Flutter Apprentice 2021 Second Edition Book

23 Nov 2025

MOBILE APPLICATIONS IT 319 WEEK 7

IT DEPT.

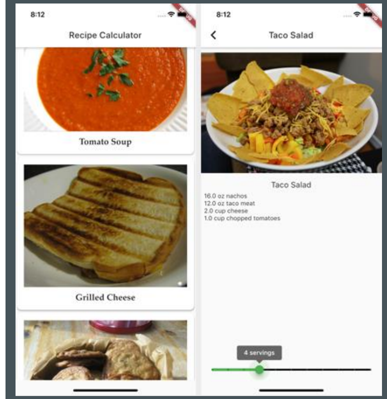
TIU3RD GRADE

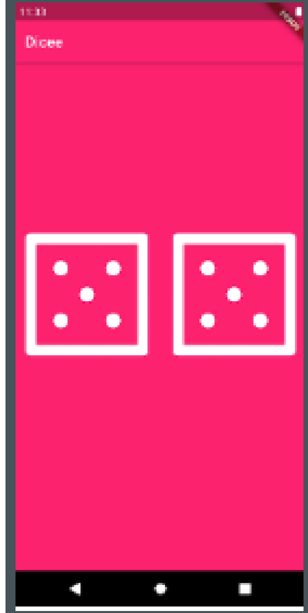
Chapter 4 – Undestanding Widgets

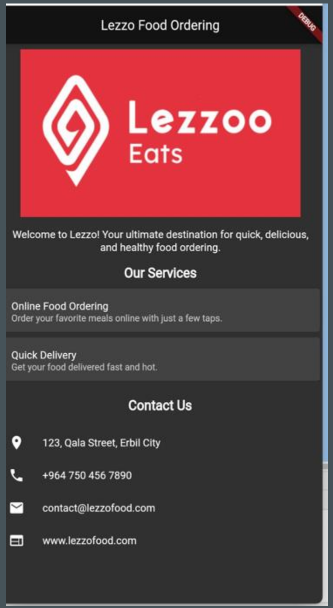
Mastering Widgets, Rendering, and Intro to State Management in Flutter.

Lect. Mohammad Salim

FLUTTER APPRENTICE







Lezzoo Food Ordering

Lezzoo Eats

Welcome to Lezzo! Your ultimate destination for quick, delicious, and healthy food ordering.

Our Services

Online Food Ordering
Order your favorite meals online with just a few taps.

Quick Delivery
Get your food delivered fast and hot.

Contact Us

123, Qala Street, Erbil City

+964 750 456 7890

contact@lezzofood.com

www.lezzofood.com

Up to Date for Flutter 2.5

Bonus chapter on: Firebase & Cloud Firestore!

Flutter Apprentice

SECOND EDITION

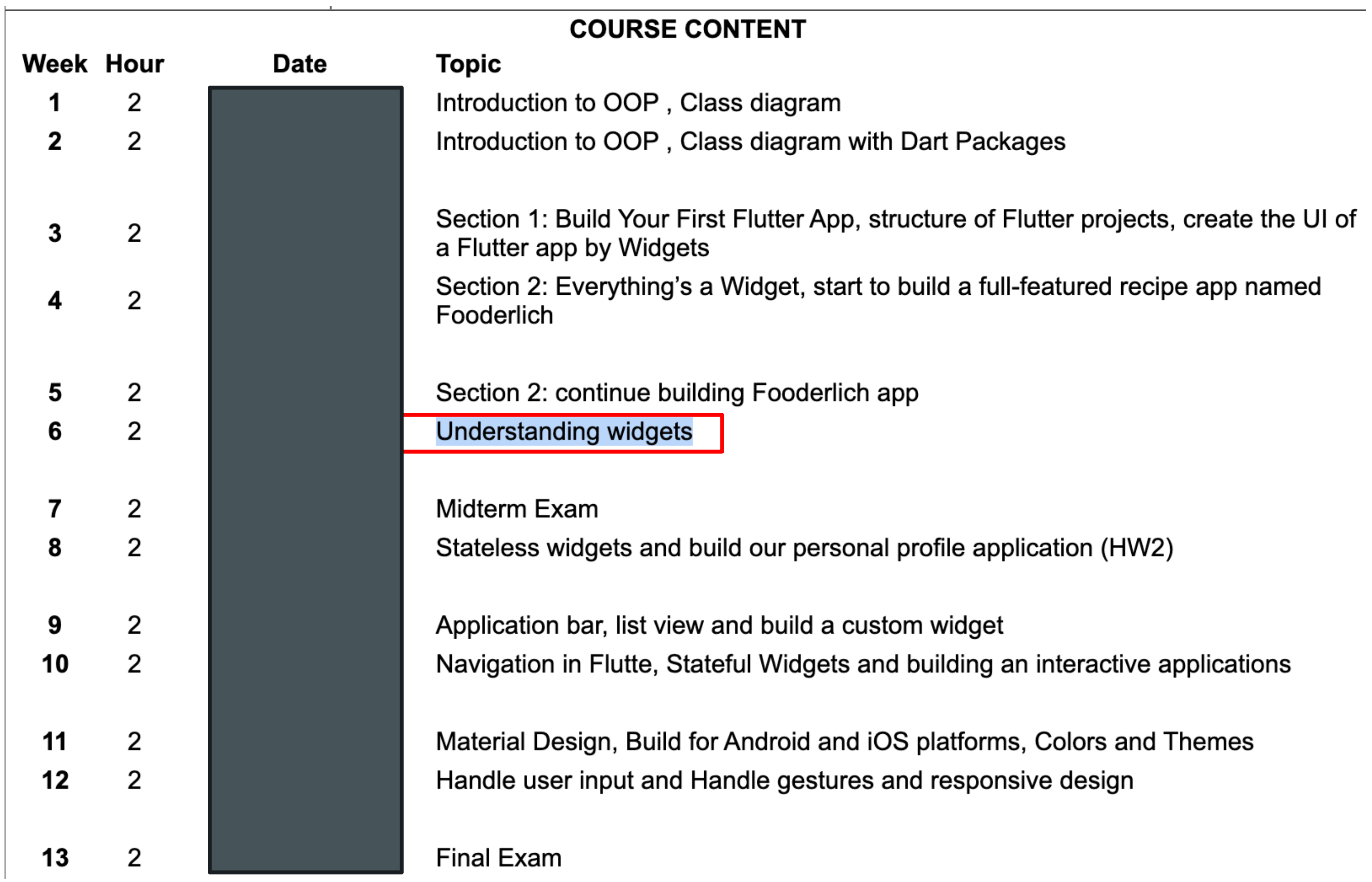
Learn to Build Cross-Platform Apps

By the raywenderlich Tutorial Team

Mike Katz, Kevin D. Moore, Vincent Ngo & Vincenzo Guzzi

COURSE CONTENT

- Flutter and OOP



COURSE CONTENT

Week	Hour	Date	Topic
1	2	[redacted]	Introduction to OOP , Class diagram
2	2	[redacted]	Introduction to OOP , Class diagram with Dart Packages
3	2	[redacted]	Section 1: Build Your First Flutter App, structure of Flutter projects, create the UI of a Flutter app by Widgets
4	2	[redacted]	Section 2: Everything's a Widget, start to build a full-featured recipe app named Fooderlich
5	2	[redacted]	Section 2: continue building Fooderlich app
6	2	[redacted]	Understanding widgets
7	2	[redacted]	Midterm Exam
8	2	[redacted]	Stateless widgets and build our personal profile application (HW2)
9	2	[redacted]	Application bar, list view and build a custom widget
10	2	[redacted]	Navigation in Flutte, Stateful Widgets and building an interactive applications
11	2	[redacted]	Material Design, Build for Android and iOS platforms, Colors and Themes
12	2	[redacted]	Handle user input and Handle gestures and responsive design
13	2	[redacted]	Final Exam

CONTENTS

CHAPTER 4: UNDERSTANDING WIDGETS

You may have heard that everything in Flutter is a widget. While that might not be absolutely true, most of the time when you're building apps, you only see the top layer: **widgets**. In this chapter, you'll dive into widget theory. You'll explore:

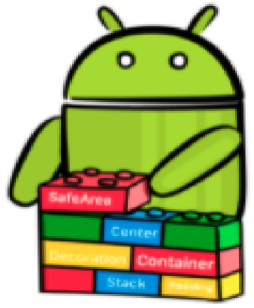
- Widgets
- Widget rendering
- Flutter Inspector
- Types of widgets
- Widget lifecycle

It's time to jump in!

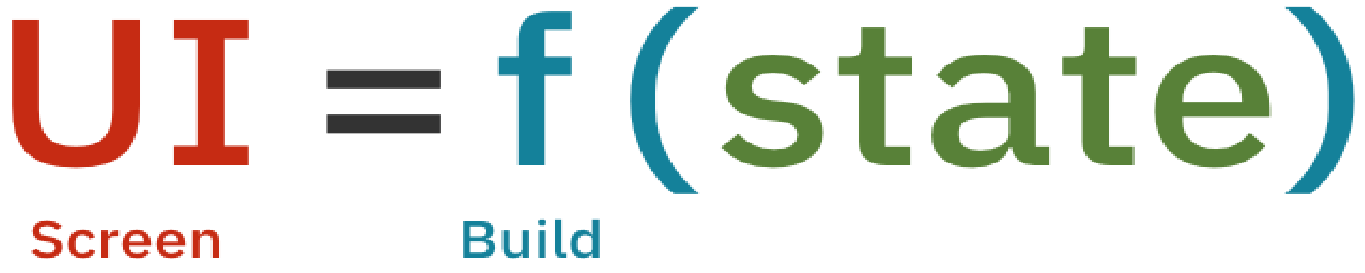
Note: This chapter is mostly theoretical. You'll make just a few code changes to the project near the end of the chapter.

WHAT IS A WIDGET?

- A **widget** is a building block for your user interface. Using widgets is like combining Legos. Like Legos, you can mix and match widgets to create something amazing.



Flutter's declarative nature makes it super easy to build a UI with widgets. A widget is a blueprint for displaying your app **state**.



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

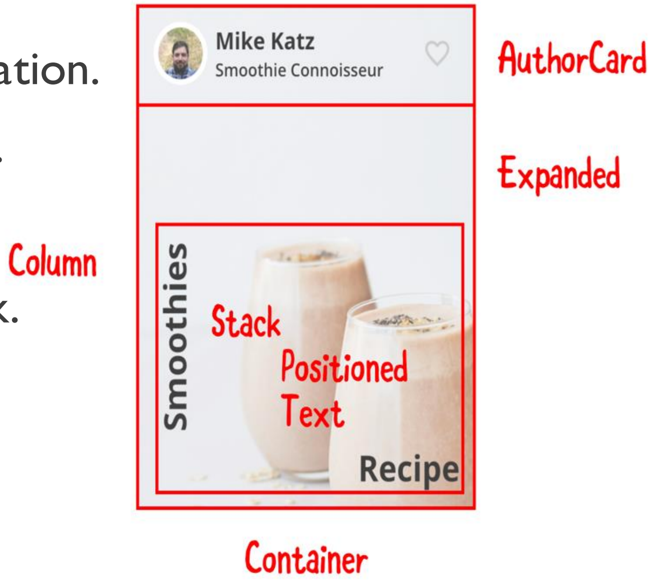
UNBOXING CARD2

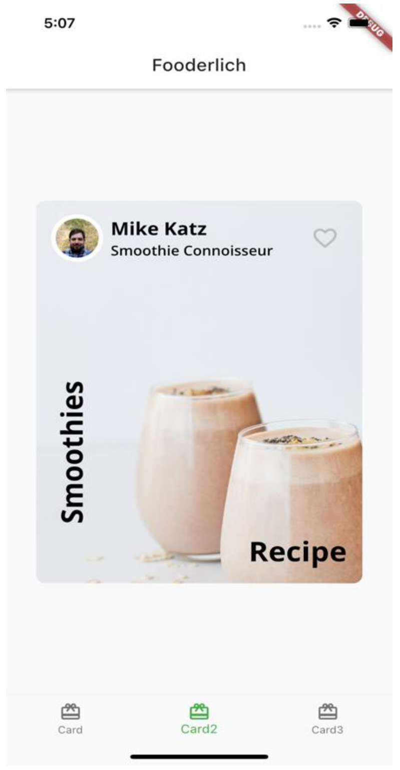
In the previous chapter, you created three recipe cards. Now, you'll look in more detail at the widgets that compose **Card2**:

Do you remember which widgets you needed to build this card?

Recall that the card consists of the following:

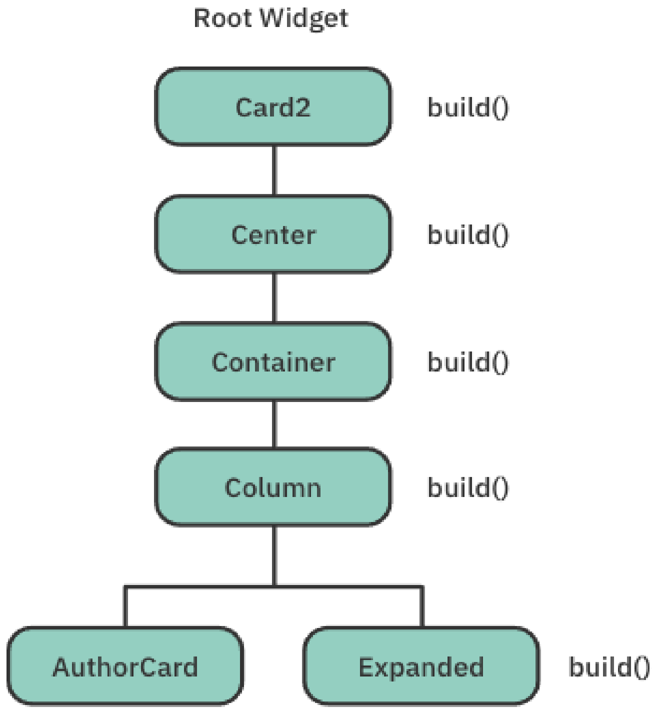
- **Container widget**: Styles, decorates and positions widgets.
- **Column widget**: Displays other widgets vertically.
- **AuthorCard custom widget**: Displays the author's information.
- **Expanded widget**: Uses a widget to fill the remaining space.
- **Stack widget**: Places widgets on top of each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





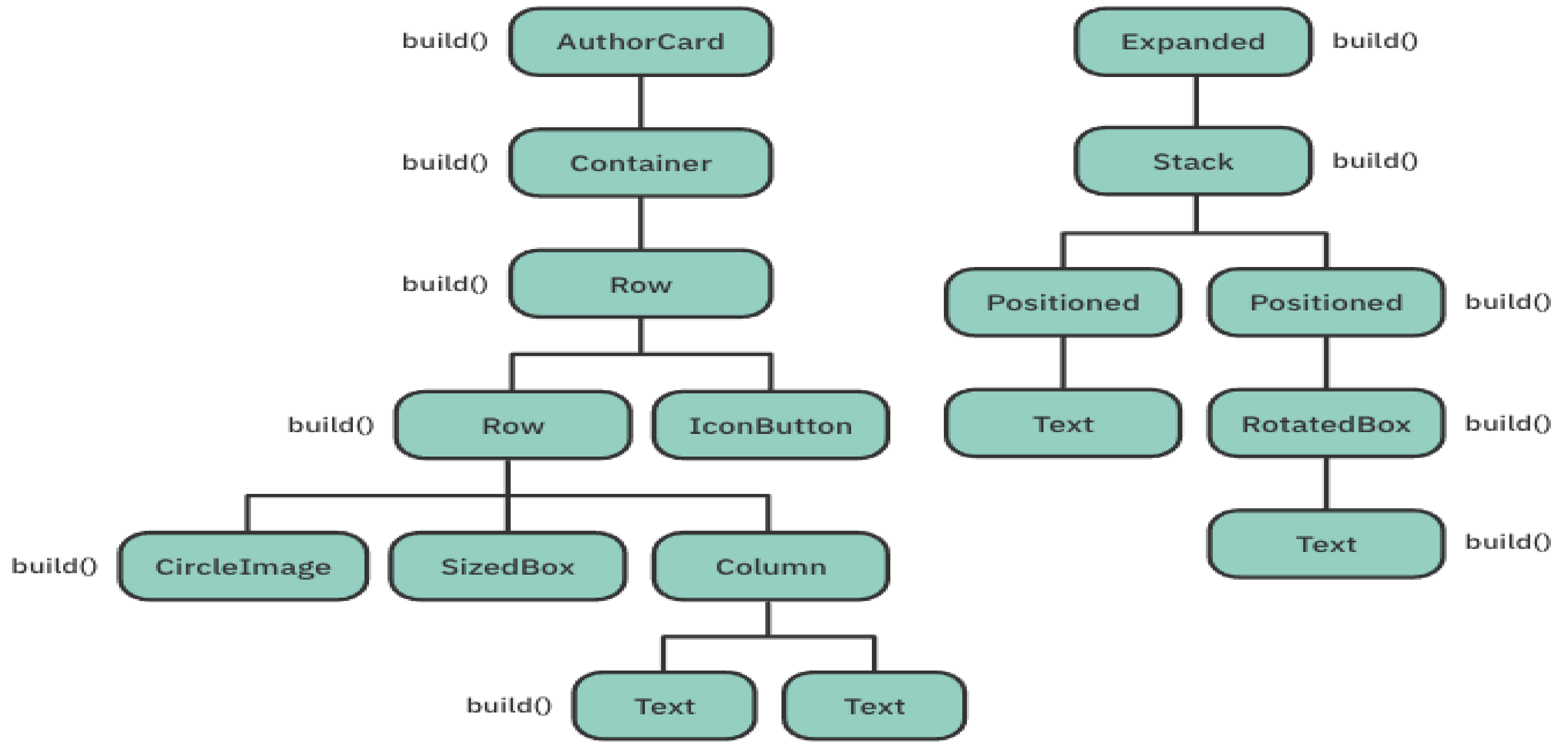
WIDGET TREES

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**. Below is a visualization of **Card2**'s widget tree:



WIDGET TREES

You can also break down `AuthorCard` and `Expanded`:



The widget tree provides a blueprint that describes how you want to lay out your UI.

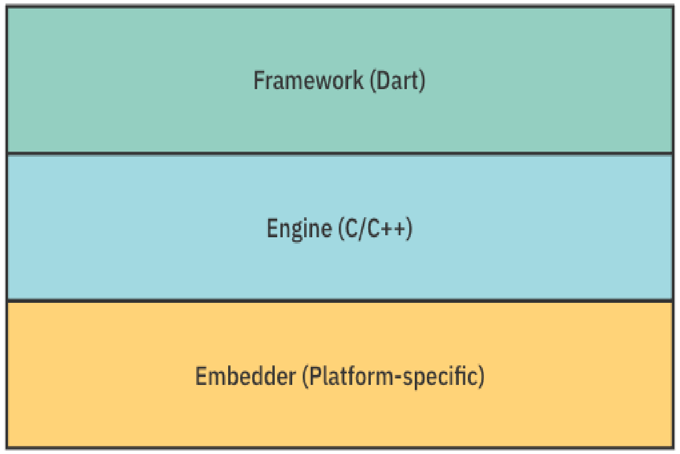
The framework traverses the nodes in the tree and calls each build() method to compose your entire UI.

LAB TASK - NESTED WIDGET TREE DESIGN AND CUSTOM WIDGET

- Work collaboratively to design and implement a **profile card** UI using a nested widget tree.
- Gain hands-on experience with Flutter widgets, widget trees, and their hierarchy.
- **Design Requirements**:
- Create a list of profile cards with the following features:
  - **Icon**: Display a circular profile Icon at the top.
  - **Name**: Add the user's name in bold.
  - **Bio or position**: Include a short bio below the name.
  - Use the **custom widget** for that.

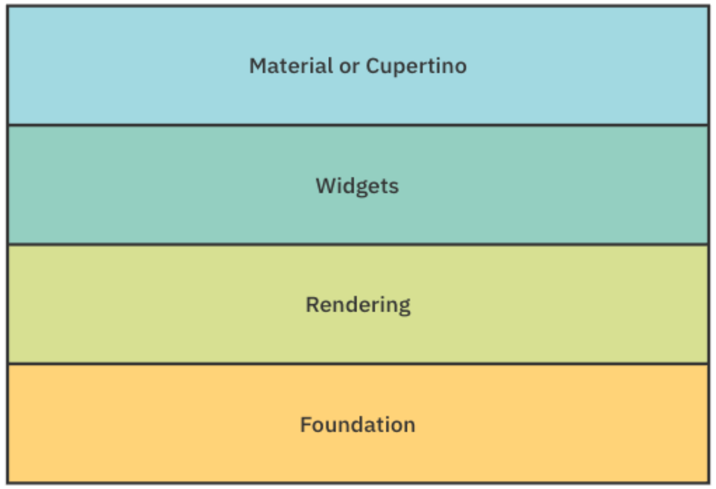
RENDERING WIDGETS

In Chapter 1, "Getting Started", you learned that Flutter's architecture contains **three layers**:



In this chapter, you'll focus on the **framework layer**. You can break this layer into **four parts**:

- **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. Check out the official documentation here: https://api.flutter.dev/flutter/widgets/widgets-library.html.
- The **Rendering** layer is a layout abstraction that draws and handles the widget's layout. Imagine having to recompute every widget's coordinates and frames manually. Yuck!
- **Foundation**, also known as the **dart:ui** layer, contains core libraries that handle animation, painting and gestures.



RENDERING WIDGETS

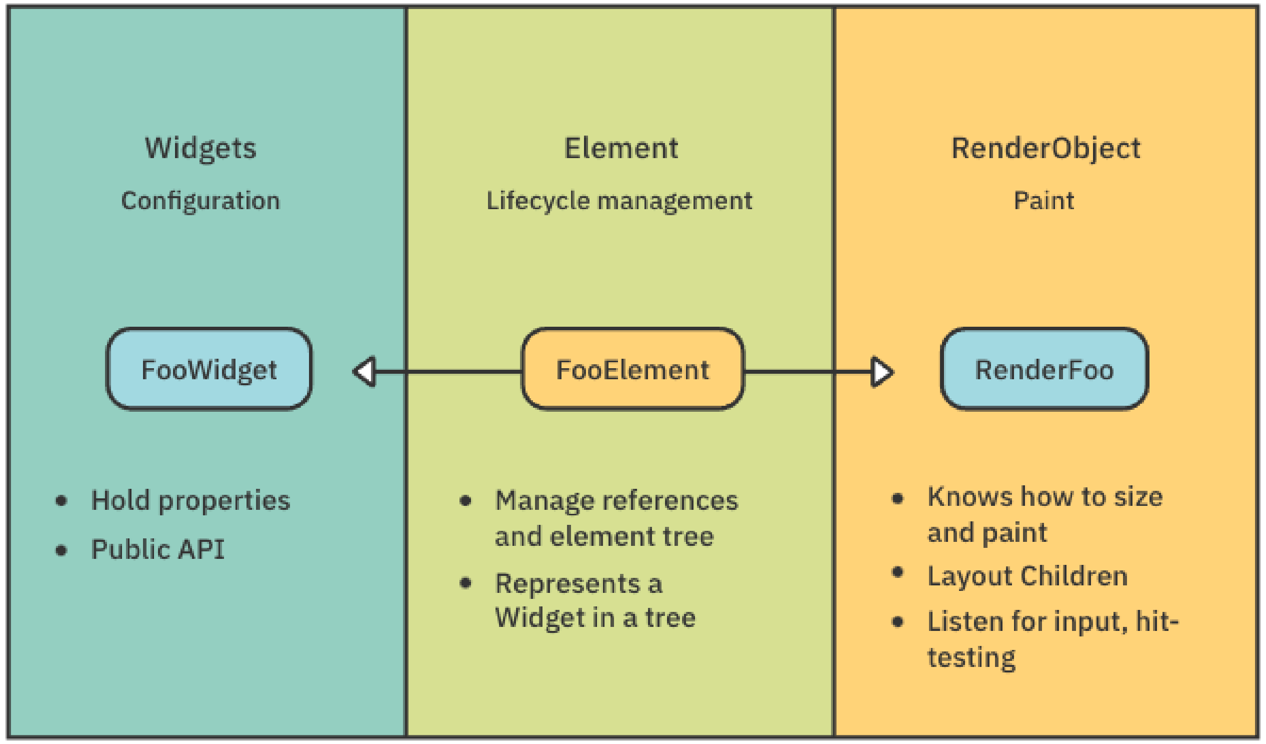
Three Trees

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

- Widget Tree
- Element Tree
- RenderObject Tree

Here's how a single widget works under the hood:

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



RENDERING WIDGETS

Types of elements

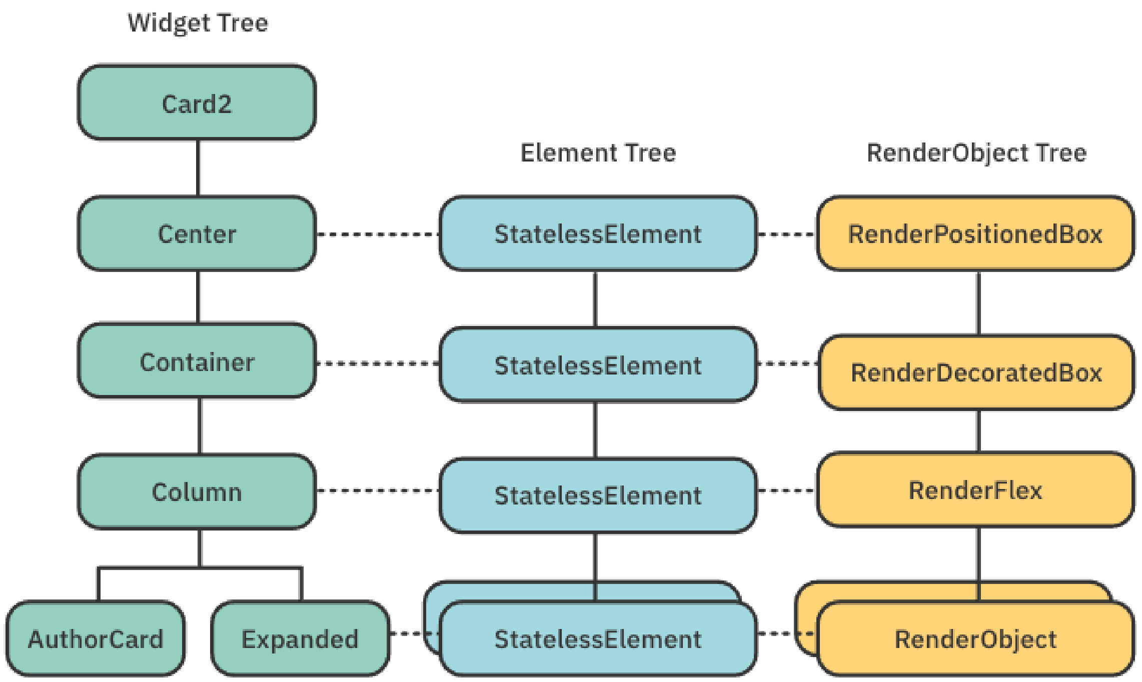
There are two types of elements:

- **ComponentElement**: A type of element that's composed of other elements. This corresponds to composing widgets inside other widgets.
- **RenderObjectElement**: A type of element that holds a render object.

You can think of **ComponentElement** as a group of elements, and **RenderObjectElement** as a single element. Remember that each element contains a render object to perform widget painting, layout and hit testing.

Example trees for Card2

The image on the right shows an example of the three trees for the **Card2** UI:



RENDERING WIDGETS

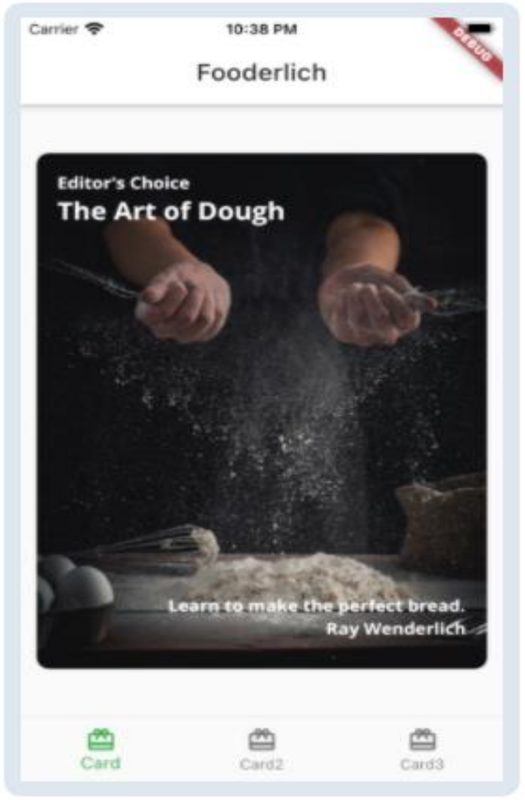
Types of elements

- As you saw in previous chapters, Flutter starts to build your app by calling runApp().
- Every widget's build() method then composes a subtree of widgets. For each widget in the widget tree, Flutter creates a corresponding **element**.
- The element tree manages each widget instance and associates a render object to tell the framework how to render a particular widget.

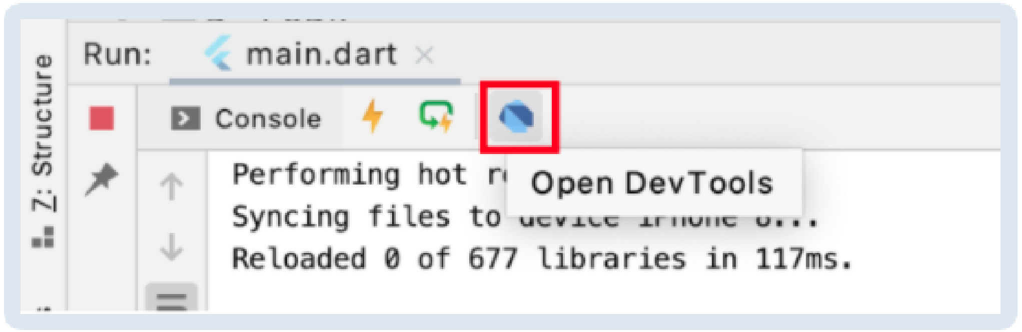
Note: For more details on Flutter widget rendering, check out the Flutter team's talk they gave in China on how to render widgets: https://youtu.be/996ZgFRENMs.

GETTING STARTED

Open the **starter** project in Android Studio, run `flutter pub get` if necessary, then run the app. You'll see the **Fooderlich** app from the previous chapter:



Next, open **DevTools** by tapping the **blue Dart** icon, as shown below:

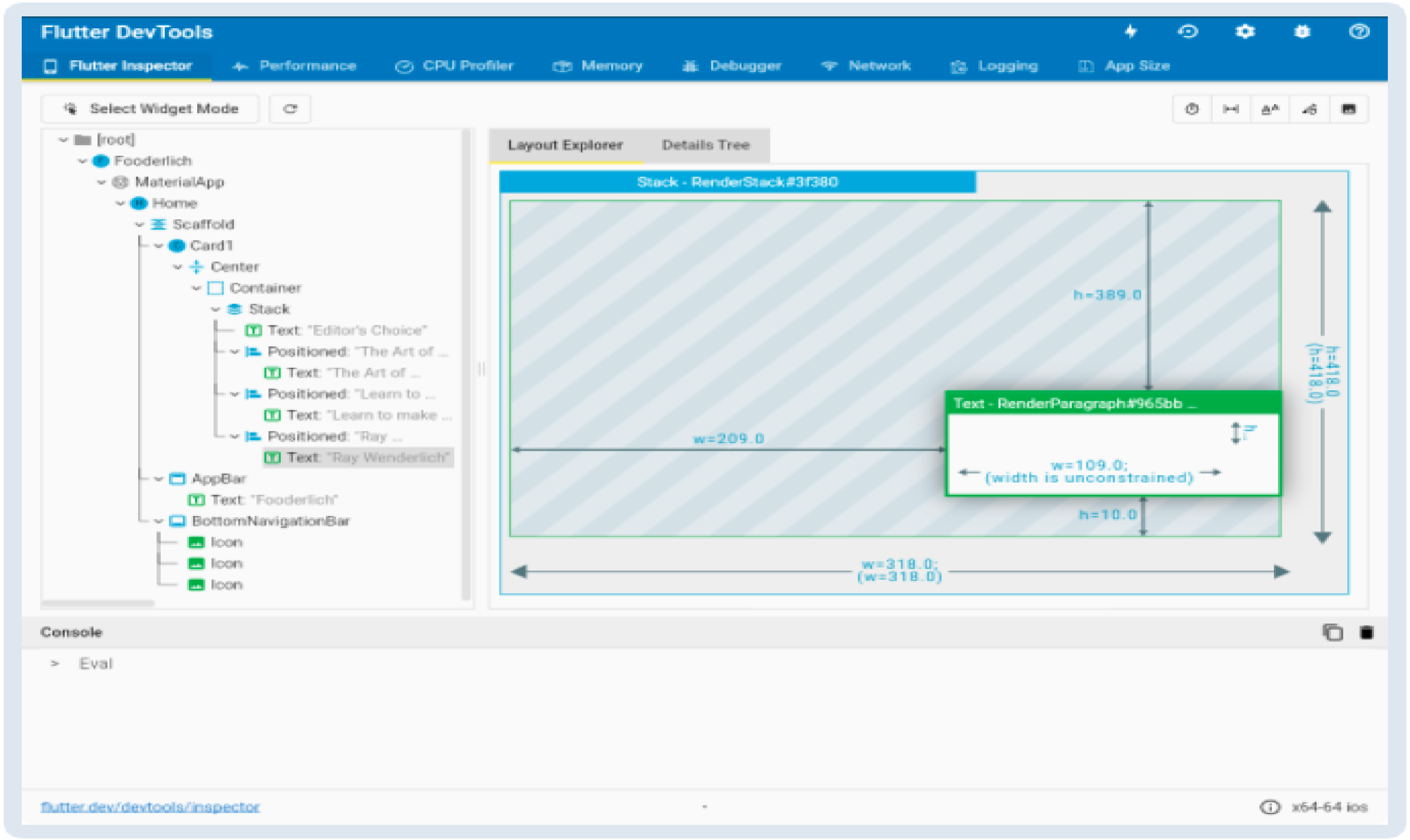


```
Performing hot r
Syncing files to device iPhone 8...
Reloaded 0 of 677 libraries in 117ms.
```

DevTools will open in your browser. Select a widget on the left to see its layout on the right.

Note: It works best with the Google Chrome web browser. Click the ⚙ icon to switch between dark and light mode!

GETTING STARTED

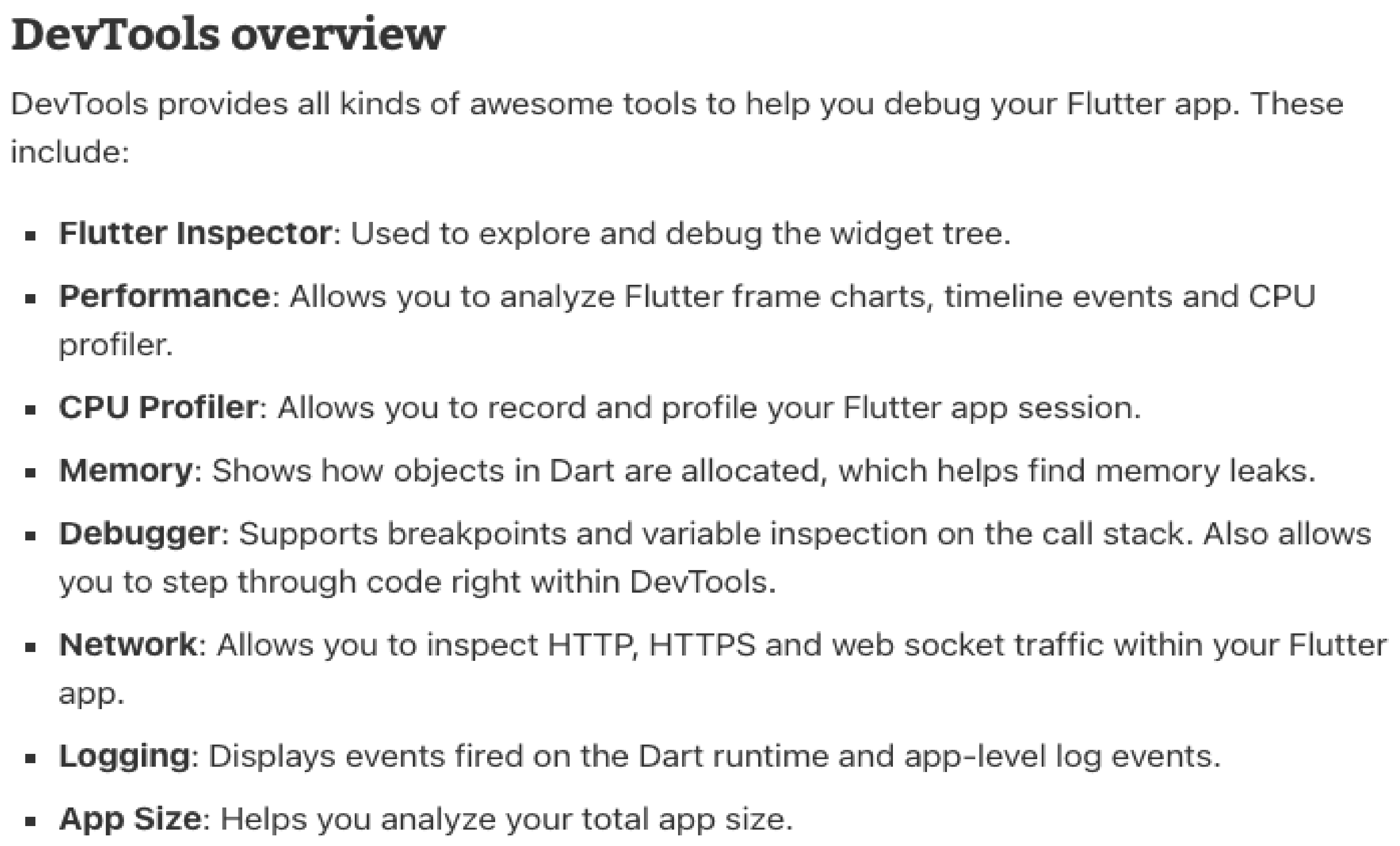


```
[root]
  Fooderlich
    MaterialApp
      Home
        Scaffold
          Card1
            Center
              Container
                Stack
                  Text: "Editor's Choice"
                  Positioned: "The Art of ...
                    Text: "The Art of ...
                  Positioned: "Learn to ...
                    Text: "Learn to make ...
                  Positioned: "Ray ...
                    Text: "Ray Wenderlich"
          AppBar
            Text: "Fooderlich"
          BottomNavigationBar
            Icon
            Icon
            Icon
```

flutter.dev/devtools/inspector

x64-64 ios

GETTING STARTED



DevTools overview

DevTools provides all kinds of awesome tools to help you debug your Flutter app. These include:

- **Flutter Inspector**: Used to explore and debug the widget tree.
- **Performance**: Allows you to analyze Flutter frame charts, timeline events and CPU profiler.
- **CPU Profiler**: Allows you to record and profile your Flutter app session.
- **Memory**: Shows how objects in Dart are allocated, which helps find memory leaks.
- **Debugger**: Supports breakpoints and variable inspection on the call stack. Also allows you to step through code right within DevTools.
- **Network**: Allows you to inspect HTTP, HTTPS and web socket traffic within your Flutter app.
- **Logging**: Displays events fired on the Dart runtime and app-level log events.
- **App Size**: Helps you analyze your total app size.

There are many different tools to play with, but in this chapter, you'll only look at the **Flutter Inspector**.
For information about how the other tools work, check out: https://flutter.dev/docs/development/tools/devtools/overview.

FLUTTER INSPECTOR

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

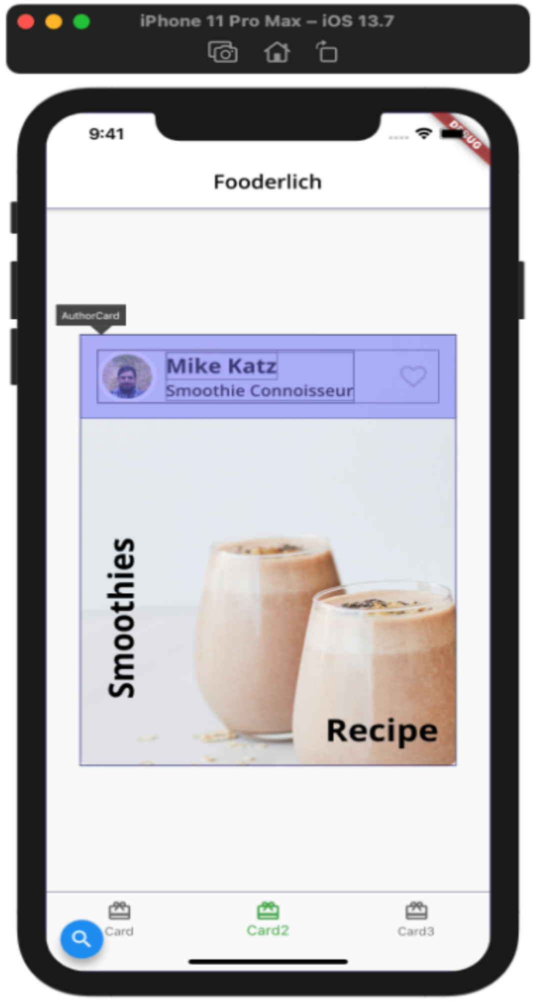
- Visualize your widget tree.
- Inspect the properties of a specific widget in the tree.
- Experiment with different layout configurations using the **Layout Explorer**.
- Enable slow animation to show how your transitions look.

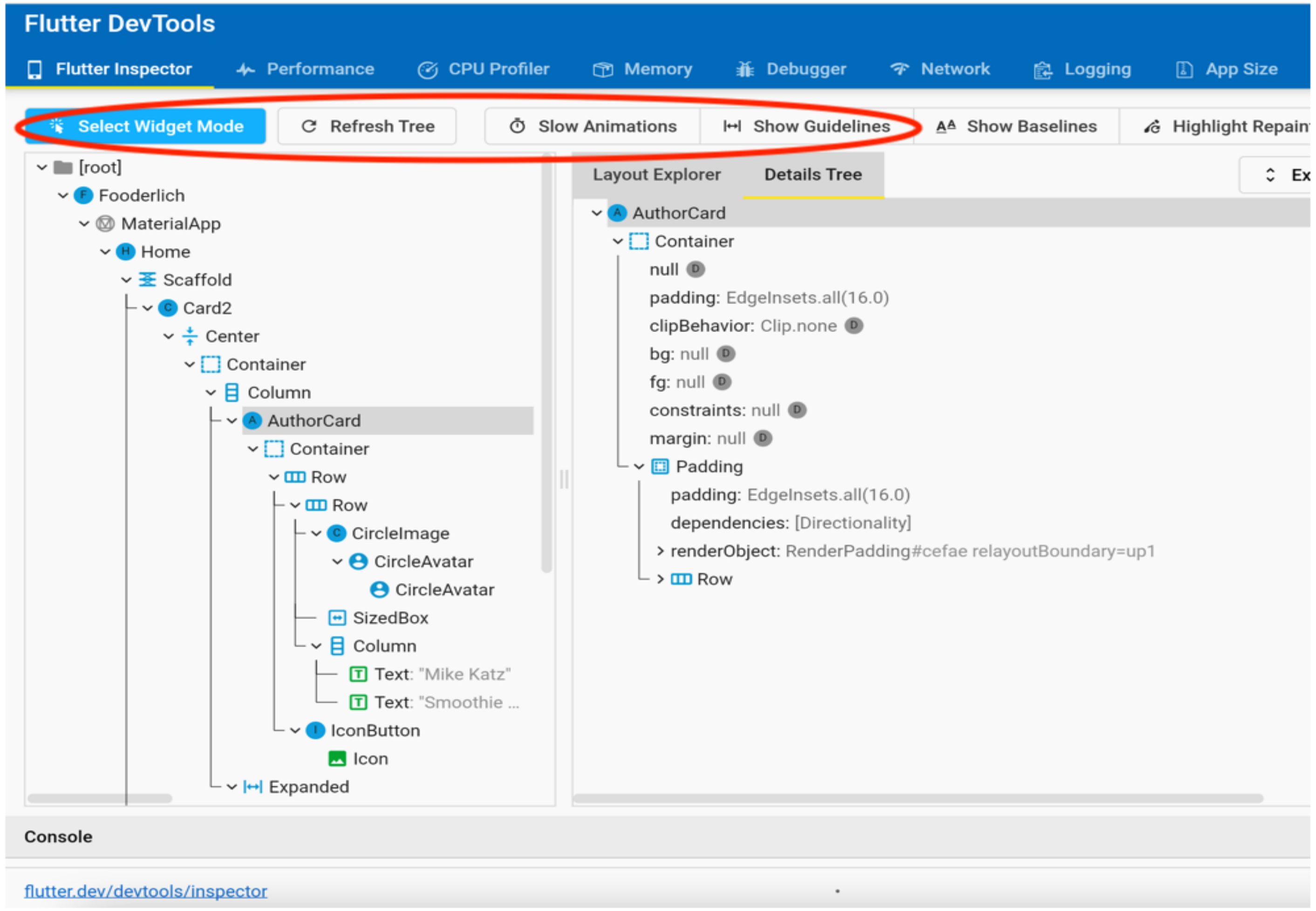
Flutter Inspector tools

Here are some of the important tools to use with the Flutter Inspector.

- **Select Widget Mode**: When enabled, this allows you to tap a particular widget on a device or simulator to inspect its properties.

FLUTTER INSPECTOR





```
[root]
  Fooderlich
    MaterialApp
      Home
        Scaffold
          Card2
            Center
              Container
                Column
                  AuthorCard
                    Container
                      Row
                        Row
                          CircleImage
                            CircleAvatar
                              CircleAvatar
                          SizedBox
                          Column
                            Text: "Mike Katz"
                            Text: "Smoothie ...
                        IconButton
                          Icon
                  Expanded
```

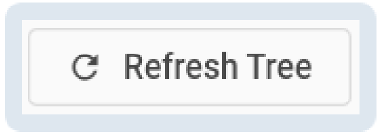
```
AuthorCard
  Container
    null
    padding: EdgeInsets.all(16.0)
    clipBehavior: Clip.none
    bg: null
    fg: null
    constraints: null
    margin: null
    Padding
      padding: EdgeInsets.all(16.0)
      dependencies: [Directionality]
      > renderObject: RenderPadding#cefae relayoutBoundary=up1
      > Row
```

flutter.dev/devtools/inspector

FLUTTER INSPECTOR

Clicking any element in the widget tree also highlights the widget on the device and jumps to the exact line of code. How cool is that!

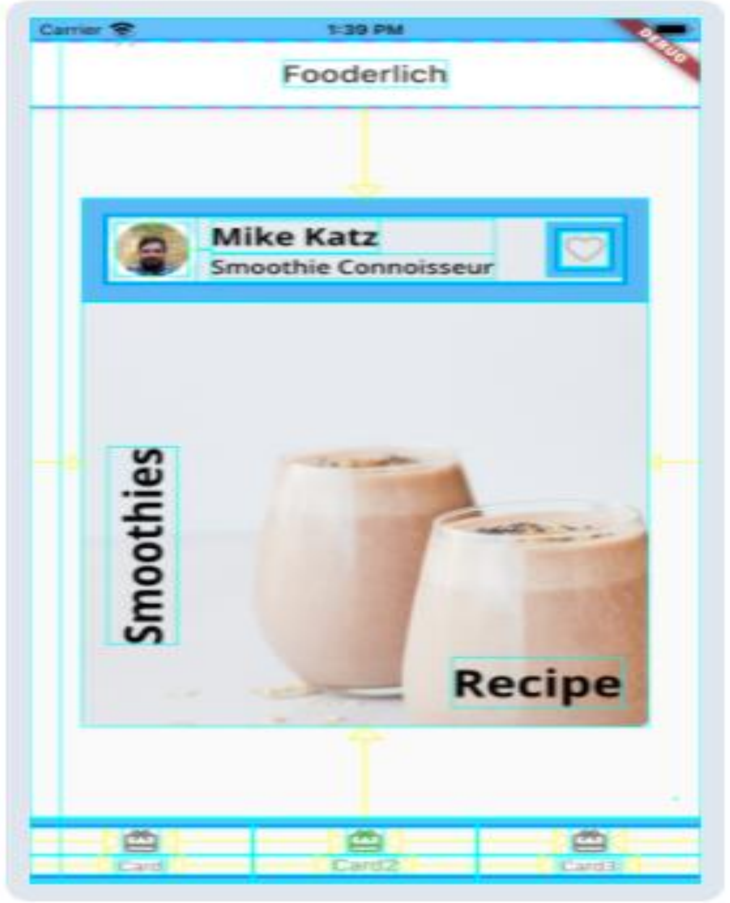
- **Refresh Tree**: Simply reloads the current widget's info.



- **Slow Animation**: Slows down the animation so you can visually inspect the UI transitions.

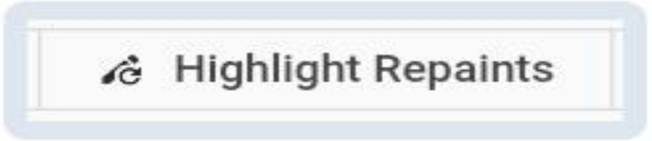
- **Show Guidelines**: Shows visual debugging hints. That allows you to check the borders, paddings and alignment of your widgets.

Here's a screenshot of how it looks on a device:

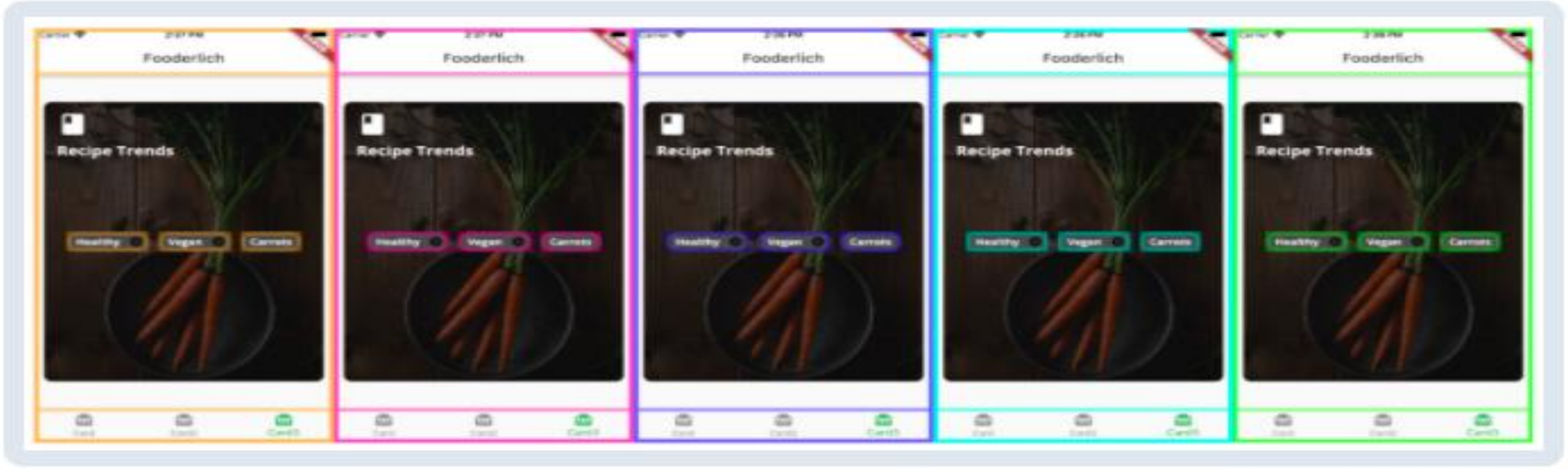


FLUTTER INSPECTOR

- **Highlight Repaints**: Adds a random border to a widget every time Flutter repaints it. This is useful if you want to find unnecessary repaints.

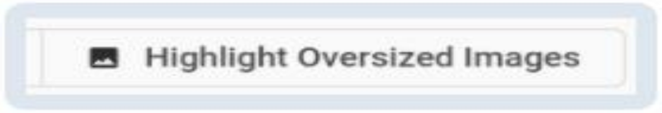


If you feel bored, you can spice things up by enabling disco mode, as shown below:

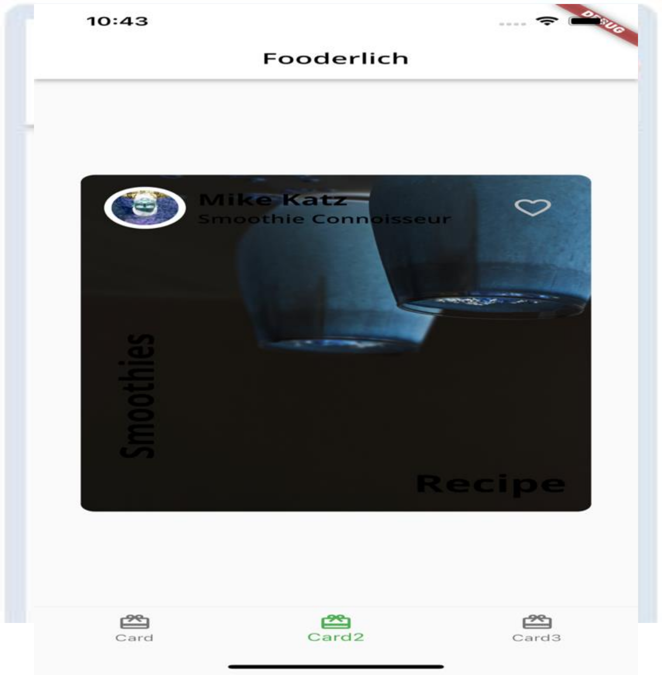


FLUTTER INSPECTOR

- **Highlight Oversized Images**: Tells you which images in your app are oversized.



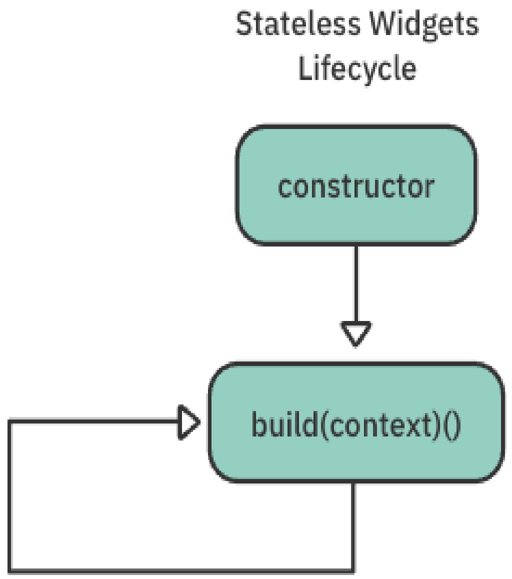
If an image is oversized it will invert the image's colors and flip it upside down. As shown below:



TYPES OF WIDGETS

Stateless widgets

You can't alter the state or properties of Stateless widget once it's built. 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.

The following **events** trigger this kind of widget to update:

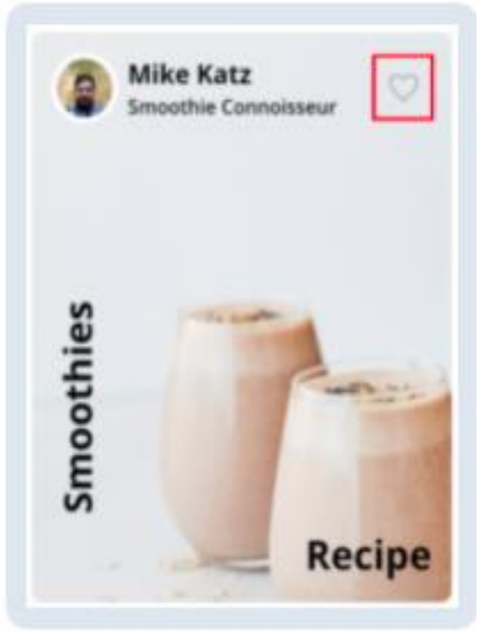
1. The widget is inserted into the widget tree for the first time.
2. The state of a dependency or inherited widget — ancestor nodes — changes.

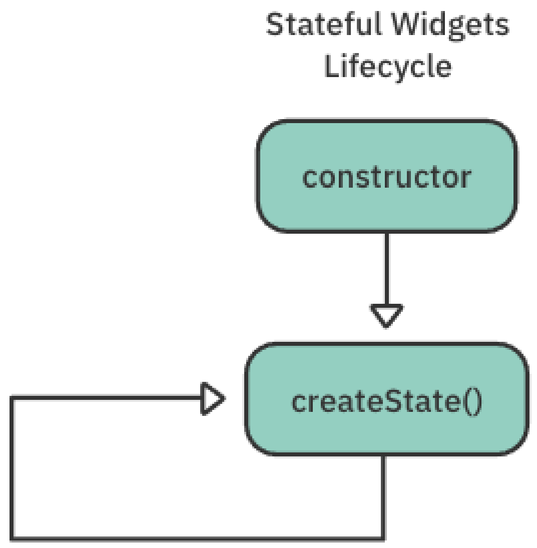
TYPES OF WIDGETS

Stateful widgets

Stateful widgets preserve state, which is useful when parts of your UI need to change dynamically.

For example, one good time to use a stateful widget is when a user taps a **Favorite** button to toggle a simple Boolean value on and off.



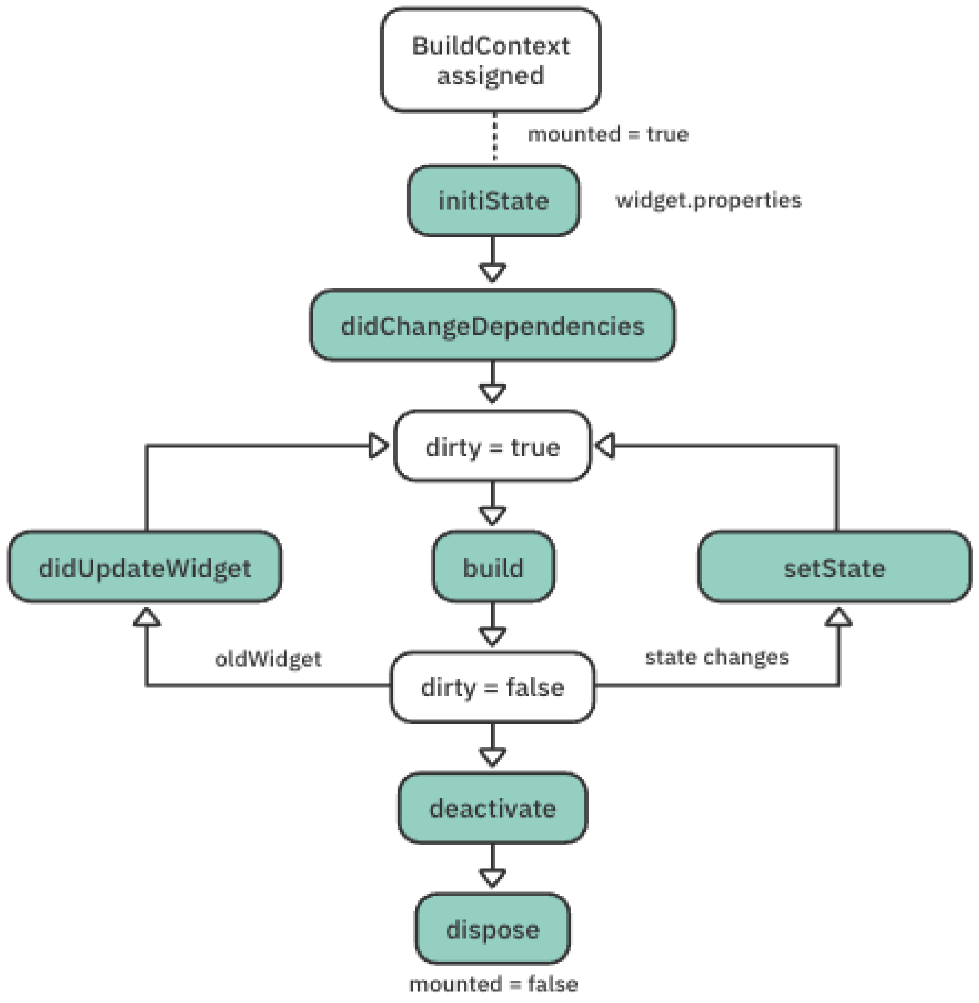


Stateful widgets store their mutable state in a separate `State` class. That's why every stateful widget must override and implement `createState()`.

Next, take a look at the stateful widget's lifecycle.

TYPES OF WIDGETS

- Every widget's build() method takes a BuildContext as an argument. The build context tells you where you are in the tree of widgets. You can access the **element** for any widget through the BuildContext.
- Later, you'll see why the build context is important, especially for accessing state information from parent widgets.

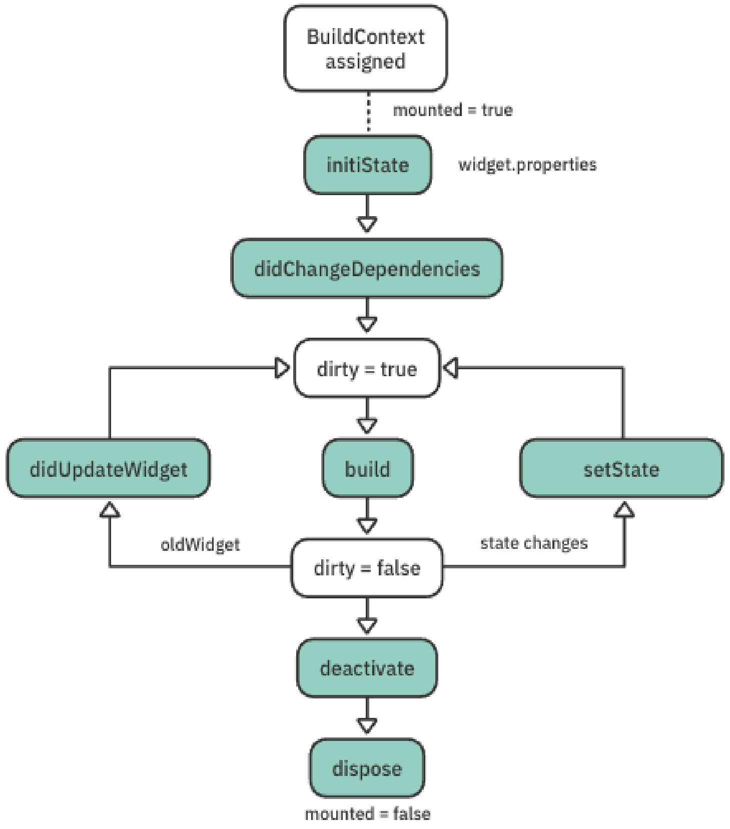


TYPES OF WIDGETS

Now, take a closer look at the 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.
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.

Note: You should always perform heavy computational functions asynchronously and store their results as part of the state for later use with the `build()` function. `build()` should never do anything that's computationally demanding. This is similar to how you think of the iOS or Android main thread. For example, you should never make a network call that stalls the UI rendering.



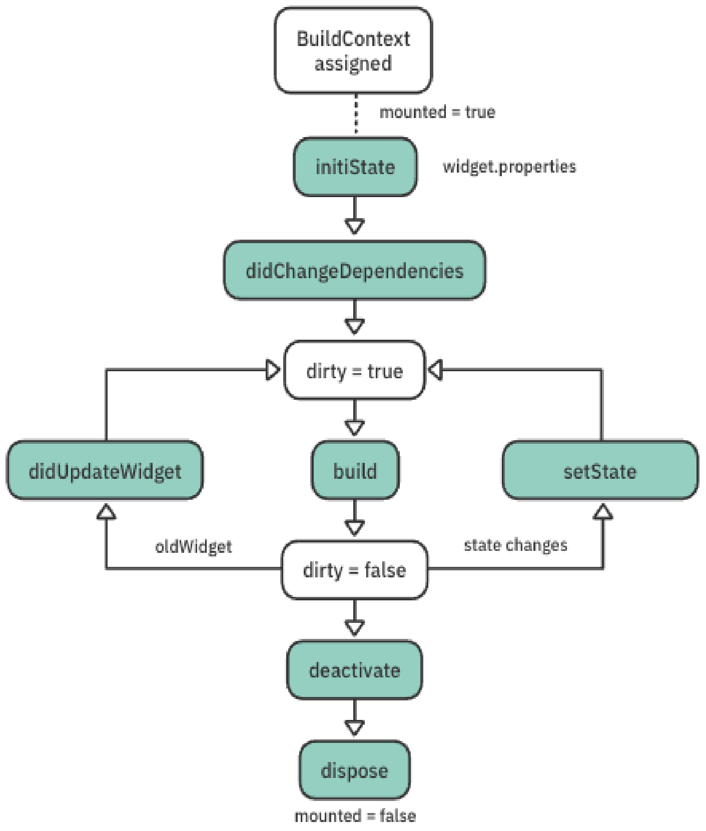
TYPES OF WIDGETS

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.
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.

Note: Asynchronous code should always check if the `mounted` property is true before calling `setstate()`, because the widget may no longer be part of the widget tree.

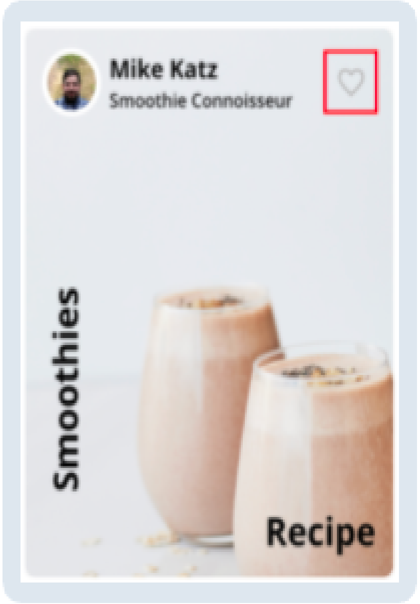
7. When you remove the object from the tree, the framework calls `deactivate()`. The framework can, in some cases, reinsert the state object into another part of the tree.
8. The framework calls `dispose()` when you permanently remove the object and its state from the tree. This method is very important because you'll need it to handle memory cleanup, such as unsubscribing streams and disposing of animations or controllers.

The rule of thumb for `dispose()` is to check any properties you define in your state and make sure you've disposed of them properly.

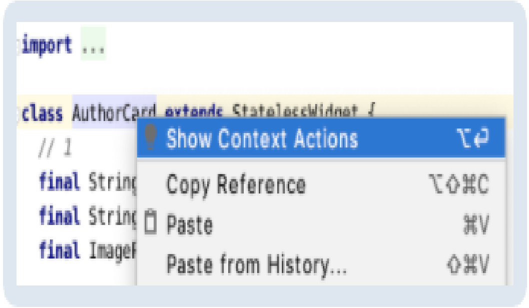


ADDING STATEFUL WIDGETS

`Card2` is currently a `StatelessWidget`. Notice that the **Heart** button on the top-right currently only displays a `SnackBar()`, but nothing else like turning a solid color like a typical **Favorite** button. This isn't because you haven't hooked up any actions. It's because the widget, as it is, can't manage state dynamically. To fix this, you'll change this card into a `StatefulWidget`.

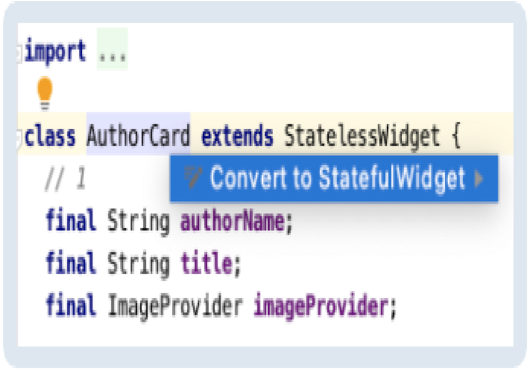


`AuthorCard` is nested within `Card2`. Open **author_card.dart** and right-click on `AuthorCard`. Then click **Show Context Actions** from the menu that pops up:



```
import ...

class AuthorCard extends StatelessWidget {
  // 1
  final Strin
  final Strin
  final ImageP
```

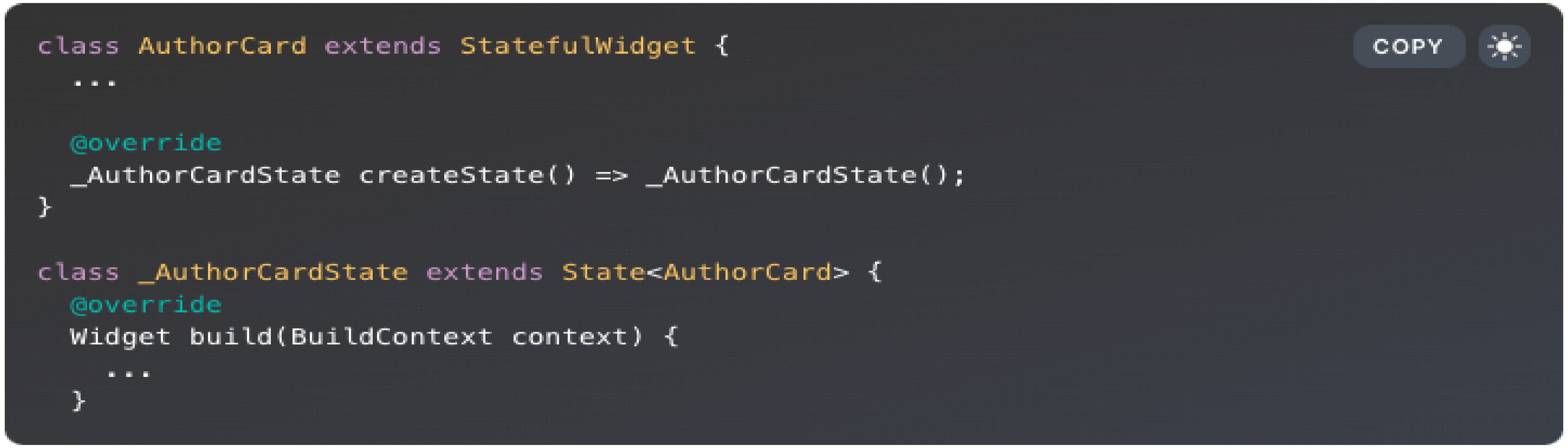
Select **Convert to StatefulWidget**. Instead of converting manually, you can just use this menu shortcut to do it automatically:



```
import ...

class AuthorCard extends StatelessWidget {
  // 1
  final String authorName;
  final String title;
  final ImageProvider imageProvider;
```

ADDING STATEFUL WIDGETS

There are now two classes:



```
class AuthorCard extends StatefulWidget {
  ...

  @override
  _AuthorCardState createState() => _AuthorCardState();
}

class _AuthorCardState extends State<AuthorCard> {
  @override
  Widget build(BuildContext context) {
    ...
  }
```

A couple of things to notice in the code above:

- The refactor converted `AuthorCard` from a `StatelessWidget` into a `StatefulWidget`. It added a `createState()` implementation.
- The refactor also created the `_AuthorCardState` state class. It stores mutable data that can change over the lifetime of the widget.

ADDING STATEFUL WIDGETS

Implementing favorites

In `_AuthorCardState`, add the following property right after the class declaration:

```
bool _isFavorited = false;
```

COPY

Now that you've created a new state, you need to manage it. Replace the current `IconButton` in `_AuthorCardState` with the following:

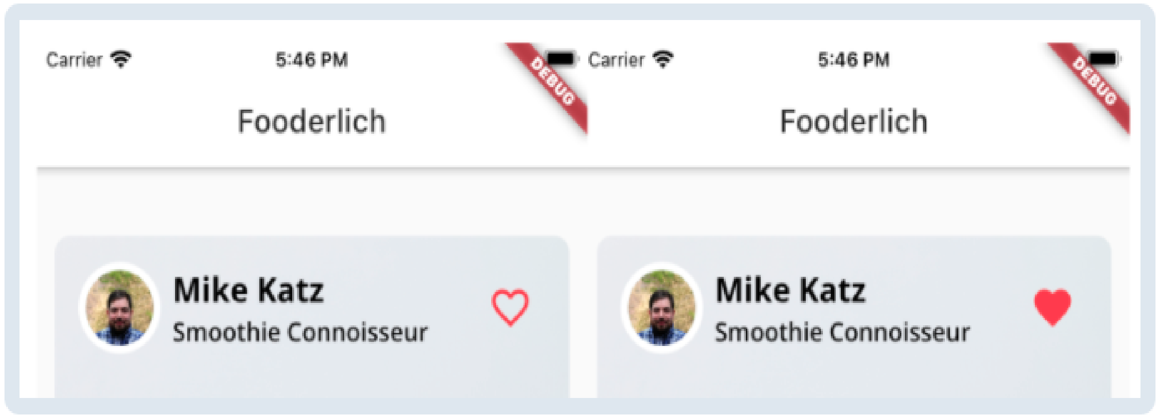
```
IconButton(
  // 1
  icon: Icon(_isFavorited ? Icons.favorite : Icons.favorite_border),
  iconSize: 30,
  // 2
  color: Colors.red[400],
  onPressed: () {
    // 3
    setState(() {
      _isFavorited = !_isFavorited;
    });
  },
)
```

COPY

Here's how the new state works:

1. First, it checks if the user has favorited this recipe card. If `true`, it shows a filled heart. If `false`, it shows an outlined heart.
2. It changes the color to red to give the app more life.
3. When the user presses the `IconButton`, it toggles the `_isFavorited` state via a call to `setState()`.

Save the change to trigger a hot reload and see the heart button toggle on and off when you tap it, as shown below:

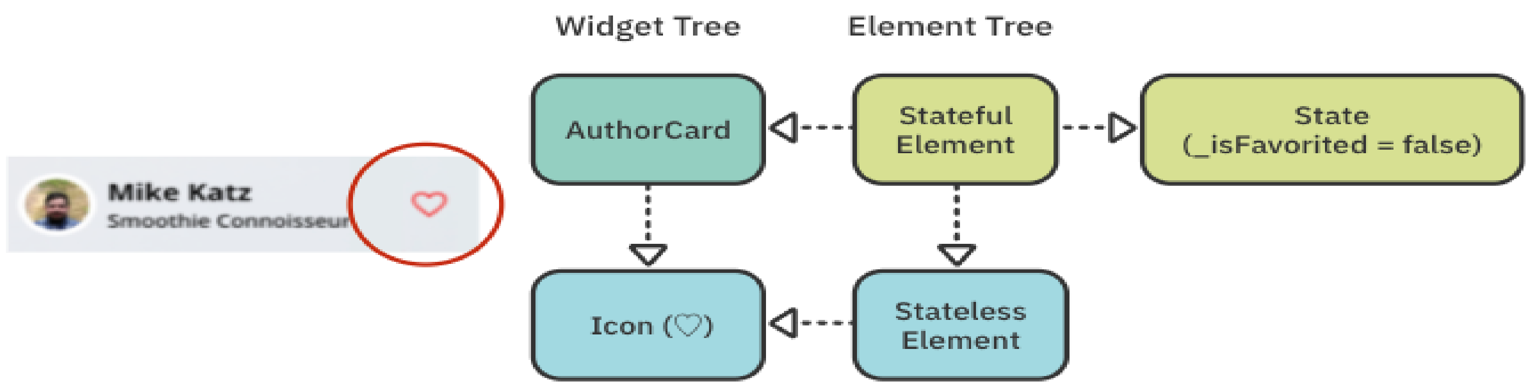


ADDING STATEFUL WIDGETS

Examining the widget tree

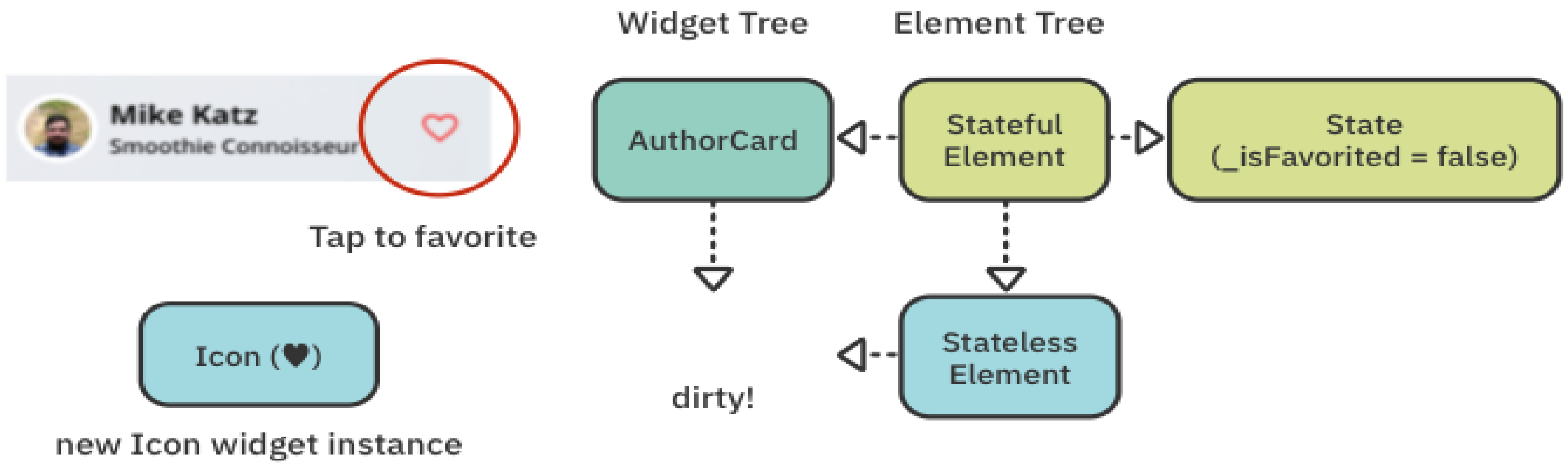
Now that you've turned `AuthorCard` into a stateful widget, your next step is to look at how the element tree manages state changes.

Recall that the framework will construct the widget tree and, for every widget instance, create an element object. The element, in this case, is a `StatefulElement` and it manages the state object, as shown below:



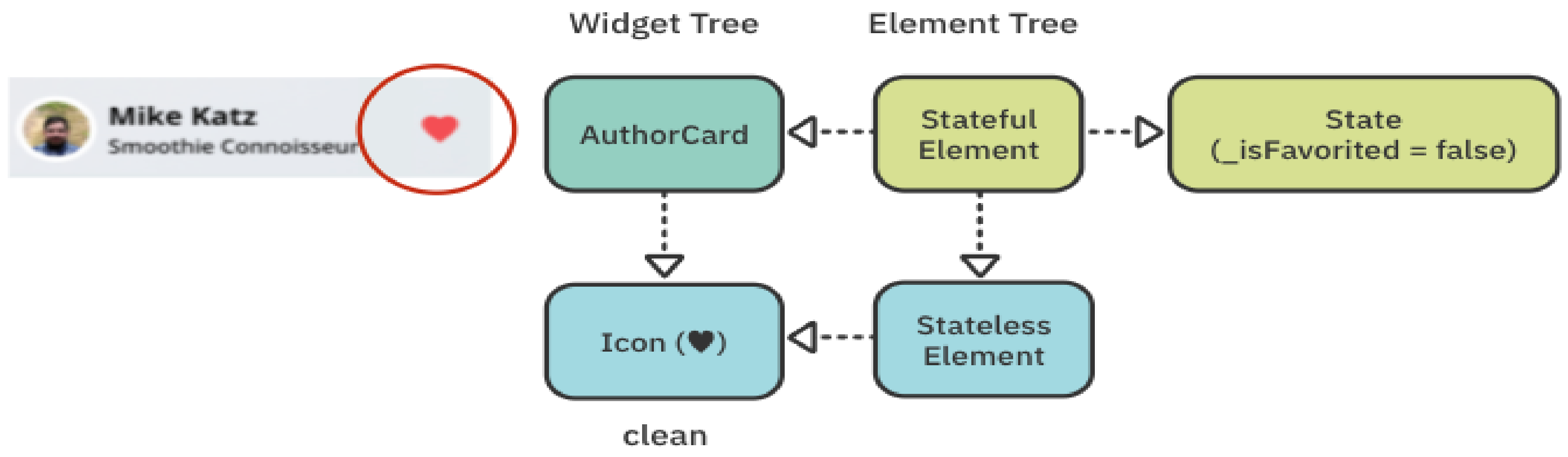
When the user taps the heart button, `setState()` runs and toggles `_isFavorited` to true. Internally, the state object marks this element as **dirty**. That triggers a call to `build()`.

ADDING STATEFUL WIDGETS



This is where the element object shows its strength. It removes the old widget and replaces it with a new instance of `Icon` that contains the filled heart icon.

ADDING STATEFUL WIDGETS



Rather than reconstructing the whole tree, the framework only updates the widgets that need to be changed. It walks down the tree hierarchy and checks for what's changed. It reuses everything else.

Now, what happens when you need to access data from some other widget, located elsewhere in the hierarchy? You use inherited widgets.

SCENARIOS FOR BRAINSTORMING:

Scenario 1: Profile Screen

- A profile screen shows a user's profile picture, name, and bio. The information is static and fetched once from a backend API.
- **Question**: Should the profile screen components (picture, name, bio) use Stateless or Stateful widgets?

Scenario 2: Counter App

- A button increments a counter displayed on the screen. The counter value changes every time the button is pressed.
- **Question**: Should the counter and button components use Stateless or Stateful widgets?

Scenario 3: Login Form

- A login form with two text fields (username and password) and a login button. The button becomes enabled only when both fields are filled.
- **Question**: Should the text fields and button components use Stateless or Stateful widgets?

Scenario 4: Chat Application

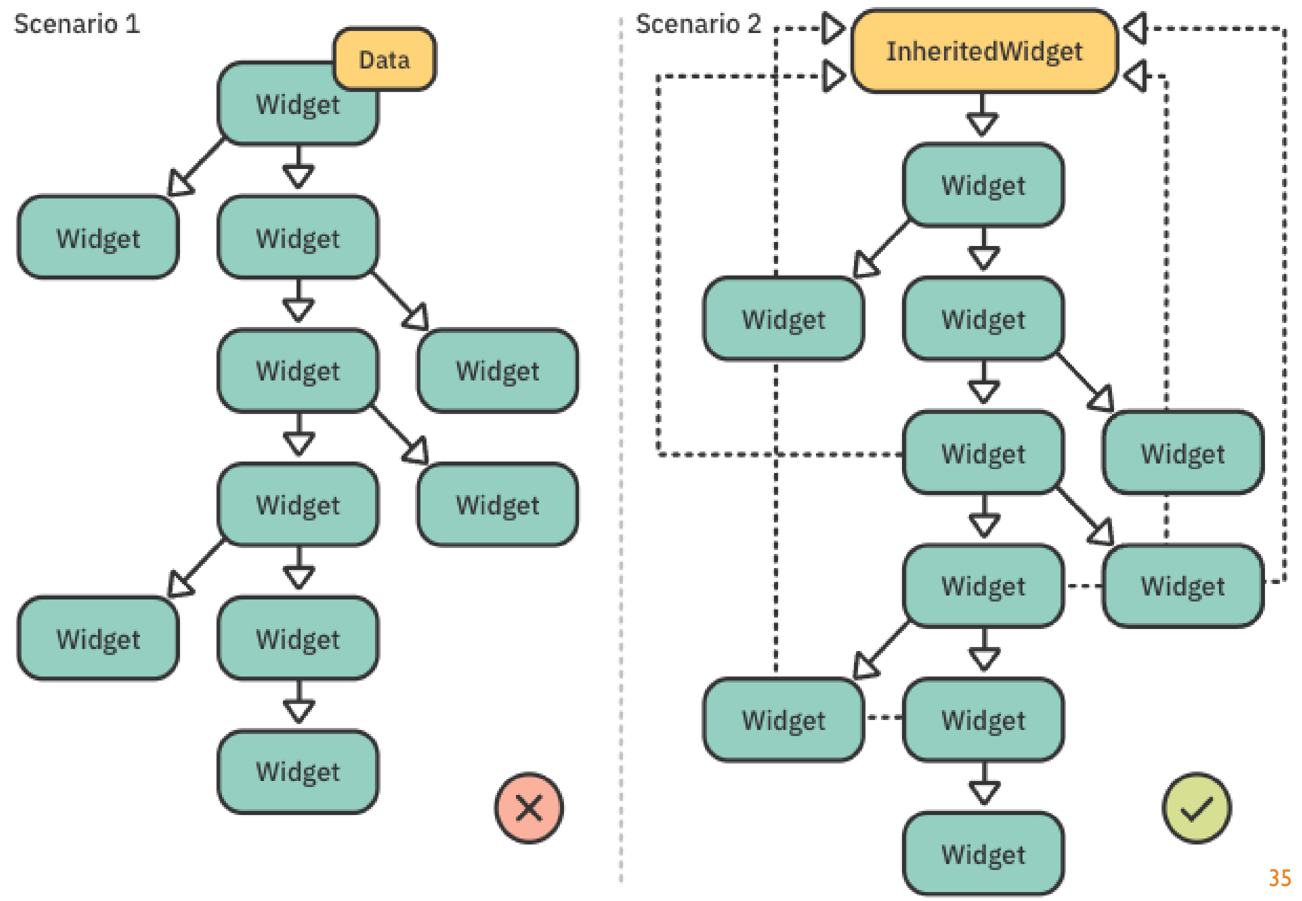
- In a chat app, a message list displays incoming and outgoing messages in real time. Messages are updated dynamically as new ones arrive.
- **Question**: Should the message list use Stateless or Stateful widgets?

Scenario 5: Dark Mode Toggle

- A toggle switch lets the user switch between light and dark mode. The theme changes dynamically throughout the app when the toggle is pressed.
- **Question**: Should the toggle switch component use Stateless or Stateful widgets?

INHERITED WIDGETS

- Inherited widgets let you access state information from the parent elements in the tree hierarchy.
- Imagine you have a piece of data way up in the widget tree that you want to access. One solution is to pass the data down as a parameter on each nested widget — but that quickly becomes annoying and cumbersome.
- Wouldn't it be great if there was a centralized way to access such data?



INHERITED WIDGETS

That’s where inherited widgets come in! By adding an inherited widget in your tree, you can reference the data from any of its descendants. This is known as **lifting state up**.

For example, you use an inherited widget when:

- Accessing a Theme object to change the UI’s appearance.
- Calling an API service object to fetch data from the web.
- Subscribing to streams to update the UI according to the data received.
- Inherited widgets are an advanced topic. You’ll learn more about them in Section 4, “Networking, Persistence and State”, which covers state management and the **Provider** package—a wrapper around an inherited widget.

WHAT IS "LIFTING STATE UP"?

"is a technique used to move shared state to the nearest common ancestor in the widget tree so that it can be accessed by **Lifting state up"** multiple child widgets.

Why It's Important:

- Avoids passing data down through a long chain of widget parameters (**prop drilling**).
- Keeps state centralized for better organization and easier debugging.

Prop drilling refers to the process of passing data (props) from a parent widget to deeply nested child widgets by passing it through intermediate widgets that do not directly use the data. It can make code harder to maintain as the app grows.

EXAMPLE SCENARIO: PROP DRILLING

- **Scenario:**
- A parent widget (**App**) holds a **themeColor** state.
- A deeply nested widget (**NestedChild**) needs to use this **themeColor**.
- The intermediate widgets (**Parent** and **Child**) must pass the **themeColor** down even though they don't use it themselves.

EXAMPLE SCENARIO: PROP DRILLING

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

void main() {
  runApp(App());
}

class App extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: ThemeProvider(),
    );
  }
}
```

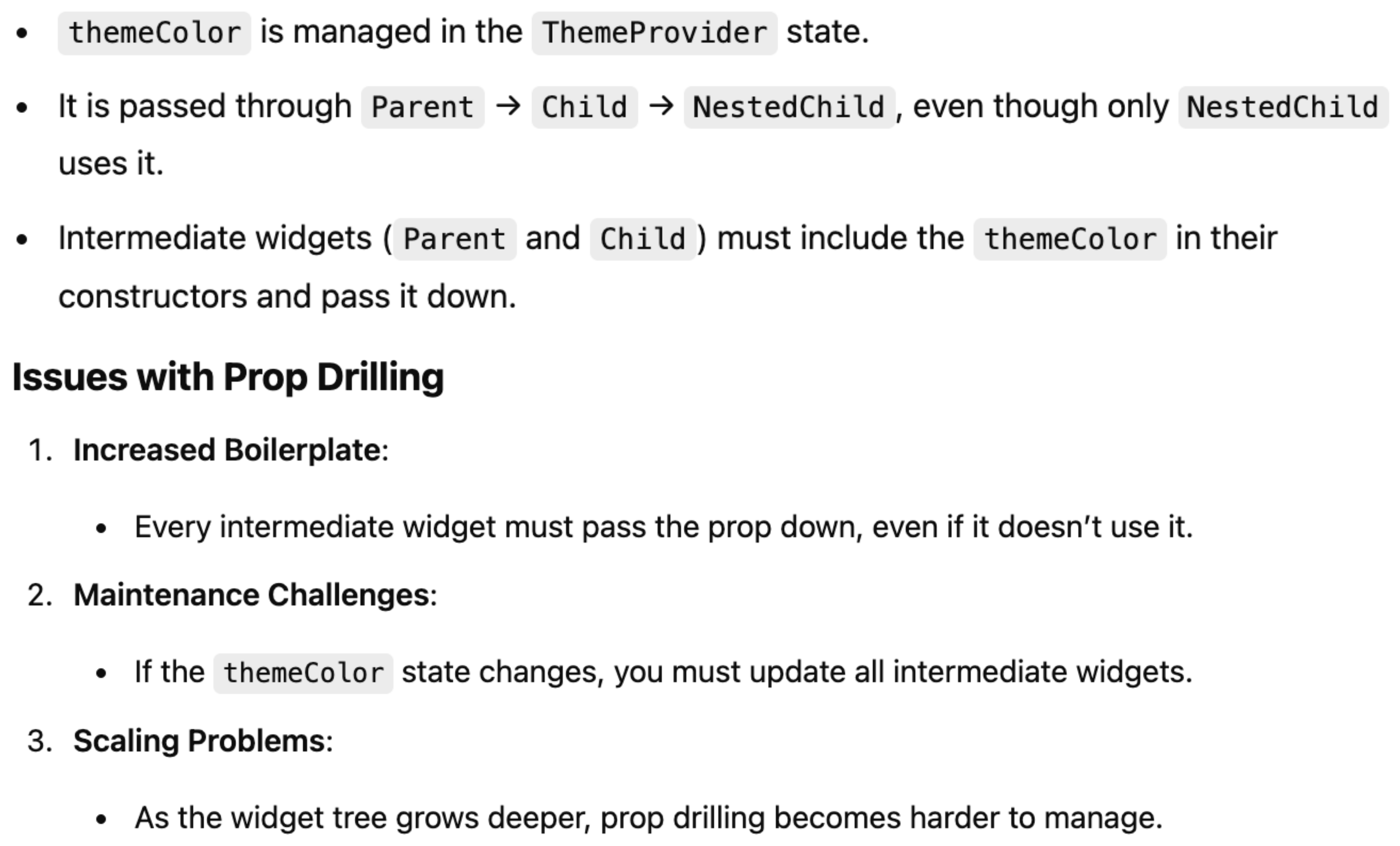
```dart
class ThemeProvider extends StatefulWidget {
  @override
  _ThemeProviderState createState() => _ThemeProviderState();
}

class _ThemeProviderState extends State<ThemeProvider> {
  Color themeColor = Colors.blue;

  void changeTheme() {
    setState(() {
      themeColor = themeColor == Colors.blue ? Colors.red : Colors.blue;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("Prop Drilling Example"),
      ),
      body: Parent(
        themeColor: themeColor, // Passing themeColor to Parent
        onThemeChange: changeTheme, // Passing changeTheme to Parent
      ),
    );
  }
}
```

EXAMPLE SCENARIO: PROP DRILLING

```dart
class Parent extends StatelessWidget {
  final Color themeColor;
  final VoidCallback onThemeChange;

  Parent({required this.themeColor, required this.onThemeChange});

  @override
  Widget build(BuildContext context) {
    return Column(
      children: [
        ElevatedButton(
          onPressed: onThemeChange,
          child: Text("Change Theme"),
        ),
        Child(themeColor: themeColor), // Passing themeColor to Child
      ],
    );
  }
}
```

```dart
class Child extends StatelessWidget {
  final Color themeColor;

  Child({required this.themeColor});

  @override
  Widget build(BuildContext context) {
    return Container(
      margin: EdgeInsets.all(16),
      child: NestedChild(themeColor: themeColor), // Passing themeColor to NestedCh
    );
  }
}

class NestedChild extends StatelessWidget {
  final Color themeColor;

  NestedChild({required this.themeColor});

  @override
  Widget build(BuildContext context) {
    return Center(
      child: Text(
        "Nested Child with theme color!",
        style: TextStyle(color: themeColor, fontSize: 20),
      ),
    );
  }
}
```

HOW THIS DEMONSTRATES PROP DRILLING



- `themeColor` is managed in the `ThemeProvider` state.
- It is passed through `Parent` → `Child` → `NestedChild`, even though only `NestedChild` uses it.
- Intermediate widgets (`Parent` and `Child`) must include the `themeColor` in their constructors and pass it down.

Issues with Prop Drilling

1. **Increased Boilerplate**:
    - Every intermediate widget must pass the prop down, even if it doesn't use it.
2. **Maintenance Challenges**:
    - If the `themeColor` state changes, you must update all intermediate widgets.
3. **Scaling Problems**:
    - As the widget tree grows deeper, prop drilling becomes harder to manage.

Alternative Solution:
InheritedWidget or State Managements Solutions

You can use an InheritedWidget to avoid prop drilling or anyother state management solutions.

REAL-WORLD EXAMPLES: HOW COMPANIES USE FLUTTER

1.Alibaba

Use Case:

- **Platform**: Xianyu (Idle Fish)
- **Purpose**: A second-hand goods marketplace in China.
- **Why Flutter?**
    - Cross-platform development for iOS and Android reduced development time.
    - High-performance rendering for smooth scrolling and animations in a marketplace app.

Key Flutter Features Used:

- **Custom Widgets**: To create visually rich interfaces.
- **Flutter's GPU Rendering**: Ensures smooth scrolling even with a high volume of product images.
- **Platform Integration**: Seamlessly integrates with existing native modules.

- Alibaba Group Flutter Story
- Alibaba used Flutter to build 50+ million user Xianyu app (Flutter Developer Story)

REAL-WORLD EXAMPLES: HOW COMPANIES USE FLUTTER

2. Google

Use Case:

- **Platform**: Google Ads
- **Purpose**: Mobile app for campaign management, tracking performance, and receiving real-time notifications.
- **Why Flutter?**
  - Single codebase for both Android and iOS.
  - Flutter's declarative UI design simplifies creating intuitive, responsive dashboards.

Key Flutter Features Used:

- **Interactive Dashboards**: Built with widgets like ListView, DataTable, and GridView.
- **Push Notifications**: Integrated seamlessly for real-time updates.

```
import 'package:google_mobile_ads/
google_mobile_ads.dart';
```

REAL-WORLD EXAMPLES: HOW COMPANIES USE FLUTTER

3. BMW

Use Case:

- **Platform**: My BMW App
- **Purpose**: Offers a unified app experience for users across different regions to manage their cars.
- **Why Flutter?**
  - Faster iteration with shared UI across Android and iOS.
  - High-performance animations for car controls.

Key Flutter Features Used:

- **Adaptive Layouts**: Ensures a consistent look across platforms.
- **Real-Time Connectivity**: Integrates with IoT features for car control.

https://flutter.dev/showcase/bmw

EXTENSION TASKS

For advanced learners:

- Build a small clone of a real-world app (e.g., **Google Ads dashboard** or **BMW car control interface**).
- Integrate animations, state management, and real-time updates for a complete app experience.

KEY POINTS (CHAPTER 4)

- What are the three trees in Flutter?
- Flutter maintains three trees in parallel: the **Widget, Element and RenderObject** trees.
- When should you use a StatefulWidget?
- You should always start by creating **StatelessWidgets** and only use **StatefulWidgets** when you need to manage and maintain the state of your widget.
- A Flutter app is performant because it maintains its structure and only updates the widgets that need redrawing.
- How does the Flutter Inspector help developers?
- The **Flutter Inspector** is a useful tool to debug, experiment with and inspect a widget tree.
- Inherited widgets are a good solution to access state from the top of the tree.

WHERE TO GO FROM HERE?

- If you want to learn more theory about how widgets work, check out the following links:

Detailed architectural overview of Flutter and widgets: https://flutter.dev/docs/resources/architectural-overview.

- The Flutter team created a YouTube series explaining widgets under the hood: https://www.youtube.com/playlist?list=PLjxrf2q8roU2HdJQDjJzOeO6J3FoFLWr2.
- The Flutter team gave a talk in China on how to render widgets: https://youtu.be/996ZgFRENMs.

In the next chapter, you'll get back to more practical concerns and see how to create **scrollable widgets**.