2
Miles or Kilometers? Using Stateful Widgets

The world is a strange place. Most of us are aware that when you travel to other countries, you may find different languages, culture, and food, but you would expect that at least numbers and measures would stay the same wherever you go, right? Well, this is not so.

Measures such as distance, speed, weight, volume, and temperature change based on where you live. Actually, there are two main measurement systems in use today: the imperial system, which is used mainly in the United States; and the metric system, which is used in most of the other countries.

In this chapter, you'll bring some order to this confusing world: you will build a measures conversion app, in which distance and weight measures will be converted from imperial to metric, and vice versa.

We'll cover the following aspects in this chapter:

- Project overview
- Understanding state and stateful widgets
- Creating the measure converter project

Technical requirements

Should you get lost in the construction of the app, you'll find the completed app code at the end of this chapter, or on the book's GitHub repository at `https://github.com/PacktPublishing/Google-Flutter-Projects`.

To follow along the code examples in this book, you should have the following software installed on your Windows, Mac, Linux, or Chrome OS device:

- The Flutter SDK.
- When developing for Android, you'll need: the Android SDK – easily installed by Android Studio.
- When developing for iOS, you'll need: macOS and Xcode.
- An emulator (Android), a simulator (iOS), or a connected iOS or Android device enabled for debugging.
- An editor: Visual Studio Code, Android Studio, or IntelliJ IDEA are recommended. All should have the Flutter/Dart extensions installed.

You'll find an installation guide in the *Appendix* of this book.

The necessary time to build the app in this chapter should be approximately 2.5 hours.

Project overview

The measures conversion app will allow your users to select a measure – metric or imperial – and convert it to another measure. For example, they'll be able to convert a distance in miles to a distance in kilometers, or a weight in kilograms to a weight in pounds. So, next time you travel to a country with a different system, you'll be able to easily understand the speed of your car (and maybe avoid a fine), or the weight of the food you can buy at the market, and along the way, you'll build on your Flutter skills.

By the end of this chapter, you'll know how to leverage `State` using widgets such as `TextFields` to interact with users and make your apps interactive.

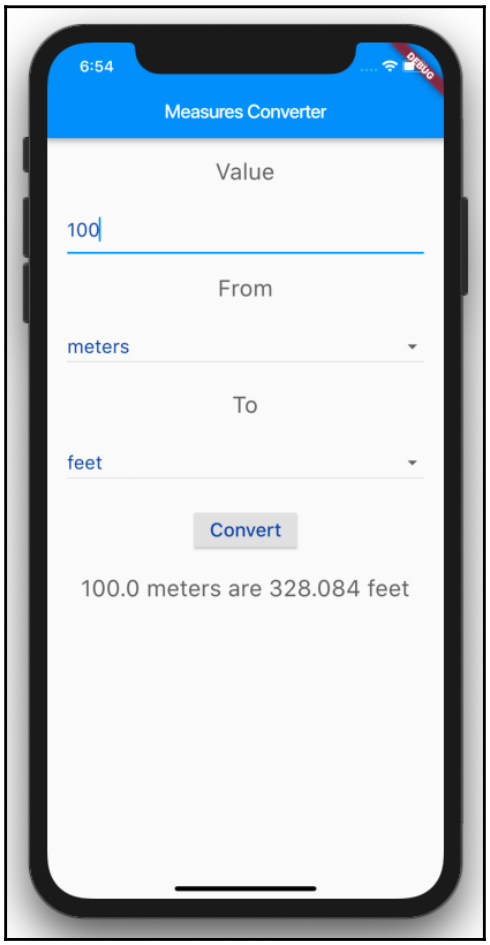
While doing so, you'll encounter several fundamental concepts in Flutter, and in particular, the following:

- You'll see what State is in Flutter, start using s*tateful widgets,* and understand when you should use *stateless* or *stateful* widgets.
- You'll see how and when to update the *State* in your app.
- You'll also see how to handle events, such as `onChanged` and `OnSubmitted` in a `TextField`.

- You'll see how to use the most common user input widget—`TextField`.
- Another very important widget that you'll use for this project is `DropDownButton`. It's a drop-down list where you decide the choices that your users have. And those choices are called `DropDownItems` in Flutter.
- You'll see how to start separating the logic of your app from the **User Interface** (**UI**), and you'll gain a few tips on how to build the structure of your app.

While stateful widgets are the most basic way to deal with State in an app, there are other, more efficient ways to deal with State in Flutter. Some of those will be shown in the upcoming projects.

The following is the final layout of the project that you'll build in this chapter:



As you can see, this is a rather standard form with Material Design widgets, which should be very easy to compile for your users. You can use it as a starting point for any form that you use in your future apps.

Understanding state and stateful widgets

The widgets that we've seen so far are stateless widgets, meaning that once created they are immutable, and they do not keep any state information. When you interact with your users, you expect things to change. For example, if you want to convert a measure from one system to another, the result must change, based on some user input.

The most basic way to deal with changes in Flutter is using State.

State is information that can be used when a widget is built and can change during the lifetime of a widget.

An important part of this definition is that state is **information that can change**, and the most obvious takeaway of this concept is that when you want to add interactivity to your app, you can use State. But, if you read this definition thoroughly, it also means that **it's not the widget itself that will change, it's the State of a widget that will change**, and when it does, the widget will be rebuilt. When a widget has a State, it's called a stateful widget. And in Flutter, stateful widgets are immutable. It's only the State itself that changes.

Each time the State changes, the widget gets rebuilt.

Let's have a look at the main differences between a stateless widget, which we've used so far, and a stateful widget. Of course, the most obvious difference is explained by the name itself, the State: State/*less* and State/*ful*.

But there is a different implementation as well. In order to see it in detail, we'll create a new app and see it in practice.

Creating the measure converter project

We will now create a new app that we'll use throughout this chapter to build a fully functioning measure converter:

1. From your favorite editor, create a new app. Name the new app `Unit Converter`.
2. In the `main.dart` file, remove the example code and write the code given as follows:

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

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
@override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Measures Converter',
      home: Scaffold(
      appBar: AppBar(
          title: Text('Measures Converter'),
      ),
      body: Center(
          child: Text('Measures Converter'),
      ),),
  );}
}
```

As you may have noticed, the preceding code makes use of a Stateless widget*:*

```
class MyApp extends StatelessWidget {
```

A Stateless widget is a **class** that extends a `StatelessWidget`. Extending a `StatelessWidget` class requires overriding a `build()` method.

In the `build()` method, you describe the widget returned by the method:

```
@override
Widget build(BuildContext context) {
```

The `build()` method that takes a *context* and returns a *widget*:

```
return MaterialApp(...)
```

So to summarize, in order to have a stateless widget you need to do the following:

1. Create a class that extends `StatelessWidget`.
2. Override the `build()` method.
3. Return a widget.

Once built, a Stateless widget never changes.

Using stateful widgets

Let's now transform the `MyApp` class into a stateful widget, so that you can see the different implementations of the class:

```
class MyApp extends StatefulWidget {
```

You can see immediately that you get two errors. If you hover over the `MyApp` class, the error that you see is "**Missing concrete implementation of StatefulWidget.createState**," and if you hover over the `build` method you see "**The method doesn't override an inherited method**."

What these errors are trying to tell us is the following:

1. A stateful widget requires a `createState()` method.
2. In a stateful widget, there is no `build()` method to override.

Let's fix both these issues using the following steps:

1. Add the necessary `createState()` method, which will return `MyAppState`, which we'll create shortly. In the `MyApp` class, just under its definition, write the following code:

```
@override
MyAppState createState() => MyAppState();
```

2. Create a new class called `MyAppState`, that extends the State, and in particular, the State of `MyApp`:

```
class MyAppState extends State<MyApp> {}
```

3. In order to solve the second error ("**Missing concrete implementation of State.build"**), cut the `build()` method that is now in the `MyApp` class, and paste it into the `MyAppState` class. The revised code should look like this:

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

void main() => runApp(MyApp());

class MyApp extends StatefulWidget {
  @override
  MyAppState createState() => MyAppState();
}

class MyAppState extends State<MyApp> {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Measures Converter',
      home: Scaffold(
        appBar: AppBar(
          title: Text('Measures Converter'),
        ),
        body: Center(
          child: Text('Measures Converter'),
        ),
      ),
    );
  }
}
```

To sum it up, from a *syntax* perspective, the difference between a Stateless widget and a stateful widget is that the former overrides a `build()` method and returns a widget, whereas a stateful widget overrides a `createState()` method, which returns a State. The `State` class overrides a `build()` method, returning a widget.

From a functional point of view, in the code that we have written, there is no difference whatsoever between the two, as in both cases the app looks and behaves exactly in the same way. So, let's add a feature that **requires** a stateful widget, and could not be achieved with a Stateless Widget.

Here, you can see the app layout so far:

Next, let's see how to read the user input from `TextField`.

Reading user input from TextField

In the `State` class, let's add a member called `_numberFrom`. As shown in the following code, this is a value that will change based on user input:

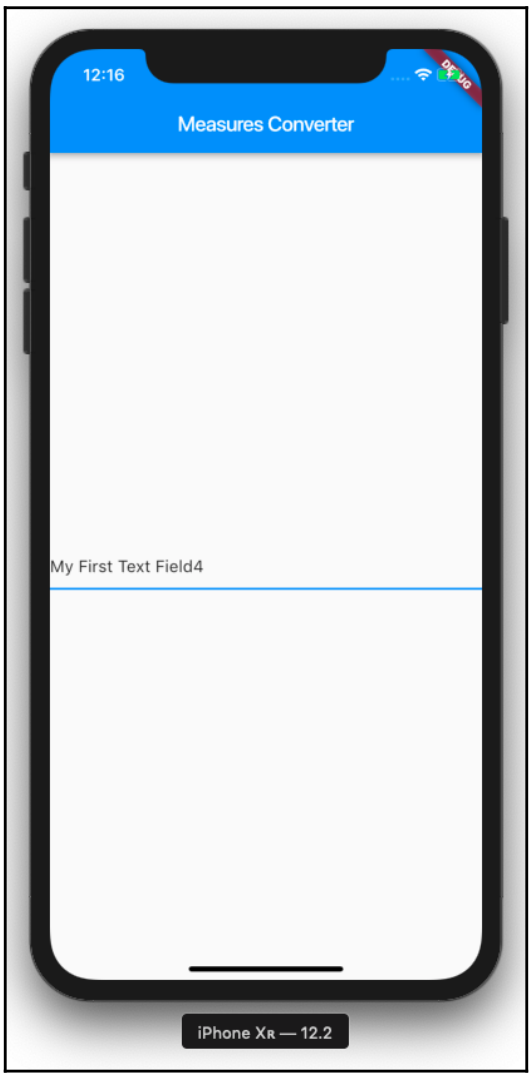
```
double _numberFrom;
```

Then, in the body of the `build()` method, let's delete the text widget, and add `TextField` instead:

```
body: Center(
    child: TextField(),
),
```

You generally use `TextField` when you want to take some input from your users.

As you can see, there's now `TextField` in the center of your app, and you can write into it by clicking over the line and typing something:



Right now, `TextField` does nothing, so the first thing we need to do is *read* the value that the user inputs into it.

While there are different ways to read from `TextField`, for this project, we'll respond to each change in the content of `TextField` through the `onChanged` method, and then we'll update the State.

In order to update the State, you need to call the `setState()` method.

The `setState()` method tells the framework that the state of an object has changed, and that the UI needs to be updated.

Inside the `setState()` method, you change the class members that you need to update (in our case, `_numberFrom`):

```
child: TextField(
   onChanged: (text) {
     var rv = double.tryParse(text);
     if (rv != null) {
       setState(() {
          _numberFrom = rv;
       });
     }
  },
),
```

In the preceding code, each time the value of `TextField` changes (`onChanged`), we check whether the value that was typed is a number (`tryParse`). If it's a number, we change the value of the `_numberForm` member: in this way, we have actually updated the State. In other words, when you call the `setState()` method to update a class member, you are also updating the State of the class.

We are not giving any feedback to the user, so unless we use the debugging tools of our editor, we cannot actually check whether this update actually happened. In order to solve that, let's add a `Text` widget that will show the content of the `TextEdit` widget, and then wrap the two widgets into a `Column` widget:

```
body: Center(
  child: Column(
    children: [
      TextField(
        onChanged: (text) {
          var rv = double.tryParse(text);
          if (rv != null) {
            setState(() {
              _numberFrom = rv;
```

```
                    });
                  }
                },
              ),Text((_numberFrom == null) ? '' : _numberFrom.toString())
            ],
          ),
        ),
```

Before trying the app, let's add another method to the `MyAppState` class:

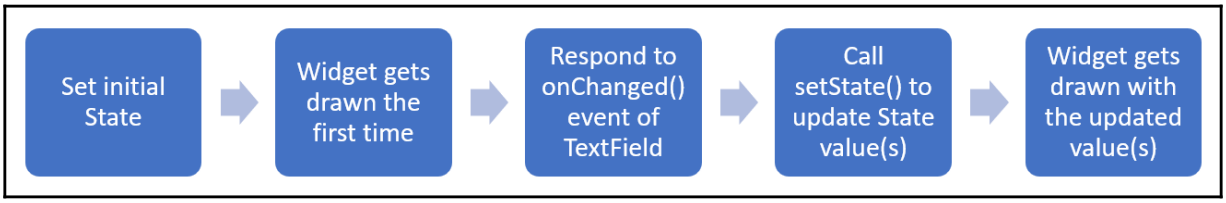
```
 @override
void initState() {
_numberFrom = 0;
super.initState();
}
```

The `initState()` method is called once for each `State` object when the State is built. This is where you generally put the initial values that you might need when you build your classes. In this case, we are setting the `_numberFrom` initial value. Also note, that you should always call `super.initState()` at the end of the `initState()` method.

Now, if you write a number in the `TextField`, you'll see the same number in the `Text` widget, as well. In this apparently simple example, many things are happening at once:

- You are setting an initial State of the app through the `_numberForm` class member in the `InitState()` method.
- The widget is drawn on screen.
- You are responding to a `TextField` event: the `onChanged` event, which is called every time the content of the `TextField` changes.
- You are changing the State by calling the `setState()` method, and there you change the value of `_numberForm`.
- The widget is redrawn with the new State, which contains the number that you write in `TextField`, so the `Text` widget, which reads `_numberForm`, contains the modified value of the State.

Here is a diagram that highlights the steps described previously: with a few variations, you'll notice a similar pattern whenever you use stateful widgets in your apps:



To sum it up, calling `setState()`, does the following:

- Notifies the framework that the internal state of this object has changed
- Calls the `build()` method and redraws its children with the updated `State` object

Now you have the ability to create an app that responds to the user input and changes the UI based on a changing State, which in Flutter is the most basic way to create interactive apps.

Next, we need to complete the UI of our app, and in order to do that, we need another widget: `DropDownButton`. Let's create this in the following section.

Creating a DropdownButton widget

`DropdownButton` is a widget that lets users select a value from a list of items. `DropdownButton` shows the currently selected item, as well as a small triangle that opens a list for selecting another item.

Here are the steps that are required to add `DropdownButton` to your apps:

1. Create an instance of `DropdownButton`, specifying the type of data that will be included in the list.
2. Add an `items` property that will contain the list of items that will be shown to the user.

3. The `items` property requires a list of `DropdownMenuItem` widgets. Therefore, you need to map each value that you want to show into `DropdownMenuItem`.
4. Respond to the user actions by specifying an event; typically, for `DropdownButton`, you will call a function in the `onChanged` property.

As an example, the following code creates a `DropdownButton` widget that shows a list of fruits (that are good for your health):

```
var fruits = ['Orange', 'Apple', 'Strawberry', 'Banana'];

DropdownButton<String>(
   items: fruits.map((String value) {
      return DropdownMenuItem<String>(
         value: value,
         child: Text(value),);
      }).toList(),
      onChanged: (String newValue) {}
),
```

`DropDownButton` is a generic, as it's built as `DropDownButton<T>`, where the generic type, `T`, is the type of item in your `DropDownButton` widget (in this case, *T* is a string).

Dart supports **generics** or **generic types**. For example, a list can contain several types: `List<int>` is a list of integers, `List<String>` is a list of strings, and `List<dynamic>` is a list of objects of any type. Using generics helps to ensure type safety: in the example of the list, for instance, you cannot add a number to `List<String>`.

The `map()` method iterates through all the values of the array, and performs a function on each value of the list. The function inside the `map()` method returns a `DropDownMenuItem` widget, which, in the previous example, has a `value` property and a `child` property. The child is what the user will see, in this case, a `Text` widget. The `value` is what you'll use to retrieve the selected item on the list.

The `map()` method returns an iterable, which is a collection of values that can be accessed sequentially.

Over that, you call the `toList()` method, which creates a list that contains the elements that should be returned. This is required by the items property.

In our app, we need two `DropdownButton` widgets, one for the starting unit, and one for the converted unit:

1. Let's create a *list* of *strings* that will contain all the measures that we want to deal with. At the beginning of the `State` class, let's add the following code:

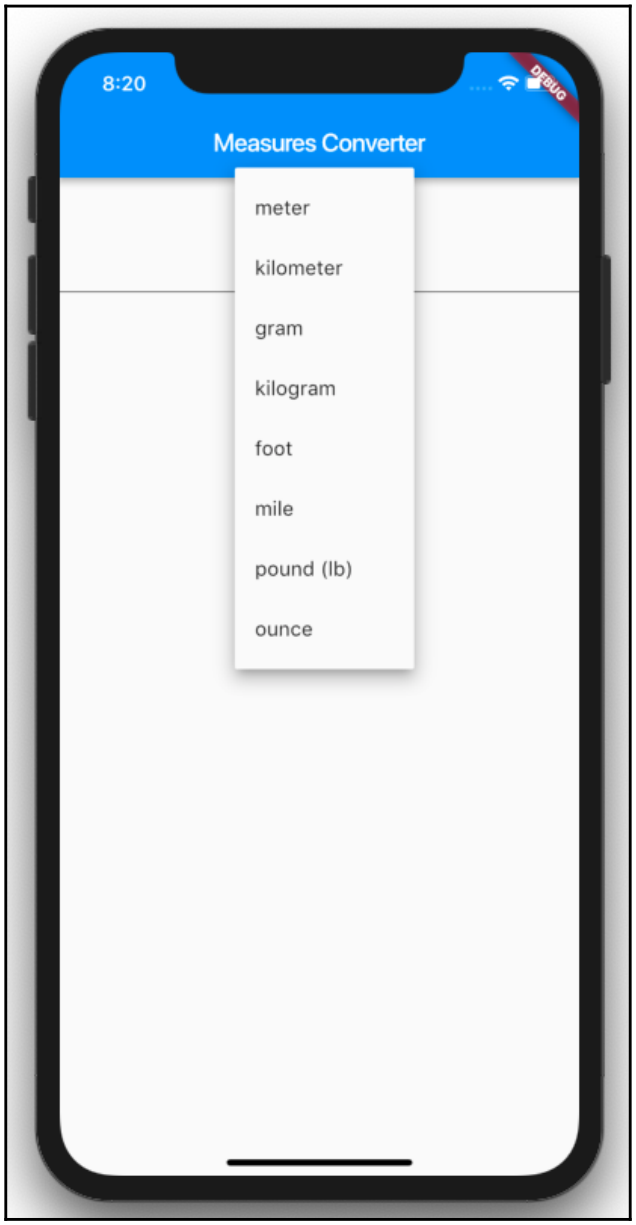
```
final List<String> _measures = [
    'meters',
    'kilometers',
    'grams',
    'kilograms',
    'feet',
    'miles',
    'pounds (lbs)',
    'ounces',
];
```

2. Next, we'll create a `DropDownButton` widget, which will read the values of the list, and place it at the top of the *column*, above the `TextField`*:*

```
DropdownButton(
 items: _measures.map((String value) {
 return DropdownMenuItem<String>(value: value, child:
Text(value),);
 }).toList(),
 onChanged: (_) {},
),
```

If you try out the app now, you'll see that at the top of the screen there's a small triangle. When you click on it, the list of measures is shown, and you can click on any of them to select one. At this time, when you select a value, `DropdownButton` still remains empty. This is because we need to implement the function inside the `onChanged` member of `DropDownButton`.

The following screenshot shows how `DropdownButton` contains a list of items:



In the next section, we will learn how to respond to the user input when they change the value in `DropDownButton`.

Updating a DropdownButton widget

Let's modify the `onChanged` property using the following steps:

1. Create a new string called `_startMeasure` at the top of the `MyAppState` class. It will contain the selected value from `DropdownButton`:

```
String _startMeasure;
```

2. Instead of the underscore, call the parameter that is passed to the function, `value`.
3. Inside the function, call the `setState()` method to update `_startMeasure` with the new value that's passed. Here is the resulting code:

```
onChanged: (value) {
 setState(() {
      _startMeasure = value;
   });
}
```

4. The last step of this task is reading the selected value so that `DropdownButton` reads it when the app starts and every time it changes. Let's add the following line to `DropDownButton`:

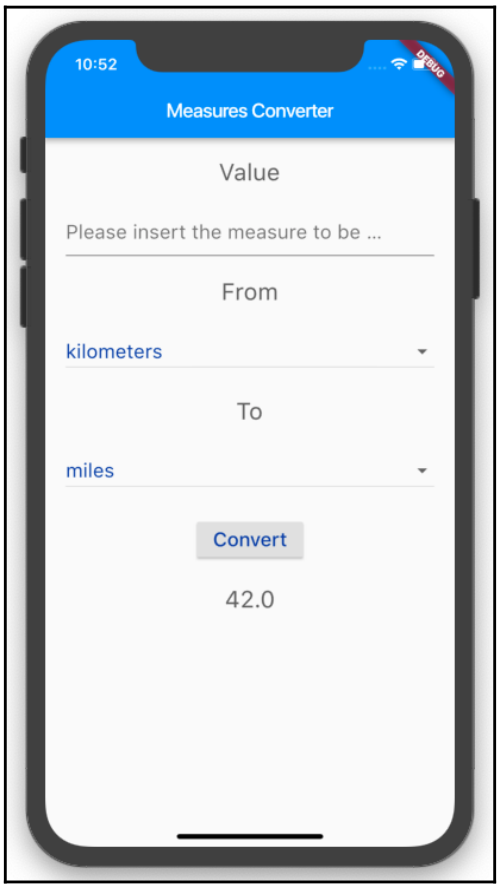
```
value: _startMeasure,
```

Now, if you try the app, when you select a value from the list, the value shows up in `DropdownButton`, which is exactly the behavior that you would expect from it.

In the next section, we'll complete the UI for this screen.

Completing the UI of the app

Let's now complete the UI of our app. The final result is shown in the following screenshot:



We actually need to show eight widgets on screen:

- `Text` containing **Value**
- `TextField` for the start value
- `Text` containing **From**
- A `DropdownButton` widget for the start measure
- Another `Text` containing **To**
- A `DropdownButton` widget for the measure of the conversion
- `RaisedButton` to call the method that will convert the value.
- `Text` for the result of the conversion

Each element of the `Column` should also be spaced and styled.

Let's begin by creating two `TextStyle` widgets. The advantage of this approach is that we can use them several times without needing to specify the styling details for each widget:

1. At the top of the `build()` method, let's first create a `TextStyle` widget, which we'll use for `TextFields`, `DropDownButtons`, and `Button`. We'll call it `inputStyle`:

```
final TextStyle inputStyle = TextStyle(
    fontSize: 20,
    color: Colors.blue[900],
);
```

2. Then, let's create a second `TextStyle` widget, which we'll use for the `Text` widgets in the column. We'll call it `labelStyle`:

```
final TextStyle labelStyle = TextStyle(
    fontSize: 24,
    color: Colors.grey[700],
);
```

3. We also want `Column` to take some distance from the horizontal device borders. So, instead of returning a `Center` widget, we can return `Container`, which takes a `padding` of 20 logical pixels. `EdgeInsets.symmetric` allows you to specify a value for the horizontal or vertical padding:

```
body: Container(
   padding: EdgeInsets.symmetric(horizontal: 20),
   child: Column(
```

4. And speaking of spacing, we want to create some space between the widgets in the column. A simple way of achieving this is using the **Spacer** widget: `Spacer` creates an empty space that can be used to set spacing between widgets in a flexible container, such as the `Column` in our interface. A `Spacer` widget has a `flex` property, whose default value is `1`, which determines how much space we want to use. For instance, if you have two `Spacer` widgets, one with a `flex` property of `1`, and another with a `flex` property of `2`, the second will take double the space of the first. At the top of the `Column` let's add an initial `Spacer` widget:

```
child: Column(
   children: [
      Spacer(),
```

5. Under the `Spacer` widget, add the first text in the `Column` containing the `'Value'` string. We'll also apply `labelStyle` to this widget, and under `Text` we will place another `Spacer`:

```
Text(
   'Value',
   style: labelStyle,
),
Spacer(),
```

6. Under the `Text` that contains `'Value'` and its `Spacer`, we need to place the `TextField` that we previously created, to allow the user to input the number that they want to convert. Let's edit `TextField` so that it takes the `inputStyle TextStyle`. We'll also set the `decoration` property of the `TextField`.

The `decoration` property of a `TextField` takes an `InputDecoration` object. `InputDecoration` allows you to specify the border, labels, icons, and styles that will be used to decorate a text field.

7. `hintText` is a piece of text that is shown when `TextField` is empty, to suggest which kind of input is expected from the user. In this case, add `"Please insert the measure to be converted"` as a `hintText` prompt for our `TextField`:

```
TextField(
   style: inputStyle,
   decoration: InputDecoration(
      hintText: "Please insert the measure to be converted",
   ),
   onChanged: (text) {
      var rv = double.tryParse(text);
      if (rv != null) {
         setState(() {
            _numberFrom = rv;
         });
     }
  },
),
```

8. Under `TextField`, place another `Spacer()`, then a `Text` with `'From'` and the `labelStyle` style:

```
Spacer(),
   Text(
      'From',
      style: labelStyle,
   ),
```

9. Under the `'From' Text`, place the `DropDownButton` widget, whose value is `_startMeasure`, which you wrote in the previous section:

```
DropdownButton(
   isExpanded: true,
   items: _measures.map((String value) {
      return DropdownMenuItem<String>(
         value: value,
         child: Text(value),
      );
   }).toList(),
   onChanged: (value) {
      setState(() {
         _startMeasure = value;
      });
   },
   value: _startMeasure,
),
```

10. Next, add another `Text` for the second dropdown: in this case, the `Text` will contain `'To'`, and the style will be `labelStyle`, as before:

```
Spacer(),
Text(
   'To',
   style: labelStyle,
),
```

11. Under the `'To' Text` we need to place the second `DropdownButton` widget, and this requires another class member: the first `DropdownButton` widget used `_startMeasure` for its value; this new one will use `_convertedMeasure`. **At the top of the `MyAppState` class,** add the following declaration:

```
 String  _convertedMeasure;
```

12. Now, we are ready to add the second `DropDownButton` widget: this will contain the same measures list as the previous one. The only difference here is that it will reference the `_convertedMeasure` variable. As usual, don't forget to add a `Spacer()` before the widget:

```
Spacer(),
DropdownButton(
   isExpanded: true,
   style: inputStyle,
   items: _measures.map((String value) {
      return DropdownMenuItem<String>(
         value: value,
         child: Text(
            value,
            style: inputStyle,
         ),
       );
      }).toList(),
   onChanged: (value) {
      setState(() {
         _convertedMeasure = value;
      });
   },
   value: _convertedMeasure,
),
```

13. Next, add the button that will apply the conversion: it will be a `RaisedButton` with a `Text` of `'Convert'`, and the style of `inputStyle`. At this time, the `onPressed` event will do nothing, as we don't have the logic of the app ready yet. Before and after the button we'll place a `Spacer`, but this time, we will also set its `flex` property to `2`. This way, the space between the button and the other widgets on screen will be twice the amount of the other spacers:

```
Spacer(flex: 2,),
   RaisedButton(
      child: Text('Convert', style: inputStyle),
      onPressed: () => true,
   ),
   Spacer(flex: 2,),
```

14. Finally, we'll add the `Text` for the result of the conversion. For now, let's just leave the `_numberFrom` value as `Text`; we'll change that in the next section. At the end of the result, we'll add the largest `Spacer` of this screen, with a `flex` value of `8`, in order to leave some space at the end of the screen:

```
Text((_numberFrom == null) ? '' : _numberFrom.toString(),
   style: labelStyle),
Spacer(flex: 8,),
```

15. There's one very last step that we need to perform before we complete the UI. On some devices, the UI that we have designed may be bigger than the available screen when the keyboard appears on screen. This may cause an error in your app. In order to solve this issue, there's a simple solution, which I recommend that you always use when designing your layouts with Flutter. You should put the `Column` widget into a scrollable widget, in this case, `SingleChileScrollView`. What this will do is make the widgets on the screen scroll if they take more space than is available on screen. So just enclose `Column` into a `SingleChildScrollView` widget like in the following example:

```
body: Container(
    padding: EdgeInsets.symmetric(horizontal: 20),
    child: SingleChildScrollView(
        child: Column(
            ...
        ),
    ),
```

If you try the app now, you should see the final look of the app, but other than for choosing values from the `DropdownButton` widgets, and adding some text to `TextField`, the screen doesn't do anything useful. Let's add the logic of the app next.

Adding the business logic

You have completed the layout of the app, but right now the app is missing the part that converts the values that are based on the user input.

Generally speaking, it's always a good idea to separate the logic of your apps from the UI, and there are great patterns in Flutter that help you achieve this result. You'll use some of those, such as `ScopedModel` and **Business Logic Components** (**BLoCs**), in the following projects, but for now, we can just add the conversion functions into our class.

There are certainly several ways to write the code to perform the conversion between measures for this app. The approach that I find easiest is seeing the formulas that we need to apply as a two-dimensional array of values, also called a *matrix*. This matrix contains all the possible combinations of choices that the user can perform.

A diagram of this approach is shown here:

MEASURES	0 - Meters	1 - Kilometers	2 - Grams	3 - Kilograms	4 - Feet	5 - Miles	6 - Pounds	7 - Ounces
0 – Meters	1	0.0001	0	0	3.28084	0.00062	0	0
1 – Kilometers	1000	1	0	0	3280.84	0.62137	0	0
2 – Grams	0	0	1	0.0001	0	0	0.0022	0.03527
3 – Kilograms	0	0	1000	1	0	0	2.20462	35.274
4 – Feet	0.3048	0.0003	0	0	1	0.00019	0	0
5 – Miles	1609.34	1.60934	0	0	5280	1	0	0
6 – Pounds	0	0	453.592	0.45359	0	0	1	16
7 – Ounces	0	0	28.3495	0.02835	0	0	0.0625	1

So, for example, when you want to convert 100 kilometers into miles, you multiply 100 by the number that you find in the array (in this case, **0.621371**). It's a bit like playing Battleships. When the conversion is not possible, the multiplier is 0, so any impossible conversion returns 0.

As you might recall from `Chapter 1`, *Hello Flutter!*, in Dart we use `List` in order to create arrays. In this case, it's a two-dimensional array or matrix, and therefore we'll create an object that contains `List`'s. Let's look at the steps:

1. We'll need to convert the `Strings` of the measure units into numbers. At the top of the `MyAppState` class, add the following code, using `Map`:

```
final Map<String, int> _measuresMap = {
 'meters' : 0,
 'kilometers' : 1,
 'grams' : 2,
 'kilograms' : 3,
 'feet' : 4,
 'miles' : 5,
```

```
 'pounds (lbs)' : 6,
 'ounces' : 7,
};
```

2. Maps allow you to insert key–value pairs, where the first element is the key, and the second is the value. When you need to retrieve a value from `Map`, you can use the following syntax:

```
myValue = measures['miles'];
```

   The `myValue` variable will have a value of 5.

3. Next, we'll create a list that contains all of the multipliers that were shown in the previous diagram:

```
final dynamic _formulas = {
'0':[1,0.001,0,0,3.28084,0.000621371,0,0],
'1':[1000,1,0,0,3280.84,0.621371,0,0],
'2':[0,0,1,0.0001,0,0,0.00220462,0.035274],
'3':[0,0,1000,1,0,0,2.20462,35.274],
'4':[0.3048,0.0003048,0,0,1,0.000189394,0,0],
'5':[1609.34, 1.60934,0,0,5280,1,0,0],
'6':[0,0,453.592,0.453592,0,0,1,16],
'7':[0,0,28.3495,0.0283495,3.28084,0,0.0625, 1],
};
```

   If you don't want to type this code, I've created a `Gist` file that contains the `Conversion` class. You'll find the full file at `https://gist.github.com/simoales/66af9a23235abcb537621e5bf9540bc6`.

4. Now that we have created a matrix that contains all of the possible combinations of conversion formulas, we only need to write the method that will convert the values using the formulas and the measures `Map`. Add the following code at the bottom of the `MyAppState` class:

```
void convert(double value, String from, String to) {
 int nFrom = _measuresMap[from];
 int nTo = _measuresMap[to];
 var multiplier = _formulas[nFrom.toString()][nTo];
 var result = value * multiplier;
}
```

The `convert()` method takes three parameters:

- The number that will be converted (*double value*)
- The unit of measure in which this value is currently expressed, as a `String` (*String from*)
- The unit of measure unit in which the value will be converted, also a `String` (*String to*)

For example, if you want to convert 10 meters into feet, 10 is the number, meters is the unit in which the value is currently expressed, and feet is the unit to which the number will be converted.

Let's see in detail how the `convert()` method has worked so far:

1. Inside the `convert()` method, you find the number associated with the `from` the measure:

```
int nFrom = measures[from];
```

2. Then, you do the same with the `to` measure:

```
int nTo = measures[to];
```

3. Next, you create a `multiplier` value that takes the correct conversion formula from the `formulas` matrix:

```
var multiplier = formulas[nFrom.toString()][nTo];
```

4. Finally, you calculate the result of the conversion:

```
double result = value * multiplier;
```

In this case, if the conversion is not possible, for example, when the user tries to convert a weight measure into a distance measure, this function does not raise any error.

Next, we need to show the result of the conversion to the user:

1. Declare a `String` variable at the top of the `MyAppState` class:

```
String _resultMessage;
```

2. In the `convert()` method, after calculating the result, populate the `_resultMessage String`, and call the `setState()` method to notify the framework that an update to the UI is needed:

```
if (result == 0) {
    _resultMessage = 'This conversion cannot be performed';
  }
  else {
    _resultMessage = '${_numberFrom.toString()} $_startMeasure are
${result.toString()} $_convertedMeasure';
  }
  setState(() {
    _resultMessage = _resultMessage;
  });
```

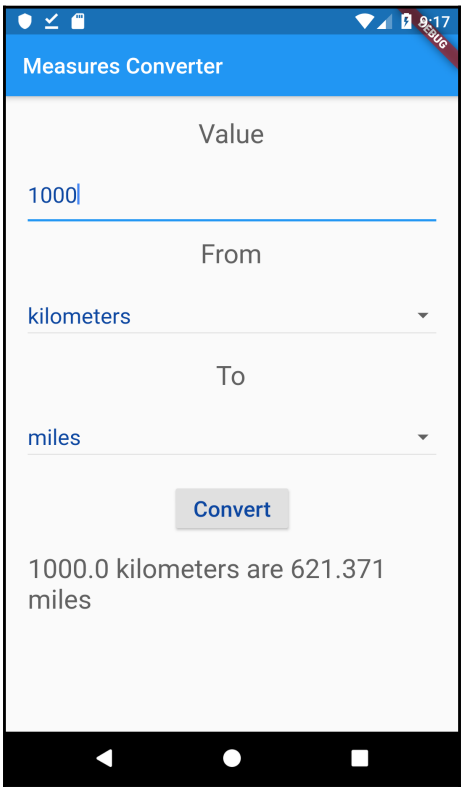
3. Finally, we need to call the `convert()` method when the user taps on the **Convert** button. Before calling the method, we'll check that every value has been set to prevent potential errors. Edit `RaisedButton`, as shown here:

```
RaisedButton(
   child: Text('Convert', style: inputStyle),
   onPressed: () {
      if (_startMeasure.isEmpty || _convertedMeasure.isEmpty ||
_numberFrom==0) {
         return;
      }
      else {
         convert(_numberFrom, _startMeasure, _convertedMeasure);
      }
  },
),
```

4. In order to show the result, let's also update the `Text` widget, so that it shows the string that contains the message to the user:

```
Text((_resultMessage == null) ? '' : _resultMessage,
   style: labelStyle),
```

Congratulations, the app is now complete! If you try it out now, you should see a screen like the one shown here:



As you can see in the preceding screenshot, when we select two compatible measures, you should get the correct result on the screen.

Summary

In the project that you've built in this chapter, you've seen how to create interactive apps using State.

You've created a Stateless widget and transformed it into a stateful widget. In doing so, you've seen the different implementations between the two, and you've learned that in Flutter, widgets are immutable. It's the State that changes.

You have used two very important widgets, which help you to interact with the users: `TextField` and `DropdownButton`.

For `TextField`, you've used one of the possible ways to respond to the user input, which is using the `onChanged()` event, and from there, you called the `setState()` method, which updates the inner `State` of a widget.

You've seen how to add a `DropdownButton` widget to your apps, and also how to set the `items` property that will contain a list of `DropdownMenuItem` widgets to show to the user, and again, how to use the `onChanged` property to respond to the user input.

In other projects in this book, you'll see other, more efficient ways to deal with State in Flutter. In the next chapter, in particular, you'll see how to leverage streams of data in your apps in order to build a timer app.

Questions

At the end of each project, you'll find a few questions to help you remember and review the contents that have been covered in the chapter, and this chapter is no exception. Please try to answer the following questions, and when in doubt, have a look at the content in the chapter itself: you'll find all the answers there!

1. When should you use stateful widgets in your apps?
2. Which method updates the `State` of your class?
3. Which widget would you use to allow your user to select an option from a dropdown list?
4. Which widget would you use to allow your user to type some text?
5. Which event can you use when you want to react to some user input?
6. What happens when your widgets take more space than what's available on the screen? How do you solve this issue?
7. How can you get the width of the screen?
8. What is `Map` in Flutter?
9. How can you style your text?
10. How can you separate the logic of your apps from the UI?

Further reading

As Flutter is rapidly gaining momentum, you'll find a lot of articles and documents on the topics that we've touched in this project.

For padding, `EdgeInsets`, the box model, and layouts in general, the Flutter official documentation has a fantastic article to get you started at: `https://flutter.dev/docs/development/ui/layout`.

For `TextFields` have a look at: `https://flutter.dev/docs/cookbook/forms/text-input`.

For use cases of `DropdownButton` widgets, again the official documentation has a nice page at: `https://docs.flutter.io/flutter/material/DropdownButton-class.html`.