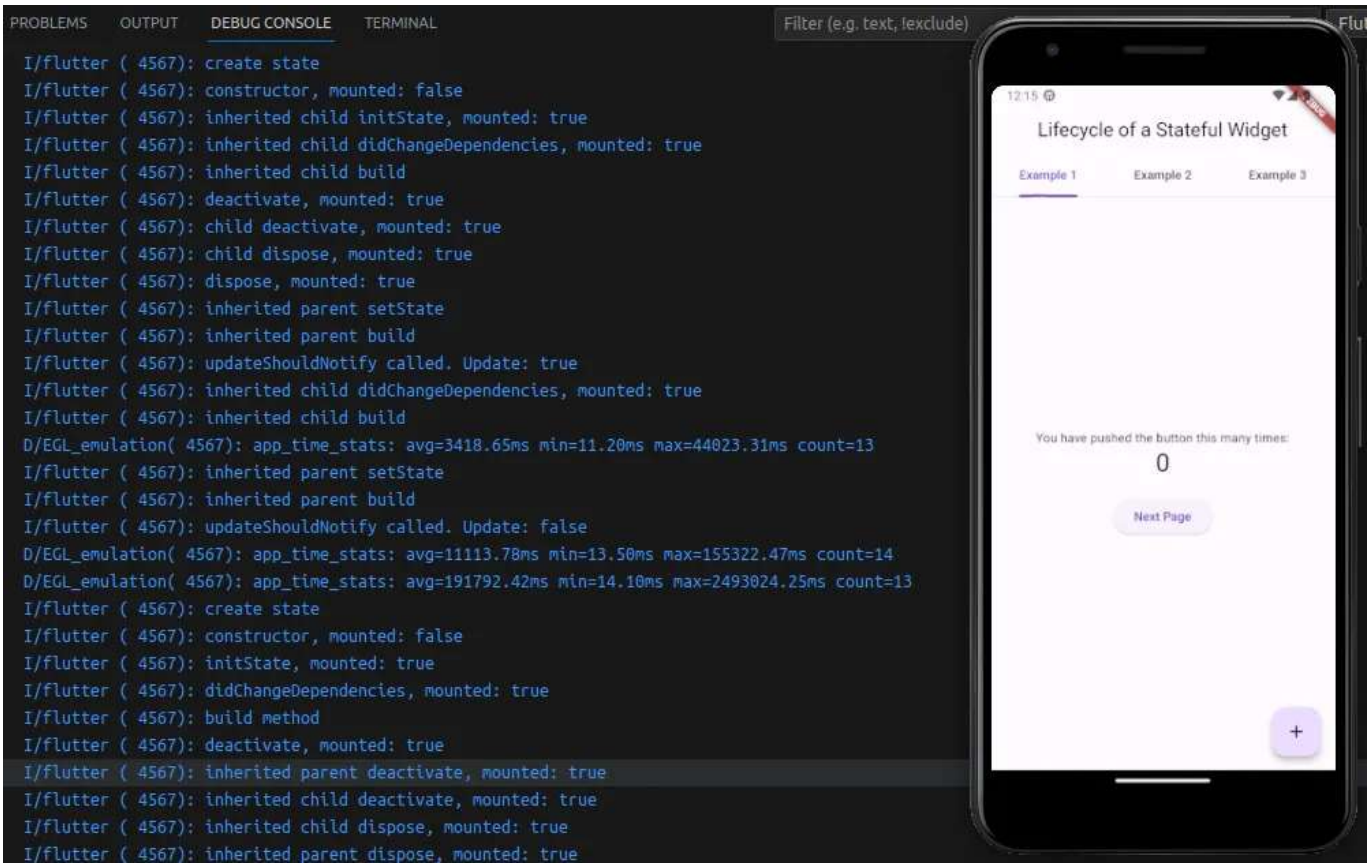
Lifecycle of a Stateful Widget

Hadiyaaamir · Follow
13 min read · Jun 20, 2023

A stateful widget in Flutter is a component that can maintain state and update its appearance in response to changes. The lifecycle of a stateful widget consists of a series of methods that are invoked at different stages of its existence.

In this article, we'll explore stateful widgets in Flutter, diving into their



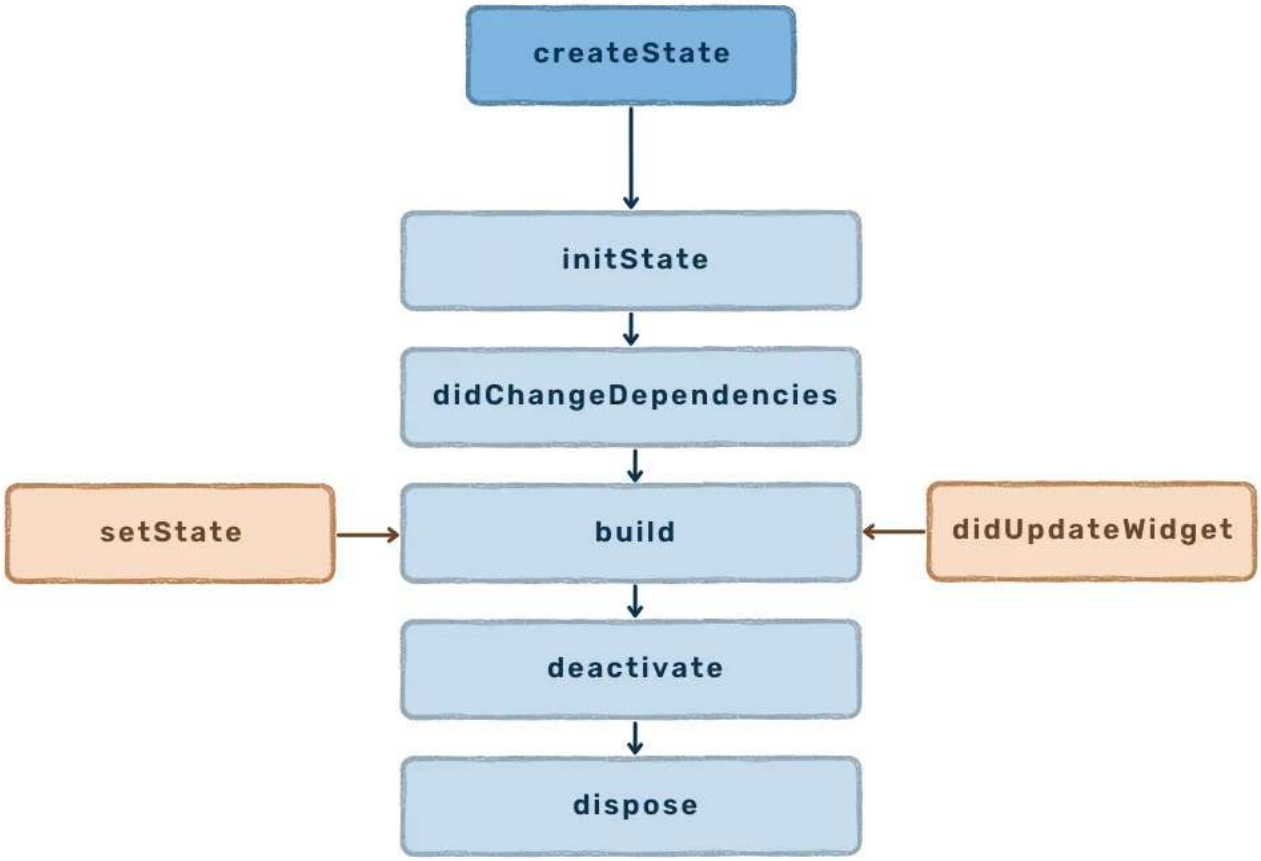
Overview

First, let's have an overview of the stages in the lifecycle of a stateful widget, and which method is invoked at which stage of the lifecycle.

1. **createState():** Upon creation of the stateful widget, its constructor is called, which initializes the widget. The createState() method is then invoked, which creates the state object for the widget.
2. **initState:** This method is called after the widget is inserted into the widget tree, when the object is created for the first time.
3. **didChangeDependencies:** This method is called when a dependency of this widget changes. It is useful when the widget depends on some

external data or inherits data from its parent widget. It is also called immediately after initState().

4. **build:** Builds the widget's user interface based on its current state. This method is called initially when the widget is first inserted into the widget tree, and may be called repeatedly during the lifecycle whenever the widget needs to be rebuilt.
5. **setState:** Triggers a rebuild of the widget when the state changes and the associated methods are called again.
6. **didUpdateWidget:** Triggered when the widget is rebuilt with updated properties. It is called after build() and allows you to compare the previous and current widget properties.
7. **deactivate:** Called when the widget is removed from the tree but might be inserted again.
8. **dispose:** Called when the widget is removed from the widget tree permanently, allowing you to release resources held by the widget.
9. **reassemble:** Called when the application is reassembled during hot reload.



Demonstration Using a Coding Example

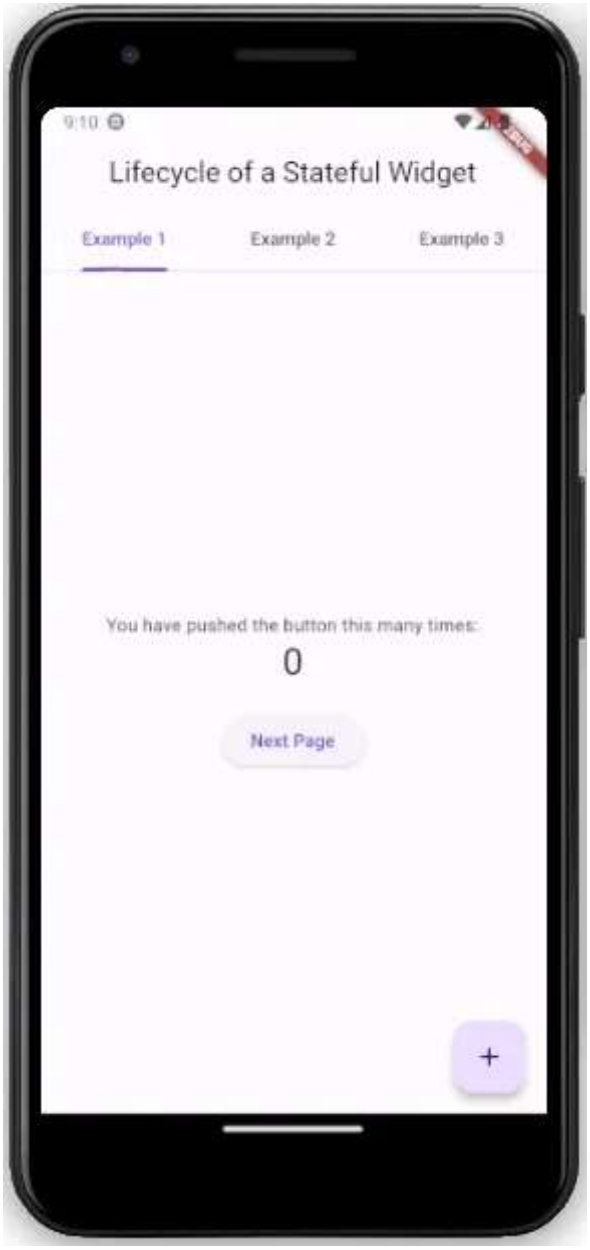
To understand the widget lifecycle better, I created a few simple example codes consisting of stateful widgets. The relevant methods have been overridden with print statements added to help better visualise which method is being called at which stage.

The complete code can be found at this GitHub repo.

Example 1: Simple Implementation Using a Single Widget

First of all, let's start with a very basic example, involving only a single Stateful Widget.

This example utilises the code of the default Flutter counter application; a Scaffold consisting of a Column and a Floating Action Button.



The Column contains a constant Text widget, a number signifying the amount of times the counter has been pressed, and additionally, an Elevated

Button that allows the user to navigate to another page. The Floating Action Button with a + sign increments the counter value.

Below is the code for the Stateful Widget used in this example:

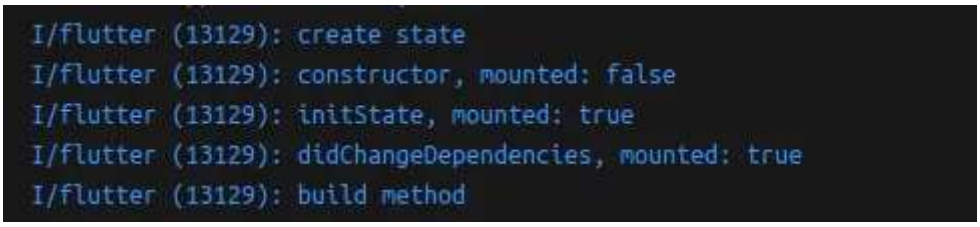
```
import 'package:flutter/material.dart';
import 'package:stateful_lifecycle/new_screen.dart';

class Example1 extends StatefulWidget {
  const Example1({super.key});

  @override
  State<Example1> createState() {
    print('create state');
    return _Example1State();
  }
}

class _Example1State extends State<Example1> {
  int _counter = 0;

  _Example1State() {
    print('constructor, mounted: $mounted');
  }

  @override
  void initState() {
    super.initState();
    print('initState, mounted: $mounted');
  }

  @override
  void didChangeDependencies() {
    super.didChangeDependencies();
    print('didChangeDependencies, mounted: $mounted');
  }

  @override
  void setState(VoidCallback fn) {
    print('setState');
    super.setState(fn);
  }

  void _incrementCounter() {
```

```
    setState(() {
      _counter++;
    });
  }

  @override
  Widget build(BuildContext context) {
    print('build method');

    return Scaffold(
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            const Text('You have pushed the button this many times:'),

            // counter text
            Text(
              '$_counter',
              style: Theme.of(context).textTheme.headlineMedium,
            ),

            // navigation button
            Padding(
              padding: const EdgeInsets.only(top: 20),
              child: ElevatedButton(
                onPressed: () => Navigator.of(context).pushReplacement(
                  MaterialPageRoute(builder: (context) => const NewScreen()),
                ),
                child: const Text('Next Page'),
              ),
            ),
          ],
        ),
      ),

      // increment floating action button
      floatingActionButton: FloatingActionButton(
        onPressed: _incrementCounter,
        tooltip: 'Increment',
        child: const Icon(Icons.add),
      ),
    );
  }

  @override
  void didUpdateWidget(covariant Example1 oldWidget) {
    super.didUpdateWidget(oldWidget);
    print('didUpdateWidget, mounted: $mounted');
  }
```

```
  @override
  void deactivate() {
    super.deactivate();
    print('deactivate, mounted: $mounted');
  }

  @override
  void dispose() {
    super.dispose();
    print('dispose, mounted: $mounted');
  }

  @override
  void reassemble() {
    super.reassemble();
    print('reassemble, mounted: $mounted');
  }
}
```

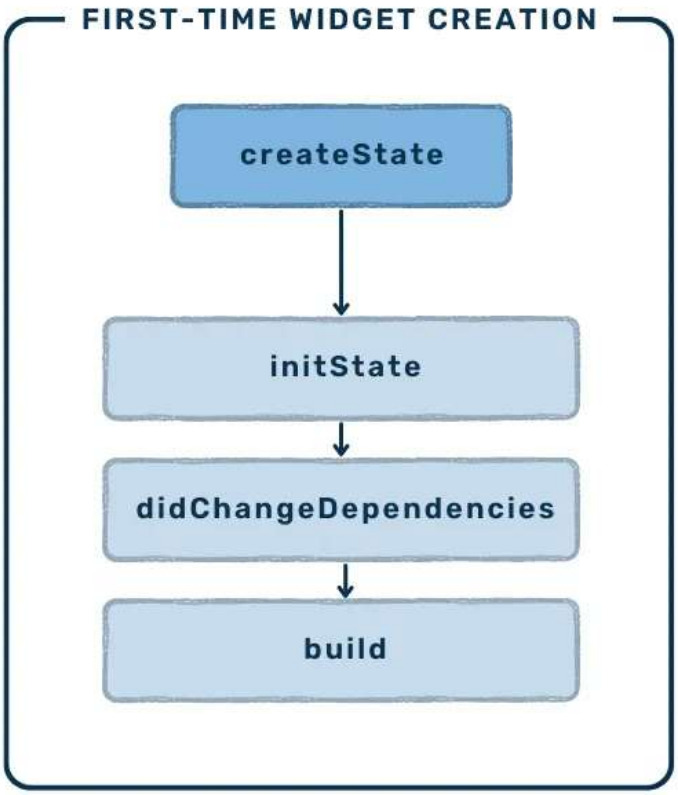
App is Started

When the app is started for the first time, these are the methods that are called.



First, as expected, the CreateState method is called as the widget is initialised. Then, the constructor is called. At this point, the “mounted” property is false, which means that this widget has not yet been inserted in the widget tree.

The object is then created as initState is called, and inserted into the widget tree, as can be seen through the true mounted value. As discussed earlier, didChangedDependencies is called immediately after initState, which can also be seen here. Then, the build method is called, and the Widget is built.

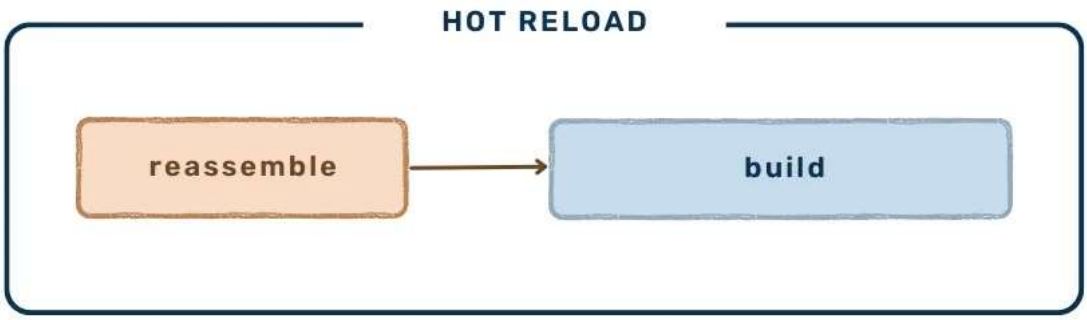


Hot reload

When the hot reload command is called, the reassemble() method is invoked. Hot reload is a mechanism that allows you to see changes made in your running code without having to restart the application.

On hot reload, the widget is rebuilt with the updated values.

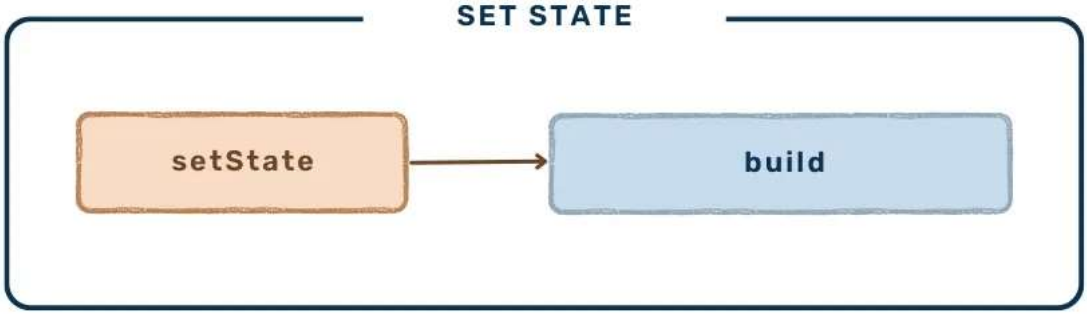
```
I/flutter (13129): reassemble, mounted: true
I/flutter (13129): build method
```



Increment (value update)

Clicking the Floating Action Button increments the counter value, which causes the state of the Widget to change. The setState method is called to trigger a rebuild, and then the widget is rebuilt to reflect this update.

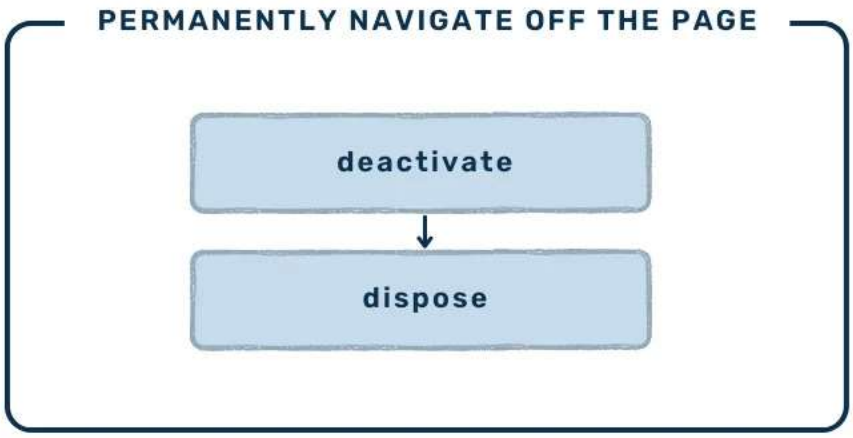
```
I/flutter (13129): setState
I/flutter (13129): build method
```



Next Page — PushReplacement

The next page button pushes a new page as a replacement. This invokes the deactivate() and dispose() methods.

```
I/flutter (13129): deactivate, mounted: true
I/flutter (13129): dispose, mounted: true
```



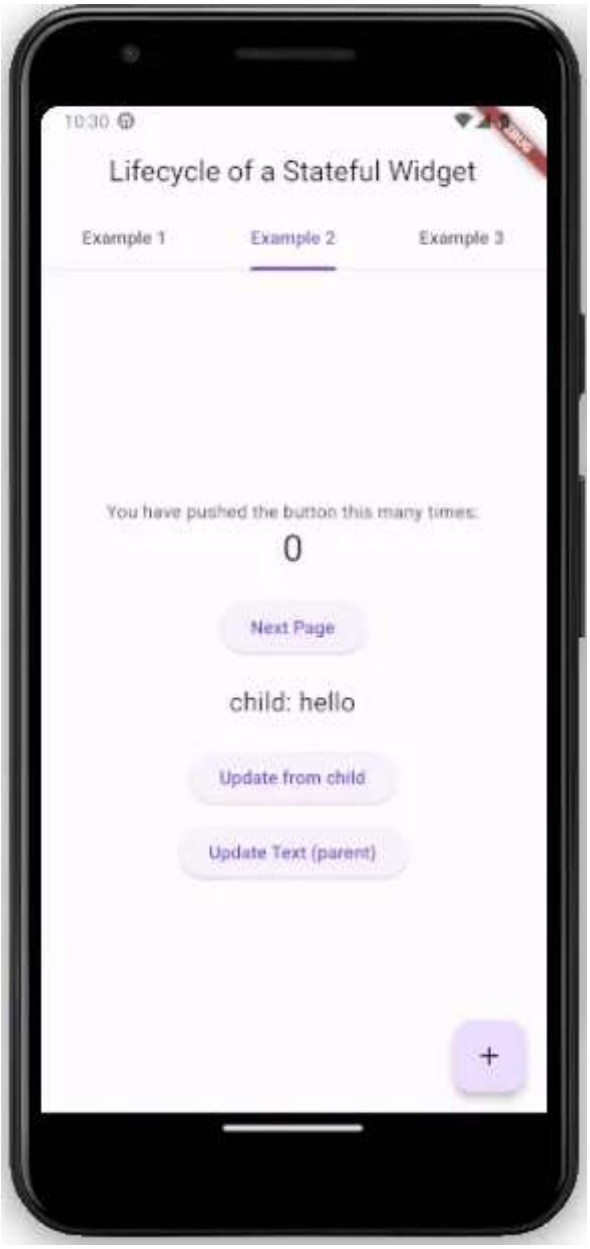
If we were to simply navigate to the next page, these methods would not be called, since the Widget would not have been removed from the widget tree.

Example 2: Child Widget

Next, we wish to see how a Child Widget, if placed inside our current Widget, would act based on changes made to its Parent.

I have added a simple Child Widget to our old example. This Widget takes a text as an input, and displays this text, along with an Elevated Button in the UI. This new button sets the state of the child from within the child widget itself. As done previously, all the Stateful Widget methods have been overriden, and print statements added for better visibility.

Additionally, an Elevated Button has been added to our Column, which changes the value of a variable inside the Parent Widget that may be passed onto the child.



Here is the code for the Child Widget:

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

class Child extends StatefulWidget {
  const Child({
    super.key,
    required this.text,
  });

  final String text;
  @override
```

```
  State<Child> createState() {
    print('create child state');
    return _ChildState();
  }
}

class _ChildState extends State<Child> {
  @override
  void initState() {
    super.initState();
    print('child initState, mounted: $mounted');
  }

  @override
  void didChangeDependencies() {
    super.didChangeDependencies();
    print('child didChangeDependencies, mounted: $mounted');
  }

  @override
  void setState(VoidCallback fn) {
    print('child setState');
    super.setState(fn);
  }

  @override
  Widget build(BuildContext context) {
    print('child build method');
    return Column(
      children: <Widget>[
        // text data displayed
        Padding(
          padding: const EdgeInsets.all(20),
          child: Text(
            'child: ${widget.text}',
            style: const TextStyle(fontSize: 20),
          ),
        ),
        // Update button
        ElevatedButton(
            onPressed: () {
              setState(() {});
            },
            child: const Text('Update from child')),
      ],
    );
  }

  @override
  void didUpdateWidget(covariant Child oldWidget) {
```

```
    super.didUpdateWidget(oldWidget);
    print('child didUpdateWidget, mounted: $mounted');
  }

  @override
  void deactivate() {
    super.deactivate();
    print('child deactivate, mounted: $mounted');
  }

  @override
  void dispose() {
    super.dispose();
    print('child dispose, mounted: $mounted');
  }

  @override
  void reassemble() {
    super.reassemble();
    print('child reassemble, mounted: $mounted');
  }
}
```

Here is the updated build method of the Parent Widget:

```
  @override
  Widget build(BuildContext context) {
    print('build method');

    return Scaffold(
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            const Text(
              'You have pushed the button this many times:',
            ),
            Text(
              '$_counter',
              style: Theme.of(context).textTheme.headlineMedium,
            ),
            Padding(
              padding: const EdgeInsets.only(top: 20),
```

```
            child: ElevatedButton(
              onPressed: () => Navigator.of(context).pushReplacement(
                MaterialPageRoute(builder: (context) => const NewScreen()),
              ),
              child: const Text('Next Page'),
            ),
          ),

          // --- addition ---

          Child(text: text),
          // const Child(text: 'constant text'), //constant child

          // Button to update text value
          const SizedBox(height: 10),
          ElevatedButton(
            onPressed: () {
              setState(() => text += 'hello!');
            },
            child: const Text('Update Text (parent)'),
          ),

          // --- addition ---

        ],
      ),
    ),
    floatingActionButton: FloatingActionButton(
      onPressed: _incrementCounter,
      tooltip: 'Increment',
      child: const Icon(Icons.add),
    ),
  );
}
```

We wish to observe the following:

- What happens when the value of the Parent Widget is updated?
- What happens when the value of the Parent, which is passed onto the child is updated?

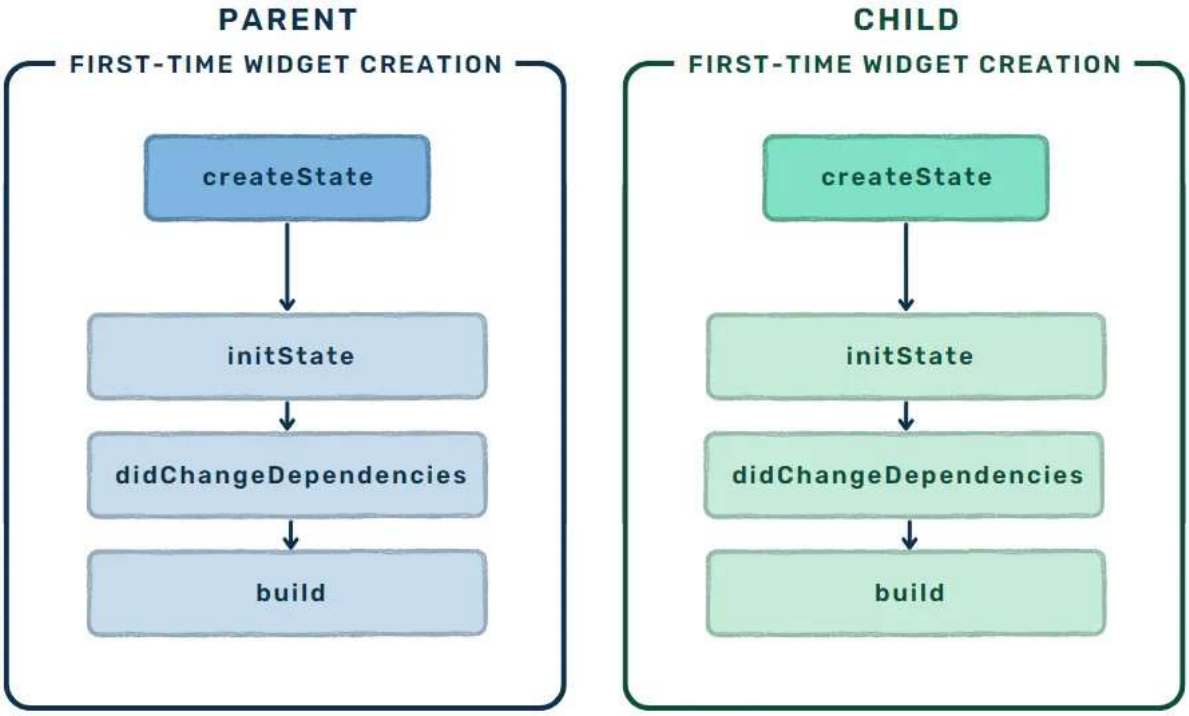
- What happens when the child's state is set from within the child? Is the parent affected?
- What about if the child is constant, and takes in a fixed value? What happens to it then when the parent's state is updated?

App is started

When the app is first started, the following things happen, first for the parent Widget, and then for its child:

- The widget is initialised and the CreateState method is invoked.
- The constructor is called.
- The object is then created as initState is called, and inserted into the widget tree.
- didChangedDependencies is called immediately after initState.
- The build method is called, and the Widget is built.

```
I/flutter (13129): create state
I/flutter (13129): constructor, mounted: false
I/flutter (13129): initState, mounted: true
I/flutter (13129): didChangeDependencies, mounted: true
I/flutter (13129): build method
I/flutter (13129): child initState, mounted: true
I/flutter (13129): child didChangeDependencies, mounted: true
I/flutter (13129): child build method
```

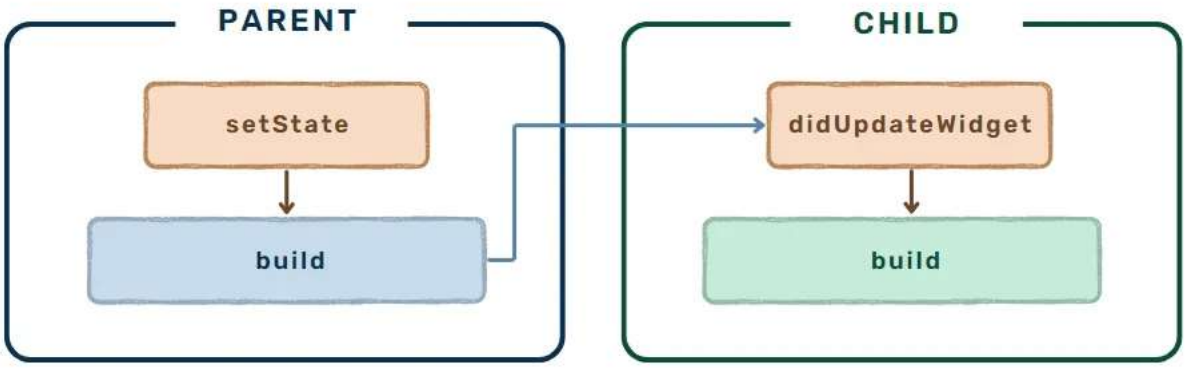


Updating Value of Parent, Which Is Passed Onto the Child

Our example is designed in such a way that our child widget takes as input the String variable “text” from the Parent Widget. Let’s change the value of the text variable by pressing the Elevated Button, and see what methods of the Stateful Widget are invoked for both the parent and the child.

```
I/flutter (13129): setState
I/flutter (13129): build method
I/flutter (13129): child didUpdateWidget, mounted: true
I/flutter (13129): child build method
```

When the button is pressed, we can see through the print statements that the parent Widget’s setState is called, and a rebuild is triggered. Then, the didUpdateWidget method in the child is invoked to check for updates, and the child widget is also rebuilt.

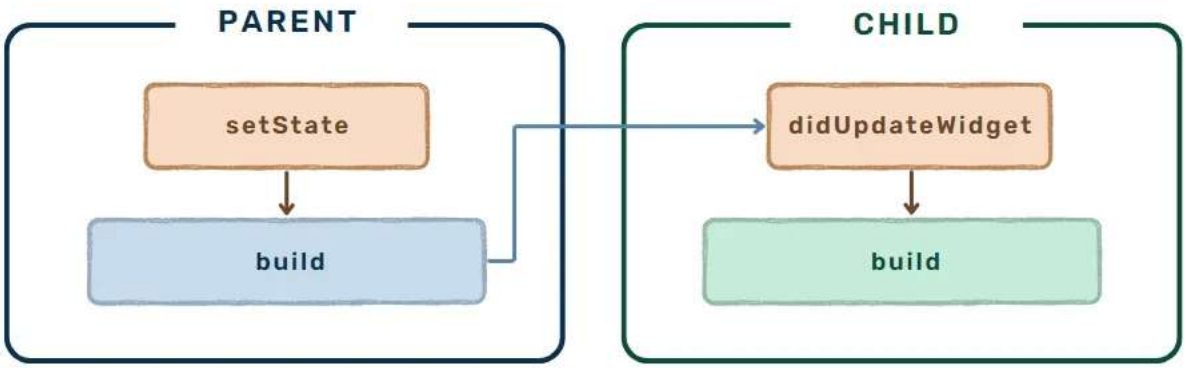


Updating Value Inside Parent

We will now update a value inside the parent that has no direct relation with the child, and see how the child Widget is affected. To do this, we can simply increment the counter value by pressing the Floating Action Button.

When the button is pressed, the same thing occurs as in the previous step; the parent's setState method triggers a rebuild in the Parent Widget, and the child's didUpdateWidget method triggers a rebuild in the Child Widget.

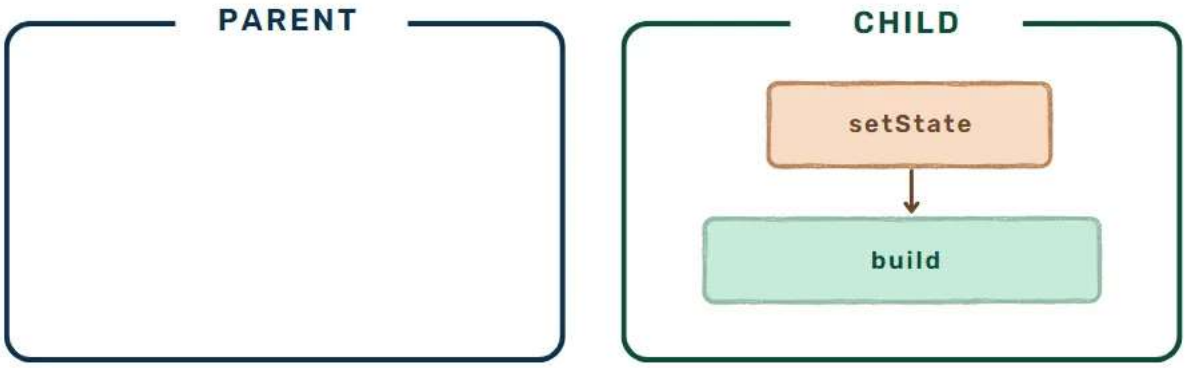
```
I/flutter (13129): setState
I/flutter (13129): build method
I/flutter (13129): child didUpdateWidget, mounted: true
I/flutter (13129): child build method
```



Update Value of Child from Inside the Child Itself

What if a value that is inside the child were to be updated from inside the child itself? We can do this by pressing the “Update from Child” Button. In this case, we can see that the child Widget’s setState is called, and only the child is rebuilt. The parent is not affected in any way.

```
I/flutter (13129): child setState
I/flutter (13129): child build method
```



What if the Child was Constant?

We saw earlier that our Child Widget was rebuilt whenever the parent was rebuilt.

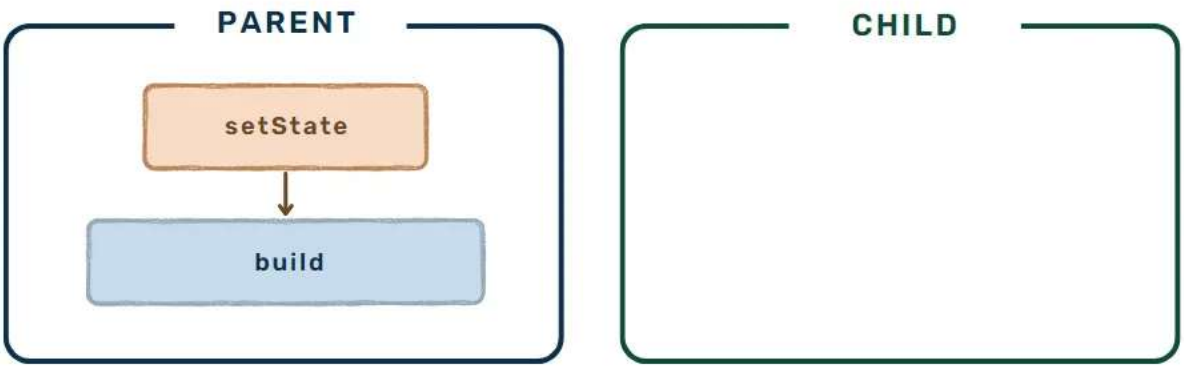
However, let us change the scenario slightly: what if we used a Constant Child Widget instead of passing a variable value into it? What would happen when we update the value of the Parent Widget in this case?

We can comment in the constant child in our previous code:

```dart
@override
  Widget build(BuildContext context) {
    print('build method');

    return Scaffold(
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            // ... same as previous code

            // Child(text: text),
            const Child(text: 'constant text'), //constant

            // ... same as previous code
          ],
        ),
      ),
     // ... same as previous code
    );
  }
```

Now, if we Increment the counter value, we can see that only the Parent is rebuilt. The Child Widget remains unaffected.

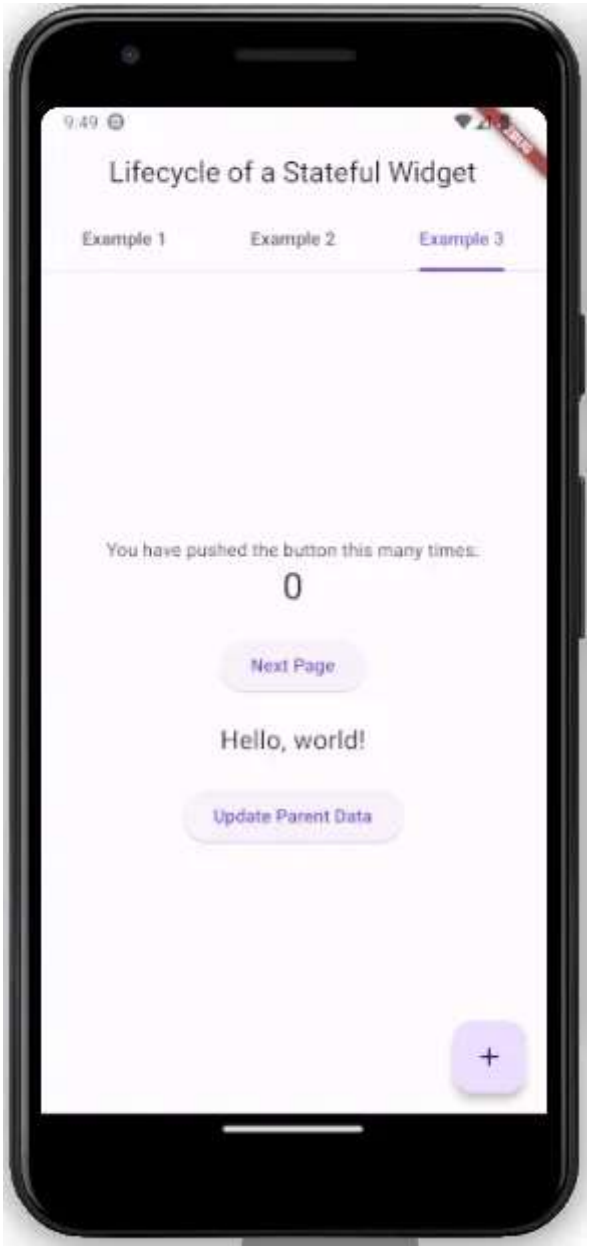
```
I/flutter (13129): setState
I/flutter (13129): build method
```



Example 3: Inherited Widgets

Inherited widgets in Flutter provide a way to propagate data and perform dependency injection down the widget tree, allowing parent widgets to share data and dependencies with their descendants efficiently. They establish a relationship where a parent widget can share data with its children, and any changes to the shared data will automatically update the dependent widgets.

The original example file has been edited to include an InheritedParent Widget in our original Column.



Here is the updated build method of our Example Widget.

```
@override
Widget build(BuildContext context) {
  print('build method');

  return Scaffold(
    body: Center(
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
          const Text(
            'You have pushed the button this many times:',
          ),
          Text(
            '$_counter',
            style: Theme.of(context).textTheme.headlineMedium,
          ),
          Padding(
            padding: const EdgeInsets.only(top: 20),
            child: ElevatedButton(
              onPressed: () => Navigator.of(context).pushReplacement(
                MaterialPageRoute(builder: (context) => const NewScreen()),
              ),
              child: const Text('Next Page'),
            ),
          ),
          const InheritedParent(), //addition: inherited parent widget
        ],
      ),
    ),
    floatingActionButton: FloatingActionButton(
      onPressed: _incrementCounter,
      tooltip: 'Increment',
      child: const Icon(Icons.add),
    ),
  );
}
```

A custom Inherited Widget class has been added, which extends the InheritedWidget class. The static of(context) method allows Inherited widget data to be accessed by descendants using the BuildContext and the context.dependOnInheritedWidgetOfExactType<T>() method. The UpdateShouldNotify method informs of any changes to this inherited data.

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

class MyInheritedWidget extends InheritedWidget {
  final String data;

  const MyInheritedWidget({
    required this.data,
    required Widget child,
    Key? key,
  }) : super(key: key, child: child);

  static MyInheritedWidget of(BuildContext context) {
    return context.dependOnInheritedWidgetOfExactType<MyInheritedWidget>()!;
  }

  @override
  bool updateShouldNotify(MyInheritedWidget oldWidget) {
    bool update = data != oldWidget.data;
    // ignore: avoid_print
    print('updateShouldNotify called. Update: $update');
    return update;
  }
}
```

The InheritedParent Widget consists of an InheritedWidget and an Elevated Button.

The InheritedWidget contains some shared data, stored inside the “parentData” variable, and has a child Widget called InheritedChild. The

Elevated Button updates the shared data value from “Hello, World!” to “Updated data”.

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

class InheritedParent extends StatefulWidget {
  const InheritedParent({super.key});

  @override
  State<InheritedParent> createState() {
    print('create state');
    return _InheritedParentState();
  }
}

class _InheritedParentState extends State<InheritedParent> {
  String parentData = 'Hello, world!';

  _InheritedParentState() {
    print('constructor, mounted: $mounted');
  }

  void updateParentData() {
    setState(() {
      parentData = 'Updated data';
    });
  }

  @override
  void initState() {
    super.initState();
    print('inherited parent initState, mounted: $mounted');
  }

  @override
  void didChangeDependencies() {
    super.didChangeDependencies();
    print('inherited parent didChangeDependencies, mounted: $mounted');
  }

  @override
  void setState(VoidCallback fn) {
    print('inherited parent setState');
```

```
    super.setState(fn);
  }

  @override
  Widget build(BuildContext context) {
    print('inherited parent build');

    return Column(
      children: [
        MyInheritedWidget(
          data: parentData,
          child: const InheritedChild(),
        ),
        ElevatedButton(
          onPressed: updateParentData,
          child: const Text('Update Parent Data'),
        ),
      ],
    );
  }

  @override
  void didUpdateWidget(covariant InheritedParent oldWidget) {
    super.didUpdateWidget(oldWidget);
    print('inherited parent didUpdateWidget, mounted: $mounted');
  }

  @override
  void deactivate() {
    super.deactivate();
    print('inherited parent deactivate, mounted: $mounted');
  }

  @override
  void dispose() {
    super.dispose();
    print('inherited parent dispose, mounted: $mounted');
  }

  @override
  void reassemble() {
    super.reassemble();
    print('inherited parent reassemble, mounted: $mounted');
  }
}
```

The InheritedChild Widget accesses the inherited data using the of(context) method, and displays this data on the UI.

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

class InheritedChild extends StatefulWidget {
  const InheritedChild({super.key});

  @override
  State<InheritedChild> createState() {
    print('create state');
    return _InheritedChildState();
  }
}

class _InheritedChildState extends State<InheritedChild> {
  _InheritedChildState() {
    print('constructor, mounted: $mounted');
  }

  @override
  void initState() {
    super.initState();
    print('inherited child initState, mounted: $mounted');
  }

  @override
  void didChangeDependencies() {
    super.didChangeDependencies();
    print('inherited child didChangeDependencies, mounted: $mounted');
  }

  @override
  void setState(VoidCallback fn) {
    print('inherited child setState');
    super.setState(fn);
  }

  @override
  Widget build(BuildContext context) {
    print('inherited child build');

    //get shared data from inherited widget
    final inheritedData = MyInheritedWidget.of(context).data;
```

```
      //display this data
      return Padding(
        padding: const EdgeInsets.all(20),
        child: Text(
          inheritedData,
          style: const TextStyle(fontSize: 20),
        ),
      );
    }

    @override
    void didUpdateWidget(covariant InheritedChild oldWidget) {
      super.didUpdateWidget(oldWidget);
      print('inherited child didUpdateWidget, mounted: $mounted');
    }

    @override
    void deactivate() {
      super.deactivate();
      print('inherited child deactivate, mounted: $mounted');
    }

    @override
    void dispose() {
      super.dispose();
      print('inherited child dispose, mounted: $mounted');
    }

    @override
    void reassemble() {
      super.reassemble();
      print('inherited child reassemble, mounted: $mounted');
    }
  }
```

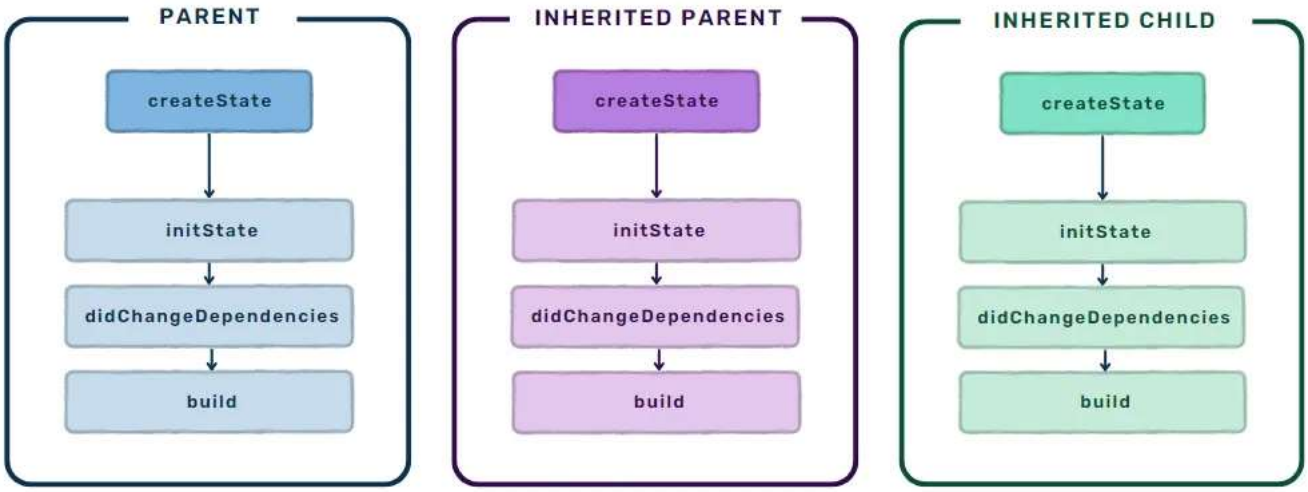
As always, all the Widgets are Stateful Widgets with their methods overridden and print statements added.

Application Started

When the app is started, as always all the Widgets' are initialised, their constructor is called, the object is created and inserted into the widget tree, and didChangedDependencies is called.

This occurs for all the widgets in the sequence they are created — so first the outside Widget is created, then the Inherited parent, followed by the inherited child.

```
I/flutter ( 4567): create state
I/flutter ( 4567): constructor, mounted: false
I/flutter ( 4567): initState, mounted: true
I/flutter ( 4567): didChangeDependencies, mounted: true
I/flutter ( 4567): build method
I/flutter ( 4567): create state
I/flutter ( 4567): constructor, mounted: false
I/flutter ( 4567): inherited parent initState, mounted: true
I/flutter ( 4567): inherited parent didChangeDependencies, mounted: true
I/flutter ( 4567): inherited parent build
I/flutter ( 4567): create state
I/flutter ( 4567): constructor, mounted: false
I/flutter ( 4567): inherited child initState, mounted: true
I/flutter ( 4567): inherited child didChangeDependencies, mounted: true
I/flutter ( 4567): inherited child build
```



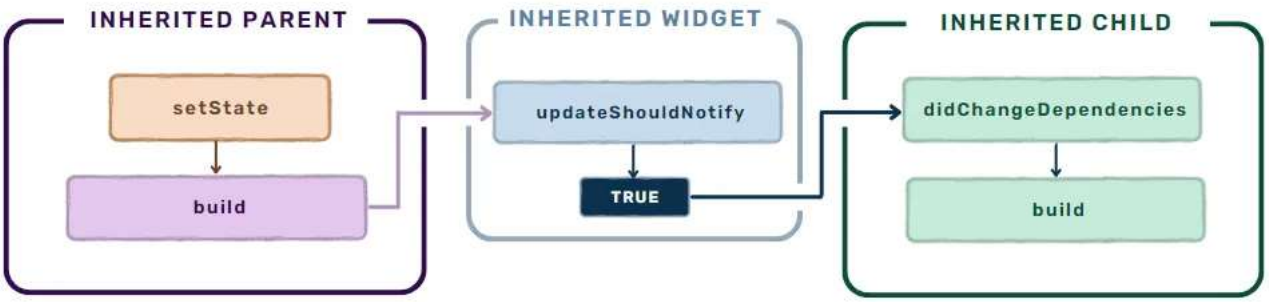
Pressing the "Update Parent Data" Button

When the “Update Parent Data” button is pressed, setState is called in InheritedParent, and the value is changed from “Hello World” to “Updated Data”. When this happens, as is the case with Inherited Widgets when the Parent’s is rebuilt, the updateShouldNotify method is invoked. This method checks whether there has been a change in the shared data, and returns true if there has been, and false if there is no change.

```
I/flutter ( 4567): inherited parent setState
I/flutter ( 4567): inherited parent build
I/flutter ( 4567): updateShouldNotify called, Update: true
I/flutter ( 4567): inherited child didChangeDependencies, mounted: true
I/flutter ( 4567): inherited child build
```

In this case, the value was modified from “Hello, World!” to “Updated Data”, therefore the updateShouldNotify method returns true, as can also be seen in the print statement.

As this method returns true, the child widget’s didChangeDependencies is invoked, which causes the child inherited widget to rebuild.

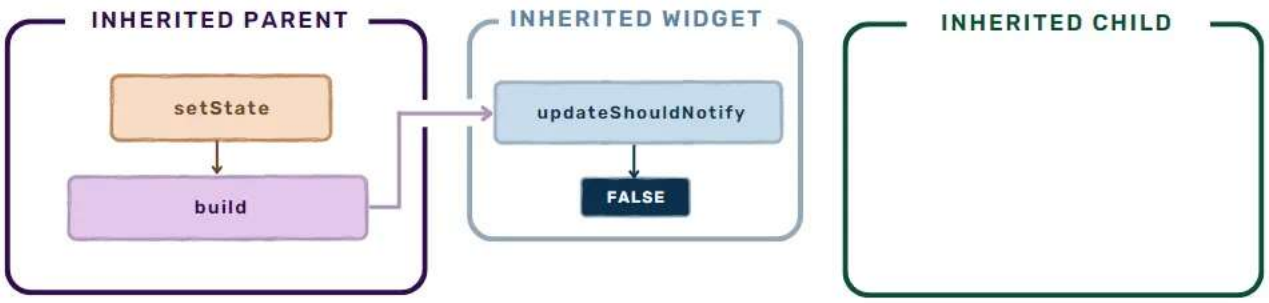


Button Pressed Again

When we press the same button again, setState is called in Parent, and once again the value is set to “Updated Data”.

```
I/flutter ( 4567): inherited parent setState
I/flutter ( 4567): inherited parent build
I/flutter ( 4567): updateShouldNotify called. Update: false
```

Since the Parent's setState is called, the parent is rebuilt. As before, updateShouldNotify is called to check for any changes in the shared data. In this scenario, there are no changes to the actual data — the value still reads "Updated Data". Therefore, the updateShouldNotify method returns false, and the Child Widget is not rebuilt.



Conclusion

In conclusion, this article delved into the lifecycle of Stateful Widgets in Flutter and explored the significance of each stage. To further explore the examples discussed here and dive deeper into Flutter's widget lifecycle, you can visit the GitHub repository.

Happy Coding!

Flutter | Statefulwidget | Mobile App Development | Coding | Dart

Written by Hadiyaaamir

9 Followers

Follow

More from Hadiyaaamir

Hadiyaaamir

Mason — A Complete and Comprehensive Guide

Everything you might want to know about Mason

12 min read · Nov 14, 2023

55

Hadiyaaamir

Mason — Beyond the Basics

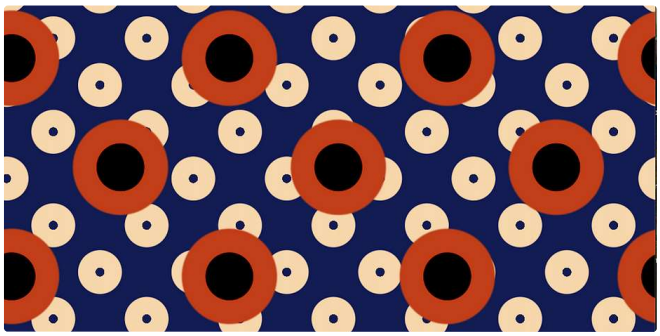
A Personal Journey into Advanced Features

9 min read · Nov 14, 2023

3

See all from Hadiyaaamir

Recommended from Medium



Martin Moyseev

Improve your Flutter code with Dart Patterns and Destructuring

Patterns in Dart are an efficient way to check if data conforms to a specific format. Upon a...

6 min read · Feb 8, 2024

173 2

Abubakar Saddique

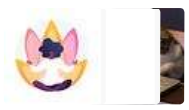
Drawer Widget Flutter

What is Drawer Widget in Flutter ??

11 min read · Sep 6, 2023

51

Lists



Stories to Help You Grow as a Software Developer

19 stories · 824 saves

General Coding Knowledge

20 stories · 929 saves

The New Chatbots: ChatGPT, Bard, and Beyond

12 stories · 307 saves

data science and AI

40 stories · 79 saves

Henry Ifebunandu

Supercharge Your Flutter Apps with Google's App Architecture.

Explore the powerful use of Google's App Architecture and learn how it can elevate yo...

10 min read · Sep 28, 2023

276

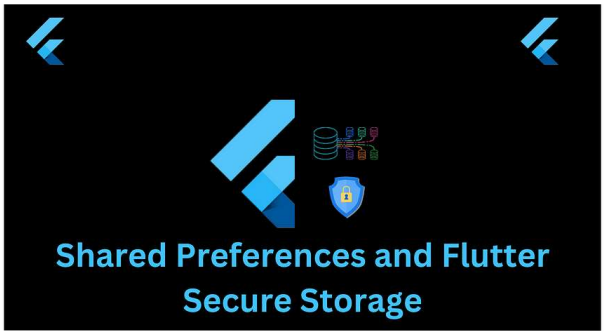
Kamlesh Lakhani

Dart | Interview Question

Is Dart compiled or interpreted language?

· 13 min read · Feb 3, 2024

92



Kalid Meftu in Stackademic

Shared Preferences and Flutter Secure Storage: when and how to...

Shared Preferences is the best option to store a small amount of data in flutter projects and...

Kevin Chisholm in Flutter

What's new in Flutter 3.19

Revolutionizing App Development with the Gemini API, Impeller Updates, and Windows...

5 min read · Oct 1, 2023

22

11 min read · 4 days ago

2K 12

See more recommendations