Adding Interactivity and Navigation to Your App

Frontend application design is often divided into two categories – **user interface** (**UI**) and **user experience** (**UX**). The user interface is made up of all the elements on the screen – images, colors, panels, text, and so on. The user experience is what happens when your users **interact** with your interfaces. It governs interactivity, navigation, and animations. If the UI is the "*what*" of app design, then the UX is the "*how*."

So far, we have covered some of the user interface components in Flutter. Now, it's time to make our widgets useful and start building interactivity. We're going to cover some of the primary widgets that are used to deal with user interactions – in particular buttons, TextFields, ScrollViews, and dialogs. You will also use the Navigator to create apps with multiple screens.

Throughout this lesson, you are going to build a single app called *Stopwatch*. This will be a fully functioning stopwatch that will keep track of laps and show a full history of every completed lap.

In this lesson, we're going to cover the following recipes:

- Adding state to your app
- Interacting with buttons
- Making it scroll
- Handling large datasets with list builders
- Working with TextFields
- Navigating to the next screen
- Invoking navigation routes by name
- Showing dialogs on the screen
- Presenting bottom sheets

Adding state to your app

So far, we've only used `StatelessWidget` components to create user interfaces. These widgets are perfect for building static layouts, **but they cannot change**. Flutter has another type of widget called `StatefulWidget`. **Stateful widgets can keep information** and know how to recreate themselves whenever their `State` changes.

Compared to StatelessWidgets, StatefulWidgets have a few more moving parts. In this recipe, we're going to create a very simple stopwatch that increments its counter once a second.

Getting ready

Let's start off by creating a brand new project. Open your IDE (or Terminal) and create a new project called `stopwatch`. Once the project has been generated, delete all the code in `main.dart` to start with a clean app.

How to do it...

Let's start building our stopwatch with a basic counter that auto increments:

1. We need to create a new shell for the app that will host the `MaterialApp` widget. This root app will still be stateless, but things will be more mutable in its children. Start off by adding the following code for the initial shell (`StopWatch` has not been created yet, so you will see an error that we will fix shortly):

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

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

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

2. Create a new file called `stopwatch.dart`. A `StatefulWidget` is divided into two classes – the widget and its state. There are IDE shortcuts that can generate this in one go, just like there are for `StatelessWidget`. However, for your first one, create the widget manually. Add the following code to create the `StatefulWidget` stopwatch:

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

class StopWatch extends StatefulWidget {
 @override
 State createState() => StopWatchState();
}
```

3. Every `StatefulWidget` needs a `State` object that will maintain its life cycle. This is a completely separate class. `StatefulWidgets` and their `State` are so tightly coupled that this is one of the few scenarios where you should keep more than one class in a single file. Create the private `_StopWatchState` class right after the `StopWatch` class:

```
class StopWatchState extends State<StopWatch> {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Stopwatch'),
      ),
      body: Center(
        child: Text(
          '0 seconds',
          style: Theme.of(context).textTheme.headline5,
```

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

`State` looks almost like a `StatelessWidget`, right? **In StatefulWidgets, you put the `build` method in the `State` class, not in the widget.**

4. In the `main.dart` file, add an import for the `stopwatch.dart` file:

```
import './stopwatch.dart';
```

5. Run the app. You should see a screen with text stating "0 Seconds" at the center of the screen, but this text won't change. We can solve this by keeping track of a `seconds` property in our `State` class and using a `Timer` to increment that property every second. Every time the timer ticks, we're going to tell the stopwatch to redraw by calling `setState`.

6. Add the following code just after the `class` definition, but before the `build` method:

```
class StopWatchState extends State<StopWatch> {
  int seconds;
  Timer timer;

  @override
  void initState() {
    super.initState();

    seconds = 0;
    timer = Timer.periodic(Duration(seconds: 1), _onTick);
  }

  void _onTick(Timer time) {
    setState(() {
      ++seconds;
    });
  }
```

7. Now, it's just a simple matter of updating the `build` method so that it uses the current value of the `seconds` property instead of a hardcoded value. First, let's add this helper function just after the `build` method to make sure that our text is always grammatically correct:

```
String _secondsText() => seconds == 1 ? 'second' : 'seconds';
```

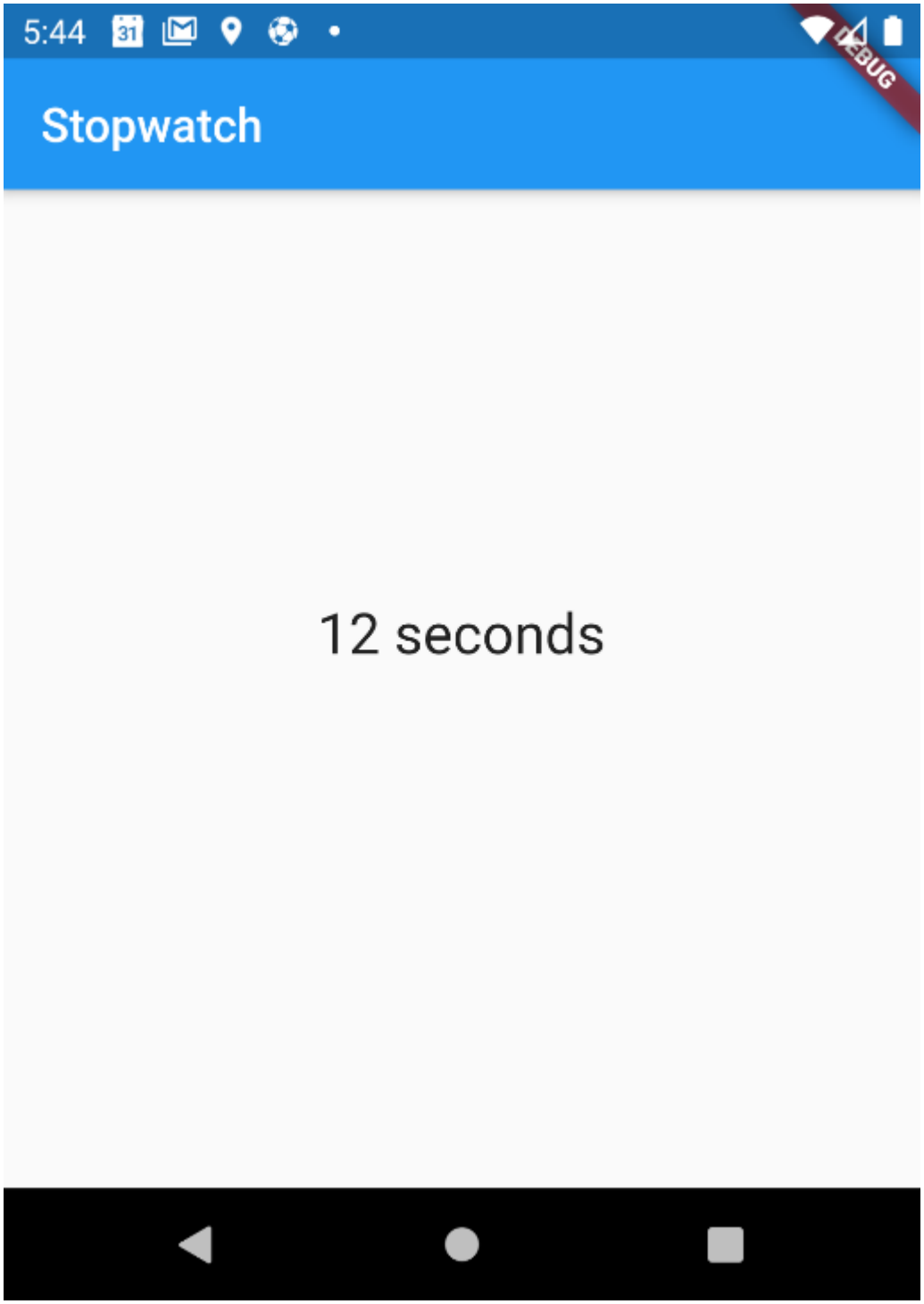
8. Update the `Text` widget in the build method so that it can now use the `seconds` property:

```
Text(
  '$seconds ${_secondsText()}',
  style: Theme.of(context).textTheme.headline5,
),
```

9. Finally, we just need to make sure the timer stops ticking when we close the screen. Add the following `dispose` method at the bottom of the state class, just after the `_secondsText` method:

```
@override
void dispose() {
 timer.cancel();
 super.dispose();
}
```

Run the app. You should now see a counter incrementing once a second. Pretty fancy!



How it works...

StatefulWidgets are made up of two classes: the widget and the state. The *widget* part of `StatefulWidget` really doesn't do much, and all the properties that you store in it must be final; otherwise, you will get a compile error.

All widgets, whether they are stateless or stateful, are still immutable. In `Stateful` widgets, the state can change.

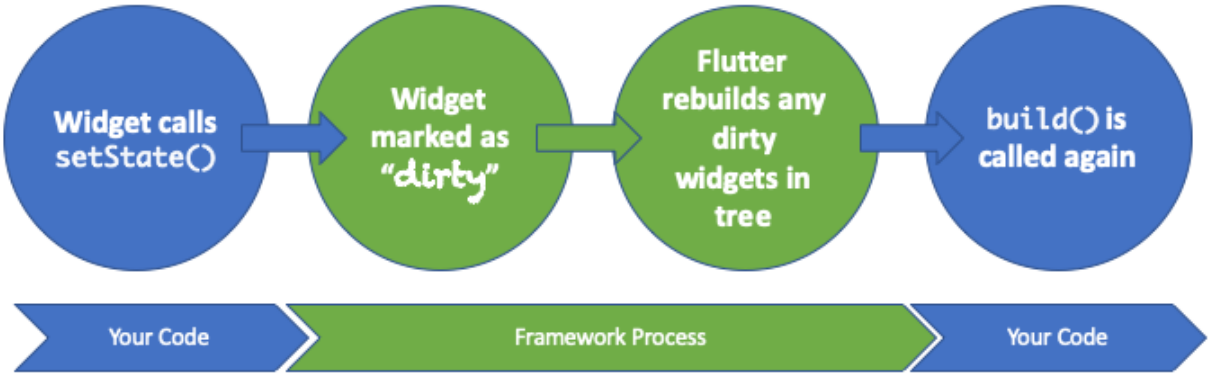
What doesn't have to be immutable is the `State` object. The `State` object takes over the build responsibilities from the widget. States can also be marked as dirty, which is what will cause them to repaint on the next frame. Take a close look at this line:

```
setState(() {
    ++seconds;
});
```

The `setState` function tells Flutter that a widget needs to be repainted. In this specific example, we are incrementing the `seconds` property by one, which means that when the `build` function is called again, it will replace the `Text` widget with different content.

Each time you call `setState`, the widget is repainted.

The following diagram summarizes how Flutter's render loop is impacted by `setState`:



Please note that the closure that you use in `setState` is completely optional. It's more for code legibility purposes. We could just as easily written the following code and it would have had identical results:

```
seconds++
setState(() {});
```

You should also avoid performing any complex operations inside the `setState` closure since that can cause performance bottlenecks. It is typically used for simple value assignments.

There's more...

The `State` class has a **life cycle**. Unlike `StatelessWidget`, which is nothing more than a build method, StatefulWidgets have a few different life cycle methods that are called in a specific order. In this recipe, you used `initState` and `dispose`, but the full list of life cycle methods, in order, is as follows:

- **initState**
- **didChangeDependencies**
- didUpdateWidget
- **build (required)**
- reassemble
- deactivate
- **dispose**

The methods in bold are the most frequently used life cycle methods. While you could override all of them, you will mostly use the methods in bold. Let's briefly discuss these methods and their purpose:

- **initState**:

This method is used to initialize any non-final value in your state class. You can think of it as performing a job similar to a constructor. In our example, we used `initState` to kick off a `Timer` that fires once a second. This method is called **before** the widget is added to the tree, so you do not have any access to the state's `BuildContext` property.

- **didChangeDependencies**:

This method is called immediately after `initState`, but unlike that method, the widget now has access to its `BuildContext`. If you need to do any setup work that requires context, then this the most appropriate method for those processes.

- **build**:

The State's build method is identical to StatelessWidget's `build` method and is required. Here, you are expected to define and return this widget's tree. In other words, you should create the UI.

- **dispose**:

This cleanup method is called when the `state` object is removed from the widget tree. This is your last opportunity to clean up any resources that need to be explicitly released. In the recipe, we used the `dispose` method to stop the Timer. Otherwise, the time would just keep on ticking, even after the widget has been destroyed. Forgetting to close long-running resources can lead to memory leaks and even crashes.

See also

Check out these resources for more information about StatefulWidgets:

- Video on StatefulWidgets by the Flutter team: https://www.youtube.com/watch?v=AqCMFXEmf3w
- State documentation: https://api.flutter.dev/flutter/widgets/State-class.html

Interacting with buttons

Buttons are one of the most important types of interactions in apps. It's almost impossible to imagine an app that doesn't have a button in some form or another. They are extremely flexible: you can customize their shape, color, and touch effects; provide haptic feedback; and more. But regardless of the styling, all buttons serve the same purpose. A button is a widget that users can touch (or press, or click). When their finger lifts off the button, they expect the button to react. Over the years, we have interacted with so many buttons that we don't even think about this interaction anymore – it has become obvious.

In this recipe, you are going to add two buttons to the stopwatch app: one to start the counter and another to stop it. You going are going to use two different styles of buttons – `ElevatedButton` and `TextButton` – for these different functions, but note that even though they *look* different, their API is the same.

Getting ready

We're going to continue with the Stopwatch project. Make sure you have completed the previous recipe since this one builds on the existing code.

How to do it...

To complete this recipe, open `stopwatch.dart` and follow these steps:

1. Add some buttons to the screen.
2. Update the `build` method in `StopWatchState` to replace the `Center` widget with a `Column`. You should be able to use the intentions dialog that we discussed in the previous chapter to quickly wrap the `Text` widget in a `Column` and then remove the `Center` widget. When you are done, the `build` method should look like this:

```
return Scaffold(
  appBar: AppBar(
    title: Text('Stopwatch'),
  ),
  body: Column(
    mainAxisAlignment: MainAxisAlignment.center,
    children: <Widget>[
      Text(
        '$seconds ${_secondsText()}',
        style: Theme.of(context).textTheme.headline5,
      ),
      SizedBox(height: 20),
      Row(
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
          ElevatedButton(
                style: ButtonStyle(
```

```
                  backgroundColor: MaterialStateProperty
                  .all<Color>(Colors.green),
                  foregroundColor: MaterialStateProperty
                  .all<Color>(Colors.white),
                  ),
                child: Text('Start'),
                onPressed: null,
              ),
          SizedBox(width: 20),
          TextButton(
                style: ButtonStyle(
                  backgroundColor: MaterialStateProperty
                  .all<Color>(Colors.red),
                  foregroundColor: MaterialStateProperty
                  .all<Color>(Colors.white),
                  ),
                child: Text('Stop'),
                onPressed: null,
              ),
        ],
      )
    ],
  ),
);
```

3. If you hot reload, you'll see two buttons on the screen.
4. You can try tapping them, but nothing will happen. This is because you have to add an `onPressed` function, before the button will activate.
5. Let's add a property at the top of the `State` class that will keep track of whether the timer is ticking or not and optionally add `onPressed` functions depending on that state value.
6. Add this line at the very top of `StopWatchState`:

```
bool isTicking = true;
```

7. Add two functions that will toggle this property and cause the widget to repaint. Add these methods under the `build` method:

```
void _startTimer() {
  setState(() {
    isTicking = true;
  });
}

void _stopTimer() {
  setState(() {
    isTicking = false;
  });
}
```

8. Now, you can hook these methods into the `onPressed` property.
9. Update your buttons so that they use these ternary operators in button declarations. Some of the setup code has been omitted for brevity:

```
ElevatedButton(
  child: Text('Start'),
  onPressed: isTicking ? null : _startTimer,
  ...
),
...
TextButton(
  child: Text('Stop'),
  onPressed: isTicking ? _stopTimer : null,
  ...
),
```

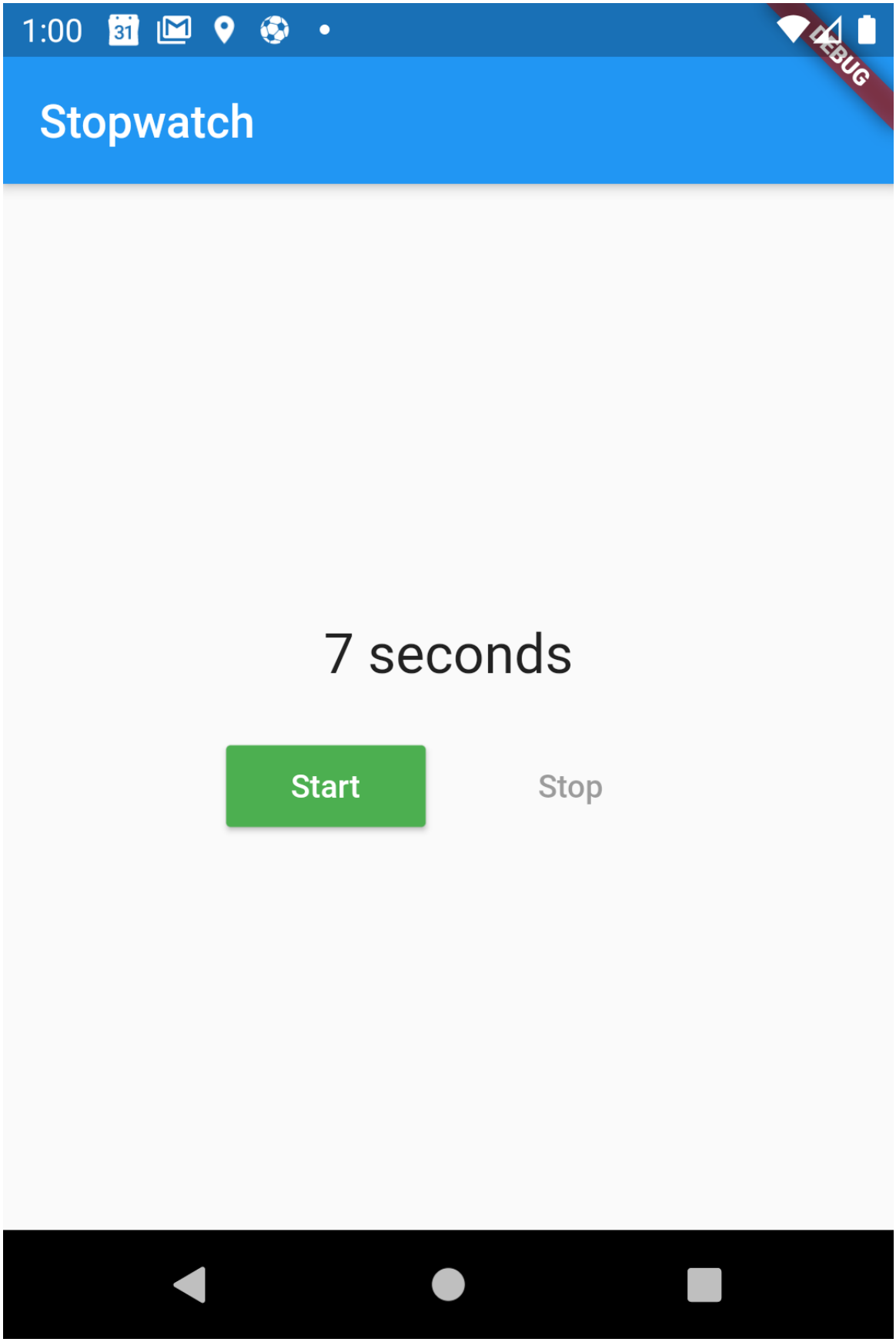
10. Now, it's time to add some logic to the start and stop methods to make the timer respond to our interactions.
11. Update those buttons so that they include the following code:

```
void _startTimer() {
  timer = Timer.periodic(Duration(seconds: 1), _onTick);

  setState(() {
    seconds = 0;
    isTicking = true;
  });
}

void _stopTimer() {
  timer.cancel();

  setState(() {
    isTicking = false;
  });
}
```

12. One more quick thing – we don't really need the `initState` method anymore now that the buttons are controlling the timer. **Delete** the `initState` method and update the `seconds` property at the top of the class so that it is initialized with a value instead of `null`:

```
int seconds = 0;
```

Congratulations – you should now have a fully functioning timer app!


1:00
DEBUG
Stopwatch
7 seconds
Start
Stop

How it works...

Buttons in Flutter are pretty simple – they are just widgets that accept a function. These functions are then executed when the button detects an interaction. If a `null` value is supplied to the `onPressed` property, Flutter considers the button to be disabled.

Flutter has several button types that can be used for different aesthetics, but their functionality is the same. They are as follows:

- `ElevatedButton`
- `TextButton`
- `IconButton`
- `FloatingActionButton`
- `DropDownButton`
- `CupertinoButton`

You can play around with any of these widgets until you find a button that matches your desired look.

In this recipe, we wrote the `onPressed` functions out as methods in the `StopWatchState` class, but it is perfectly acceptable to throw them into the functions as closures. We could have written the buttons like this:

```
ElevatedButton(
  child: Text('Start'),
  onPressed: isTicking
      ? null
      : () {
          timer = Timer.periodic(Duration(seconds: 1), _onTick);

          setState(() {
            seconds = 0;
            isTicking = true;
          });
        },
),
```

For simple actions, this is fine, but even in our simple example, where we want to control whether the button is active or not via a ternary operator, this is already becoming harder to read.

Making it scroll

It is very rare to encounter an app that doesn't have some sort of scrolling content. Scrolling, especially vertical scrolling, is one of the most natural paradigms in mobile development. When you have a list of elements that can extend beyond the height of a screen, you'll need to use some sort of scrollable widget.

Scrolling content is actually rather easy to accomplish in Flutter. To get started with scrolling, a great widget is `ListView`. Just like Columns, ListViews control a list of child widgets and place them one after another. However, ListViews will also make that content scroll automatically when their height is bigger than the height of their parent widget.

In this recipe, we're going to add laps to our stopwatch app and display those laps in a scrollable list.

Getting ready

Once again, we're going to continue with the StopWatch project. You should have completed the previous recipes in this chapter before following along with this one.

How to do it...

Open up `stopwatch.dart` to get started:

1. The first thing you are going to do is make the timer a bit more precise. Seconds are not a very interesting value to use for stopwatches.
2. Use the refactoring tools to rename the `seconds` property to `milliseconds`. We also need to update the `onTick`, `_startTimer`, and `_secondsText` methods:

```
int milliseconds = 0;

void _onTick(Timer time) {
  setState(() {
    milliseconds += 100;
  });
}
void _startTimer() {
  timer = Timer.periodic(Duration(milliseconds: 100), _onTick);
  ...
}
String _secondsText(int milliseconds) {
  final seconds = milliseconds / 1000;
  return '$seconds seconds';
}
```

3. Now, let's add a laps list so that we can keep track of the values for each lap. We're going to add to this list every time the user taps a lap button.
4. Add this property to the top of the `StopWatchState` class, just under the declaration of the timer:

```
final laps = <int>[];
```

5. Create a new `lap` method to increment the lap count and reset the current millisecond value:

```
void _lap() {
  setState(() {
    laps.add(milliseconds);
    milliseconds = 0;
  });
}
```

6. We also need to tweak the `_startTimer` method in order to reset the lap list every time the user starts a new counter.

7. Add the following line inside the `setState` closure, in `_startTimer`:

```
laps.clear();
```

8. Now, we need to organize our widget code a bit to enable adding scrollable content. Let's start off by taking the existing `Column` in the `build` method and extracting it into its own method called `_buildCounter`. The refactoring tools should automatically add a `BuildContext` to that method. Don't forget to change the return type of this new method from `Column` to `Widget`.

9. To make the UI a bit nicer, wrap `Column` into a `Container` and set its background to the app's primary color. Also, add one more `Text` widget above the counter to show which lap you are currently on.

10. Finally, make sure that you adjust the color of the text to white so that it's readable on a blue background. The top of the method will look like this:

```
Widget _buildCounter(BuildContext context) {
  return Container(
    color: Theme.of(context).primaryColor,
    child: Column(
      mainAxisAlignment: MainAxisAlignment.center,
      children: <Widget>[
        Text(
          'Lap ${laps.length + 1}',
          style: Theme.of(context)
              .textTheme
              .subtitle1
              .copyWith(color: Colors.white),
        ),
        Text(
          _secondsText(milliseconds),
          style: Theme.of(context)
              .textTheme
              .headline5
              .copyWith(color: Colors.white),
        ),
```

11. Inside the `_buildCounter` method, extract `Row`, where the buttons are built into its own method called `_buildControls`. As always, the return type of that new method should be changed to `Widget`.

12. Add a new button between the start and stop buttons that will call the `lap` method. Put a `SizedBox` before and after this button to give it some spacing:

```
SizedBox(width: 20),
ElevatedButton(
          style: ButtonStyle(
            backgroundColor: MaterialStateProperty
            .all<Color>(Colors.yellow),),
          child: Text('Lap'),
          onPressed: isTicking ? _lap : null,
        ),
SizedBox(width: 20),
```

13. That's a lot of refactoring! But it was all for a good purpose. Now, you can finally add a `ListView`.

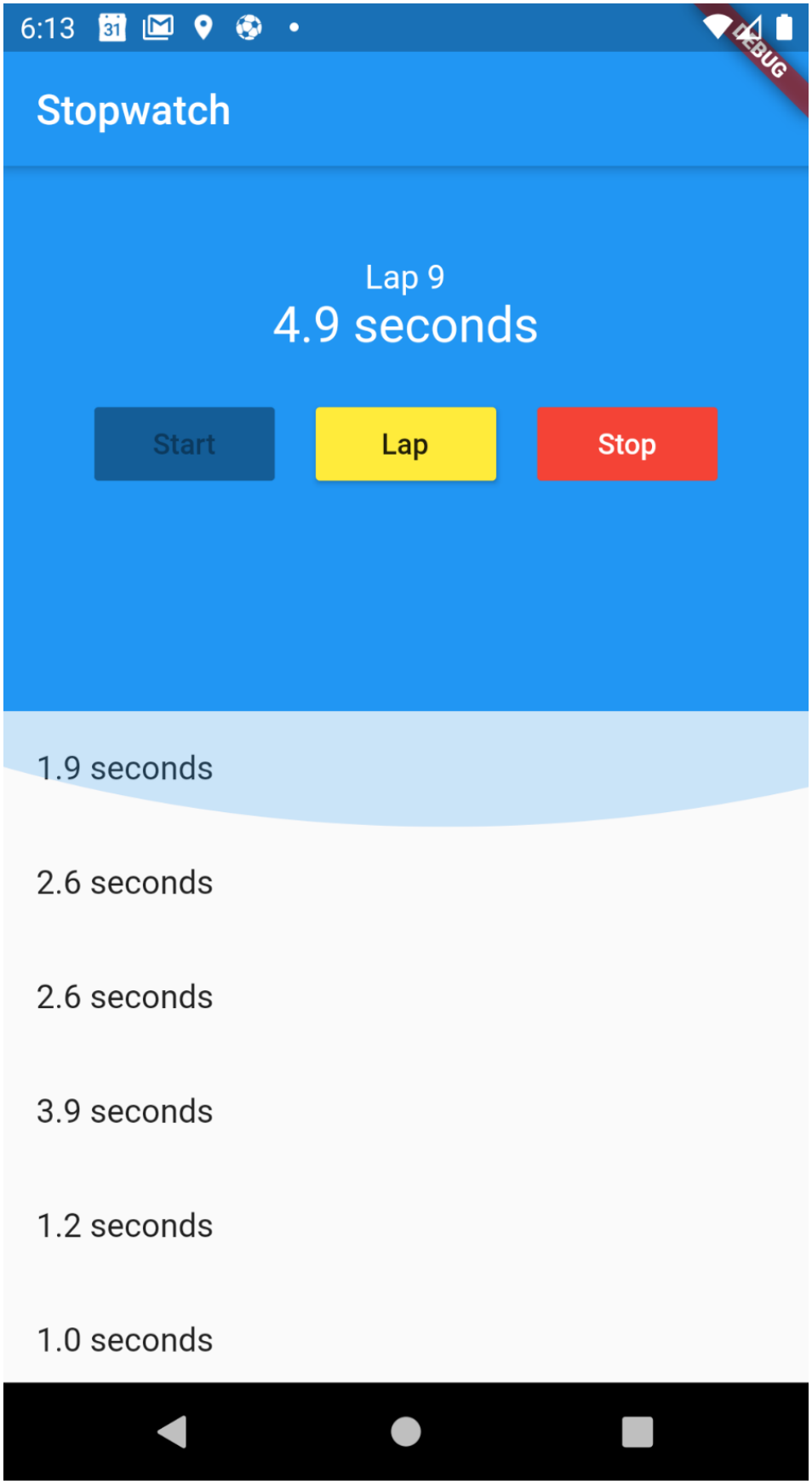
14. Add the following method before the `dispose()` method. This will create the scrollable content for the laps:

```
Widget _buildLapDisplay() {
  return ListView(
    children: [
      for (int milliseconds in laps)
        ListTile(
          title: Text(_secondsText(milliseconds)),
        )
    ],
  );
}
```

15. Update the primary `build` method to use the new scrolling content. Wrap both top-level widgets in an `Expanded` so that both the stopwatch and our `ListView` take up half the screen:

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text('Stopwatch'),
    ),
    body: Column(
      children: <Widget>[
        Expanded(child: _buildCounter(context)),
        Expanded(child: _buildLapDisplay()),
      ],
    ),
  );
}
```

16. Run the app. You should now be able to add laps to your stopwatch. After adding a few laps, you'll be able to see the laps scroll:


6:13
DEBUG
Stopwatch
Lap 9
4.9 seconds
Start
Lap
Stop
1.9 seconds
2.6 seconds
2.6 seconds
3.9 seconds
1.2 seconds
1.0 seconds

How it works...

As you have seen in this recipe, creating scrollable widgets in Flutter is easy. There is only one method that creates a scrolling widget and that method is *tiny*. Scrolling in Flutter is just a simple matter of choosing the correct widget and passing your data. The framework takes care of the rest.

Let's break down the scrolling code that is in this recipe:

```
Widget _buildLapDisplay() {
  return ListView(
    children: [
      for (int milliseconds in laps)
        ListTile(
          title: Text(_secondsText(milliseconds)),
        )
    ],
  );
}
```

We're using a type of scrolling widget called `ListView`, which is probably one of the simplest types of scrolling widgets in Flutter. This widget functions a bit like a column, except instead of throwing errors when it runs out of space, `ListView` will enable scrolling, allowing you to use drag gestures to scroll through all the data.

In our example, we're also using the **collection-for** syntax to create the widgets for this list. This will essentially create one very long column as you add laps.

One other interesting thing about scrolling in Flutter is that it is platform aware. If you can, try running the app in both the Android Emulator and the iOS Simulator; you'll notice that the scroll *feels* different. What you are encountering is something called `ScrollPhysics`. These are objects that define how the list is supposed to scroll and what happens when you get to the end of the list. On iOS, the list is supposed to bounce, whereas, on Android, you get a glow effect when you get to the edges. The widget can pick the correct `ScrollPhysics` strategy based on the platform, but you can also override this behavior if you want to make the app behave in a particular way, regardless of the platform:

```
ListView(
  physics: BouncingScrollPhysics(),
  children: [...],
);
```

You generally shouldn't override the platform's expected behavior, unless there is a good reason for doing so. It might confuse your users if they start adding iOS paradigms on Android and vice versa.

There's more...

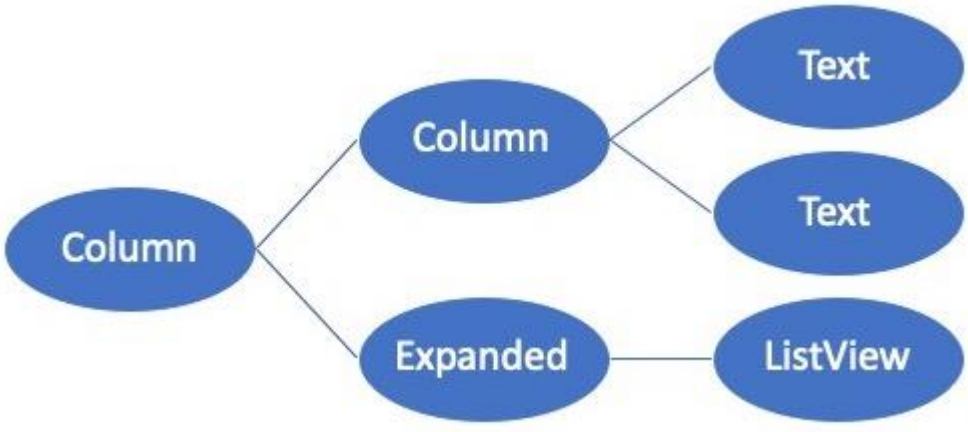
One final important thing to keep in mind about scrolling widgets is that because they need to know their parent's constraints to activate scrolling, putting scroll widgets inside widgets with unbounded constraints can cause Flutter to throw errors.

In our example, we placed `ListView` inside a `Column`, which is a flex widget that lays out its children based on their intrinsic size. This works fine for widgets such as `Containers`, `Buttons`, and `Text`, but it fails for ListViews. To make scrolling work inside `Column`, we had to wrap it in an `Expanded` widget, which will then tell `ListView` how much space it has to work with. Try removing `Expanded`; the whole widget will disappear and you should see an error in the Debug console:

```
I/flutter (28416): ═╡ EXCEPTION CAUGHT BY RENDERING LIBRARY ╞═══════════════════════════════════
I/flutter (28416): The following assertion was thrown during performResize():
I/flutter (28416): Vertical viewport was given unbounded height.
I/flutter (28416): Viewports expand in the scrolling direction to fill their container.In this case, a vertical
I/flutter (28416): viewport was given an unlimited amount of vertical space in which to expand. This situation
I/flutter (28416): typically happens when a scrollable widget is nested inside another scrollable widget.
I/flutter (28416): If this widget is always nested in a scrollable widget there is no need to use a viewport because
I/flutter (28416): there will always be enough vertical space for the children. In this case, consider using a Column
I/flutter (28416): instead. Otherwise, consider using the "shrinkWrap" property (or a ShrinkWrappingViewport) to size
I/flutter (28416): the height of the viewport to the sum of the heights of its children.
I/flutter (28416):
I/flutter (28416): When the exception was thrown, this was the stack:
I/flutter (28416): #0      RenderViewport.performResize.<anonymous closure> (package:flutter/src/rendering/viewport.dart:1147:15)
I/flutter (28416): #1      RenderViewport.performResize (package:flutter/src/rendering/viewport.dart:1200:6)
I/flutter (28416): #2      RenderObject.layout (package:flutter/src/rendering/object.dart:1604:9)
```

These types of errors can be pretty unsettling to see and don't always immediately tell you how to fix your code. There is also a long explosion of log entries that have nothing to do with your code. When you see this error, it just means that you have an unbounded scrolling widget. If you place a scrolling widget inside a `Flex` widget, which is pretty common, just don't forget to always wrap the scrolling content in `Expanded` or `Flexible` first.

Use this chart as a reference when you're designing your scrolling widget trees:



Handling large datasets with list builders

There is an interesting trick that mobile apps use when they need to render lists of data that can potentially contain more entries than your device has memory to display. This was especially critical in the early days of mobile app development, when phones were a lot less powerful than they are today. Imagine that you had to create a contacts app, where your user could potentially have hundreds and hundreds of scrollable contacts. If you put them all in a single `ListView` and asked Flutter to create all of these widgets, there would be a point where your app could run out of memory, slow down, and even potentially crash.

Take a look at the contacts app on your phone and scroll up and down really fast. These apps don't show any *delay* while scrolling, and they certainly aren't in any danger of crashing because of the amount of data. What's the secret? If you look carefully at your app, you'll see that only so many items can fit on the screen at once, regardless of how many entries there are in your list. So, some smart engineers figured out they can *recycle* these views. When a widget moves off screen, why not reuse it with the new data? This trick has existed since the beginning of mobile development and is no different today.

In this recipe, we're going to optimize the stopwatch app from the previous recipe to employ recycling when our dataset grows beyond what our phones can handle.

How to do it...

Open the `stopwatch.dart` file and dive right into the `ListView` code:

1. Let's replace `ListView` with one of its variants, `ListView.builder`. Replace the existing implementation of `_buildLapDisplay` with this one:

```
Widget _buildLapDisplay() {
  return ListView.builder(
    itemCount: laps.length,
    itemBuilder: (context, index) {
      final milliseconds = laps[index];
      return ListTile(
        contentPadding: EdgeInsets.symmetric(horizontal: 50),
        title: Text('Lap ${index + 1}'),
        trailing: Text(_secondsText(milliseconds)),
      );
    },
  );
}
```

2. ScrollViews can get too big, so it's usually a good idea to show the user their position in the list. Wrap `ListView` in a `Scrollbar` widget. There aren't any special properties to enter since this widget is entirely context aware:

```
return Scrollbar(
  child: ListView.builder(
    itemCount: laps.length,
```

3. Finally, add a quick new feature for the list to scroll to the bottom every time the lap button is tapped. Flutter makes this easy with the `ScrollController` class. At the top of `StopWatchState`, just below the laps list, add these two properties:

```
final itemHeight = 60.0;
final scrollController = ScrollController();
```

4. Now, we need to link these values with `ListView` by feeding them into the widget's constructor:

```
ListView.builder(
  controller: scrollController,
  itemExtent: itemHeight,
  itemCount: laps.length,
  itemBuilder: (context, index) {
```

5. All we have to do now is tell `ListView` to scroll when a new lap is added.
6. At the bottom of the `_lap()` method, just after the call to `setState`, add this line:

```
scrollController.animateTo(
  itemHeight * laps.length,
  duration: Duration(milliseconds: 500),
  curve: Curves.easeIn,
);
```

Run the app and try adding several laps. Your app can now handle virtually any number of laps without effort.

How it works...

To build an optimized `ListView` with its `builder` constructor, you need to tell Flutter how large the list is via the `itemCount` property. If you don't include it, Flutter will think that the list is infinitely long and it will never terminate. There may be a few cases where you want to use an infinite list, but they are rare. In most cases, you need to tell Flutter how long the list is; otherwise, you will get an "out of bounds" error.

The secret to scrolling performance is found in the `itemBuilder` closure. In the previous recipe, you added a list of known children to `ListView`. This forces Flutter to create and maintain the entire list of widgets. Widgets themselves are not that expensive, but the `Elements` and `RenderObjects` properties that sit underneath the widgets inside Flutter's internals are.

`itemBuilder` solves this problem by enabling **deferred rendering**. We are no longer providing Flutter with a list of widgets. Instead, we are waiting for Flutter to use what it needs and only creating widgets for a *subset* of our list. As the user scrolls, Flutter will continuously call the `itemBuilder` function with the appropriate index. When widgets move off the screen, Flutter can remove them from the tree, freeing up precious memory. Even if our list is thousands of

entries long, the size of the viewport is not going to change, and we are only going to need the same fixed number of visible entries at a time. The following diagram exemplifies this point:

As this viewport moves up and down the list, only seven items can fit on the screen at a time. Subsequently, nothing is gained by creating widgets for all 20 items. As the viewport moves to the left, we will probably need items 3, 2, and 1, but items 8, 9, and 10 can be dropped. The internals for how all this is executed are handled by Flutter. There is actually no API access to how Flutter optimizes your `ListView`. You just need to pay attention to the index that Flutter is requesting from `itemBuilder` and return the appropriate widget.

There's more...

We also touched on two more advanced scrolling topics in this recipe – `itemExtent` and `ScrollController`.

The `itemExtent` property is a way to supply a fixed height to all the items in `ListView`. Instead of letting the widget figure out its own height based on the content, using the `itemExtent` property will enforce a fixed height for every item. This has added performance benefits, since `ListView` now needs to do less work when laying out its children, and it also makes it easier to calculate scrolling animations.

`ScrollController` is a special object that allows to key into `ListView` from outside the build methods. This is a frequently used pattern in Flutter where you can optionally provide a controller object that has methods to manipulate its widget. `ScrollController` can do many interesting things, but in this recipe, we just used it to animate `ListView` from the `_lap` method:

```
scrollController.animateTo(
  itemHeight * laps.length,
  duration: Duration(milliseconds: 500),
  curve: Curves.easeIn,
);
```

The first property of this function wants to know where in `ListView` to scroll. Since we have previously told Flutter that these items are all going to be of a fixed height, we can easily calculate the total height of the list by multiplying the number of items by the fixed height constant. The second property dictates the length of the animation, while the final property tells the animation to slow down as it reaches its destination instead of stopping abruptly. We will discuss animations in detail later in this book.

Working with TextFields

Together with buttons, another extremely common form of user interaction is the text field. There comes a point in most apps where your users will need to type something; for example, a form where users need to type in their username and password.

Because the text is often related to the concept of forms, Flutter also has a subclass of `TextField` called `TextFormField`, which adds functionality for multiple text fields to work together.

In this recipe, we're going to create a login form for our stopwatch app so that we know which runner we're timing.

Getting ready

Once again, we're going to continue with the StopWatch project. You should have completed the previous recipes in this chapter before following along with this one.

In the `main.dart` file, in the home property of `MaterialApp`, add a call to the `LoginScreen` class. We will be creating this in this recipe:

```
home: LoginScreen(),
```

How to do it...

We're going to take a small break from the stopwatch for this recipe:

1. Create a new file called `login_screen.dart` and generate the code snippet for a new `StatefulWidget` by typing `stful` and tapping *Enter*. Your IDE will automatically create a placeholder widget and its state class.
2. Name this class `LoginScreen`. The `State` class will automatically be named `_LoginScreenState`. Don't forget to fix your missing imports by bringing in the material library.
3. Our login screen needs to know whether the user is logged in to show the appropriate widget tree. We can handle this by having a boolean property called `loggedIn` and forking the widget tree accordingly.

4. Add the following code just under the class declaration of `_LoginScreenState`:

```
bool loggedIn = false;
String name;

@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text('Login'),
```

```
    ),
    body: Center(
      child: loggedIn ? _buildSuccess() : _buildLoginForm(),
    ),
  );
}
```

5. The success widget is pretty simple – it's just a checkmark and a `Text` widget to print whatever the user typed:

```
Widget _buildSuccess() {
  return Column(
    mainAxisAlignment: MainAxisAlignment.center,
    children: <Widget>[
      Icon(Icons.check, color: Colors.orangeAccent),
      Text('Hi $name')
    ],
  );
}
```

6. Now that that's done, we can get into the meat of the recipe: the login form.
7. Add some more properties at the top of the class; that is, two TextEditingControllers to handle our `TextField` properties and `GlobalKey` to handle our `Form`:

```
final _nameController = TextEditingController();
final _emailController = TextEditingController();
final _formKey = GlobalKey<FormState>();
```

8. We can implement the form using Flutter's `form` widget to wrap a `column`:

```
Widget _buildLoginForm() {
  return Form(
      key: _formKey,
      child: Padding(
          padding: const EdgeInsets.all(20.0),
          child: Column(
            mainAxisAlignment: MainAxisAlignment.center,
            children: [
              TextFormField(
                controller: _nameController,
                decoration: InputDecoration(labelText: 'Runner'),
                validator: (text) =>
                    text.isEmpty ? 'Enter the runner\'s
                      name.' : null,
              ),
            ],
          )));
}
```

9. Run the app. You will see a single `TextField` floating in the center of the screen. Now, let's add one more field to manage the user's email address.

10. Add this widget inside `Column`, just after the first `TextFormField`. This widget uses a **regular expression** to validate its data:

```
TextFormField(
  controller: _emailController,
  keyboardType: TextInputType.emailAddress,
  decoration: InputDecoration(labelText: 'Email'),
  validator: (text) {
    if (text.isEmpty) {
      return 'Enter the runner\'s email.';
    }

    final regex = RegExp('[^@]+@[^\.]+\..+');
    if (!regex.hasMatch(text)) {
      return 'Enter a valid email';
    }

    return null;
  },
),
```

Regular expressions are sequences of characters that specify a search pattern, and they are often used for input validation. To learn more about regular expressions in Dart, go to https://api.dart.dev/stable/2.12.4/dart-core/RegExp-class.html.

11. The form items set should be all set up; now, you just need a way to validate them. This can be accomplished with a button and function that calls the form's `validateAndSubmit` method.

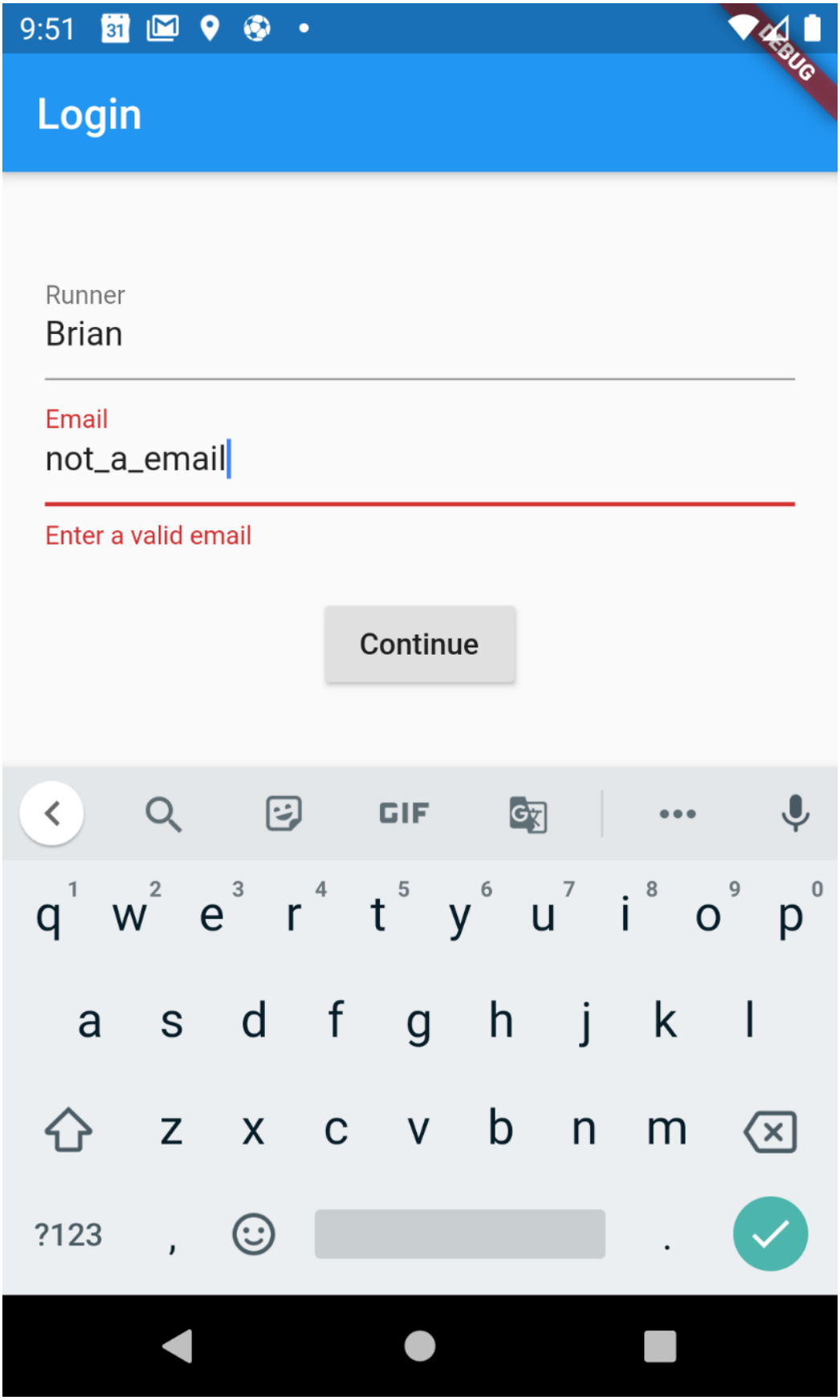
12. Add these two widgets inside the same `Column`, just after the second `TextField`:

```
SizedBox(height: 20),
ElevatedButton(
 child: Text('Continue'),
 onPressed: _validate,
),
```

13. Now, implement the `validate()` method:

```
void _validate() {
  final form = _formKey.currentState;
  if (!form.validate()) {
    return;
  }

  setState(() {
    loggedIn = true;
    name = _nameController.text;
  });
}
```

Perform a hot reload. As a bonus, try entering incorrect information into the form and see what happens:


9:51
DEBUG
Login
Runner
Brian
Email
not_a_email
Enter a valid email
Continue
GIF
q w e r t y u i o p
a s d f g h j k l
z x c v b n m
?123
,
.

How it works...

This recipe actually covered quite a few topics very quickly - `TextFields`, `Forms`, and `Keys`.

TextFields are platform-aware widgets that respect the host platform's UX paradigms. As with most things in Flutter, the look of TextFields is highly customizable. The default look respects the material design rules, but it can be fully customized using the `InputDecoration` property. By now, you should be noticing some common patterns in Flutter's API. Many widgets – `Containers`, `TextFields`, `DecoratedBox`, and so on – can all accept a secondary decoration object. It could even be argued that the consistency of the API design for these widgets has led to a sort of self-documentation. For example, can you guess what this line does to the second `TextField`?

```
keyboardType: TextInputType.emailAddress,
```

If you guessed that it lets us use the email keyboard instead of the standard keyboard, then congratulations – that's correct!

In this recipe, you used a variant of `TextField` called `TextFormField`. This subclass of `TextField` adds some extra validation callbacks that are called when submitting a form.

The first `validator` is simple – it just checks if the text is empty. Validator functions should return a string if the validation **fails.** If the validation is successful, then the function should return a `null`. This is one of the very few cases in the entire Flutter SDK where `null` is a good thing.

The `Form` widget that wraps the two TextFields is a non-rendering container widget. This widget knows how to visit any of its children that are FormFields and invokes their validators. If all the validator functions return `null`, the form is considered **valid**.

You used a `GlobalKey` to get access to the form's state class from outside the `build` method. A simple way to explain GlobalKeys is that they do the opposite of `BuildContext`. `BuildContext` is an object that can find **parents** in the widget tree. Keys are objects that are used to retrieve a **child** widget. The topic is a bit more complex than that, but in short, with the key, you can retrieve the Form's state. The `FormState` class has a public method called `validate` that will call the validator on all its children.

Keys go much deeper than this. However, they are an advanced topic that is outside the scope of this book. There is a link to an excellent article by Google's Emily Fortuna about keys in the See also... section of this recipe if you want to learn more about this topic.

Finally, we have `TextEditingController`. Just like `ScrollController` in the previous recipe, TextEditingControllers are objects that can be used to manipulate TextFields. In this recipe, we only used them to extract the current value from our `TextField`, but they can also be used to programmatically set values in the widget, update text selections, and clear the fields. They are very helpful objects to keep in your arsenal.

There are also some callback functions that be used on `TextFields` to accomplish the same thing. For instance, if you want to update the name property every time the user types a letter, you could use the `onChanged` callback that is on the core `TextField` (not `TextFormField`). In practice, having lots of callbacks and inline closures can make your functions very long and hard to read. So, while it may seem easier to use closures instead of `TextEditingControllers`, it could make your code harder to read. Clean code suggests that functions should only strive to do one thing, which means one function should handle the setup and aesthetic of your `TextView` and another function should handle the logic.

See also

Check out these resources for more information:

- Article and video about keys by Emily Fortuna. Don't miss this one!: https://medium.com/flutter/keys-what-are-they-good-for-13cb51742e7d
- Forms: https://api.flutter.dev/flutter/widgets/Form-class.html

Navigating to the next screen

So far, all our examples have taken place on a single screen. In most real-world projects, you might be managing several screens, each with their own paths that can be pushed and popped onto the screen.

Flutter, and more specifically `MaterialApp`, uses a class called `Navigator` to manage your app's screens. Screens are abstracted into a concept called `Routes`, which contain both the widget we want to show and how we want to animate them on the screen. `Navigator` also keeps a full history of your routes so that you can return to the previous screens easily.

In this recipe, we're going to link `LoginScreen` and `StopWatch` so that `LoginScreen` actually logs you in.

How to do it...

Let's start linking the two screens in the app:

1. Start by engaging in one the most enjoyable activities for a developer – deleting code.
2. Remove the `loggedIn` property and all the parts of the code where it's referenced. We're also no longer going to need the `buildSuccess()` method or the `ternary` method in the top `build` method.
3. Update the `build` method with the following snippet:

```
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text('Login'),
```

```
    ),
    body: Center(
      child: _buildLoginForm(),
    ),
  );
}
```

4. In the `_validate` method, we can kick off the navigation instead of calling `setState`:

```
void _validate() {
  final form = _formKey.currentState;
  if (!form.validate()) {
    return;
  }

  final name = _nameController.text;
  final email = _emailController.text;

  Navigator.of(context).push(
    MaterialPageRoute(
      builder: (_) => StopWatch(name: name, email: email),
    ),
  );
}
```

5. The constructor for the `StopWatch` widget needs to be updated so that it can accept the name and email. Make these changes in `stopwatch.dart`:

```
class StopWatch extends StatefulWidget {
  final String name;
  final String email;

  const StopWatch({Key key, this.name, this.email}) : super(key:
    key);
```

6. In the `build` method of the `StopWatchState` class, replace the title of `AppBar` with the runner's name, just to give the app a bit more personality:

```
AppBar(
  title: Text(widget.name),
),
```

7. You can now navigate back and forth between `LoginScreen` and `StopWatch`. However, isn't it a little strange that you can hit a back button to return to a login screen? In most apps, once you've logged in, that screen should no longer be accessible. You can use the Navigator's `pushReplacement` method to achieve this:

```
Navigator.of(context).pushReplacement(
    MaterialPageRoute(
      builder: (_) => StopWatch(name: name, email: email),
    ),
  );
```

Try logging in again. You will see that you no longer have the ability to return to the login screen.

How it works...

`Navigator` is a component of both `MaterialApp` and `CupertinoApp`. Accessing this object is yet another example of the ***of-context*** pattern. Internally, Navigators function as a stack. Routes can be pushed onto the stack and popped off the stack.

Normally, you would just use the standard `push()` and `pop()` methods to add and remove routes, but as we discussed in this recipe, we didn't just want to push `StopWatch` onto the screen – we also wanted to pop `LoginScreen` from the stack at the same time. To accomplish this, we used the `pushReplacement` method:

```
Navigator.of(context).pushReplacement(
    MaterialPageRoute(
```

We also used the `MaterialPageRoute` class to represent our routes. This object will create a platform-aware transition between the two screens. On iOS, it will push onto the screen from right, while on Android, it will pop onto the screen from the bottom.

Similar to `ListView.builder`, `MaterialPageRoute` also expects a `WidgetBuilder` instead of direct child. `WidgetBuilder` is a function that provides a `BuildContext` and expects a `Widget` to be returned:

```
builder: (_) => StopWatch(name: name, email: email),
```

This allows Flutter to delay the construction of the widget until it's needed. We also didn't need the context property, so it was replaced with an underscore.

Invoking navigation routes by name

Routing is a concept that is so engrained into the internet that we almost don't think about it anymore.

In Flutter, you can use **named routes**. This means you can assign a textual name to your screens and simply invoke them as if you were just going to another page on a website.

In this recipe, you are going to update the existing routing mechanism in the stopwatch project so that you can use named routes instead.

How to do it...

Open the existing stopwatch project to get started:

1. Named routes are referenced as strings. To reduce the potential for error, add some constants to the top of both `stopwatch.dart` and `login_screen.dart`:

```
class LoginScreen extends StatefulWidget {
  static const route = '/login';

class StopWatch extends StatefulWidget {
  static const route = '/stopwatch';
```

2. These routes need to be wired up in `MaterialApp` so that they can be fed to the app's `Navigator`.
3. Open `main.dart` and update the MaterialApp's constructor so that it includes these pages:

```
return MaterialApp(
  routes: {
    '/': (context) => LoginScreen(),
    LoginScreen.route: (context) => LoginScreen(),
    StopWatch.route: (context) => StopWatch(),
  },
  initialRoute: '/',
);
```

4. Now, we can invoke this route. Open `login_screen.dart` and scroll to the `_validate` method. Replace the existing navigator code at the bottom of the method by calling the `pushReplacementNamed` method:

```
Navigator.of(context).pushReplacementNamed(
      StopWatch.route,

    );
```

5. There is one significant difference between named routes and manually constructed routes – you cannot use custom constructors to pass data to the next screen. Instead, you can use **route arguments**.
6. Update `Navigator` so that it uses the runner's name from the form in the optional argument property:

```
final name = _nameController.text;
Navigator.of(context).pushReplacementNamed(
      StopWatch.route,
      arguments: name,
    );
```

7. To pull the data out of the route's argument, we need to retrieve the screen's route from its build method. Thankfully, this can be accomplished with the **of-context** pattern.
8. In the `build` method in `stopwatch.dart`, add the following code to the very top and update `AppBar`:

```
String name = ModalRoute.of(context).settings.arguments ?? '';
```

```
return Scaffold(
  appBar: AppBar(
    title: Text(name),
  ),
```

Hot reload the app. While you won't see any noticeable difference from the previous recipe, the code is slightly cleaner.

How it works...

If you look at the way we define the routes in `MaterialApp`, the **home** route is required. You can achieve this with the "/" symbol. Then, you can set up the other routes for your app:

```
routes: {
    '/': (context) => LoginScreen(),
    LoginScreen.route: (context) => LoginScreen(),
    StopWatch.route: (context) => StopWatch(),
  },
```

We have a bit of redundancy here because there are only two screens in this app. Once the routes have been declared, `MaterialApp` needs to know which route to start with. This is just inputted as a string:

```
initialRoute: '/',
```

It is recommended that you define constants for your routes and use those instead of string literals. In this recipe, we put the constants as static elements for each screen. There is no real requirement to organize your code like that; you could also keep your constants in a single file if you prefer.

The decision to use named routes over manually constructed routes is not entirely clear-cut. If you decide to go with named routes, there is a bit more planning and setup that is required upfront, without any significant benefits. It could be argued that code for named routes is a bit cleaner, but it is also harder to change. Ultimately, it might be easier to start developing with manually constructed routes and then ***refactor*** *toward named routes when the need arises.*

Passing data between named routes also requires a bit more thought. You cannot use any custom constructor because `WidgetBuilder` is already defined and locked in `MaterialApp`. Instead, you can use arguments to add anything you want to pass to the next screen. If you take a look at the definition of the `pushNamed` function, you'll see that the type for arguments is simply `Object`:

```
pushNamed(
  String routeName, {
  Object arguments,
})
```

While flexible, this ignores any type of safety that we might have gotten by using generics. The responsibility is now on the programmer to make sure the correct objects are sent to the route.

We retrieved the arguments with the **of-context** pattern to get the route associated with this widget:

```
String name = ModalRoute.of(context).settings.arguments;
```

This code is not safe in itself. There is no guarantee that the value that was passed into the arguments is a string or even exists at all. If the object that created this route decided to put an `integer` or a `Map` into the `arguments` property, then this line would throw an exception, causing the red error screen to take over your whole app. Because of this, you need to be especially careful when working with route arguments.

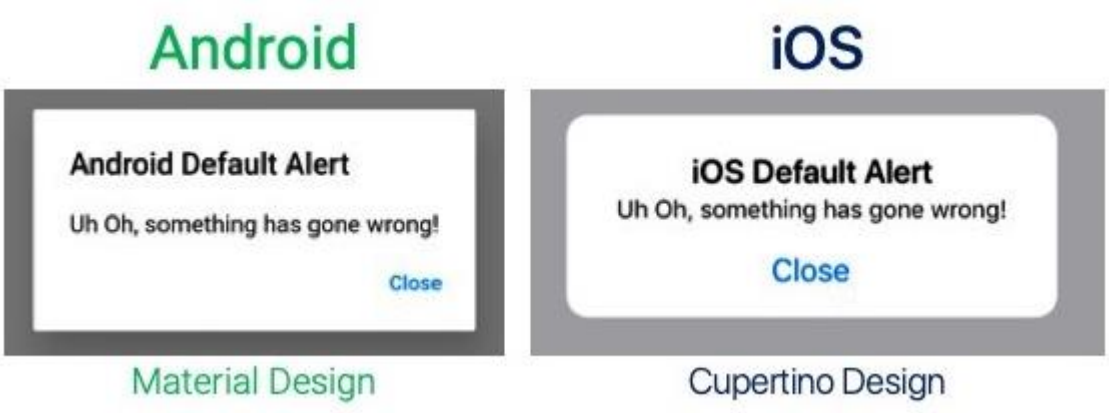
Passing arguments through named routes requires some effort, especially if you want to do so safely. For these reasons, it's recommended that you use manually constructed routes when you need to pass data back and forth between your screens.

Showing dialogs on the screen

Dialogs, or popups, are used when you want to give a message to your users that needs their attention. This ranges from telling the user about some error that occurred or asking them to perform some action before continuing, or even giving them a warning.

As alerts require some feedback from the user, you should use them for important information prompts or for actions that require immediate attention: in other words, **only when necessary.**

The following are the default alerts for Android and iOS:



In this recipe, we're going to create a platform-aware alert and use it to show a prompt when the user stops the stopwatch.

How to do it...

We will work with a new file in our project, called `platform_alert.dart`. Let's get started:

1. Open this new file and create a constructor that will accept a title and message body. This class is just going to be a simple dart object:

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

class PlatformAlert {
  final String title;
  final String message;

  const PlatformAlert({@required this.title, @required
    this.message})
      : assert(title != null),
        assert(message != null);
}
```

2. `PlatformAlert` is going to need a `show` method that will look at the app's context to determine what type of device it's running on and then show the appropriate dialog widget.
3. Add this method just after the constructor:

```
void show(BuildContext context) {
  final platform = Theme.of(context).platform;

  if (platform == TargetPlatform.iOS) {
    _buildCupertinoAlert(context);
  } else {
    _buildMaterialAlert(context);
  }
}
```

4. Showing an alert only requires invoking a global function called `showDialog`, which, just like `Navigator`, accepts a `WidgetBuilder` closure.

The `showDialog` method returns a `Future<T>`, meaning that it can return a value that you can deal with later. In the example that follows, we do not need to listen to the user response as we are only giving some information, so the return types of the methods will just be `void`.

5. Implement the `_buildMaterialAlert` method with the following code:

```
void _buildMaterialAlert(BuildContext context) {
  showDialog(
      context: context,
      builder: (context) {
        return AlertDialog(
            title: Text(title),
            content: Text(message),
            actions: [
              TextButton(
                  child: Text('Close'),
                  onPressed: () => Navigator.of(context).pop())
            ]);
      });
}
```

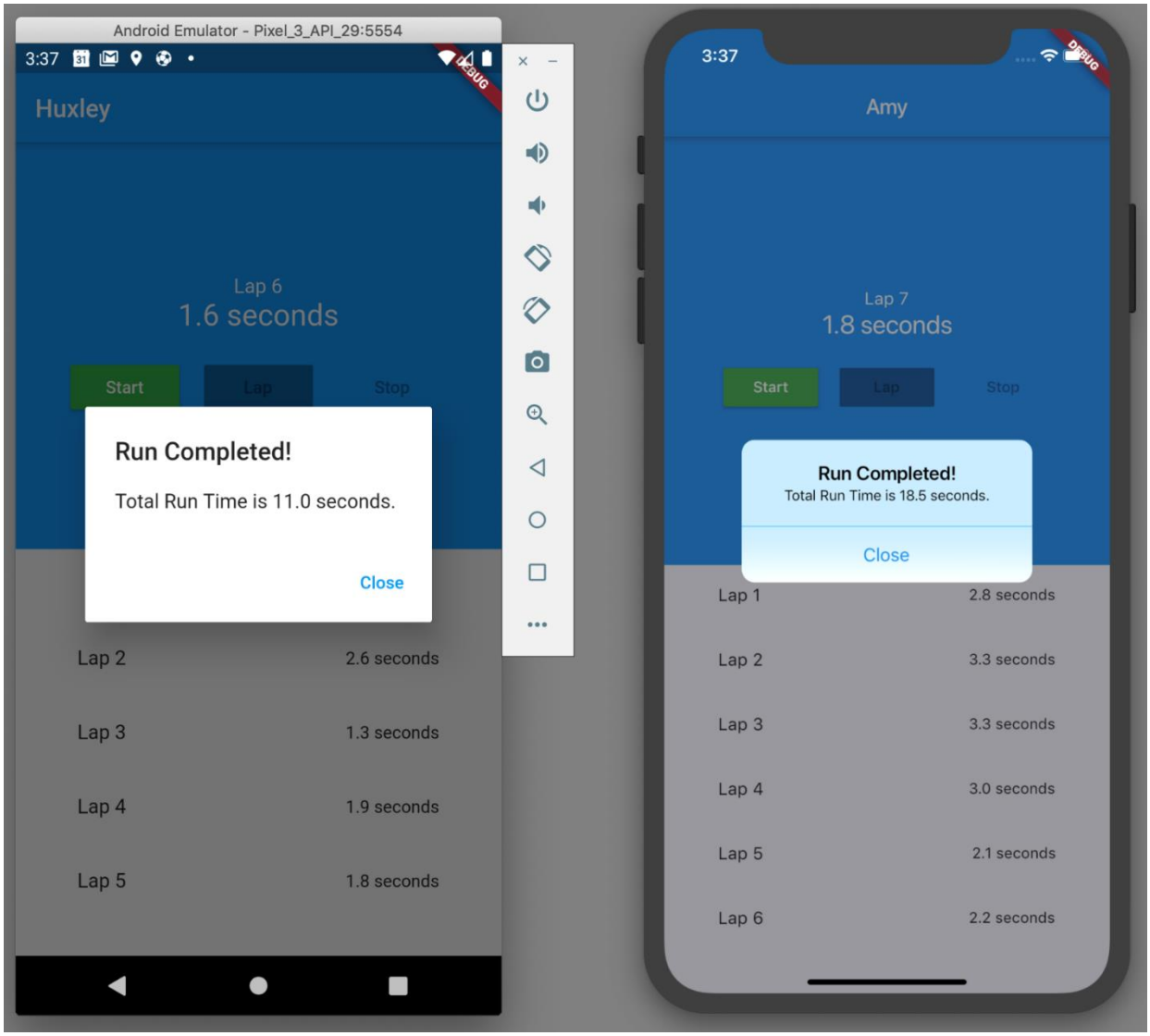
6. The iOS version is very similar – we just need to swap out the material components with their Cupertino counterparts. Add this method immediately after the material builder:

```
void _buildCupertinoAlert(BuildContext context) {
  showCupertinoDialog(
      context: context,
      builder: (context) {
        return CupertinoAlertDialog(
            title: Text(title),
            content: Text(message),
            actions: [
              CupertinoButton(
                  child: Text('Close'),
                  onPressed: () => Navigator.of(context).pop())
            ]);
      });
}
```

7. You now have a platform-aware class that wraps both `AlertDialog` and `CupertinoAlertDialog`. You can test this out in the `stopwatch.dart` file. Show a dialog when the user stops the stopwatch that shows the total time elapsed, including all the laps. Add the following code to the bottom of the `_stopTimer` method, after the call to `setState`:

```
final totalRuntime = laps.fold(milliseconds, (total, lap) => total + lap);
final alert = PlatformAlert(
  title: 'Run Completed!',
  message: 'Total Run Time is ${_secondsText(totalRuntime)}.',
);
alert.show(context);
```

Run the app and run a couple of laps. A `Dialog` will now show you the total of all the laps when you press stop. As an added bonus, try running the app on both the iOS simulator and Android emulator. Notice how the UI changes to respect the platform's standards, as shown here:



How it works...

The way Flutter handles dialogs is fascinating in terms of its simplicity. Dialogs are just routes. The only difference between a `MaterialPageRoute` and a `Dialog` is the animation that Flutter uses to display them. Since dialogs are just routes, they use the same `Navigator` API for pushing and popping. This is accomplished by calling the `showDialog` or `showCupertinoDialog` global function. Both of these functions will look for the app's `Navigator` and push a route onto the navigation stack using the platform-appropriate animation.

An alert, whether Material or Cupertino, is made up of three components:

- Title
- Content
- Actions

The `title` and `content` properties are just widgets. Typically, you would use a `Text` widget, but that's not required. If you want to put an input form and a scrolling list in a `Center` widget, you could certainly do that.

The actions are also *usually* a list of buttons, where users can perform an appropriate action. In this recipe, we used just one that will close the dialog:

```
TextButton(
  child: Text('Close'),
  onPressed: () => Navigator.of(context).pop())
```

Note that closing the dialog is just a standard call to the `Navigator` API. Since dialogs are routes, we can treat them identically. On Android, the system's back button will even pop the dialog from the stack, just as you would expect.

There's more...

In this recipe, we also used the app's theme to determine the host platform.
The `ThemeData` object has an enum called `TargetPlatform` that shows the potential options where Flutter can be hosted. In this recipe, we are only dealing with mobile platforms (iOS and Android), but currently, there are several more options in this enum, including desktop platforms. This is the current implementation of `TargetPlatform`:

```

  enum TargetPlatform {
  /// Android: <https://www.android.com/>
  android,

  /// Fuchsia: <https://fuchsia.dev/fuchsia-src/concepts>
  fuchsia,

  /// iOS: <https://www.apple.com/ios/>
  iOS,

  /// Linux: <https://www.linux.org>
  linux,

  /// macOS: <https://www.apple.com/macos>
  macOS,

  /// Windows: <https://www.windows.com>
  windows,
}
```

An interesting option here is `fuchsia`. Fuchsia is an experimental operating system that is currently under development at Google. It has been suggested that, at some point in the future, Fuchsia might replace Android. When (or if) that happens, the primary application layer for Fuchsia will be Flutter. So, congratulations – you are already covertly a Fuchsia developer! It's

still early days for this operating system and information is sparse, but this confirms that no matter what happens, the future of Flutter is bright.

Presenting bottom sheets

There are times where you need to present modal information, but a dialog just comes on too strong. Fortunately, there are quite a few alternative conventions for putting information on the screen that do not necessarily require action from users. Bottom sheets are one of the "gentler" alternatives to dialogs. With a bottom sheet, information slides out from the bottom of the screen, where it can be swiped down by the user if it displeases them. Also, unlike alerts, bottom sheets do not block the main interface, allowing the user to conveniently ignore this optional modal.

In this final recipe for the stopwatch app, we're going to replace the dialog alert with a bottom sheet and animate it away after 5 seconds.

How to do it...

Open `stopwatch.dart` to get started:

1. The bottom sheet API is not dramatically different from dialogs. The global function expects a `BuildContent` and a `WidgetBuilder`.
2. Let's create that builder as its own function.
3. Add the following code underneath the `_stopTimer` method:

```
Widget _buildRunCompleteSheet(BuildContext context) {
  final totalRuntime = laps.fold(milliseconds, (total, lap) => total + lap);
  final textTheme = Theme.of(context).textTheme;

  return SafeArea(
    child: Container(
      color: Theme.of(context).cardColor,
      width: double.infinity,
      child: Padding(
          padding: EdgeInsets.symmetric(vertical: 30.0),
          child: Column(mainAxisSize: MainAxisSize.min, children: [
            Text('Run Finished!', style: textTheme.headline6),
            Text('Total Run Time is
              ${_secondsText(totalRuntime)}.')
          ])),
    )
  );
}
```

4. Showing the bottom sheet should now be remarkably easy. In the `_stopTimer` method, delete the code that shows the dialog and replace it with an invocation to `showBottomSheet`:

```
void _stopTimer(BuildContext context) {
  setState(() {
    timer.cancel();
    isTicking = false;
  });

  showBottomSheet(context: context, builder:
   _buildRunCompleteSheet);
}
```

5. Try running the code now and tap the stop button to present the sheet. Did it work? You are probably seeing a lot of nothing right now. In actuality, you might even be seeing the following error being printed to the console:

```
flutter: ══╡ EXCEPTION CAUGHT BY GESTURE ╞═══════════════════════════════════════════
flutter: The following assertion was thrown while handling a gesture:
flutter: No Scaffold widget found.
flutter: StopWatch widgets require a Scaffold widget ancestor.
flutter: The Specific widget that could not find a Scaffold ancestor was:
flutter:   StopWatch
flutter: The ownership chain for the affected widget is:
flutter:   StopWatch ← Semantics ← Builder ← RepaintBoundary-[GlobalKey#a3f68] ← IgnorePointer ← Stack ←
flutter: _CupertinoBackGestureDetector<dynamic> ← DecoratedBox ← DecoratedBoxTransition ←
flutter: FractionalTranslation ← …
flutter: Typically, the Scaffold widget is introduced by the MaterialApp or WidgetsApp widget at the top of
flutter: your application widget tree.
flutter:
flutter: When the exception was thrown, this was the stack:
flutter: #0      debugCheckHasScaffold.<anonymous closure> (package:flutter/src/material/debug.dart:149:7)
flutter: #1      debugCheckHasScaffold (package:flutter/src/material/debug.dart:161:4)
flutter: #2      showBottomSheet (package:flutter/src/material/bottom_sheet.dart:481:10)
flutter: #3      StopWatchState._stopTimer (package:stopwatch/stopwatch.dart:142:9)
flutter: #4      StopWatchState._buildControls.<anonymous closure> (package:stopwatch/stopwatch.dart:101:40)
flutter: #5      _InkResponseState._handleTap (package:flutter/src/material/ink_well.dart:635:14)
flutter: #6      _InkResponseState.build.<anonymous closure> (package:flutter/src/material/ink_well.dart:711:32)
flutter: #7      GestureRecognizer.invokeCallback (package:flutter/src/gestures/recognizer.dart:182:24)
flutter: #8      TapGestureRecognizer._checkUp (package:flutter/src/gestures/tap.dart:365:11)
flutter: #9      TapGestureRecognizer.handlePrimaryPointer (package:flutter/src/gestures/tap.dart:275:7)
flutter: #10     PrimaryPointerGestureRecognizer.handleEvent (package:flutter/src/gestures/recognizer.dart:455:9)
flutter: #11     PointerRouter._dispatch (package:flutter/src/gestures/pointer_router.dart:75:13)
flutter: #12     PointerRouter.route (package:flutter/src/gestures/pointer_router.dart:102:11)
flutter: #13     _WidgetsFlutterBinding&BindingBase&GestureBinding.handleEvent (package:flutter/src/gestures/binding.dart:218:19)
flutter: #14     _WidgetsFlutterBinding&BindingBase&GestureBinding.dispatchEvent (package:flutter/src/gestures/binding.dart:198:22)
flutter: #15     _WidgetsFlutterBinding&BindingBase&GestureBinding._handlePointerEvent (package:flutter/src/gestures/binding.dart:156:7)
flutter: #16     _WidgetsFlutterBinding&BindingBase&GestureBinding._flushPointerEventQueue (package:flutter/src/gestures/binding.dart:102:7)
flutter: #17     _WidgetsFlutterBinding&BindingBase&GestureBinding._handlePointerDataPacket (package:flutter/src/gestures/binding.dart:86:7)
flutter: #21     _invoke1 (dart:ui/hooks.dart:250:10)
flutter: #22     _dispatchPointerDataPacket (dart:ui/hooks.dart:159:5)
flutter: (elided 3 frames from package dart:async)
flutter:
flutter: Handler: "onTap"
flutter: Recognizer:
flutter:   TapGestureRecognizer#f56db
flutter: ═══════════════════════════════════════════════════════════════════════════
```

6. Read this message carefully. This scary looking stack trace is saying that the context that we're using to present the bottom sheet requires a `Scaffold`, but it cannot find it. This is caused by using a `BuildContext` that is too high in the tree. We can fix this by wrapping the stop button with a `Builder` and passing that new context to the `stop` method. In `_buildControls`, replace the stop button with the following code:

```
Builder(
  builder: (context) => TextButton(
    child: Text('Stop'),
    onPressed: isTicking ? () => _stopTimer(context) : null,
 ...
```

We also have to update the `_stopTimer` method so that it accepts a `BuildContext` as a parameter:

```
void _stopTimer(BuildContext context) {...}
```

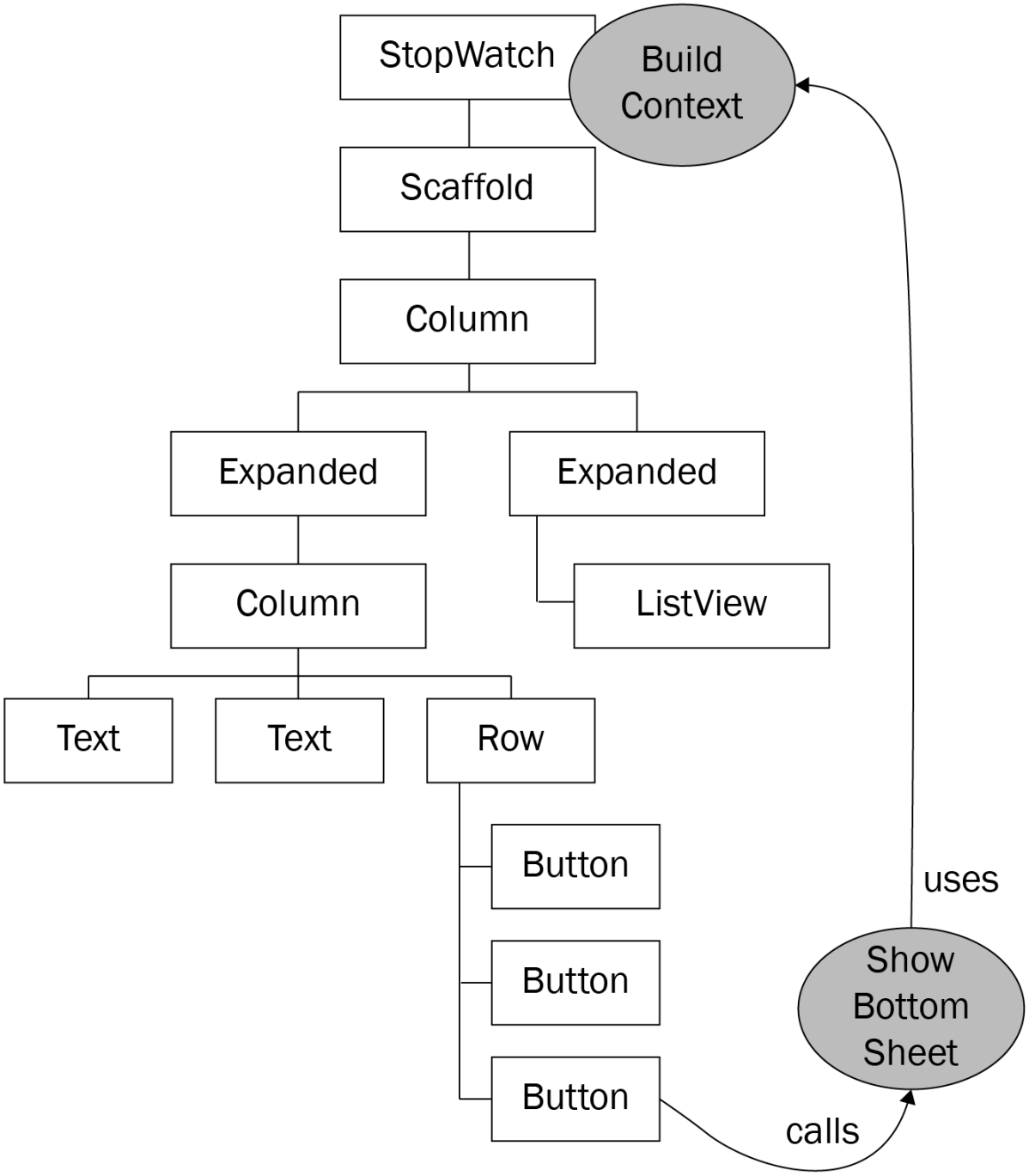
7. Hot reload the app and stop the timer. The bottom sheet now automatically appears after you hit the stop button. But it just stays there forever. It would be nice to have a short timer that automatically removes the bottom sheet after 5 seconds. We can accomplish this with the `Future` API. In the `_stopTimer` method, update the call so that it shows the bottom sheet, like so:

```
final controller =
    showBottomSheet(context: context, builder:
      _buildRunCompleteSheet);

Future.delayed(Duration(seconds: 5)).then((_) {
  controller.close();
});
```

Hot reload the app. The bottom sheet now politely excuses itself after 5 seconds.

How it works...

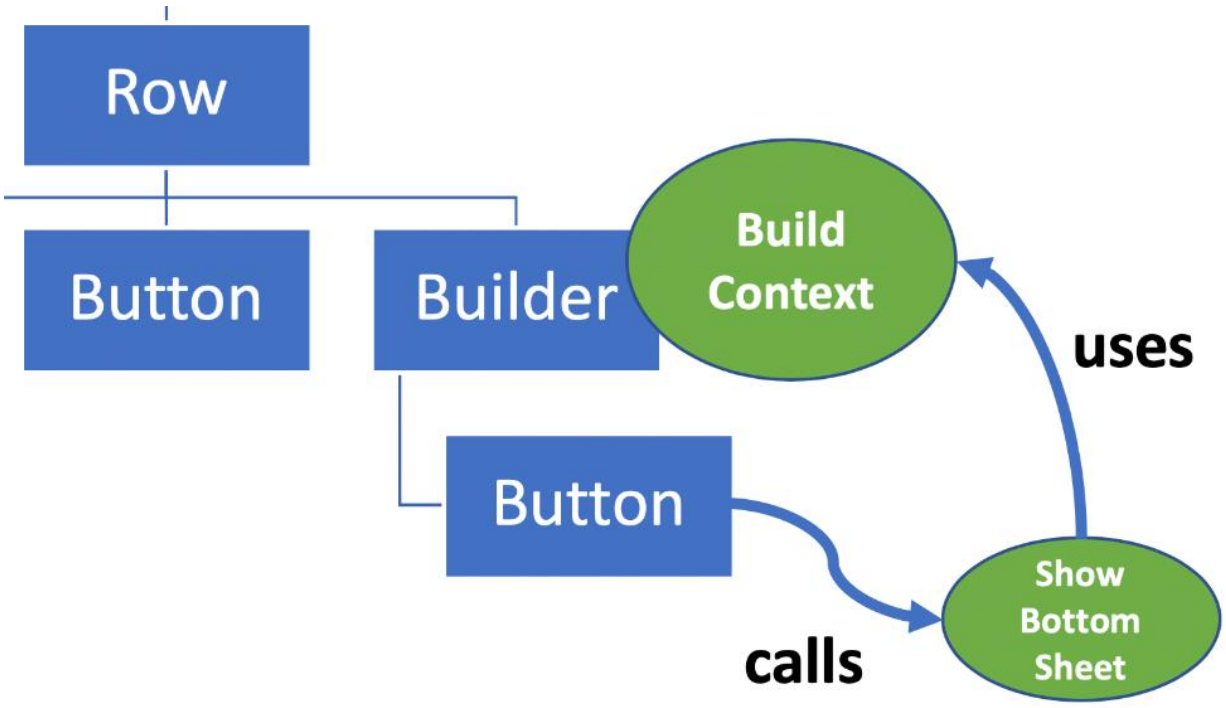
The bottom sheet part of this recipe should be pretty simple to understand, but what's going on with that error? Why did showing the bottom sheet initially fail? Take a look at how we organized the widget tree for the stopwatch screen:



Bottom sheets are a little different than Dialogs in that they are not full routes. For a bottom sheet to be presented, it attaches itself to the closest `Scaffold` in the tree using the same **of-context** pattern to find it. The problem is that the `BuildContent` class that we've been passing around and storing as a property on the `StopWatchState` class belongs to the top-level `StopWatch` widget. The `Scaffold` widget that we're using for this screen is a **child** of `StopWatch`, not a parent.

When we use `BuildContext` in the `showBottomSheet` function, it travels upward from that point to find the closest scaffold. The problem is that there aren't any scaffolds above `StopWatch`. It's only going to find a `MaterialApp` and our root widget. Consequently, the call fails.

The solution is to use a `BuildContext` that is **lower** in the tree so that it can find our `Scaffold`. This is where the `Builder` widget comes in handy. `Builder` is a widget that doesn't have a `child` or `children`, but a `WidgetBuilder`, just like routes and bottom sheets. By wrapping the button into a builder, we can grab a different `BuildContext`, one that is certainly a child of `Scaffold`, and use that to successfully show the bottom sheet:



This is one of the more interesting problems that you can come across when designing widget trees. It's important to keep the structure of the tree in mind when passing around the `BuildContext` class. Sometimes, the root context that you get from the widget's `build` method is not the context you are looking for.

The `buildBottomSheet` method also returns a `PersistentBottomSheetController`, which is just like a `ScrollController` or `TextEditingController`. These "controller" classes are attached to widgets and have methods to manipulate them. In this recipe, we used Dart's `Future` API to call the `close` method after a 5-second delay.