Chapter 2

Hello, Flutter

Recipe App

“you’ll have built a lightweight recipe app”.

Key points

- Build a new app with flutter create.
- Use widgets to compose a screen with controls and layout.
- Use widget parameters for styling.
- **MaterialApp** widget specifies the app, and **Scaffold** specifies the high-level structure of a given screen.
- State allows for interactive widgets.
- When state changes, you usually need to hot restart the app instead of hot reload. In some case, you may also need to rebuild and restart the app entirely.

Steps:

- Creating a new app.
- Making the app yours.
- Styling your app.
- Clearing the app.
- Adding a data model
- Displaying the list
- Putting the list into a card
- Looking at the widget tree
- Making it look nice
- Making a tap response
- Creating an actual target page
- Adding ingredients
- Showing the ingredients
- Adding a serving slider
- Updating the recipe

Creating a new app

There are two simple ways to start a new Flutter app:

- In the last chapter, you created a new app project through the IDE.
- Alternatively, you can create an app with the flutter command.

You'll use the second option here

Creating a new project is straightforward. In the terminal, run

```
flutter create recipes
```

This command creates a new app in a new folder, both named **recipes**.

Making the app yours:

Let's discuss the demo classes.

- **main**() is the entry point for the code when the app launches.
- **runApp**() tells Flutter which is the top-level widget for the app
- **MyApp** class, we can easily rename by right-clicking on, and then (**Refactor ▸ Rename**) menu.
- We'll rename this class into **RecipeApp**.

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'),
  );
}
```

Styling your app:

Let's discuss the demo classes.

- 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.

Styling your app:

Let's discuss the demo classes.

- 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:

```
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
}
```

Styling your app:

- 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.

Building a recipe list:

Adding a data model

Create a new Dart file in the **lib** folder, named **recipe.dart**

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

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

Building a recipe list:

Adding a data model

```
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',
  ),
];
```

Styling your app:

- 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.

Github Link

https://github.com/raywenderlich/flta-materials

Styling your app:

- 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.

```
assets:
  - assets/
```

- 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:

- 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:

Displaying the list:

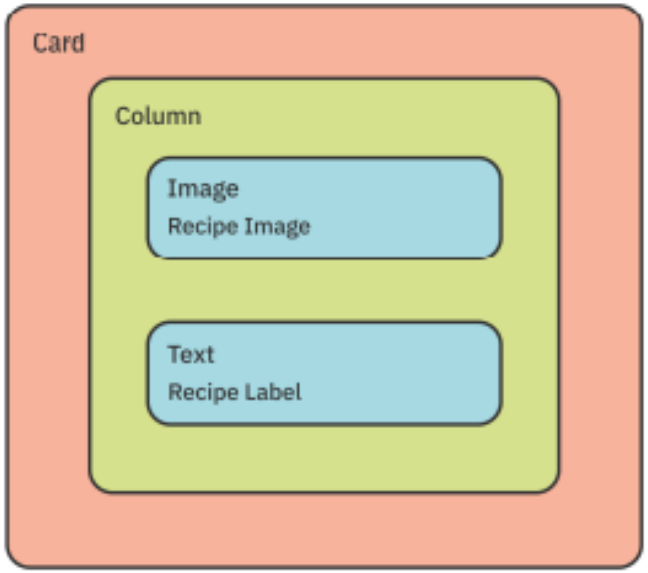
```
// 4
child: ListView.builder(
  // 5
  itemCount: Recipe.samples.length,
  // 6
  itemBuilder: (BuildContext context, int index) {
    // 7
    // TODO: Update to return Recipe card
    return Text(Recipe.samples[index].label);
  },
),
```

Displaying the list:

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

Putting the list into a card:

- In Material Design, **Cards** define an area of the **UI** where you've laid out related information about a specific entity.
- **recipe Card** will show the recipe's label and image. Its widget tree will have the following structure:



Putting the list into a card:

- 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),
        ],
      ),
  );
}
```

Putting the list into a card:

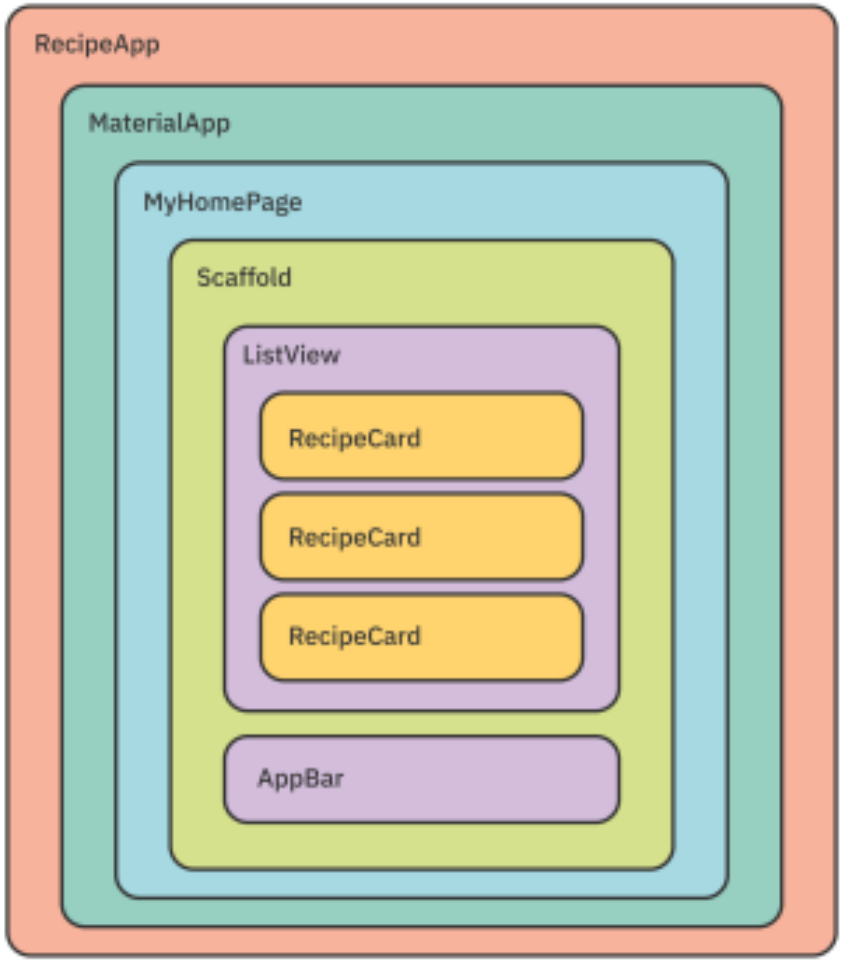
1. You return a Card from **buildRecipeCard**().
2. The Card's child property is a Column. A Column is a widget that defines a vertical layout.
3. The Column has two children.
4. The first child is an Image widget. AssetImage states that the image is fetched from the local **asset** bundle defined in **pubspec.yaml**.
5. A Text widget is the second child. It will contain the **recipe.label** value:

Putting the list into a card:

To use the card, go to **_MyHomePageState** and replace // TODO: Update to return Recipe card and the return line below it with this:

```
// TODO: Add GestureDetector
return buildRecipeCard(Recipe.samples[index]);
```

Looking at the widget tree:

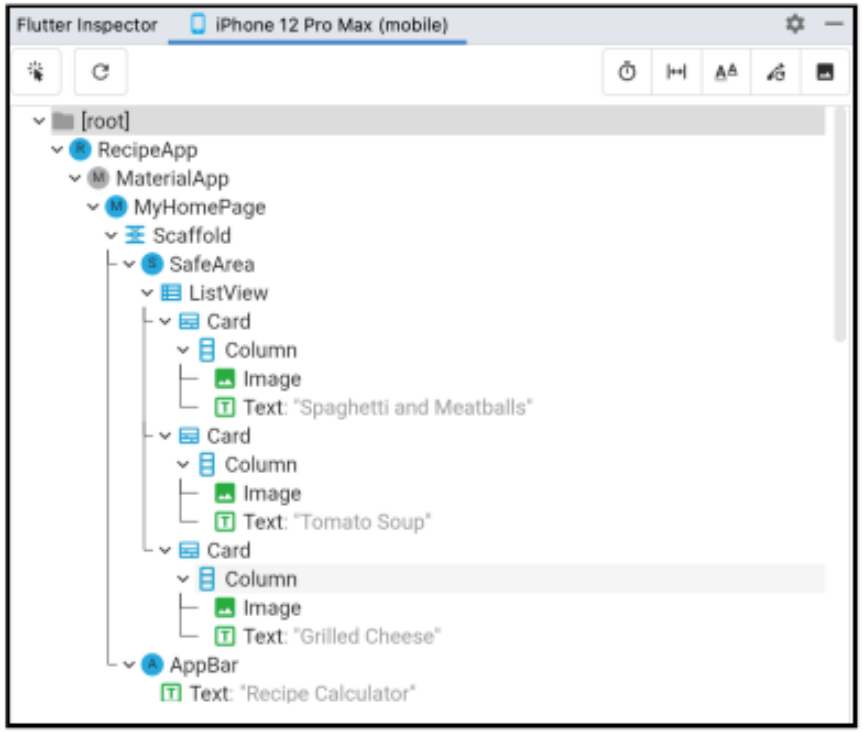


Looking at the widget tree:

- **RecipeApp** built a **MaterialApp**, which in turn used **MyHomePage** as its home.
- That builds a **Scaffold** with an **AppBar** and a **ListView**.
- You then updated the **ListView** builder to make a **Card** for each item.

Looking at the 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:

- 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 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..

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]),
);
```

Making a tap response:

- 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.

Creating an actual target page:

- 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:

- 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
```

- 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

Creating an actual target page:

- 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:

- **Scaffold** defines the page's general structure.
- In the body, there's a **SafeArea**, a Column with a Container, a **SizedBox** and Text children.
- **SafeArea** keeps the app from getting too close to the operating system interfaces, such as the notch or the interactive area of most iPhones.
- 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.
- There is a spacer **SizedBox**.
- 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.

Creating an actual target page:

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

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

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

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

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:

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

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

This is a simple data container for an ingredient. It has a name, a unit of measure — like "cup" or "tablespoon" — and a quantity.

Adding ingredients:

At the top of the `Recipe` class, replace `// TODO: Add servings and ingredients` here with the following:

```
int servings;
List<Ingredient> 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',
  ),
```

Adding ingredients:

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(
    'Grilled Cheese',
    'assets/3187380632_5056654a19_b.jpg',
    1,
    [
      Ingredient(2, 'slices', 'Cheese',),
      Ingredient(2, 'slices', 'Bread',),
    ],
  ),
```

Adding ingredients:

```
    Recipe(
      'Chocolate Chip Cookies',
      'assets/15992102771_b92f4cc00a_b.jpg',
      24,
      [
        Ingredient(4, 'cups', 'flour',),
        Ingredient(2, 'cups', 'sugar',),
        Ingredient(0.5, 'cups', 'chocolate chips',),
      ],
    ),
    Recipe(
      'Taco Salad',
      'assets/8533381643_a31a99e8a6_c.jpg',
      1,
      [
        Ingredient(4, 'oz', 'nachos',),
        Ingredient(3, 'oz', 'taco meat',),
        Ingredient(0.5, 'cup', 'cheese',),
        Ingredient(0.25, 'cup', 'chopped tomatoes',),
      ],
    ),
    Recipe(
      'Hawaiian Pizza',
      'assets/15452035777_294cefced5_c.jpg',
      4,
      [
        Ingredient(1, 'item', 'pizza',),
        Ingredient(1, 'cup', 'pineapple',),
        Ingredient(8, 'oz', 'ham',),
      ],
    ),
  ];
```

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

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}');
    },
  ),
),
```

Showing the ingredients:

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.

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;
```

Adding a serving slider:

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,
),
```

`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.

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}');
```

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

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}');
```

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