Flutter:Introduction to Stateful Widgets

Prashanth B S[1]

[1]Department of Information Science & Engineering, Nitte Meenakshi Institute of Technology,
Yelahanka - 560064, Bengaluru

1 Stateful Widgets v/s Stateless Widgets

The state of an app can be defined as anything that exists in the memory of the app while the app is running. This includes the properties of the object, the values each property holds.State is simply the information of a StatefulWidget. Every StatefulWidget has a State Object. This State Object keeps a track of the variables and functions that we define inside a StatefulWidget[1]. Based on the state property,the Widgets can also be classified into two types they are,

1. *Stateless Widgets*: 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 off change in the variables, icon, buttons, or retrieving data can not change the state of the app

2. *Stateful Widgets*: The Widgets whose state can be altered once they are built are called stateful widgets. These states are mutable and can be changes multiple times in their app lifetime.

When an app is implemented as either stateful or stateless, then only user can use the *Hot-Reload* feature of the flutter. During Hot-reload the Dart compiler instead of recompiling the full app, looks for the code which has been changed from the previous state. This code is called as dirty code/changed code. The Hot-reload will only recompile the dirty code while running the app. This saves time and app will be loaded fastly.

The stateful widgets are immutable while building it. But can change the state over time. In order to update the UI change in a stateful widget, we use *setState*() function. The code inside the setState function is marked as dirty and will be updated on save. The setState function takes a function as an argument, but it can be an empty/void function too. The syntax of setState is as shown below,

```
void setState(VoidCallback function) {
  // code marked as dirty which will be redrawn
}
// typical syntax
setState((){
   // Dirty code
   }
);
```

[1]https://dev.to/nicks101/when-to-use-setstate-in-flutter-380

2 Exercise

Implement a hybrid Dice rolling app to demonstrate the use of setState() method for marking part of the code as dirty, and refereshing the app must result in update of UI.

The following are the steps to be followed,

1. Create a new flutter app
2. Download the Dicee App Stub file from this link https://github.com/londonappbrewery/dicee-flutter. Copy the images folder from the Stub project and place them in to your project directory created by you . The snapshot upon copy-pasting the images folder into your project should look like this figure 1.

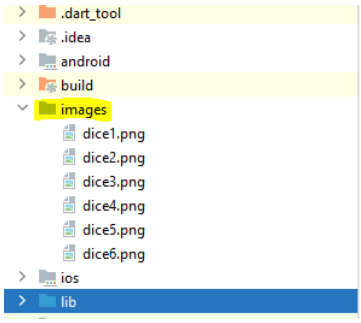


Figure 1: After copy-pasting the image folder into new Project

3. Open the pubspec.yaml file to reference the images we copied. Append the images folder under asset property as shown below . Click on pub-get to index the new resources. Return back to main.dart file and click on get-dependencies pop up to be in sync with the updated yaml file.

```
// pubspec.yaml file
flutter:
  uses-material-design: true
  assets:           // add these two line, careful about the spacing.
    - images/
```

4. Build a simple Material App with appBar and make the body points to a stateful Widget called DicePage() as shown below,

```
void main() {
  runApp(MaterialApp(
    home:Scaffold(
      appBar: AppBar(title: Text('DICEE'),centerTitle: true,), // Simple Appbar
      body: DicePage(), // body now points to a DicePage which is a stateful widget
    ),
  ));
}
class DicePage extends StatefulWidget {
  const DicePage({Key? key}) : super(key: key);

  @override
  _DicePageState createState() => _DicePageState();
}
```

```
class _DicePageState extends State<DicePage> {
    @override
    Widget build(BuildContext context) {
    return Container(); // Here we build our body widget Tree
    }
}
```

5. Build the Widget Tree for body as shown below in figure 2. The widget tree comprises of following widgets,

   (a) *Expanded*: A widget that expands a child of a Row, Column, or Flex so that the child fills the available space. Using an Expanded widget makes a child of a Row, Column, or Flex expand to fill the available space along the main axis. If multiple children are expanded, the available space is divided among them according to the flex factor.[2]. The following code snippet demonstrates the same,

   ```
   Expanded(
         flex: 2,
         child: Container(
         color: Colors.amber,
         height: 100,
         ),
   ),
   ```

   (b) *TextButton*:Text Button is a Material Design's button that comes without border or elevation change by default. Therefore, it relies on the position relative to other widgets[3]. A simple definition of the TextButton is shown below,

   ```
   TextButton(
       child: Text('Simple text Button'), // can be Text/Image button too
      style: TextButton.styleFrom(
              primary: Colors.black, // text color
              backgroundColor: Colors.blueAccent, // background color
       ),
       onPressed: () {
         print('Pressed');
       }
   )
   ```

6. The Code after implementing the left half of the Widget tree is shown below,

```
@override
  Widget build(BuildContext context) {
    return Center(
      child: Container(
        child: Row(
          children: [
            Expanded(
              flex: 1,
              child: TextButton(
                style: TextButton.styleFrom(
```

[2] https://api.flutter.dev/flutter/widgets/Expanded-class.html
[3] https://www.woolha.com/tutorials/flutter-using-textbutton-widget-examples

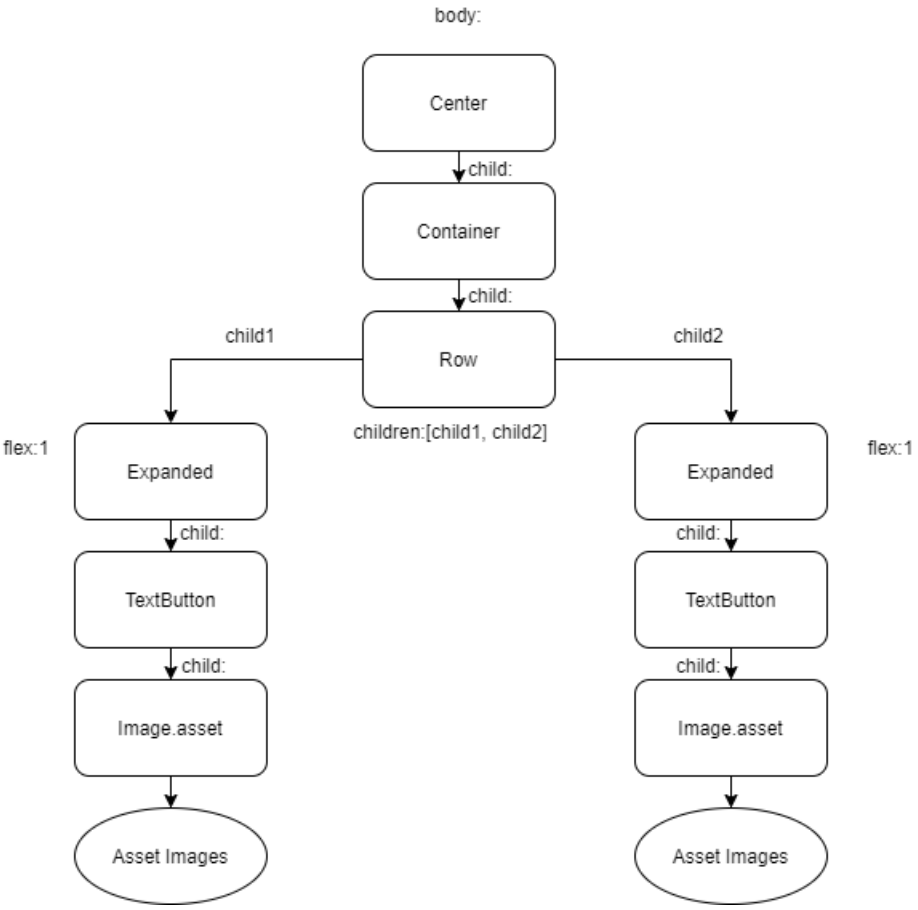


Figure 2: Widget Tree for body of the App

```
              backgroundColor: Colors.blueAccent,
            ),
           child:Image.asset('images/dice1.png'), // refer the first image of
              dicee.
            onPressed: (){
              //function to change the face
            },
        ),
        ),
        // Implement another child of the row with image asset dice2.png
        ],
      ),
    ),
  );
 }
}
```

7. Define a function called as changeface() which will use set state to change the UI and dice image number dynamically as shown below,

```
class _DicePageState extends State<DicePage> {
  int left = 1 ; // set the first image number to left
  int right = 2 ; // set the second image number to right

  void changeface(){
    setState(() { // marks the code below as dirty
```

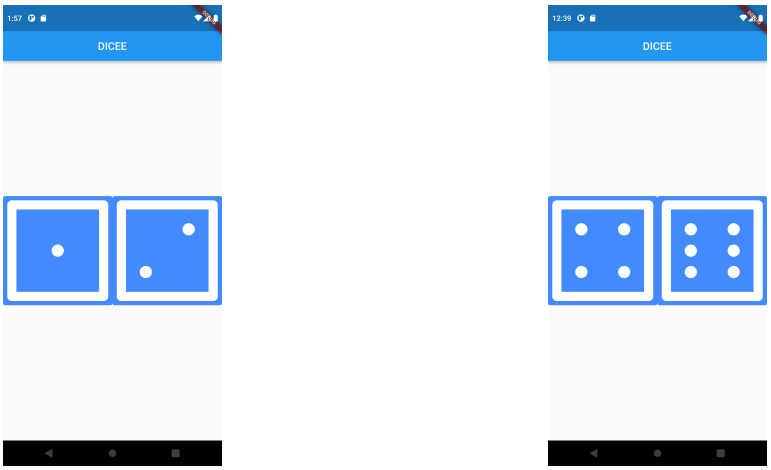
```
    left = Random().nextInt(6)+1; // Random() belongs to Math library
    right = Random().nextInt(6)+1 ; // Random().nextInt(6) - generates random
        number between 1-6, excluding 6.
    // to include 6 as well, add 1 to it.
  });
}
@override
Widget build(BuildContext context) {
..
}
}
```

8. Modify the onPressed() function in the Two text Button as follows,

```
Expanded(
          flex: 1,
          child: TextButton(
            style: TextButton.styleFrom(
              backgroundColor: Colors.blueAccent,
            ),
          child:Image.asset('images/dice$left.png'), // dynamically changes the
              image number
            onPressed: (){
              changeface(); // calls the UI change on button pressed
            },
        ),
        ),
  Expanded(
          flex: 1,
          child: TextButton(
            style: TextButton.styleFrom(
              backgroundColor: Colors.blueAccent,
            ),
            child:Image.asset('images/dice$right.png'), // dynamically changes
                the image number
            onPressed: (){
              changeface(); // calls the UI change on button pressed
            },
          ),
        ),
```

3 Results

The output of the App is shown in figure 3a & 3b.



(a) Left Button displaying dice1.png, Right Button displaying dice2.png

(b) Upon Clicking left/right button: UI updated with random dicee image(1-6)

Figure 3: Output