Link: <https://docs.flutter.dev/data-and-backend/state-mgmt/options>

Link:<https://medium.com/@enitinmehra/state-management-in-flutter-a-comprehensive-guide-7212772f026d>

State management in Flutter is a critical concept for building robust, responsive, and efficient mobile applications. It refers to the management and manipulation of data within an app to ensure that the user interface (UI) accurately reflects the current state of the application. Flutter, as a reactive framework, offers various approaches to handling and updating state data. In this guide, we’ll explore what state management is and delve into some common types of state management in Flutter.

Press enter or click to view image in full size



**What is State Management?**

State, in the context of a Flutter app, represents the information that can change over time and influences the appearance and behavior of the UI. Examples of state in a Flutter app include user input, network data, device orientation, and more. Effective state management involves handling and updating this data to keep the UI in sync with the application’s logic.

In Flutter, the UI is typically divided into widgets, and state management revolves around how data is shared and updated between these widgets. Good state management ensures that when a change occurs in one part of the app, it accurately reflects in the other relevant parts of the UI.

**Types of State Management in Flutter**

Flutter offers a range of state management solutions to cater to various project complexities and developer preferences. Here are some common types of state management in Flutter:

**1. Stateful Widgets**

Get NITIN KUMAR’s stories in your inbox

Join Medium for free to get updates from this writer.

Subscribe

Stateful Widgets are Flutter’s built-in mechanism for handling state. They consist of two classes: `StatefulWidget` and `State`. The `StatefulWidget` represents the immutable part of the widget, while the `State` object holds the mutable state data. When the state data changes, calling `setState()` triggers a rebuild of the widget, updating the UI.

class MyStatefulWidget extends StatefulWidget {  
 @override  
 \_MyStatefulWidgetState createState() => \_MyStatefulWidgetState();  
}  
  
class \_MyStatefulWidgetState extends State<MyStatefulWidget> {  
 int \_counter = 0;  
  
 void \_incrementCounter() {  
 setState(() {  
 \_counter++;  
 });  
 }  
  
 @override  
 Widget build(BuildContext context) {  
 return Column(  
 children: <Widget>[  
 Text('Counter: $\_counter'),  
 ElevatedButton(  
 onPressed: \_incrementCounter,  
 child: Text('Increment'),  
 ),  
 ],  
 );  
 }  
}

**2. Provider Package**

A well-liked state management package for Flutter is Provider. It allows you to create and manage providers that hold the application’s state. Widgets can then listen to these providers for changes in the state. Provider is particularly useful for managing global or shared state.

import 'package:flutter/material.dart';  
import 'package:provider/provider.dart';  
  
class MyWidget extends StatelessWidget {  
 @override  
 Widget build(BuildContext context) {  
 final counter = Provider.of<Counter>(context);  
  
 return Column(  
 children: <Widget>[  
 Text('Counter: ${counter.value}'),  
 ElevatedButton(  
 onPressed: () => counter.increment(),  
 child: Text('Increment'),  
 ),  
 ],  
 );  
 }  
}  
  
class Counter with ChangeNotifier {  
 int \_value = 0;  
  
 int get value => \_value;  
  
 void increment() {  
 \_value++;  
 notifyListeners();  
 }  
}

**3. Bloc Pattern**

The Bloc (Business Logic Component) pattern is commonly used for state management in Flutter, especially for complex applications. It separates the UI from the business logic and uses streams to manage and propagate state changes. The `flutter\_bloc` package is widely adopted for implementing the Bloc pattern.

import 'package:flutter/material.dart';  
import 'package:flutter\_bloc/flutter\_bloc.dart';  
  
class CounterCubit extends Cubit<int> {  
 CounterCubit() : super(0);  
  
 void increment() => emit(state + 1);  
}  
  
class MyWidget extends StatelessWidget {  
 @override  
 Widget build(BuildContext context) {  
 final counterCubit = context.read<CounterCubit>();  
  
 return Column(  
 children: <Widget>[  
 BlocBuilder<CounterCubit, int>(  
 builder: (context, state) {  
 return Text('Counter: $state');  
 },  
 ),  
 ElevatedButton(  
 onPressed: () => counterCubit.increment(),  
 child: Text('Increment'),  
 ),  
 ],  
 );  
 }  
}

**4. GetX Package**

GetX is a powerful and lightweight state management library for Flutter. It provides a simplified and high-performance way to manage state, handle navigation, and inject dependencies. GetX is known for its simplicity and efficiency.

import 'package:flutter/material.dart';  
import 'package:get/get.dart';  
  
class MyWidget extends StatelessWidget {  
 final count = 0.obs;  
  
 void increment() => count.value++;  
  
 @override  
 Widget build(BuildContext context) {  
 return Column(  
 children: <Widget>[  
 Obx(() => Text('Counter: $count')),  
 ElevatedButton(  
 onPressed: increment,  
 child: Text('Increment'),  
 ),  
 ],  
 );  
 }  
}

These are just a few examples of state management options in Flutter. The choice of state management approach depends on factors like project complexity, team preferences, and the specific requirements of your application. Flutter’s flexibility allows you to choose the most suitable state management solution or even combine multiple techniques when needed to create efficient and maintainable Flutter apps.