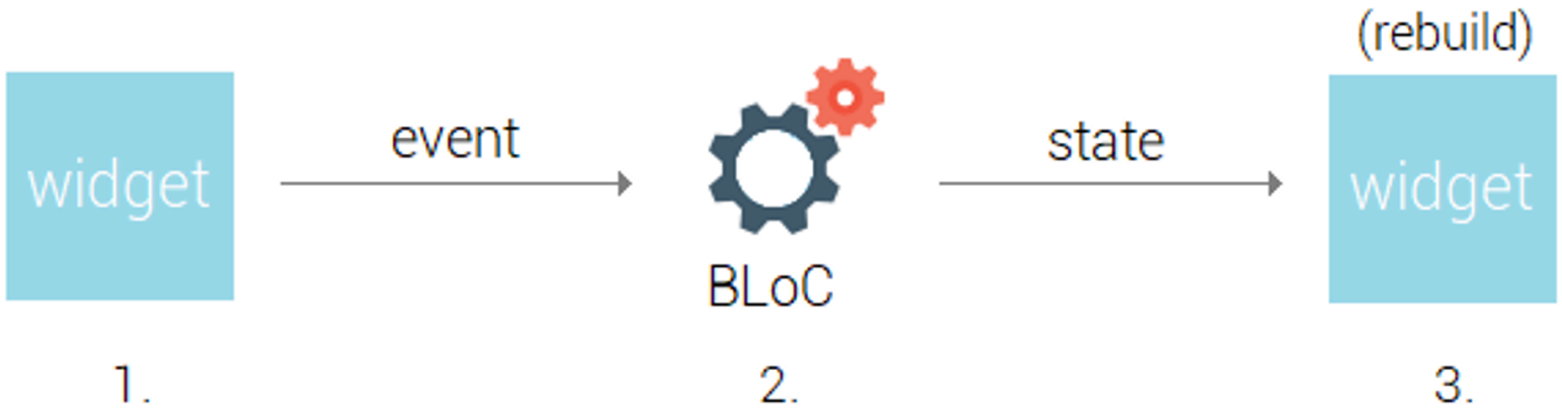
Advantages:

- Uses streams for state changes
- Events → Bloc → States
- Clear separation of concerns

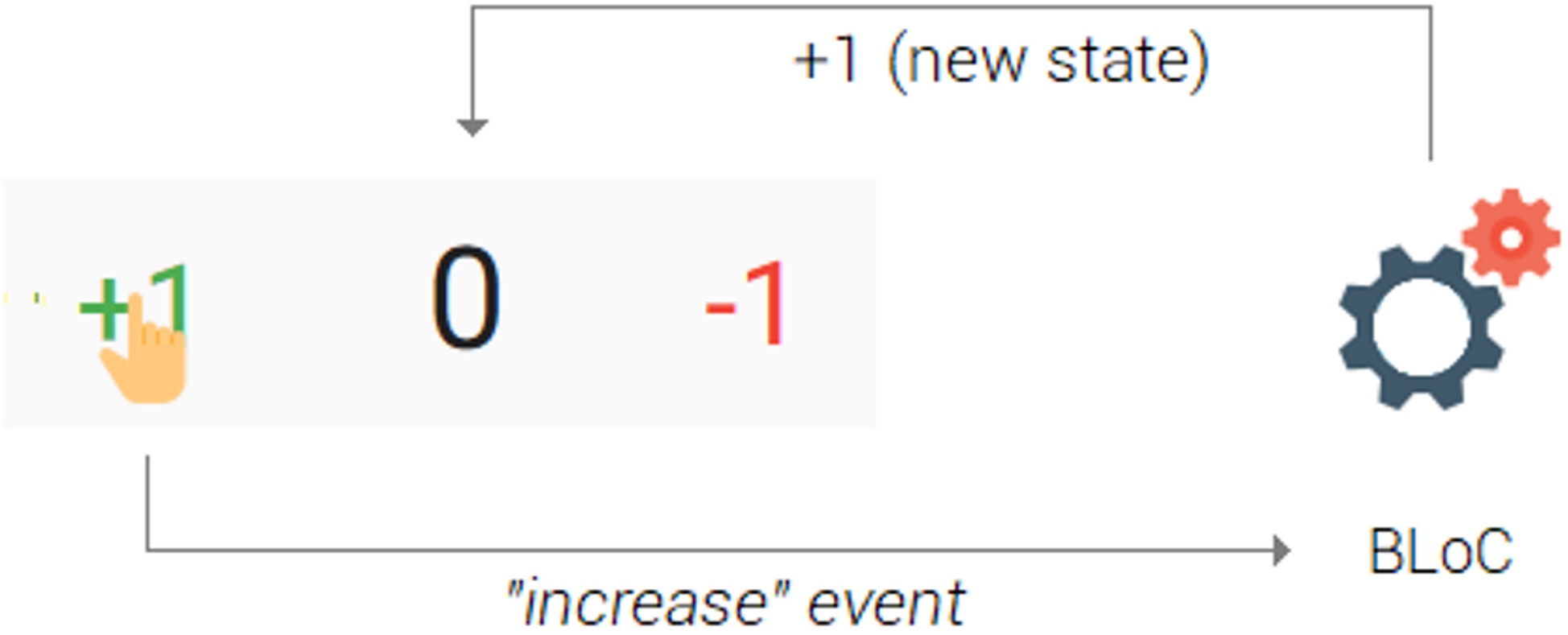
BLoC - Flow

```
Widget → Event → Bloc → State → Widget (rebuild)
```

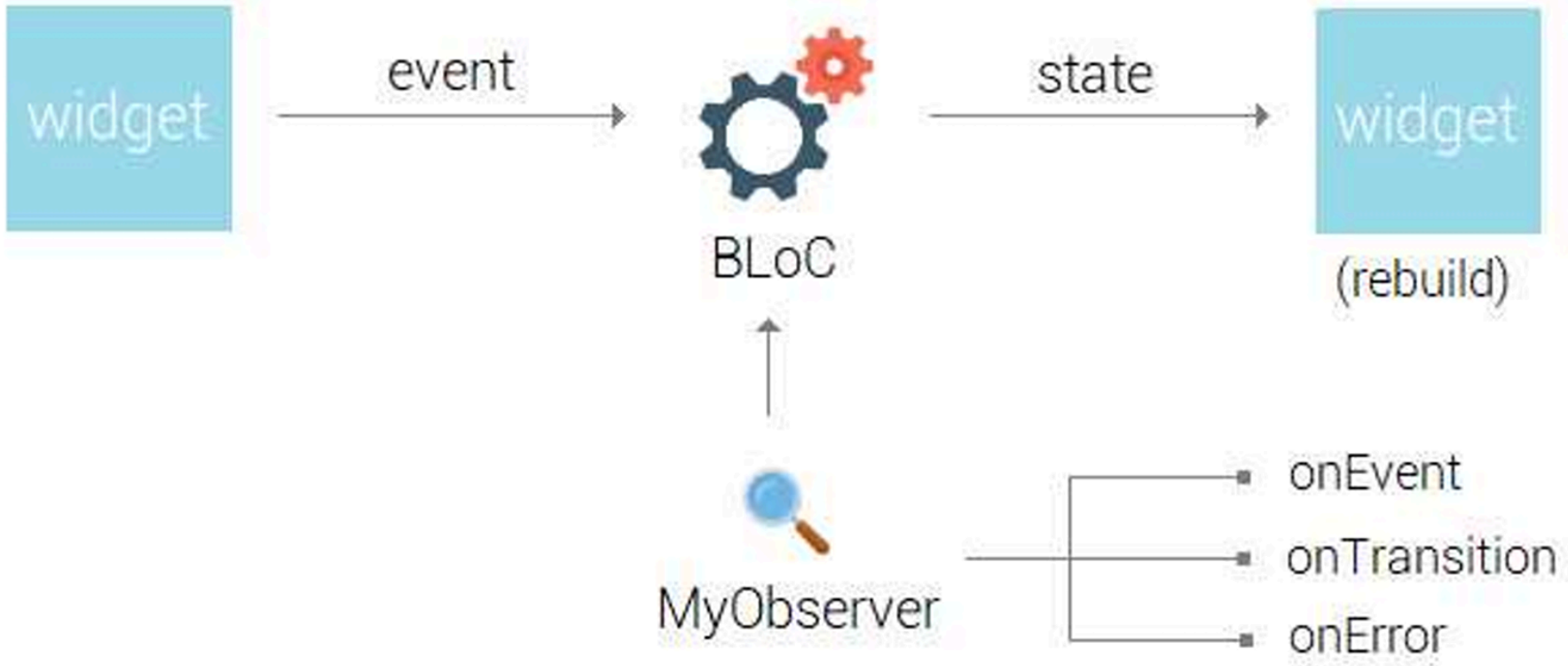
- Widget sends events
- Bloc processes → emits new state
- Widget rebuilds with new state



BLoC – Event



BLoC – Observer



BLoC - Model Example

```
enum CounterEvent { increment, decrement }

class CounterBloc extends Bloc<CounterEvent, int> {
    CounterBloc() : super(0);
    @override
    Stream<int> mapEventToState(CounterEvent event) async* {
        switch (event) {
            case CounterEvent.increment: yield state + 1;
            case CounterEvent.decrement: yield state - 1;
        }
    }
}
```

BLoC - UI Example

```
BlocProvider(
    create: (_) => CounterBloc(),
    child: DemoPage(),
);

// In widget:
final bloc = context.bloc<CounterBloc>();
bloc.add(CounterEvent.increment);
```

BLoC - Classes (Advanced)

BLoC can also use classes for events and states for more complex scenarios.

```
abstract class CounterEvent {}
class IncrementEvent extends CounterEvent {}
class DecrementEvent extends CounterEvent {}
abstract class CounterState {}
class CounterValue extends CounterState {
    final int value;
    CounterValue(this.value);
}
```

BLoC - Builder & Listener

BlocListener and **BlocConsumer** are widgets that react to state changes.

- `BlocListener`: Reacts to state changes (e.g., show snackbar)
- `BlocConsumer`: Combines `BlocBuilder` + `BlocListener`
- Reduces nesting

BLoC - HydratedBloc

HydratedBloc is an extension of BLoC that automatically persists and restores Bloc states.

- Auto-saves/restores state
- Uses `fromJson` / `toJson`
- Works after app restarts
- Great for themes, preferences

BLoC - ReplayBloc

ReplayBloc is an extension of BLoC that adds undo and redo functionality.

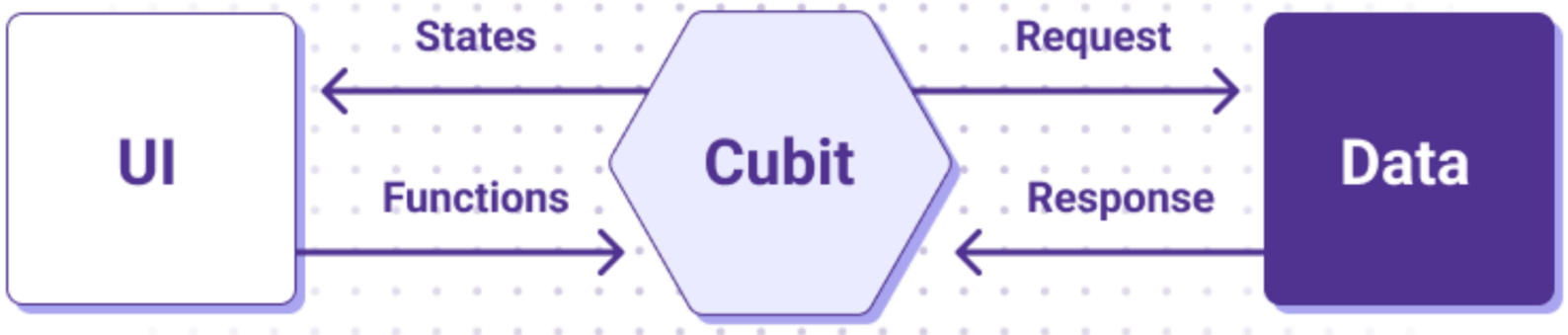
- Extends `ReplayBloc`
- Uses `undo()` / `redo()` methods
- Works with `HydratedBloc` via mixins

Cubit – Lightweight BLoC

Cubit is a simpler version of BLoC, also created by Rémi Rousselet, to manage state in Flutter applications.

Advantages:

- Simpler than BLoC
- No events, just methods
- Uses `emit()` for new states
- Good for simple state



Cubit Example

```
class CounterCubit extends Cubit<int> {
    CounterCubit() : super(0);
    void increment() => emit(state + 1);
}
```

Cubit vs BLoC

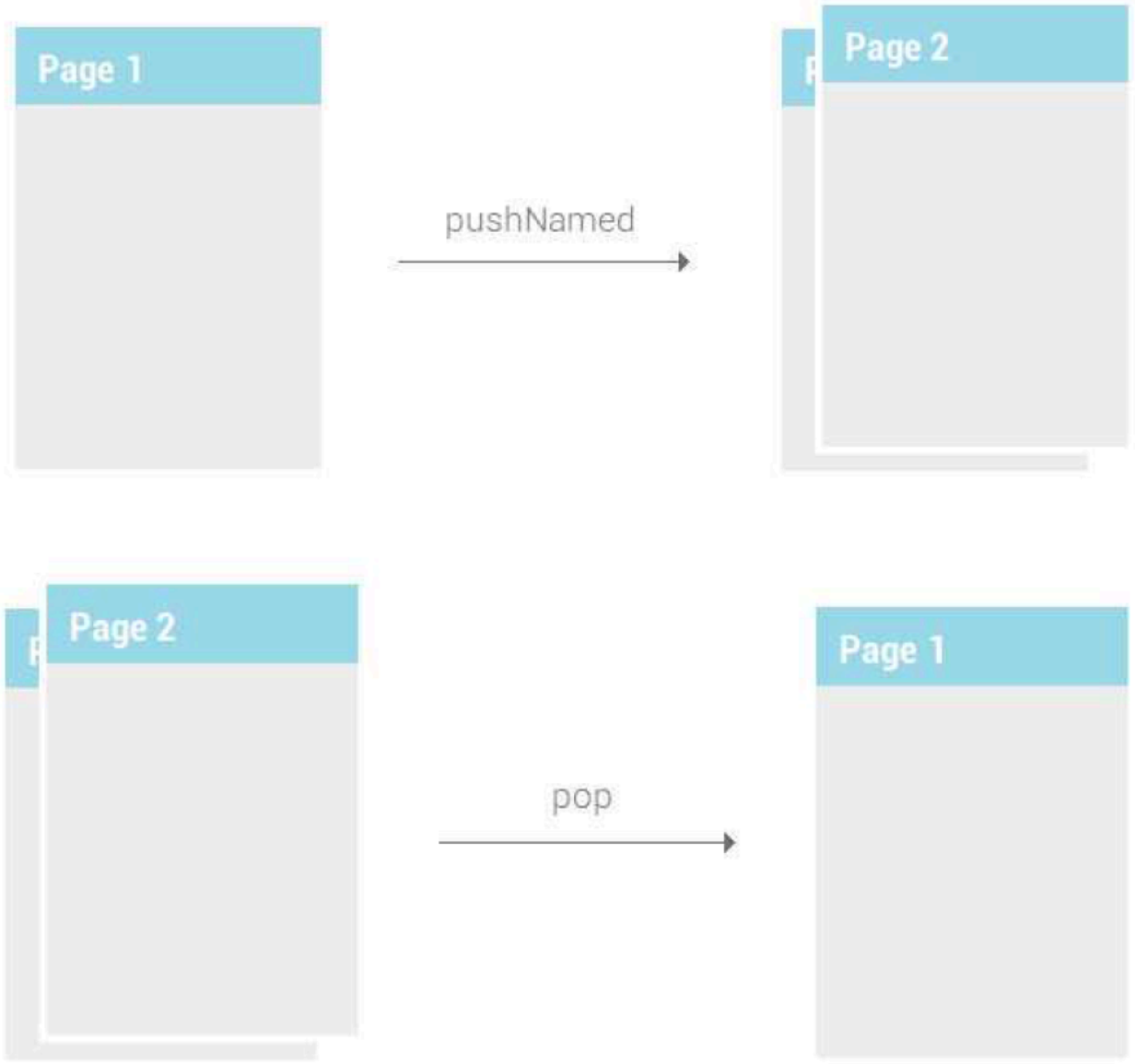
- **Cubit**: Simple, less code, good for simple state
- **BLoC**: Powerful, events + states, good for complex logic
- Both work with the same widgets (`BlocBuilder` , `BlocListener`)

Routes and Navigation in Flutter

Routes - Overview

Routes in Flutter are used to navigate between different screens or pages within an application.

- Routes = Screens = Pages
- Equivalent to Activities (Android) / ViewControllers (iOS)
- Built with `StatelessWidget` or `StatefulWidget` + `Scaffold`
- Organized in a clear folder structure


Page 1
pushNamed
Page 2
Page 2
pop
Page 1

Routes - Types

There are two main types of routing in Flutter:

1. **Named Routes**: Define routes with string names in a centralized place.
2. **Anonymous Routes**: Define routes directly in the `Navigator` calls.

Routes – Named Route Generator

```
class RouteGenerator {
    static const String homePage = '/';
    static const String randomPage = '/random';

    static Route<dynamic> generateRoute(RouteSettings settings) {
        switch (settings.name) {
            case homePage:
                return MaterialPageRoute(builder: (_) => const HomePage());
            case randomPage:
                return MaterialPageRoute(builder: (_) => const RandomPage());
            default:
                throw FormatException("Route not found");
        }
    }
}
```

Routes – Named Route Usage

```
class HomePage extends StatelessWidget {
    @override
    Widget build(BuildContext context) {
        return Scaffold(
            appBar: AppBar(title: Text("Home Page")),
            body: Center(
                child: ElevatedButton(
                    onPressed: () {
                        Navigator.of(context).pushNamed(RouteGenerator.randomPage);
                    },
                    child: Text("Go to Random Page"),
                ),
            ),
        );
    }
}
```

Routes - Exposure to the URL

```
class AnyPage extends StatelessWidget {
    @override
    Widget build(BuildContext context) {
        return Scaffold(
            route: RouteGenerator.randomPage,
        );
    }
}
```

Navigator - Overview

Navigator is a widget that manages a stack of routes.

- Push new routes onto the stack
- Pop routes off the stack
- Manages navigation history
- Uses named routes for easier navigation

Navigator - Basic Stack Model

- Use `Navigator.pushNamed()` to go to a new screen
- Use `Navigator.pop()` to go back

Example:

```
Navigator.of(context).pushNamed(RouteGenerator.randomPage);
```

Navigator - Passing Data Between Pages

Use Sparingly [Not Recommended]

```
// Sending data to another page
Navigator.of(context).push(
        MaterialPageRoute(
                builder: (_) => NewPage(input),
        ),
);

// Receiving data back from a page
Navigator.of(context).push(
        MaterialPageRoute(builder: (_) => NewPage()),
).then((output) {
        String result = output ?? '';
});
```

Use Provider [Recommended]

Step 1 – Create a model:

```
class Todo {
    final String title, description;
    const Todo(this.title, this.description);
}
```

Step 2 – Create a cache class:

```
class TodoCache with ChangeNotifier {
    final List<Todo> _todos = [...];
    int _selectedIndex = -1;
    // methods...
}
```

Navigator 2.0 – What's New?

Key Changes in Navigator 2.0:

- Declarative routing via the new `Router` widget
- `RouterDelegate` + `RouteInformationParser` for custom navigation
- Better deep linking & web URL support
- `pop()` behavior changes – no longer returns values directly

Migration Tips:

- Use `PopScope` instead of `WillPopScope`
- Consider packages like `go_router` for easier adoption

```
MaterialApp.router(
        routerConfig: GoRouter(...),
);
```

Navigation Without BuildContext

Using GlobalKey:

```
class RouteGenerator {
    static final key = GlobalKey<NavigatorState>();
}

MaterialApp(
    navigatorKey: RouteGenerator.key,
);

// Navigate without context
RouteGenerator.key.currentState?.pushNamed('/route');
```

Alternative Routing Packages

1. **Flutter Modular** – Modular architecture with dependency injection
2. **Fluro** – Flexible routing with transitions
3. **Sailor** – Simple, context-free navigation

END