State Management

in Flutter

Learning Outcome

After completing this lesson, you should be able to

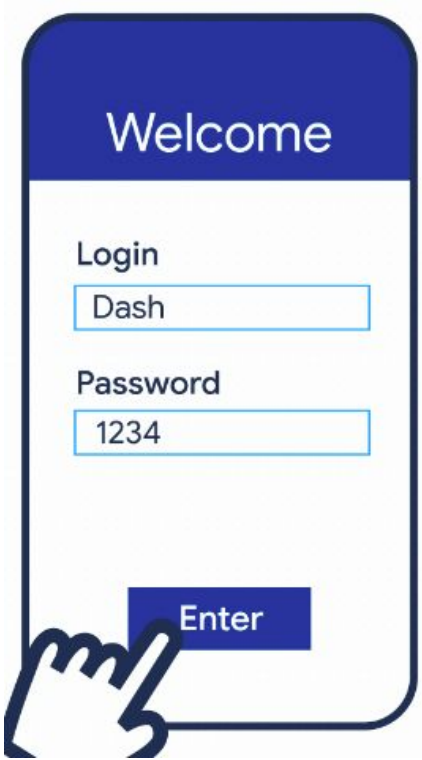
- Use other and advanced stage management solutions
- Use provider package
- Explain the architecture of Bloc state management solution
- Explain the core concepts of Bloc state management solution
- Use flutter_bloc package

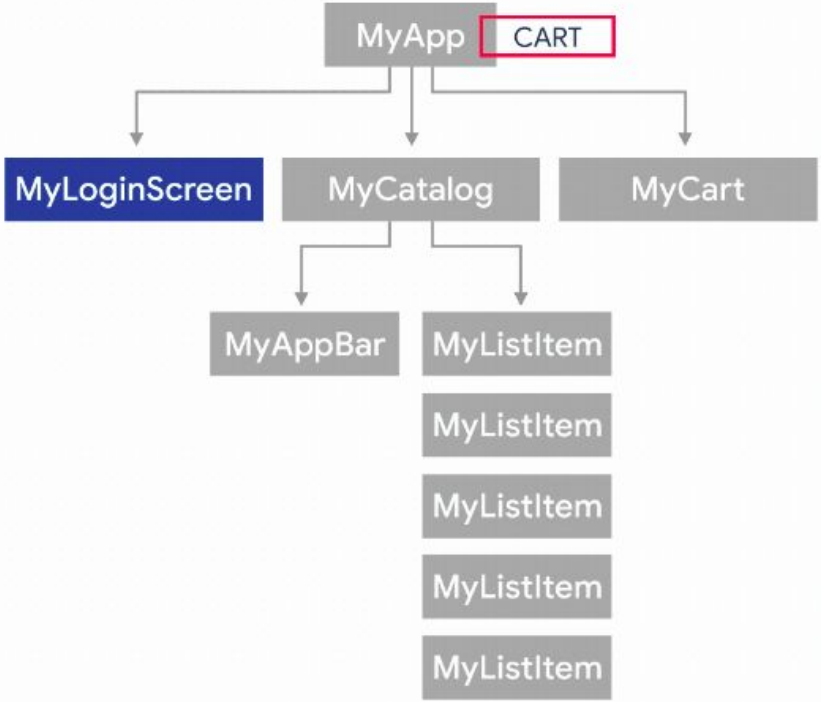
Flutter is Declarative

Flutter builds its user interface to reflect the current state of your app

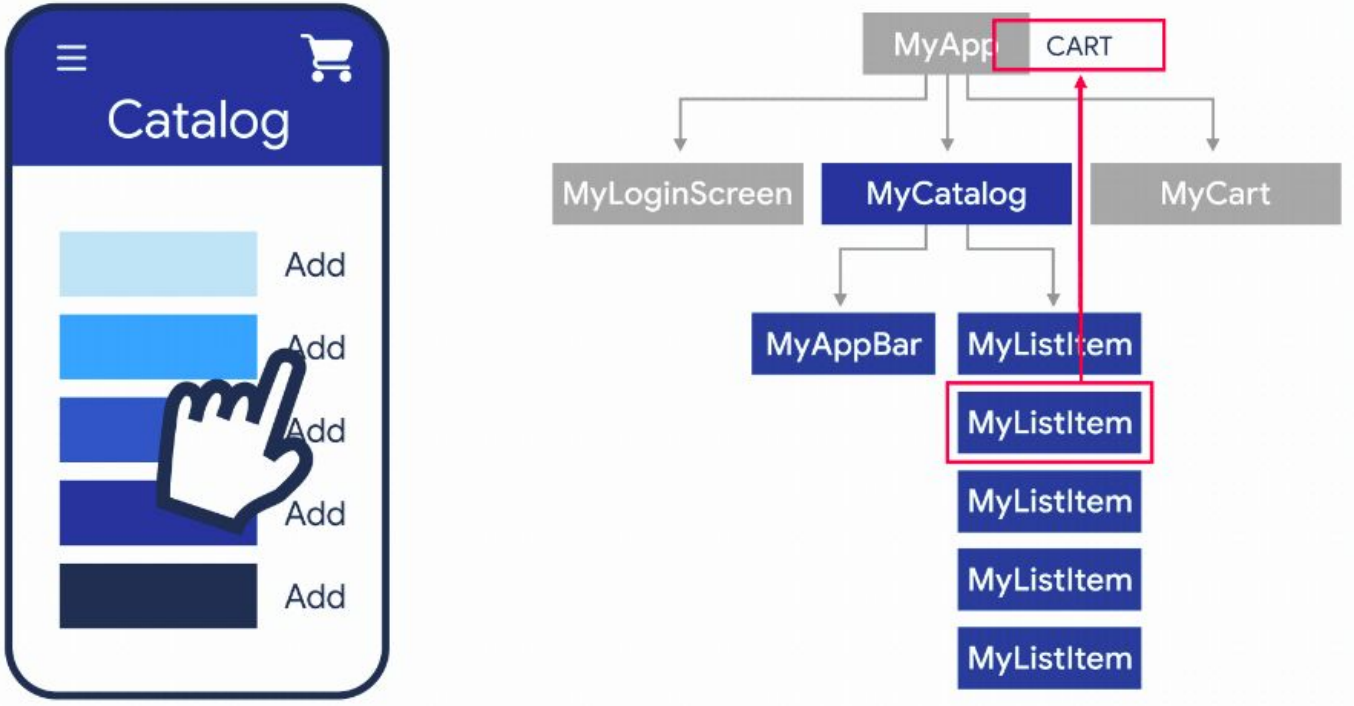
When the state of your app changes, you change the state, and that triggers a redraw of the user interface

What is State?

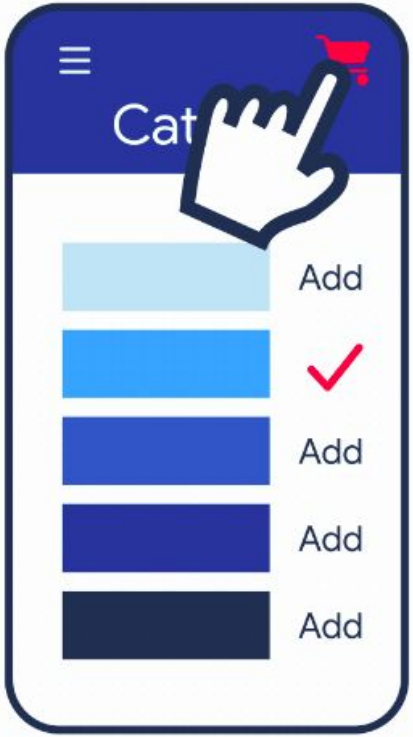


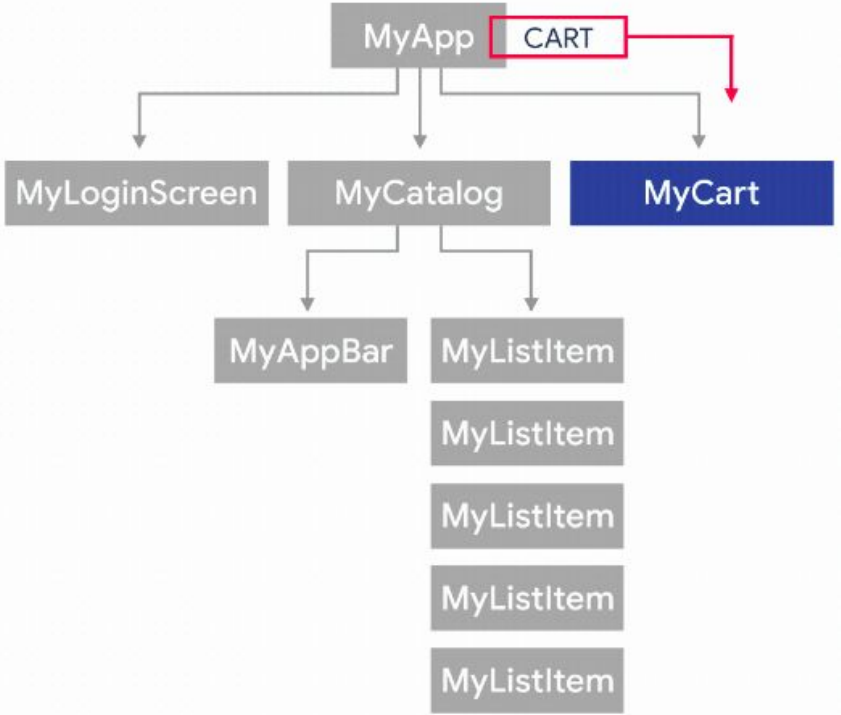


What is State?

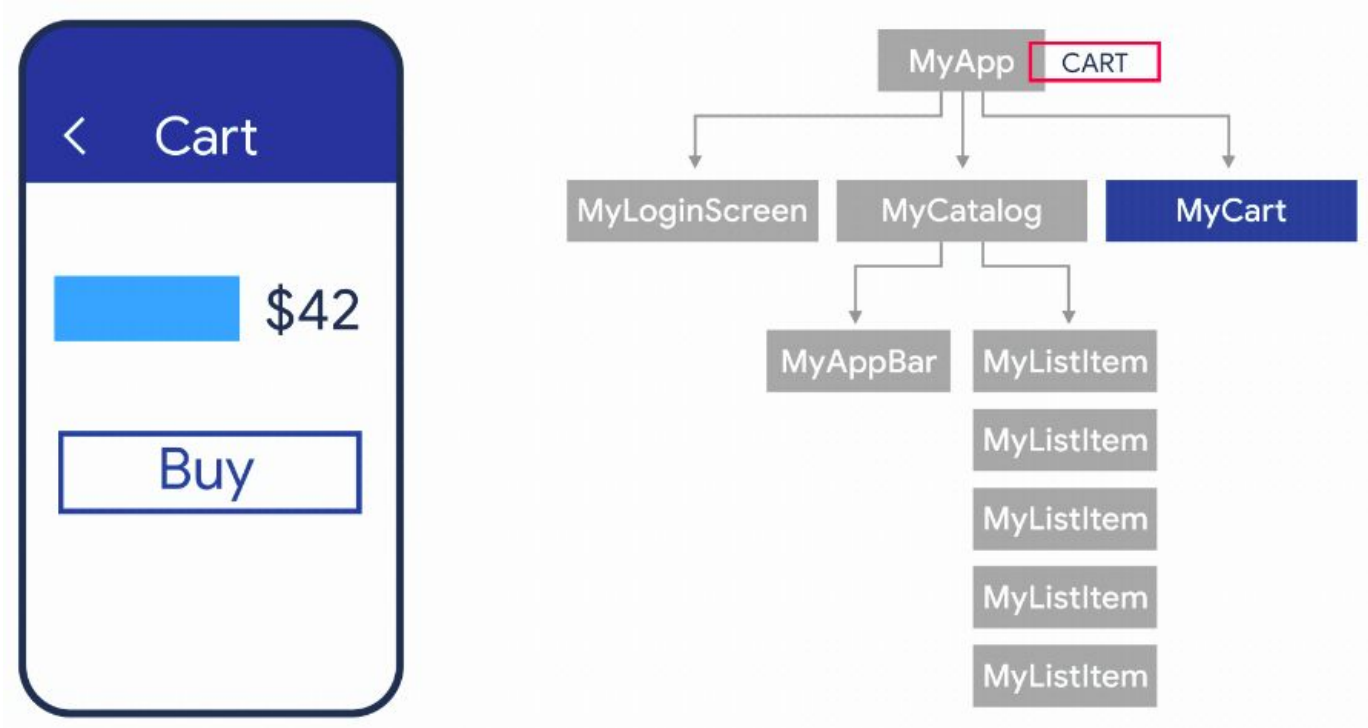


What is State?





What is State?



What is State?

State is whatever data you need in order to rebuild your UI at any moment in time

Some of the states are handled by the Flutter framework and some of them are managed by you, the programmer

The state that you 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

Below are a few examples

- current page in a `PageView`
- current progress of a complex animation
- current selected tab in a `BottomNavigationBar`

Ephemeral state can be implemented using `State` and `setState()`, and is often local to a single widget

Ephemeral State

The example code shown in the next slide demonstrates how the currently selected item in a bottom navigation bar is held in the `_index` field of the `_MyHomepageState` class

No other part of the app needs to access `_index`

- The variable only changes inside the `MyHomepage` widget
- Here, using `setState()` and a field inside the `StatefulWidget`'s `State` class is completely natural

```dart
class MyHomepage extends StatefulWidget {
  @override
  _MyHomepageState 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

Application State is a state

that is not ephemeral,

that you want to share across many parts of your app, and

that you want to keep between user sessions

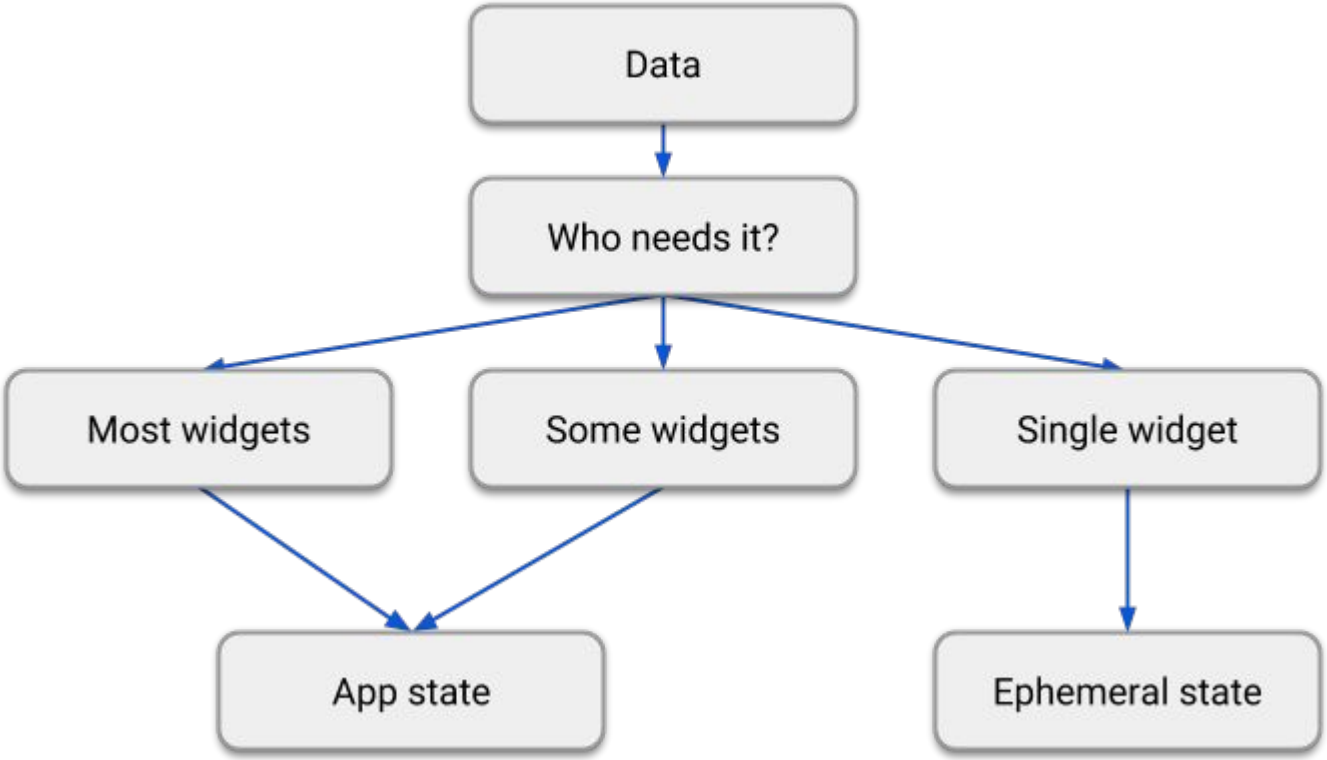
sometimes also called shared state

App State

Examples of application state

- User preferences
- Login info
- Notifications in a social networking app
- The shopping cart in an e-commerce app
- Read/unread state of articles in a news app

Ephemeral Vs App State



App State Example

Consider a simple part of shopping app

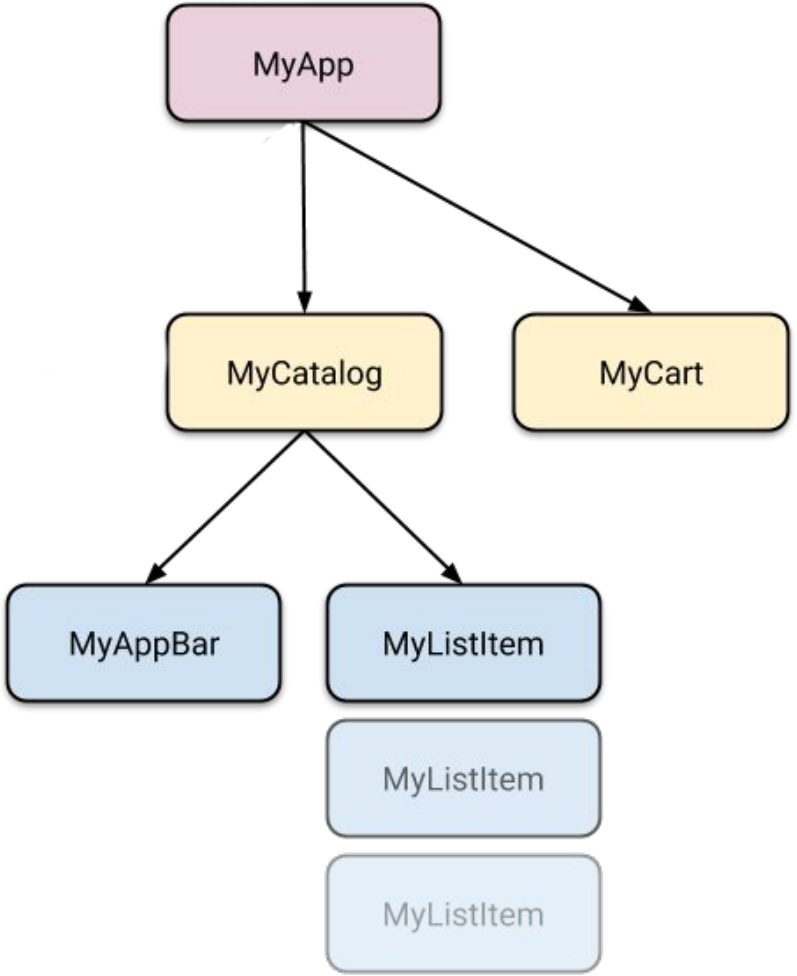
- The app has two separate screens: a **catalog**, and a **cart** (represented by the `MyCatalog`, and `MyCart` widgets, respectively)
- The **catalog** screen includes a custom app bar (`MyAppBar`) and a scrolling view of many list items (`MyListItems`)

App State Example

Here's the shopping app visualized as a **widget tree**

We have at least 5 subclasses of Widget

Many of them need access to state that "belongs" elsewhere

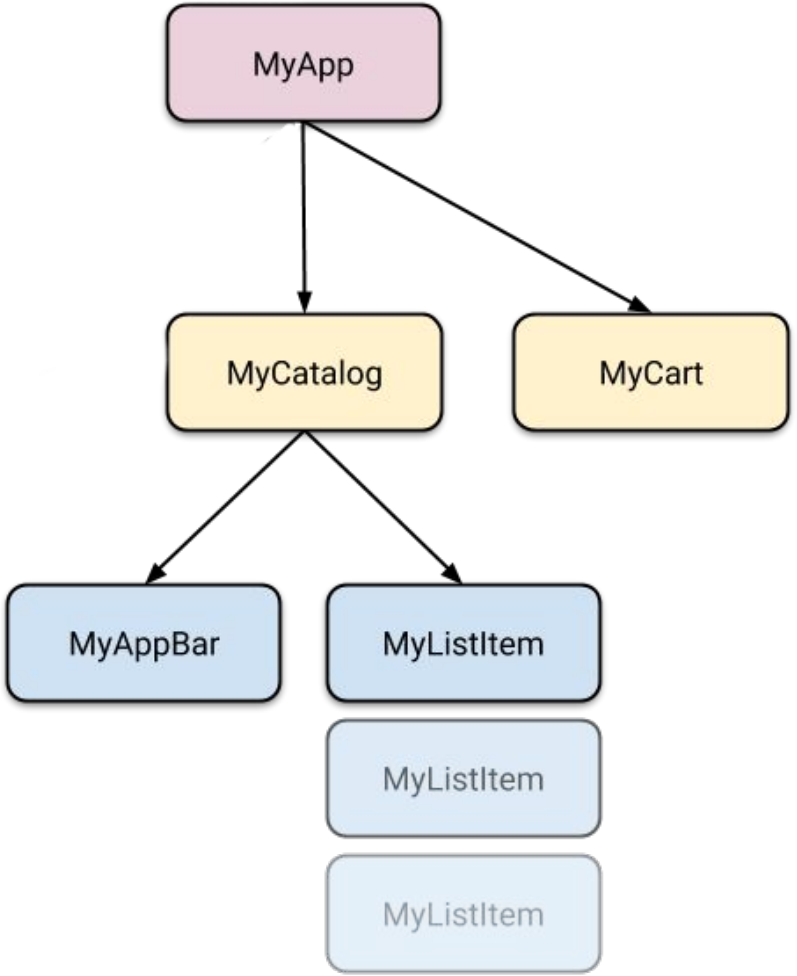


App State Example

Many of them need access to state that “belongs” elsewhere

> Each `MyListItem` needs to be able to add itself to the **cart**

Where should we put the current state of the cart?



Lifting state up

In Flutter, it makes sense to keep the state above the widgets that use it

In Flutter, you construct a new widget every time its contents change

    You normally use a constructor to do it

Because you can only construct new widgets in the build methods of their parents, if you want to change contents of a widget, **the content needs to live in parent widget or above in the widget tree**

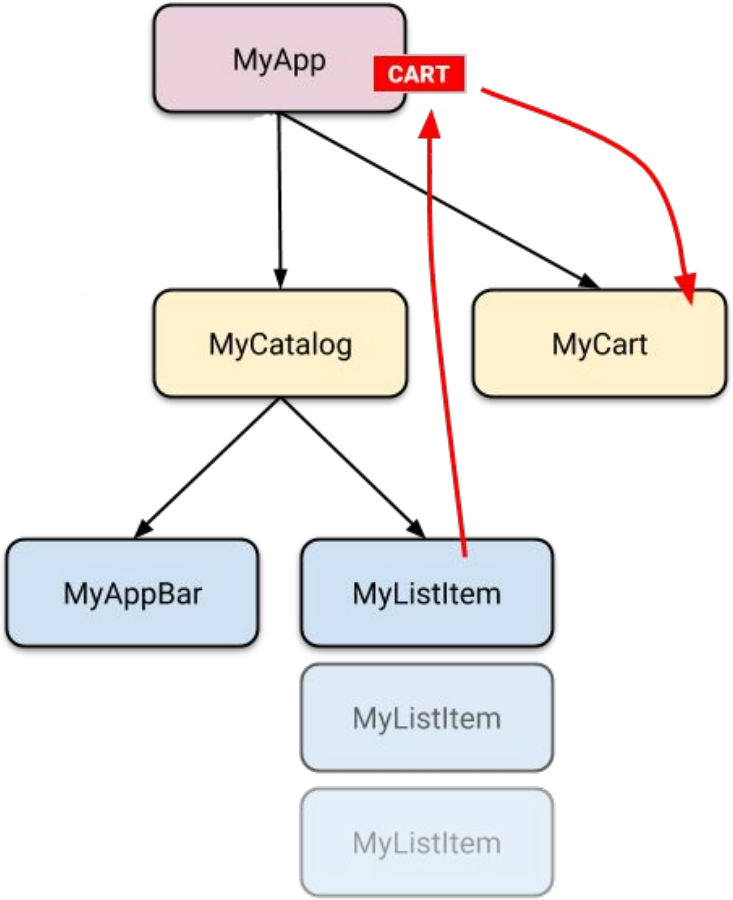
Lifting state up

In our example, contents needs to live in `MyApp`

Whenever it changes, it rebuilds **`MyCart`** from above

This is what we mean when we say that widgets are **immutable**

They don't change—they get replaced



State Management Solutions

`Provider`

`setState`

The low-level approach to use for widget-specific, ephemeral state

`InheritedWidget` & **`InheritedModel`**

The low-level approach used to communicate between ancestors and children in the widget tree

This is what provider and many other approaches use under the hood

State Management Solutions

Redux

A state container approach familiar to many web developers

Fish-Redux

Fish Redux is an assembled flutter application framework based on Redux state management

It is suitable for building medium and large applications

BLoC / Rx

A family of stream/observable based patterns

State Management Solutions

`GetIt`

A service locator based state management approach that doesn't need a **`BuildContext`**

`MobX`

A popular library based on observables and reactions

`GetX`

A simplified reactive state management solution

State Management Solutions

`Flutter Commands`

- Reactive state management that uses the Command Pattern and is based on **`ValueNotifiers`**
- Best in combination with **`GetIt`**, but can be used with Provider or other locators too

`Riverpod`

- An approach similar to **`Provider`** that is compile-safe and testable
- It doesn’t have a dependency on the Flutter SDK

State Management Solutions

`Binder`

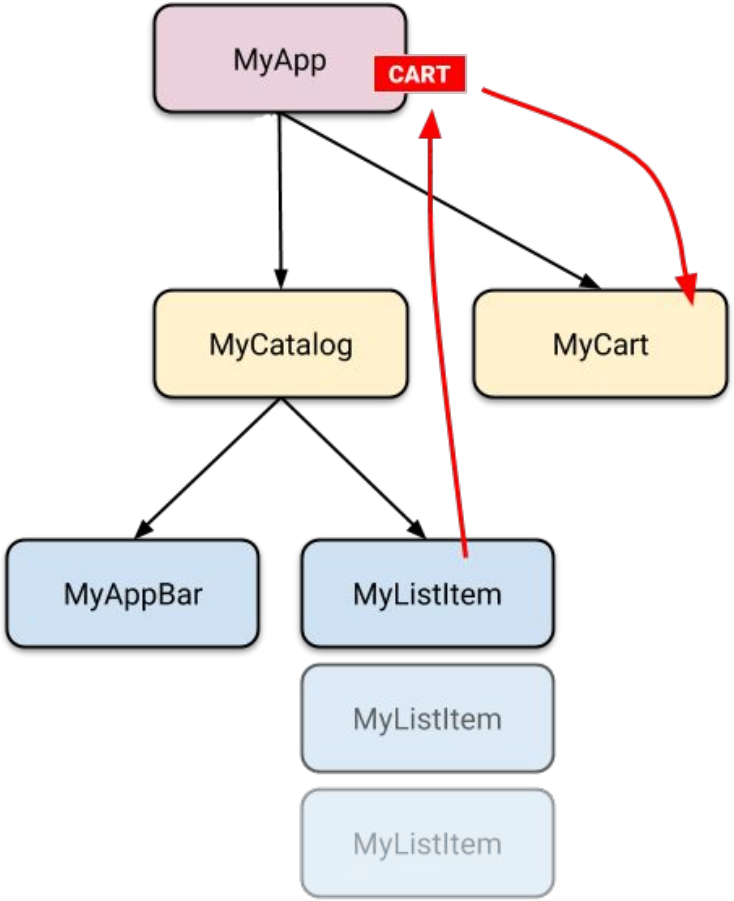
- A state management package that uses **`InheritedWidget`** at its core
- Inspired in part by recoil
- This package promotes the separation of concerns

Accessing the state

Flutter has mechanisms for widgets to provide data and services to their descendants

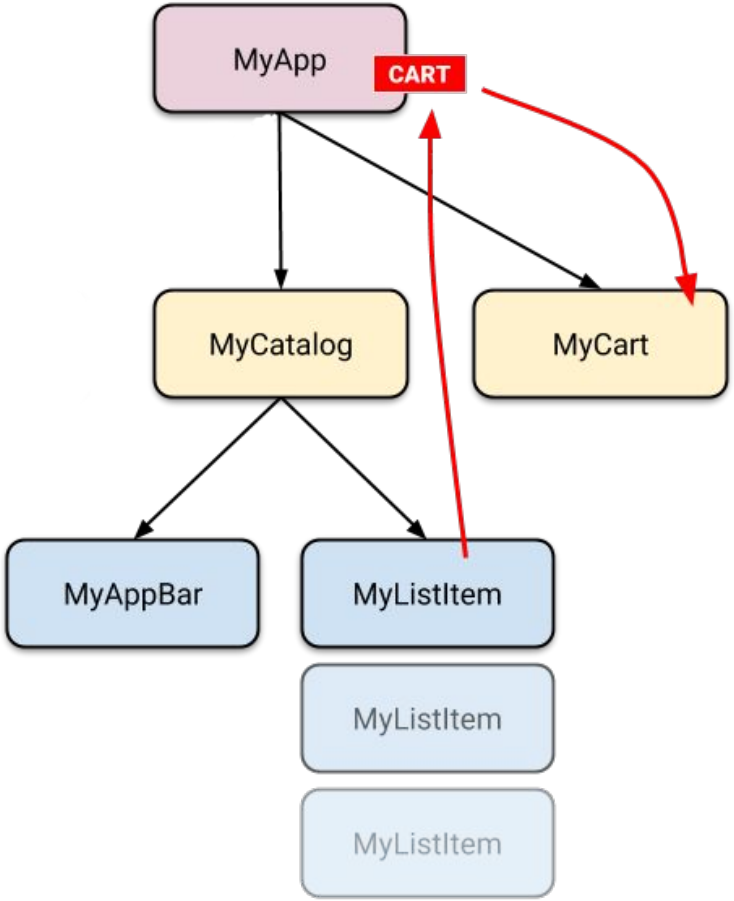
`InheritedWidget`, `InheritedNotifier`, `InheritedModel` are examples of widgets that can provide data or service to their descendants

> however they are low-level implementations



Provider

`Provider` is an alternative high-level package that uses the low-level widgets to achieve the same purpose



Provider

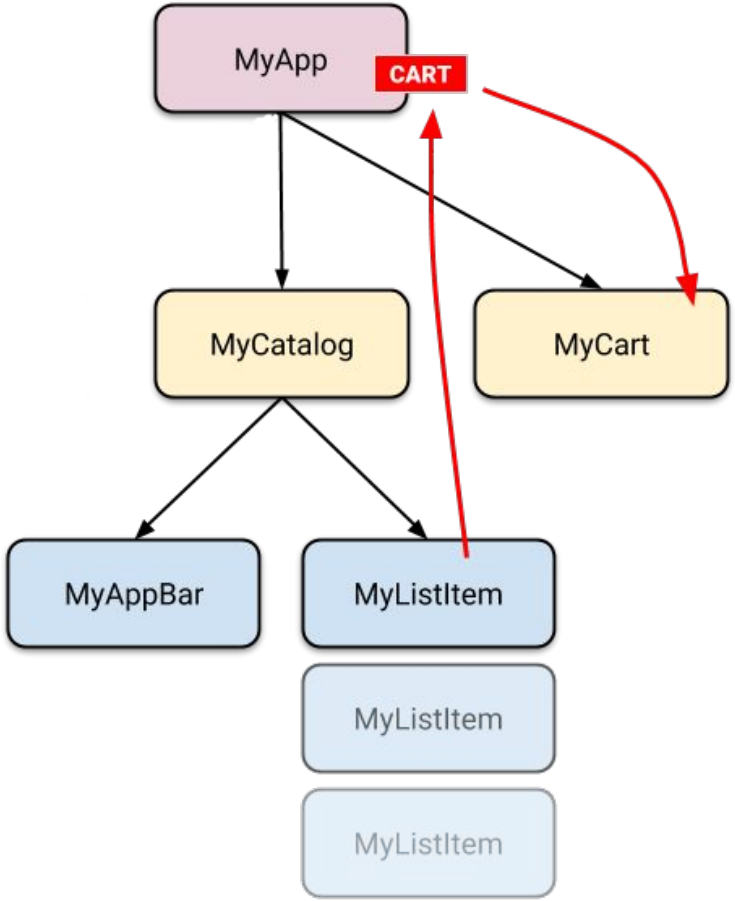
You need to add the dependency for provider in your `pubspec.yaml` file

```
# ...

dependencies:
  flutter:
    sdk: flutter

  provider: ^4.3.3

dev_dependencies:
  # ...
```



Provider

Basic concepts for understanding `Provider`

- `ChangeNotifier`
- `ChangeNotifierProvider`
- `Consumer`

Provider: `ChangeNotifier`

`ChangeNotifier` is a class which provides change notification to its listeners

I.e you can subscribe to its changes

In `provider`, `ChangeNotifier` is one way to encapsulate your application state

You need to call `notifyListeners()` after the state is mutated inside your `ChangeNotifier` class

Provider: ChangeNotifierProvider

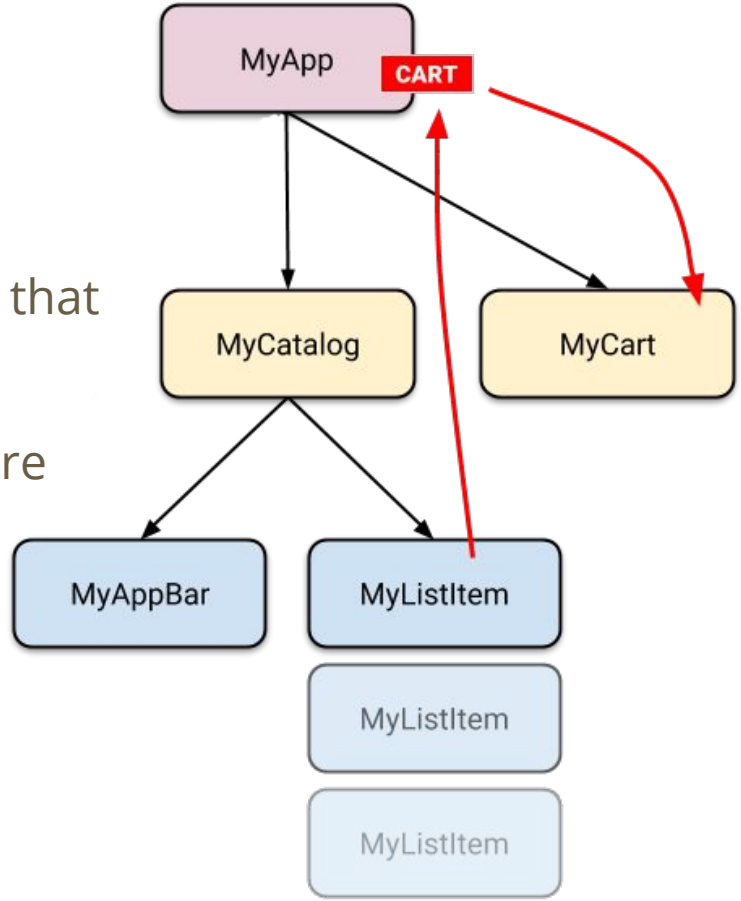
`ChangeNotifierProvider` is the widget that provides an instance of a `ChangeNotifier` to its descendants

It comes from the `provider` package

Provider: ChangeNotifierProvider

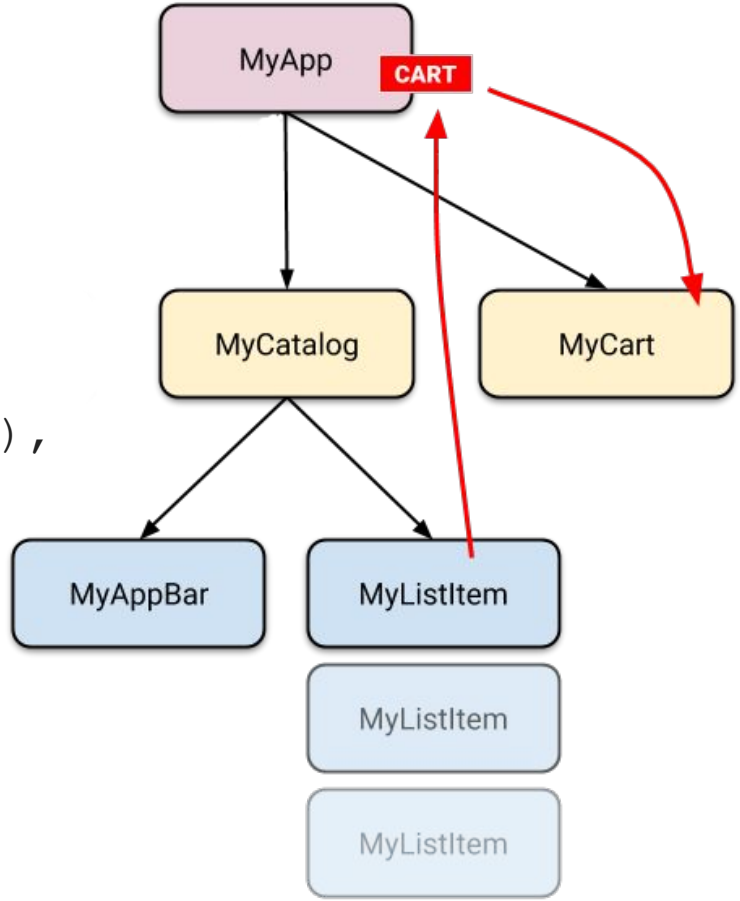
We already know where to put `ChangeNotifierProvider`: above the widgets that need to access it

In the case of **CartModel**, that means somewhere above both `MyCart` and `MyCatalog`



Provider: ChangeNotifierProvider

```
void main() {
  runApp(
    ChangeNotifierProvider(
      create: (context) => CartModel(),
      child: MyApp(),
    ),
  );
}
```



Provider: ChangeNotifierProvider

If you want to provide more than one class, you can use **`MultiProvider`**:

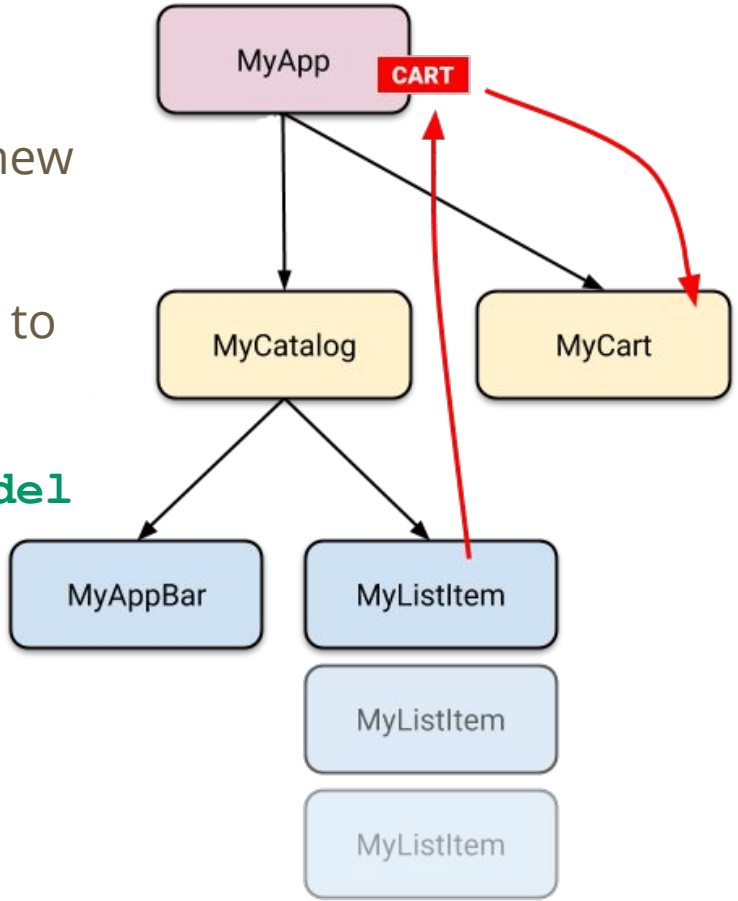
```
void main() {
  runApp(
    MultiProvider(
      providers: [
        ChangeNotifierProvider(create: (context) => CartModel()),
        Provider(create: (context) => SomeOtherClass()),
      ],
      child: MyApp(),
    ),
  );
}
```

Provider: ChangeNotifierProvider

Note that we're defining a builder that creates a new instance of `CartModel`

`ChangeNotifierProvider` is smart enough not to rebuild `CartModel` unless absolutely necessary

It also automatically calls `dispose()` on `CartModel` when the instance is no longer needed



Provider: Consumer

To use the provided state use **`Consumer`** widget

```
return Consumer<CartModel>(
  builder: (context, cart, child) {
    return Text("Total price: ${cart.totalPrice}");
  },
);
```

You must specify the type of the model that you want to access

`provider` package is based on types

Provider: Consumer

To use the provided state use **Consumer** widget

```
return Consumer<CartModel>(
  builder: (context, cart, child) {
    return Text("Total price: ${cart.totalPrice}");
  },
);
```

The only required argument of the **Consumer** widget is the **builder**

Builder is a function that is called whenever the **ChangeNotifier** changes

Provider: Consumer

```
Consumer<CartModel>(
    builder: (context, cart, child) {
      return Text("${cart.totalPrice}");
    },
);
```

The builder is called with three arguments

- The first one is **`context`**, which you also get in every **`build`** method
- The second argument is an instance of the **`ChangeNotifier.`** It is the state (**`CartModel`**) provided
- The third argument is **`child`**, which is there for optimization

Provider: Consumer

```
Consumer<CartModel>(
    builder: (context, cart, child) {
      return Text("${cart.totalPrice}");
    },
);
```

If you have a large widget subtree under your **`Consumer`** that doesn't change when the model changes, you can construct it once and get it through the builder

An example is shown in the next slide

Provider: Consumer

```
return Consumer<CartModel>(
  builder: (context, cart, child) => Stack(
        children: [
        // Use SomeExpensiveWidget here, without rebuilding every time.
          child,
          Text("Total price: ${cart.totalPrice}"),
        ],
      ),
  // Build the expensive widget here.
  child: SomeExpensiveWidget(),
);
```

Provider: Consumer

It is best practice to put your `Consumer` widgets as deep in the tree as possible

> You don’t want to rebuild large portions of the UI just because some detail somewhere changed

Next two slides show bad and good examples

Provider: Consumer

It is best practice to put your `Consumer` widgets as deep in the tree as possible

> You don’t want to rebuild large portions of the UI just because some detail somewhere changed

Next two slides show bad and good examples

Provider: Consumer

```
return Consumer<CartModel>(
  builder: (context, cart, child) {
    return HumongousWidget(
      // ...
      child: AnotherMonstrousWidget(
        // ...
        child: Text('Total price: ${cart.totalPrice}'),
      ),
    );
  },
);
```

DO NOT DO THIS

Provider: Consumer

```
return HumongousWidget(
  // ...
  child: AnotherMonstrousWidget(
    // ...
    child: Consumer<CartModel>(
      builder: (context, cart, child) {
        return Text('Total price: ${cart.totalPrice}');
      },
    ),
  ),
);
```

DO THIS

Provider.of

You can access the provided class like so

```
CartModel cartModel =
          Provider.of<CartModel>(context, listen: false);
```

Bloc Overview

State management solution

Below are common developer needs that Bloc tries to address

- knowing what state your application is in at any point in time
- Easily testing every case to make sure our app is responding appropriately
- Recording every single user interaction in your application so that you can make data-driven decisions

Bloc Overview

State management solution

Below are common developer needs that Bloc tries to address

- Working as efficiently as possible and reuse components both within your application and across other applications
- Letting many developers to seamlessly work within a single code base following the same patterns and conventions
- Developing fast and reactive apps

https://bloclibrary.dev/

Bloc Overview

Overall, Bloc attempts to make state changes predictable by regulating

when a state change can occur and

enforcing a single way to change state throughout an entire application

Installing and Importing Bloc package

Installation

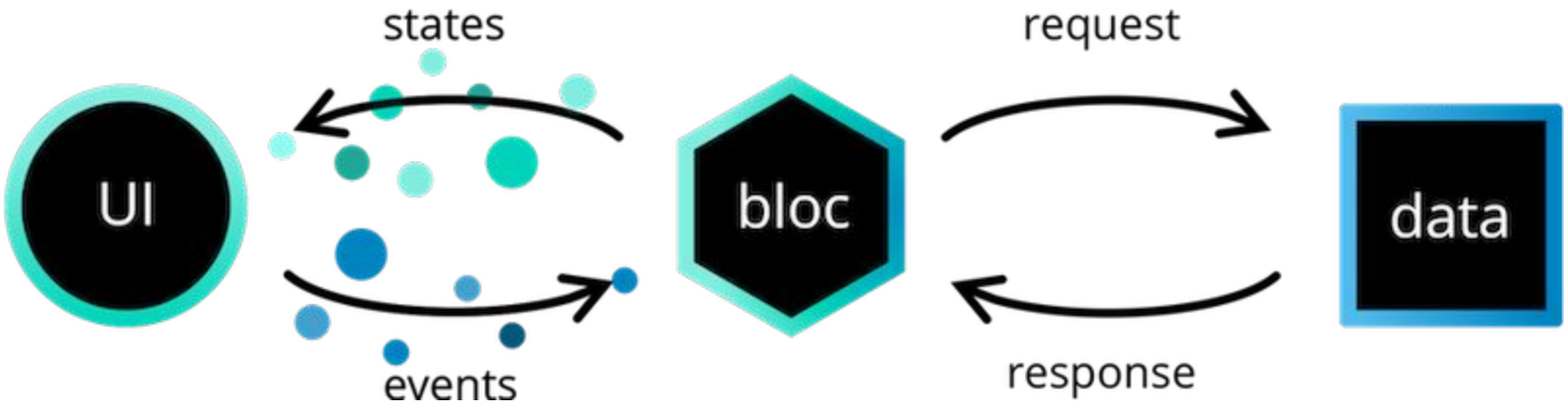
Add the `flutter_bloc` package to your `pubspec.yaml` as a dependency

```
dependencies:
  flutter_bloc: ^6.1.1
```

Import

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

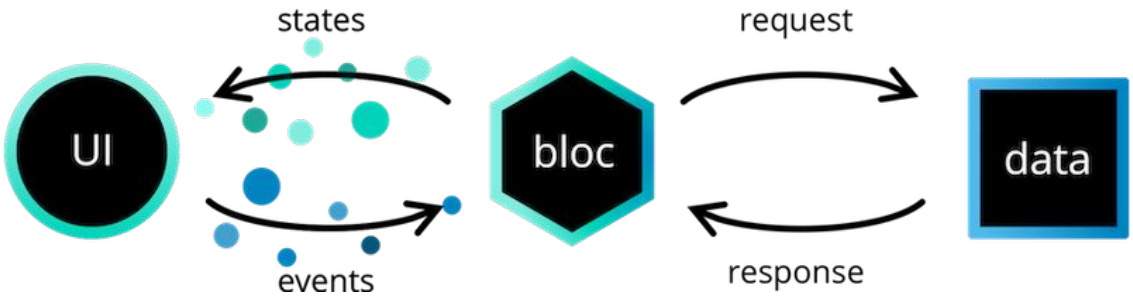
Bloc Architecture



Bloc Architecture

Using the bloc library allows us to separate our application into three layers

- **Presentation**
- **Business Logic**
- **Data**
  - **Repository**
  - **Data Provider**



Bloc Architecture: Data Layer

The data layer's responsibility is to retrieve/manipulate data from one or more sources

The data layer can be split into two parts

- **Repository**
- **Data Provider**

This layer is the lowest level of the application and **interacts with databases**, **network requests**, and **other asynchronous data sources**

Bloc Architecture: Data Layer -> Data Provider

The data provider's responsibility is to provide raw data

The data provider should be generic and versatile

The data provider will usually expose simple APIs to perform CRUD operations

We might have a `createData`, `readData`, `updateData`, and `deleteData` method as part of our data layer

Bloc Architecture: Data Layer -> Data Provider

```
class DataProvider {

    Future<RawData> readData() async {

        // Read from DB or make network request etc...

    }

}
```

Bloc Architecture: Data Layer -> Repository

The repository layer is a wrapper around one or more data providers with which the Bloc Layer communicates

The repository layer can interact with multiple data providers and perform transformations on the data before handing the result to the business logic Layer

An example structure is shown in the next slide

Bloc Architecture: Data Layer -> Repository

```
class Repository {

    final DataProviderA dataProviderA;
    final DataProviderB dataProviderB;

    Future<Data> getAllDataThatMeetsRequirements() async {

        final RawDataA dataSetA = await dataProviderA.readData();
        final RawDataB dataSetB = await dataProviderB.readData();

        final Data filteredData = _filterData(dataSetA, dataSetB);

        return filteredData;
    }
}
```

Bloc Architecture: Business Logic Layer

The business logic layer's responsibility is to respond to input from the presentation layer with new states

This layer can depend on one or more repositories to retrieve data needed to build up the application state

Think of the business logic layer as the **bridge** between the user interface **(presentation layer) and the data layer**

Bloc Architecture: Business Logic Layer

The business logic layer is notified of events/actions from the presentation layer and then communicates with repository in order to build a new state for the presentation layer to consume

An example code is shown in the next slide

Bloc Architecture: Business Logic Layer

```
class BusinessLogicComponent extends Bloc<MyEvent, MyState> {

    final Repository repository;

    Stream mapEventToState(event) async* {
        if (event is AppStarted) {
            try {
             final data = await repository.getAllDataThatMeetsRequirements();
             yield Success(data);
            } catch (error) {
                yield Failure(error);
            }
        }
    }
}
```

Bloc Architecture: Presentation Layer

The presentation layer's responsibility is to figure out how to render itself based on one or more bloc states

In addition, it should handle user input and application lifecycle events

Most applications flows will start with a `AppStart` event which triggers the application to fetch some data to present to the user

In this scenario, the presentation layer would add an `AppStart` event

An example is shown in the next slide

Bloc Architecture: Presentation Layer

```
class PresentationComponent {
    final Bloc bloc;

    PresentationComponent() {
        bloc.add(AppStarted());
    }

    build() {
        // render UI based on bloc state
    }
}
```

Core Concepts of Bloc in Flutter

Bloc Widgets

- `BlocProvider`
- `BlocBuilder`
- `BlocListener`
- `BlocConsumer`
- `RepositoryProvider`
- `MultiBlocProvider`
- `MultiBlocListener`
- `MultiRepositoryProvider`

Core Concepts of Bloc: `BlocProvider`

`BlocProvider` is a Flutter widget which provides a **bloc** to its children via **`BlocProvider.of<T>(context)`**

It is used as a **dependency injection (DI)** widget so that a **single instance** of a bloc can be provided to multiple widgets within a subtree

In most cases, **`BlocProvider`** should be used to create new blocs which will be made available to the rest of the subtree

> In this case, since **`BlocProvider`** is responsible for creating the bloc, it will automatically handle closing the bloc

Core Concepts of Bloc: `BlocProvider`

```
BlocProvider(
  create: (BuildContext context) => BlocA(),
  child: ChildA(),
);
```

Core Concepts of Bloc: `BlocProvider`

By default, **`BlocProvider`** will create the bloc lazily, meaning **create** will get executed when the bloc is looked up via **`BlocProvider.of<T>(context)`**

To override this behavior and force create to be run immediately, lazy can be set to false

```
BlocProvider(
  lazy: false,
  create: (BuildContext context) => BlocA(),
  child: ChildA(),
);
```

Core Concepts of Bloc: `BlocProvider`

In some cases, **`BlocProvider`** can be used to provide an existing bloc to a new portion of the widget tree

This will be most commonly used when an existing bloc needs to be made available to a new route

In this case, **`BlocProvider`** will not automatically close the bloc since it did not create it

Core Concepts of Bloc: `BlocProvider`

```
BlocProvider.value(
  value: BlocProvider.of<BlocA>(context),
  child: ScreenA(),
);
```

from either `ChildA`, or `ScreenA` we can retrieve `BlocA` with

```
// with extensions
context.read<BlocA>();
```

```
// without extensions
BlocProvider.of<BlocA>(context)
```

Core Concepts of Bloc: `BlocBuilder`

`BlocBuilder` is a Flutter widget which requires a **`Bloc`** and a **`builder`** function

`BlocBuilder` handles building the widget in response to new states

The **`builder`** function will potentially be called many times and should be a pure function that returns a widget in response to the state

Core Concepts of Bloc: `BlocBuilder`

If the `cubit` parameter is omitted, `BlocBuilder` will automatically perform a lookup using `BlocProvider` and the current `BuildContext`

```
BlocBuilder<BlocA, BlocAState>(
  builder: (context, state) {
    // return widget here based on BlocA's state
  }
)
```

Core Concepts of Bloc: `BlocBuilder`

Only specify the **`bloc`** if you wish to provide a **`bloc`** that will be scoped to a single widget and isn't accessible via a parent **`BlocProvider`** and the current **`BuildContext`**

```
BlocBuilder<BlocA, BlocAState>(
  cubit: blocA, // provide the local cubit instance
  builder: (context, state) {
    // return widget here based on BlocA's state
  }
)
```

Core Concepts of Bloc: `BlocBuilder`

For fine-grained control over when the builder function is called an optional `buildWhen` can be provided

`buildWhen` takes the **previous bloc state** and **current bloc state** and returns a boolean

If `buildWhen` returns true, builder will be called with state and the widget will rebuild

If `buildWhen` returns false, builder will not be called with state and no rebuild will occur

Core Concepts of Bloc: `BlocBuilder`

```
BlocBuilder<BlocA, BlocAState>(
  buildWhen: (previousState, state) {
    // return true/false to determine whether or not
    // to rebuild the widget with state
  },
  builder: (context, state) {
    // return widget here based on BlocA's state
  }
)
```

Core Concepts of Bloc: `BlocListener`

`BlocListener` is a Flutter widget which takes a **`BlocWidgetListener`** and an optional **`Bloc`** and invokes the listener in response to state changes in the **bloc**

It should be used for functionality that needs to occur once per state change such as **`navigation`**, showing a **`SnackBar`**, showing a **`Dialog`**, etc...

Core Concepts of Bloc: `BlocListener`

`BlocListener` is a Flutter widget which takes a **`BlocWidgetListener`** and an optional **`Bloc`** and invokes the listener in response to state changes in the **bloc**

It should be used for functionality that needs to occur once per state change such as **`navigation`**, showing a **`SnackBar`**, showing a **`Dialog`**, etc...

`listener` is only called once for each state change (NOT including the initial state) unlike **`builder`** in **`BlocBuilder`** and is a void function

Core Concepts of Bloc: `BlocListener`

If the `cubit` parameter is omitted, `BlocListener` will automatically perform a lookup using `BlocProvider` and the current `BuildContext`

```
BlocListener<BlocA, BlocAState>(
  listener: (context, state) {
    // do stuff here based on BlocA's state
  },
  child: Container(),
)
```

Core Concepts of Bloc: `BlocListener`

Only specify the **`bloc`** if you wish to provide a **`bloc`** that is otherwise not accessible via **`BlocProvider`** and the current **`BuildContext`**

```
BlocListener<BlocA, BlocAState>(
  cubit: blocA,
  listener: (context, state) {
    // do stuff here based on BlocA's state
  },
  child: Container()
)
```

Core Concepts of Bloc: `BlocListener`

For fine-grained control over when the listener function is called an optional `listenWhen` can be provided

`listenWhen` takes the **previous bloc state** and **current bloc state** and returns a boolean

If `listenWhen` returns true, listener will be called with state

If `listenWhen` returns false, listener will not be called with state

Core Concepts of Bloc: `BlocListener`

```
BlocListener<BlocA, BlocAState>(
  listenWhen: (previousState, state) {
    // return true/false to determine whether or not
    // to call listener with state
  },
  listener: (context, state) {
    // do stuff here based on BlocA's state
  },
  child: Container(),
)
```

Core Concepts of Bloc: `BlocConsumer`

`BlocConsumer` exposes a **`builder`** and **`listener`** in order to react to new states

`BlocConsumer` is analogous to a nested **`BlocListener`** and **`BlocBuilder`** but reduces the amount of boilerplate needed

`BlocConsumer` should only be used when it is necessary to both rebuild UI and execute other reactions to state changes in the cubit

Core Concepts of Bloc: `BlocConsumer`

```
BlocConsumer<BlocA, BlocAState>(
  listener: (context, state) {
    // do stuff here based on BlocA's state
  },
  builder: (context, state) {
    // return widget here based on BlocA's state
  }
)
```

Core Concepts of Bloc: `BlocConsumer`

An optional **`listenWhen`** and **`buildWhen`** can be implemented for more granular control over when **`listener`** and **`builder`** are called

The **`listenWhen`** and **`buildWhen`** will be invoked on each cubit state change

They each take the **previous state** and **current state** and must return a bool which determines whether or not the **`builder`** and/or **`listener`** function will be invoked

Core Concepts of Bloc: `BlocConsumer`

The previous state will be initialized to the state of the **`cubit`** when the **`BlocConsumer`** is initialized

`listenWhen` and **`buildWhen`** are optional and if they aren't implemented, they will default to true

Core Concepts of Bloc: `BlocConsumer`

```
BlocConsumer<BlocA, BlocAState>(
  listenWhen: (previous, current) {
    // return true/false to determine whether or not
    // to invoke listener with state
  },
  listener: (context, state) {
    // do stuff here based on BlocA's state
  },
  buildWhen: (previous, current) {
    // return true/false to determine whether or not
    // to rebuild the widget with state
  },
  builder: (context, state) {
    // return widget here based on BlocA's state
  }
)
```

Core Concepts of Bloc: `RepositoryProvider`

`RepositoryProvider` is a Flutter widget which provides a repository to its children via `RepositoryProvider.of<T>(context)`

It is used as a **dependency injection (DI)** widget so that a single instance of a repository can be provided to multiple widgets within a subtree

`BlocProvider` should be used to provide blocs whereas `RepositoryProvider` should only be used for repositories

Core Concepts of Bloc: `RepositoryProvider`

```
RepositoryProvider(
  create: (context) => RepositoryA(),
  child: ChildA(),
);
```

then from `ChildA` we can retrieve the `Repository` instance with:

```
// with extensions
context.read<RepositoryA>();
```

```
// without extensions
RepositoryProvider.of<RepositoryA>(context)
```

Core Concepts of Bloc: `MultiBlocProvider`

`MultiBlocProvider` is a Flutter widget that merges multiple `BlocProvider` widgets into one

`MultiBlocProvider` improves the readability and eliminates the need to nest multiple BlocProviders

Core Concepts of Bloc: `MultiBlocProvider`

By using `MultiBlocProvider` we can go **from**

```
BlocProvider<BlocA>(
  create: (BuildContext context) => BlocA(),
  child: BlocProvider<BlocB>(
    create: (BuildContext context) => BlocB(),
    child: BlocProvider<BlocC>(
      create: (BuildContext context) => BlocC(),
      child: ChildA(),
    )
  )
)
```

To the one shown in the next slide

Core Concepts of Bloc: `MultiBlocProvider`

```
MultiBlocProvider(
  providers: [
    BlocProvider<BlocA>(
      create: (BuildContext context) => BlocA(),
    ),
    BlocProvider<BlocB>(
      create: (BuildContext context) => BlocB(),
    ),
    BlocProvider<BlocC>(
      create: (BuildContext context) => BlocC(),
    ),
  ],
  child: ChildA(),
)
```

Core Concepts of Bloc: `MultiBlocListener`

`MultiBlocListener` is a Flutter widget that merges multiple **`BlocListener`** widgets into one

`MultiBlocListener` improves the readability and eliminates the need to nest multiple **`BlocListeners`**

Core Concepts of Bloc: `MultiBlocListener`

By using `MultiBlocListener` we can go from

```
BlocListener<BlocA, BlocAState>(
  listener: (context, state) {},
  child: BlocListener<BlocB, BlocBState>(
    listener: (context, state) {},
    child: BlocListener<BlocC, BlocCState>(
      listener: (context, state) {},
      child: ChildA(),
    ),
  ),
)
```

To the one shown in the next slide

Core Concepts of Bloc: `MultiBlocListener`

```
MultiBlocListener(
  listeners: [
    BlocListener<BlocA, BlocAState>(
      listener: (context, state) {},
    ),
    BlocListener<BlocB, BlocBState>(
      listener: (context, state) {},
    ),
    BlocListener<BlocC, BlocCState>(
      listener: (context, state) {},
    ),
  ],
  child: ChildA(),
)
```

Core Concepts of Bloc: `MultiRepositoryProvider`

`MultiRepositoryProvider` is a Flutter widget that merges multiple `RepositoryProvider` widgets into one

`MultiRepositoryProvider` improves the readability and eliminates the need to nest multiple `RepositoryProvider`

Core Concepts of Bloc: `MultiBlocListener`

By using `MultiRepositoryProvider` we can go from

```
RepositoryProvider<RepositoryA>(
  create: (context) => RepositoryA(),
  child: RepositoryProvider<RepositoryB>(
    create: (context) => RepositoryB(),
    child: RepositoryProvider<RepositoryC>(
      create: (context) => RepositoryC(),
      child: ChildA(),
    )
  )
)
```

To the one shown in the next slide

Core Concepts of Bloc: `MultiBlocListener`

```
MultiRepositoryProvider(
  providers: [
    RepositoryProvider<RepositoryA>(
      create: (context) => RepositoryA(),
    ),
    RepositoryProvider<RepositoryB>(
      create: (context) => RepositoryB(),
    ),
    RepositoryProvider<RepositoryC>(
      create: (context) => RepositoryC(),
    ),
  ],
  child: ChildA(),
)
```

Examples

https://bloclibrary.dev/#/fluttercountertutorial

https://bloclibrary.dev/#/fluttertimertutorial

https://bloclibrary.dev/#/flutterlogintutorial

https://bloclibrary.dev/#/flutterweathertutorial

https://bloclibrary.dev/#/fluttertodostutorial