Chapter 7: State Management

7.1. What is the State?

A state is information that can be read when the widget is built and might change or modified over a lifetime of the app. If you want to change your widget, you need to update the state object, which can be done by using the **setState()** function available for Stateful widgets. The **setState()** function allows us to set the properties of the state object that triggers a redraw of the UI.

The state management is one of the most popular and necessary processes in the lifecycle of an application. According to official documentation, Flutter is declarative. It means Flutter builds its UI by reflecting the current state of your app. The following figure explains it more clearly where you can build a UI from the application state.

In Flutter, the state management categorizes into two conceptual types, which are given below:

1. Ephemeral State
2. App State

Ephemeral State

This state is also known as UI State or local state. It is a type of state which is related to the specific widget, or you can say that it is a state that contains in a single widget. In this kind of state, you do not need to use state management techniques. The common example of this state is Text Field.

Example

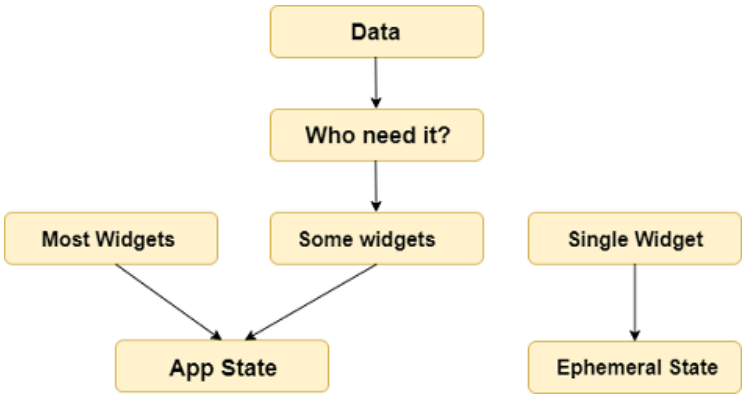
```
class _MyAppState extends State<MyApp> {
  String college = 'CCI';
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
          body: ElevatedButton(
              onPressed: () {
                setState(() {
                  college = "BECO";
                });
              },
              child: Text(college))),
    );
  }
}
```

In the above example, the **college** is an ephemeral state. Here, only the **setState()** function inside the StatefulWidget's class can access the **college**. The build method calls a **setState()** function, which does the modification in the state variables. When this method is executed, the widget object is replaced with the new one, which gives the modified variable value.

App State

It is different from the ephemeral state. It is a type of state that we want to **share** across various parts of our app and want to keep between user sessions. Thus, this type of state can be used globally. Sometimes it is also known as application state or shared state. Some of the examples of this state are 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, etc.

The following diagram explains the difference between the ephemeral state and the app state more appropriately.



7.2. State management with Provider package

The simplest example of app state management can be learned by using the **provider package.** The state management with the provider is easy to understand and requires less coding. A provider is a **third-party** library. Here, we need to understand three main concepts to use this library.

1. ChangeNotifier
2. ChangeNotifierProvider
3. Consumer

ChangeNotifier

ChangeNotifier is a simple mixin, which provides change notification to its listeners. It is easy to understand, implement, and optimized for a small number of listeners. It is used for the listener to observe a model for changes. In this, we only use the **notifyListener()** method to inform the listeners.

For example, let us define a model based on ChangeNotifier. In this model, the **Counter** is extended with ChangeNotifier, which is used to notify its listeners when we call notifyListeners(). It is the only method that needs to be implemented in a ChangeNotifier model. In this example, we declared two functions the **increment** and **decrement,** which are used to increase and decrease the value. We can call notifyListeners() method any time the model changes in a way that might change your app's UI.

```dart
import 'package:flutter/material.dart';

class Counter with ChangeNotifier {
 int _counter;

 Counter(this._counter);

 getCounter() => _counter;
 setCounter(int counter) => _counter = counter;

 void increment() {
  _counter++;
  notifyListeners();
 }

 void decrement() {
  _counter--;
  notifyListeners();
 }
}
```

ChangeNotifierProvider

ChangeNotifierProvider is the widget that provides an **instance** of a ChangeNotifier to its descendants. It comes from the provider package. The following code snippets help to understand the concept of ChangeNotifierProvider.

Here, we have defined a **builder/create** who will create a new instance of the **Counter** model. ChangeNotifierProvider does not rebuild Counter unless there is a need for this. It will also automatically call the **dispose()** method on the Counter model when the instance is no longer needed.

```
class MyApp extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
  return MaterialApp(
   theme: ThemeData(
    primarySwatch: Colors.indigo,
   ),
   home: ChangeNotifierProvider<CounterModel>(
    builder: (_) => CounterModel(),
    child: CounterView(),
   ),
  );
 }
}
```

If there is a need to provide more than one class, you can use **MultiProvider.** The MultiProvider is a list of all the different Providers being used within its scope. Without using this, we would have to nest our Providers with one being the child of another and another. We can understand this from the below code.

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

Consumer

It is a type of provider that does not do any fancy work. It just calls the provider in a new widget and delegates its build implementation to the builder. The following code explains it more clearly./p>

```
return Consumer<Counter>(
 builder: (context, count, child) {
  return Text("Total price: ${count.total}");
 },
);
```

In the above example, you can see that the **consumer widget** only requires a builder function, which is called whenever the ChangeNotifier changes. The builder function contains **three** arguments, which are **context, count, and child.** The first argument, context, contain in every build() method. The second argument is the instance of the ChangeNotifier, and the third argument is the child that is used for optimization. It is the best idea to put the consumer widget as deep as in the tree as possible.