



Flutter State Management

With Edd Gachira



Understanding state

The state of an app can very simply be defined as anything that exists in the memory of the app while the app is running. This includes all the widgets that maintain the UI of the app including the buttons, text fonts, icons, animations, etc. So now as we know what are these states let's dive directly into our main topic i.e what are these stateful and stateless widgets and how do they differ from one another.

State: The State is the information that can be read synchronously when the widget is built and might change during the lifetime of the widget.



Stateless and stateful widgets

Stateless Widget: The widgets whose state can not be altered once they are built are called *stateless widgets*. These widgets are *immutable* once they are built i.e any amount of change in the variables, icons, buttons, or retrieving data can not change the state of the app.

Stateful Widgets: The widgets whose state can be altered once they are built are called *stateful Widgets*. These states are *mutable* and can be changed multiple times in their lifetime. This simply means the state of an app can change multiple times with different sets of variables, inputs, data.

Classes that inherit “*Stateful Widget*” are *immutable*. But the *State* is *mutable* which changes in the runtime when the user interacts with it.



Difference between stateless and stateful widgets

Stateless Widget:

- Stateless Widgets are static widgets.
- They do not depend on any data change or any behavior change.
- Stateless Widgets do not have a state, they will be rendered once and will not update themselves, but will only be updated when external data changes.
- For Example: Text, Icon, *RaisedButton* are Stateless Widgets.




Stateful Widget:

- Stateful Widgets are dynamic widgets.
- They can be updated during runtime based on user action or data change.
- Stateful Widgets have an internal state and can re-render if the input data changes or if Widget's state changes.
- For Example: Checkbox, Radio Button, Slider are Stateful Widgets



State Review

In Flutter, state management refers to the process of managing the state, or data, of an app in a consistent and predictable way. The basic idea is that the state of the app is stored in a single, centralized location, and any part of the app that needs to access or update that state can do so in a consistent and predictable way.



There are several popular state management techniques in Flutter, including:

setState(): A built-in Flutter method that allows you to update the state of a widget and rebuild the widget tree.

InheritedWidget: A built-in Flutter widget that allows you to store and access state at a higher level in the widget tree.

Provider package: A popular third-party package that makes it easy to manage state across a large app, by allowing you to provide and access state at different levels in the widget tree.

Ultimately, the best approach for state management in a Flutter app will depend on the specific needs of the app and the preferences of the development team.



Flutter Riverpod

flutter_riverpod is a package that provides additional functionality on top of the **riverpod** package, specifically for use in Flutter applications. It is an extension of the riverpod package, which is a state management library for Flutter.

flutter_riverpod provides additional widgets and functions that make it even easier to use riverpod in Flutter apps. For example, it provides a **ProviderScope** widget that automatically **disposes** providers when they're no longer in use, and a **Consumer** widget that allows you to rebuild a part of your widget tree when the state of a provider changes.

flutter_riverpod also provides additional functionality such as **AutoDisposeProvider** which will automatically dispose the provider when the widget is removed from the widget tree, and **Watching** which allows you to observe the state of a provider without rebuilding the widget tree.

In summary, **flutter_riverpod** is an extension of the riverpod package that makes it even easier to use in Flutter apps by providing additional widgets and functionality specifically tailored for Flutter.



Advantages of riverpod

Simplicity: Riverpod is designed to be simple and easy to use, with a small set of core concepts that are easy to understand.

Reactive: Riverpod is built on top of the provider package, which uses a reactive programming paradigm, so the state updates automatically when the dependencies change. This makes it easy to write predictable and efficient code.

Scoped: Riverpod allows you to create scoped providers, which limits the scope of the state to a specific part of the widget tree. This makes it easy to manage the state of complex apps and avoid global state issues.

Testing: Riverpod makes it easy to test the state of an app by allowing you to mock or override providers.



Performance: Riverpod is designed with performance in mind, it uses the provider package to avoid unnecessary rebuilds, which makes it efficient and fast for large apps.

Flexibility: Riverpod allows you to easily combine multiple providers to create more complex state management.


Community support: Riverpod is an actively maintained library, it has a growing community and it is getting more popular among flutter developers.



Understanding Riverpod

First, you will need to create a **Provider** which is a way to store and manage the state in Riverpod. You can create a provider by using the `Provider` class and passing in the type of the state you want to store and a create function that returns the initial state. Here is an example of creating a simple provider that stores a string:

```
final myProvider = Provider<String>((ref) => "Initial value");
```




Then, you can use the `Provider.of` method to access the state of the provider and the `ref.read` method to read the state of the provider. Here is an example of how to use the `Provider.of` method to access the state of the provider in a widget:

```
class MyWidget extends StatelessWidget {  
  
  @override  
  
  Widget build(BuildContext context) {  
  
    final value = Provider.of<String>(context);  
  
    return Text(value);  
  
  }  
}
```



To update the state of the provider, you can use the **ref.state = operator**, here an example of how to update the state of the provider inside a button's onPressed:



```
class MyWidget extends StatelessWidget {  
  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
      body: Column(  
        children: [  
          Text(Provider.of<String>(context)),  
          RaisedButton(  
            onPressed: () {  
              context.read(myProvider).state = "New value";  
            },  
            child: Text("Update"),  
          )  
        ],  
      ),  
    );  
  }  
}
```



It's important to note that you should wrap the root of your app with the **ProviderScope** widget in order to make all providers available to the entire app.

```
void main() {  
  runApp(  
    ProviderScope(  
      child: MyApp(),  
    ),  
  );  
}
```



You can also use the **Consumer** widget to rebuild a part of your widget tree when the state of the provider changes, this is useful when you have complex widgets and you want to avoid unnecessary rebuilds.



```
class MyWidget extends StatelessWidget {  
  
  @override  
  
  Widget build(BuildContext context) {  
  
    return Scaffold(  
  
      body: Consumer<String>(  
  
        builder: (context, value, child) {  
  
          return Text(value);  
  
        },  

```



These are just some basic examples of how to use Riverpod to manage the state of a Flutter app. There are many more advanced features and concepts in Riverpod that can be used to create more complex state management.