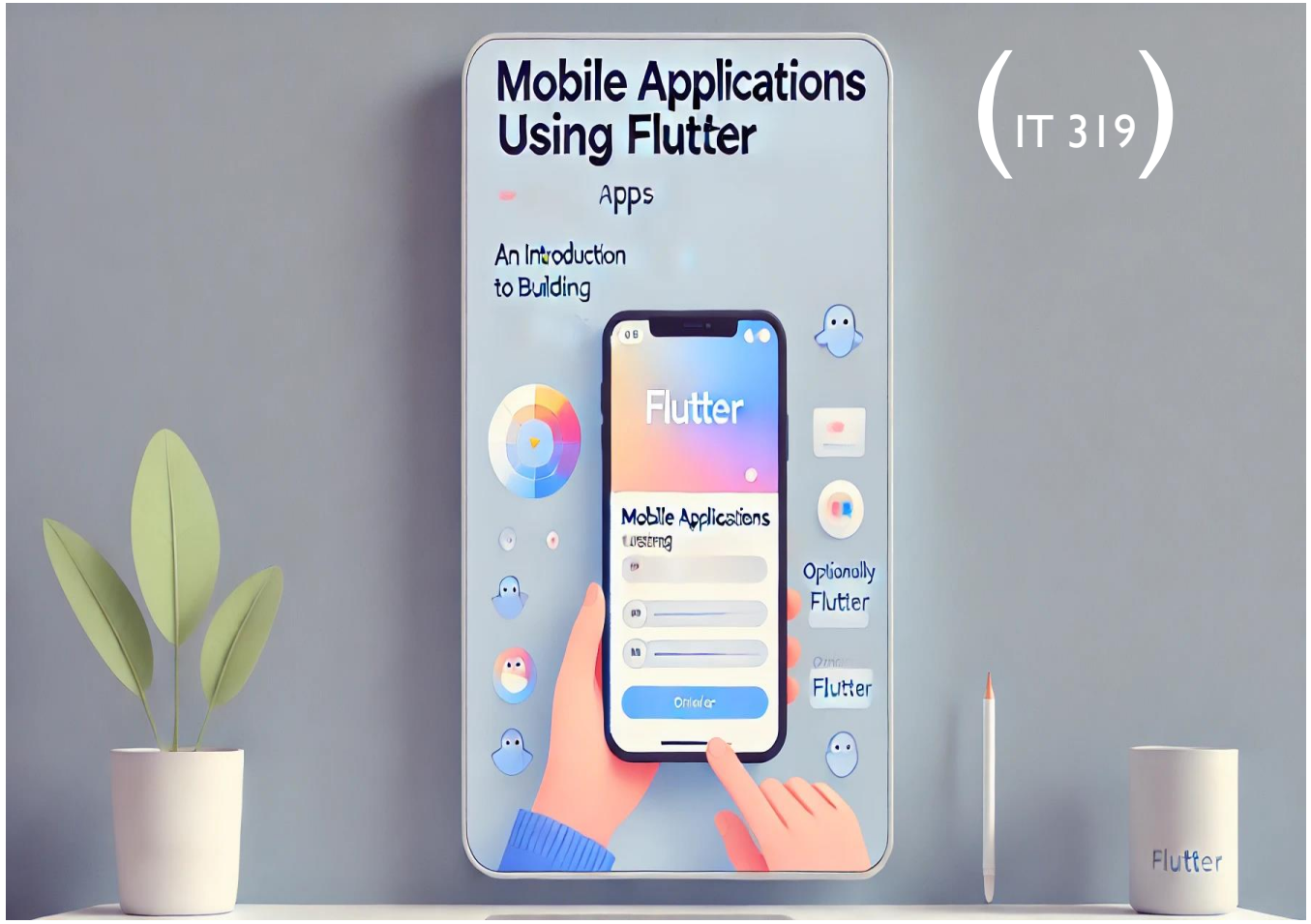
# MOBILE APPLICATIONS

OOPI earlier




(IT 319)

IT DEPT.

TIU

3RD GRADE

INTRO TO FLUTTER AND RECIPE APP

Lect. Mohammad Salim Al-Othman

Fall 2025-26

Week 3-4

# FLUTTER APPRENTICE

# COURSE CONTENT

- Flutter and OOP

**COURSE CONTENT**

**Topic**

Introduction to OOP , Class diagram

Introduction to OOP , Class diagram and Dart Packages

Section 1: Build Your First Flutter App, structure of Flutter projects, create the UI of a Flutter app by Widgets

Section 2: Everything's a Widget, start to build a full-featured recipe app named Fooderlich

Section 2: Everything's a Widget, layout widgets, scrollable widgets and interactive widgets

Section III: Navigating Between Screens, routes and navigation

Midterm Exam

Midterm Exam

Section III: Navigating Between Screens : deep links and URLs

Section IV: Networking, Persistence & State: Share Preference

Section IV: Networking, Persistence & State: Serialization with JSON

Section IV: Networking, Persistence & State: Networking in Flutter

Section IV: Networking, Persistence & State: Chopper Library

Section IV: Networking, Persistence & State: State Management

Final Exam

Final Exam

# CONTENTS

# What You Need

To follow along with this book, you'll need the following:

- **Xcode 12.5.1 or later**. Xcode is iOS's main development tool, so you need it to build your Flutter app for iOS. You can download the latest version of Xcode from Apple's developer site here: apple.co/2asi58y or from the Mac App Store. Xcode 12.5.1 requires a Mac running **macOS Big Sur (11) or later**.

  > **Note**: You also have the option of using Linux or Windows, but you won't be able to install Xcode or build apps for iOS on those platforms.

- **Cocoapods 1.10.2 or later**. Cocoapods is a dependency manager Flutter uses to run code on iOS.
- **Flutter SDK 2.5.1 or later**. You can download the Flutter SDK from the official Flutter site at https://flutter.dev/docs/get-started/install/macos. Installing the Flutter SDK will also install the **Dart SDK**, which you need to compile the Dart code in your Flutter apps.
- **Android Studio 2020.3.1 or later**, available at https://developer.android.com/studio. This is the IDE in which you'll develop the sample code in this book. It also includes the Android SDK and the build system for running Flutter apps on Android.
- **Flutter Plugin for Android Studio 60.1.2 or later**, installed by going to Android Studio **Preferences** on macOS (or **Settings** on Windows/Linux) and choosing **Plugins**, then searching for "Flutter".

You have the option of using **Visual Studio Code** for your Flutter development environment instead of Android Studio. You'll still need to install Android Studio to have access to the Android SDK and an Android emulator. If you choose to use Visual Studio Code, follow the instructions on the official Flutter site at https://flutter.dev/docs/get-started/editor?tab=vscode to get set up.

Chapter 1, "Getting Started" explains more about Flutter history and architecture. You'll learn how to start using the Flutter SDK, then you'll see how to use Android Studio and Xcode to build and run Flutter apps.

# INTRODUCTION

- Flutter is an incredible user interface (UI) toolkit that lets you build apps for iOS and Android — and even the web and desktop platforms like macOS, Windows and Linux — all from a single codebase.
- Flutter has all the benefits of other cross-platform tools, especially because you're targeting multiple platforms from one codebase. Furthermore, it improves upon most cross-platform tools thanks to a super-fast rendering engine that makes your Flutter apps perform as native apps.
- If you're coming from a platform like iOS or Android, you'll find the Flutter development experience refreshing! Thanks to a feature called "**hot reload**", you rarely need to rebuild your apps as you develop them. A running app in a simulator or emulator will refresh with code changes automatically as you save your source files!

# SECTION I: BUILD YOUR FIRST FLUTTER APP

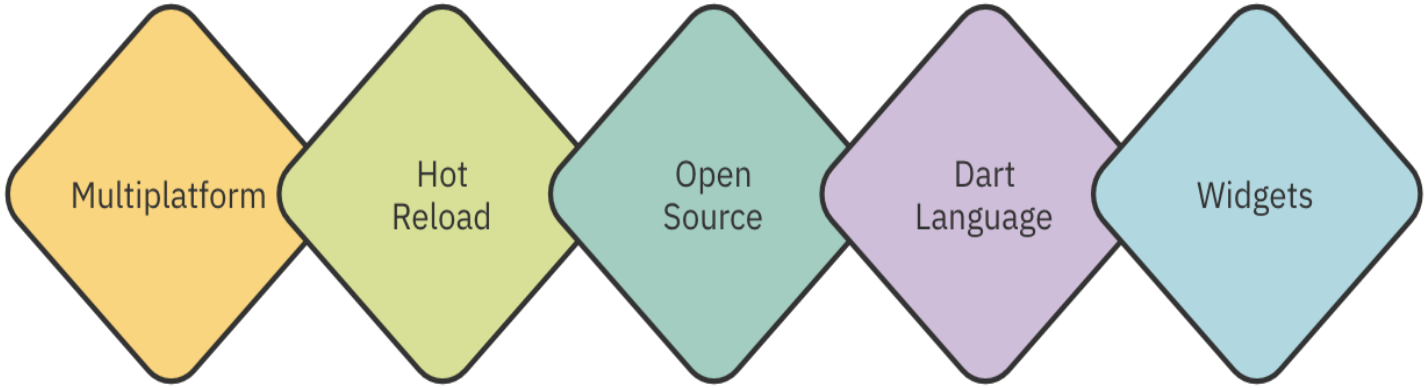
The chapters in this section introduce you to Flutter, get you up and running with a Flutter development environment and walk you through building your first Flutter app.

You'll learn about where Flutter came from and why it exists, understand the structure of Flutter projects and see how to create the UI of a Flutter app.

You'll also get your first introduction to the key component found in Flutter user interfaces: widgets!

# WHAT IS FLUTTER?

- In the simplest terms, Flutter is a software development toolkit from Google for building cross-platform apps. Flutter apps consist of a series of packages, plugins and widgets — but that's not all.
- Flutter is a process, a philosophy and a community as well.



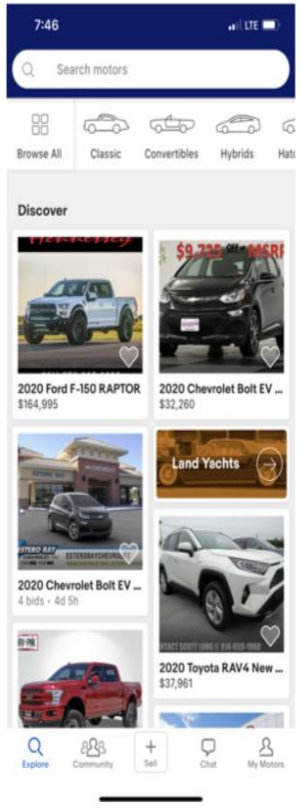

One thing Flutter is not is a language. Flutter uses **Dart** as its programming language. If you know Kotlin, Swift, Java or Typescript, you'll find Dart familiar, since it's an object-oriented C-style language.

For years, programmers have been promised the ability to **write once and run anywhere**; Flutter may well be the best attempt yet at achieving that goal.

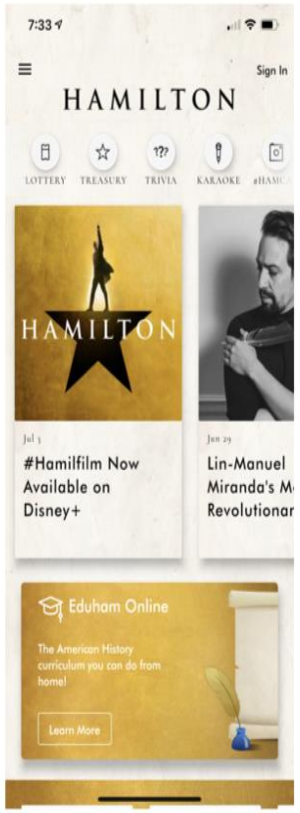
# FLUTTER SHOWCASE

Popular apps from some of the world's biggest companies are built with Flutter. These include:

- Very Good Ventures
- Tencent
- Realtor.com
- Google Assistant
- New York Times
- Policygenius
- Google Stadia
- Take a look at some recent examples:

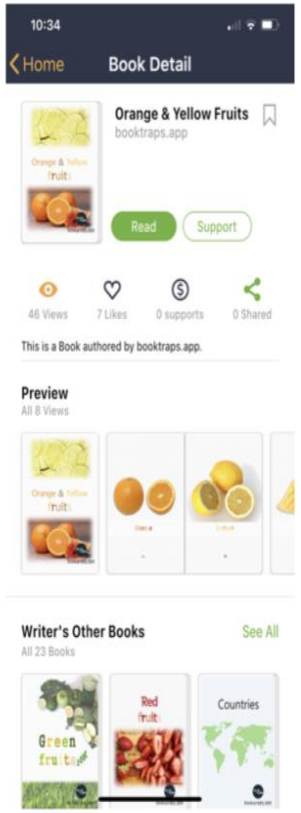



eBay Motors




Hamilton


Hue BT




BookTraps

# WHEN NOT TO USE FLUTTER

- **Games and audio**

While you can create simple 2D games using Flutter, for complex 2D and 3D games, you'd probably prefer to base your app on a cross-platform game engine technology like Unity or Unreal. They have more domain-specific features like physics, sprite and asset management, game state management, multiplayer support and so on. Flutter doesn't have a sophisticated audio engine yet, so audio editing or mixing apps are at a disadvantage over those that are purpose-built for a specific platform.

- **Apps with specific native SDK needs**

Flutter supports many, but not all, native features. Flutter might not be a practical choice if you are only interested in a single platform app and you have deep knowledge of that platform's tools and languages. *For example, if you're working with a highly-customized iOS app based on **CloudKit** that uses all the native hardware, **MLKit**, **StoreKit**, extensions and so on, maintaining and taking advantage of those features will be easier using **SwiftUI**. Of course, the same goes for a heavily-biased Android app using Jetpack Compose.*

- **Certain platforms**

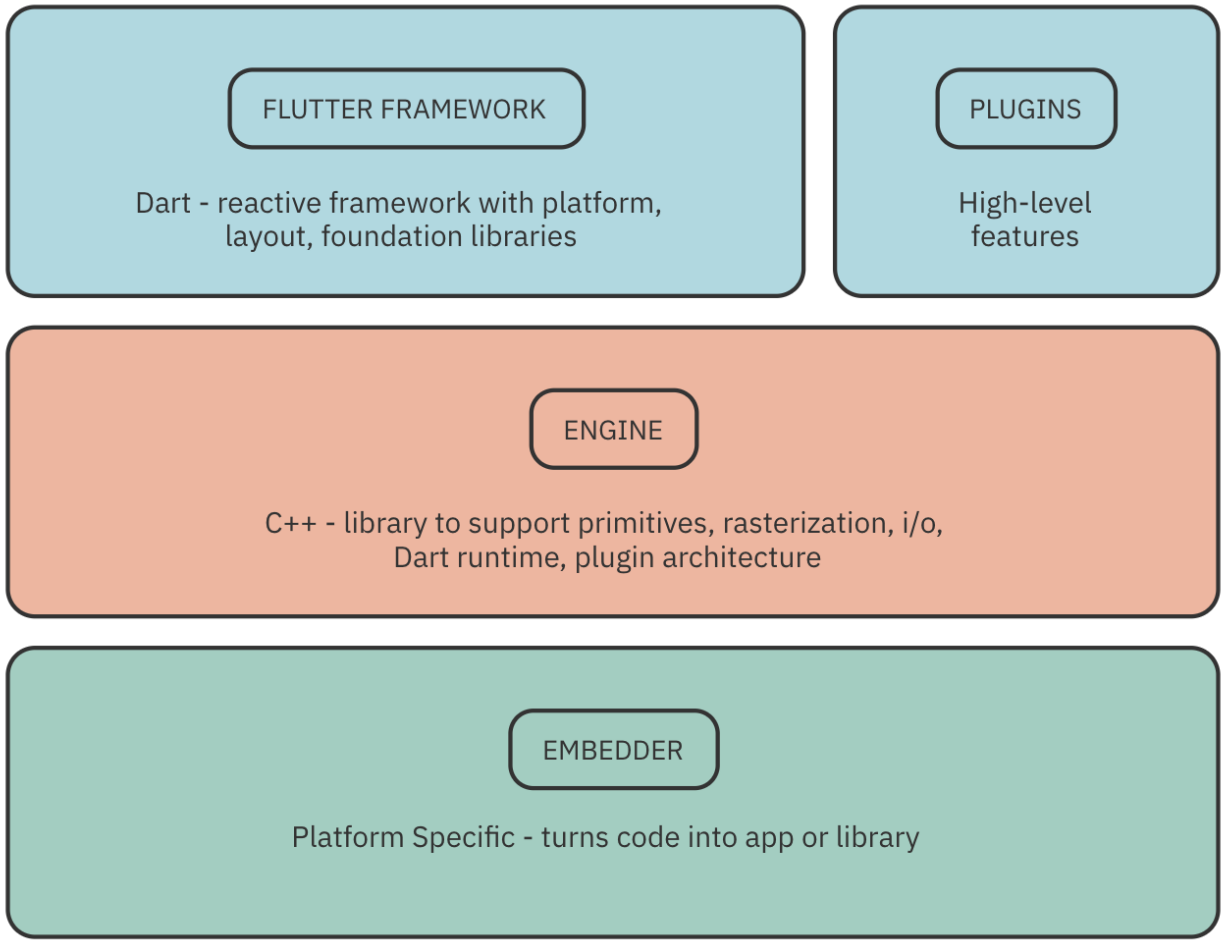
Flutter doesn't run everywhere. It doesn't support Apple **Bitcode** yet, which means that it doesn't support watchOS, tvOS or certain iOS app extensions. Its support for the web is still in its early days, which means that Flutter has many features and performance improvements ahead of it — but they're coming.

# FLUTTER'S HISTORY

- Flutter comes from a tradition of trying to improve web performance. It's built on top of several open-source technologies developed at Google to bring native performance and modern programming to the web through Chrome.
- Flutter is an open source software development kit (SDK) created by Google in 2015 with the name "**Sky**". Its alpha version was on May 2017 and it came to light on **December 11, 2019** with its release version *Flutter 1.12*.
- The Flutter team chose the Dart language, which **Google** also developed, for its productivity enhancements. Its object-oriented type system and support for reactive and asynchronous programming give it clear advantages over **Javascript**. Most importantly, Google built the Dart VM into the Chrome browser, allowing web apps written in Dart to run at native speeds.
- Another piece of the puzzle is the inclusion of **Skia** as the graphics rendering layer. **Skia** is another Google-based open source project that powers the graphics on Android, Chrome browsers, Chrome OS and Firefox. It runs directly on the GPU using **Vulcan** on Android and **Metal** on iOS, making the graphics layer fast on mobile devices. Its API allows Flutter widgets to render quickly and consistently, regardless of the host platform.

# THE FLUTTER ARCHITECTURE

- Flutter has a modular, **4** layered architecture.
- This allows you to write your application logic once and have consistent behavior across platforms, even though the underlying engine code differs depending on the platform.
- The layered architecture also exposes different points for customization and overriding, as necessary.



# THE FLUTTER ARCHITECTURE CONSISTS OF **FOUR MAIN LAYERS**:

1. The **Framework** layer is written in Dart and contains the high-level libraries that you'll use directly to build apps. This includes the UI theme, widgets, layout and animations, gestures and foundational building blocks.
2. Alongside the main Flutter framework are **plugins**: high-level features ***like JSON serialization, geolocation, camera access, in-app payments*** and so on. This plugin-based architecture lets you include only the features your app needs.
3. The **Engine** layer contains the core **C++ libraries that make up the primitives that support Flutter apps**. The engine implements the low-level primitives of the Flutter API, **such as I/O, graphics, text layout, accessibility, the plugin architecture and the Dart runtime**. The engine is also responsible for rasterizing Flutter scenes for fast rendering onscreen.
4. The **Embedder** is different for each target platform and handles packaging the code as a stand-alone app or embedded module.

# MAKEUP OF THE FRAMEWORK LAYER:

The Flutter framework layer consists of **4 sublayers**:

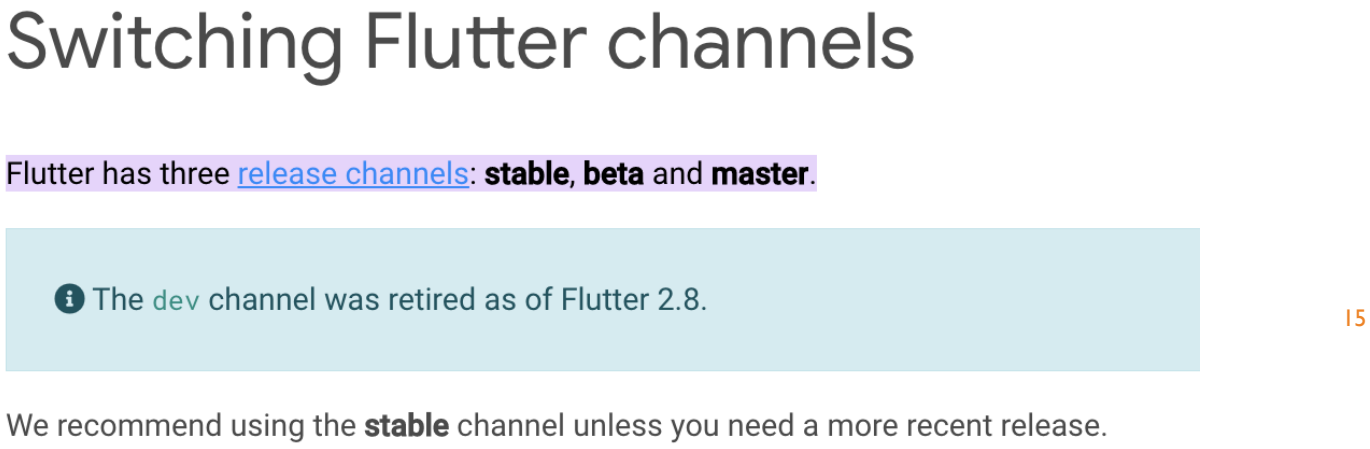
- At the top is the **UI theme**, which uses either the Material (Android) or Cupertino (iOS) design language. This affects how the controls appear, allowing you to make your app look just like a native one.
- The **widget layer** is where you'll spend the bulk of your UI programming time. This is where you compose design and interactive elements to make up the app.
- Beneath the widgets layer is the **rendering layer**, which is the abstraction for building a layout.
- The **foundation layer** provides basic building blocks, like animations and gestures, that build up the higher layers.

# GETTING THE FLUTTER SDK

- The first step is to download the SDK. You can follow the steps on **flutter.dev** or jump right in here: https://flutter.dev/docs/development/tools/sdk/releases

- One thing to note is that Flutter organizes its SDK around **channels**, which are different development branches. New features or platform support will be available first on a **beta channel** for **developers** to try out. This is a great way to get early access to certain features like **new platforms or native SDK support.**

- For this book and development in general, use the **stable channel**. That branch has been vetted and tested and has little chance of breaking.



# GETTING EVERYTHING ELSE

- in addition to the Flutter SDKs, you'll need Java, the Android SDK, the iOS SDKs and an IDE with Flutter extensions. To make this process easier, Flutter includes the **Flutter Doctor, which guides you through installing all the missing tools.**
- **Run : flutter doctor**

That checks for all the necessary components and provides the links or ir to download ones you're missing.

Here's an example:

```
Doctor summary (to see all details, run flutter doctor -v):
[✓] Flutter (Channel stable, 2.5.1, on macOS 11.5 20G71 darwin-x64,
locale en-US)
[✗] Android toolchain - develop for Android devices
    ✗ Flutter requires Android SDK 30 and the Android BuildTools
30.0.2
      To update using sdkmanager, run:
        "/Users/michael/Library/Android/sdk/tools/bin/sdkmanager"
        "platforms;android-30" "build-tools;30.0.2"
      or visit https://flutter.dev/docs/get-started/install/macos
      for detailed instructions.
[!] Xcode - develop for iOS and macOS (Xcode 12.5.1)
    ✗ CocoaPods not installed.
        CocoaPods is used to retrieve the iOS platform side's plugin
        code that responds to your plugin usage on the Dart side.
        Without CocoaPods, plugins will not work on iOS or macOS.
        For more info, see https://flutter.dev/platform-plugins
      To install:
        sudo gem install cocoapods
[✗] Chrome - develop for the web (Cannot find Chrome executable at
    /Applications/Google Chrome.app/Contents/MacOS/Google Chrome)
    ! Cannot find Chrome. Try setting CHROME_EXECUTABLE to a Chrome
executable.
[!] Android Studio (not installed)

[☠] Connected device (the doctor check crashed)
    ✗ Due to an error, the doctor check did not complete. If the
error message below is not helpful, please let us know
      about this issue at https://github.com/flutter/flutter/issues.
    ✗ Exception: Unable to run "adb", check your Android SDK
installation and ANDROID_HOME environment variable:
      /Users/michael/Library/Android/sdk/platform-tools/adb

! Doctor found issues in 4 categories.
```

# EXERCISE: INSTALL AND CONFIGURE FLUTTER

- **Challenge**: Install Flutter SDK, set up your development environment, and run the **flutter doctor** command to ensure your system is properly configured.
- **Question**: What does the flutter doctor command do, and why is it essential for setting up a Flutter environment?

# KEY POINTS

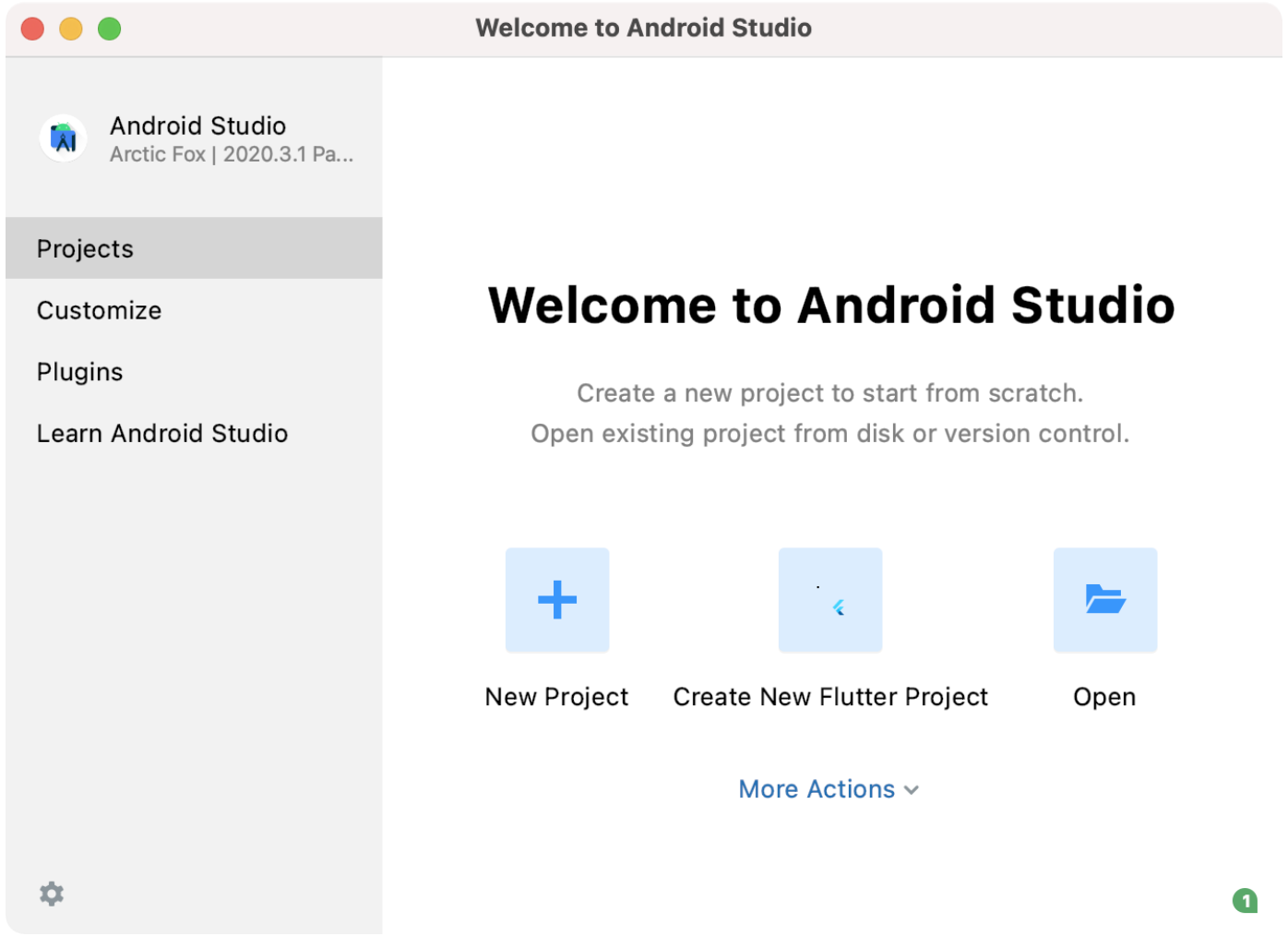
- Flutter is a **software development toolkit** from Google for building **cross-platform apps** using the **Dart** programming language.
- With Flutter, you can build a **high-quality app** that's performant and looks great, very quickly.
- Flutter is for both **new** and **experienced developers** who want to start a mobile app with minimal overhead.
- Install the **Flutter SDK** and associated tools using instructions found at https://flutter.dev.
- The **flutter doctor** command helps you **install** and **update** your Flutter tools.
- This **course** will mostly use **Visual Studio Code** and **Android Studio** as the **IDEs** for Flutter development.

# SETTING UP AN IDE

The Flutter team officially supports three editors:

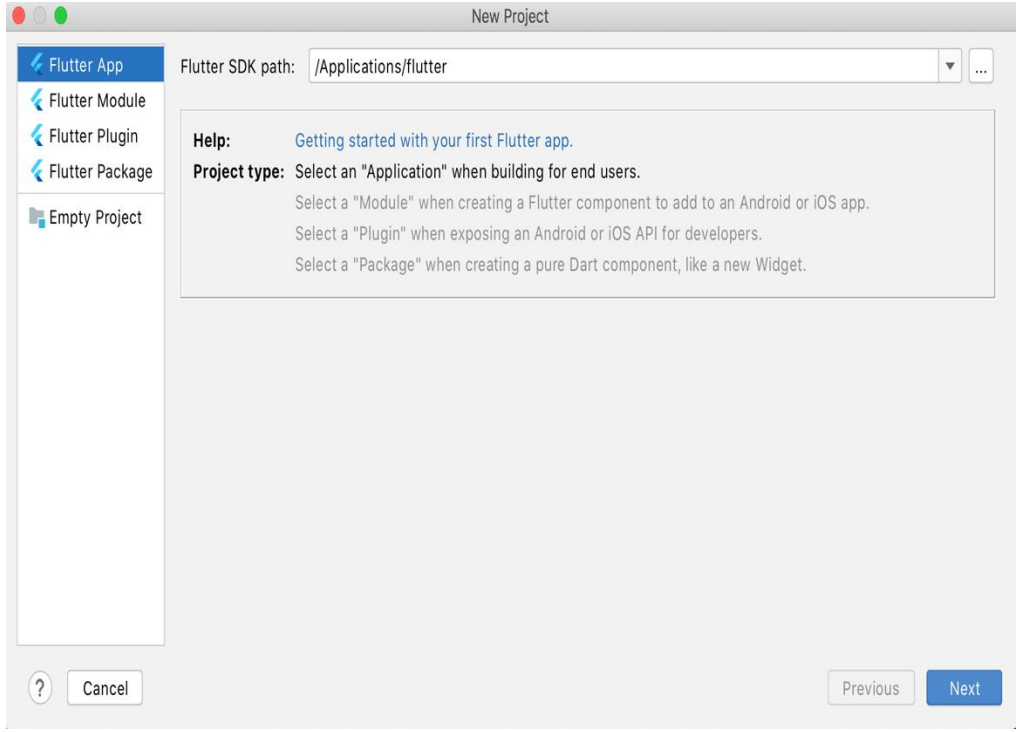
**Android Studio, Visual Studio Code and Emacs**.

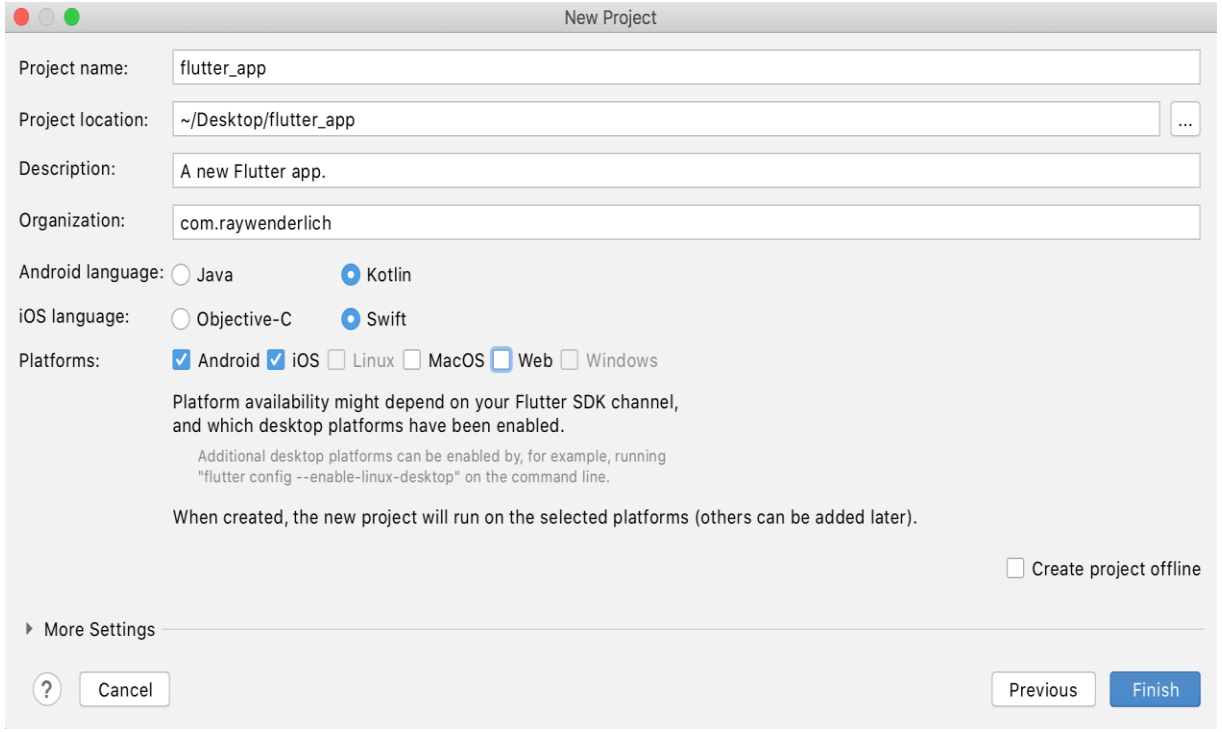
However, there are many other editors that support the Dart language, work with the Flutter command line or have third-party Flutter plugins.



# TRYING IT OUT

- Downloading all the components is the hardest part of getting a Flutter app up and running. Next, you'll try actually building an app.
- There are **two recommended ways to create a new project**: with the **IDE** or through the **flutter command-line tool in a terminal.** In this chapter, you'll use the IDE shortcut.



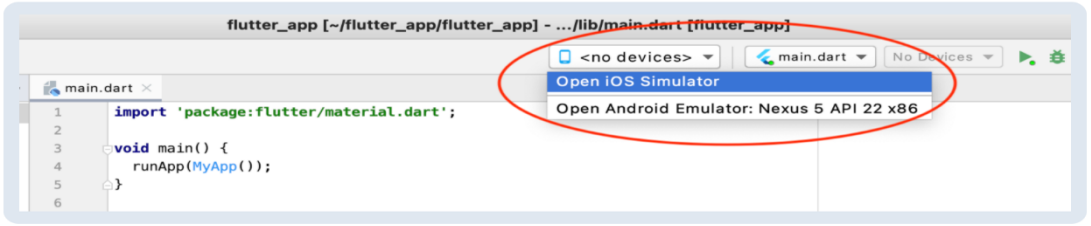
# THE TEMPLATE PROJECT



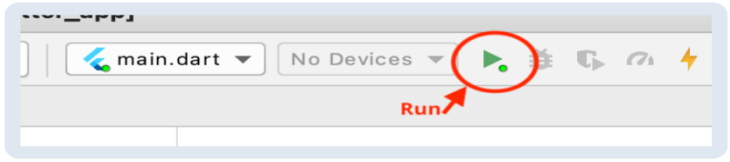
## The template project

The default new project is the same in either editor. It's a simple Flutter demo. The demo app counts the number of times you tap a button.

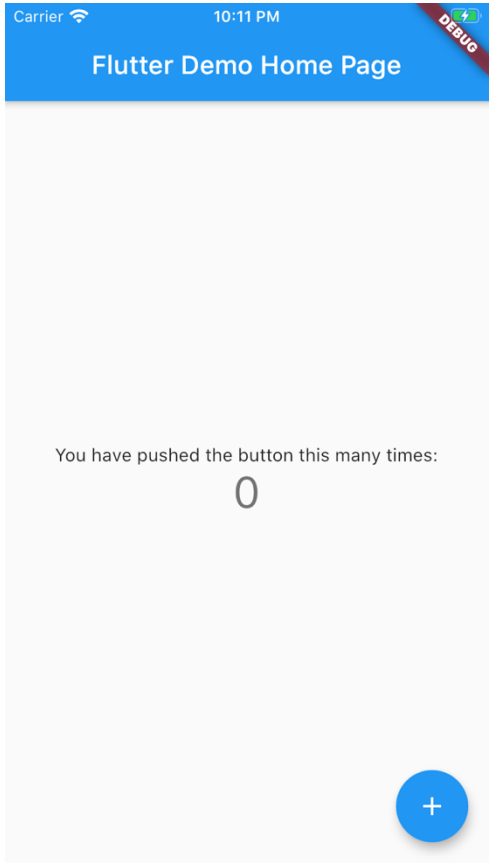
To give it a try, select a connected device, an iOS simulator or an Android emulator.



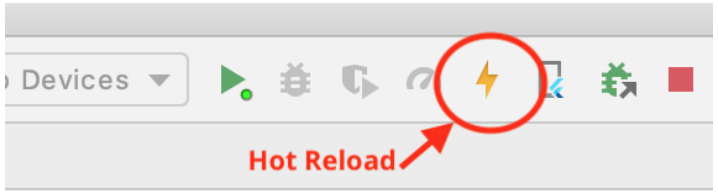

Launch the app by clicking the **Run** icon:




It might take a while to compile and launch the first time. When you're done, you'll see the following:



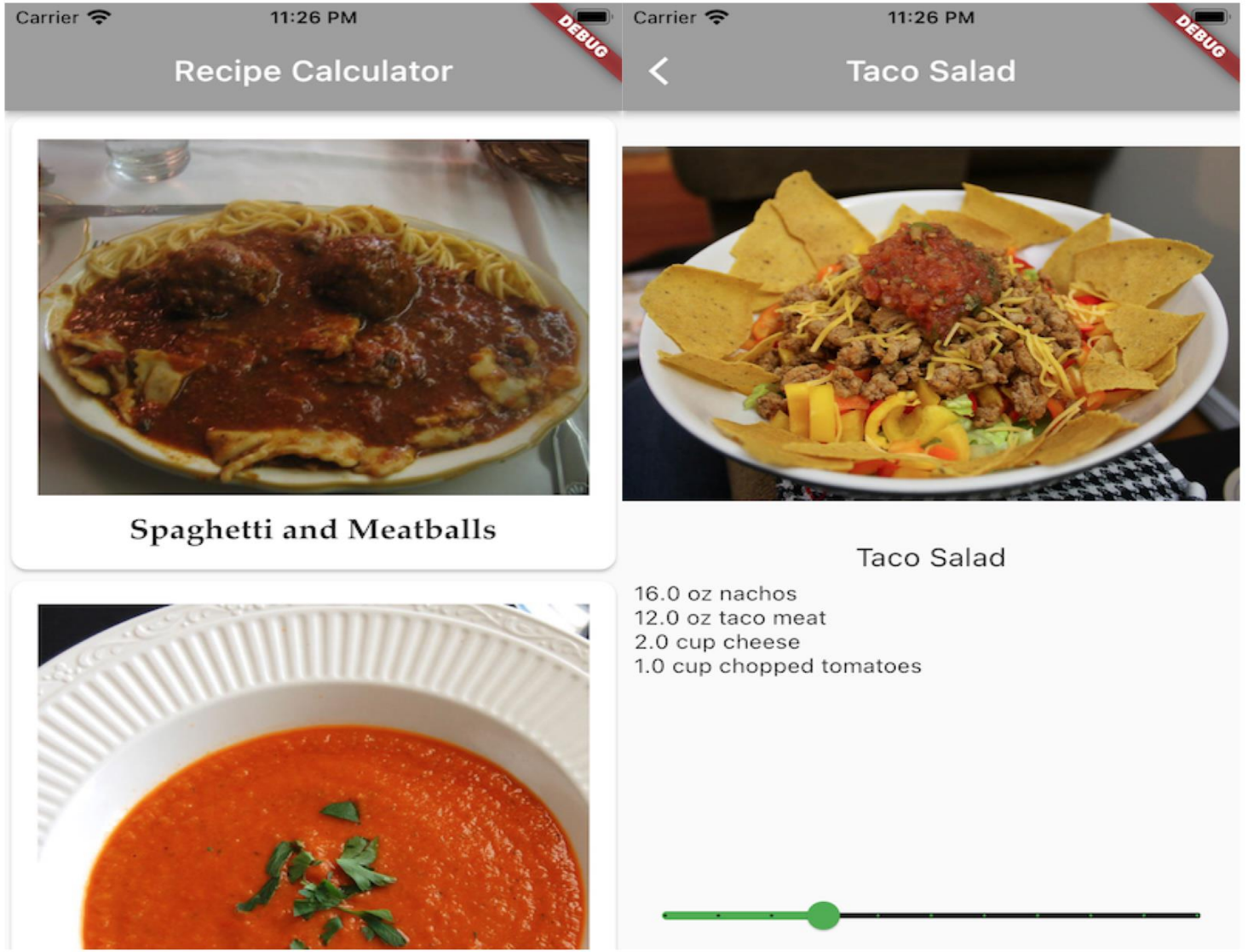

**Bonus: Try hot reload**




All the code for this app is in **lib\main.dart** in the default project.
Feel free to take a look at it. Throughout the rest of this book, you'll dive into *Flutter apps, widgets, state, themes* and many other concepts that will help you build beautiful apps.

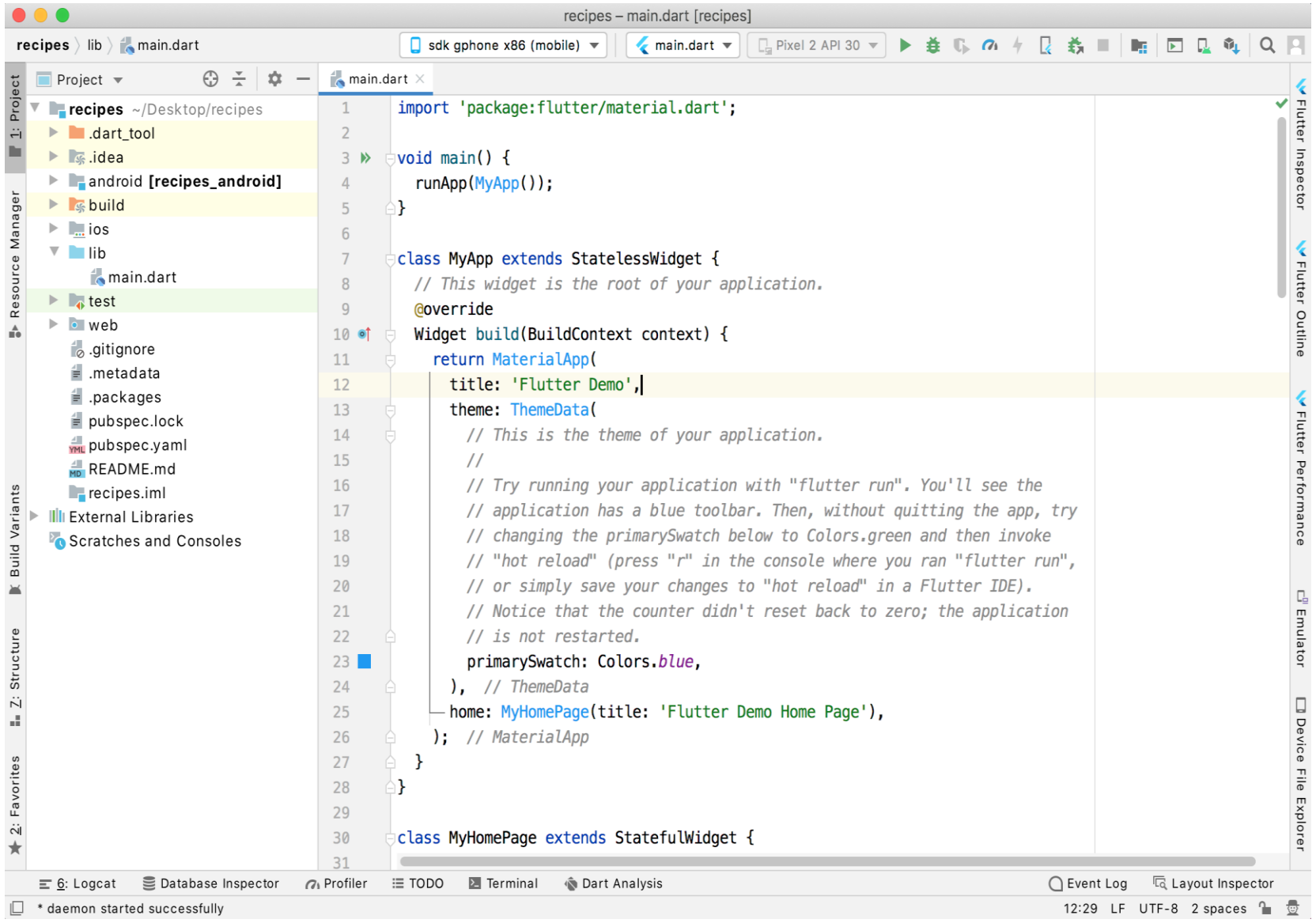
# SECTION1: CHAPTER 2 HELLO FLUTTER

- Your first task is to build a basic app from scratch, giving you the chance to get the hang of the tools and the basic Flutter app structure. You'll customize the app and find out how to use a few popular widgets
  like **ListView** and **Slider** to update its **UI** in response to changes.
- Creating a simple app will let you see just how quick and easy it is to build cross-platform apps with Flutter — and it will give you a quick win.
- By the end of the chapter, you'll have built a **lightweight recipe app**. Since you're just starting to learn **Flutter**, your app will offer a **hard-coded list** of **recipes** and let you use a **Slider** to recalculate quantities based on the number of servings.
- Here's what your finished app will look like:

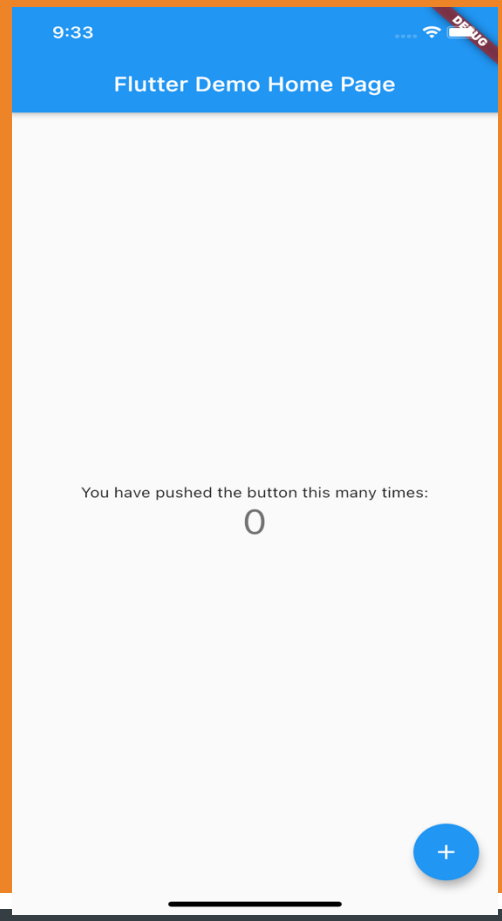



# LIGHTWEIGHT RECIPE APP

Creating a new project is straightforward, by click on new Flutter Project then name it as **recipes**




# CREATING A NEW APP




Build and run and you'll see the same demo app as in Chapter 1, "Getting Started".

The ready-made app is a good place to start because the `flutter create` command puts all the boilerplate together for you to get up and running. But this is not *your* app. It's literally **MyApp**, as you can see near the top of **main.dart**:

```
class MyApp extends StatelessWidget {
```

COPY

This defines a new Dart `class` named `MyApp` which **extends** — or inherits from — `StatelessWidget`. In Flutter, almost everything that makes up the user interface is a **Widget**. A `StatelessWidget` doesn't change after you build it. You'll learn a lot more about widgets and state in the next section. For now, just think of `MyApp` as the container for the app.

Since you're building a recipe app, you don't want your main `class` to be named `MyApp` — you want it to be `RecipeApp`.

While you could change it manually in multiple places, you'll reduce the chance of a copy-and-paste error or typo by using the IDE's **rename** action instead. This lets you rename a symbol at its definition and all its callers at the same time.

# MAKING THE APP YOURS

# STYLING YOUR APP

```
// 1
@override
Widget build(BuildContext context) {
  // 2
  final ThemeData theme = ThemeData();
  // 3
  return MaterialApp(
    // 4
    title: 'Recipe Calculator',
    // 5
    theme: theme.copyWith(
        colorScheme: theme.colorScheme.copyWith(
            primary: Colors.grey,
            secondary: Colors.black,
        ),
    ),
    // 6
    home: const MyHomePage(title: 'Recipe Calculator'),
  );
}
```

This code changes the appearance of the app:

- A widget's **build**() method is the entry point for composing together other widgets to make a new widget.
- A theme determines visual aspects like color. The default **ThemeData** will show the standard Material defaults.
- **MaterialApp** uses Material Design and is the widget that will be included in **RecipeApp**.
- The title of the app is a description that the device uses to identify the app. The UI won't display this.
- By copying the theme and replacing the color scheme with an updated copy lets you change the app's colors. Here, the primary color is Colors.grey and the secondary color is **Colors.black**.
- This still uses the same **MyHomePage** widget as before, but now, you've updated the title and displayed it on the **device**.

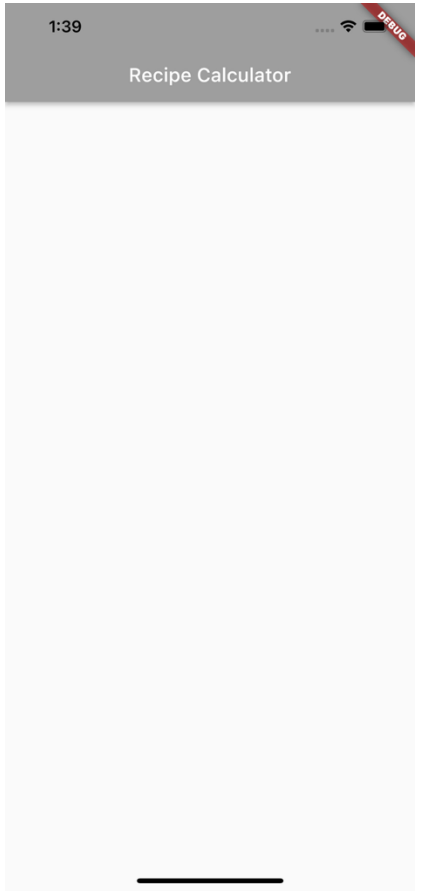
# CLEARING THE APP

A quick look at what this shows:

- A **Scaffold** provides the high-level structure for a screen. In this case, you're using two properties.
- **AppBar** gets a title property by using a Text widget that has a title passed in from home: MyHomePage(title: 'Recipe Calculator') in the previous step.
- **body** has **SafeArea**, which keeps the app from getting too close to the operating system interfaces such as the notch or interactive areas like the Home Indicator at the bottom of some iOS screens.
- **SafeArea** has a child widget, which is an empty Container widget.
- One hot reload later, and you're left with a clean app:

```
class _MyHomePageState extends State<MyHomePage> {
  @override
  Widget build(BuildContext context) {
    // 1
    return Scaffold(
      // 2
      appBar: AppBar(
        title: Text(widget.title),
      ),
      // 3
      body: SafeArea(
        // TODO: Replace child: Container()
        // 4
        child: Container(),
        ),
    );
  }

  // TODO: Add buildRecipeCard() here
}
```



# BUILDING A RECIPE LIST

- An empty recipe app isn't very useful. The app should have a nice list of recipes for the user to scroll through. Before you can display these, however, you need the data to fill out the UI.

**Adding a data model**

- You'll use Recipe as the main data structure for recipes in this app.
- Create a new **Dart file** in the **lib** folder, named **recipe.dart**.
- Add the following class to the file:

```
class Recipe {
  String label;
  String imageUrl;
  // TODO: Add servings and ingredients here

  Recipe(
    this.label,
    this.imageUrl,
  );
  // TODO; Add List<Recipe> here
}

// TODO: Add Ingredient() here
```

# BUILDING A RECIPE LIST

This is the start of a **Recipe model** with a label and an **image**.

- You'll also need to supply some data for the app to display. In a ***full-featured app, you'd load this data either from a local database or a JSON-based API***. For the sake of simplicity as you get started with Flutter, however, you'll use **hard-coded data in this chapter**.
- Add the following method to **Recipe** by replacing

// TODO: **Add List<Recipe>** here with :

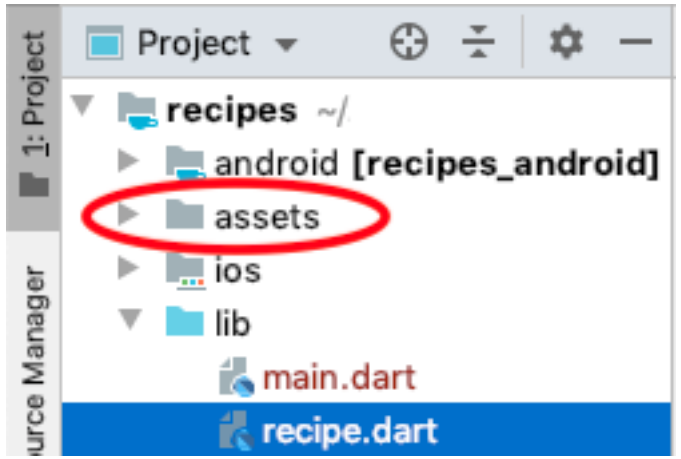
- This is a hard-coded list of recipes. You'll add more detail later, but right now, it's just a list of **names and images**.



```
static List<Recipe> samples = [
  Recipe(
    'Spaghetti and Meatballs',
    'assets/2126711929_ef763de2b3_w.jpg',
  ),
  Recipe(
    'Tomato Soup',
    'assets/27729023535_a57606c1be.jpg',
  ),
  Recipe(
    'Grilled Cheese',
    'assets/3187380632_5056654a19_b.jpg',
  ),
  Recipe(
    'Chocolate Chip Cookies',
    'assets/15992102771_b92f4cc00a_b.jpg',
  ),
  Recipe(
    'Taco Salad',
    'assets/8533381643_a31a99e8a6_c.jpg',
  ),
  Recipe(
    'Hawaiian Pizza',
    'assets/15452035777_294cefced5_c.jpg',
  ),
];
```

# BUILDING A RECIPE LIST

- You've created a List with images, but you don't have any images in your project yet. To add them, go to **Finder and copy the assets folder from the top level of 02-hello-flutter** in the book materials of your project's folder structure. When you're done, it should live at the same level as the lib folder. That way, the app will be able to find the images when you run it.
- You'll notice that by copy-pasting in Finder, the folder and images automatically display in the Android Studio project list.




But just adding assets to the project doesn't display them in the app. To tell the app to include those assets, open **pubspec.yaml** in the **recipes** project root folder.

Under `# To add assets to your application...` add the following lines:

```
assets:
  - assets/
```

COPY

These lines specify that **assets/** is an assets folder and must be included with the app. Make sure that the first line here is aligned with the `uses-material-design: true` line above it.

# DISPLAYING THE LIST

- With the data ready to go, your next step is to create a place for the data to go **to**.
- Back in **main.dart**, you need to import the data file so the code in **main.dart** can find it. Add the following to the top of the file, under the other import lines:

```
import 'recipe.dart';
```

- Next, in **_MyHomePageState SafeArea's** child, find and replace // TODO: Replace child: Container() and the two lines beneath it with:

This code does the following:

- Builds a list using **ListView**.
- **itemCount** determines the number of rows the list has. In this case, length is the number of objects in the **Recipe.samples** list.
- **itemBuilder** builds the widget tree for each row.
- A **Text widget** displays the name of the recipe.

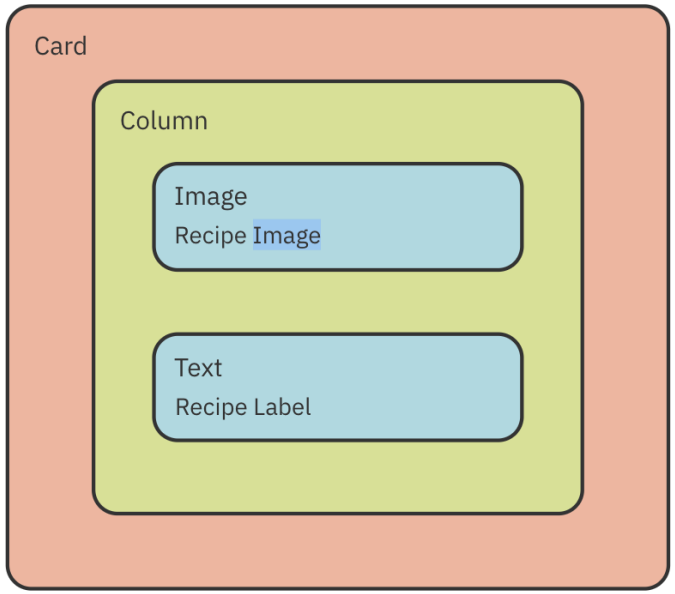
```
// 4
child: ListView.builder(
  // 5
  itemCount: Recipe.samples.length,
  // 6
  itemBuilder: (BuildContext context, int index) {
    // 7
    // TODO: Update to return Recipe card
    return Text(Recipe.samples[index].label);
  },
),
```

# DISPLAYING THE LIST


Perform a hot reload now and you'll see the following list:

# PUTTING THE LIST INTO A CARD

- It's great that you're displaying real data now, but this is barely an app. To spice things up a notch, you need to add images to go along with the titles.
- To do this, you'll use a **Card**. In **Material Design**, **Cards define an area of the UI where you've laid out related information about a specific entity**. For example, a **Card** in a music app might have labels for an album's *title, artist and release date along with an image for the album cover and maybe a control for rating it with stars*.
- Your **recipe Card will show the recipe's label and image**. Its widget tree will have the following structure:




In **main.dart**, at the bottom of `_MyHomePageState` create a **custom widget** by replacing `// TODO: Add buildRecipeCard() here` with:

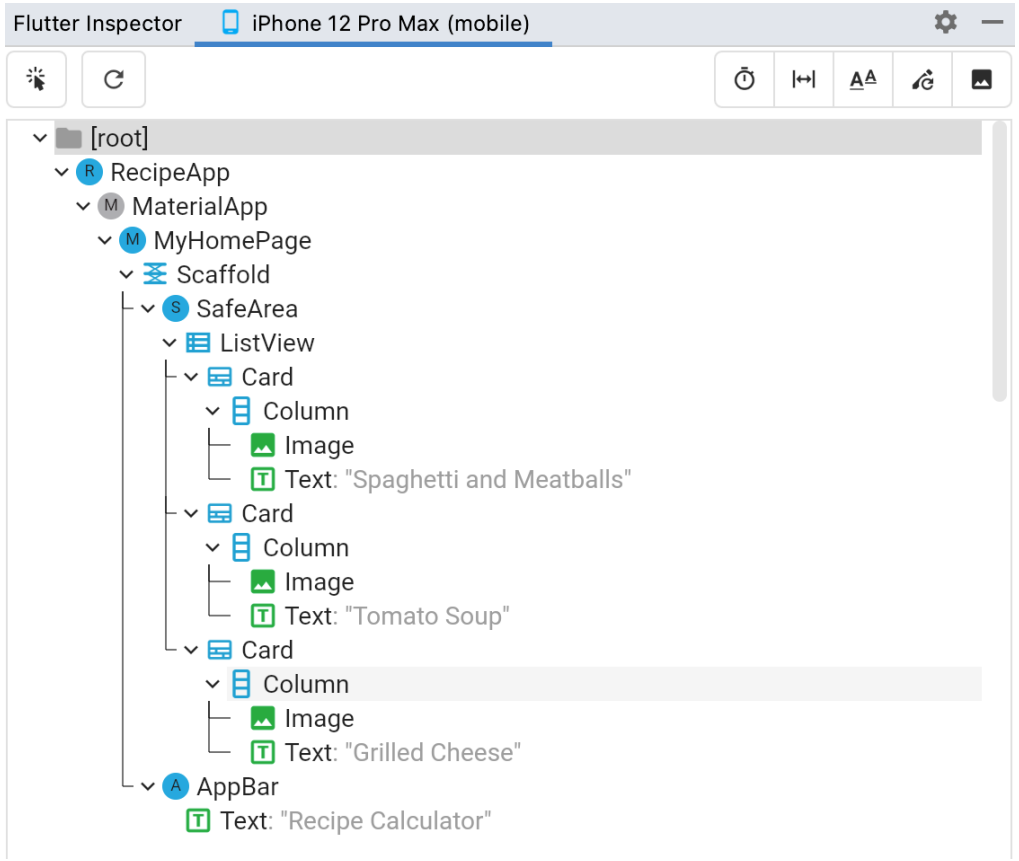
```
Widget buildRecipeCard(Recipe recipe) {
  // 1
  return Card(
    // 2
      child: Column(
        // 3
        children: <Widget>[
          // 4
          Image(image: AssetImage(recipe.imageUrl)),
          // 5
          Text(recipe.label),
        ],
      ),
  );
}
```

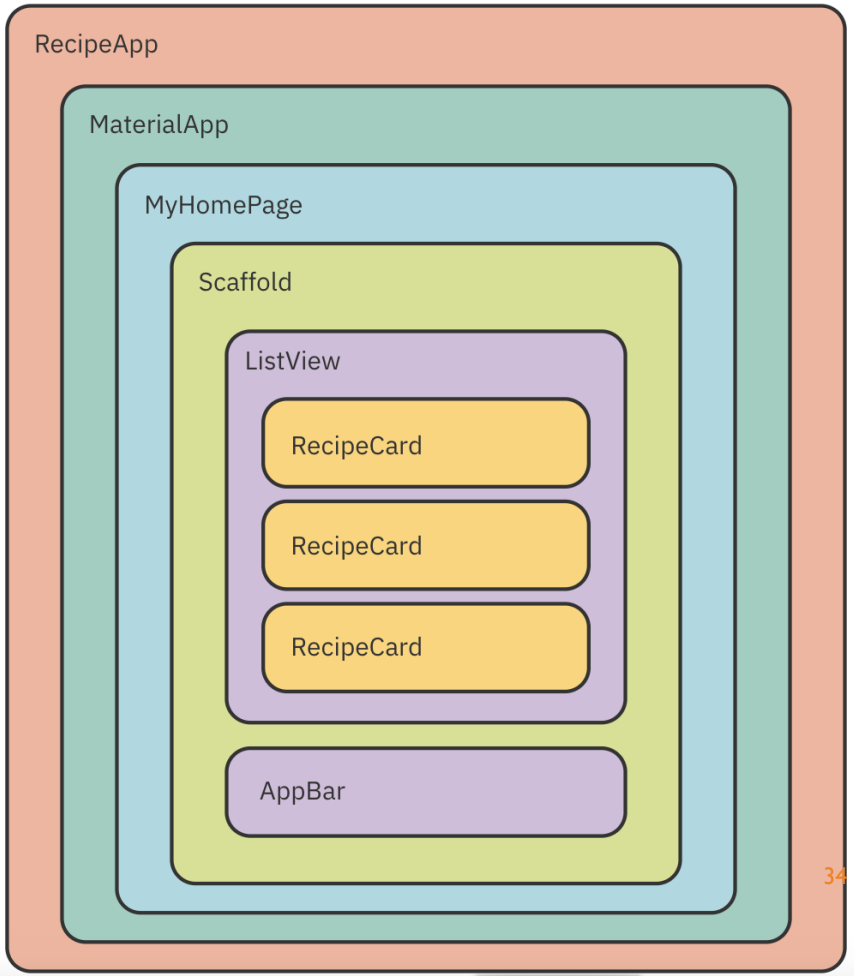
COPY

# LOOKING AT THE WIDGET TREE

Now's a good time to think about the widget tree of the overall app. Do you remember that it started with RecipeApp from main()?

To **see your App Widget Tree**, in Android Studio, open the **Flutter Inspector** from the **View ▸ Tool Windows ▸ Flutter Inspector** menu while your app is running. This opens a powerful UI debugging tool.



# MAKING IT LOOK NICE

- The default cards look okay, but they're not as nice as they could be.
- With a few added extras, you can spiffy the card up.
- These include wrapping widgets in layout widgets like **Padding** or specifying additional styling parameters.
- Get started by replacing **buildRecipeCard**() with:

```
Widget buildRecipeCard(Recipe recipe) {
  return Card(
    // 1
    elevation: 2.0,
    // 2
    shape: RoundedRectangleBorder(
      borderRadius: BorderRadius.circular(10.0)),
    // 3
    child: Padding(
      padding: const EdgeInsets.all(16.0),
      // 4
      child: Column(
        children: <Widget>[
          Image(image: AssetImage(recipe.imageUrl)),
          // 5
          const SizedBox(
            height: 14.0,
          ),
          // 6
          Text(
            recipe.label,
            style: const TextStyle(
              fontSize: 20.0,
              fontWeight: FontWeight.w700,
              fontFamily: 'Palatino',
            ),
          )
        ],
      ),
    ),
  );
}
```

# MAKING IT LOOK NICE

This last slide has a few updates to look at:

- A card's elevation determines **how high off the screen** the card is, affecting its shadow.
- shape handles the shape of the card. This code defines a rounded rectangle with a 10.0 corner radius.
- Padding insets its child's contents by the specified padding value.
- The padding child is still the same vertical Column with the image and text.
- Between the image and text is a SizedBox. This is a blank view with a fixed size.
- You can customize Text widgets with a style object. In this case, you've specified a Palatino font with a size of 20.0 and a bold weight of w700.

Hot reload and you'll see a more styled list.

# ADDING A RECIPE DETAIL PAGE

- You now have a pretty list, but the app isn't interactive yet. What would make it great is to show the user details about a recipe when they tap the card. You'll start implementing this by making the card react to a tap.

**Making a tap response**

- Inside _MyHomePageState, locate // TODO: Add **GestureDetector** and replace the return statement beneath it with the following:

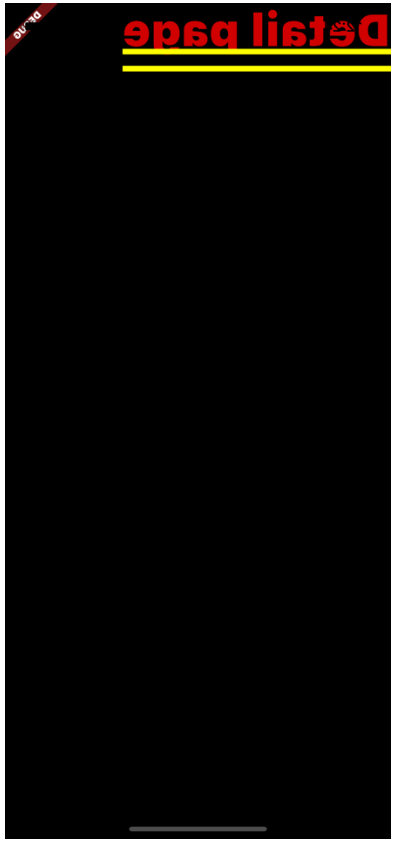
```
// 7
return GestureDetector(
  // 8
  onTap: () {
    // 9
    Navigator.push(
      context,
      MaterialPageRoute(
        builder: (context) {
        // 10
        // TODO: Replace return with return RecipeDetail()
        return Text('Detail page');
      },
    ),
   );
  },
  // 11
  child: buildRecipeCard(Recipe.samples[index]),
);
```

# ADDING A RECIPE DETAIL PAGE

Last Slide Note introduces a few new widgets and concepts. Looking at the lines one at a time:

- Introduces a **GestureDetector** widget, which, as the name implies, detects gestures.
- Implements an **onTap** function, which is the callback called when the widget is tapped.
- The Navigator widget manages a stack of pages. Calling push() with a **MaterialPageRoute** will **push** a new Material page onto the stack.
- Section III, "**Navigating Between Screens**", will cover navigation in a lot more detail.
- **builder** creates the destination page widget.
- **GestureDetector's** child widget defines the area where the gesture is active.

Hot reload the app and now each card is tappable. **Tap** a recipe and you'll see a black **Detail page**:



# CREATING AN ACTUAL TARGET PAGE

- The resulting page is obviously just a placeholder. Not only is it ugly, but because it doesn't have all the normal page trappings, the user is now stuck here, at least on iOS devices without a back button. But don't worry, you can fix that!
- In **lib**, create a new **Dart file** named **recipe_detail.dart**.
- Now, add this code to the file, ignore the red squiggles:

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

class RecipeDetail extends StatefulWidget {
  final Recipe recipe;

  const RecipeDetail({
    Key? key,
    required this.recipe,
  }) : super(key: key);

  @override
  _RecipeDetailState createState() {
    return _RecipeDetailState();
  }
}

// TODO: Add _RecipeDetailState here
```

# CREATING AN ACTUAL TARGET PAGE

- This creates a new **StatefulWidget** which has an initializer that takes the Recipe details to display. This is a **StatefulWidget** because you'll add some interactive state to this page later.
- You need **_RecipeDetailState** to build the widget, replace

// TODO: Add **_RecipeDetailState** here with:

```
class _RecipeDetailState extends State<RecipeDetail> {
  // TODO: Add _sliderVal here

  @override
  Widget build(BuildContext context) {
    // 1
    return Scaffold(
      appBar: AppBar(
        title: Text(widget.recipe.label),
      ),
      // 2
      body: SafeArea(
        // 3
        child: Column(
          children: <Widget>[
            // 4
            SizedBox(
              height: 300,
              width: double.infinity,
              child: Image(
                image: AssetImage(widget.recipe.imageUrl),
              ),
            ),
            // 5
            const SizedBox(
              height: 4,
            ),
            // 6
            Text(
              widget.recipe.label,
              style: const TextStyle(fontSize: 18),
            ),
            // TODO: Add Expanded

            // TODO: Add Slider() here
          ],
        ),
      ),
    );
  }
}
```

# CREATING AN ACTUAL TARGET PAGE

The body of the widget is the same as you've already seen. Here are a few things to notice:

1. **Scaffold** defines the page's general structure.
2. In the body, there's a **SafeArea**, a Column with a Container, a **SizedBox** and **Text** children.
3. **SafeArea** keeps the app from getting too close to the operating system interfaces, such as the notch or the interactive area of most iPhones.
4. One new thing is the **SizedBox** around the Image, which defines resizable bounds for the image. Here, the height is fixed at 300 but the width will adjust to fit the aspect ratio. The unit of measurement in Flutter is **logical pixels**.
5. There is a spacer **SizedBox**.
6. The **Text** for the label has a style that's a little different than the main Card, to show you how much customizability is available.

Next, go back to **main.dart** and add the following line to the top of the file:

```
import 'recipe_detail.dart';
```

# CREATING AN ACTUAL TARGET PAGE

Then find `// TODO: Replace return with return RecipeDetail()` replace it and the existing `return` statement with:

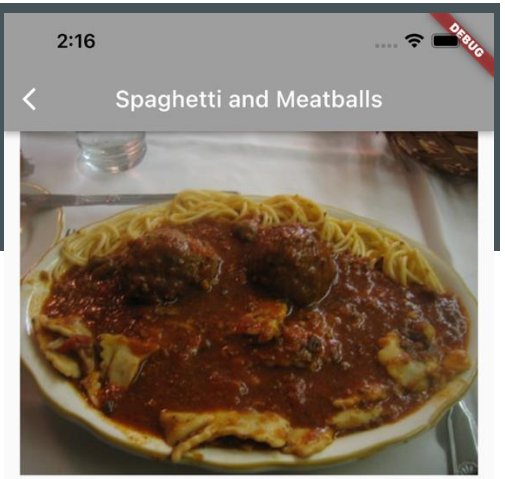
```
return RecipeDetail(recipe: Recipe.samples[index]);
```

COPY

Perform a hot restart by choosing **Run ▸ Flutter Hot Restart** from the menu to set the app state back to the original list. Tapping a recipe card will now show the RecipeDetail page.

**Note**: You need to use hot restart here because hot reload won't update the UI after you update the state.

Because you now have a **Scaffold** with an **appBar**, Flutter will automatically include a back button to return the user to the main list.



Spaghetti and Meatballs

# ADDING INGREDIENTS

- To complete the detail page, you'll need to add additional details to the Recipe class. Before you can do that, you have to add an ingredient list to the recipes.
- Open **recipe.dart** and replace // TODO: Add Ingredient() here with the following class:
- This is a simple data container for an ingredient. It has a name, a unit of measure — like "cup" or "tablespoon" — and a quantity.
- At the top of the Recipe class, replace

// TODO: Add servings and ingredients here with the following:

```
int servings;
List<Ingredient> ingredients;
```

```
class Ingredient {
  double quantity;
  String measure;
  String name;

  Ingredient(
    this.quantity,
    this.measure,
    this.name,
  );
}
```

# ADDING INGREDIENTS

- This adds properties to specify that **serving** is how many people the specified quantity feeds and **ingredients** is a simple list.
- To use these new properties, go to your **samples** list inside the **Recipe** class and change the **Recipe** constructor from:

```
Recipe(
  this.label,
  this.imageUrl,
);
```

to:

```
Recipe(
  this.label,
  this.imageUrl,
  this.servings,
  this.ingredients,
);
```

# ADDING INGREDIENTS

You'll see red squiggles under part of your code because the values for `servings` and `ingredients` have not been set. You'll fix that next.

```
Recipe(
  this.label,
  this.imageUrl,
  this.servings,
  this.ingredients,
);

static List<Recipe> samples = [
  Recipe(
    'Spaghetti and Meatballs',
    'assets/2126711929_ef763de2b3_w.jpg',
  ),
  Recipe(
    'Tomato Soup',
    'assets/27729023535_a57606c1be.jpg',
  ),
```

To include the new `servings` and `ingredients` properties, replace the existing `samples` definition with the following:

```
static List<Recipe> samples = [
  Recipe(
    'Spaghetti and Meatballs',
    'assets/2126711929_ef763de2b3_w.jpg',
    4,
    [
      Ingredient(1, 'box', 'Spaghetti',),
      Ingredient(4, '', 'Frozen Meatballs',),
      Ingredient(0.5, 'jar', 'sauce',),
    ],
  ),
  Recipe(
    'Tomato Soup',
    'assets/27729023535_a57606c1be.jpg',
    2,
    [
      Ingredient(1, 'can', 'Tomato Soup',),
    ],
  ),
  Recipe(
```

# ADDING INGREDIENTS

That fills out an ingredient list for these items. Please don't cook these at home, these are just examples. :]

Hot reload the app now. No changes will be visible, but it should build successfully.

# SHOWING THE INGREDIENTS

A recipe doesn't do much good without the ingredients. Now, you're ready to add a widget to display them.

In **recipe_detail.dart**, replace `// TODO: Add Expanded` with:

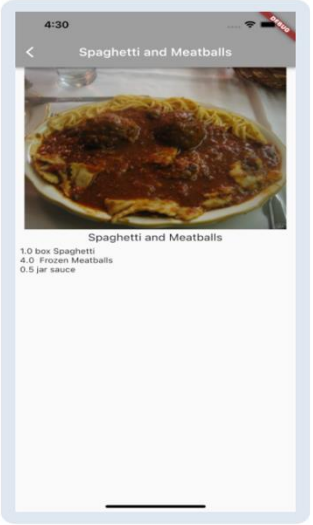
```
// 7
Expanded(
  // 8
  child: ListView.builder(
    padding: const EdgeInsets.all(7.0),
    itemCount: widget.recipe.ingredients.length,
    itemBuilder: (BuildContext context, int index) {
      final ingredient = widget.recipe.ingredients[index];
      // 9
      // TODO: Add ingredient.quantity
      return Text(
          '${ingredient.quantity} ${ingredient.measure}
${ingredient.name}');
    },
  ),
),
```

COPY

This code adds:

7. An `Expanded` widget, which expands to fill the space in a `Column`. This way, the ingredient list will take up the space not filled by the other widgets.
8. A `ListView`, with one row per ingredient.
9. A `Text` that uses **string interpolation** to populate a string with runtime values. You use the `${expression}` syntax inside the string literal to denote these.

Hot restart by choosing **Run ▸ Flutter Hot Restart** and navigate to a detail page to see the ingredients.






Nice job, the screen now shows the recipe name and the ingredients. Next, you'll add a feature to make it interactive.

# ADDING A SERVING SLIDER

You're currently showing the ingredients for a suggested serving. Wouldn't it be great if you could change the desired quantity and have the amount of ingredients update automatically?

You'll do this by adding a **Slider** widget to allow the user to adjust the number of servings.

First, create an instance variable to store the slider value at the top of `_RecipeDetailState` by replacing `// TODO: Add _sliderVal here`:

```
int _sliderVal = 1;
```

COPY

Now find `// TODO: Add Slider() here` replace it with the following:

```
Slider(
  // 10
  min: 1,
  max: 10,
  divisions: 10,
  // 11
  label: '${_sliderVal * widget.recipe.servings} servings',
  // 12
  value: _sliderVal.toDouble(),
  // 13
  onChanged: (newValue) {
    setState(() {
      _sliderVal = newValue.round();
    });
  },
  // 14
  activeColor: Colors.green,
  inactiveColor: Colors.black,
),
```

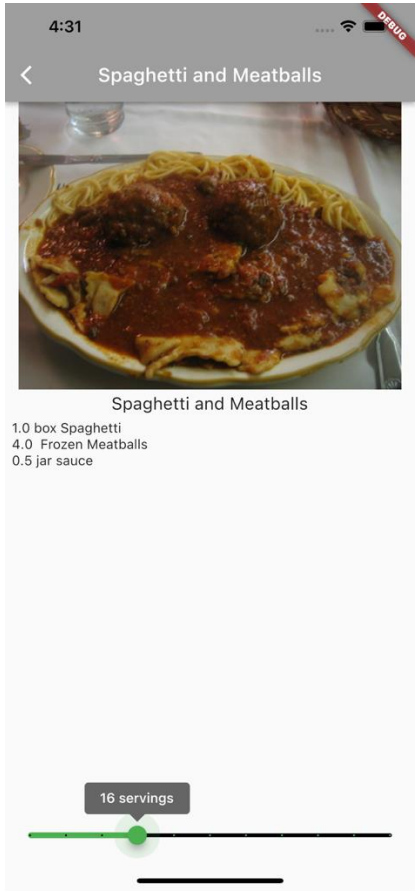
COPY

# ADDING A SERVING SLIDER

`Slider` presents a round thumb that can be dragged along a track to change a value. Here's how it works:

10. You use `min`, `max` and `divisions` to define how the slider moves. In this case, it moves between the values of 1 and 10, with 10 discreet stops. That is, it can only have values of 1, 2, 3, 4, 5, 6, 7, 8, 9 or 10.
11. `label` updates as the `_sliderVal` changes and displays a scaled number of servings.
12. The slider works in `double` values, so this converts the `int` variable.
13. Conversely, when the slider changes, this uses `round()` to convert the `double` slider value to an `int`, then saves it in `_sliderVal`.
14. This sets the slider's colors to something more "on brand". The `activeColor` is the section between the minimum value and the thumb, and the `inactiveColor` represents the rest.

Hot reload the app, adjust the slider and see the value reflected in the indicator.



# UPDATING THE RECIPE

It's great to see the changed value reflected in the slider, but right now, it doesn't affect the recipe itself.

To do that, you just have to change the `Expanded` ingredients `itemBuilder` return statement to include the current value of `_sliderVal` as a factor for each ingredient.

Replace `// TODO: Add ingredient.quantity` and the whole `return` statement beneath it with:

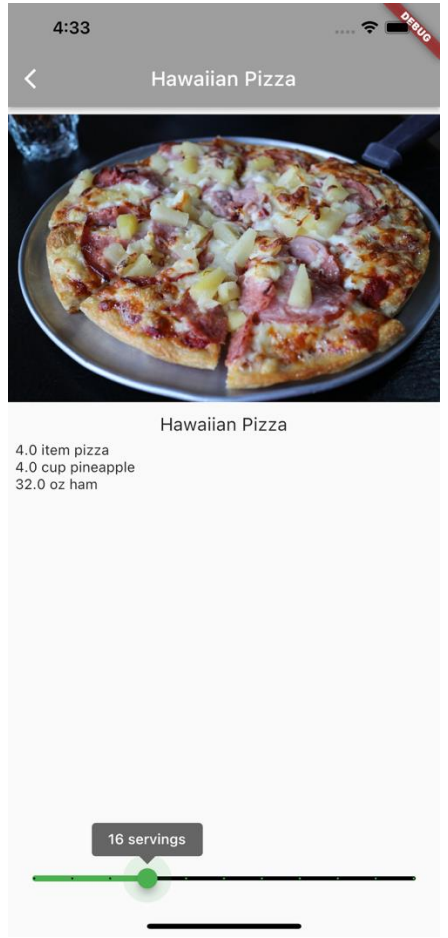
```
return Text('${ingredient.quantity * _sliderVal} '
                    '${ingredient.measure} '
                    '${ingredient.name}');
```

COPY

After a hot reload, you'll see that the recipe's ingredients change when you move the slider.

That's it! You've now built a cool, interactive Flutter app that works just the same on iOS and Android.

In the next few sections, you'll continue to explore how widgets and state work. You'll also learn about important functionality like networking.



# EXERCISE: MODIFYING THE RECIPE APP

- **Challenge 1**: Customize the recipe list by adding your own recipes, images, and descriptions. Replace the hardcoded recipes with at least three of your own, including images.
- **Challenge 2**: Add a "**favorite**" feature that lets users mark a recipe as their favorite and display a star icon beside it in the list.
- **Question**: What are the key differences between **StatelessWidget** and **StatefulWidget** in Flutter?

# INTERACTIVE QUIZ

**Q:**What does the MaterialApp widget do in a Flutter app?

1. Handles the app's routing
2. Specifies the app's theme and appearance
3. Defines the high-level structure of the app
4. All of the above

**Q**:Which widget in Flutter is primarily used to organize the structure of a screen?

1. Scaffold
2. MaterialApp
3. AppBar
4. SafeArea

# EXERCISE: BASIC UI EXPLORATION

- **Challenge**: Add a **"dark mode" toggle** to your recipe app. Switch between **light** and **dark** themes by modifying the **ThemeData** and updating the app's **color scheme**.
- **Question**: How can you use the **SafeArea** widget to avoid screen cutouts like the notch on iPhones?
- **Challenge**: Implement a **GestureDetector** that allows users **to swipe left** or right to navigate between recipes.
- **Challenge**: Recipe App with **CRUD** Operations: Use local storage (like **shared_preferences** or **SQLite**) to save and retrieve recipes.
- **Question**: What is the role of the **GestureDetector** widget in Flutter, and how can it be used to add interactivity to your app?



- **Question**: How does the **Navigator** widget manage transitions between different screens in a Flutter app?

# SUMMARY: KEY TAKEAWAYS FROM FLUTTER DEVELOPMENT

- **Building Your First App:**
  - Use **flutter create** to build new apps.
  - Widgets compose the UI: every UI element is a widget in Flutter.
  - Styling: Customize widgets using parameters like **ThemeData**, **Scaffold**, and **Text**.
- **State Management:**
  - **StatefulWidget** allows for interactive features.
  - **Hot Restart**: Restores app to its initial state.
  - **Hot Reload**: Refreshes UI without losing the current state.
- **Key Widgets:**
  - **MaterialApp**: Defines the app structure and theme.
  - **Scaffold**: Provides the basic layout structure (**AppBar**, **Body**).
  - **Your Recipe App**: Utilized **ListView**, **Card**, and **GestureDetector** to create an interactive list.

# WHERE TO GO FROM HERE?

- Congratulations, you've written your first app!
- To get a sense of all the widget options available, the documentation at https://api.flutter.dev/ should be your starting point. In particular, the Material library https://api.flutter.dev/flutter/material/material-library.html and Widgets library https://api.flutter.dev/flutter/widgets/widgets-library.html will cover most of what you can put onscreen. Those pages list all the parameters, and often have in-browser interactive sections where you can experiment.
- Chapter 3, "Basic Widgets", is all about using widgets and
- Chapter 4, "Understanding Widgets", goes into more detail on the theory behind widgets.
- Future chapters will go into more depth about other concepts briefly introduced in this chapter.