Chapter 4

Understanding Widgets

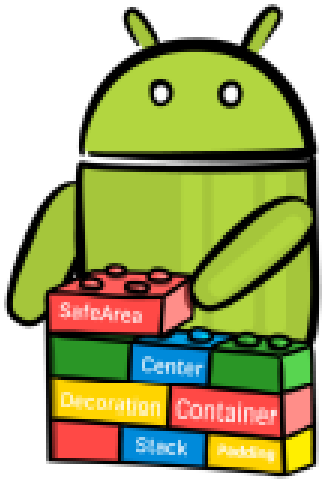
Widget Theory. We'll explore:

- Widgets
- Widget rendering
- Flutter Inspector
- Types of widgets
- Widget lifecycle

What is a widget?

1

- A widget is a building block for your user interface.
- Like Legos, you can mix and match widgets to create something amazing.
- A widget is a blueprint for displaying your app **state.**



What is a widget?

- You can think of widgets as a function of UI. Given a state, the build() method of a widget constructs the widget UI.

$$\underset{\text{Screen}}{UI} = \underset{\text{Build}}{f}(state)$$

Example: Card2

Example: Card2

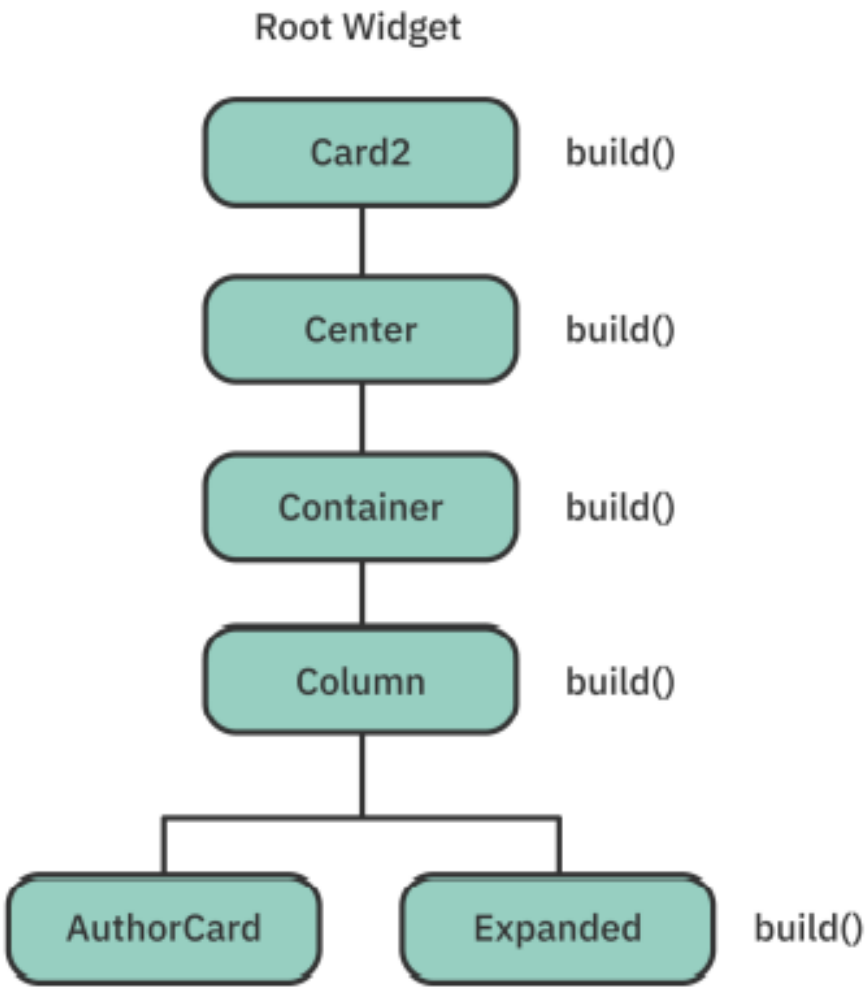
Recall that the card consists of the following:

- **Container widget**: Styles, decorates and positions widgets.
- **Column widget**: Displays other widgets vertically.
- **AuthorCard custom widget**: Displays the author's information.
- **Expanded widget**: Uses a widget to fill the remaining space.
- **Stack widget**: Places widgets on top of each other.
- **Positioned widget**: Controls a widget's position in the stack.

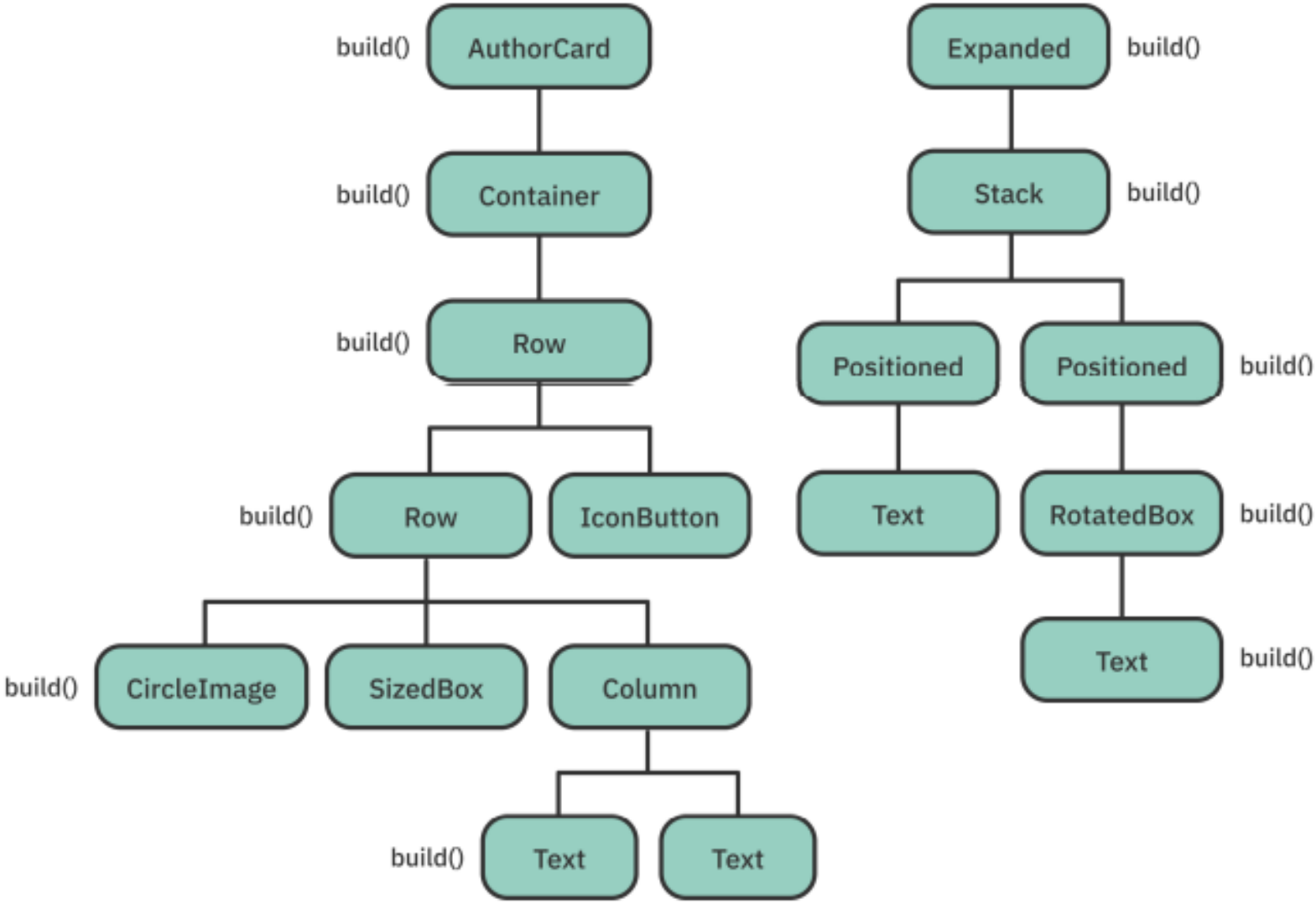
Widget tree:

- Every widget contains a build() method.
- In this method, you create a UI composition by nesting widgets within other widgets.
- This forms a **tree-like data structure**
- Each widget can contain other widgets, commonly called **children**.
- The widget tree provides a blueprint that describes how you want to lay out your UI.

Widget tree:


Root Widget
Card2
build()
Center
build()
Container
build()
Column
build()
AuthorCard
Expanded
build()

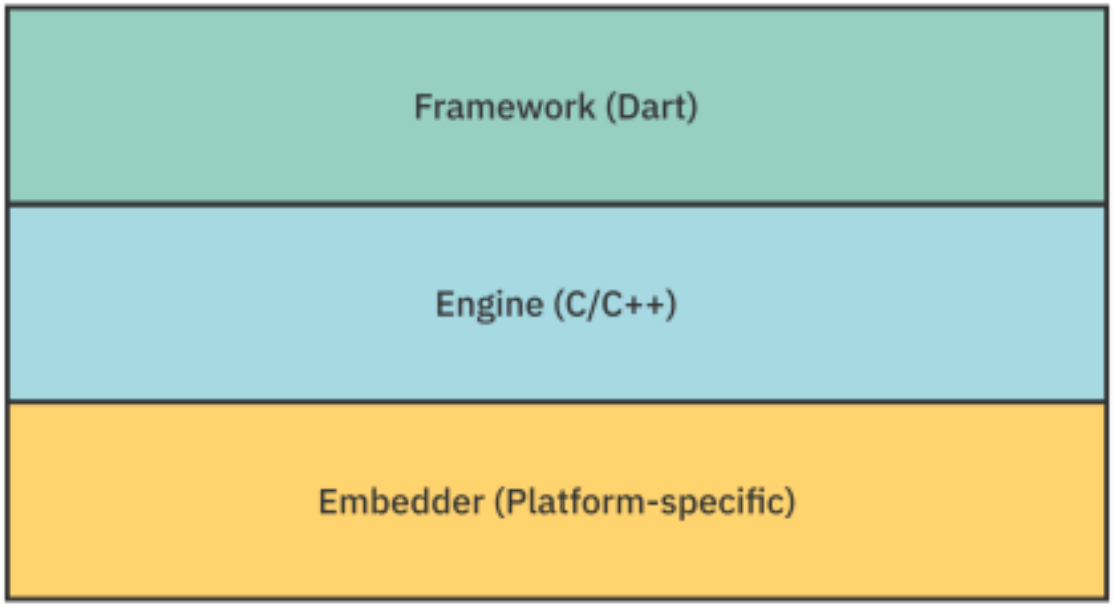
Widget tree:


build()
AuthorCard
build()
Container
build()
Row
build()
Row
IconButton
build()
CircleImage
SizedBox
Column
build()
Text
Text
Expanded
build()
Stack
build()
Positioned
Positioned
build()
Text
RotatedBox
build()
Text
build()

Rendering widgets:

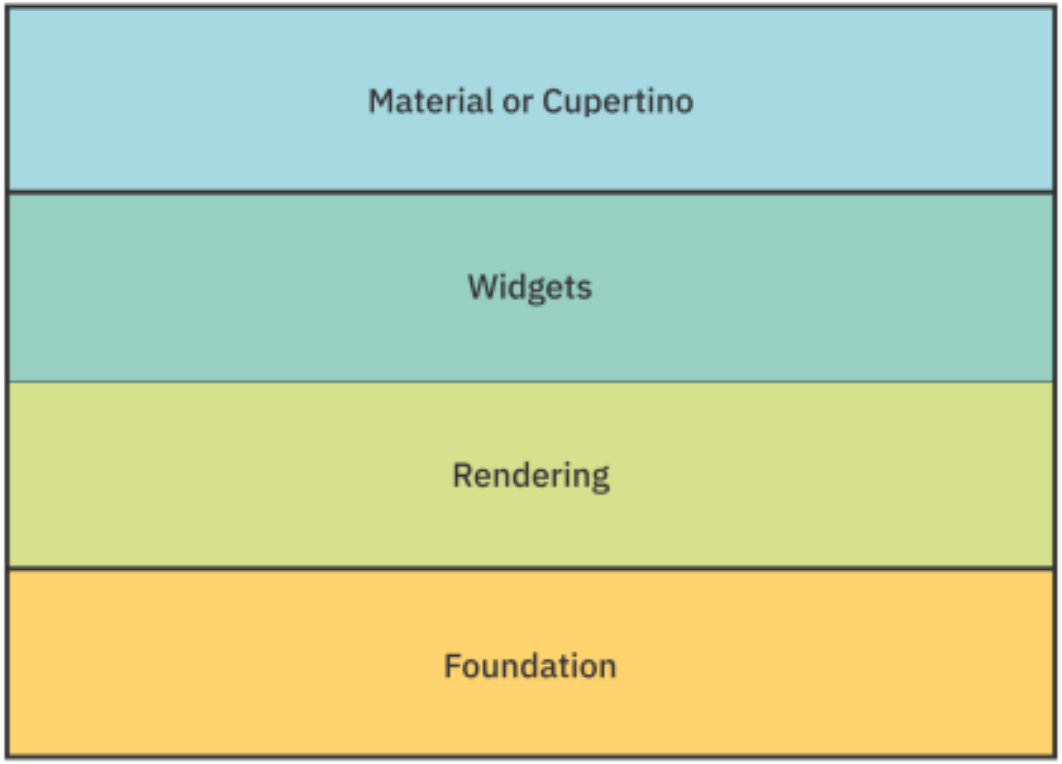
2

Flutter's architecture contains three layers



Rendering widgets:

We can break Framework layer into four parts



Rendering widgets:

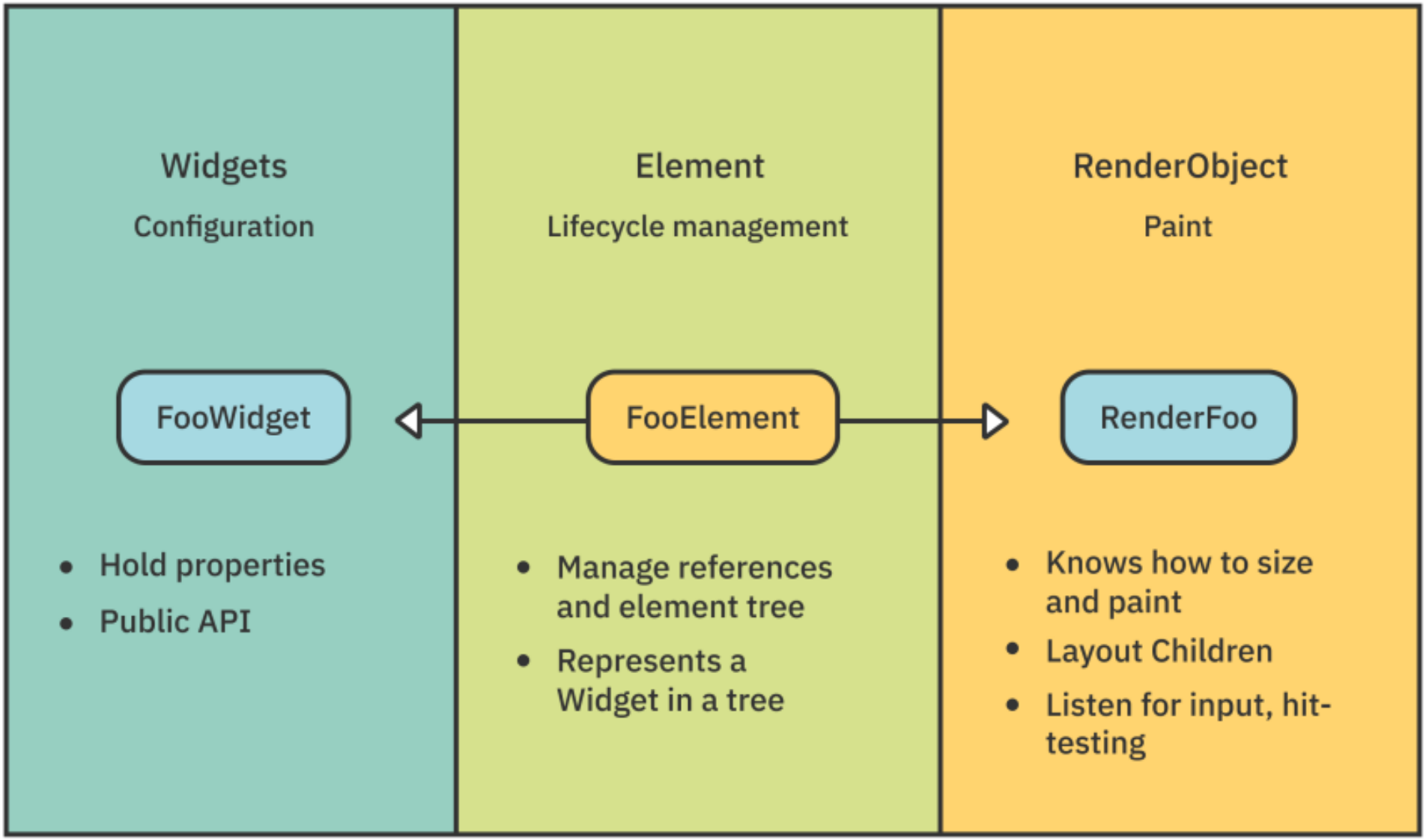
- **Material** and **Cupertino** are UI control libraries built on top of the widget layer. They make your UI look and feel like Android and iOS apps, respectively.
- The **Widgets** layer is a composition abstraction on widgets. It contains all the primitive classes needed to create UI controls.
- The **Rendering** layer is a layout abstraction that draws and handles the widget's layout.
- **Foundation**, also known as the **dart:ui** layer, contains core libraries that handle animation, painting and gestures.

Three trees:

Flutter's framework actually manages not one, but three trees in parallel.

- **Widget** Tree: The public API or blueprint for the framework. Developers usually just deal with composing widgets.
- **Element** Tree: Manages a widget and a widget's render object. For every widget instance in the tree, there is a corresponding element.
- **RenderObject** Tree: Responsible for drawing and laying out a specific widget instance. Also handles user interactions, like hit-testing and gestures.

Three trees:


Widgets
Configuration
FooWidget
Hold properties
Public API
Element
Lifecycle management
FooElement
Manage references and element tree
Represents a Widget in a tree
RenderObject
Paint
RenderFoo
Knows how to size and paint
Layout Children
Listen for input, hit-testing

Flutter Inspector:

3

The Flutter Inspector has four key benefits. It helps us:

- **Visualize** your widget tree.
- **Inspect** the properties of a specific widget in the tree.
- **Experiment** with different layout configurations using the Layout Explorer.
- **Enable** slow animation to show how your transitions look.

Types of widgets:

4

- There are three major types of widgets: **Stateless**, **Stateful** and **Inherited**.
- All widgets are immutable but some have state attached to them using their element.

Stateless widgets:

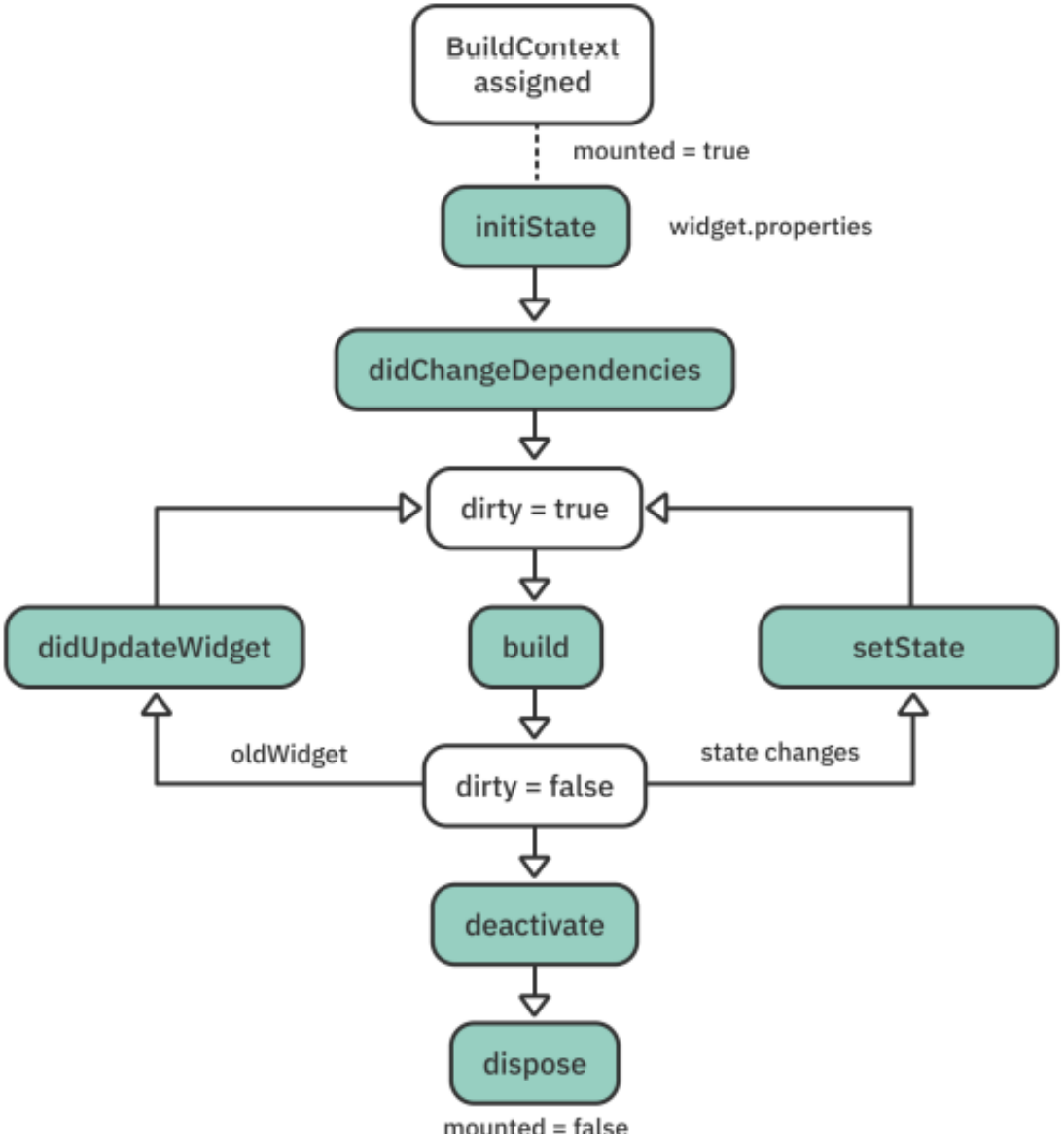
- When your properties don't need to change over time, it's generally a good idea to start with a **stateless widget**.
- The **lifecycle** of a **stateless widget** starts with a constructor, which you can pass parameters to, and a **build**() method, which you override.
- The visual description of the widget is determined by the **build**() method.

Stateful widgets:

- **Stateful widgets** preserve state, which is useful when parts of our UI need to **change dynamically**.
- Stateful widgets store their mutable state in a **separate State class**.
- That's why every stateful widget must override and implement **createState**().

State object lifecycle:

5



State object lifecycle:

1. When you assign the build context to the widget, an internal flag, mounted, is set to true. This lets the framework know that this widget is currently on the widget tree.

2. **initState**() is the first method called after a widget is created. This is similar to onCreate() in Android or viewDidLoad() in **iOS**.

3. The first time the framework builds a widget, it calls didChangeDependencies() after **initState**(). It might call didChangeDependencies() again if your state object depends on an inherited widget that has changed. There is more on **inherited widgets** below.

State object lifecycle:

4. Finally, the framework calls build() after didChangeDependencies().

   This function is the most important for developers because it's called every time a widget needs rendering. Every widget in the tree triggers a build() method recursively, so this operation has to be very fast.

5. The framework calls didUpdateWidget(_) when a parent widget makes a change or needs to redraw the UI.

   When that happens, you'll get the oldWidget instance as a parameter so you can compare it with your current widget and do any additional logic.

State object lifecycle:

6. Whenever you want to modify the state in your widget, you call setState(). The framework then marks the widget as dirty and triggers a build() again.

7. When you remove the object from the tree, the framework calls deactivate(). The framework can, in some cases, reinsert the state object into another part of the tree.

8. The framework calls dispose() when you permanently remove the object and its state from the tree.

   This method is very important because you'll need it to handle memory cleanup, such as unsubscribing streams and disposing of animations or controllers.