StatefulWidget's Key & State

Greg Perry Follow
Jan 10 · 16 min read ★

An in-depth look at Flutter's StatefulWidget and its Lifecycle.

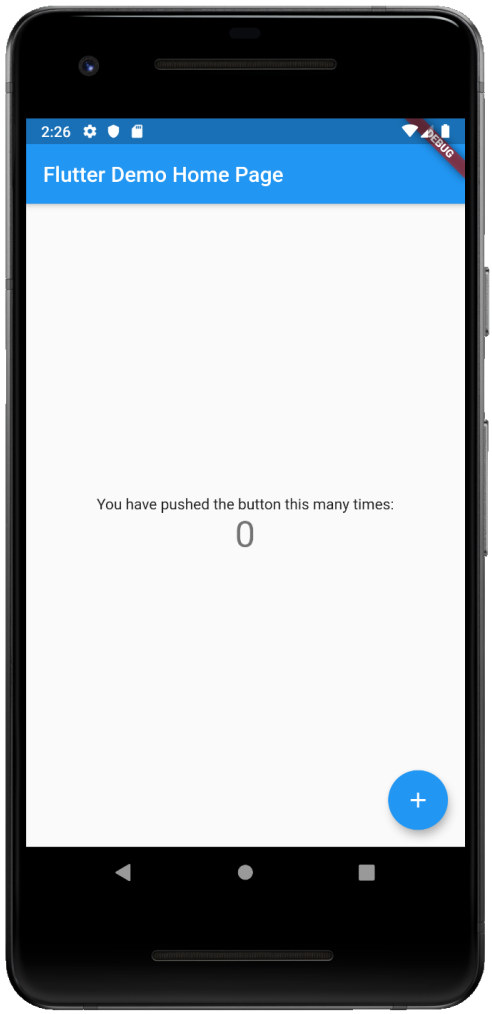
You're going to use StatefulWidget's. If you're going to use Flutter, you're going to use StatefulWidget's. A lot of them. Together. One on top of the other. It's inevitable. As you learn Flutter more and more, your apps will get more complicated… with more Widgets. More StatefulWidgets. A Stateless widget never changes. A Stateful widget can change in response to user interactions or other events. This widget's 'state' is represented by a separate class object called, well… State. The State object consists of values that can change. The State object contains its associated widget's 'mutable state.' — it stores values that can change over time. When those values have changed, more often than not, the associated StatefulWidget is re-created.

Learn By Example

In many of my articles, I try to use established examples from

Google's own Flutter website to convey a concept, a particular Widget, etc. In this article, I'll use the 'example app' generated for you every time you create a 'New Flutter Project...'. Below, I've isolated both the StatefulWidget and its State object found in that generated code. Comments and other code removed for brevity.

```
class MyHomePage extends StatefulWidget {
  MyHomePage({Key key, this.title}) : super(key: key);
  final String title;
  @override
  _MyHomePageState createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {
  int _counter = 0;
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(widget.title),
      ), // AppBar
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            const Text(
              'You have pushed the button this many times:',
            ), // Text
            Text(
              '$_counter',
              style: Theme.of(context).textTheme.display1,
            ), // Text
          ], // <Widget>[]
        ), // Column
      ), // Center
      floatingActionButton: FloatingActionButton(
        onPressed: () {
          setState(() {
            _counter++;
          });
        },
        tooltip: 'Increment',
        child: Icon(Icons.add),
      ), // FloatingActionButton
    ); // Scaffold
  }
}
```



I Like Screenshots. Click For Gists.

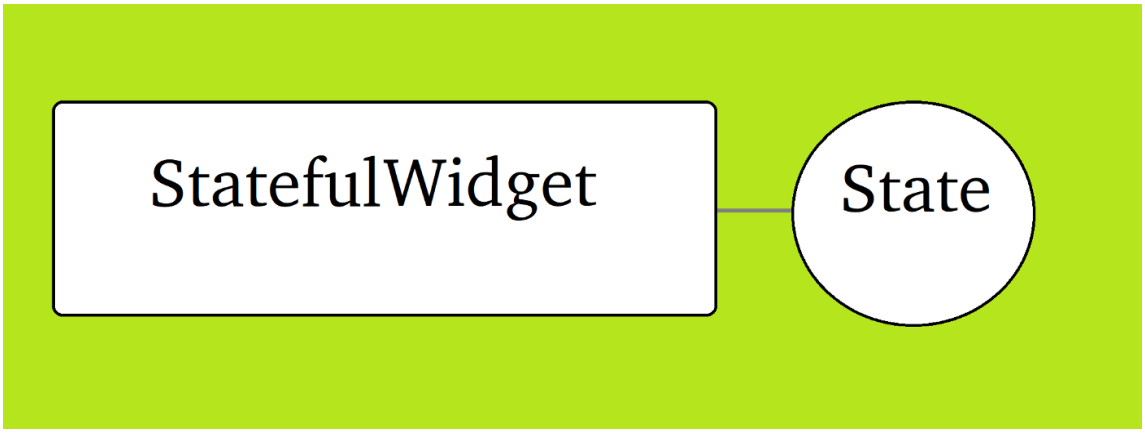
As always, I prefer using screenshots over gists to show concepts rather than just show code in my articles. I find them easier to work with frankly. However, you can click or tap on these screenshots to see the code they represent in a gist or in

Github. Ironically, it's better to read this article about mobile development on your computer than on your phone. Besides, we program on our computers — not on our phones. For now.

No Moving Pictures No Social Media

Note, there will be *gif* files in this article demonstrating aspects of the topic at hand. However, it's said viewing such *gif* files is not possible when reading this article on platforms like Instagram, Facebook, etc. Please, be aware of this and maybe read this article on medium.com

Let's begin.



Build The Screen

When you run that generated code, you get the good ol' counter app. When you see a screen like the one in the screenshot above, you know the **build**() function for the State object, *_MyHomePageSate*, has run and a Scaffold widget was returned. Inside that Scaffold widget, are many other widgets. I

count eight off-hand. They've all run and now are displayed on the screen. Note the red arrow below. You can see that once displayed, one widget is now posed with an event handler to do something if and when it's pressed on.

```
floatingActionButton: FloatingActionButton(
  onPressed: () {
    setState(() {
      _counter++;
    });
  },
  tooltip: 'Increment',
  child: Icon(Icons.add),
), // FloatingActionButton
```

Set To Rebuild

Do you know what a State object's **setState**() function does? It does a lot of things, but one thing in particular that we're interested in is that it causes that State object's **build**() function to also run. As a result, in this case, an anonymous function then runs and increments the State object's instance variable called, *_counter*. The Flutter framework is then notified to call that State object's **build**() function again displaying now the updated value of the instance variable. Every time a State object's **setState**() function is called, its **build**() function will be called soon after.

Count Your Builds

And so, if you press on that floating blue button, you'll see the number 1 on the screen. The State object's **build**() function was called again — displaying the field property or instance

variable, *_counter*, with its new integer value of 1. The screen you see before you is literally 'rebuilt' from the contents in the State object's **build**() function. Understand so far?

```
class MyHomePage extends StatefulWidget {
  MyHomePage({Key key, this.title}) : super(key: key);
  final String title;
  @override
  _MyHomePageState createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {
  int _counter = 0;
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(widget.title),
      ), // AppBar
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            const Text(
              'You have pushed the button this many times:',
            ), // Text
            Text(
              '$_counter',
              style: Theme.of(context).textTheme.display1,
            ), // Text
          ], // <Widget>[]
        ), // Column
      ), // Center
      floatingActionButton: FloatingActionButton(
        onPressed: () {
          setState(() {
            _counter++;
          });
        },
        tooltip: 'Increment',
        child: Icon(Icons.add),
      ), // FloatingActionButton
    ); // Scaffold
  }
}
```

Let's take a closer look at this. There's a number of **print**() functions now introduced to the code. To emphasize when each class is instantiated (i.e. when each widget is created), I've explicitly defined the constructor for both the StatefulWidget object and the State object — supplying a **print**() function in each. There are print commands placed in other locations of interest to demonstrate the 'lifecycle' of the StatefulWidget and its State object. Click on the screenshot below to get your own copy. You can then follow along.

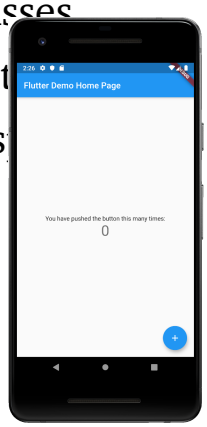
```dart
class MyHomePage extends StatefulWidget {
  MyHomePage({Key key, this.title}) : super(key: key){
    print(">>>>>>>>>>>>>>>>>>>>>>>>>   MyHomePage being created. Constructor cal
  }
  final String title;
  @override
  _MyHomePageState createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {
  // constructor
  _MyHomePageState() : super() {
    print(">>>>>>>>>>>>>>>>>>>>>>>>>   _MyHomePageState being created. Construct
  }
  int _counter = 0;
  @override
  Widget build(BuildContext context) {
    print(">>>>>>>>>>>>>>>>>>>>>>>>>   _MyHomePageState build() called!");
    return Scaffold(
      appBar: AppBar(
        title: Text(widget.title),
      ), // AppBar
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            const Text(
              'You have pushed the button this many times:',
            ), // Text
            Text(
              '$_counter',
              style: Theme.of(context).textTheme.display1,
            ), // Text
          ], // <Widget>[]
        ), // Column
      ), // Center
      floatingActionButton: FloatingActionButton(
        onPressed: () {
          print(">>>>>>>>>>>>>>>>>>>>>>>>>   'Floating Blue' button pressed. Fur
          setState(() {
            _counter++;
          });
        },
        tooltip: 'Increment',
        child: Icon(Icons.add),
      ), // FloatingActionButton
    ); // Scaffold
  }
}
```

counter_app.dart

What Happens On Startup

And so, with those **print**() functions in place, we can readily see what happens when the counter app is first started up. As expected, looking at the console screen below, the two classes

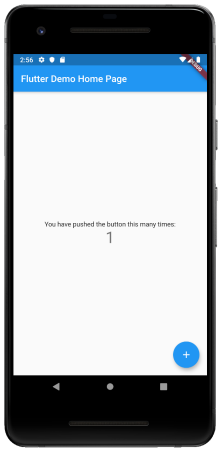
```
Launching lib\main.dart on Android SDK built for x86 64 in debug mode...
Running Gradle task 'assembleDebug'...
√ Built build\app\outputs\apk\debug\app-debug.apk.
Installing build\app\outputs\apk\app.apk...
D/FlutterActivity( 5464): Using the launch theme as normal theme.
D/FlutterActivityAndFragmentDelegate( 5464): Setting up FlutterEngine.
D/FlutterActivityAndFragmentDelegate( 5464): No preferred FlutterEngine was provided. Creating a ne
D/FlutterActivityAndFragmentDelegate( 5464): Attaching FlutterEngine to the Activity that owns this
D/FlutterView( 5464): Attaching to a FlutterEngine: io.flutter.embedding.engine.FlutterEngine@ed7e5
D/FlutterActivityAndFragmentDelegate( 5464): Executing Dart entrypoint: main, and sending initial r
Syncing files to device Android SDK built for x86 64...
D/EGL_emulation( 5464): eglMakeCurrent: 0x7e08c34fd9c0: ver 3 0 (tinfo 0x7e08c1b158e0)
I/flutter ( 5464): >>>>>>>>>>>>>>>>>>>>>>>>>>>   MyHomePage being created. Constructor called.
I/flutter ( 5464): >>>>>>>>>>>>>>>>>>>>>>>>>>>   _MyHomePageState being created. Constructor called.
I/flutter ( 5464): >>>>>>>>>>>>>>>>>>>>>>>>>>>   _MyHomePageState build() called!
```



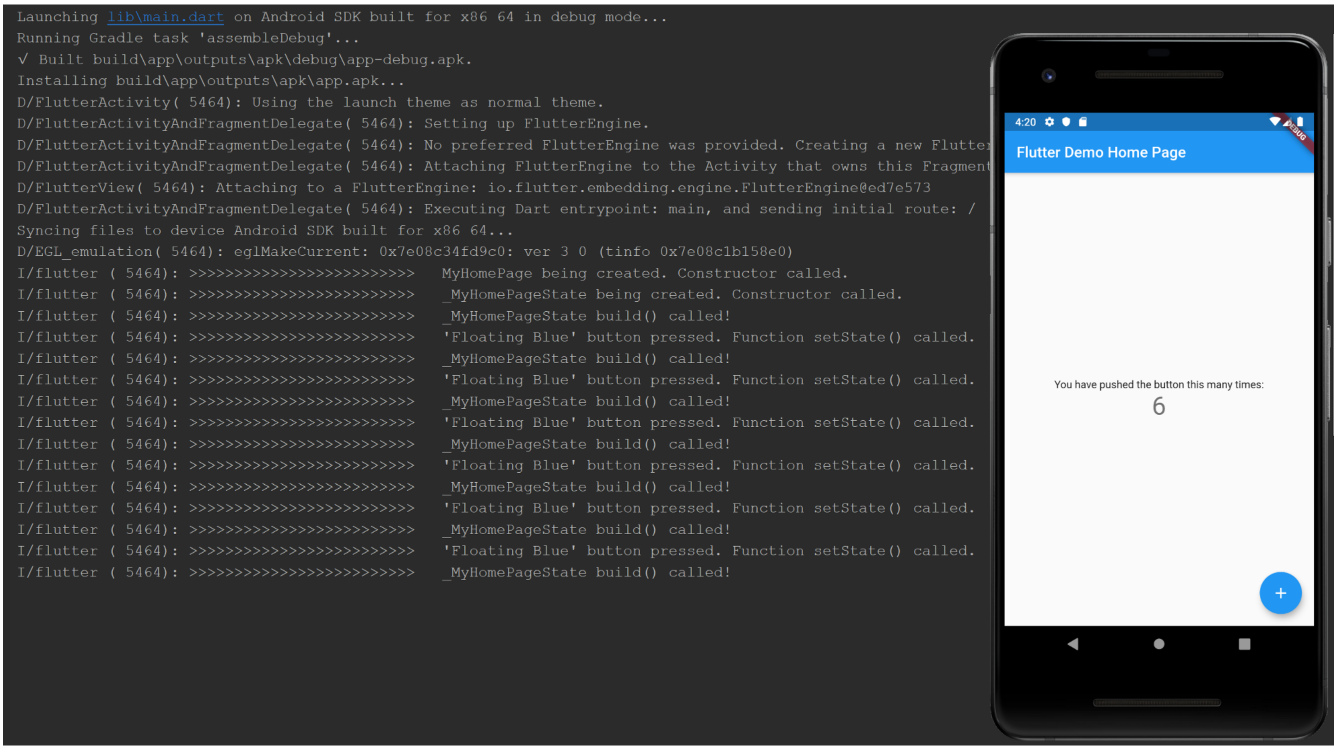
Press The Button. See What Happens.

Press the floating blue button once, and you get what follows on the console screen. Make sense. As you know, calling the State object's **setState**() function will cause the State object's **build**() function to be called again. Looking good.

```
Launching lib\main.dart on Android SDK built for x86 64 in debug mode...
Running Gradle task 'assembleDebug'...
√ Built build\app\outputs\apk\debug\app-debug.apk.
Installing build\app\outputs\apk\app.apk...
D/FlutterActivity( 5464): Using the launch theme as normal theme.
D/FlutterActivityAndFragmentDelegate( 5464): Setting up FlutterEngine.
D/FlutterActivityAndFragmentDelegate( 5464): No preferred FlutterEngine was provided. Creating a ne
D/FlutterActivityAndFragmentDelegate( 5464): Attaching FlutterEngine to the Activity that owns this
D/FlutterView( 5464): Attaching to a FlutterEngine: io.flutter.embedding.engine.FlutterEngine@ed7e5
D/FlutterActivityAndFragmentDelegate( 5464): Executing Dart entrypoint: main, and sending initial r
Syncing files to device Android SDK built for x86 64...
D/EGL_emulation( 5464): eglMakeCurrent: 0x7e08c34fd9c0: ver 3 0 (tinfo 0x7e08c1b158e0)
I/flutter ( 5464): >>>>>>>>>>>>>>>>>>>>>>>>>>>   MyHomePage being created. Constructor called.
I/flutter ( 5464): >>>>>>>>>>>>>>>>>>>>>>>>>>>   _MyHomePageState being created. Constructor called.
I/flutter ( 5464): >>>>>>>>>>>>>>>>>>>>>>>>>>>   _MyHomePageState build() called!
I/flutter ( 5464): >>>>>>>>>>>>>>>>>>>>>>>>>>>   'Floating Blue' button pressed. Function setState()
I/flutter ( 5464): >>>>>>>>>>>>>>>>>>>>>>>>>>>   _MyHomePageState build() called!
```



You pretty much get the idea. Every time you press the button, you're going to see the State object's **build**() function get called again — displaying the value in the State object's instance variable, *_counter*. Below, is a screenshot of the console screen when the button was pressed six times. Pretty simple.



Initialize The State

Before we continue, let's step back a bit and look further at that the State object. Traditionally, a lot of the logic for your app will be found, accessed, and run in your app's State objects. You'll find yourself defining this logic in the class that makes up your State object. Thus, a commonplace to initialize such logic is in the State object's **initState**() function. Let's demonstrate this now by modifying our ol' counter app and initialize, in this

case, its instance variable, *_counter*, with the integer value of 0 inside the State object's **initState**() function.

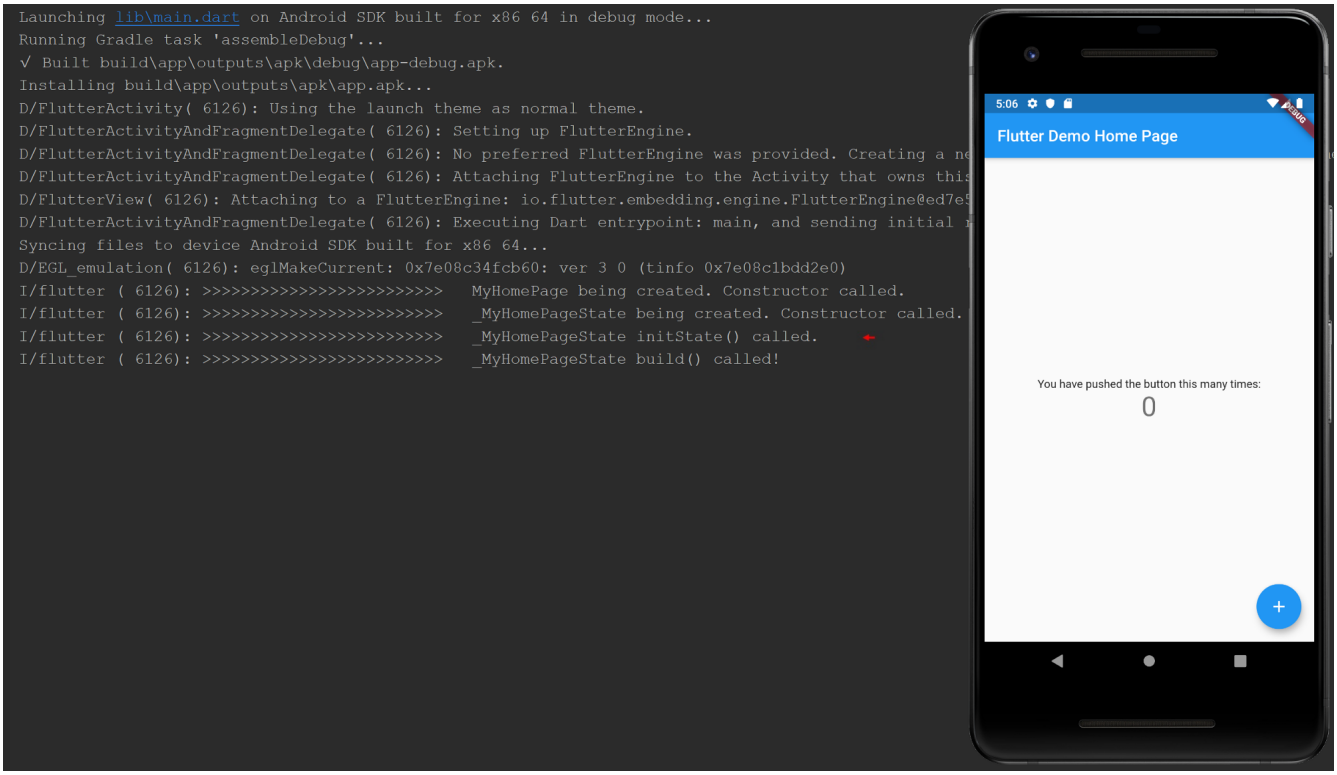
With such a simple example to work with, I nevertheless wish to convey some of the common practices found in more complex applications — one being the supplying of default values to such instance variables in the **initState**() function. This function is called only when a State object is first created, and so the app will behave just as before. Note, a print command will be placed inside the **initState**() function as well.

```
class _MyHomePageState extends State<MyHomePage> {
  // constructor
  _MyHomePageState() : super() {
    print(">>>>>>>>>>>>>>>>>>>>>>>>>>   _MyHomePageState being created. Construct
  }
  int _counter;

  @override
  void initState(){
    super.initState();
    print(">>>>>>>>>>>>>>>>>>>>>>>>>>   _MyHomePageState initState() called.");
    _counter = 0;
  }

  @override
  Widget build(BuildContext context) {
    print(">>>>>>>>>>>>>>>>>>>>>>>>>>   _MyHomePageState build() called!");
    return Scaffold(
      appBar: AppBar(
        title: Text(widget.title),
      ), // AppBar
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            const Text(
              'You have pushed the button this many times:',
            ), // Text
            Text(
              '$ counter',
```

counter_app.dart

It all behaves the same. However, now you see an additional

line in the console screen. Again, you will see this line only once when the State object is first instantiated.



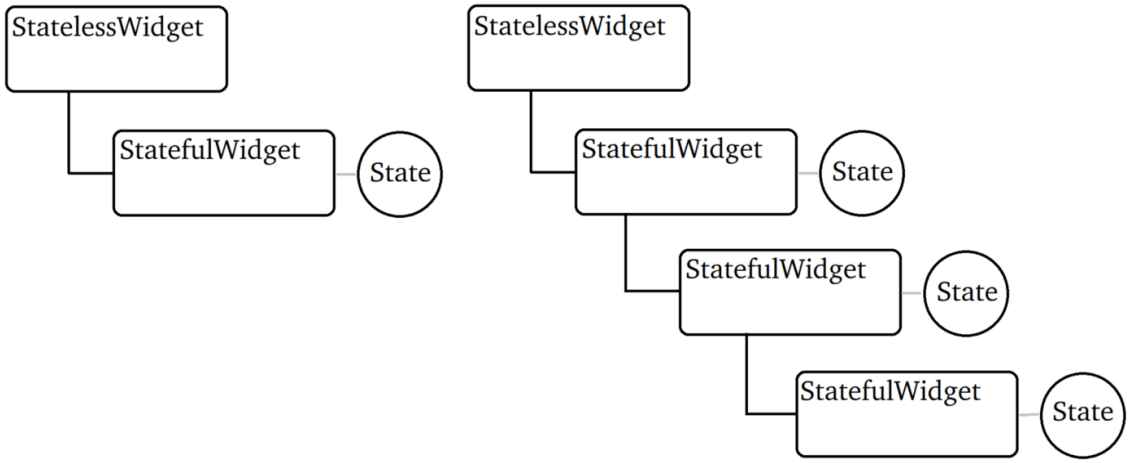
It's Going To Get Complicated

Ok, let's get back to it. So far, this has been really simple. It's a really simple app. However, your apps are not going to be that ridiculously simple. They're going to be more complicated. Let's see if I can introduce such complexity while still using the concept of a simple counter app.

Two Screens; Three Counters

And so I'll now provide you another version of that counter app. In this version, there are three counters across two separate screens! Things have changed in this version — in ways you would see in more complex apps. First and foremost,

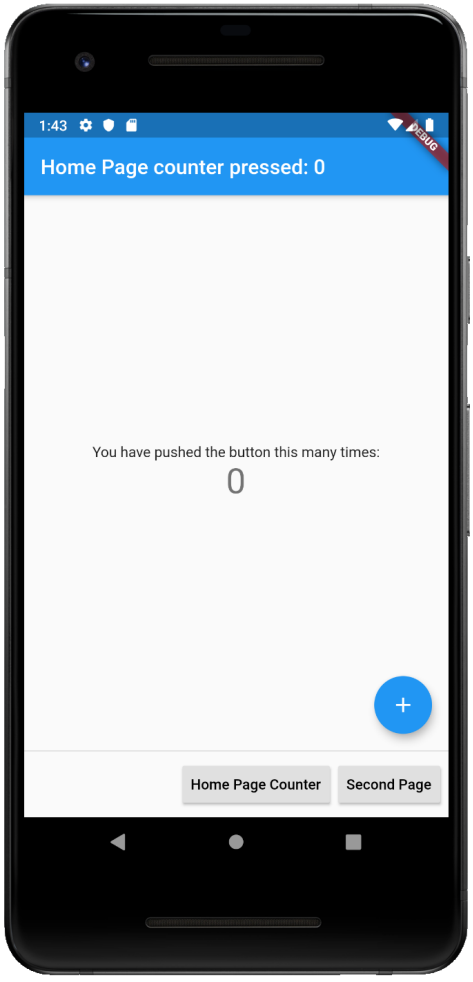
compared to the original counter app, you're going to see StatefulWidgets starting up other StatefulWidgets.



Original version vs Latest version

Below are now screenshots of this more complicated counter app. Tap or click on the screenshots to get a copy for yourself. You can see the original State object, *_MyHomePageState*, now has its own counter?! Additionally, a new StatefulWidget called, *_FirstPage*, now displays the original screen!

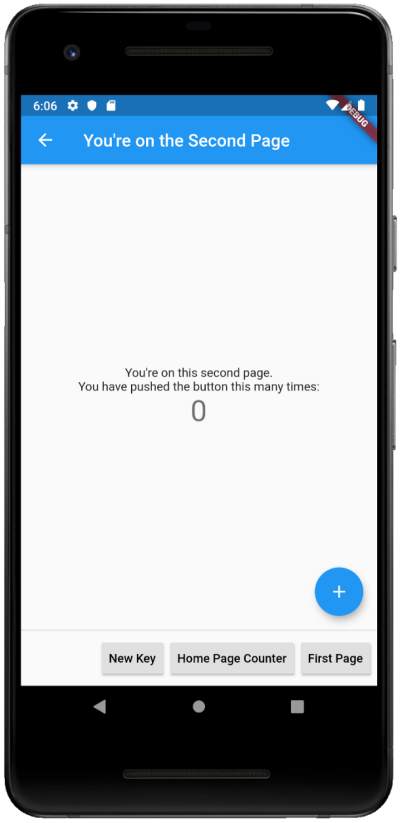
```dart
class _MyHomePageState extends State<MyHomePage> {
  // constructor
  _MyHomePageState() : super() {
    print(
        ">>>>>>>>>>>>>>>>>>>>>>>>>>   _MyHomePageState being created. Constructo
  }
  int _counter = 0;

  void initState() {
    print(">>>>>>>>>>>>>>>>>>>>>>>>>>   _MyHomePageState initState() called.");
    super.initState();
  }

  @override
  Widget build(BuildContext context) {
    print(">>>>>>>>>>>>>>>>>>>>>>>>>>   _MyHomePageState build() called!");
    return _FirstPage(
      key: _key,
      title: "${widget.title} counter pressed: $_counter",
      state: this,
    );
  }
}

class _FirstPage extends StatefulWidget {
  // constructor
  _FirstPage({Key key, @required this.title, @required this.state})
      : super(key: key) {
    print(
        ">>>>>>>>>>>>>>>>>>>>>>>>>>   FirstPage being created. Constructor calle
  }
  final String title;
  final _MyHomePageState state;

  @override
  _FirstPageState createState() {
    return _FirstPageState();
  }
}
```



three_counter_app.dart

Further on in the code, you will find the ‘Second Page’ StatefulWidget that will display a separate counter on a separate screen. Again, it’s peppered with print commands so we can follow the ‘sequence of events’ for such widgets.

```
class _SecondPage extends StatefulWidget {
  _SecondPage({Key key, @required this.title}) : super(key: key) {
    print(
        ">>>>>>>>>>>>>>>>>>>>>>>>>   SecondPage being created. Constructor call
  }

  final String title;

  @override
  _SecondPageState createState() {
    return _SecondPageState();
  }
}

class _SecondPageState extends State<_SecondPage> {
  _SecondPageState() : super() {
    print(
        ">>>>>>>>>>>>>>>>>>>>>>>>>   _SecondPageState being created. Constructo
  }
  int _counter = 0;

  void initState() {
    print(">>>>>>>>>>>>>>>>>>>>>>>>>   _SecondPageState initState() called.");
    super.initState();
  }

  @override
  Widget build(BuildContext context) {
    print(">>>>>>>>>>>>>>>>>>>>>>>>>   _SecondPageState build() called!");
    return Scaffold(
```



three_counter_app.dart

And so, looking at the console screen, you can see the print commands highlight the sequence in which the widget and State objects are created and the sequence in which their functions are called. For example, when the home screen is first displayed, you can see the State objects involved first call their **initState**() functions before finally calling their **build**() functions. When it comes to the home screen, it's the State object, *_FirstPageState*, with widgets in its **build**() function that displays the first screen or first page.

Follow the ‘arrow lines’ on after the other, and you can see the logic involved when displaying the first screen. Each StatelessWidget and StatefulWidget are created one after the other. And with each StatefulWidget, you can see its corresponding State object is created and their functions called.

To The Second

Now, while watching the console screen, if you then tap on the button, *Second Page*, you will see it goes to the third StatefulWidget in the app named, *SecondPage*. It creates that widget; it creates its accompanying State object named, _SecondPageState. It goes through that State object’s **initState**() function and finally to its **build**() function to display the second screen. But as you can see in the console, it’s not done yet.

But Don't Forget The First

The Flutter framework, once it's displayed the second screen, then calls the first screen's State object, *_FirstPageState*, once again! This time to call that State object's **deactivate**() function. Of course, as you can see, we're still not done. Flutter then calls the **build**() function of the first State object's in the app, *_MyHomePageState ,* to actually re-create the StatefulWidget, *FirstPage*, You can see it's constructor is called again. Lastly, there's again a call to the **build**() function in the State object, *_FirstPageState*. Now, why would the Flutter framework do all that?? We'll get to that another time. For now, did you notice some things were missing in that calling sequence?

In Flutter, Widget's are being re-created all the time. Constantly. Every time you cause a **build**() function to run, you're re-creating the Widgets found therein. Get used to that fact, but did you notice what didn't get called in that sequence

of events? That's right! You didn't see 'the first' State object get re-created and or it's **initState**() function get called again! I've put them where they would have gone in the sequence below, but, in fact, they're not called. The State object, *_FirstPageState*, was left alone only to run its **build**() function again. So why weren't they called?

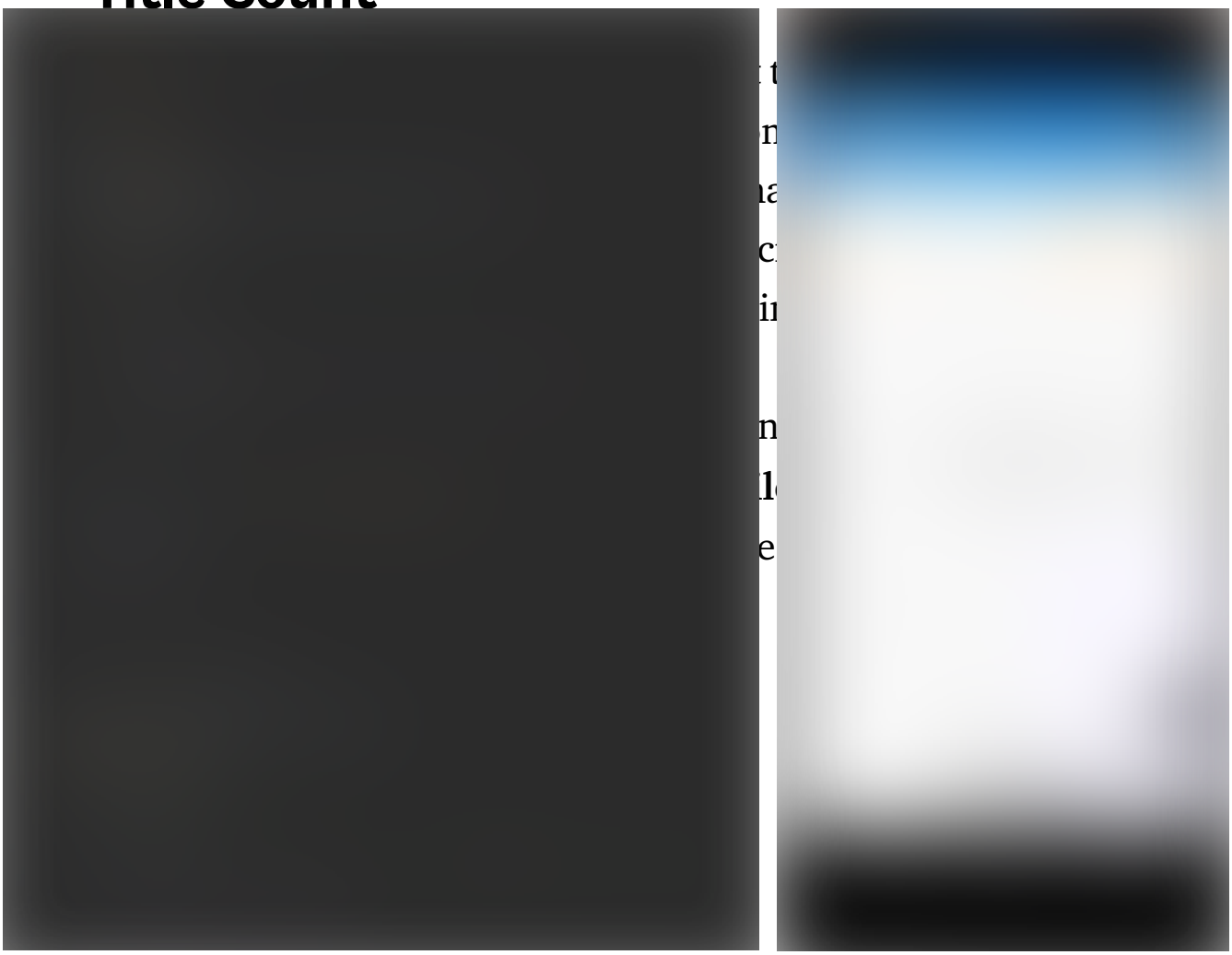
No constructor call or **initState**() call

By the way, the first State object calls its **build**() function and yet is not displayed due to Flutter's routing mechanism. We're on a separate overlay on the stack of routes by this time and so are presented with the 'Second Page' StatefulWidget and the contents of it's State object's **build**() function instead.

'Second Page' Button calls a Navigator

The State Remains

The fact remains that the 'First Page' State object was retained even when its StatefulWidget was destroyed and created again. State objects and, more importantly, the contents inside State objects are preserved while the framework around them are rebuilt again and again. That's what I want you to appreciate at this time. By design, the State object's constructor and **initState**() is not fired again in such a sequence of events. Let's look at another example of this.



three_counter_app.dart

Build Your Home

You can see when pressing the ‘Home Page Counter’ button, the counter defined way back up in the State object, *_MyHomePageSate*, is incremented. Further, that State object’s **setState**() is called, and you know what that means.

And so, back up to the State object, *_MyHomePageSate*, its

three_counter_app.dart

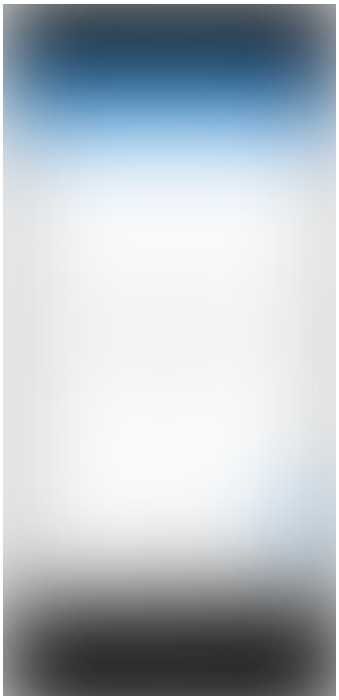
The Console Knows

Again, when the 'Home Page Counter' button is pressed, you can see back in the Home Page, its State object's **build**() function is called (see the screenshot above) and thus updates the title bar for the app. Of course, because the _MyHomePageState **build**() function is run, the StatefulWidget, *FirstPage*, is re-created (its constructor called) and its State object's **build**() function called again in turn.

Again, however, the State object itself is not re-created. It's otherwise left alone. See how this works?

Wait A Second!

Let's try the counter on the second screen. You can see the sequence of events that got us to the screenshot we see below. The app was started up, and the button labelled, 'Second Page', was pressed on. The second page was brought up on the screen (via the Navigator class), and its floating blue button was pressed once. Finally, you can see, the State object, *_SecondPageState*, has its **build**() function called to then 'redraws' the screen to display the number 1.



Looks pretty straightforward, right? With every push of the button, the State object, *_SecondPageState*, has it's **setState**() and **build**() functions called in turn.

Back To First

Well now, let's return to the first screen. Since we've used the Navigator class to open up and present the second screen, we can use the Navigator again to retreat back the 'route stack of widgets' and return to the first screen using the command, *"Navigator.pop(context);*". Now, look at the console screen below.

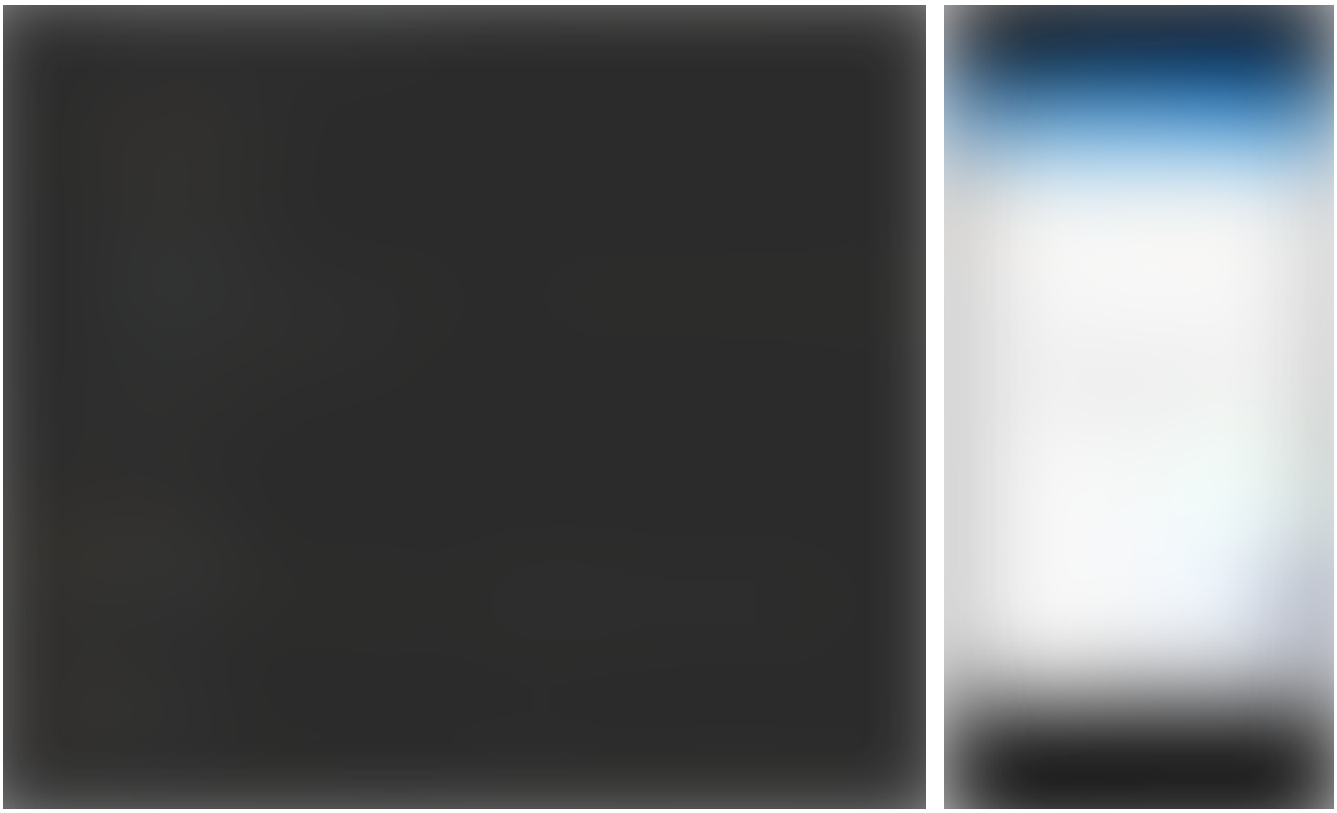
The First One Goes

Look at what happens when the app returns to the first screen! The app actually re-creates the 'First Page' StatefulWidget all over again. First, it calls the **deactivate**() function from the First Page's State object again, the _HomePageState **build**() is then called (so to build the 'First Page' StatefulWidget again). You can see that it is because its constructor then fires. Finally, that StatefulWidget's State object (left otherwise untouched) fires its **build**() function again. Whoa! A lot going on there, uh? But that's not all!

Pop Goes The State

With the 'Second Page' widget removed from the routing stack, it's cleared from memory — and so is its State object. You can see the 'Second Page' State object's **deactivate**() function is

called (see below), as well as another function. One that may be new to you. The **dispose**() function. That's telling you the State object has been released from memory and, like its StatefulWidget counterpart, is a candidate now for the platform's garbage collection. Note, any 'heavy resources' like open files and or open Streams should be explicitly closed in the **deactivate**() function or in the **dispose**() function. (Some say it's best to use the **deactivate**() vs the **dispose**()).



three_counter_app.dart

And so, if you return to the Second Page, you'll find the count set back to zero. The StatefulWidget and its accompanying State object have both been created again when you return to the Second Page through the Navigator class.

Count On Home

We can go on with this stuff. What happens, for example, if the Home Page counter on the title bar back on the first screen is incremented while we're on the Second Page? Well, let's see. Below is the sequence when the 'Home Page Counter' on the Second Page is pressed three times in a row. We then return to the first screen. We can see, with every press of that button, the first screen's StatefulWidget is being re-created. Of course, that's because the Home Page's State object is running its **build**() function (to update the title bar). As a result, the first screen's StatefulWidget is re-created and its associating State object's **build**() function is also called. Finally, when we do return to the first screen, we see the title bar is indeed

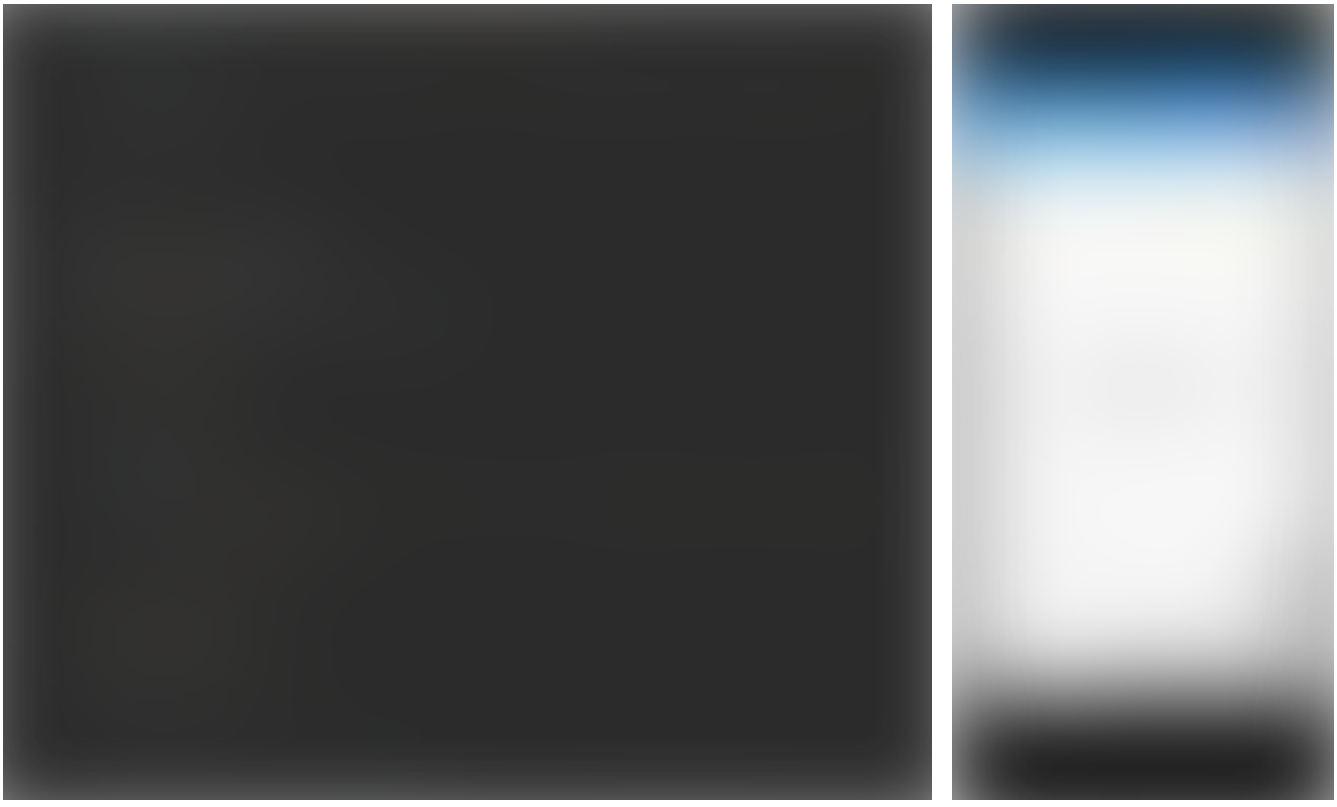
The Key State

Have you noted that additional button on the second screen? That 'New Key' button? Do you know what that does? Well, I'll

tell you. It re-creates the first screen's State object! The first screen; not the second screen. I just got finished telling you the first screen's State object is left alone, and now I'm telling you this. Can you believe it?

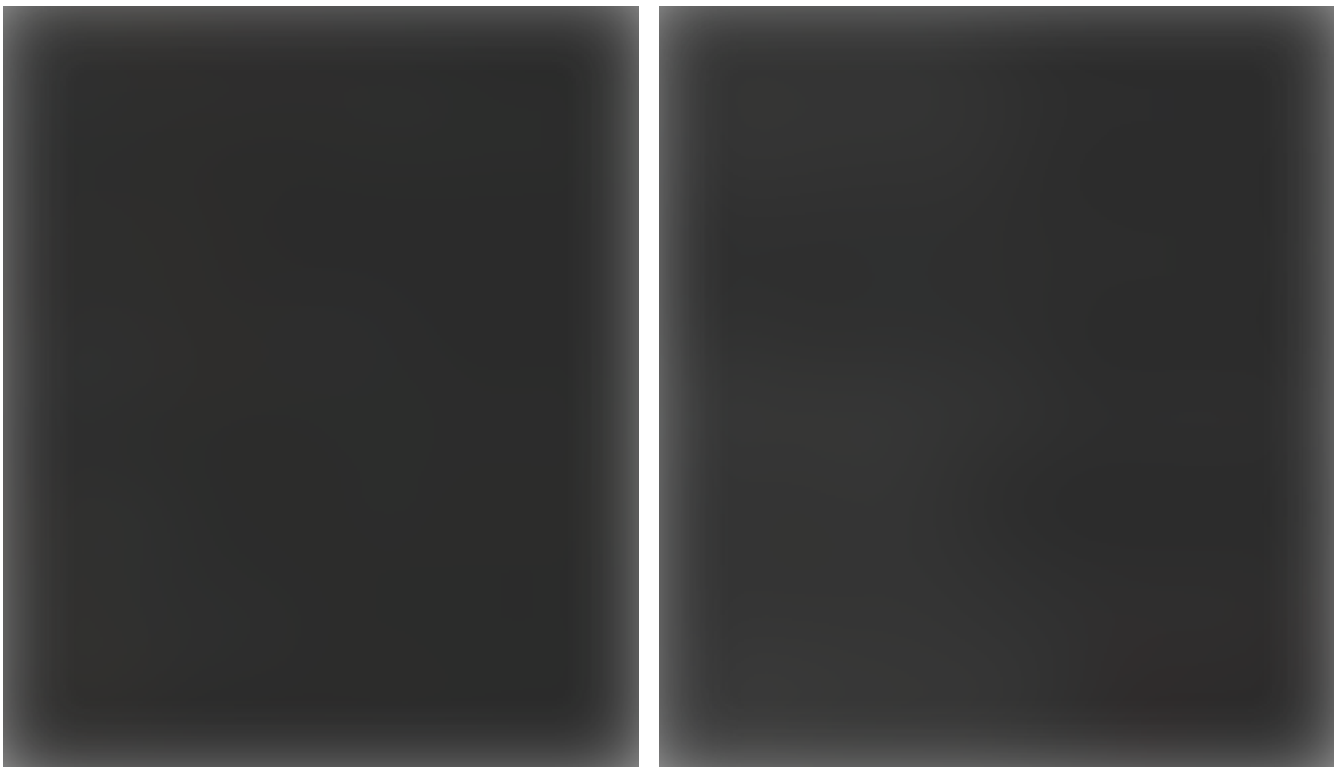
Now, why would you want 'to clear' the State object of a screen? Because, and I can say this from experience, you may want to reset the 'mutable values' in a State object for one reason or another — and Flutter easily allows for this.

When you press the 'New Key' button, you can see below in the code that what happens is that a private variable is assigned a new unique Key value. Doing so has the effect we're looking for. Returning to the first screen, the counter is returned to zero. That's because the State object has been re-created.



three_counter_app.dart

That variable is a high-level variable — defined within the dart file and not in any particular class. Now there's no special reason to do this. In fact, it could have just as easily been defined as an instance variable in the State object, _MyHomePageState. Regardless, note it's defined in the file at compile-time and then assigned to the StatefulWidget, *_FirstPage*. Below, you can see the sequence that occurs after the 'New Key' button is pressed.



three_counter_app.dart

You can see in the screenshot above when you click on the 'New Key' button, the first screen's StatefulWidget, *FirstPage*, gets re-created as usual. However, something's changed. It's State object _FirstPageState is re-created as well. Now the sequence highlighted with red arrows is a little confusing, but we'll walk through it. Starting with the first red arrow, we see the 'original instance' of the State object calling its **deactivate**() function.

The next three arrows depict the ‘new instance’ of the State object being created and calling its **build**() function. The last arrow points back to the ‘original instance’ of the State object as it calls its **dispose**() function and prepares itself to be recycled.

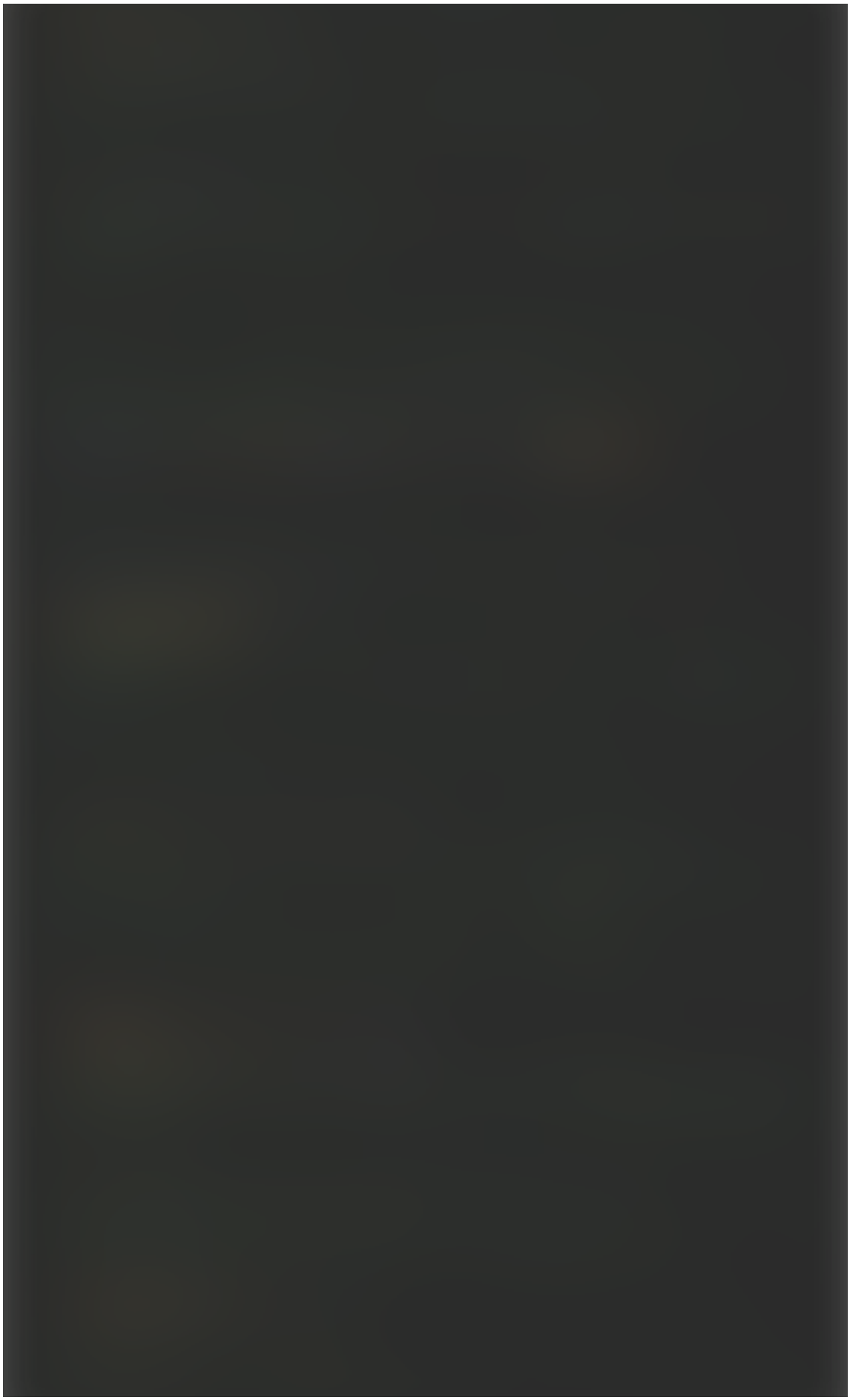
Note, that **dispose**() function could appear any time after the **deactivate**() function call — right after or much much later. That’s why people suggest you clear up ‘time constraint’ resources in the **deactivate**() function as it’s consistently and predictably called when the State object’s replaced by a new instance.

And so, when the ‘First Page’ button is pressed to return to the first screen, we know the first screen’s StatefulWidget is always re-created. However, this time, it’s assigned a new ‘unique key.’ Doing so has the framework dispose the original State object so as to create a new one. So there you go. Besides testing widgets, there’s another reason to assign keys to widgets. The Flutter team, of course, has a video on when to use Keys and would likely supply more insight.

When to Use Keys

Get A Lifecycle

Please note the free article, Flutter App Lifecycle, is an excellent read. It introduces the class, *WidgetsBindingObserver*, that you would use to implement further 'lifecycle' event handlers. Those of you with an Android background would recognize these event handlers as those found in the Activity class. Below is a screenshot of some of those callback event handlers then available to you.



Mobile App Development Android App Development iOS App Development

Flutter Programming many StatefulWidgets — and their States.

Cheers.