Widget - State - Context - InheritedWidgetBlog

June 03, 2018

This article covers the important notions of Widget, State, Context 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 **Context** 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 of Stateful and Stateless widgets
- what is a Context
- what is a State and how to use it
- relationship between a context and its state object
- InheritedWidget and the way to propagate the information inside a Widgets tree
- notion of rebuild

This article is also available on Medium - Flutter Community.

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

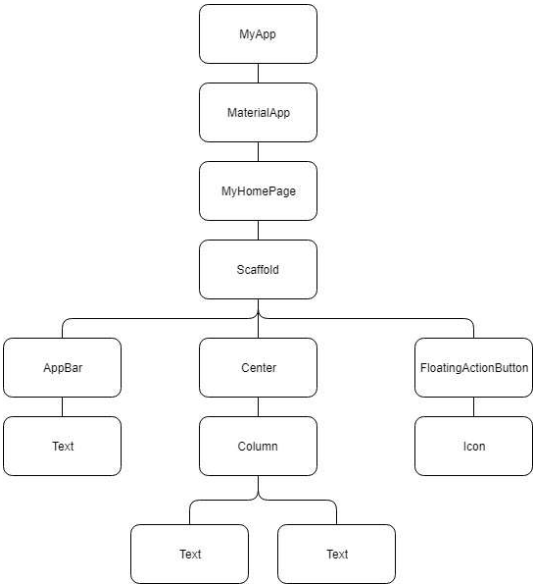
Widgets are organized in tree structure(s).

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

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 Context

Another important notion is the **Context**.

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

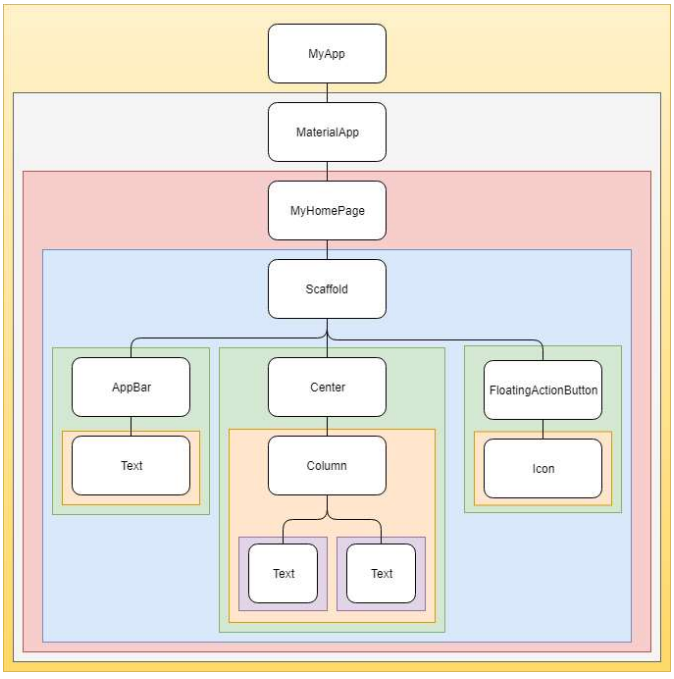
> In short, think of a *context* as the part of Widgets tree where the Widget is attached to this tree.

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

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

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

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



> **Context Visibility** (*Simplified statement*):
>
> *Something* is only visible within its own context or in the context of its parent(s) context.

From this statement we can derive that from a child context, 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 context, 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 building process.

> A stateless widget can only be drawn only 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
    Widget build(BuildContext context){
        return new ...
    }
}
```

Even if there is another method that could be overridden (*createElement*), the latter is barely 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 / interferring** with the Widget in terms of:

- behaviour
- layout

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

Relation between a State and a Context

For *Stateful widgets*, a *State* is associated with a *Context*. This association is *permanent* and the *State* object will never change its *context*.

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

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

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

Stateful Widget lifecycle

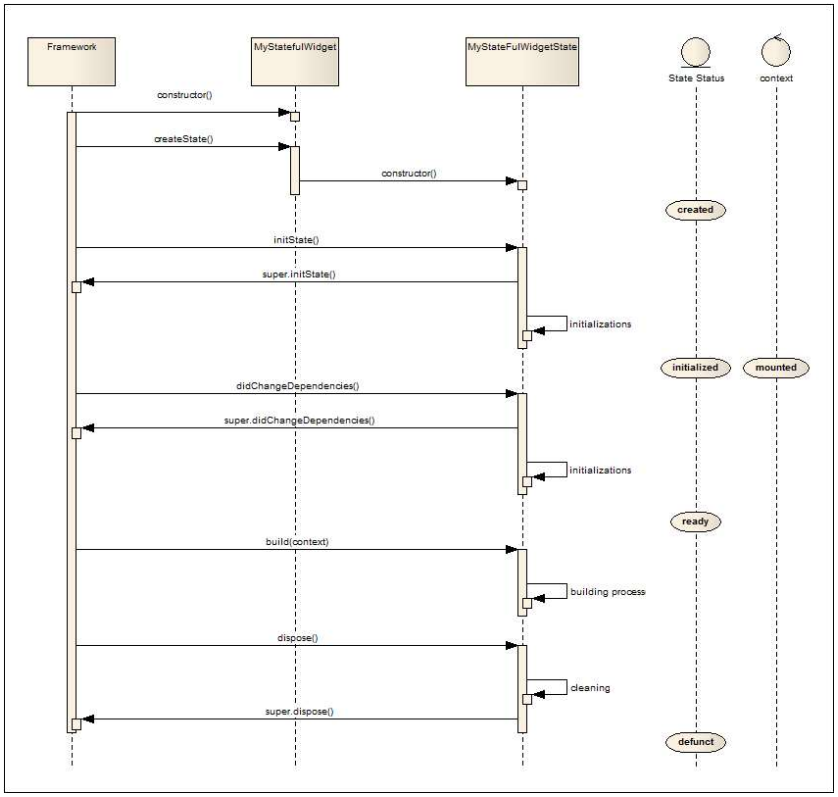
Now that the base concepts have been introduced, it is time to dive a bit deeper...

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 a next article.

```dart
class MyStatefulWidget extends StatefulWidget {

    MyStatefulWidget({
        Key key,
        this.parameter,
    }): super(key: key);

    final parameter;

    @override
    _MyStatefulWidgetState createState() => new _MyStatefulWidgetState();
}

class _MyStatefulWidgetState extends State<MyStatefulWidget> {

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

        // Additional initialization of the State
    }

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

        // Additional code
    }

    @override
    void dispose(){
        // Additional disposal code

        super.dispose();
    }

    @override
    Widget build(BuildContext context){
        return new ...
    }
}
```

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 *context*).

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), 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

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

    final Color color;

    @override
    _MyStatefulWidgetState createState() => new _MyStatefulWidgetState();
}
```

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*, *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 Context** and a **Context** 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 GlobalKey<MyStatefulWidgetState>();
...
@override
Widget build(BuildContext context){
    return new MyStatefulWidget(
        key: myWidgetStateKey,
        color: Colors.blue,
    );
}
```

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

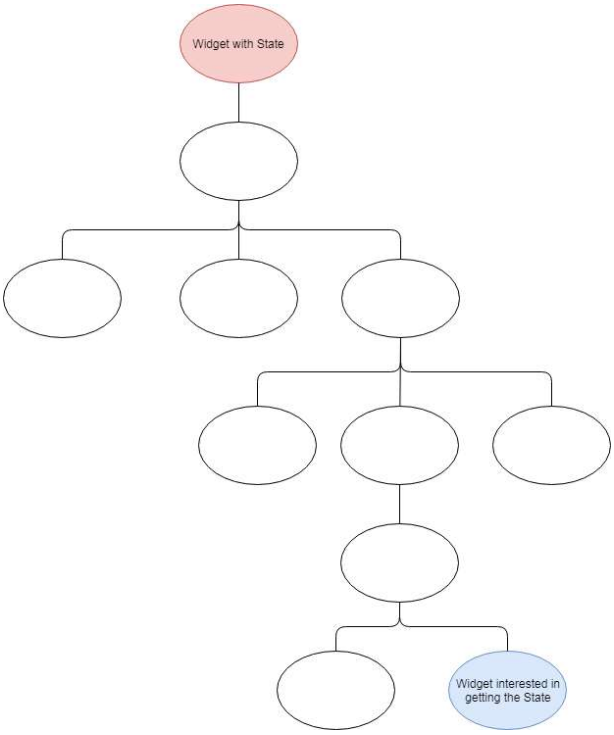
```
myWidgetStateKey.currentState
```

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 GlobalKey<ScaffoldState>();

    @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.showSnackBar(
                            new SnackBar(
                                content: new Text('This is the 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();
      return myState;
   }
}
```

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.ancestorWidgetOfExactType(MyExposingWidget);
      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(MyInheritedWidget);
   }

   @override
   bool updateShouldNotify(MyInheritedWidget oldWidget) => data != oldWidget.data;
}
```

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>[
               ...
            ],
         ),
      );
   }
}
```

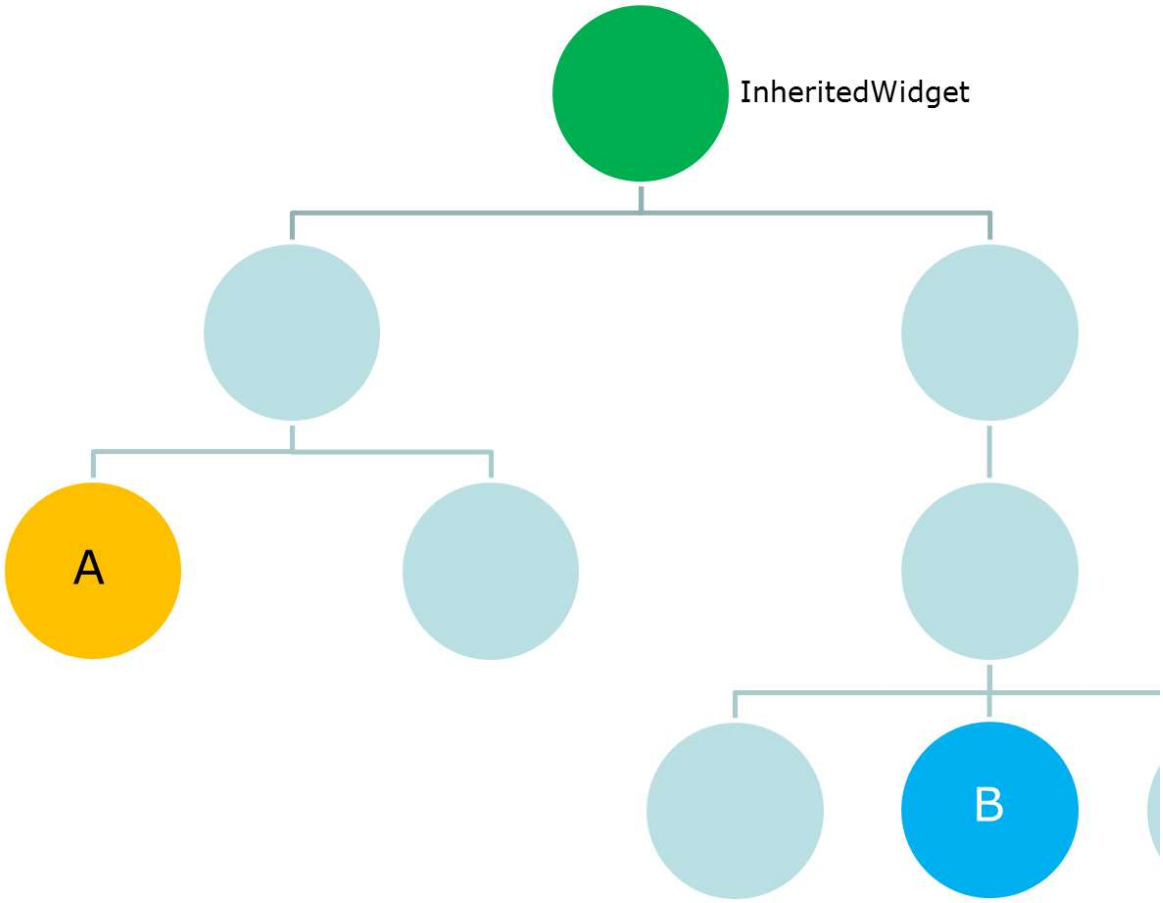
How does a child get access to the data of the InheritedWidget?

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 = MyInheritedWidget.of(context);

      ///
      /// From this moment, the widget can use the data, exposed by the MyInheritedWidget
      /// by calling:  inheritedWidget.data
      ///
      return new Container(
         color: inheritedWidget.data.color,
      );
   }
}
```

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 MyInheritedWidgetState();

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

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

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

  /// Helper method to add an Item
  void addItem(String reference){
    setState((){
      _items.add(new Item(reference));
    });
  }

  @override
  Widget build(BuildContext context){
    return new _MyInherited(
      data: this,
      child: widget.child,
    );
  }
}

class MyTree extends StatefulWidget {
  @override
  _MyTreeState createState() => new _MyTreeState();
}

class _MyTreeState extends State<MyTree> {
  @override
  Widget build(BuildContext context) {
    return new MyInheritedWidget(
      child: new Scaffold(
        appBar: new AppBar(
          title: new Text('Title'),
        ),
        body: new Column(
          children: <Widget>[
            new WidgetA(),
            new Container(
              child: new Row(
                children: <Widget>[
                  new Icon(Icons.shopping_cart),
                  new WidgetB(),
                  new WidgetC(),
                ],
              ),
            ),
          ],
        ),
      ),
    );
  }
}

class WidgetA extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final MyInheritedWidgetState state = MyInheritedWidget.of(context);
    return new Container(
```

```
      child: new RaisedButton(
        child: new Text('Add Item'),
        onPressed: () {
          state.addItem('new item');
        },
      ),
    );
  }
}

class WidgetB extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final MyInheritedWidgetState state = MyInheritedWidget.of(context);
    return new Text('${state.itemsCount}');
  }
}

class WidgetC extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return new Text('I am Widget C');
  }
}
```

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 *context*.

```
  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:

1. A call is made to the *addItem* method of *MyInheritedWidgetState*
2. *MyInheritedWidgetState.addItem* method adds a new Item to the List
3. *setState()* is invoked in order to rebuild the *MyInheritedWidget*
4. A new instance of *_MyInherited* is created with the new content of the List
5. *_MyInherited* records the new *State* which is passed in argument (*data*)
6. As an *InheritedWidget*, it checks whether there is a need to *notify* the *consumers* (answer is true)
7. It iterates the whole list of *consumers* (here Widget A and Widget B) and requests them to rebuild
8. 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 rebuild = true]){
    return (rebuild ? context.inheritFromWidgetOfExactType(_MyInherited) as _MyInherited
                    : context.ancestorWidgetOfExactType(_MyInherited) as _MyInherited).data;
  }
```

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.of(context, false);
    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 contexts 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**.

In a next article I will introduce the notion of **Notifiers / Listeners** which is also very interesting to use in the context of **State** and the way of conveying data.

So, stay tuned and happy coding.

Didier,

Share:

ALSO ON DIDIERBOELENS

StatefulWidget
2 years ago • 2 comments
Flutter - How to interact with or have interactions between several ...

Is a Widget ...
3 years ago • 4 comments
How to know if a Widget, part of a Scollable, is visible? Question Lately ...

ScopedModel
3 years ago • 1 comment
BLoC, Scoped Model, Redux... Comparaison et quand les utiliser? ...

Cas ...
4 years ago • 1 comment
BLoC, Reactive Programming, Streams - Cas practiques ...

Points to
3 years ago
Tour of the package, p and to care

didierboelens Comment Policy
All comments are welcome. Non-relevant comments will be removed.

30 Comments didierboelens Disqus' Privacy Policy Login
Favorite 13 Tweet Share Sort by Best
Join the discussion...
LOG IN WITH
OR SIGN UP WITH DISQUS
Name

Preslav Rachev • 4 years ago • edited
Hi Didier, a very helpful article. I just have one observation, however. Correct me if I am wrong, but is the following statement 100% true?

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

In theory, nothing stops the stateless widget from propagating an event, or changing some globally shared observer, which in turn causes a stateful widget higher up the hierarchy to redraw the entire subtree. Am I correct?

3 • Reply • Share ›

boeledi Mod → Preslav Rachev • 4 years ago

Hi Preslav,

You are right but the rationale behind my statement was to highlight the fact that a Stateless Widget is immutable, does not contain any state and should not try to force a rebuilding of itself. As opposed to a StatefulWidget, it is "naturally" not possible to force a Stateless Widget to rebuild itself (however, it is still possible but in 99.99% of the cases, it is possibility not used). Also, a Stateless Widget depends on others to be rebuilt.

Regards,

Didier

1 • Reply • Share ›

Ale Digitale • 4 years ago

Great article Didier, keep up the good work !!!! Your explanations are wonderful. Thank you so much from Italy

2 • Reply • Share ›

boeledi Mod → Ale Digitale • 4 years ago

Thanks so much for your good words. Didier

• Reply • Share ›

Tosho Ajibade • 3 years ago

Thanks for this

1 • Reply • Share ›

John F • 4 years ago

Merci pour ces documents très très bien faits et qui me permettent d'avancer ! Clairs et vraiment utiles pour un débutant comme moi
Encore merci

JF

1 • Reply • Share ›

João Vitor Retamero • 9 months ago

Thank you Didier! This article is 3 years old but is one of the few that explained InheritedWidget so cleanly and with helpful examples.

So, considering the age of the article, is this the simplest way to work with state in a Flutter application?

• Reply • Share ›

boeledi Mod → João Vitor Retamero • 9 months ago

Hi João,

You are right, this article is already 3 years old but is still valid as, from that perspective, Flutter did not change drastically. However, this article is not about "State Management" as you can read all around but only about:
1. StatefulWidget: widget that needs to keep reference of something in order to be able to work (e.g. consider a checkbox. Is is selected or not? To be able to show whether it is checked or not, you need a variable, which will be stored in a StatefulWidget, normally)
2. How to share an information with all the Widget, part of a tree (InheritedWidget)

State management is also used in other contexts, such as, e.g., authentication, flow, ... but this is something totally different. To handle this there are tons of solutions, good and bad, heavy and very light.
Personally, I don't use any dedicated package. My old own BlocProvider helps me a lot with the notion of streams but I am also using ValueNotifier with ValueListenableBuilder. The combination of both is just enough in 99% of the cases and makes your code very clean and totally understandable.

Have a great day

• Reply • Share ›

João Vitor Retamero → boeledi • 9 months ago

Thanks for the clarification.

I'm relatively new to Flutter and I like to extract everything I can from the framework. I found that simple solutions like your BlocProvider (which is a good project btw) or a "vanilla" InheritedWidget (like the one in this article) give us developers a peace of mind at the end of the day.

Thanks again for the content.

• Reply • Share ›

the famousdjx • a year ago

Nice and clear, first time I've got a better understanding of context after countless videos , courses and books. Thanks!

• Reply • Share ›

xangam • a year ago

Thanks a lot. This article helps me to understand a lot of things about flutter. awesome!

• Reply • Share ›

Uma Gadhvi • 2 years ago

Hello Didier, this is really helpful article thank you for sharing

• Reply • Share ›

hellboy61 • 3 years ago

Thanks..

• Reply • Share ›

W. Brandon Martin • 3 years ago

Great Article, thank you.

• Reply • Share ›

Liu Tom • 3 years ago

Great~

• Reply • Share ›

Herry X-Gianuxer • 3 years ago

Hi Didier, very helpful article, i need your permission to share this link article to my blog, i just make tutorial flutter indonesia version.

thanks.

Reply • Share ›

boeledi Mod ➚ Herry X-Gianuxer • 3 years ago

Hi Herry,

Please be my guest.

Regards,

Reply • Share ›

Michel Onwordi • 3 years ago

Hi Didier, thanks for this informative article, it helped me understand these concepts really well!

Reply • Share ›

Tim Freebern II • 4 years ago

Top notch article. Been really pushing myself to understand Flutter architecture, Widget trees, State, etc as I build various personal projects.

Reply • Share ›

baeharam • 4 years ago

Very good post! Can I translate into korean to post on my blog?

Reply • Share ›

boeledi Mod ➚ baeharam • 4 years ago

Yes of course with pleasure

Reply • Share ›

winson • 4 years ago

Thanks a lot, your article let me clarify the state details and solved many of my questions!!!

Reply • Share ›

badr ahmad • 4 years ago

Hi Mr Didier can you tell me the difference between these both examples (Navigator.push(context)) and(Theme.of(context)) please? what is that (of) method? and when we use context?thanks

Reply • Share ›

boeledi Mod ➚ badr ahmad • 4 years ago

Little question that could lead to a very long answer...

In short:

The "of(BuildContext context)" method is by convention, created as a static method aimed at returning an instance of a specific Widget type (or state, or data, ...), which is retrieved by the position of a 'context', passed in argument.

It is usually combined with the notion of InheritedWidget.

[general answer] Suppose you have an inheritedWidget which exposes the method "of". Another Widget, part of the Widget tree (which the inheritedWidget is one of the ancestors) will invoke the "of" method, passing its own "context". The "of" method will look for the "closest" instance of the InheritedWidget and will return it (or some data it contains, ...).

For example, a call to "Theme.of(context)" or "MediaQuery.of(context)" or "Navigator.of(context)" will look for the "closest" instance of (Theme, MediaQuery, Navigator...) and will return the data it refers to, to the caller.

In hope this answer will be of any help.

Kind Regards,

Reply • Share ›

Jihua Huang • 4 years ago

Thank you very much for your wonderful article! Looking forward to reading the article about notion of Notifiers / Listeners.

Reply • Share ›

Sebastian • 4 years ago

Didier great article.
I'm wondering about the "Special note for Routes, Dialogs…" section.

Do you think that may be why I'm having this error? https://stackoverflow.com/q...

It's really driving me crazy and I haven't been able to resolve it yet.

Reply • Share ›

boeledi Mod ➚ Sebastian • 4 years ago

Hi Sebastian, I answer in stackoverflow

However, additional thoughts:

1) I would rather not instantiate the bloc inside the didChangeDependencies() method, since the latter is called every single time you rebuild (and mainly when you deal with InheritedWidgets). A better solution would be to initialize the bloc, inside the initState()... but with a little trick.

In fact, as I explained, the context is not yet available at the initState level. Therefore, we use to use this trick: to request the framework to call us as soon as the first "build" is done, via the WidgetsBinding.instance.addPostFrameCallback((_){bloc = LoginBlogProvider.of(context);}); At that instant, the context is available.

2) another way would be to get it at time of building...

```
Widget build(BuildContext context){
var bloc = LoginBlocProvider.of(context);
return Scaffold...
}
```

Anyways, could you please share the code of your LoginBlocProvider so that I can further have a look at it?

Regards,

Didier

Reply • Share ›

• Reply • Share ›

anandguptastar • 4 years ago
This is really good topic with detailed description, which is what I was looking for. Thanks.
• Reply • Share ›

Pavel _ • 4 years ago
Thank you, I really enjoyed this article :)

didierboelens.com assumes no responsibility or liability for any errors or omissions in the content of this site. The information contained in this site is provided on an 'as is' basis with no guarantees of completeness, accuracy, usefulness or timeliness.

Quick Links

- About
- Blog / Articles
- Contact
- Services

Contact Info

mail@didierboelens.com

Copyright © 2020 - Didier Boelens