Widget — State — BuildContext — InheritedWidget

Didier Boelens Follow
Aug 6, 2018 · 15 min read

This article covers the important notions of Widget, State, BuildContext and InheritedWidget in Flutter Applications.

Special attention is paid on the InheritedWidget which is one of the most important and less documented widgets.

Difficulty: *Beginner*

. . .

Foreword

The notions of **Widget**, **State** and **BuildContext** in Flutter are ones of the most important concepts that every Flutter developer needs to fully understand.

However, the documentation is huge and this concept is not always clearly explained.

I will explain these notions with my own words and shortcuts, knowing that this might risk to shock some purists, but the real objective of this article to try to clarify the following topics:

- difference between Stateful and Stateless widgets
- what is a BuildContext
- what is a State and how to use it
- relationship between a BuildContext and its State object
- InheritedWidget and the way to propagate the information inside a Widgets tree
- notion of rebuild

. . .

Part 1: Concepts

Notion of Widget

In Flutter, almost everything is a **Widget**.

> *Think of a Widget as a visual component (or a component that interacts with the visual aspect of an application).*

When you need to build anything that directly or indirectly is in relation with the layout, you are using Widgets.

Notion of Widgets tree

Widgets are organized in tree structure(s).

A widget that contains other widgets is called **parent Widget** (or *Widget container*).

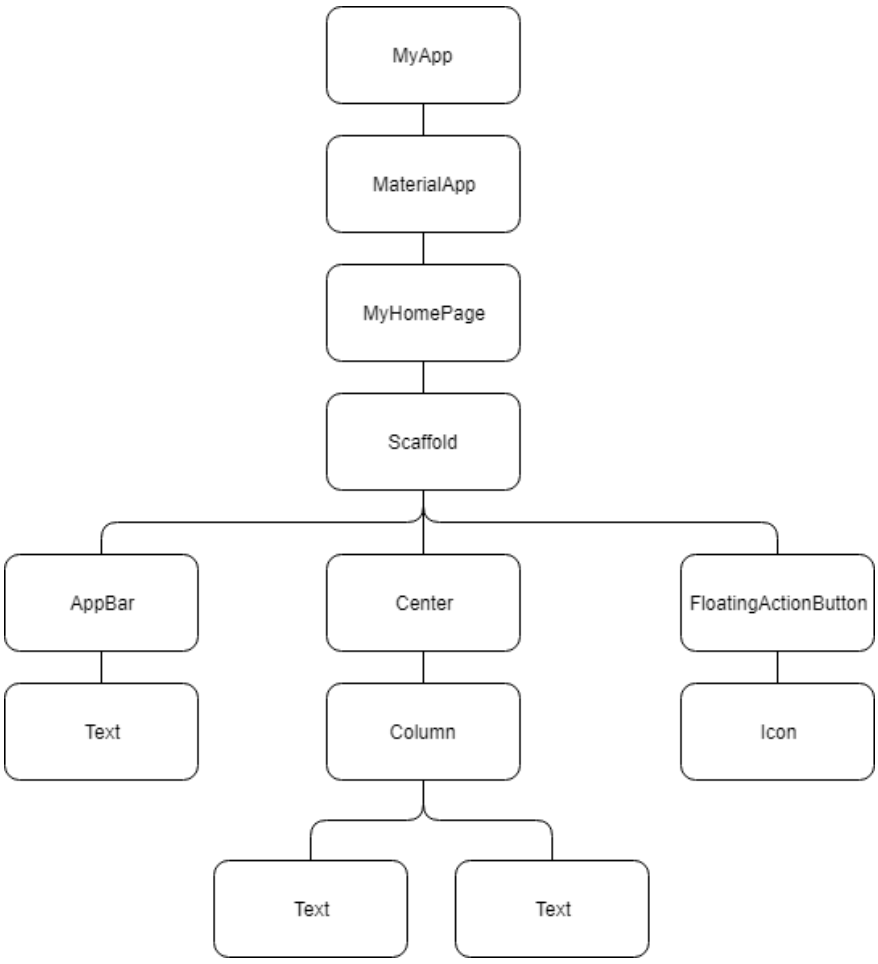
Widgets which are contained in a *parent Widget* are called **children Widgets**.

Let's illustrate this with the basic application which is automatically generated by *Flutter*.

Here is the simplified code, limited to the **build** method:

```
@override
Widget build(BuildContext){
    return new Scaffold(
      appBar: new AppBar(
        title: new Text(widget.title),
      ),
      body: new Center(
        child: new Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            new Text(
              'You have pushed the button this many times:'
            ),
            new Text(
              '$_counter',
              style: Theme.of(context).textTheme.display1,
            ),
          ]
```

If we now consider this basic example, we obtain the following Widgets tree structure (*limited the list of Widgets present in the code*):



Notion of BuildContext

Another important notion is the **BuildContext**.

A BuildContext is nothing else but a reference to the location of a Widget within the tree structure of all the Widgets which are built.

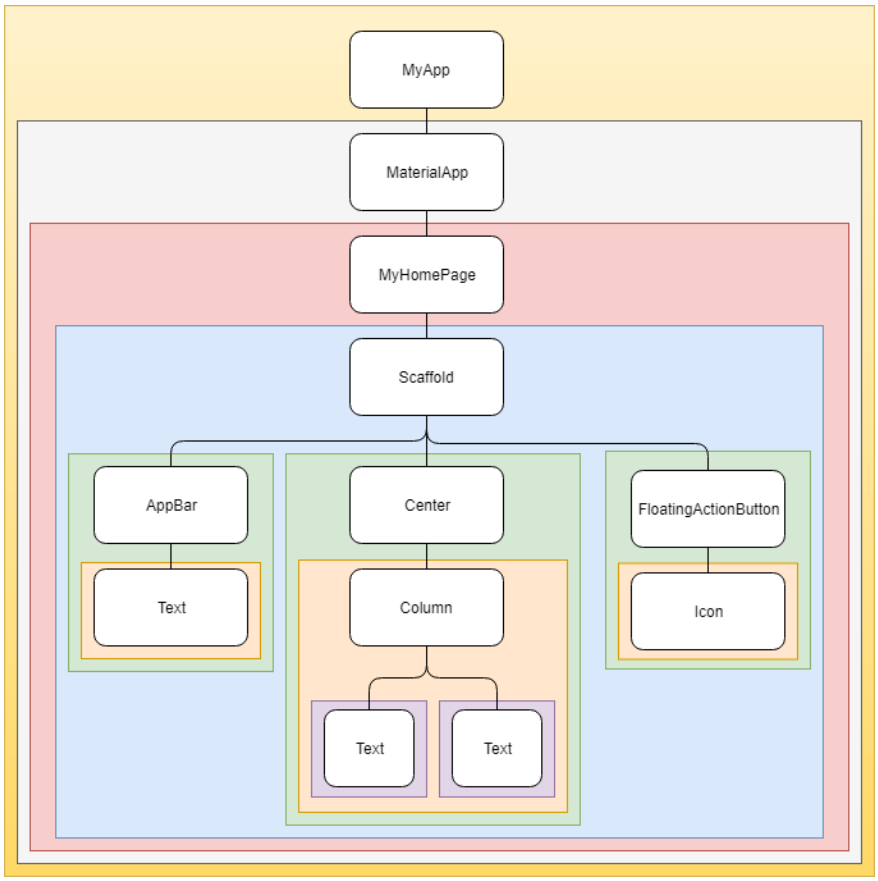
> *In short, think of a BuildContext as the part of Widgets tree where the Widget is attached to this tree.*

A *BuildContext* only belongs to **one** widget.

If a widget 'A' has children widgets, the *BuildContext* of widget 'A' will become the *parent BuildContext* of the direct *children BuildContexts*.

Reading this, it is clear that **BuildContexts are chained** and are composing a tree of BuildContexts (parent-children relationship).

If we now try to illustrate the notion of **BuildContext** in the previous diagram, we obtain (*still as a very simplified view*) where each color represents a **BuildContext** (*except the MyApp one, which is different*):



> ***BuildContext visibility*** *(*Simplified statement*):*
>
> *'Something' is only visible within its own BuildContext or in the BuildContext of its parent(s) BuildContext.*

From this statement we can derive that from a child BuildContext, it is easily possible to find an **ancestor** (= parent) Widget.

> *An example is, considering the Scaffold > Center > Column > Text: context.ancestorWidgetOfExactType(Scaffold) => returns the first Scaffold by going up to tree structure from the Text context.*

From a parent BuildContext, it is also possible to find a **descendant** (= child) Widget but it is not advised to do so (*we will discuss this later*).

. . .

Types of Widgets

Widgets are of 2 types:

Stateless Widget

Some of this visual components do not depend on anything else but their own configuration information, which is provided **at time of building it** by its direct parent.

In other words, these Widgets will not have to care about any *variation*, once created.

These Widgets are called **Stateless Widgets**.

Typical examples of such Widgets could be Text, Row, Column, Container… where during the building time, we simply pass some parameters to them.

Parameters might be anything from a decoration, dimensions, or even other widget(s). It does not matter. The only thing which is important is that this configuration, once applied, will not change before the next build process.

> *A stateless widget can only be drawn once when the Widget is loaded/built, which means that that Widget cannot be redrawn based on any events or user actions.*

Stateless Widget lifecycle

Here is a typical structure of the code related to a *Stateless Widget*.

As you may see, we can pass some additional parameters to its constructor. However, bear in mind that these parameters will ***NOT*** change (mutate) at a later stage and have to be only used *as is*.

```
class MyStatelessWidget extends StatelessWidget {
        MyStatelessWidget({
                Key key,
                this.parameter,
        }): super(key:key);

        final parameter;

        @override
```

Even if there is another method that could be overridden (*createElement*), the latter is almost never overridden.

The only one that **needs** to be overridden is **build**.

The lifecycle of such Stateless widget is straightforward:

- initialization
- rendering via build()

. . .

Stateful Widget

Some other Widgets will handle some *inner data* that will change during the Widget's lifetime. This *data* hence becomes **dynamic**.

The set of *data* held by this Widget and which may vary during the lifetime of this Widget is called a **State**.

These Widgets are called **Stateful Widgets**.

An example of such Widget might be a list of Checkboxes that the user can select or a Button which is disabled depending on a condition.

Notion of State

A **State** defines the "*behavioural*" part of a *StatefulWidget* instance.

It holds information aimed at **interacting / interfering** with the Widget in terms of:

- behavior
- layout

> *Any changes which is applied to a* State *forces the Widget to* ***rebuild****.*

Relation between a State and a BuildContext

For *Stateful widgets*, a *State* is associated with a *BuildContext*.

This association is ***permanent*** and the *State* object will never change its *BuildContext*.

Even if the Widget BuildContext can be moved around the tree structure, the *State* will remain associated with that *BuildContext*.

When a *State* is associated with a *BuildContext*, the *State* is considered as **mounted**.

> ***HYPER IMPORTANT****:*
>
> *As a* State object *is associated with a* BuildContext*, this means that the* State object *is* ***NOT*** *(directly) accessible through* another BuildContext *! (we will further discuss this in a few moment).*

. . .

Stateful Widget lifecycle

Now that the base concepts have been introduced, it is time to dive a bit deeper…

Here is a typical structure of the code related to a *Stateful Widget*.

As the main objective of this article is to explain the notion of *State* in terms of "variable" data, I will intentionally skip any explanation related to some Stateful Widget *overridable* methods, which do not specifically relate to this.

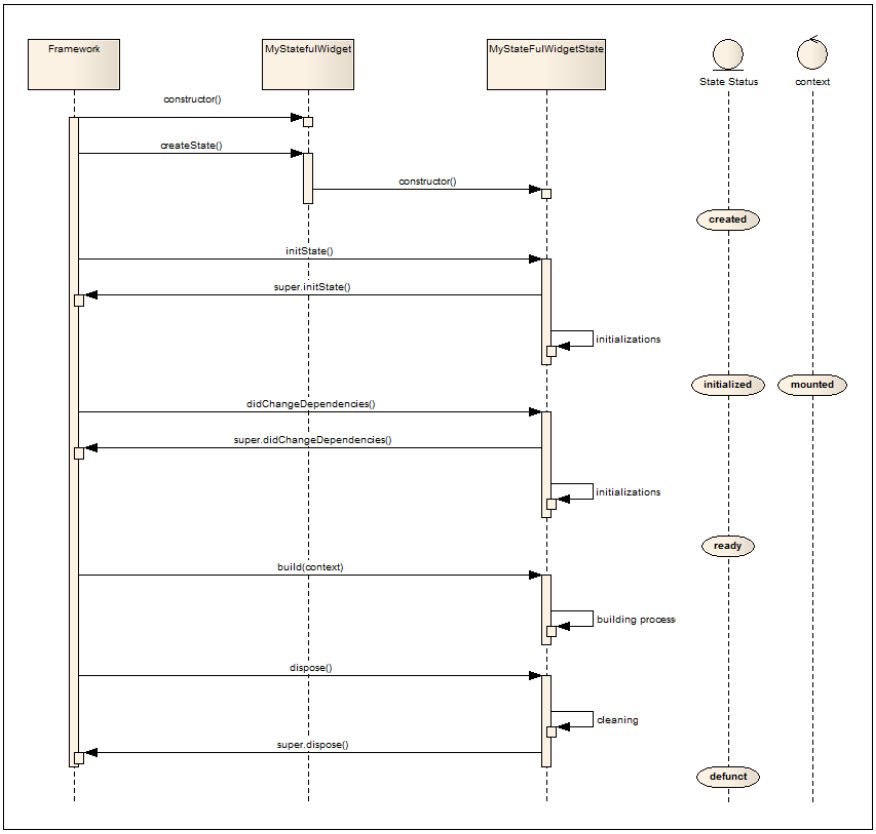
These overridable methods are *didUpdateWidget, deactivate, reassemble*. These will be discussed in another article.

```
class MyStatefulWidget extends StatefulWidget {
        MyStatefulWidget({
                Key key,
                this.parameter,
        }): super(key: key);

        final parameter;

        @override
        _MyStatefulWidgetState createState() => new _MyStat
}

class _MyStatefulWidgetState extends State<MyStatefulWidget

        @override
        void initState(){
                super.initState();

                // Additional initialization of the State
        }

        @override
        void didChangeDependencies(){
                super.didChangeDependencies();

                // Additional code
```

The following diagram shows (*a simplified version of*) the sequence of actions/calls related to the creation of a Stateful Widget.

At the right side of the diagram, you will notice the inner status of the *State* object during the flow.

You will also see the moment when the context is associated with the state, and thus becomes available (*mounted*).



So let's explain it with some additional details:

initState()

The *initState()* method is the very first method (after the constructor) to be called once the State object has been created.

This method is to be overridden when you need to perform additional initializations. Typical initializations are related to animations, controllers…

If you override this method, you need to call the **super.initState()** method and normally at first place.

In this method, a *context* is available but you **cannot** really use it yet since the framework has not yet fully associated the state with it.

Once the *initState()* method is complete, the State object is now initialized and the *context*, available.

> *This method will not be invoked anymore during the lifetime of this State object.*

didChangeDependencies()

The *didChangeDependencies()* method is the second method to be invoked.

At this stage, as the *context* is available, you may use it.

This method is usually overridden if your Widget is linked to an **InheritedWidget** and/or if you need to initialize some *listeners* (based on the *BuildContext*).

> *Note that if your widget is linked to an* InheritedWidget*, this method will be called each time this Widget will be rebuilt.*

If you override this method, you should invoke the *super.didChangeDependencies()* at first place.

build()

The *build(BuildContext context)* method is called after the *didChangeDependencies()* (and *didUpdateWidget*).

This is the place where you build your widget (and potentially any sub-tree).

This method will be called **each time your State object changes** (or when an InheritedWidget needs to notify the "*registered*" widgets) !!

In order to force a rebuild, you may invoke *setState(){…})* method.

dispose()

The *dispose()* method is called when the widget is discarded.

Override this method if you need to perform some cleanup (e.g. listeners, controllers…), then invoke the *super.dispose()* right after.

• • •

Stateless or Stateful Widget?

This is a question that many developers need to ask themselves: "*do I need my Widget to be Stateless or Stateful?*"

In order to answer this question, ask yourself:

> *In the lifetime of my widget, do I need to consider a **variable** that will change and when changed, will force the widget to be **rebuilt**?*

If the answer to the question is *yes*, then you need a *Stateful* widget, otherwise, you need a *Stateless* widget.

Some examples:

- a widget to display a list of checkboxes. To display the checkboxes, you need to consider an array of items. Each item is an object with a title and a status. If you click on a checkbox, the corresponding item.status is toggled;
  In this case, you need to use a *Stateful* widget to remember the status of the items to be able to redraw the checkboxes.
- a screen with a Form. The screen allows the user to fill the Widgets of the Form and send the form to the server.
  In this case, *_unless you need to validate the Form or do any other action before submitting it_*, a *Stateless* widget might be enough.

. . .

Stateful Widget is made of 2 parts

Remember the structure of a **Stateful** widget? There are 2 parts:

The Widget main definition

```
1  class MyStatefulWidget extends StatefulWidget {
2      MyStatefulWidget({
3                  Key key,
4                  this.color,
5          }): super(key: key);
6
7          final Color color;
8
```

The first part "*MyStatefulWidget*" is *normally* the **public** part of the Widget. You instantiate this part when you want to add it to a widget tree.

This part does not vary during the lifetime of the Widget but may accept parameters that could be used by its corresponding *State*

instance.

> *Note that any variable, defined at the level of this first part of the Widget will* normally ***NOT*** *change during its lifetime.*

The Widget State definition

```
class _MyStatefulWidgetState extends State<MyStatefulWidget>
    ...
        @override
        Widget build(BuildContext context){
            ...
```

The second part "*_MyStatefulWidgetState*" is the part which **varies** during the lifetime of the Widget and forces this specific instance of the Widget to rebuild each time a modification is applied.

The '_' character in the beginning of the name makes the class **private** to the *.dart* file. If you need to make a reference to this class outside the *.dart* file, do not use the '_' prefix.

The *_MyStatefulWidgetState* class can access any variable which is stored in the *MyStatefulWidget*, using **widget.{name of the variable}**.

In this example: *widget.color*

. . .

Widget unique identity—Key

In Flutter, each Widget is uniquely identified. This unique identity is defined by the framework **at build/rendering time**.

This unique identity corresponds to the optional **Key** parameter. If omitted, Flutter will generate one for you.

In some circumstances, you might need to force this **key**, so that you can access a widget by its key.

To do so, you can use one of the following helpers: *GlobalKey<T>*, *LocalKey*, *UniqueKey* or *ObjectKey*.

The *GlobalKey* ensures that the key is unique across the whole application.

To force a unique identity of a Widget:

```
GlobalKey myKey = new GlobalKey();
    ...
    @override
    Widget build(BuildContext context){
        return new MyWidget(
            key: myKey
```

. . .

Part 2: How to access the State?

As previously explained, a **State** is linked to **one BuildContext** and a **BuildContext** is linked to an **instance** of a Widget.

1. The Widget itself

In theory, the only one which is able to access a *State* is the **Widget State itself**.

In this case, there is no difficulty. The Widget State class accesses any of its variables.

2. A direct child Widget

Sometimes, a parent widget might need to get access to the State of one of its direct children to perform specific tasks.

In this case, to access these direct children *State*, you need to **know** them.

The easiest way to call somebody is via a *name*. In Flutter, each Widget has a unique identity, which is determined at **build/rendering time** by the framework.

As shown earlier, you may force the identity of a Widget, using the **key** parameter.

```
...
    GlobalKey<MyStatefulWidgetState> myWidgetStateKey = new
    ...
    @override
    Widget build(BuildContext context){
        return new MyStatefulWidget(
            key: myWidgetStateKey,
```

Once identified, a *parent* Widget might access the *State* of its child via:

> *myWidgetStateKey.currentState*

> *Side note:*
>
> *Some of you might have noticed that in the previous example, I applied a little change to the name of the Widget state (removal of the "_" at the beginning of the name). As previously said, prefixing the name of a class with "_", makes the class* private *to the* *.dart *file. As in the example, we need to expose the class, we need to remove this "_".*

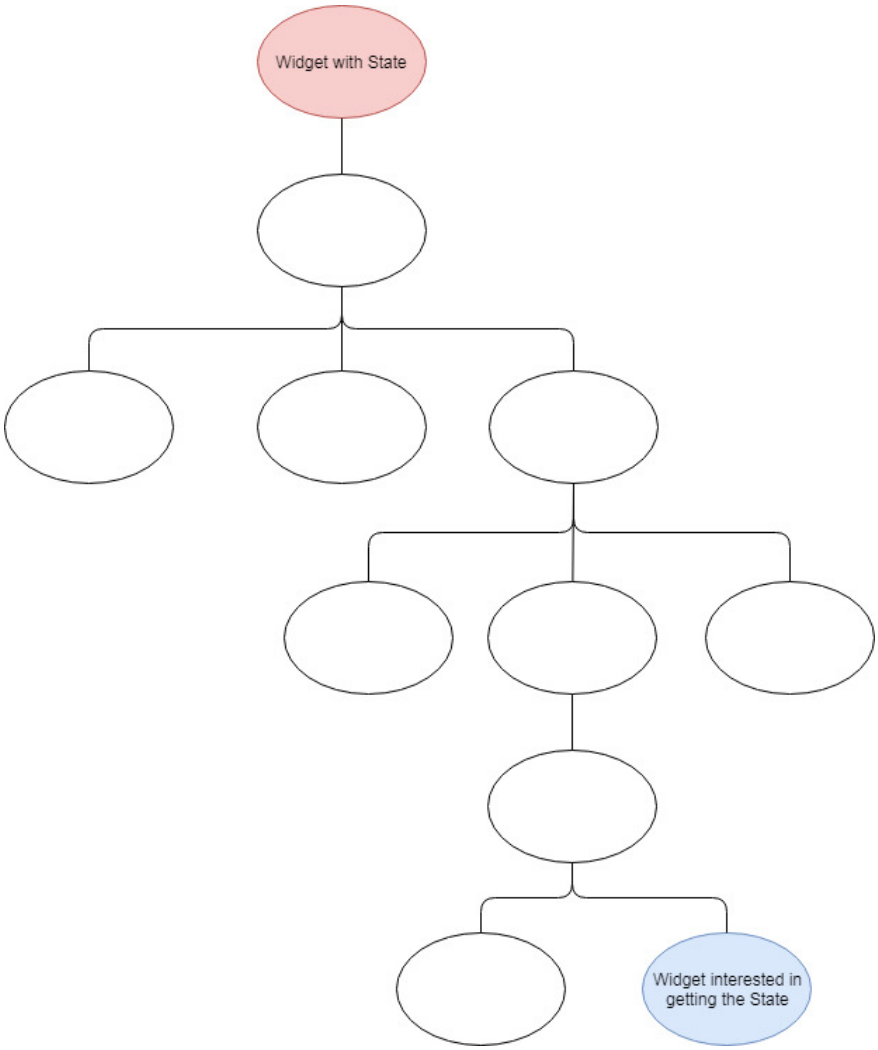
Let's consider a basic example that shows a SnackBar when the user hits a button.

As the SnackBar is a child Widget of the Scaffold it is not directly accessible to any other child of the body of the Scaffold (*remember the notion of context and its hierarchy/tree structure ?*). Therefore, the only way to access it, is via the *ScaffoldState*, which exposes a public method to show the SnackBar.

```
class _MyScreenState extends State<MyScreen> {
        /// the unique identity of the Scaffold
        final GlobalKey<ScaffoldState> _scaffoldKey = new G

        @override
        Widget build(BuildContext context){
            return new Scaffold(
                key: _scaffoldKey,
                appBar: new AppBar(
                    title: new Text('My Screen'),
                ),
                body: new Center(
                    new RaiseButton(
                        child: new Text('Hit me'),
                        onPressed: (){
                            _scaffoldKey.currentState.showS
                                new SnackBar(
```

3. Ancestor Widget

Suppose that you have a Widget that belongs to a sub-tree of another Widget as shown in the following diagram.



3 conditions need to be met to make this possible:

1. the "Widget with State" (in red) needs to expose its State

In order to *expose* its *State*, the Widget needs to record it at time of creation, as follows:

```
class MyExposingWidget extends StatefulWidget {

   MyExposingWidgetState myState;

   @override
   MyExposingWidgetState createState(){
      myState = new MyExposingWidgetState();
```

2. the "Widget State" needs to expose some getters/setters

In order to let a "*stranger*" to set/get a property of the State, the *Widget State* needs to authorize the access, through:

- public property (not recommended)
- getter / setter

Example:

```
class MyExposingWidgetState extends State<MyExposingWidget>{
   Color _color;

   Color get color => _color;
   ...
```

3. the "Widget interested in getting the State" (in blue) needs to get a reference to the State

```
class MyChildWidget extends StatelessWidget {
   @override
   Widget build(BuildContext context){
      final MyExposingWidget widget = context.ancestorWidge
      final MyExposingWidgetState state = widget?.myState;

      return new Container(
         color: state == null ? Colors.blue : state.color,
```

This solution is easy to implement but how does the child widget know when it needs to rebuild?

With this solution, it **does not**. It will have to wait for a rebuild to happen to refresh its content, which is not very convenient.

The next section tackles the notion of **Inherited Widget** which gives a solution to this problem.

. . .

InheritedWidget

In short and with simple words, the **InheritedWidget** allows to efficiently propagate (and share) information down a tree of *widgets*.

The **InheritedWidget** is a special Widget, that you put in the Widgets tree as a parent of another sub-tree. All widgets part of that sub-tree will have to ability to *interact* with the data which is exposed by that **InheritedWidget**.

Basics

In order to explain it, let's consider the following piece of code:

```
class MyInheritedWidget extends InheritedWidget {
   MyInheritedWidget({
      Key key,
      @required Widget child,
      this.data,
   }): super(key: key, child: child);

   final data;

   static MyInheritedWidget of(BuildContext context) {
      return context.inheritFromWidgetOfExactType(MyInherit
```

This code defines a Widget, named "*MyInheritedWidget*", aimed at "*sharing*" some data across all widgets, part of the child sub-tree.

As mentioned earlier, an **InheritedWidget** needs to be positioned at the top of a widgets tree in order to be able to propagate/share some data, this explains the "@required Widget child" which is passed to the InheritedWidget base constructor.

The "*static MyInheritedWidget of(BuildContext context)*" method, allows all the children widgets to get the instance of the closest *MyInheritedWidget*" which encloses the context (see later).

Finally the "*updateShouldNotify*" overridden method is used to tell the *InheritedWidget* whether notifications will have to be passed to all the children widgets (that registered/subscribed) if a modification be applied to the *data* (see later).

Therefore, we need to put it at a tree node level as follows:

```
class MyParentWidget... {
   ...
   @override
   Widget build(BuildContext context){
      return new MyInheritedWidget(
         data: counter,
         child: new Row(
            children: <Widget>[
               ...
            ],
```

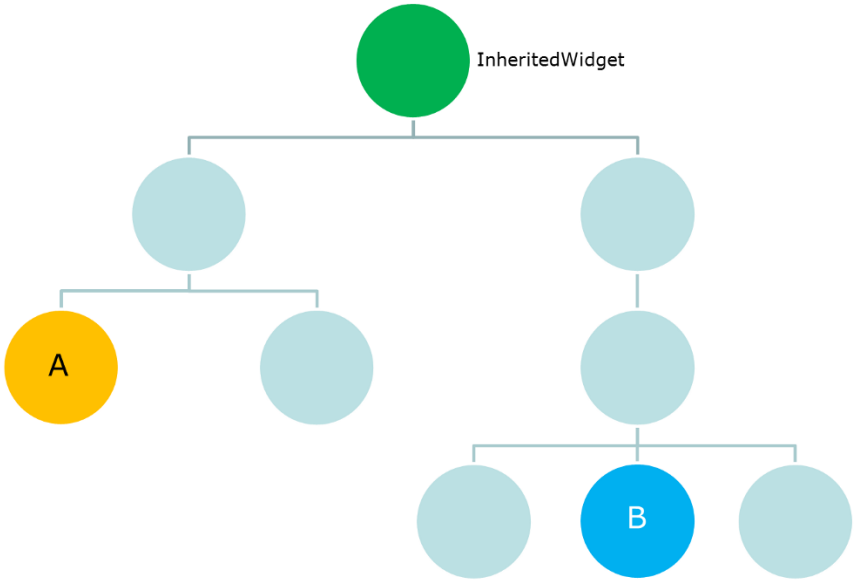
How does a child get access to the data of the InheritedWidget?

At time of building a child, the latter will get a reference to the InheritedWidget, as follows:

```
class MyChildWidget... {
   ...

   @override
   Widget build(BuildContext context){
      final MyInheritedWidget inheritedWidget = MyInherited

      ///
      /// From this moment, the widget can use the data, ex
      /// by calling:  inheritedWidget.data
      ///
```

How to make interactions between Widgets?

Consider the following diagram that shows a widgets tree structure.



In order to illustrate a type of interaction, let's suppose the following:

- 'Widget A' is a button that adds an item to the shopping cart;
- 'Widget B' is a Text that displays the number of items in the shopping cart;
- 'Widget C' is next to Widget B and is a Text with any text inside;
- We want the 'Widget B' to automatically display the right number of items in the shopping cart, as soon as the 'Widget A' is pressed but we do not want 'Widget C' to be rebuilt

The **InheritedWidget** is just the right Widget to use for that!

Example by the code

Let's first write the code and explanations will follow:

```
class Item {
   String reference;

   Item(this.reference);
}

class _MyInherited extends InheritedWidget {
  _MyInherited({
    Key key,
    @required Widget child,
    @required this.data,
  }) : super(key: key, child: child);

  final MyInheritedWidgetState data;

  @override
  bool updateShouldNotify(_MyInherited oldWidget) {
    return true;
  }
}

class MyInheritedWidget extends StatefulWidget {
  MyInheritedWidget({
    Key key,
    this.child,
  }): super(key: key);

  final Widget child;

  @override
  MyInheritedWidgetState createState() => new MyInheritedW

  static MyInheritedWidgetState of(BuildContext context){
    return (context.inheritFromWidgetOfExactType(_MyInheri
  }
}

class MyInheritedWidgetState extends State<MyInheritedWidg
  /// List of Items
  List<Item> _items = <Item>[];

  /// Getter (number of items)
  int get itemsCount => _items.length;
```

```
45      /// Helper method to add an Item
46      void addItem(String reference){
47        setState((){
48          _items.add(new Item(reference));
49        });
50      }
51
52      @override
53      Widget build(BuildContext context){
54        return new _MyInherited(
55          data: this,
56          child: widget.child,
57        );
58      }
59    }
60
61    class MyTree extends StatefulWidget {
62      @override
63      _MyTreeState createState() => new _MyTreeState();
64    }
65
66    class _MyTreeState extends State<MyTree> {
67      @override
68      Widget build(BuildContext context) {
69        return new MyInheritedWidget(
70          child: new Scaffold(
71            appBar: new AppBar(
72              title: new Text('Title'),
73            ),
74            body: new Column(
75              children: <Widget>[
```

Explanations

In this very basic example,

- *_MyInherited* is an **InheritedWidget** that is recreated each time we add an Item via a click on the button of 'Widget A'
- *MyInheritedWidget* is a Widget with a **State** that contains the list of Items. This *State* is accessible via the "*static MyInheritedWidgetState of(BuildContext context)*"
- *MyInheritedWidgetState* exposes one getter (*itemsCount*) and one method (*addItem*) so that they will be usable by the widgets, part

of the *child* widgets tree

- Each time we add an Item to the State, the *MyInheritedWidgetState* rebuilds
- *MyTree* class simply builds a widgets tree, having the *MyInheritedWidget* as parent of the tree
- *WidgetA* is a simple *RaisedButton* which, when pressed, invokes the *addItem* method from the **closest** *MyInheritedWidget*
- *WidgetB* is a simple *Text* which displays the number of items, present at the level of the **closest** *MyInheritedWidget*

How does all this work?

Registration of a Widget for later notifications

When a child Widget invokes the *MyInheritedWidget.of(***context***)*, it makes a call to the following method of MyInheritedWidget, passing its own *BuildContext*.

```
static MyInheritedWidgetState of(BuildContext context) {
  return (context.inheritFromWidgetOfExactType(_MyInherited)
as _MyInherited).data;
}
```

Internally, on top of simply returning the instance of *MyInheritedWidgetState*, it also subscribes the *consumer* widget to the changes notifications.

Behind the scene, the simple call to this static method actually does 2 things:

- the "*consumer*" widget is automatically added to the list of **subscribers** that will be **rebuilt** when a modification is applied to the **InheritedWidget** (here *_MyInherited*)
- the *data* referenced in the *_MyInherited* widget (aka *MyInheritedWidgetState*) is returned to the "*consumer*"

Flow

Since both 'Widget A' and 'Widget B' have subscribed with the **InheritedWidget** so that if a modification is applied to the

_MyInherited, the flow of operations is the following (simplified version) when the *RaisedButton* of Widget A is clicked:

- A call is made to the *addItem* method of *MyInheritedWidgetState*
- *MyInheritedWidgetState.addItem* method adds a new Item to the List<Item>
- *setState()* is invoked in order to rebuild the *MyInheritedWidget*
- A new instance of *_MyInherited* is created with the new content of the List<Item>
- *_MyInherited* records the new *State* which is passed in argument (*data*)
- As an *InheritedWidget*, it checks whether there is a need to "*notify*" the "*consumers*" (answer is true)
- It iterates the whole list of *consumers* (here Widget A and Widget B) and requests them to rebuild
- As Wiget C is not a *consumer*, it is not rebuilt.

So it works !

However, both Widget A and Widget B are rebuilt while it is useless to rebuild Wiget A since nothing changed for it.

How to prevent this from happening?

Prevent some Widgets from rebuilding while still accessing the Inherited Widget

The reason why Widget A was also rebuilt comes from the way it accesses the *MyInheritedWidgetState*.

As we saw earlier, the fact of invoking the "*context.inheritFromWidgetOfExactType()*" method automatically subscribed the Widget to the list of "*consumers*".

The solution to prevent this automatic subscription while still allowing the Widget A access the *MyInheritedWidgetState* is to change the static method of *MyInheritedWidget* as follows:

```
static MyInheritedWidgetState of([BuildContext context, bool
    return (rebuild ? context.inheritFromWidgetOfExactType(_
                    : context.ancestorWidgetOfExactType(_MyI
  }
```

By adding a boolean extra parameter…

- If the "*rebuild*" parameter is true (by default), we use the normal approach (and the Widget will be added to the list of subscribers)
- If the "*rebuild*" parameter is false, we still get access to the data **but** without using the *internal implementation* of the *InheritedWidget*

So, to complete the solution, we also need to slightly update the code of Widget A as follows (we add the false extra parameter):

```
class WidgetA extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final MyInheritedWidgetState state = MyInheritedWidget.
    return new Container(
      child: new RaisedButton(
        child: new Text('Add Item'),
        onPressed: () {
          state.addItem('new item');
        },
```

There it is, Widget A is no longer rebuilt when we press it.

. . .

Special note for Routes, Dialogs…

> *Routes, Dialogs BuildContexts are tied to the* ***Application****.*

> *This means that even if inside a Screen A you request to display another Screen B (on top of the current, for example), there is* "no easy way" *from any of the 2 screens to relate their own contexts.*

> *The only way for Screen B to know anything about the context of Screen A is to obtain it from Screen A as parameter of*

> *Navigator.of(context).push(….)*

. . .

Interesting links

- Maksim Ryzhikov
- Chema Molins
- Official documentation
- Video from Google I/O 2018
- Scoped_Model

. . .

Conclusions

There is still so much to say on these topics… especially on **InheritedWidget**.

Other topics of interest are **Notifiers / Listeners** and also (maybe mainly) the notion of **Streams** but this will be covered in other articles.

Thanks for reading this quite long article, stay tuned for the next to come and happy coding…

Didier (alias boeledi)

. . .

Many thanks to Simon Lightfoot and Nash for encouraging me to post this article on Medium.

Other articles can also be found on my personal blog, also in French.

My Twitter: @DidierBoelens