What Exactly is BuildContext in Flutter?

`BuildContext` is present everywhere in Flutter. It is used in the build method of `StatelessWidget`s and `State` classes. It is also used in different scenarios like

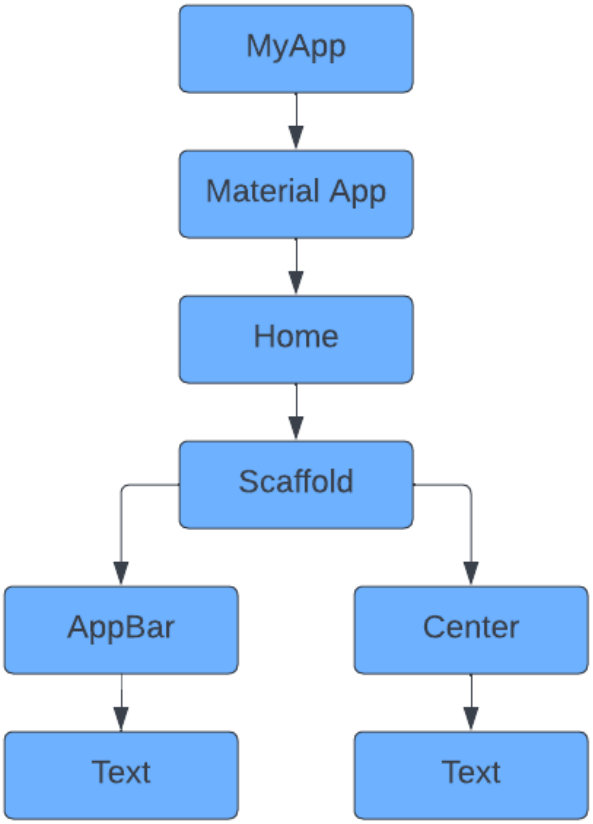
- When calling `Navigator.of(context)` to navigate to a different screen.
- When calling `ScaffoldMessenger.of(context)` to show a Snackbar.
- When calling `Theme.of(context)` to obtain theme-related information.

Even with its extensive usage, a lot of developers especially beginners struggle to understand what `BuildContext` is and what it does exactly. In this article, we'll break down `BuildContext`, its usefulness, and why it exists.

Widget Tree

In Flutter, almost everything is a widget. Widgets are building blocks that are used to create user interfaces. While some widgets are visible to the user, for instance, the `Text` and `Image` widgets. Others like `Navigator` and `Builder` are not. These building blocks (widgets) are nested within each other to create what is known as the **Widget tree**. A flutter application can be thought of as a deeply nested widget tree with different branches.

A flutter app could have the following widget tree structure:



From the image above, The `MyApp()` widget is the root widget. It is the parent widget of the `MaterialApp()` widget which makes `MaterialApp()` its child widget. This relationship also applies to other widgets in the tree.

Element Tree

The widget tree is not the only tree in Flutter, two other trees exist parallel to it. They are

- The `Element` tree
- The `RenderObject` tree

I'll only talk about the `Element` tree so as not to deviate from the focus of this article. You can read more about `RenderObject` here.

You may be wondering what an element tree is. What even is an element? and how any of this relates to `BuildContext`. I'll get to that shortly.

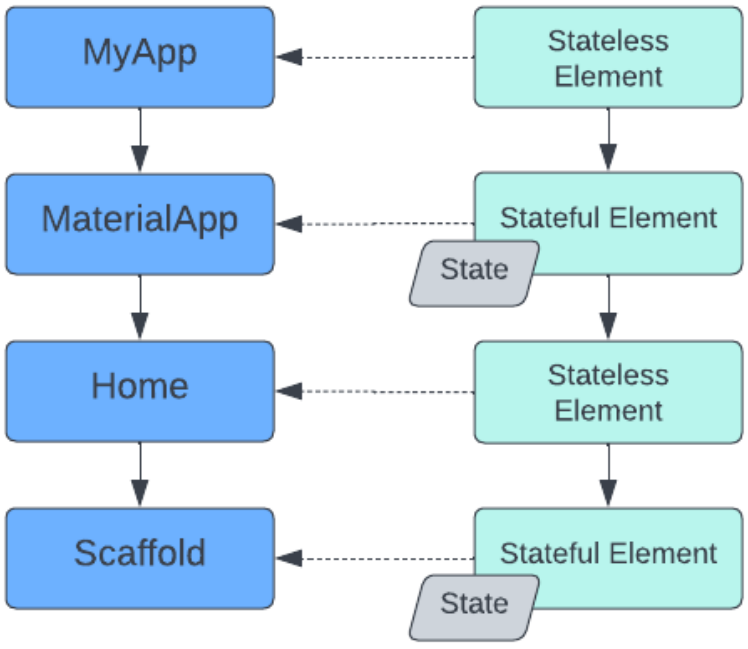
Every widget in Flutter has a `createElement()` method, which returns either a `StatelessElement` or `StatefulElement` depending on whether the widget is stateless or

stateful. When you run a flutter application, this method is called internally by the Flutter framework as it traverses through each widget in the tree, simultaneously creating an `Element` object, which is added to the `Element` tree. This object holds a reference to a widget and state (for `StatefulElement`), representing the location of that widget in the tree.

Think of the widget tree as a blueprint for creating the element tree. While the blueprint is described by developers, the Flutter framework uses the blueprint to create and lay out individual elements.

An `Element` object holds a reference to its parent element, creating a link between each element in the tree. Unlike elements, widgets do not actually know about each other. The parent-child relationship between elements is what actually creates a link between each item in the tree. An `Element` object may remain unchanged while the widget it references can be disposed of and recreated over and over again.

The diagram below shows an `Element` tree:



BuildContext

`BuildContext` is an abstraction, an interface that is implemented by the `Element` class. In essence, `Element` is a concrete implementation of `BuildContext`. The `BuildContext` interface was used by the flutter team to avoid direct manipulation of `Element` objects. In summary, an `Element` is a `BuildContext` object.

Static .of(context) method

A common usage of `BuildContext` is passing it as an argument to the static `.of(context)` method like when retrieving theme information. Internally, this method performs a look-up for the element that holds the referenced state or widget. For instance, when calling `Navigator.of(context)`, a look-up for the nearest element that holds a `NavigatorState` object is performed.

Common Wrong Assumptions about BuildContext

A lot of wrong assumptions are made by developers when working with `BuildContext`. Some of these include:

- **Assuming that Widgets Returned by the build method all share the same BuildContext**

This can be tricky since every function that takes `BuildContext` as a parameter often uses context as the parameter name. Every widget has its own `BuildContext` and when explicitly defined like with builder functions, any line of code referencing context in that scope is actually using the parent's widget `BuildContext`.

The code below explains this better:

```
@override
  Widget build(BuildContext context) {
    return Scaffold(
        appBar: AppBar(),
        body: Builder(
          builder: (context) {
            return Center(
              child: Column(
                mainAxisAlignment: MainAxisAlignment.center,
                children: [
                  Text(
                    'Text One',
                    //Uses the outer Builder context
                    style: Theme.of(context).textTheme.headline2,
                  ),
                  const SizedBox(height: 50),
                  Theme(
                    data: ThemeData(
                        textTheme: const TextTheme(
```

```
                        headline2: TextStyle(color: Colors.red))),
                child: Builder(builder: (context) {
                  return Text(
                    'Text Two',
                    //Uses the immediate Builder context
                    style: Theme.of(context).textTheme.headline2,
                  );
                }),
              ),
            ],
          ),
        );
      },
    ));
}
```

In the code above, the second `Text` displays a red text because the `BuildContext` of its immediate `Builder` parent is used to retrieve the `TextTheme`. Since the Builder widget is a child of the `Theme` widget where the new `TextTheme` is defined, descendant widgets are able to access the new `TextTheme`. While the first `Text` widget displays black text as it uses the `BuildContext` of the outer `Builder` widget.

- **Assuming that** BuildContext **defined at a widget level can be used at the same level**

Another common assumption is that the `BuildContext` of a widget can be used at the same level that it was declared. This often happens when working with Provider and other InheritedWidgets.

Using our previous code, this assumption would lead to making the second `Text` widget a direct child of the theme widget as shown below:

```
Theme(
    data: ThemeData(
      textTheme: const TextTheme(
        headline2: TextStyle(color: Colors.red),
      ),
    ),
    child: Text(
      'Text Two',
       style: Theme.of(context).textTheme.headline2,
    ),)
```

The code above tries to obtain the newly defined `TextTheme` but because the `Text` widget is now using the `BuildContext` of the outer `Builder`, the default `TextTheme` is returned instead. A solution to this is wrapping the `Text` widget with a `Builder` widget like in the previous code snippet. The `Builder` widget provides a `BuildContext` that is a child of the parent `Theme` widget context, which makes the new theme available to the `Text` widget.

Conclusion

`BuildContext` is an important Flutter concept. Understanding it can help in architecting robust Flutter applications and gives a new perspective into the inner workings of the framework.

Author: Nsikak Isaac