

Author Picks

FREE

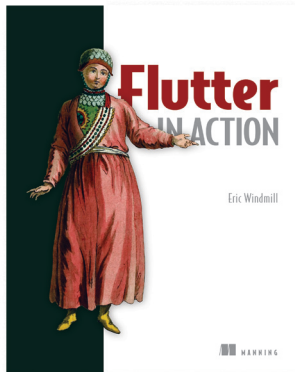


Exploring Cross-Platform Development with Flutter, React Native, and Xamarin

Chapters selected by Eric Windmill

 manning

Save 50% on these books and videos – eBook, pBook, and MEAP. Enter **mescdo50** in the Promotional Code box when you checkout. Only at manning.com.



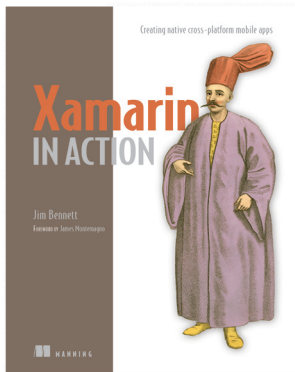
Flutter in Action

by Eric Windmill

ISBN 9781617296147

310 pages

\$39.99



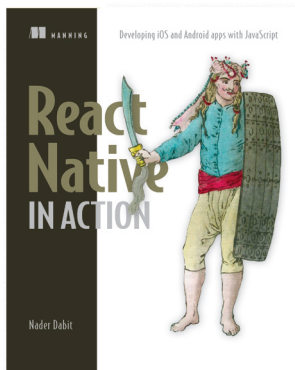
Xamarin in Action

by Jim Bennett

ISBN 9781617294389

608 pages

\$43.99



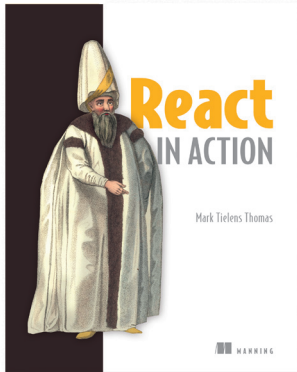
React Native in Action

by Nader Dabit

ISBN 9781617294051

320 pages

\$39.99



React in Action

by Mark Tielens Thomas

ISBN 9781617293856

360 pages

\$35.99



React in Motion

by Zac Braddy

Course duration: 3h 51m

\$59.99



*Exploring Cross-Platform Development with Flutter,
React Native, and Xamarin*

Chapters chosen by Eric Windmill

Manning Author Picks

Copyright 2019 Manning Publications
To pre-order or learn more about these books go to www.manning.com

Licensed to Vincent VAUBAN <vvauban@gmail.com>

For online information and ordering of these and other Manning books, please visit www.manning.com. The publisher offers discounts on these books when ordered in quantity.

For more information, please contact


Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: Erin.Twohey@manning.com

©2019 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

- ⊗ Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.

 Manning Publications Co.
20 Baldwin Road Technical
PO Box 761
Shelter Island, NY 11964

Cover designer: Leslie Haimes

ISBN: 9781617296789

contents

introduction iv

Building your first React Native app 2

Chapter 3 from *React Native in Action*

Hello MVVM—creating a simple cross-platform app using MVVM 31

Chapter 2 from *Xamarin in Action*

Flutter UI: Important widgets, theme, and layout 57

Chapter 4 from *Flutter in Action*

index 87

introduction

Technology moves fast. Programming technology is constantly changing, which is a good thing because we can create higher-quality products at a faster rate every day. The tools that make this possible—SDKs, frameworks, and developer tools—are essentially about abstracting away hard problems that exist for every app. In the world of mobile app development, which is a fairly new field, the next set of tools that makes our lives easier is *cross-platform SDKs*.

These days, cross-platform options are becoming widely accepted, viable options for all mobile apps. Historically, mobile developers have been extremely skeptical of cross-platform tools, and for good reason. Native development (in iOS and Android) is a pain compared with using cross-platform tools, but the apps have always been much higher-quality. A few newer tools are finally challenging that situation—notably, Flutter, React Native, and Xamarin.

All three of these options are worth considering because they all have different advantages. React Native’s biggest advantage is that it’s written in JavaScript and ReactJS. Web developers can jump in without missing a beat. Xamarin is the most established and therefore likely has the most engaged community. Flutter is the newest tool. Its main advantage is that it’s written in Dart but compiles completely to native mobile code, so it’s as performant as native apps.

If you’re new to mobile and considering jumping in, you should consider all your options. The following chapter excerpts give you a brief intro to each of these platforms. I’ve made my choice: I think that Flutter is the best development experience that exists right now. But don’t take my word for it. The best way to figure out which one you like is to give each a try!

In this chapter, you'll jump right into building a simple to-do app in React, learning React concepts as you go.

Building your first React Native app

This chapter covers

- Building a todo app from the ground up
- Light debugging

When learning a new framework, technology, language, or concept, diving directly into the process by building a real app is a great way to jump-start the learning process. Now that you understand the basics of how React and React Native work, let's put these pieces together to make your first app: a todo app. Going through the process of building a small app and using the information we've gone over so far will be a good way to reinforce your understanding of how to use React Native.

You'll use some functionality in the app that we haven't yet covered in depth, and some styling nuances we've yet to discuss, but don't worry. Instead of going over these new ideas one by one now, you'll build the basic app and then learn about these concepts in detail in later chapters. Take this opportunity to play around with the app as you build it to learn as much as possible in the process: feel free to break and fix styles and components to see what happens.

3.1 Laying out the todo app

Let's get started building the todo app. It will be similar in style and functionality to the apps on the TodoMVC site (<http://todomvc.com>). Figure 3.1 shows how the app will look when you're finished, so you can conceptualize what components you need and how to structure them. As in chapter 1, figure 3.2 breaks the app into components and container components. Let's see how this will look in the app using a basic implementation of React Native components.

Listing 3.1 Basic todo app implementation

```
<View>
  <Heading />
  <Input />
  <TodoList />
  <Button />
  <TabBar />
</View>
```

The app will display a heading, a text input, a button, and a tab bar. When you add a todo, the app will add it to the array of todos and display the new todo beneath the input. Each todo will have two buttons: Done and Delete. The Done button will mark it as complete, and the Delete button will remove it from the array of todos. At the bottom of the screen, the tab bar will filter the todos based on whether they're complete or still active.

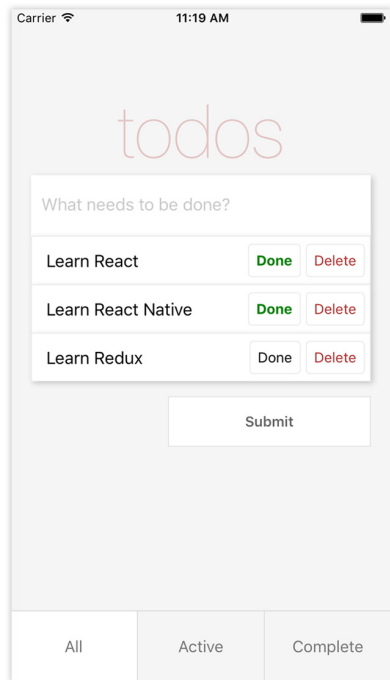


Figure 3.1 Todo app design

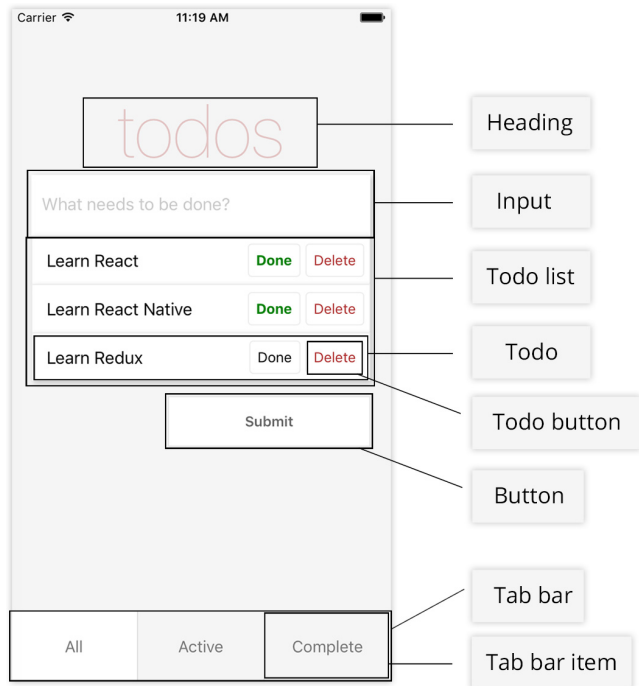


Figure 3.2 Todo app with descriptions

3.2 Coding the todo app

Let's get started coding the app. Create a new React Native project by typing `react-native init TodoApp` in your terminal (see figure 3.3). Now, go into your index file: if you're developing for iOS, open `index.iOS.js`; and if you're developing for Android, open `index.Android.js`. The code for both platforms will be the same.

NOTE I'm using React Native version 0.51.0 for this example. Newer versions may have API changes, but nothing should be broken for building the todo app. You're welcome to use the most recent version of React Native, but if you run into issues, use the exact version I'm using here.

In the index file, import an `App` component (which you'll create soon), and delete the styling along with any extra components you're no longer using.

Listing 3.2 index.js

```
import React from 'react'
import { AppRegistry } from 'react-native'
import App from './app/App'

const TodoApp = () => <App />

AppRegistry.registerComponent('TodoApp', () => TodoApp)
```

Here, you bring in `AppRegistry` from `react-native`. You also bring in the main `App` component, which you'll create next.

In the `AppRegistry` method, you initiate the application. `AppRegistry` is the JS entry point to running all React Native apps. It takes two arguments: the `appKey`, or the name of the application you defined when you initialized the app; and a function that returns the React Native component you want to use as the entry point of the app. In this case, you're returning the `TodoApp` component declared in listing 3.2.

Now, create a folder called `app` in the root of the application. In the `app` folder, create a file called `App.js` and add the basic code shown in the next listing.

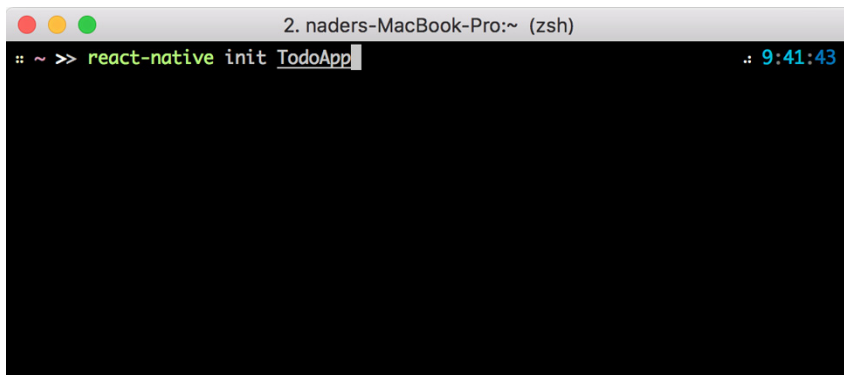


Figure 3.3 Initializing a new React Native app

Listing 3.3 Creating the App component

```
import React, { Component } from 'react'
import { View, ScrollView, StyleSheet } from 'react-native'

class App extends Component {
  render() {
    return (
      <View style={styles.container}>
        <ScrollView keyboardShouldPersistTaps='always'
          style={styles.content}>
          <View/>
        </ScrollView>
      </View>
    )
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#f5f5f5'
  },
  content: {
    flex: 1,
    paddingTop: 60
  }
})

export default App
```

You import a new component called `ScrollView`, which wraps the platform `ScrollView` and is basically a scrollable `View` component. A `keyboardShouldPersistTaps` prop of `always` is added: this prop will dismiss the keyboard if it's open and allow the UI to process any `onPress` events. You make sure both the `ScrollView` and the parent `View` of the `ScrollView` have a `flex:1` value. `flex:1` is a style value that makes the component fill the entire space of its parent container.

Now, set up an initial state for some of the values you'll need later. You need an array to keep your todos, which you'll name `todos`; a value to hold the current state of the `TextInput` that will add the todos, named `inputValue`; and a value to store the type of todo that you're currently viewing (All, Current, or Active), named `type`.

In `App.js`, before the render function, add a constructor and an initial state to the class, and initialize these values in the state.

Listing 3.4 Setting the initial state

```
...

class App extends Component {
  constructor() {
    super()
    this.state = {
```



```

        inputValue: '',
        todos: [],
        type: 'All'
      }
    }
    render() {
      ...
    }
  }
}

...

```

Next, create the `Heading` component and give it some styling. In the `app` folder, create a file called `Heading.js`. This will be a stateless component.

Listing 3.5 Creating the `Heading` component

```

import React from 'react'
import { View, Text, StyleSheet } from 'react-native'

const Heading = () => (
  <View style={styles.header}>
    <Text style={styles.headerText}>
      todos
    </Text>
  </View>
)

const styles = StyleSheet.create({
  header: {
    marginTop: 80
  },
  headerText: {
    textAlign: 'center',
    fontSize: 72,
    color: 'rgba(175, 47, 47, 0.25)',
    fontWeight: '100'
  }
})

export default Heading

```

Note that in the styling of `headerText`, you pass an `rgba` value to `color`. If you aren't familiar with `RGBA`, the first three values make up the `RGB` color values, and the last value represents the `alpha` or `opacity` (red, blue, green, alpha). You pass in an `alpha` value of `0.25`, or `25%`. You also set the `font weight` to `100`, which will give the text a thinner weight and look.

Go back into `App.js`, bring in the `Heading` component, and place it in the `ScrollView`, replacing the empty `View` you originally placed there.

Run the app to see the new heading and app layout: see figure 3.4. To run the app in iOS, use `react-native run-ios`. To run in Android, use `react-native run-android` in your terminal from the root of your React Native application.

Listing 3.6 Importing and using the `Heading` component

```
import React, { Component } from 'react'
import { View, ScrollView, StyleSheet } from 'react-native'
import Heading from './Heading'

class App extends Component {
  ...
  render() {
    return (
      <View style={styles.container}>
        <ScrollView
          keyboardShouldPersistTaps='always'
          style={styles.content}>
          <Heading />
        </ScrollView>
      </View>
    )
  }
}
```

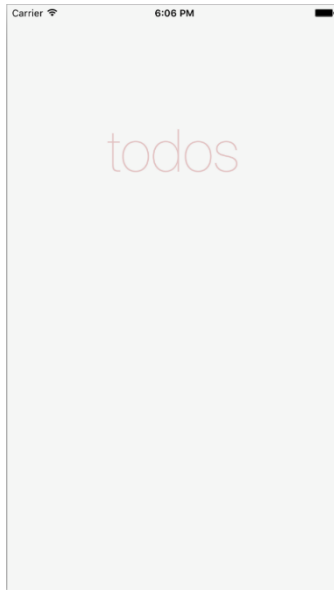


Figure 3.4 Running the app

Next, create the `TextInput` component and give it some styling. In the app folder, create a file called `Input.js`.

Listing 3.7 Creating the `TextInput` component

```
import React from 'react'
import { View, TextInput, StyleSheet } from 'react-native'

const Input = () => (
  <View style={styles.inputContainer}>
    <TextInput
      style={styles.input}
      placeholder='What needs to be done?'
      placeholderTextColor='#CACACA'
      selectionColor='#666666' />
    </View>
  )

const styles = StyleSheet.create({
  inputContainer: {
    marginLeft: 20,
    marginRight: 20,
    shadowOpacity: 0.2,
    shadowRadius: 3,
    shadowColor: '#000000',
    shadowOffset: { width: 2, height: 2 }
  },
  input: {
    height: 60,
    backgroundColor: '#ffffff',
    paddingLeft: 10,
    paddingRight: 10
  }
})

export default Input
```

You're using a new React Native component called `TextInput` here. If you're familiar with web development, this is similar to an HTML input. You also give both the `TextInput` and the outer `View` their own styling.

`TextInput` takes a few other props. Here, you specify a placeholder to show text before the user starts to type, a `placeholderTextColor` that styles the placeholder text, and a `selectionColor` that styles the cursor for the `TextInput`.

The next step, in section 3.4, will be to wire up a function to get the value of the `TextInput` and save it to the state of the `App` component. You'll also go into `App.js` and add a new function called `inputChange` below the constructor and above the render function. This function will update the state value of `inputValue` with the value passed in, and for now will also log out the value of `inputValue` for you to make sure the function is working by using `console.log()`. But to view `console.log()` statements in React Native, you first need to open the developer menu. Let's see how it works.

3.3 Opening the developer menu

The developer menu is a built-in menu available as a part of React Native; it gives you access to the main debugging tools you'll use. You can open it in the iOS simulator or in the Android emulator. In this section, I'll show you how to open and use the developer menu on both platforms.

NOTE If you aren't interested in the developer menu or want to skip this section for now, go to section 3.4 to continue building the todo app.

3.3.1 Opening the developer menu in the iOS simulator

While the project is running in the iOS simulator, you can open the developer menu in one of three ways:

- Press Cmd-D on the keyboard.
- Press Cmd-Ctrl-Z on the keyboard.
- Open the Hardware > Shake Gesture menu in the simulator options (see figure 3.5).

When you do, you should see the developer menu, shown in figure 3.6.

NOTE If Cmd-D or Cmd-Ctrl-Z doesn't open the menu, you may need to connect your hardware to the keyboard. To do this, go to Hardware > Keyboard > Connect Hardware Keyboard in your simulator menu.

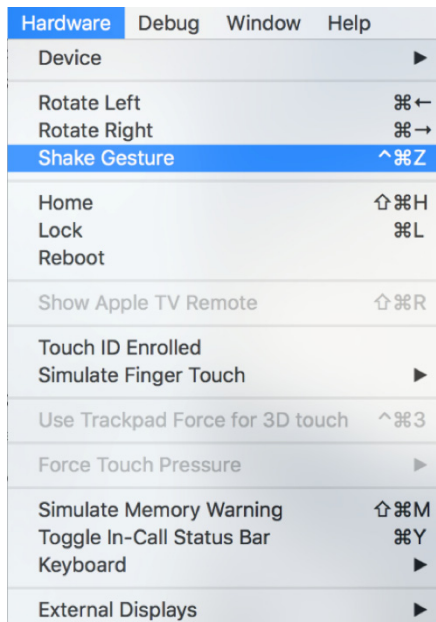


Figure 3.5 Manually opening the developer menu (iOS simulator)

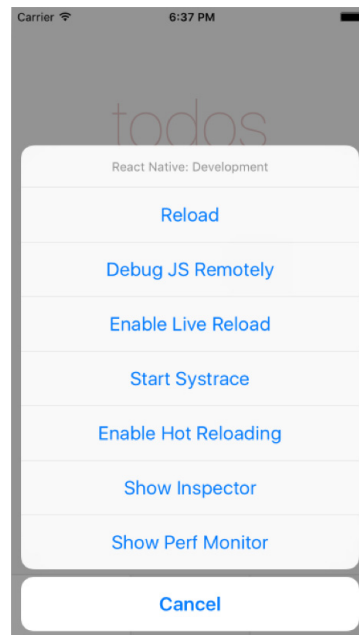


Figure 3.6 React Native developer menu (iOS simulator)

3.3.2 Opening the developer menu in the Android emulator

With the project open and running in the Android emulator, the developer menu can be opened in one of three ways:

- Press F2 on the keyboard.
- Press Cmd-M on the keyboard.
- Press the Hardware button (see figure 3.7).

When you do, you should see the developer menu shown in figure 3.8.

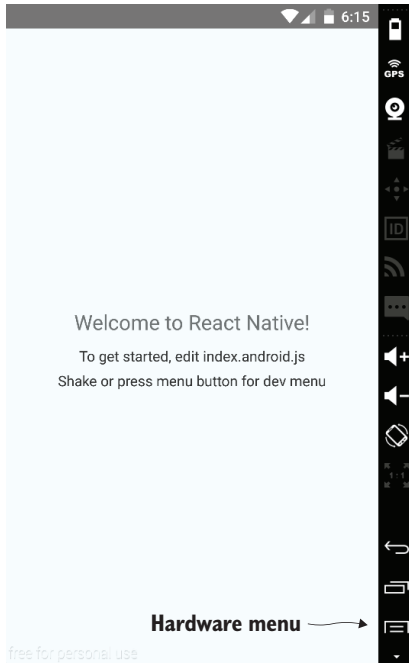


Figure 3.7 Manually opening the hardware menu (Android emulator)

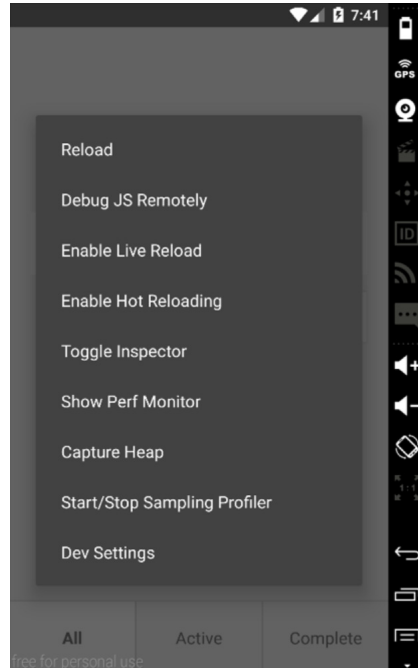


Figure 3.8 React Native developer menu (Android emulator)

3.3.3 Using the developer menu

When the developer menu opens, you should see the following options:

- *Reload (iOS and Android)*—Reloads the app. This can also be done by pressing Cmd-R on the keyboard (iOS) or pressing R twice (Android).
- *Debug JS Remotely (iOS and Android)*—Opens the Chrome dev tools and gives you full debugging support through the browser (figure 3.9). Here, you have access not only to logging statements in your code, but also to breakpoints and whatever you're used to while debugging web apps (with the exception of the DOM). If you need to log any information or data in your app, this is usually the place to do so.

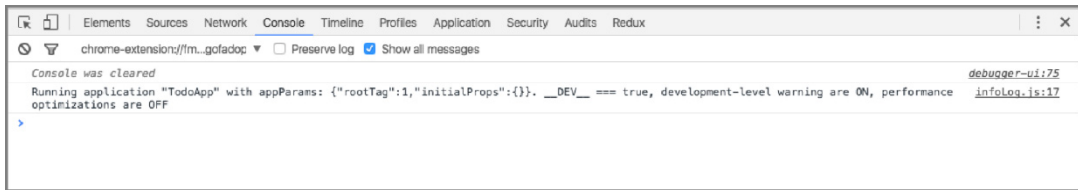


Figure 3.9 Debugging in Chrome

- *Enable Live Reload (iOS and Android)*—Enables live reload. When you make changes in your code, the entire app will reload and refresh in the simulator.
- *Start Systrace (iOS only)*—Systrace is a profiling tool. This will give you a good idea of where your time is being spent during each 16 ms frame while your app is running. Profiled code blocks are surrounded by start/end markers that are then visualized in a colorful chart format. Systrace can also be enabled manually from the command line in Android. If you want to learn more, check out the docs for a very comprehensive overview.
- *Enable Hot Reloading (iOS and Android)*—A great feature added in version .22 of React Native. It offers an amazing developer experience, giving you the ability to see your changes immediately as files are changed without losing the current state of the app. This is especially useful for making UI changes deep in your app without losing state. It's different than live reloading because it retains the current state of your app, only updating the components and state that have been changed (live reloading reloads the entire app, therefore losing the current state).
- *Toggle Inspector (iOS and Android)*—Brings up a property inspector similar to what you see in the Chrome dev tools. You can click an element and see where it is in the hierarchy of components, as well as any styling applied to the element (figure 3.10).



Figure 3.10 Using the inspector (left: iOS, right: Android)

- *Show Perf Monitor (iOS and Android)*—Brings up a small box in the upper-left corner of the app, giving some information about the app's performance. Here you'll see the amount of RAM being used and the number of frames per second at which the app is currently running. If you click the box, it will expand to show even more information (figure 3.11).
- *Dev Settings (Android emulator only)*—Brings up additional debugging options, including an easy way to toggle between the `__DEV__` environment variable being true or false (figure 3.12).



Figure 3.11 Perf Monitor (left: iOS, right: Android)

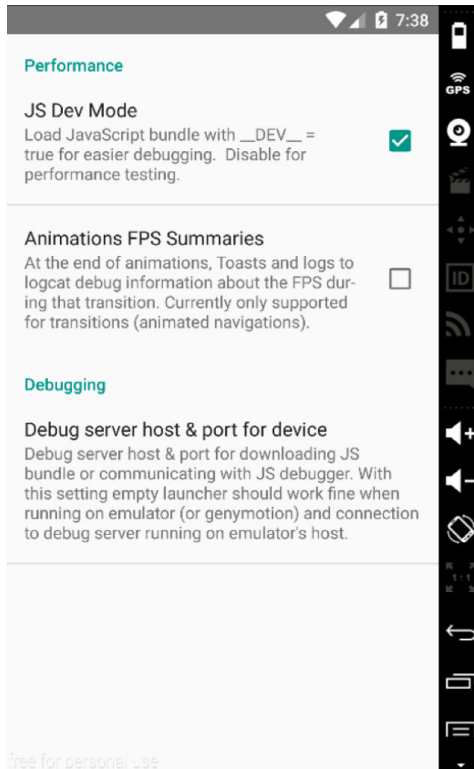


Figure 3.12 Dev Settings (Android emulator)

3.4 *Continuing building the todo app*

Now that you know how the developer menu works, open it and press Debug JS Remotely to open the Chrome dev tools. You're ready to start logging information to the JavaScript console.

You'll import the `Input` component into `app/App.js` and attach a method to `TextInput`, which you'll give as a prop to the `Input`. You'll also pass the `inputValue` stored on the state to `Input` as a prop.

Listing 3.8 Creating the `inputChange` function

```
...
import Heading from './Heading'
```

```

import Input from './Input'
class App extends Component {
  constructor() {
    ...
  }
  inputChange(inputValue) {
    console.log(' Input Value: ', inputValue)
    this.setState({ inputValue })
  }
  render() {
    const { inputValue } = this.state
    return (
      <View style={styles.container}>
        <ScrollView
          keyboardShouldPersistTaps='always'
          style={styles.content}>
          <Heading />
          <Input
            inputValue={inputValue}
            inputChange={(text) => this.inputChange(text)} />
        </ScrollView>
      </View>
    )
  }
}

```

Creates the inputChange method, which takes inputValue as an argument

Logs out the inputValue value to make sure the method is working

Sets the state with the new value—same as this.setState({inputValue: inputValue})

Passes inputValue as a property to the Input component

Passes inputChange as a property to the Input component

inputChange takes one argument, the value of the TextInput, and updates the inputValue in the state with the returned value from the TextInput.

Now, you need to wire up the function with the TextInput in the Input component. Open app/Input.js, and update the TextInput component with the new inputChange function and the inputValue property.

Listing 3.9 Adding inputChange and inputValue to the TextInput

```

...
const Input = ({ inputValue, inputChange }) => (
  <View style={styles.inputContainer}>
    <TextInput
      value={inputValue}
      style={styles.input}
      placeholder='What needs to be done?'
      placeholderTextColor='#CACACA'
      selectionColor='#666666'
      onChangeText={inputChange} />
  </View>
)
...

```

Deconstructs the inputValue and inputChange props

Sets the onChangeText method to inputChange

You destructure the props inputValue and inputChange in the creation of the stateless component. When the value of the TextInput changes, the inputChange function is called, and the value is passed to the parent component to set the state of inputValue. You also set the value of the TextInput to be inputValue, so you can later control and reset the TextInput. onChangeText is a method that will be called every time the value of the TextInput component is changed and will be passed the value of the TextInput.

Run the project again and see how it looks (figure 3.13). You're logging the value of the input, so as you type you should see the value being logged out to the console (figure 3.14).

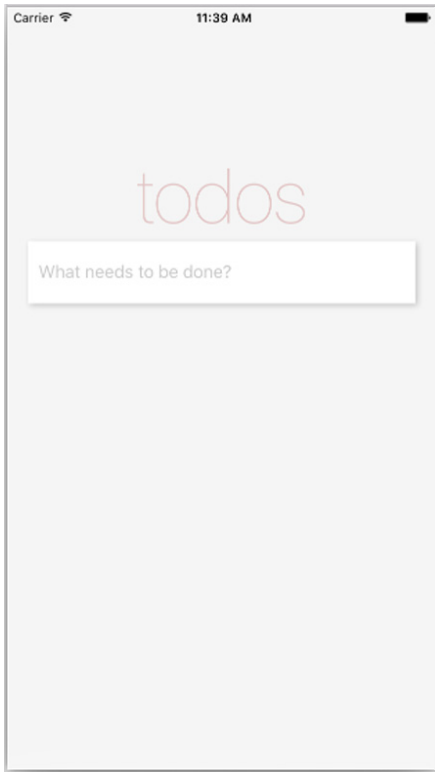


Figure 3.13 Updated view after adding the `TextInput`

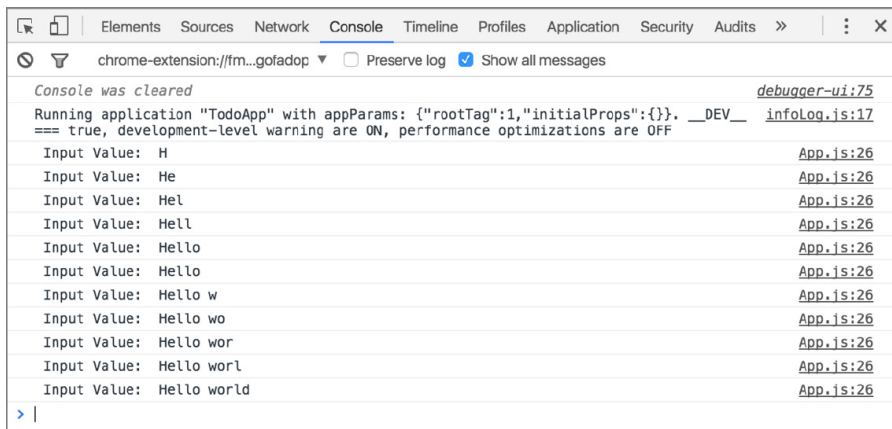


Figure 3.14 Logging out the `TextInput` value with the `onChange` method

Now that the value of the `inputValue` is being stored in the state, you need to create a button to add the items to a list of todos. Before you do, create a function that you'll bind to the button to add the new todo to the array of todos defined in the constructor. Call this function `submitTodo`, and place it after the `inputChange` function and before the render function.

Listing 3.10 Adding the `submitTodo` function

If `inputValue` isn't empty, create a variable an object with a `complete` Boolean (you'll c

```
...
submitTodo () {
  if (this.state.inputValue.match(/^\s*$/)) {
    return
  }
  const todo = {
    title: this.state.inputValue,
    todoIndex,
    complete: false
  }
  todoIndex++
  const todos = [...this.state.todos, todo]
  this.setState({ todos, inputValue: '' }, () => {
    console.log('State: ', this.state)
  })
}
```

Checks whether `inputValue` is empty or only contains whitespace. If it's empty, returns without doing anything else.

Increments the `todoIndex`

Pushes the new todo to the existing array of todos

Sets the state of the todos to match the updated array of `this.state.todos`, and resets `inputValue` to an empty string

Once the state is set, you have the option to pass a callback function. Here, a callback function from `setState` logs out the state to make sure everything is working.

Next, create the `todoIndex` at the top of the `App.js` file, below the last import statement.

Listing 3.11 Creating the `todoIndex` variable

```
...
import Input from './Input'

let todoIndex = 0

class App extends Component {
  ...
```

Now that the `submitTodo` function has been created, create a file called `Button.js` and wire up the function to work with the button.

Listing 3.12 Creating the `Button` component

```
import React from 'react'
import { View, Text, StyleSheet, TouchableHighlight } from 'react-native'
```

```

const Button = ({ submitTodo }) => (
  <View style={styles.buttonContainer}>
    <TouchableHighlight
      underlayColor='#efefef'
      style={styles.button}
      onPress={submitTodo}>
      <Text style={styles.submit}>
        Submit
      </Text>
    </TouchableHighlight>
  </View>
)

const styles = StyleSheet.create({
  buttonContainer: {
    alignItems: 'flex-end'
  },
  button: {
    height: 50,
    paddingLeft: 20,
    paddingRight: 20,
    backgroundColor: '#ffffff',
    width: 200,
    marginRight: 20,
    marginTop: 15,
    borderWidth: 1,
    borderColor: 'rgba(0,0,0,.1)',
    justifyContent: 'center',
    alignItems: 'center'
  },
  submit: {
    color: '#666666',
    fontWeight: '600'
  }
})

export default Button

```

Destructures the submitTodo function, which was passed as a prop to the component

Attaches submitTodo to the onPress function available to the TouchableHighlight component. This function will be called when the TouchableHighlight is touched or pressed.

In this component, you use `TouchableHighlight` for the first time. `TouchableHighlight` is one of the ways you can create buttons in React Native and is fundamentally comparable to the HTML button element.

With `TouchableHighlight`, you can wrap views and make them respond properly to touch events. On press down, the default `backgroundColor` is replaced with a specified `underlayColor` property that you'll provide as a prop. Here you specify an `underlayColor` of `'#efefef'`, which is a light gray; the background color is white. This will give the user a good sense of whether the touch event has registered. If no `underlayColor` is defined, it defaults to black.

`TouchableHighlight` supports only one main child component. Here, you pass in a `Text` component. If you want multiple components in a `TouchableHighlight`, wrap them in a single `View`, and pass this `View` as the child of the `TouchableHighlight`.

NOTE There's also quite a bit of styling going on in listing 3.12. Don't worry about styling specifics in this chapter: we cover them in depth in chapters 4 and 5. But do look at them, to get an idea how styling works in each component. This will help a lot in the in-depth later chapters, because you'll already have been exposed to some styling properties and how they work.

You've created the Button component and wired it up with the function defined in App.js. Now bring this component into the app (app/App.js) and see if it works!

Listing 3.13 Importing the Button component

```
...
import Button from './Button'
let todoIndex = 0

...
constructor() {
  super()
  this.state = {
    inputValue: '',
    todos: [],
    type: 'All'
  }
  this.submitTodo = this.submitTodo.bind(this)
}
...
render () {
  let { inputValue } = this.state
  return (
    <View style={styles.container}>
      <ScrollView
        keyboardShouldPersistTaps='always'
        style={styles.content}>
        <Heading />
        <Input
          inputValue={inputValue}
          inputChange={ (text) => this.inputChange(text) } />
        <Button submitTodo={this.submitTodo} />
      </ScrollView>
    </View>
  )
}
```

Imports the new Button component

Binds the method to the class in the constructor. Because you're using classes, functions won't be auto-bound to the class.

Place the Button below the Input component, and pass in submitTodo as a prop.

You import the Button component and place it under the Input component in the render function. submitTodo is passed in to the Button as a property called this.submitTodo.

Now, refresh the app. It should look like figure 3.15. When you add a todo, the TextInput should clear, and the app state should log to the console, showing an array of todos with the new todo in the array (figure 3.16).

Now that you're adding todos to the array of todos, you need to render them to the screen. To get started with this, you need to create two new components: `TodoList` and `Todo`. `TodoList` will render the list of `Todos` and will use the `Todo` component for each individual todo. Begin by creating a file named `Todo.js` in the app folder.

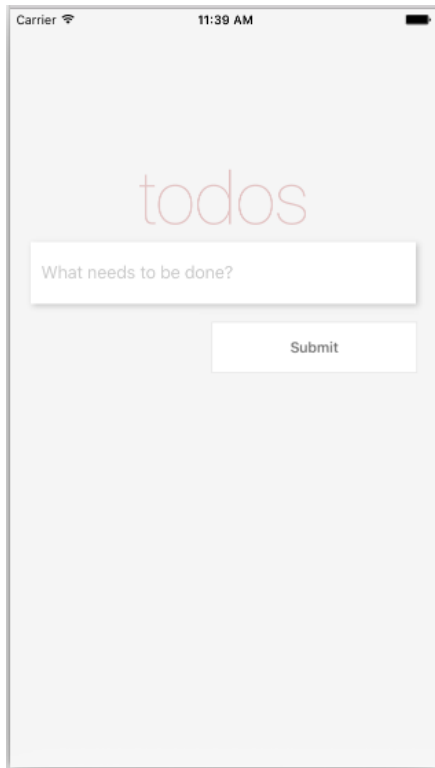


Figure 3.15 Updated app with the `Button` component

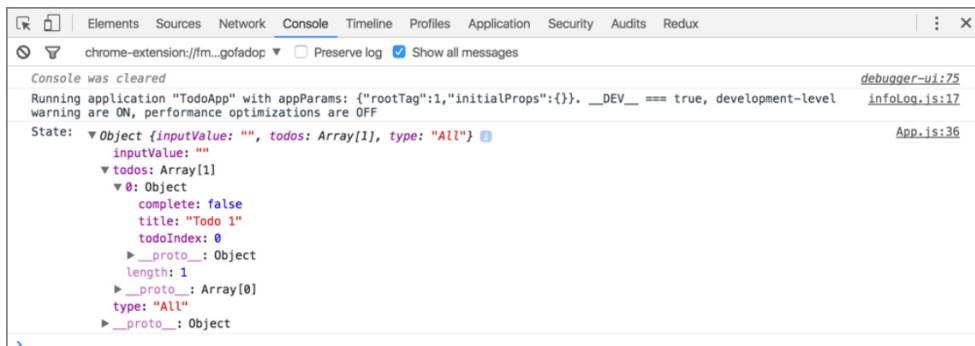


Figure 3.16 Logging the state

Listing 3.14 Creating the Todo component

```
import React from 'react'
import { View, Text, StyleSheet } from 'react-native'

const Todo = ({ todo }) => (
  <View style={styles.todoContainer}>
    <Text style={styles.todoText}>
      {todo.title}
    </Text>
  </View>
)

const styles = StyleSheet.create({
  todoContainer: {
    marginLeft: 20,
    marginRight: 20,
    backgroundColor: 'ffffff',
    borderTopWidth: 1,
    borderRightWidth: 1,
    borderLeftWidth: 1,
    borderColor: 'ededed',
    paddingLeft: 14,
    paddingTop: 7,
    paddingBottom: 7,
    shadowOpacity: 0.2,
    shadowRadius: 3,
    shadowColor: '000000',
    shadowOffset: { width: 2, height: 2 },
    flexDirection: 'row',
    alignItems: 'center'
  },
  todoText: {
    fontSize: 17
  }
})

export default Todo
```

The Todo component takes one property for now—a todo—and renders the title in a Text component. You also add styling to the View and Text components.

Next, create the TodoList component (app/TodoList.js).

Listing 3.15 Creating the TodoList component

```
import React from 'react'
import { View } from 'react-native'
import Todo from './Todo'

const TodoList = ({ todos }) => {
  todos = todos.map((todo, i) => {
    return (
      <Todo
        key={todo.todoIndex}
        todo={todo} />
    )
  })
}
```

```

    )
  })
  return (
    <View>
      {todos}
    </View>
  )
}

export default TodoList

```

The `TodoList` component takes one property for now: an array of todos. You then map over these todos and create a new `Todo` component (imported at the top of the file) for each todo, passing in the todo as a property to the `Todo` component. You also specify a key and pass in the index of the todo item as a key to each component. The key property helps React identify the items that have changed when the diff with the virtual DOM is computed. React will give you a warning if you leave this out.

The last thing you need to do is import the `TodoList` component into the `App.js` file and pass in the todos as a property.

Listing 3.16 Importing the `TodoList` component

```

...
import TodoList from './TodoList'
...
render () {
  const { inputValue, todos } = this.state
  return (
    <View style={styles.container}>
      <ScrollView
        keyboardShouldPersistTaps='always'
        style={styles.content}>
        <Heading />
        <Input inputValue={inputValue} inputChange={(text) =>
this.inputChange(text)} />
        <TodoList todos={todos} />
        <Button submitTodo={this.submitTodo} />
      </ScrollView>
    </View>
  )
}
...

```

Run the app. When you add a todo, you should see it pop up in the list of todos (figure 3.17).

The next steps are to mark a todo as complete, and to delete a todo. Open `App.js`, and create `toggleComplete` and `deleteTodo` functions below the `submitTodo` function. `toggleComplete` will toggle whether the todo is complete, and `deleteTodo` will delete the todo.

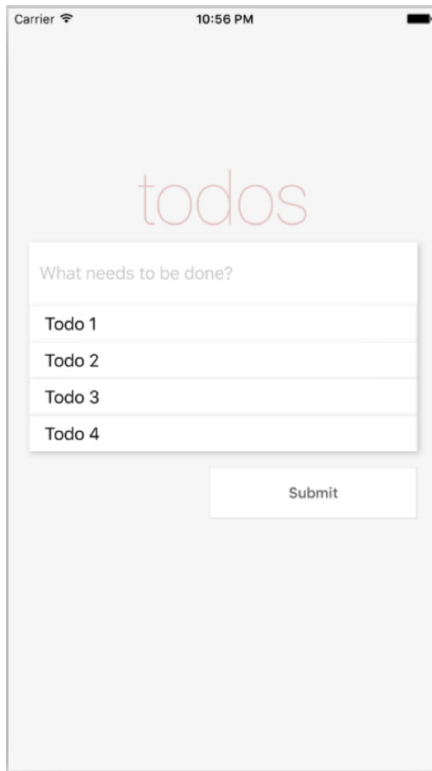


Figure 3.17 Updated app with the `TodoList` component

Listing 3.17 Adding `toggleComplete` and `deleteTodo` functions

Binds the `toggleComplete` method to the class in the constructor

Binds the `deleteTodo` method to the class in the constructor

```

constructor () {
  ...
  > this.toggleComplete = this.toggleComplete.bind(this)
  this.deleteTodo = this.deleteTodo.bind(this) <
}
...
deleteTodo (todoIndex) { <
  let { todos } = this.state
  todos = todos.filter((todo) => todo.todoIndex !== todoIndex)
  this.setState({ todos })
}

```

`deleteTodo` takes the `todoIndex` as an argument, filters the `todos` to return all but the `todo` with the index that was passed in, and then resets the state to the remaining `todos`.

```
toggleComplete (todoIndex) {
  let todos = this.state.todos
  todos.forEach((todo) => {
    if (todo.todoIndex === todoIndex) {
      todo.complete = !todo.complete
    }
  })
  this.setState({ todos })
}
...

```

toggleComplete also takes the `todoIndex` as an argument, and loops through the todos until it finds the todo with the given index. It changes the `complete` Boolean to the opposite of `complete`'s current setting, and then resets the state of the todos.

To hook in these functions, you need to create a button component to pass in to the todo. In the app folder, create a new file called `TodoButton.js`.

Listing 3.18 Creating `TodoButton.js`

```
import React from 'react'
import { Text, TouchableHighlight, StyleSheet } from 'react-native'

const TodoButton = ({ onPress, complete, name }) => (
  <TouchableHighlight
    onPress={onPress}
    underlayColor='#efefef'
    style={styles.button}>
    <Text style={
      [
        styles.text,
        complete ? styles.complete : null,
        name === 'Delete' ? styles.deleteButton : null ]
      }>
      {name}
    </Text>
  </TouchableHighlight>
)

const styles = StyleSheet.create({
  button: {
    alignSelf: 'flex-end',
    padding: 7,
    borderColor: '#ededed',
    borderWidth: 1,
    borderRadius: 4,
    marginRight: 5
  },
  text: {
    color: '#666666'
  },
  complete: {
    color: 'green',
    fontWeight: 'bold'
  },
  deleteButton: {
    color: 'rgba(175, 47, 47, 1)'
  }
})
export default TodoButton

```

Takes `onPress`, `complete`, and `name` as props

Checks whether `complete` is true, and applies a style

Checks whether the `name` property equals "Delete" and, if so, applies a style

Now, pass the new functions as props to the `TodoList` component.

Listing 3.19 Passing `toggleComplete` and `deleteTodo` as props to `TodoList`

```
render () {
  ...
  <TodoList
    toggleComplete={this.toggleComplete}
    deleteTodo={this.deleteTodo}
    todos={todos} />
  <Button submitTodo={this.submitTodo} />
  ...
}
```

Next, pass `toggleComplete` and `deleteTodo` as props to the `Todo` component.

Listing 3.20 Passing `toggleComplete` and `deleteTodo` as props to `ToDo`

```
...
const TodoList = ({ todos, deleteTodo, toggleComplete }) => {
  todos = todos.map((todo, i) => {
    return (
      <Todo
        deleteTodo={deleteTodo}
        toggleComplete={toggleComplete}
        key={i}
        todo={todo} />
    )
  })
}
```

Finally, open `Todo.js` and update the `Todo` component to bring in the new `TodoButton` component and some styling for the button container.

Listing 3.21 Updating `Todo.js` to bring in `TodoButton` and functionality

```
import TodoButton from './TodoButton'
...
const Todo = ({ todo, toggleComplete, deleteTodo }) => (
  <View style={styles.todoContainer}>
    <Text style={styles.todoText}>
      {todo.title}
    </Text>
    <View style={styles.buttons}>
      <TodoButton
        name='Done'
        complete={todo.complete}
        onPress={() => toggleComplete(todo.todoIndex)} />
      <TodoButton
        name='Delete'
        onPress={() => deleteTodo(todo.todoIndex)} />
    </View>
  </View>
)
```

```
const styles = StyleSheet.create({  
  ...  
  buttons: {  
    flex: 1,  
    flexDirection: 'row',  
    justifyContent: 'flex-end',  
    alignItems: 'center'  
  },  
  ...  
})
```

You add two `TodoButtons`: one named `Done`, and one named `Delete`. You also pass `toggleComplete` and `deleteTodo` as functions to be called as the `onPress` you defined in `TodoButton.js`. If you refresh the app and add a `todo`, you should now see the new buttons (figure 3.18).

If you click `Done`, the button text should be bold and green. If you click `Delete`, the `todo` should disappear from the list of `todos`.

You're now almost done with the app. The final step is to build a tab bar filter that will show either all the `todos`, only the complete `todos`, or only the incomplete `todos`. To get this started, you'll create a new function that will set the type of `todos` to show.

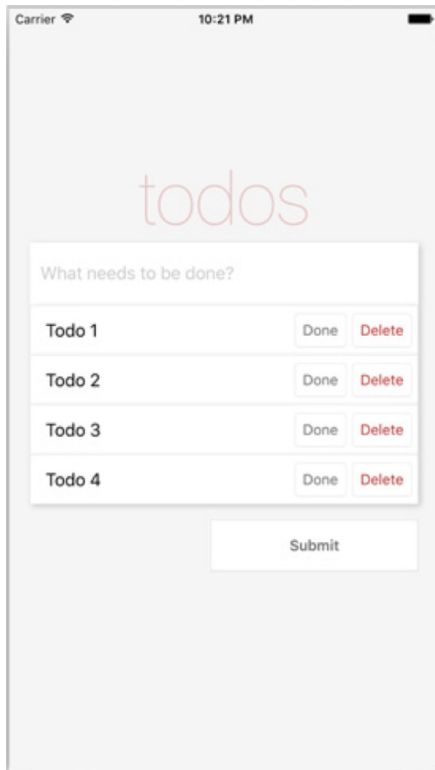


Figure 3.18 App with `TodoButtons` displayed

In the constructor, you set a state type variable to 'All' when you first created the app. You'll now create a function named `setType` that will take a type as an argument and update the type in the state. Place this function below the `toggle-Complete` function in `App.js`.

Listing 3.22 Adding the `setType` function

```
constructor () {
  ...
  this.setType = this.setType.bind(this)
}
...
setType (type) {
  this.setState({ type })
}
...
```

Next, you need to create the `TabBar` and `TabBarItem` components. First, create the `TabBar` component: add a file in the app folder named `TabBar.js`.

Listing 3.23 Creating the `TabBar` component

```
import React from 'react'
import { View, StyleSheet } from 'react-native'
import TabBarItem from './TabBarItem'

const TabBar = ({ setType, type }) => (
  <View style={styles.container}>
    <TabBarItem type={type} title='All'
      setType={() => setType('All')} />
    <TabBarItem type={type} border title='Active'
      setType={() => setType('Active')} />
    <TabBarItem type={type} border title='Complete'
      setType={() => setType('Complete')} />
  </View>
)

const styles = StyleSheet.create({
  container: {
    height: 70,
    flexDirection: 'row',
    borderTopWidth: 1,
    borderTopColor: '#dddddd'
  }
})

export default TabBar
```

This component takes two props: `setType` and `type`. Both are passed down from the main `App` component.

You're importing the yet-to-be-defined `TabBarItem` component. Each `TabBarItem` component takes three props: `title`, `type`, and `setType`. Two of the components also take a `border` prop (Boolean), which if set will add a left border style.

Next, create a file in the `app` folder named `TabBarItem.js`.

Listing 3.24 Creating the `TabBarItem` component

```
import React from 'react'
import { Text, TouchableHighlight, StyleSheet } from 'react-native'

const TabBarItem = ({ border, title, selected, setType, type }) => (
  <TouchableHighlight
    underlayColor='#efefef'
    onPress={setType}
    style={[
      styles.item, selected ? styles.selected : null,
      border ? styles.border : null,
      type === title ? styles.selected : null ]}>
    <Text style={[ styles.itemText, type === title ? styles.bold : null ]}>
      {title}
    </Text>
  </TouchableHighlight>
)

const styles = StyleSheet.create({
  item: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center'
  },
  border: {
    borderLeftWidth: 1,
    borderLeftColor: '#dddddd'
  },
  itemText: {
    color: '#777777',
    fontSize: 16
  },
  selected: {
    backgroundColor: 'ffffff'
  },
  bold: {
    fontWeight: 'bold'
  }
})

export default TabBarItem
```

In the `TouchableHighlight` component, you check a few props and set styles based on the prop. If `selected` is true, you give it the style `styles.selected`. If `border` is true, you give it the style `styles.border`. If `type` is equal to the `title`, you give it `styles.selected`.

In the `Text` component, you also check to see whether `type` is equal to `title`. If so, add a bold style to it.

To implement the TabBar, open app/App.js, bring in the TabBar component, and set it up. You'll also bring in type to the render function as part of destructuring this.state.

Listing 3.25 Implementing the TabBar component

```
...
import TabBar from './TabBar'
class App extends Component {
  ...
  render () {
    const { todos, inputValue, type } = this.state
    return (
      <View style={styles.container}>
        <ScrollView
          keyboardShouldPersistTaps='always'
          style={styles.content}>
          <Heading />
          <Input inputValue={inputValue}
            inputChange={(text) => this.inputChange(text)} />
          <TodoList
            type={type}
            toggleComplete={this.toggleComplete}
            deleteTodo={this.deleteTodo}
            todos={todos} />
          <Button submitTodo={this.submitTodo} />
        </ScrollView>
        <TabBar type={type} setType={this.setType} />
      </View>
    )
  }
  ...
}
```

Here, you bring in the TabBar component. You then destructure type from the state and *pass it not only to the new TabBar component, but also to the TodoList component*; you'll use this type variable in just a second when filtering the todos based on this type. You also pass the setType function as a prop to the TabBar component.

The last thing you need to do is open the TodoList component and add a filter to return only the todos of the type you currently want back, based on the tab that's selected. Open TodoList.js, destructure the type out of the props, and add the following getVisibleTodos function before the return statement.

Listing 3.26 Updating the TodoList component

```
...
const TodoList = ({ todos, deleteTodo, toggleComplete, type }) => {
  const getVisibleTodos = (todos, type) => {
    switch (type) {
      case 'All':
        return todos
      case 'Complete':
        return todos.filter((t) => t.complete)
    }
  }
  ...
}
```

```

    case 'Active':
      return todos.filter((t) => !t.complete)
    }
  }

  todos = getVisibleTodos(todos, type)
  todos = todos.map((todo, i) => {
    ...

```

You use a switch statement to check which type is currently set. If 'All' is set, you return the entire list of todos. If 'Complete' is set, you filter the todos and only return the complete todos. If 'Active' is set, you filter the todos and only return the incomplete todos.

You then set the todos variable as the returned value of `getVisibleTodos`. Now you should be able to run the app and see the new TabBar (figure 3.19). The TabBar will filter based on which type is selected.

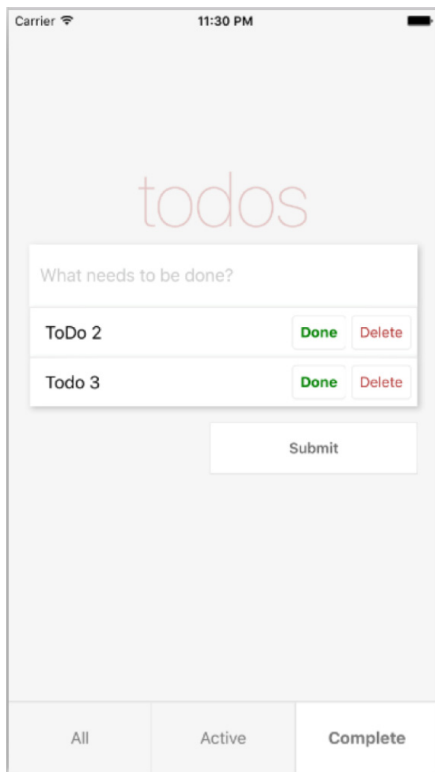


Figure 3.19 Final todo app

Summary

- AppRegistry is the JavaScript entry point to running all React Native apps.
- The React Native component TextInput is similar to an HTML input. You can specify several props, including a placeholder to show text before the user starts to type, a placeholderTextColor that styles the placeholder text, and a selection-Color that styles the cursor for the TextInput.
- TouchableHighlight is one way to create buttons in React Native; it's comparable to the HTML button element. You can use TouchableHighlight to wrap views and make them respond properly to touch events.
- You learned how to enable the developer tools in both iOS and Android emulators.
- Using the JavaScript console (available from the developer menu) is a good way to debug your app and log useful information.

In this chapter, we'll discuss what the MVVM (model-view–view model) design pattern is and how it maximizes your cross-platform code. You'll also roll up your sleeves and write some code!

Hello MVVM—creating a simple cross-platform app using MVVM

This chapter covers

- What MVVM is and why it's the best choice for cross-platform Xamarin apps
- What the MVVM design pattern is all about, and why you'd want to use it to maximize your cross-platform code
- Getting set up with Xamarin and the MvvmCross extension
- Creating HelloCrossPlatformWorld, your first Xamarin mobile app
- Running your app on iOS and Android

Typically at this point in a book, it's traditional to build a Hello World app to show off the technology in question. For this book, though, I'm going to go slightly against tradition and start by discussing the MVVM (model-view-view model) design pattern. Then we'll get our hands dirty with some code toward the end of this chapter.

WE'RE DISCUSSING MVVM FOR CROSS-PLATFORM XAMARIN APPS The principles discussed in this chapter are for using MVVM with Xamarin apps. Although these follow the principles for MVVM on other platforms, such as desktop Windows apps or the web, there's a lot more to it for Xamarin apps. If you've done MVVM before (maybe with WPF) it's still worth reading this chapter as there are some important differences.

2.1 What are UI design patterns?

Over time, developers have come across and solved the same problems again and again. Out of this has come a set of abstract solutions that can be applied when building your code. These are known as *design patterns*—repeatable solutions to common problems that occur when designing and building software.

Building apps that interact with the user through a user interface (UI) is no different. There are standard problems that developers want to solve, and a number of patterns have come about as solutions to these problems.

Let's consider a simple square-root calculator app called Sqrt that has a text box you can put a number in, and a button. When you tap the button, it calculates the square root of the number in the text box and shows the answer on a label. An example of this app is shown in figure 2.1.

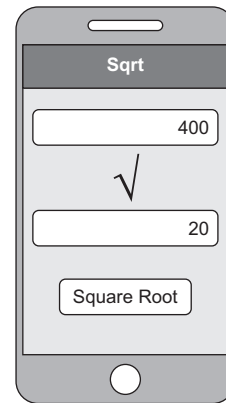


Figure 2.1 A simple square-root calculator app that calculates the square root of a given number

The simplest way to write this app is to wire up the button to an event that takes the value directly from the text box, calculates the square root, and writes the value to a label. All this can be done in the code-behind file for the UI. Simple, and all in one class. The following listing has some pseudocode for the kind of thing you might write.

Listing 2.1 Pseudocode for adding numbers by wiring to the UI directly

| | |
|---|---|
| <p>Listens for the Click event of the button</p> <pre> MyAddButton.Click += (s, e) => { var number = double.Parse(NumberTextBox.Text); var sqrt = Math.Sqrt(number); MyResultLabel.Text = sqrt.ToString(); } </pre> | <p>The number comes from reading the value from the Text property of the text box.</p> |
| <p>Once the square root is calculated, the Text property of the label is set directly.</p> | |

Although this seems simple, it has a number of flaws.

First, this isn't easily testable. You can only test this app by updating the value in the text box and tapping the button. It would be better if you could write unit tests so you could programmatically test the code, covering multiple cases including edge cases, such as missing inputs or large or negative numbers. This way you could run a set of automated tests quickly and repeatably every time you change your code.

Second, this isn't cross-platform. One of the reasons for building apps using Xamarin is so that parts of your app can be written in shared code that works on both iOS and Android. If your calculation is wired directly to the view, you can't do this. Think back to the layers introduced in chapter 1, shown in figure 2.2.

In a Xamarin app we have three layers:

- *Application layer*—This is a small part of the code that makes your app runnable on each platform and has different platform-specific implementations for iOS and Android.
- *UI layer*—The UI layer also has separate platform-specific implementations for iOS and Android.
- *Business logic layer*—The business logic layer is shared between the two platforms.

To fit the calculator code into this structure, you'd need to have your calculation code in the cross-platform business logic layer, and the button, text box, label, and all the wiring in the UI layer. This is the kind of problem all UI developers come across on a daily basis, and, as you'd expect, there's a design pattern to help with this—MVVM.

2.2 MVVM—the design pattern for Xamarin apps

MVVM (model-view-view model) is the most popular design pattern for cross-platform apps built using Xamarin, and it has a history of being a very successful design pattern for building Windows desktop apps using WPF, Silverlight apps, and now Windows 10 UWP apps. It has even made its way onto the web with frameworks like Knockout.js using it. When Xamarin designed Xamarin.Forms, whose goal was to have as much code sharing as possible, the principles of MVVM were baked into the underlying framework right off the bat.

Think back to the three layers in the Xamarin app. These three layers enable a reasonable amount of code sharing, but we can do better. In the UI layer there are really two layers—the actual UI widgets, and some logic around these widgets. For example, we could put some logic around the answer label to make it only visible once the square root has been calculated. This expands our three layers to four.

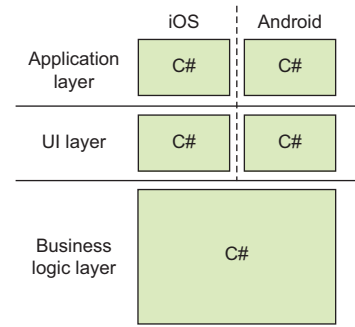


Figure 2.2 Xamarin apps are written in C# so you can share any common business logic while having a platform-specific UI.

Figure 2.3 shows the how the layers would look if we could move this UI logic into shared code. If we did this, the label in our example would be in the UI layer, and the logic that decides whether it should be visible or hidden would be in the cross-platform UI logic layer. This is a great way to do things—we’re maximizing code reuse by abstracting the UI logic into cross-platform code.

MVVM helps with this splitting-out of the UI and its logic. This pattern is named based on the three layers that you use in your app, as shown in figure 2.4. Let’s look at these layers in the context of our calculator example:

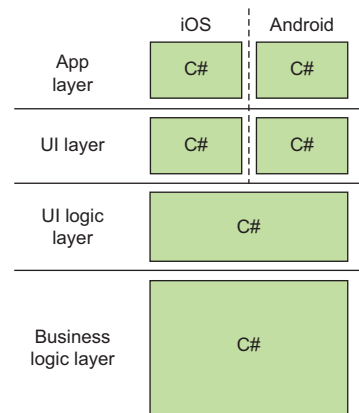


Figure 2.3 To maximize code reuse, it would be good to have UI logic in shared code.

- *Model*—Your data and business logic.

The model is the data, business logic, and access to any external resources (such as web services or databases) defined in terms of the domain, and this maps to the business logic layer in our Xamarin app. In our example, the model contains the number, the logic to calculate the square root, and the result.

- *View*—The actual UI, buttons, text controls, and all other widgets.

The view is the UI with all its widgets and layouts, and this maps to part of the UI layer and holds the UI widgets (the text box, button, and label). This is a passive view, so it doesn’t have any code to get or set the values or to handle events, such as the button click.

- *View model*—The UI data and logic.

For our calculator app, this has properties that represent the numbers on the model—the input value and the result. It also has a command property that wraps the square root calculation logic on the model into an object (more on commands in the next chapter). The view model knows about the model but has no knowledge of the view.

In addition to these three layers, it has a *binder*, a binding layer that you can think of as glue that connects the view model to the view. This removes the need to write boilerplate code to synchronize the UI—the binder can watch for changes in the view model and update the view to match, or update the view model to match changes made by the user in the UI. This binder is loosely coupled rather than tightly coupled, and the connection is often made based on wiring up properties in the view and view model based on their names (so in the case of a binding between a property called `Text` and a property called `Name`, at runtime the binder will use reflection to map these string values to the underlying properties).

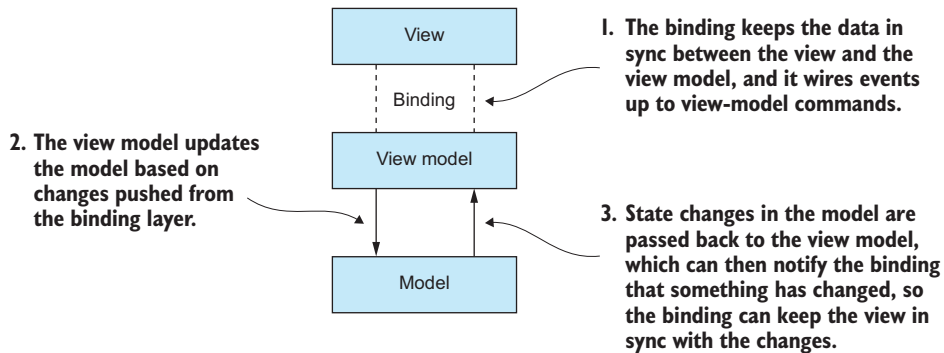


Figure 2.4 MVVM has a model, a view model, a view, and a binding layer that keeps the view and view model in sync and connects events on the view to the view model.

Reflecting on reflection

If you've never heard of reflection before, it's a part of the C# API that allows you to query details about a class—you can discover properties, fields, methods, or events. Once you've found out the details, you can also execute code. For example, you can find a property based on its name and then get the value of that property from a particular instance of that class. Reflection is also common in other languages such as Java—C# reflection is basically the same as Java reflection.

This is great for binding—if you bind a property called `Name`, the binding code can use reflection to find a property on your view-model class with that same name, and then it can get the value on your view-model instance.

For our calculator app, the binding would wire up the text box, button, and label on the UI to the equivalent properties and a command on the view model.

There's a bit of magic involved in making this binder work, and this is usually implemented in an MVVM framework—a third-party library that gives you a set of base classes providing the implementation of this pattern. I cover how this works later in this chapter.

MVVM FRAMEWORKS There are multiple MVVM frameworks that work with Xamarin native apps, such as `MvvmCross`, `MVVM Light`, and `Caliburn.Micro`. Although each one has differences, they all follow the same basic principles and do roughly the same things. Later in this book we'll be using `MvvmCross`, but everything in this book is applicable to most frameworks.

For example, as shown in figure 2.5, we could have a text box on our calculator app UI that's *bound* to a `Number` property. This means that at runtime it will try to find a public property called `Number` on the view model that it's bound to using reflection,

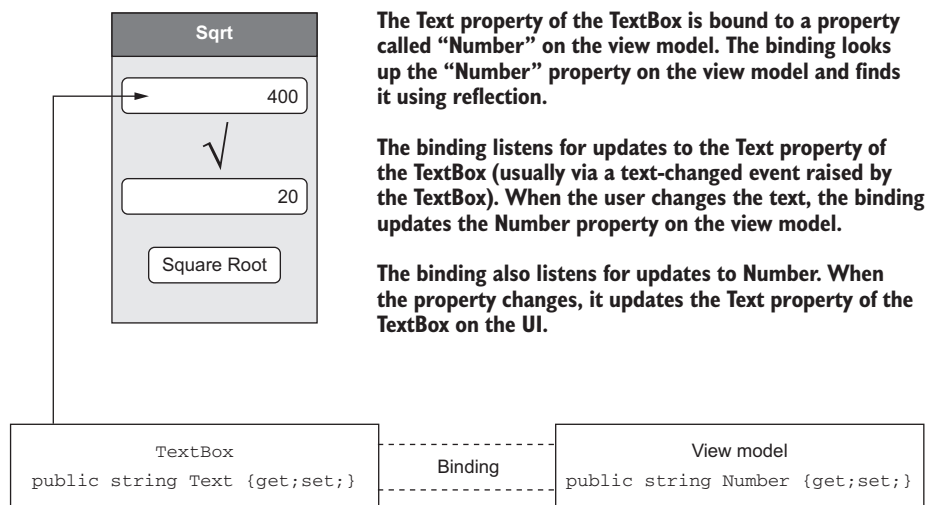


Figure 2.5 Binding keeps the value on the view in sync with the value in the view model.

and it will show the string contained in that property in the text box. If the user changes the value inside the text box, it will update the value of the **Number** property to match what the user has typed in. Conversely, if the value of the **Number** property on the view model changes, the binding will update the text box to match.

The binder doesn’t care what the underlying class type is of the view model you’re using, just that it has a public property called **Number** that it can extract the value from. In some of the MVVM frameworks, it doesn’t even care if the property is there or not. If it can’t find the property, it just treats it as an empty value. This loose coupling is what makes MVVM especially powerful—it allows view models to be completely agnostic to the view, so you can write unit tests against the view model that simulate the UI without worrying about UI code getting in the way. It also supports code reuse, so a view could be glued to any view model that has properties with the names it’s expecting.

Figure 2.6 expands on the previous figures by showing how these layers map to the three layers of MVVM:

- *App layer*—The app layer is one that doesn’t really come under the pure MVVM design pattern, but the different MVVM frameworks do provide some application-layer features. This allows us to have some cross-platform code in our app layer that can control app logic, such as which view is shown first and how the different classes in the app are wired together, such as defining which view model is used for each view.
- *UI layer*—The UI layer is our view layer, and this is platform-specific code.
- *Binding*—The binding between the UI layer and the UI logic layer is the binder—the glue that connects the UI layer to its logic layer. This is usually

a mix of cross-platform and platform-specific code provided by a third-party framework.

- *UI logic layer*—The UI logic layer is our view-model layer. It provides logic for the UI and other device interactions in a cross-platform way. Part of this logic is value conversion—converting from data in your domain objects to data on the UI. For example, you could model a user in your domain with a first name and last name but on the UI want to show the full name. The view model will provide this value conversion by concatenating the names and giving one string value that will be shown by the UI.
- *Business logic layer*—The business logic layer is the model layer. This contains data, domain objects, logic, and connectivity to external resources such as databases or web services. Again, this is cross-platform.

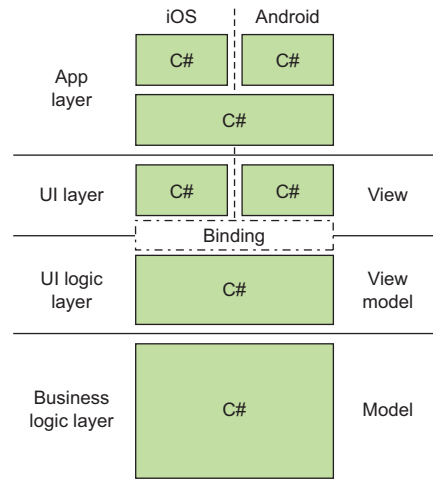


Figure 2.6 The different layers of MVVM fit with the different layers of a Xamarin app.

A QUICK HISTORY LESSON MVVM has been around since 2005 and was developed by two architects from Microsoft, Ken Cooper and Ted Peters. It was primarily created for use with the new UI technology stack coming out of Microsoft called WPF, and it leverages the data binding that was a key feature of WPF. In WPF you write your UI using XAML, a UI-based markup language, and in this XAML you can bind the properties of a UI widget to properties defined in the data context of the view—essentially the view model. This allowed UI/UX experts to design the UI using more designer-based tools, and to simply wire the widgets, based on their names, to code written independently by developers.

2.3 What is cross-platform code?

Some of the layers in our MVVM-based app use cross-platform code—specifically, part of the app layer, the UI logic (view-model) layer, and the business logic (model) layer. The reason for this is simple—we’re building an app for both iOS and Android, so the app will need to work the same way on both platforms, use the same type of data, and have roughly the same UI logic. It makes a lot of sense to build this once and use the same code on both apps—code that we write once and can run on iOS and Android. The term *cross-platform code* has come up a lot already in this book, and it will continue to be a theme throughout. But what exactly do we mean when we talk about cross-platform code in C#?

Cross-platform native apps are not truly cross-platform

In the Xamarin world we talk of cross-platform native apps, but these are not true cross-platform apps where exactly the same app will run on all platforms. Neither is it cross-platform in that all the code runs on all platforms (with a hidden app layer).

What I mean here is that we have two apps, one that runs on iOS and one that runs on Android, both developed using the same language and sharing a large portion of the code. They're cross-platform in that the business logic (and ideally the UI logic) is truly cross-platform, and the smallest possible device-specific UI and feature layer is built to be platform-specific.

The MVVM design pattern is very well suited to helping you get as much code-sharing as possible.

2.3.1 .NET Standard class libraries

When Microsoft released the .NET Framework, they provided a set of APIs that work on Windows, and with each version of the framework they added more APIs that developers can use. Over time, support for more platforms was added, such as Microsoft's Silverlight (apps running in a browser) or the Windows Store (apps running in a sandbox and distributed via a Microsoft app store). These different platforms didn't provide the same capabilities, so code written against the core .NET Framework might not work on Silverlight if it required APIs that Silverlight didn't (or couldn't) implement. The initial solution to this was portable class libraries (PCLs)—libraries that targeted a common subset of the .NET APIs that would run on all platforms. Xamarin took advantage of this, using the same model to allow you to write portable class libraries that targeted the subset of the .NET Framework that runs on iOS or Android.

This worked after a fashion, but it caused a lot of confusion. PCLs come in *profiles*—a profile being a defined subset that will work on a particular combination of platforms. One profile would work on iOS, Android, and Windows under .NET 4.5, whereas another would also run on iOS and Android but require .NET 4.6. This meant that not only would you need to choose the right profile for the platforms you were targeting, but you'd also need any third-party libraries to also target a compatible profile. If your profile included .NET 4.5 on Windows, you couldn't use a library that used a profile that needed .NET 4.6, for example.

Things are now a lot better, thanks to a new initiative from Microsoft called .NET Standard. This is an attempt to standardize the different .NET platforms into a versioned set of APIs. Each platform, such as Xamarin iOS, Xamarin Android, or the .NET Framework on Windows implements a particular version of the standard, as well as all previous versions. This is an inclusive standard, so if a platform supports .NET Standard 1.6, it also includes 1.5, 1.4, and so on, all the way back to 1.0. The idea

behind this is simple—each version has more APIs available than the previous version, and your code can use libraries that target the same or an earlier version of the standard. For example, if your code targets .NET Standard 1.6, you can use a library that targets 1.4. You can think of the .NET Framework on Windows as the most complete set of APIs, and each .NET Standard version as a more complete implementation of the full .NET Framework.

You can read more on .NET Standard libraries, and see what version of the standard is implemented by which version of each platform on Microsoft Docs at <http://mng.bz/sB0y>. At the time of writing, Xamarin iOS and Android supports version 2.0, so you can use code that targets 2.0 or earlier from your Xamarin apps. Be aware, though, that targeting higher versions may limit the platforms you support. At the time of writing, UWP only supports 1.4, so if you decide to add a UWP project to your Xamarin apps to support Windows 10, you'll need to ensure the core projects used by your app target 1.4 or lower.

These .NET Standard libraries are perfect for the cross-platform layer in your Xamarin apps. The set of APIs that .NET Standard libraries implement includes all the bits that would work on all platforms—collections, Tasks, simple I/O, and networking. What isn't included is anything that's specific to a platform, such as UI code. This is left up to platform-specific code to implement. .NET Standard is just an API specification, it's not the actual implementation. Under the hood, the code that makes up the subset of the .NET APIs isn't the same on all platforms, each platform implements their features using the native API that the platform provides. But the interface to it—the classes and namespaces—are the same.

When you write your cross-platform app, you want as much code as possible inside .NET Standard libraries, as this is the code that's shared. Thinking again about the layers in our app, you can easily see which layers would live in a .NET Standard library, as shown in figure 2.7.

To map this to the project structure you're probably used to in a C# solution, you'd have (at least) three projects. One (or more) would be a .NET Standard project containing all your cross-platform UI and business logic code. Another would be an iOS app project that contains the iOS application code and the iOS UI code. And the last would be an Android app project that contains the Android-specific UI and application code. This is illustrated in figure 2.8.

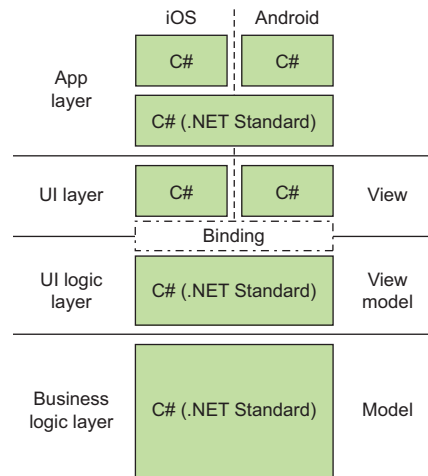


Figure 2.7 The cross-platform layers in a mobile app are implemented in .NET Standard libraries.

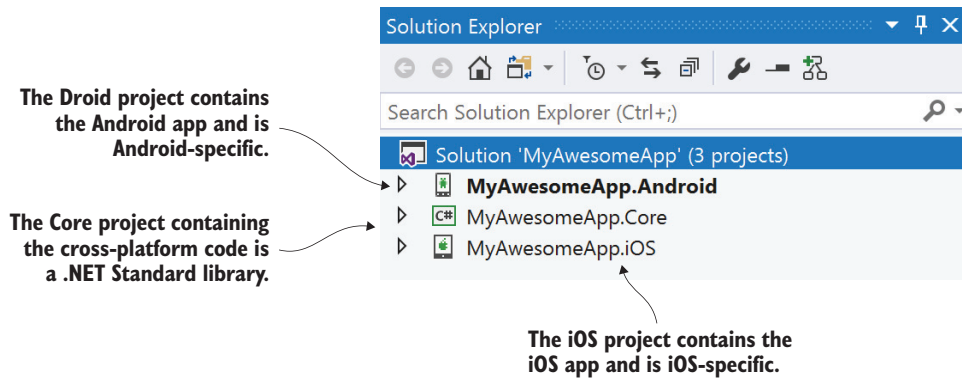


Figure 2.8 A typical cross-platform app would contain a .NET Standard library with the core code, an Android app with Android-specific code, and an iOS app with iOS-specific code

Now that you’ve seen some of the basics, let’s build a simple example app using the MvvmCross MVVM framework.

2.4 Getting started—creating your first solution

As promised, you’re now going to create a Hello World app—a simple app that doesn’t do very much but allows you to be sure your development environment is working correctly, and to see how simple it is to create a working app. Because the big strength of Xamarin is to allow you to create cross-platform apps with a large amount of code sharing, you’re going to create two apps: one for iOS and one for Android. They will share a common core library with all the business logic—inasmuch as you can have business logic in a Hello World app. You’ll also leverage what you’ve learned in this chapter and build it using MVVM. The MvvmCross framework you’ll be using here will save you writing a lot of boilerplate code. This framework is hugely popular with developers building cross-platform Xamarin apps, and it’s very actively maintained and enhanced.

MVMCROSS We’ll be covering what you need to know about MvvmCross to build your example apps in this book. If you want to read more about it (or contribute to the framework—it’s fully open source and welcomes contributions) then head to <https://mvvmcross.com>.

We’ll be following these steps to achieve this:

- *Creating and running a new cross-platform app*—We’ll be creating a cross-platform MvvmCross app using a Visual Studio extension that we’ll be installing. Once this solution has been created, we’ll fire it up on iOS and Android as a sanity check.
- *Proving the code is cross-platform*—Just to prove we have a cross-platform app with shared code, we’ll be tweaking the code in one place and seeing the effect it has on both apps.

Despite using MvvmCross here and in the apps we'll build in later chapters, the aim is not to lock you into this framework. We'll only be using some small parts of it, and the principles behind those parts are pretty standard for the MVVM pattern. These principles are easy to apply when using other frameworks, such as MVVM Light.

2.4.1 Requirements—what hardware or software do you need for each mobile platform?

In chapter 1 we discussed Xamarin's platform support and the tooling you can use. Here's a quick refresher:

- If you have a Windows PC, you need to install Visual Studio 2017 and ensure the "Xamarin" workload is ticked in the installer.
- If you have a Mac, you need to install Visual Studio for Mac, which includes Visual Studio as well as the iOS and Android Xamarin components, the Android SDK, and Google's Android emulator. You also need to install Xcode from the Mac App Store for iOS development.
- If you want to develop iOS apps using Visual Studio on Windows, you need to have access to a Mac with Xamarin and Xcode installed.
- Always install or upgrade to the latest stable versions of all components, such as the latest version of VS 2017, the latest VS for Mac, the latest Xcode, and the latest Android SDK and tools. To install the latest Android SDK and tools, you'll need to use the Android SDK manager, available from Visual Studio by going to Tools > Android > Android SDK Manager on Windows or Tools > SDK Manager on the Mac.

This book doesn't cover installation

The Visual Studio installers change pretty often, so it's hard to keep up with them in print. Although this book does outline what's needed, it doesn't cover installation and configuration in detail.

At the time of writing, the Visual Studio for Mac installer gives you everything you need on Mac, including Android SDKs and emulators. The only extra thing you need to install is Xcode from the Mac App Store to build iOS apps.

On Windows, the Visual Studio 2017 installer installs everything, as long as you tick the right options for cross-platform development, Android SDKs, and emulators, which change a bit with each update. If you're using a Windows virtual machine on your Mac to run Visual Studio, you'll need to enable your virtual machine to host a nested virtual machine if you want to run the Android emulators—check the VM documentation for how to do this. If you use a PC, you'll need an Intel CPU with virtualization enabled (most modern CPUs have this). The system requirements for running the emulators are listed at the Android Studio site (<http://mng.bz/hkXV>).

(continued)

If you get stuck, Xamarin has great documentation on its website (<https://aka.ms/XamDocs>) covering everything you need for installation and setup. The site also has helpful forums with a great community of users, and Xamarin's own engineers if you get a particularly strange problem. And obviously, there's always Stack Overflow.

At this point I'm going to assume you already have everything you need installed. If not, now would be a good time to do it.

For this little test app, we're only going to test on the Android emulator and iOS simulator, so don't worry if you don't have a physical device to use. If you do have a physical device, then put it to one side for now and just use the emulator/simulator as there's a bit of device setup you need to do to run and debug apps on real devices. On Android this is simple, but on iOS it's a bit more complicated. We'll be discussing this in chapter 13.

As previously mentioned, we'll be using the MvvmCross framework, and luckily for us there's an extension available for Visual Studio that allows us to create a new cross-platform solution. This solution contains a core library and platform-specific apps for all supported platforms (so on Visual Studio for Mac you get an iOS app and an Android app; on Windows it's iOS, Android, WPF, and UWP). Seeing as we'll be installing an extension, and the projects we create will need NuGet packages, you'll need internet access. This may sound obvious, but if you're in a coffee shop, now would be a good time to grab their WiFi password.

2.4.2 Creating the solution

Let's look at how to install the extension and create our first solution.

CREATING THE SOLUTION USING VISUAL STUDIO FOR MAC

From Visual Studio, select Visual Studio > Extensions. This will bring up a dialog box to allow you to add or remove extensions. From here, select the Gallery tab, ensure the repository is set to Visual Studio Extension Repository, and look for MvvmCross Template Pack under IDE Extensions, or by using the search (see figure 2.9). Select this and click Install. Then click Install on the dialog box that pops up.

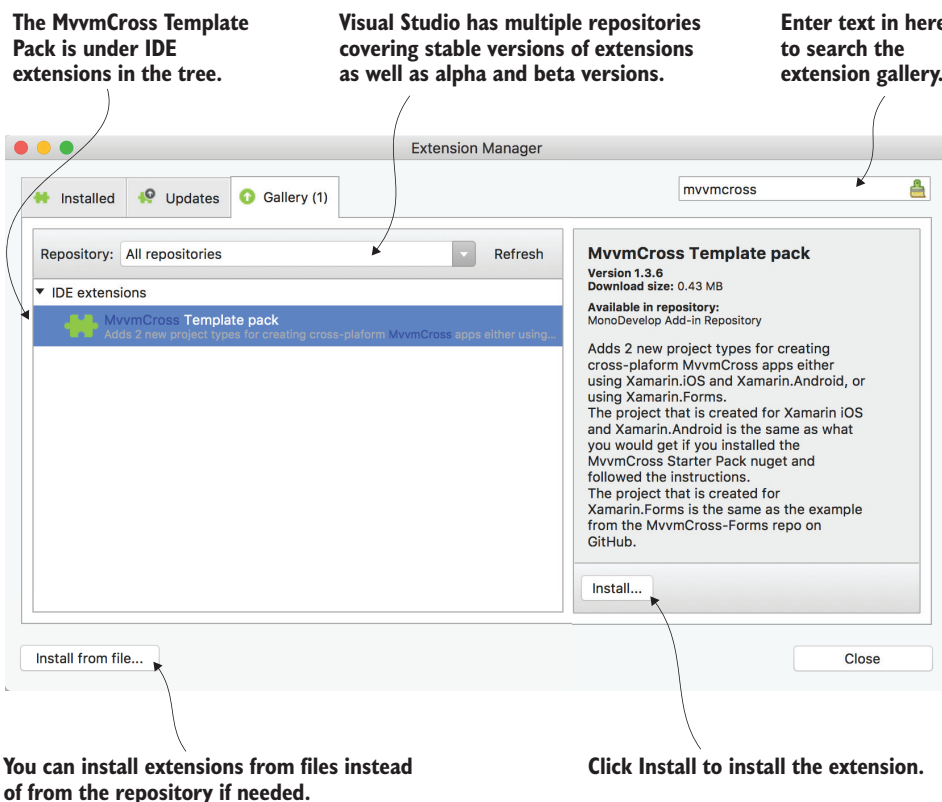
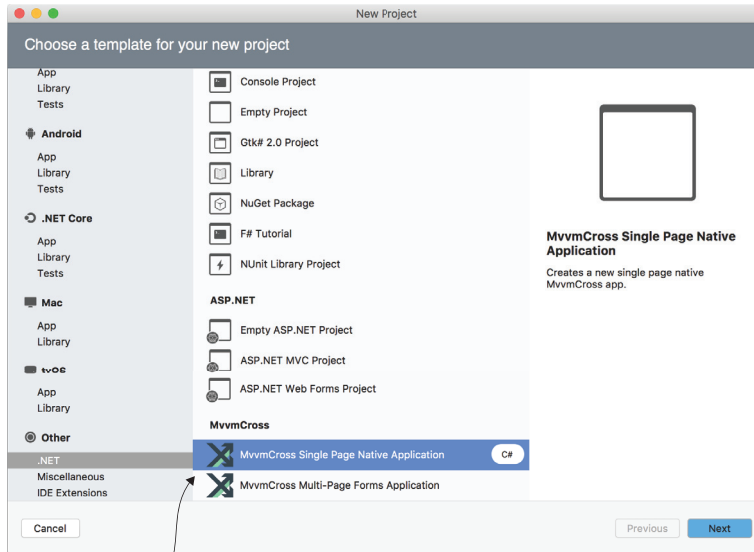


Figure 2.9 Selecting the MvvmCross Template Pack from the Visual Studio extension manager

Once this is installed, it's a good idea to restart Visual Studio, as the new solution type won't appear in the right place until you do.

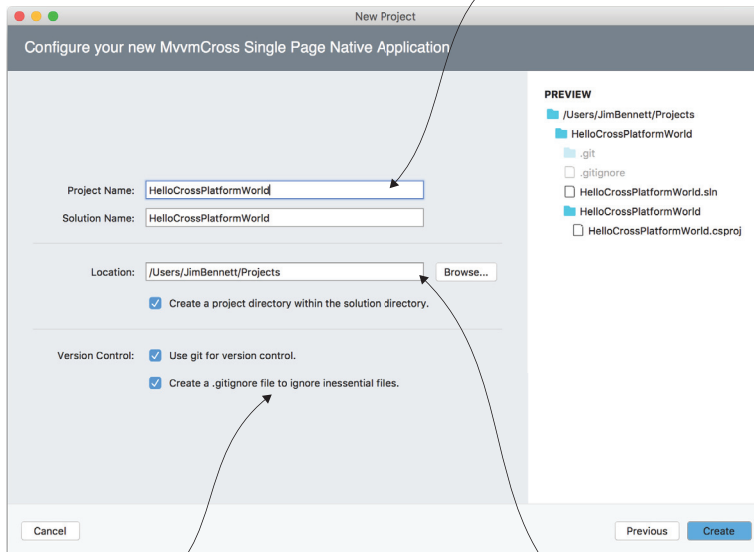
Once Visual Studio is restarted, you can start creating a new solution. You can access the New Solution dialog box in three ways.

- From the menu by going to File > New > Solution
- Using the keyboard shortcut Shift-Command-N ($\hat{+}+\mathbb{C}+N$)
- By clicking the New Project button at the bottom of the Get Started page shown when you open Visual Studio for the first time. Whichever way you choose, you'll then be presented with the New Solution dialog box (figure 2.10).



Select Other > .Net, then select MvvmCross Single Page Native Application from the MvvmCross section.

Enter the project name here. By default, the solution is given the same name as the project.



Visual Studio will, by default, create all the files needed to push this to a Git repository, even creating an appropriate .gitignore file for you.

You can change the folder the project is created in here.

Figure 2.10 The New Solution dialog boxes showing the MvvmCross cross-platform app solution template, and setting the project name

From this dialog box select Other > .NET from the left-side list, and then select MvvmCross Single Page Native Application from the list in the middle. Click Next. On the next screen enter `HelloCrossPlatformWorld` as the project name and click Create.

This will create a new solution for you containing three projects: a .NET Standard core project (`HelloCrossPlatformWorld.Core`), an iOS app (`HelloCrossPlatformWorld.iOS`), and an Android app (`HelloCrossPlatformWorld.Droid`), as shown in figure 2.11. Once the solution has been created, it will try to download all the NuGet packages it needs—you’ll see the status bar at the top showing Adding Packages. This may take a while, depending on the speed of your internet connection, and you may be asked to agree to some license agreements as they download. You’ll need to let them fully download before building the apps.

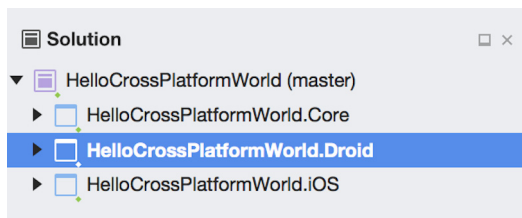


Figure 2.11 The three projects that are created for you in the new solution

WHY NOT HELLOCROSSPLATFORMWORLD.ANDROID The convention for Android apps is to use “Droid” in their names instead of Android. This is because the project name becomes the default namespace, and if you have “<something>.Android” in your namespace, you can get a clash with the global “Android” namespace. You end up littering your code with `global::Android.<whatever>` in using directives or types, making it harder to read. Stick to .Droid, it’s easier!

CREATING THE SOLUTION USING VISUAL STUDIO FOR WINDOWS

From Visual Studio select Tools > Extensions and Updates. Select the Online section on the left, and use the search box to search for MvvmCross for Visual Studio (figure 2.12). There are multiple extensions with the same and similar names, so ensure the one you install is named “MvvmCross for Visual Studio” and is at least version 2.0. Select it and click the Download button, and click Install in the dialog box that pops up.

Once this is downloaded, you’ll be prompted to restart Visual Studio to install the extension, so close Visual Studio and wait for the extension installer to finish. After this has finished, restart Visual Studio, and you can create the new solution in two ways:

- From the File menu by selecting File > New > Project
- By clicking the New Project option from the Start section of the Start Page that’s shown whenever you open Visual Studio

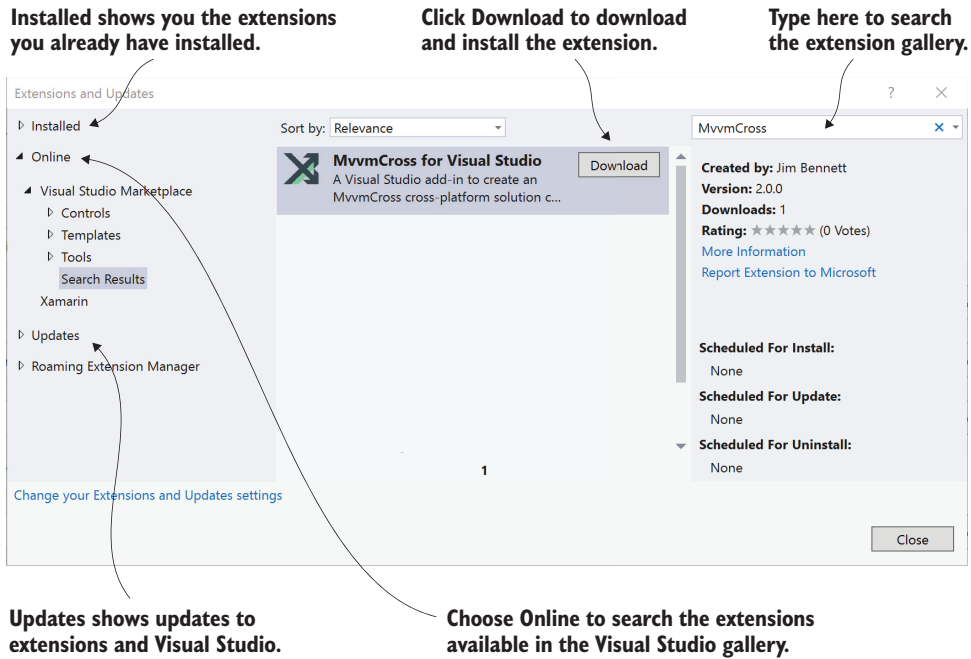


Figure 2.12 Selecting the MvvmCross for Visual Studio extension from the Visual Studio Extension manager

From the New Project dialog box (shown in figure 2.13), select the MvvmCross section under Visual C# on the left, choose MvvmCross Single Page Native Application from the list in the middle, enter HelloCrossPlatformWorld as the project name, and click OK. Windows has problems with paths longer than 256 characters, and some of the directories that will be created when your app is built have long names, so you may want to ensure your solution is created in a folder close to the root of a drive. If you do it in `C:\Users\<username>\Documents\visual studio 2017\Projects`, your path may be too long.

This will create five projects for you: a .NET Standard core project, an iOS app, an Android app, and a couple of Windows apps covering WPF and UWP. We're only interested in supporting iOS and Android here, so you can delete the Universal Windows and WPF projects by selecting them and pressing Delete or using Remove from the right-click context menu. This will leave you with the same three projects as on Visual Studio for Mac: the core project, the iOS app, and the Android app, as shown in figure 2.14.

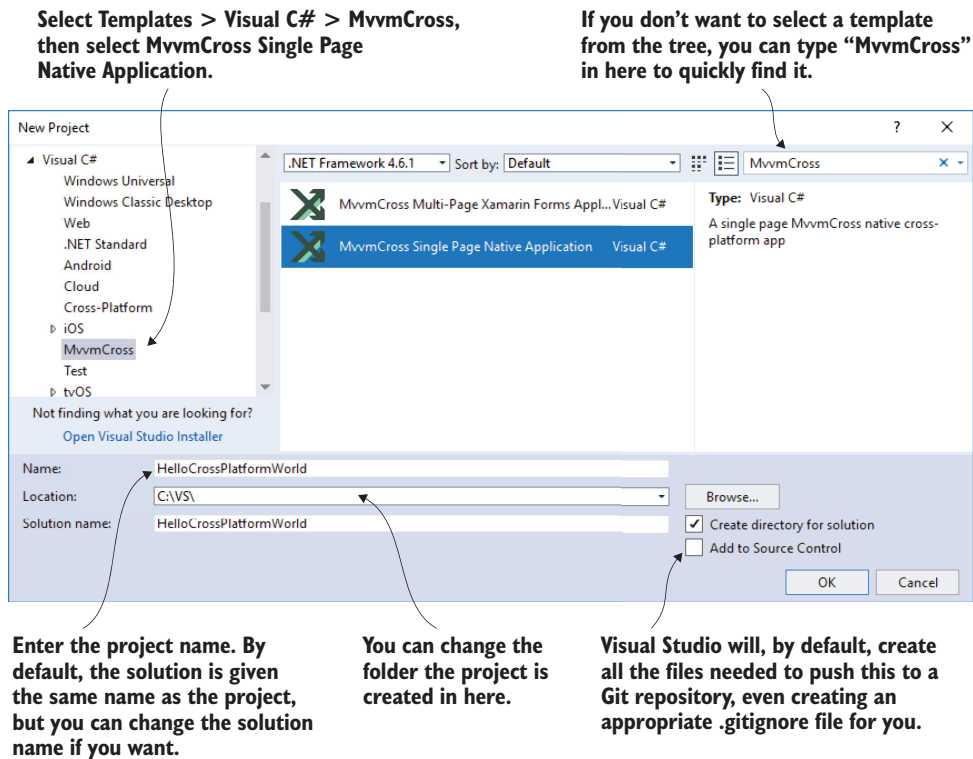


Figure 2.13 The New Project dialog box, where you can create your new solution

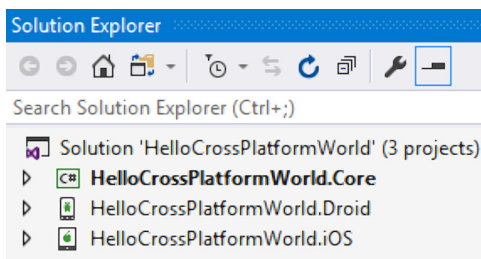


Figure 2.14 The three projects left in the solution after deleting the unwanted ones

Connecting Visual Studio to a Mac for iOS development

I won't be covering this in detail here, as this is well documented in Xamarin's "Getting Started" guide, on the developer site at <http://mng.bz/KbiM>, and it could potentially change between the time of writing and when you are reading this.

Essentially, though, you need to allow remote login on a Mac that already has Xamarin and Xcode installed. Visual Studio then connects to this Mac to build your iOS app. The process is pretty simple, and if you use a Mac hosted in the cloud, your provider should be able to provide instructions about how to set it up.

2.4.3 What have we just created?

The MvvmCross extension has given us three projects that we care about. We have a cross-platform core project and two app projects. These projects reference MvvmCross NuGet packages providing the MvvmCross MVVM framework.

When you create this project, the NuGet packages may not be the latest

NuGet packages are versioned. You can install version 1.0 of a package from the public NuGet server, and later the author could update it to version 1.1. You can then easily update the NuGet package from inside Visual Studio.

Be wary though. Sometimes packages may not be backwards compatible. The MvvmCross extension may not always install the latest versions of the MvvmCross NuGet packages, and if you update them, the code created by the extension will probably still work, but there are no guarantees.

The core project is a combination of two of our layers—the cross-platform business logic layer and the cross-platform UI logic layer. These layers don't need to exist in separate projects—they're just conceptual layers. The core contains a view model for the app plus some cross-platform application logic (we'll discuss the application layer in the next chapter). Figure 2.15 shows the structure of this project in the solution pad.

You'll notice here that we don't have any models. In this simple example, the model is just a string that's wrapped up inside the view model (and we'll play with this string a bit later). This isn't normal—in a real-world case, the view model would need something in the model layer so that it could represent the model layer's state and behavior. For now though, as this is a trivial Hello World, there's no model layer.

The platform-specific app and view layers, as well as the binding, live inside the two app projects—one for iOS and one for Android—as the code for these apps is

Solution

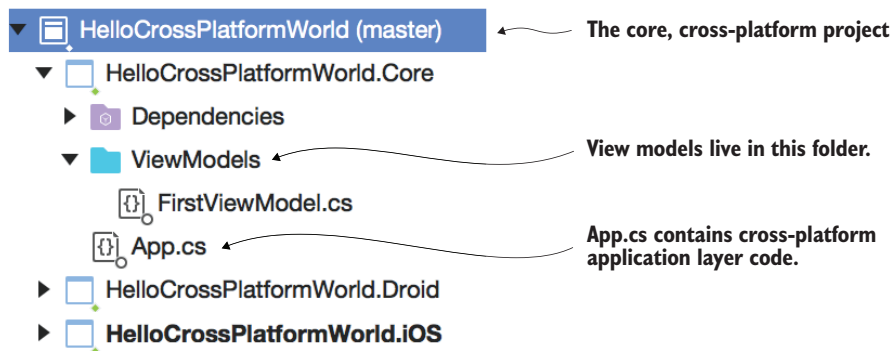


Figure 2.15 The structure of the cross-platform core project



Figure 2.16 The structure of the iOS and Android app projects

platform-specific. The structure is shown in figure 2.16. In the upcoming chapters we'll go into more detail about how Android and iOS define their application layers and their views.

2.4.4 Building and running the apps

We have two apps now, so let's run them and see what happens. Figure 2.17 shows what you'll see when they're running. In both cases we have an app that has an editable text box and a label. If you change the text in the text box, the label will be updated instantly.

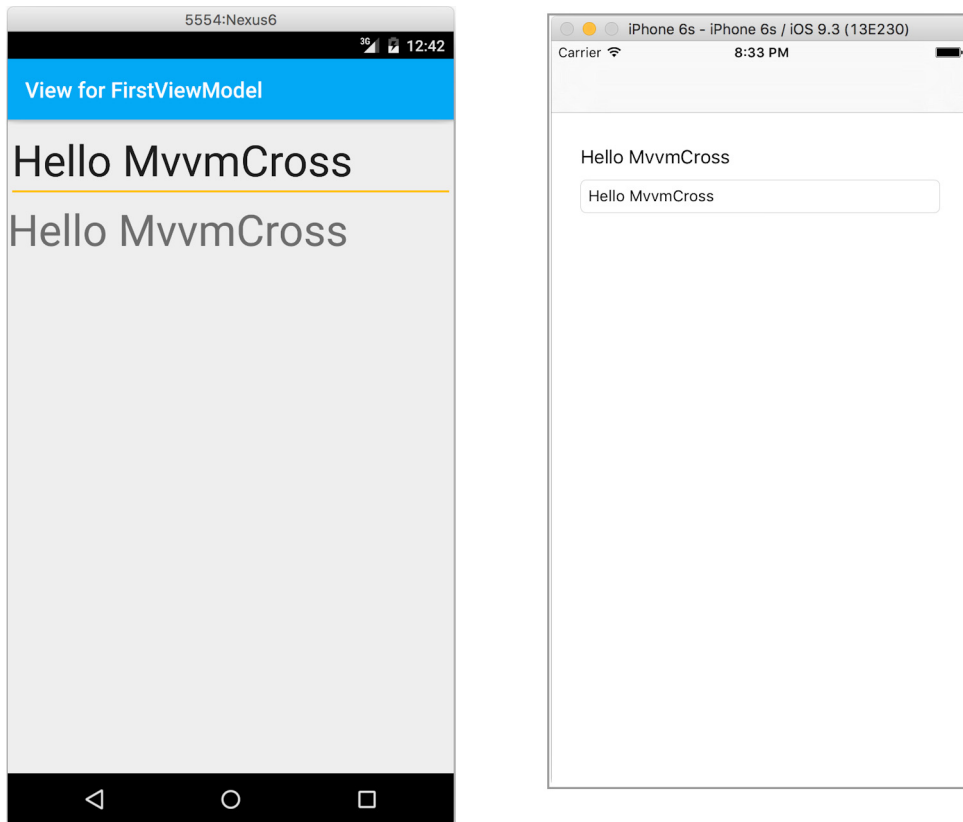


Figure 2.17 Our Hello Cross-Platform World apps running on both Android and iOS

When you used the MvvmCross extension to create the solution, it created these two apps for you, both using some shared common code.

ANDROID

Let's start by taking the Android app for a spin.

SWITCHING FROM MAC TO WINDOWS The project and solution files created by Visual Studio for Mac are fully compatible with Visual Studio on Windows, and vice versa. This means if you use one tool and want to change to the other, you can. It also means you can load anyone else's solution, regardless of what tools were used to create it.

The first thing to do is to ensure the Android app is the startup project, so right-click it and select Set as Startup Project. Once this is selected, you'll see options for choosing the device to run it on in the menu.

On Visual Studio for Mac (on the left in figure 2.18), you'll see two drop-down menus in the top left, and from the second one you can choose the device to run on—

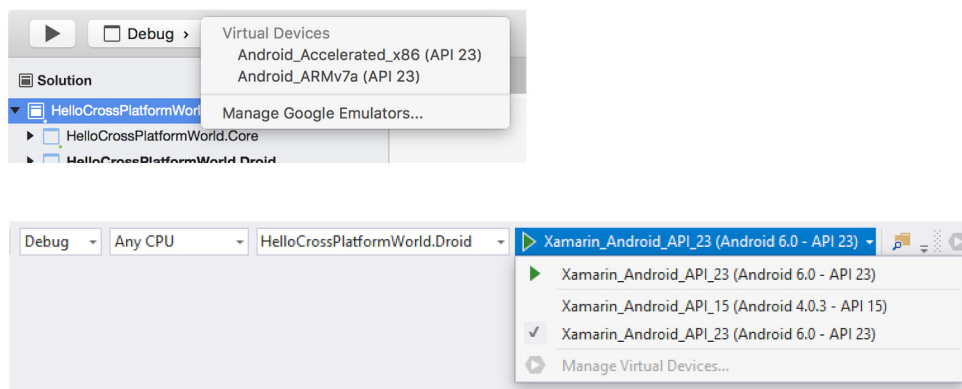


Figure 2.18 The Android device selection menus

an emulator or a physical device (if you have one plugged in). Visual Studio uses the emulators from Google and installs and configures two of these by default. You should select the Accelerated x86 emulator, as this will be faster on a Mac; ARM-based emulators run about 10 times slower than the x86 version.

Visual Studio for Windows installs the Visual Studio Emulator for Android as part of its installer (assuming the option was ticked when you ran the installer), and it will configure a few of these inside Visual Studio for you to use.

These emulators come in different hardware types and different Android OS versions. You'll need to use an x86-based emulator (it's much faster than the ARM version), and all the x86 emulators are basically the same in terms of hardware, just using a different version of the Android OS. For now, just choose the latest OS version, and run the app either by clicking the Run button on the toolbar, or by choosing Run > Start Debugging on Visual Studio for Mac or Debug > Start Debugging on Windows. Sit back and relax as your app is built and the emulator is launched.

Be aware that the first time your app builds, it will take a very long time—there are a number of SDK files that Xamarin needs to download in order to build an Android app, and it downloads these the first time your app is built with no feedback in the output except that it's building. Don't kill the build—if you do, you may have to manually clean up half-downloaded zip files. If you do get errors about corrupt zip files, you can find information on how to fix them in Xamarin's Android Troubleshooting guide at <http://mng.bz/MKSQ>.

DON'T RUN MORE THAN ONE ANDROID EMULATOR Android emulators can be a bit fussy sometimes, as they run inside virtual machines. If you try to run more than one, they can freeze up and not start. If you ever get this happening—the emulator screen stays black and nothing happens—quit it and close all other emulators you have running, and try again.

This app doesn't do much. It just shows off the very basic features of MvvmCross. If you change the text in the text box, the label below will update to reflect this. We'll dive into what's happening a bit more later, but for now you're over the first hurdle—you have an app that runs. Let's crack on with iOS.

IOS

Building and running the iOS app is very similar to Android. First, ensure the iOS app is the startup project, just as you did for the Android app.

Next you need to select the device to run on. This is slightly different from Android. Android always builds the same code for emulators and physical devices, so all you need to do is choose the device. On Visual Studio for Mac, this is the same—from the drop-down menu choose a simulator or a physical device if one is available (on the left in figure 2.19). From here, select the iPhone simulator of your choice, though a recent one is always good.

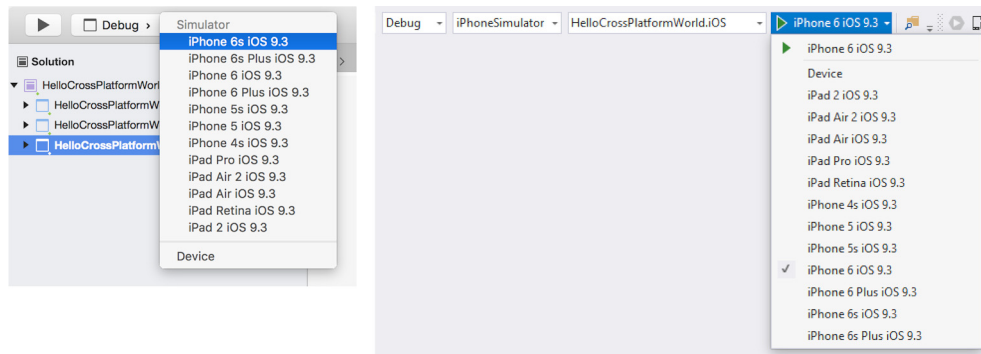


Figure 2.19 The iOS device selection menus

Visual Studio for Windows is similar, though it breaks this out into two drop-down menus—one to choose either a physical device or a simulator, and another that shows the available devices or simulators (on the right in figure 2.19). In this case, choose iPhoneSimulator from the first menu, and select the simulator of your choice from the second.

Once the appropriate simulator is selected, run the app. If you're using Visual Studio for Mac, the simulator will run on your Mac. If you're using Windows, the simulator will either launch on your Mac, or on your Windows PC if you have the iOS simulator for Windows installed.

Once the simulator fires up, you'll see the basic MvvmCross sample app. This is identical to the Android app—edit the text and the label updates to match. Awesome—your Xamarin app is running on iOS without any extra work.

2.5 Is this really a cross-platform app?

One of the big upsides of Xamarin is being able to write cross-platform apps—separate apps for each platform with shared core code. The question on your lips now is probably “is this what we’re seeing here?” The answer is yes! The iOS and Android projects have part of the application layer (the code to actually run an application), and the view layer (the UI is defined in platform-specific code), but the core of everything is in a shared core project. This is pretty simple to prove, so let’s make a simple code change to demonstrate it.

In the apps you’ve run on Android and iOS, you have a text box with “Hello MvvmCross” in it, and a label that matches this text, updating whenever the text changes. Let’s now change the initial value of this text.

In the Core project there’s a ViewModels folder (figure 2.20), and inside this is a view-model class called `FirstViewModel` (in the `FirstViewModel.cs` file). Look at the `hello` field, and you’ll see it’s initialized to `Hello MvvmCross`. Update this to be `Hello Xamarin in Action` as follows.

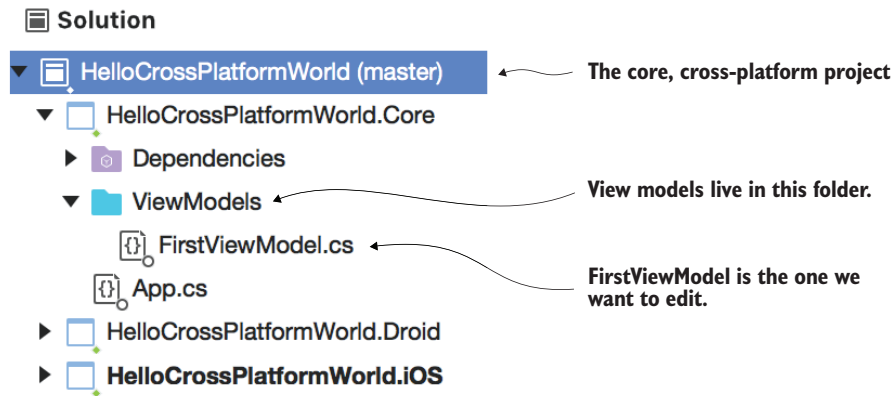


Figure 2.20 The structure of the core project showing the location of the `FirstViewModel` class

Listing 2.2 Updated `hello` field in `FirstViewModel`

```
string hello = "Hello Xamarin in Action";
```

This is a one-line code change in one file in shared code. If you build and run the Android and iOS apps now, you’ll see that both have the new text showing in the text box and label, as in figure 2.21.

The apps look the same and work the same. The only difference is the original string value that’s shown on startup.

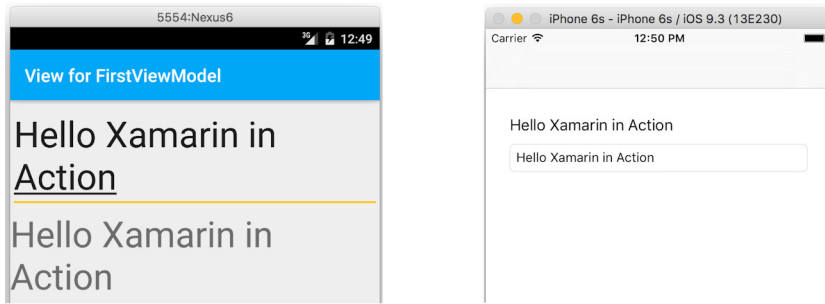


Figure 2.21 Both sample apps showing the new text, changed by changing only one line of code

So how does this all work? Let's look at this solution to see how it fits into our layers. This app has two views, one on iOS and one on Android, a view model in shared cross-platform code, and a string that acts as a model (figure 2.22).

Before we can go into much more detail about what's happening here, there's a lot more about MVVM we need to discuss. In the next chapter we'll take that deeper dive into MVVM, and once you've seen in more depth how MVVM works we'll look in more detail at the code we've just built.

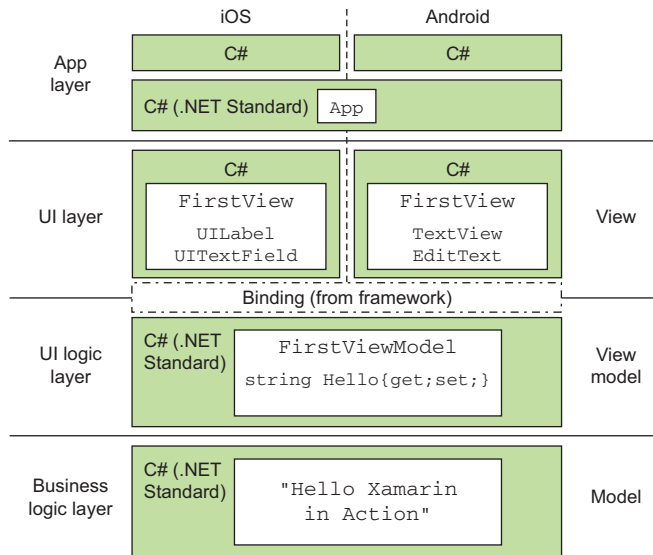


Figure 2.22 Our UI code is in the platform-specific UI layer; our core class with its string property is in the cross-platform business logic layer.

Summary

In this chapter you learned that

- A number of design patterns have evolved over time to help in making better UI applications. The latest incarnation of these, MVVM, is very well suited to building Xamarin apps, as it maximizes the amount of cross-platform code in our apps.
- A cross-platform Xamarin app is not totally cross-platform. Instead it's an app where all platforms are written in the same language (C#) so that you can share a large portion of your code.
- Cross-platform code is written in .NET Standard libraries that provide access to a subset of .NET that works on all platforms.
- The MVVM pattern consists of three layers. You can write two of these layers, the model and the view model, once inside a .NET Standard library and share the code between your iOS and Android apps.

You also learned how to

- Use an extension in Visual Studio to create a cross-platform Xamarin app, with projects for iOS and Android, and a .NET Standard library core project for shared code.
- Run these apps inside the iOS simulator and Android emulator.

In this chapter, you'll start to build a Flutter app with a focus on the (UI) user interface. You'll explore the elements that help you make your Flutter apps beautiful including layout, themes, styling, and Flutter's built-in widgets.

Flutter UI: Important widgets, theme, and layout

This chapter covers

- Your first Flutter app
- User interface in Flutter
- Layout widgets
- Themes and styling
- Custom form elements
- `builder` patterns

Flutter is more than a framework; it's also a complete SDK. And perhaps the most exciting piece of this SDK to me, as a web developer, is the massive library of built-in widgets that make building the front end of your mobile app easy.

This chapter is all about the user interface and making an app beautiful. It includes exploring some of the widgets built into Flutter, as well as layout, styling, and more.

Figure 4.1 shows the app I'll use to explain the UI in Flutter.

Weather app screenshots



Figure 4.1 Screenshots of the weather app

In this chapter, I look at these high-level categories:

- Structural widgets that outline the app.
- Theme and styling, which this app is heavy on. Here, you set the custom color scheme and look at the `MediaQuery` class to help with styling.
- Next, you look in the broad category of widgets that help with layout, including building-block widgets like `Table`, `Stack`, and `BoxConstraint`, as well some fantastic convenience widgets like `TabBar`, which provides entire features for free.
- Finally, you move on to a section about layout widgets—specifically, `ListView`. This widget can be scrollable and uses something called the builder pattern.

Before I get started, there are a couple of caveats and disclaimers that I'd like to mention:

- There's no way that a single book or app could (or should) cover all (or even most) of Flutter's built-in widgets and features. The purpose of this minibook is learning, and although you won't learn about every single widget, you do learn how to find and use what you're looking for when the time comes. Flutter's documentation is among the best I've ever seen, and all the widgets' descriptions are robust. The Flutter team is hard at work adding more widgets and plug-ins every day.¹
- A lot of code in the app doesn't have anything to do with Flutter. Models are models, for example, regardless of the language and framework you're using. I won't leave you wondering, though. I'll point out where the relevant code is when the time is right; I won't walk through it line by line.

¹ You can find all the widgets and their descriptions in the official Widget Catalog: <https://flutter.dev/docs/development/ui/widgets>

4.1 Setting up and configuring a Flutter app

In the `flutter_in_action` repository² is a directory called `chapter_4_5_6`. This repository is where this chapter will begin.

Listing 4.1 Weather app file structure

```
weather_app
├── README.md
├── lib
│   ├── blocs
│   │   └── forecast_bloc.dart
│   ├── main.dart
│   ├── models
│   │   └── // models...
│   ├── page
│   │   └── // pages...
│   ├── styles.dart
│   ├── utils
│   │   └── // many utils files...
│   └── widget
│       └── // all the custom widget's for this app
├── pubspec.lock
└── pubspec.yaml
```

This business logic component (bloc) initializes some data for the app. All the data in this app is fake. It's randomly generated in the `utils/generate_weather_data` file.

This app uses a lot of colors. I made an `AppColors` class so that referencing all the colors would be clean and easy. This class extends `Color` from `dart:ui`, which is the class that Flutter uses to define `Color`. This file is covered in this chapter.

This line is the app's entry point.

There's new Flutter-specific configuration in this yaml file.

I'll begin in the `pubspec.yaml` file.

4.1.1 Configuration: `pubspec` and `main.dart`

All Dart applications require a `pubspec.yaml` file, which describes some configuration for the app. Dart has a build system that builds your app, and the first thing it does when you run your app is look for the `pubspec` file. When you're building a Flutter app, some specific configuration items need to exist in the `pubspec` file for the app to run:

```
// weather_app/pubspec.yaml
name: weather_app
description: Chapters 4 - 6, Flutter in Action by Eric Windmill
version: 1.0.0+1

environment:
  sdk: ">=2.0.0-dev.68.0 <3.0.0"

dependencies:
  flutter:
    sdk: flutter

flutter:
  uses-material-design: true
  fonts:
    - family: Cabin
```

This app is going to use material design. This flag tells Flutter to include the material package.

When importing a font, give it a family name, which you'll use to reference the font throughout your app.

² The repository is at https://github.com/ericwindmill/flutter_in_action_public, or you can download the source code from Manning's website.

```

fonts:
- asset: assets/fonts/Cabin-Regular.otf
- asset: assets/fonts/Cabin-Bold.otf
// ...

```

List all the variations of this font you want to use.

Along with declaring assets and importing libraries, this information is all you need for your Flutter pubspec file.

TIP If you're writing an app that's heavy on iOS-style widgets, you can include an additional flag to import iOS icons.

Along with the pubspec file, your app must have an entry point: a file that includes a main function. In Flutter apps, the entry point is, by convention, a file called `main.dart`.

Check out `weather_app/main.dart`. The `main()` function runs the app, as in every Dart program, but it's also useful for setting up some configuration for the app before the app runs.

Listing 4.2 The weather app's main function

```

void main() {
  AppSettings settings = new AppSettings();

  // Don't allow landscape mode
  SystemChrome.setPreferredOrientations(
    [DeviceOrientation.portraitUp, DeviceOrientation.portraitDown])
    .then((_) => runApp(new MyApp(settings: settings)));
}

```

Create an instance of `AppSettings`, which is a class I made to fake persisting user settings in the weather app.

The main function also talks to the `SystemChrome` class, which is the subject of the next section.

You must include a call to `runApp` and pass it your root-level widget!

4.1.2 *SystemChrome*

`SystemChrome` is a Flutter class that exposes some easy methods to control how your app displays on the native platform. This class is one of the only classes you'll use to manipulate the phone itself (unless you're writing plug-ins, which are outside the scope of this chapter.)

In this app, I'm using `SystemChrome.setPreferredOrientations` to restrict the app to portrait mode. This class also exposes methods that control what the phone's overlays look like. If you have a light-colored app, for example, you can ensure that the time and battery icons on your phone's status bar are dark (and vice versa):

```

void main() {
  AppSettings settings = new AppSettings();

  // Don't allow landscape mode
  SystemChrome.setPreferredOrientations(
    [DeviceOrientation.portraitUp, DeviceOrientation.portraitDown])
    .then((_) => runApp(MyApp(settings: settings)));
}

```

Use `then(callback)` to asynchronously execute code when a `Future` completes. This line is also the entry point of your app. Passing a widget into `runApp` is always the entry point.

The `SystemChrome` class is something you'll set once and then forget. I'm showing it to you up front so that you're aware of it, but there's no need to spend too much time on it. If you're curious, you can find more on it here: <https://api.flutter.dev/flutter/services/SystemChrome-class.html>.

Before moving on, I need to address the `then` function used in that code.

EXAMPLE 1. JUST IN TIME: DART FUTURES

You won't get far into Dart without seeing some `async` methods here and there. A `Future` is the foundational class of all `async` programming in Dart.

Futures are a lot like receipts at a burger quick-serve restaurant. You, the burger orderer, tell the employee that you want a burger. The server at the restaurant says, "Okay, here's a receipt. This receipt guarantees that sometime in the future, I will give you a burger as soon as it's ready."

So you, the caller, wait until the employee calls your number and then delivers on the guarantee of a burger. The receipt is the `Future`. It's a guarantee that a value *will* exist but isn't quite ready.

Futures are *then-able*, so when you call a `Future`, you can always say `myFuture.Method().then((returnValue) => ... do some code ...)`;

`Future.then` takes a callback, which is executed when the `Future` value resolves. In the burger restaurant, the callback is what you decide to do with the burger when you get it (such as eat it). The value passed into the callback is the return value of the original `Future`:

```
Future<Burger> orderBurgerFromServer() async {
  var burger = await prepareBurger();
  return burger;
}
```

prepareBurger is likely going to take time (for the burger to cook). When it's done being prepared, return it.

```
orderBurgerFromServer().then((Burger burger) => eatBurger(burger));
```

The callback, `(Burger burger) ? eatBurger(burger)`, will be passed the return value of `orderBurgerFromServer` without the `Future` when the `Future` has finished processing.

The `orderBurgerFromServer` method returns the type of `Future`, with the subtype of `Burger` (which, in a program, looks like `Future<Burger>`). So `orderBurgerFromServer` processes and *then* the callback is called, with the return value passed as an argument.

Asynchronous programming is a big topic; this excerpt is meant to be an introduction. Don't get too bogged down here.

That's it for app configuration. I'll be talking about widgets for the rest of the chapter, starting with the top-level widget: `MyApp` in the `weather_app/main.dart` file.

4.2 Configuring structural widgets and more

You'll likely use a few convenience widgets in every Flutter app you ever build. These widgets provide configuration and structure to your app with little work on your part. In this section, I explain the `MaterialApp`, `Scaffold`, `AppBar`, and `Theme` widgets.

4.2.1 **MaterialApp** widget

The `MaterialApp` widget provides a ton of benefits that affect its entire subtree. This section is the beginning of many widgets that provide helpful functionality for free.

`MaterialApp` is an extension of the generic top-level widget provided by Flutter: `WidgetsApp`. `WidgetsApp` is a convenience widget that abstracts away several features that are required for most mobile apps, such as setting up a navigator and using an appwide theme. The `WidgetsApp` is completely customizable and makes no assumptions about default configuration, style, or structure of the app's UI. Although the widget abstracts away some difficult pieces of functionality in your app, it requires more work to set up than the `MaterialApp` or `CupertinoApp`. I won't discuss the `WidgetsApp` here because it's a base class for the other two and not intended to be used directly.

`MaterialApp` is even more convenient than `WidgetsApp`, adding `Material` design-specific functionality and styling options to your app. It doesn't only *help* set up the `Navigator`, but also does that for you. If you use the `MaterialApp` widget, you don't have to worry about implementing the animations that happen when a user navigates between pages. This widget takes care of that for you. It also allows you to use widgets that are specifically in the `Material` widgets collection, and there are plenty of those.

The app is called a `Material` app because it leans on `Material` style guidelines. Page animations from one route to another, for example, are designed as you'd expect on an Android device, and all the widgets in the `Material` widget library have the standard Google look and feel. These facts can be concerns if you have a specific design system that isn't similar to `Material`. But I don't see a drawback to using `MaterialApp` even if you don't want to use `Material` design guidelines. Your theme is still fully customizable. (In fact, in this app, you'll build an app that doesn't look `Material` at all.) You can overwrite routing animations, and you don't have to use the widgets in the `Material` library. The `MaterialApp` widget provides quite a bit of convenience, but everything is reversible.

In the weather app, the `MaterialApp` widget is used in the `build` method of the `MyApp` widget—the convention used in every Flutter app. This is the code in the main file in the app, showing the main function again as well as the root widget:

Listing 4.3 The top-level widget in `main.dart`

```
// weather_app/main.dart
void main() { (1)
  AppSettings settings = new AppSettings();

  // Don't allow landscape mode
  SystemChrome.setPreferredOrientations(
    [DeviceOrientation.portraitUp, DeviceOrientation.portraitDown])
    .then((_) => runApp(MyApp(settings: settings)));
}

class MyApp extends StatelessWidget {
  final AppSettings settings;
}
```

The entry point of your app.

`MyApp` is a widget, like everything else.

`runApp` is being passed `MyApp`, the root of your Flutter app.

```
const MyApp({Key key, this.settings}) : super(key: key);

@override
Widget build(BuildContext context) {
  // ...
  return MaterialApp( (4)
    title: 'Weather App',
    debugShowCheckedModeBanner: false,
    theme: theme,
    home: PageContainer(settings: settings),
  );
}
```

The `build` method of `MyApp` returns a `MaterialApp` as the top-level app.

Again, this is standard in Flutter apps. Your top-level widget is one that you write yourself—in this case, `MyApp`. That widget turns around and uses `MaterialApp` in its `build` method, providing a widget in your app in which you can do additional setup.

Looking at that `build` method again, the following arguments are being passed to `MaterialApp`.

Listing 4.4 The `build` method of the `MyApp` widget

```
//
@override
Widget build(BuildContext context) {
  // ...
  return MaterialApp( (1)
    title: 'Weather App',
    debugShowCheckedModeBanner: false,
    theme: theme, (3)
    home: PageContainer(settings: settings),
  );
}
```

Return a `MaterialApp`.

One of the aspects of your app that `MaterialApp` takes care of for you is the appwide Theme (covered shortly).

This flag removes a banner that is shown when you're developing your app and running it locally. I turned it off only so the screenshots in this book would be cleaner.

`home` represents the home page of your app. It can be a widget. `PageContainer` is a widget written for the weather app that will be covered later.

4.2.2 Scaffold

Like the `MaterialApp` widget, a `Scaffold` is a convenience widget that's designed to make applications that follow Material guidelines as easy as possible to build. The `MaterialApp` widget provides configuration and functionality to your app. `Scaffold` is the widget that gives your app *structure*. You can think of the `MaterialApp` widget as being the plumbing and electricity of your app, whereas `Scaffold` is the foundation and beams that give your app structure.

Like the `MaterialApp`, a `Scaffold` provides functionality that you'd otherwise have to write yourself. Again, even if you have a highly customized design style that's not Material at all, you'll want to use `Scaffold`.

Per the Flutter docs, a `Scaffold` defines the basic Material design visual layout, which means that it can make your app look like figure 4.2 pretty easily.

`Scaffold` provides functionality for adding a *drawer* (an element that animates in from one of the sides and is commonly used for menus) and a *bottom sheet*, which is

■ Scaffold example

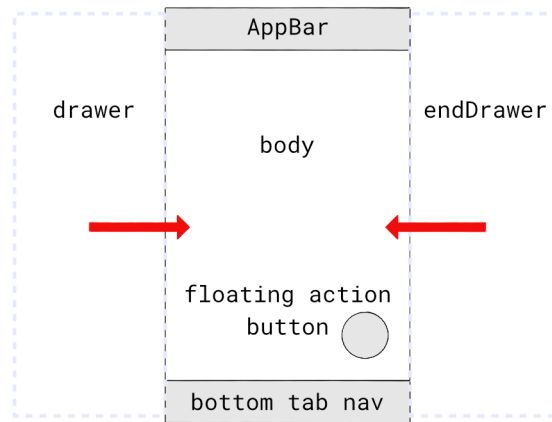


Figure 4.2 Diagram of the most important `Scaffold` widget properties

an element that animates into view from the bottom of the screen and is common in iOS-style apps. And unless you configure it otherwise, the `AppBar` in a `Scaffold` is automatically set up to display a menu button in the top-left corner of your app, which will open the drawer; when you aren't on a screen that has a menu, it changes that menu button to a back button. Those buttons are already wired up and work as expected on tap.

It's important to know, though, that you can choose the features you want and the ones you don't. If your app doesn't have a drawer-style menu, you can simply not pass it a drawer, and those automatic menu buttons disappear.

The `Scaffold` widget provides many optional features, all of which you configure from the constructor. Here's the constructor method for the `Scaffold` class:

Listing 4.5 `Scaffold` full property list

```
// From Flutter source code. Scaffold constructor.
const Scaffold({
  Key key,
  this.appBar,
  this.body,
  this.floatingActionButton,
  this.floatingActionButtonLocation,
  this.floatingActionButtonAnimator,
  this.persistentFooterButtons,
  this.drawer,
  this.endDrawer,
  this.bottomNavigationBar,
  this.bottomSheet,
  this.backgroundColor,
  this.resizeToAvoidBottomPadding = true,
  this.primary = true,
}) : assert(primary != null), super(key: key);
```

I wanted to show this code so you can see that none of these properties is marked as `@required`. You can use an `AppBar`, but you don't have to. The same is true of drawers, navigation bars, and so on. For this app, I used only the `AppBar`. The point is (again) that even if you're building an app that you don't want to look Material, the `Scaffold` widget is valuable, and I recommend using it.

In the weather app, you can see the `Scaffold` in the `ForecastPage` widget.³ The part I want to point out right now is at the bottom of the file: the return statement of the `ForecastPageState.build` method. I only want to show you that the `Scaffold` is a widget, and like many widgets, most of its arguments are optional, making it highly customizable.

```
// weather_app/lib/page/forecast_page.dart
return Scaffold(
  appBar: // ... preferred sized widget
  body: // ... gesture detector
```

You can pass a widget to the `appBar` argument, and that widget is placed at the top of the screen of your app. (`AppBar` and `PreferredSize` are covered in the next section.)

body is the argument on the `Scaffold` that represents the the main portion of the screen. If there's no `appBar`, the `body` is the entire screen for all intents and purposes.

Recall the `Scaffold` constructor method that I showed earlier, which had more than ten named arguments. Here, I'm using only two. The point is that these widgets give you a lot but are highly customizable. In the next section, I give concrete examples of how the `Scaffold` is used in the weather app.

4.2.3 AppBar widget

The `AppBar` widget is yet another convenience widget that gives you all kinds of features for free. The `AppBar` is typically used in the `Scaffold.appBar` property, which fixes it to the top of the screen at a certain height.

The most notable feature of the `AppBar` provides navigation features for free. The `AppBar` automatically inserts a menu button if the `AppBar`'s parent is a `Scaffold` and the `drawer` argument isn't null. And if the `Navigator` of your app detects that you're on a page that can navigate back (like a browser's back button), it automatically inserts a back button.

In the `AppBar` widget, multiple parameters expect widgets as arguments. These arguments correspond to specific positions within the `AppBar` (figure 4.3).

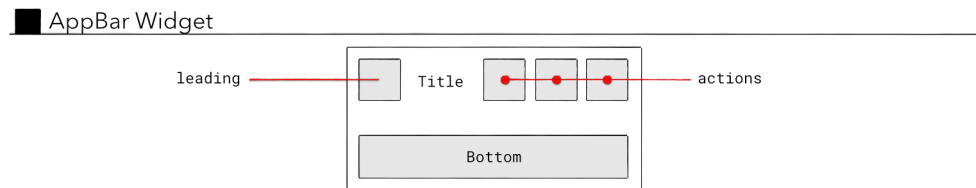


Figure 4.3 Most important properties of the `AppBar` widget

³ The `ForecastPage` is in the directory at `weather_app/lib/page/forecast_page.dart`.

The property that handles these menu buttons and back buttons is called the leading action; it can be configured with the `AppBar.leading` property and the `AppBar.automaticallyImplyLeading` property. Suppose that you don't want that menu button to appear. You can set `AppBar.automaticallyImplyLeading` to `false` and then pass the leading argument to whatever widget you want. This argument attempts to place that widget on the far-left side of the `AppBar` (figure 4.4).

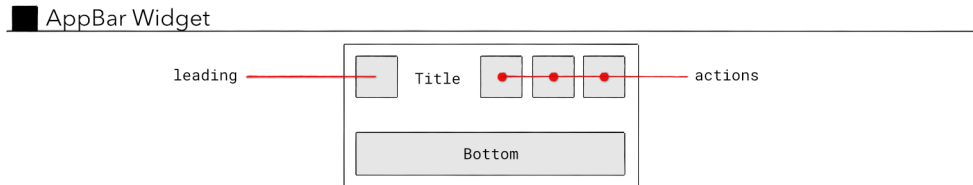


Figure 4.4 Most important properties of the `AppBar` widget

4.2.4 Preferred Size widget

In Flutter, widgets' sizes are generally constrained by the parent. When a widget knows its constraints, it chooses its own final size. The importance of this concept can't be understated when it comes to UI. The constraints passed to a widget by its parent tell the widget how big it *can* be, but isn't concerned with the widget's final size. The advantage of this system (as opposed to HTML, for example, in which elements control their own constraints), is flexibility. It allows Flutter to make intelligent decisions about what widgets should look like and removes some of that burden from the developer.

There are some cases in which flexibility isn't desirable, though. You may want to set explicit sizes for widget. A good example is the `AppBar`.

The `AppBar` class extends a widget called `PreferredSizeWidget`. This widget allows you to define an explicit height and width, and Flutter does its best to make sure it's that size when the screen renders. This widget isn't commonly used, but it serves as an example for a valuable lesson.

The `Scaffold.appBar` property expects a widget that's specifically of the `PreferredSizeWidget` class because it wants to know the size of the `AppBar` before it sets constraints.

In this app, I use a `PreferredSize` widget directly rather than using an `AppBar` in the `Scaffold.appBar` argument. The practical application of this approach is that you can wrap any widget in a `PreferredSize` and use it in place of the Material-specific `AppBar` widget. The lesson, again, is that Flutter widgets are fleshed out by default but are also customizable.

Listing 4.6 Using `PreferredSize` in a `Scaffold`

```
// weather_app/lib/page/forecast_page.dart -- line ~217
return Scaffold(
  appBar: PreferredSize( ←
```

Use the `PreferredSize` widget to use any arbitrary widget in the `Scaffold.appBar` property. This approach works because the `AppBar` widget extends `PreferredSize`, and the `Scaffold.appBar` expects a `PreferredSize` rather than an `AppBar` specifically.

The second required argument is its child.

```
preferredSize: Size.fromHeight(ui.appBarHeight(context)),
child: ...
),
),
```

PreferredSize needs two bits of information as arguments. The first is its `preferredSize`, which takes a `Size` class. The `Size` class defines a height and width.

Now that I've talked about `PreferredSize` at a high-level, I'll use the weather app for a concrete example. The point of using a `PreferredSize` for the app is that the built-in `AppBar` widget doesn't provide a way to animate its colors by default. If you've poked around the app, you may notice that the colors change as the time of day changes, which required a custom widget called `TransitionAppBar`. I've wrapped it in a `PreferredSize` so that the `Scaffold` accepts it in the `appBar` argument.

Named imports in Dart

You may have noticed in previous examples that I'm calling `ui.appBarHeight`, but `ui` doesn't seem to be a class. `ui` refers to the `utils` file with this name: `import 'package:weather_app/utils/flutter_ui_utils.dart' as ui;`

This name requires you to prefix any class, method, or variable in that library with `ui`.

4.3 Styling in Flutter and Theme

Styling your app in Flutter can be simpler than you'd expect. If you're diligent about setting up a `Theme` when you start your Flutter app, you shouldn't have to do much work to keep the app looking consistent. The `Theme` widget allows you to set many default styles in your app, such as colors, font styles, and button styles. In this section, you look at how to use the `Theme`.

Along with `Themes`, I talk about other important pieces of styling in Flutter: `MediaQuery`, fonts, animations, and Flutter's `Color` class.

4.3.1 Theme widget

The `Theme` widget allows you to declare styles that in some instances will be applied throughout your app automatically. In instances in which your styles aren't applied or need to be overridden, the `Theme` widget is accessible anywhere in your widget tree.

To give you an idea of the many color-related styles that your `Theme` can control, here are some (but not all) properties you can set on the widget that will permeate throughout the app.

Properties that affect all the widgets in your app:

- `brightness` (which sets a dark or light theme)
- `primary Swatch`
- `primary Color`
- `accent Color`

These are some properties that control specific features:

- `canvas Color`
- `scaffoldBackgroundColor`
- `dividerColor`
- `cardColor`
- `buttonColor`
- `errorColor`

That's only 6 of about 20 that are available, for colors. But there are almost 20 more arguments you can pass to `Theme` that set defaults for fonts, page animations, icon styles, and more. Some of those arguments expect classes themselves, which have their own properties, offering even more customizations for your app. The point is, the `Theme` widget is robust and can do a lot of the heavy lifting for you when it comes to styling.

Although this level of theming is nice, it can be overwhelming to think about every last one of those properties. Flutter thought about them, though. If you're using the `MaterialApp` widget at the root of your app, every property has a default value, and you can elect to override only the properties that you care about. `Theme.primaryColor`, for example, affects almost all widgets in your app. It changes the color of all widgets to your brand's color. In the app I'm building at my current job, we have an app that looks completely on brand (and not material), and we set only properties on our theme.

In other words, you can be as granular or hands-off as you'd like to be. I've said it many times, but one of the aspects of Flutter that you should take advantage of is the fact that it does so much for you *until* you decide that you need more control.

Next, you look at how you can implement a `Theme` in your Flutter app.

4.3.2 Using Themes in your app

The class you use to configure your `Theme` is called `ThemeData`, and to add a `Theme` to your app, you'd pass a `ThemeData` object to the `MaterialApp.theme` property of your app. You can also create your own `Theme` widget and pass it a `ThemeData` object. `Theme` is a widget, so you can use it anywhere you can use any widget! The theme properties that any given widget uses are inherited from the *closest* `Theme` widget up the tree. In practice, you can create multiple `Theme` widgets throughout your app, and these widgets will override the top-level theme for everything in that subtree.

Here's an example of using `ThemeData` in real life.

Listing 4.7 `ThemeData` in the weather app

```
// weather_app/lib/main.dart
var theme = ThemeData(
  fontFamily: "Cabin",
  primaryColor: AppColor.midnightSky,
  accentColor: AppColor.midnightCloud,
  primaryTextTheme: Theme.of(context).textTheme.apply(
    bodyColor: AppColor.textColorDark,
    displayColor: AppColor.textColorDark,
  ),
);
```

This is how you tell Flutter to use the font that you've told it about in the `pubspec.yaml` file.

`AppColor` is a class I created because this app will use almost all custom colors. You can find the class in `styles.dart`.

`apply` is a method on theme classes that copies the current theme but changes the properties you've passed it.


```

textTheme: Theme.of(context).textTheme.apply(
  bodyColor: AppColor.textColorDark,
  displayColor: AppColor.textColorDark,
),
);

```

The other case in which you'd want to use the `ThemeData` is when you want to set a style property explicitly. You may want set a container's background to be the accent Color of the Theme, for example. Anywhere in your app, you can grab that Theme data thanks to `BuildContext`.

`BuildContext` provides information about a widget's place in the widget tree, including information about certain widgets that are higher in the tree, such as Theme. If you want to know the accent Color of the theme for any given widget, you can say "Hey, `BuildContext`, what's the accent color assigned to the `ThemeData` that's closest up the tree from this widget?" In the next section, I explain that sentence further and make it less abstract.

4.3.3 *MediaQuery and the of method*

If you came from the web like me, you may find writing styles in Flutter to be cumbersome at first, particularly spacing and layout. On the web, you use CSS, and CSS has many units of measurement that you can use anywhere sizing comes into play. CSS has standard pixels, units of measurement based on the percentage of space the element can take up, and units of measurement based on the size of the viewport.

Flutter has only one unit measurement: the logical pixel. The consequence is that most of the layout and sizing problems can be solved with math. Much of the math you'll want to do is based on screen size. Suppose that you want a widget to be one-third the width of the screen. Because Flutter has no percentage unit of measurement, you have to grab the screen size programmatically by using the `MediaQuery` widget.

`MediaQuery` is similar to Theme in that you can use the `BuildContext` to access it anywhere in the app via a method of the `MediaQuery` class called `of`. The `of` method looks up the tree, finds the nearest `MediaQuery` class, and gives you a reference to that `MediaQuery` instance anywhere in your app. A few widgets built into Flutter provide an `of` method.

NOTE You can create widgets that have their own `of` method, and you can access the state of those widgets anywhere in the tree. For now, all that matters is that certain built-in widgets can be accessed anywhere in your app.

As mentioned, the `MediaQuery` class is great for getting size information about the entire screen on which your app is rendered. You access that information by calling the static method `MediaQuery.of(context).size`. This method returns a `Size` object with the device's width and height. Let me break that down a bit more.

Because `of` is a static method, you call it directly on the `MediaQuery` class rather than on an instance of the class. Also, the `of` method can provide the `MediaQuery` class only if

it knows the `BuildContext` in which of is called. That's why you pass it context. Finally, `size` is a getter on the `MediaQuery` class that represents the device's width and height.

Look at this example of using `MediaQuery`. After grabbing the information, you can use it to determine the size of a widget, based on the screen size. To get 80 percent of the width of the phone, for example, you could write

```
var width = MediaQuery.of(context).size.width * 0.8;
```

Again, a widget's build context gives Flutter a reference to that widget's place in the tree. The `of` method, which always takes a context regardless of which object it's defined on, says "Hey, Flutter, give me a reference to the nearest widget of this type in the tree above myself."

`MediaQuery` is the first place you should look if you're trying to get specific information about the physical device your app is running on or if you want to manipulate the device. You can use it to

- Ask whether the phone is in portrait or landscape orientation.
- Disable animations and invert colors for accessibility reasons.
- Ask the phone whether the user has the text-size factor scaled up.
- Set the padding for your entire app.

In the weather app, I use `MediaQuery` to ensure that widgets are scaled to the proper sizes based on the size of the screen. The following section shows an example.

4.3.4 *ScreenAwareSize* method

Recall this code from the Scaffold in the `ForecastPage`:

Listing 4.8 Using `PreferredSize` in a Scaffold

```
// weather_app/lib/page/forecast_page.dart -- line ~217
return Scaffold(
  appBar: PreferredSize(
    preferredSize: Size.fromHeight(ui.appBarHeight(context)),
    child: ...
  ),
),
```

The method `Size.fromHeight` is a constructor on the `Size` class that creates a `Size` object with the given height and an infinite width. That leaves the `ui.appBarHeight` method.

NOTE `ui` is a package alias for a local file. It's imported as `import 'package:weather_app/utils/flutter_ui_utils.dart' as ui;` at the top of the `forecast_page.dart` file.

In the file at `weather_app/lib/utils/flutter_ui_utils.dart`, you'll find the code that defines the function `ui.appBarHeight(context)` from the preceding code snippet.

Listing 4.9 Screen-aware sizing methods

```
// weather_app/lib/wutils/flutter_ui_utils.dart

final double kToolbarHeight = 56.0;
double appBarHeight(BuildContext context) {
  return screenAwareSize(kToolbarHeight, context);
}

const double kBaseHeight = 1200.0;
double screenAwareSize(double size, BuildContext context) {
  double drawingHeight = MediaQuery.of(context).size.height -
    MediaQuery.of(context).padding.top;
  return size * drawingHeight / kBaseHeight;
}
```

56.0 is the default heights of toolbars in Flutter.

I'm passing context into the method so I can use the context to get MediaQuery information.

The bulk of the functionality is in this line. I'm using the context to get some information about the app and screen size.

`MediaQuery.of(context).size` returns a size that represents the screen size. `MediaQuery.of(context).padding` returns a `Padding` that gives padding details for the app itself—that is, the padding between the edge of the device screen and the top-level widget.

The purpose of these methods is to provide accurate sizing for the `PreferredSize` widget (and to see the `MediaQuery` class in action). These methods are mapping the height of the `appBar` in the weather app to its appropriate size on *any given screen*. If the “average” screen is 1200 pixels tall, and on that screen, the `appBar` is 56 pixels high, these functions give you the equivalent height for the `appBar` on a screen of any size.

NOTE The built-in `AppBar` widget is smart itself, but you need a custom widget because you’ll eventually add custom style and animation to it.

Again, this function is used in the `ForecastPageState.Scaffold`:

Listing 4.10 Using `PreferredSize` in a `Scaffold`

```
// weather_app/lib/page/forecast_page.dart -- line ~217
return Scaffold(
  appBar: PreferredSize(
    preferredSize: Size.fromHeight(ui.appBarHeight(context)),
    child: ...
  ),
),
```

This bit of code is going to tell the `Scaffold` (the preferred size’s parent) how big the `AppBar` wants to be. Specifically, it tells Flutter to create a `Size` instance from a height that’s appropriate for any screen.

This example is specific, to be sure. The `appBarHeight` method is useful only for the `AppBar`. The `screenAwareSize` method could be reused. In any case, the point is to show off the `MediaQuery` widget, which you’ll likely use quite a bit for styling and layout.

For now, that’s it for the `MediaQuery` class.

4.4 Using common layout and UI widgets

This section is devoted to individual layout widgets and widgets that represent physical UI elements. In Flutter, everything is a widget, so I'll never stop talking about widgets, but after this section, I talk about complex widgets that *do stuff* rather than *show stuff*. In particular in this chapter, I cover Stack, Table, and TabBar, three built-in widgets that make layout easier.

4.4.1 Stack widget

Stack is what it sounds like. It's used to layer widgets or stack them. Its API can be used to tell Flutter exactly where to position widgets relative to the stack's border on the screen. (If you come from the web development world, this is much like `position: fixed` in CSS.) In this section, you use it to make a fancy background that reflects the time of day and current weather via images. The color of the sun is animated to change as the time of day changes, and it also shows clouds and weather conditions (figure 4.5).

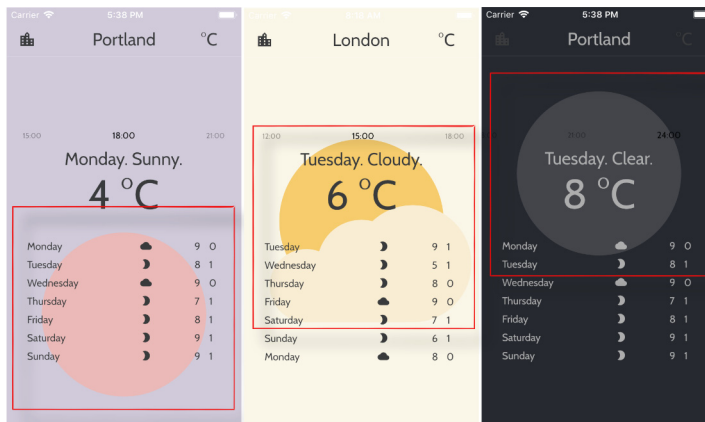


Figure 4.5 The background of the weather app

The sun, the clouds and the content are different widgets stacked on top of one another. All the children of a Stack are *positioned* or (by default) *nonpositioned*. Before I talk about the idea of being positioned, it's important to understand the Stack's default behavior. The widget treats nonpositioned children the same way that a column or row treats its children. It aligns its children widgets by their top-left corners and lays them out one after the other next to each other. You can tell the widget which direction to align in with the `alignment` property. If you set the alignment to `horizontal`, for example, the Stack behaves like a row.

In other words, a Stack could work exactly like a column, laying its children out vertically unless you explicitly make a child positioned, in which case it's removed from the layout flow and placed where you tell it to be.

To make a widget positioned, you wrap it in a Positioned widget. The Positioned widget has the properties top, left, right, bottom, width, and height. You don't have to set any of these properties, but you can set at most two horizontal properties (left, right, and width) and two vertical properties (top, bottom, and height). These properties tell Flutter where to paint the widget. The children are painted by the RenderStack algorithm:

- 1 It lays out all its nonpositioned children in the same way that a row or column would, which tells the stack its final size. If there are no nonpositioned children, the stack tries to be as big as possible.
- 2 It lays out all its positioned children relative to the Stack's render box, using its properties: top, left, and so on. The positioned properties tell Flutter where to place the Stack's children in relation to the Stack's parallel edge. top: 10.0, for example, insets the positioned widget 10.0 pixels from the top edge of the Stack's box.
- 3 When everything is laid out, Flutter paints the widgets in order, with the first child being on the bottom of the stack (figure 4.6).

■ positioned diagram

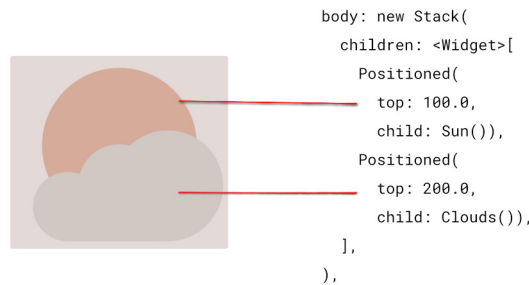


Figure 4.6 An example of using Positioned

In the weather app, you use a Stack in the ForecastPage. In the Scaffold.body property, which has three children, the children are the content of the ForecastPage, which looks like this:

Listing 4.11 Stack code in ForecastPage

```
Stack(
  children: <Widget>[
    SlideTransition(
      position:
      // ...
      child: Sun(...),
    ),
    SlideTransition(
      // ...
      child: Clouds(...),
    ),
  ],
),
```

This widget represents the sun (or moon) painting in the background.

A Stack takes children, such as a row or column.

The position property is similar to the Positioned.position property because it tells the widget *explicitly* where to be. The difference is that this property is animated.

The clouds are the second child in the stack, which overlaps the sun.

```

Column(
  children: <Widget>[
    forecastContent,
    mainContent,
    Flexible(child: timePickerRow),
  ],
),
],
),

```

← This last widget represents all the content that's on the topmost layer of the stack.

For the sake of example, the app could look like this if it weren't animated:

Listing 4.12 Example of nonanimated Stack code

```

Stack(
  children: <Widget>[
    Positioned(
      left: 100.0,
      top: 100.0,
      child: Sun(...),
    ),
    Positioned(
      left: 110.0,
      top: 110.0,
      child: Clouds(...),
    ),
    Column(
      children: <Widget>[
        forecastContent,
        mainContent,
        Flexible(child: timePickerRow),
      ],
    ),
  ],
),

```

Stack is your go-to widget if you want to place widgets either on top of each other, or in an explicit way in relationship to each other.

4.4.2 Table widget

The final static multichild widget that I want to show off is the Table, which uses a table-layout algorithm to make a table of widgets. Along with stacks, rows, and columns, tables are the building blocks of layout (ignoring *scrollable* widgets for now).

In the weather app, you'll use a Table to lay out the weekly weather data on the bottom half of the screen (figure 4.7).

Tables are more strict than other layout widgets you've seen because they have one purpose: to display data in a readable manner. Tables line up widgets in columns and rows, and each cell in the table has the same height as every other cell in its row and the same width as every widget in its column. Flutter tables require explicit column widths in

Weather app table widget

| | | |
|-----------|---|-----|
| Saturday | ☾ | 8 1 |
| Sunday | ☾ | 9 1 |
| Monday | ☾ | 7 4 |
| Tuesday | ☾ | 9 2 |
| Wednesday | ☾ | 8 2 |
| Thursday | ☾ | 7 1 |
| Friday | ☾ | 9 3 |

Figure 4.7 Screenshot of the `Table` widget in the context of the weather app

advance, and no table cell can be empty. Given these rules, you implement a table in code similar to that of other multichild widgets. The simple version API looks like this:

Listing 4.13 The API for the `Table` widget

```
Table(
  columnWidths: Map<int, TableColumnWidth>{},
  border: Border(), (2)
  defaultColumnWidth: TableColumnWidth(),
  defaultVerticalAlignment: TableCellVerticalAlignment,
  children: List<TableRow>[]
);
```

A map of the widths for each column, starting with the 0th row.

A default column width for column widths you don't want to set explicitly.

The border for the entire table.

A list of the column rows. The table works by establishing rows, each of which has multiple children that represent the cells in the rows.

This optional argument tells Flutter where to align the content of the cells within the cell itself.

4.4.3 Working with tables

These are a few things worth mentioning when working with tables:

- You don't have to pass in `columnWidths`, but `defaultColumnWidth` can't be null.
- `defaultColumnWidth` has a default argument, `FlexColumnWidth(1.0)`, so you don't have to pass in anything, but it can't be null. You can't pass in null explicitly: `defaultColumnWidth: null` would throw an error. Because it has a default argument, however, you can ignore it if you want each column to be the same size and you want the table to be as wide as possible.
- You define column widths by passing a map to the `columnWidths`. The map takes the index of the column (starting at 0) as the key and how much space you want it to take up as the value. I discuss `TableColumnWidth` later in this chapter.
- The `children` argument expects `List<TableRow>`, so you can't pass it any widget. It's a rare occasion in Flutter when you can't pass in any widget willy-nilly.
- `Border` is optional.
- `TableCellVerticalAlignment` works only if your row's children are `TableCells` (another widget that you see shortly).

With all that in mind, if you pass in some children, all the columns have the same width because they're all *flexed*, sizing themselves in relationship to one other. The elements in a row work together to take up the full width. I've configured the `Table` widget that displays in the `ForecastPage` to be spaced as shown in figure 4.8. (The dotted lines are added for the example and not in the code.)

Table cells

| | | | |
|-----------|---|---|---|
| Tuesday | ☾ | 9 | 1 |
| Wednesday | ☾ | 5 | 1 |
| Thursday | ☾ | 8 | 0 |
| Friday | ☁ | 9 | 0 |
| Saturday | ☾ | 7 | 1 |
| Sunday | ☾ | 6 | 1 |
| Monday | ☁ | 8 | 0 |

Figure 4.8 Table diagram with borders to show rows and columns

Following is the code snippet that defines the sizes of some rows. This code is important! Notice that there's no definition for the width of column one.

Listing 4.14 Using `FixedColumnWidth` on rows 0,2,3

The look I wanted required column 1 (in a 0-based column count, so it's the second column visually) to take up as much space as possible while the rest are fixed. Because the `defaultColumnWidth` defaults to being flexed, you don't need to give it a width.

```
// weather_app/lib/widget/forecast_table.dart -- line ~39
Table(
  columnWidths: {
    0: FixedColumnWidth(100.0),
    2: FixedColumnWidth(20.0),
    3: FixedColumnWidth(20.0),
  },
  defaultVerticalAlignment: TableCellVerticalAlignment.middle,
  children: <TableRow>[...],
);
```

To reiterate, I skipped 1 in the map, forcing the table to be as big as possible and take up the space that's left over after distributing the fixed widths to their columns.

This constant value of `TableCellVerticalAlignment` tells Flutter to lay out the content of the cells halfway between their tops and bottoms.

The remaining piece of the puzzle is a `TableRow`, which is a bit simpler than a normal row. Keep in mind two important configurations:

- Every row in a table must have an equal number of children.
- You can, but don't have to, use `TableCell` in the children's subwidget trees. The `TableCell` doesn't have to be a direct child of the `TableRow`, either, as long as somewhere above it in the widget tree, it has a `TableRow` as an ancestor.

In this app, you're going to use `TableCell` because it makes alignment super-easy. This widget knows how to control the children's alignment in the context of the table.

To complete this example, look at the code for the cells themselves. This table has four columns and seven rows. It would be cumbersome to write 28 widgets, so you're going to generate each row. Later in this chapter, you explore what Flutter calls the builder pattern, which is important and used commonly in Flutter apps.

4.4.4 Generating widgets from Dart's `List.generate()` constructor

Everything in Flutter is Dart code. What's more, Dart has features that make it useful specifically as a language for creating UI. Here, I want to show you a nifty example of how helpful it is that everything is Dart code. Rather than pass a list to the `children` property of the table, you can give it a function that returns a list of widgets.

Listing 4.15 Table code from weather app

```
Table(
  columnWidths: {
    0: FixedColumnWidth(100.0),
    2: FixedColumnWidth(20.0),
    3: FixedColumnWidth(20.0),
  },
  defaultVerticalAlignment: TableCellVerticalAlignment.middle,
  children: List.generate(7, (int index) {
    ForecastDay day = forecast.days[index];
    Weather dailyWeather = forecast.days[index].hourlyWeather[0];
    var weatherIcon = _getWeatherIcon(dailyWeather);
    return TableRow(
      children: [
        // ....
      ],
    ); // TableRow
  });
); // Table
```

This is more data you need. It's not pertinent to Flutter but it provides *hourly* weather, which is used to display the current temperature.

The callback returns whatever should be inserted into the generated list at the current index.

`List.generate` is a constructor for the Dart `List` class. It takes `int` as the first argument, which is the number of items the list will hold, and it takes a callback to generate that many items in the list. The callbacks receive the current index as an argument and are called exactly as many times as the `int` that they're passed. In this case, it's called seven times.

The data that the table cells need to display. The interesting thing is that you're using the index to get different data for each iteration of the table row. `forecast.days` is a variable that represents a list of daily weather descriptions.

This is the same idea. `_getWeatherIcon` is a method that returns the correct icon to represent the current weather.

This `List.generate` constructor function is going to execute at build time. If the concept seems confusing, it's fine to think of the `List.generate` function as a loop. It's going to run seven times. (The index at each loop iteration is actually be 0-6, though.) At each iteration, you have an opportunity to do some logic. You know that in each iteration, you have access to an index, which is different in each iteration, so you can fetch the data for this widget without knowing what that data is.

`List.generate` is a Dart feature that's not specific to Flutter, but it's a great tool to use when you need to build several widgets for a row, column, table, or list, each with different data. Without using `List.generate`, you'd have to write more verbose code, which would look something like this:

Listing 4.16 Verbose code without a function

```
Table (
  children: [
    TableRow(
      children: [
        TableCell(),
        TableCell(),
        TableCell(),
        TableCell(),
      ]
    ),
    TableRow(
      children: [
        TableCell(),
        TableCell(),
        TableCell(),
        TableCell(),
      ]
    ),
    //... etc, 5 more times,
  ]
)
```

Ack! Even with all the content of each `TableCell` stripped out, you can see how cumbersome this code is, especially because each group of rows and cells is the same as every other one. Using a function to build the rows programmatically is nice and common in Flutter.

WARNING The caveat is that this example works only because the array of the data is specifically ordered. If you can't guarantee the order of your list, and if order matters, this solution may not be the best one.

The important point is that all this code is doing is creating a list of widgets. It's not the most profound discovery, but it's an example of the advantage of writing purely Dart code without a markup language. Flutter takes advantage of this feature quite a bit.

The remaining code to implement creates the table rows themselves, which display basic widgets by using `TableCell`, `Text`, `Icon`, and `Padding`. For the sake of familiarizing yourself with Flutter code, here's a snippet of the rows:

Listing 4.17 Table cell examples from the weather app

```
// weather_app/lib/widget/forecast_table.dart -- line ~52
children: List.generate(7, (int index) {
  ForecastDay day = forecast.days[index];
  Weather dailyWeather = forecast.days[index].hourlyWeather[0];
  var weatherIcon = _getWeatherIcon(dailyWeather);
  return TableRow(
    children: [
      TableCell(
        child: Padding(
          padding: const EdgeInsets.all(4.0),
          child: ColorTransitionText(
            text: DateUtils.weekdays[dailyWeather.dateTime.weekday],
            color: weatherIcon,
          ),
        ),
      ),
    ],
  );
});
```

This widget is returned once for each iteration of `List.generate`.

This `TableCell` displays the day of the week.

```

        style: textStyle,
        animation: textColorTween.animate(controller),
      ),
    ),
    TableCell(
      child: ColorTransitionIcon(
        icon: weatherIcon,
        animation: textColorTween.animate(controller),
        size: 16.0,
      ),
    ),
    TableCell(
      child: ColorTransitionText(
        text: _temperature(day.max).toString(),
        style: textStyle,
        animation: textColorTween.animate(controller),
      ),
    ),
    TableCell(
      child: ColorTransitionText(
        text: _temperature(day.min).toString(),
        style: textStyle,
        animation: textColorTween.animate(controller),
      ),
    ),
  ],
);
}},
// ...

```

This TableCell displays the icon that corresponds to current weather conditions.

This TableCell displays the daily high temperature.

This TableCell displays the daily low temperature.

This code is standard Flutter UI code. It's adding four table cells to each row with standard table cells and other widgets. Outside the `List.generate` portion, there are no special tricks here.

Finally, look at the code that adds this Table widget to the tree. It's located in the `ForecastPageState.build` method.

Listing 4.18 A portion of the `ForecastPageState.build` method

```

// weather_app/lib/page/forecast_page.dart
return Scaffold(
  appBar: // ...
  body: new Stack(
    children: <Widget>[
      // ... sun and clouds positioned widgets
      Column( 1)
        verticalDirection: VerticalDirection.up,
        children: <Widget>[
          forecastContent,
          mainContent,
          // Flexible(child: timePickerRow),
        ],
      ],
  ),
);

```

This column houses all the content of the ForecastPage.

This variable represents the Table widget.

More widgets in the weather app.

This reverses the direction of the column. The first child widget is at the bottom of the Column render box. In this case, you want the content of the column to be aligned at the bottom of the screen and laid out accordingly. This nice little feature of Flutter's Column widget makes it much easier to achieve this alignment than by writing your own code.

Table generally isn't much different from any other widgets, but the lessons in that section are valuable. Soon, you'll learn about the builder pattern, which is similar to the `List.generate` method.

4.4.5 **TabBar widget**

Tabs are common UI element in mobile apps. The Flutter material library provides built-in tab widgets, which make working with tabs relatively easy.

The built-in `TabBar` widget displays its children in a scrollable horizontal view, and makes them tappable. The widgets in the `TabBar`, when tapped, executes a callback to which you can pass the `TabBar`. Tabs are most commonly used to switch between pages or UI components without navigating, so, the callback passed to the `TabBar`'s children widgets are most commonly used to swap out some widgets on the page (figure 4.9).

■ TabBar widget

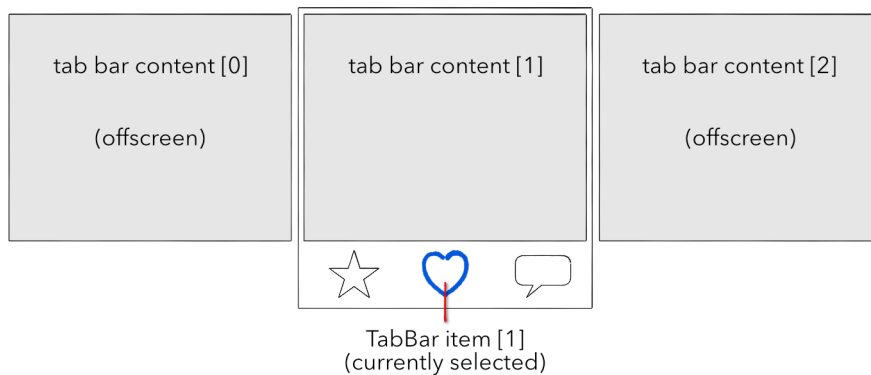
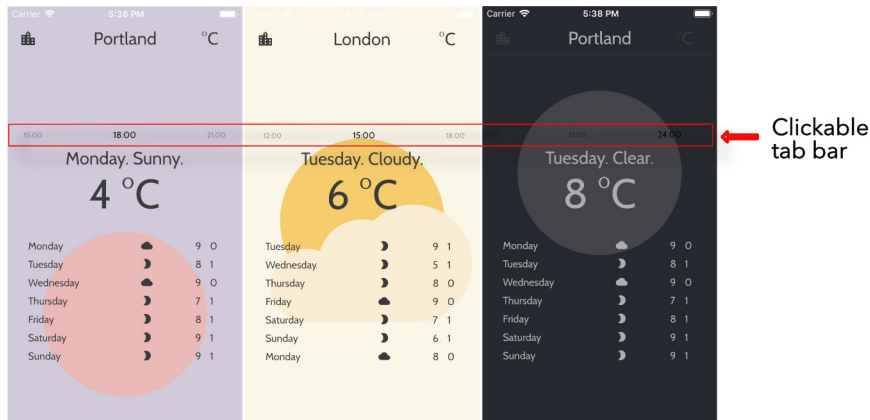


Figure 4.9 Diagram of tab-related widgets in Flutter

This diagram represents the basic idea of tabs. When you click an element on the tab bar, the corresponding tab content changes. In the Flutter app, I use a tab bar to build the row of times that can be clicked to update the temperature for that time of day.

The `TabBar` widget has two important pieces, one of which is the children—the widgets that display the time of day that the user wants to select. The other important part is `TabController`, which handles the functionality.



4.4.6 TabController

In Flutter, many widgets that involve interaction have corresponding controllers to manage events. `TextEditingController` is used with widgets that allow users to type input, for example. In this case, you're using `TabController`. The controller is responsible for notifying Flutter app when a new tab is selected so that your app can update the tab to display the desired content. The controller is created higher in the tree than the `TabBar` and passed into the `TabBar` widget. This architecture is required because the parent of the `TabBar` is also the parent of the tab widgets. For a concrete example, the `TabBar` code in the weather app is in the `weather_app/lib/widget/time_picker_row.dart` file.

In that file, you'll find the custom widget called `TimePickerRow`, a stateful widget that displays the tabs and tells its parent when a tab-change event happens, by using `TabController`.

Listing 4.19 TabController and TabBar widget setup

This `StatefulWidget` establishes some properties that are passed into it. In this case, the widget expects a list of `Strings`, which are displayed as the times of day ("12:00", "3:00", and so on).

```
// weather_app/lib/widget/time_picker_row.dart
class TimePickerRow extends StatefulWidget {
  final List<String> tabItems;
  final ForecastController forecastController;
  final Function onTabChange;
  final int startIndex;
  // ...
}
```

The `TabBar` needs to know which tab is selected by—in this case, the widget that represents the current time of day.

This controller is a class that I made to make it easier to fetch the forecast data. It's not important to Flutter.

This is the callback that the parent passes in, which in this case should be used to notify the parent when a new tab is selected.

Those are the important properties passed into the widget itself, but all the functionality lives in the State object.

Listing 4.20 Flutter tabs implementation in weather app

```
// weather_app/lib/widget/time_picker_row.dart -- line ~30
class _TimePickerRowState extends State<TimePickerRow> with
    SingleTickerProviderStateMixin {
    TabController _tabController;
    int activeTabIndex;

    @override
    void initState() {
        _tabController = TabController(
            length: utils.hours.length,
            vsync: this,
            initialIndex: widget.startIndex,
        );
        _tabController.addListener(handleTabChange);
        super.initState();
    }

    void handleTabChange() {
        if (_tabController.indexIsChanging) return;
        widget.onTabChange(_tabController.index);
        setState(() {
            activeTabIndex = _tabController.index;
        });
    }

    // ...
}
```

Declare a TabController to handle tab functionality. It's created in the constructor.

vsync has to do with animations.

TickerProviderStateMixin is a mix-in that tells Flutter this widget has some properties that will animate. TabBars have built-in animations, so it's needed.

Here, the controller is created. TabController must know how many tabs will exist.

You can add a listener to your controller that will execute the callback whenever the tab changes.

This check prevents a new event from starting in the middle of an animation.

JUST IN TIME: LISTENERS

Listeners aren't specific objects or type of object, but naming conventions used for different asynchronous functionality.

At many places in the Flutter library, you see the words *listener*, *change notifier*, and *stream*. These words are different flavors and pieces of the same kind of programming concept: observables. Observables are known as Streams in Dart.

A listener is an aptly named piece of the observable ecosystem. It generally refers to a function that's called in response to some event that will happen at an unknown time. The function is sitting around listening for someone to say "Okay, now's your time to execute."

The TabController's `addListener` function is called whenever a user changes the tabs, giving you a chance to update some values or state whenever a user changes tabs. In this example, the listener knows to execute the callback provided to it whenever a tab in the TabBar is tapped.

Along with listeners, the TabController has getters that help you manage your tabs and corresponding content. Inside the `_handleTabChange` method, you could do something like this to make sure that your app knows which tab is active (currently displayed onscreen):

Listing 4.21 TabController index getter

```
int activeTab;
void _handleTabChange() {
  setState(() => this.activeTab = _tabController.index);
}
```

TabController.index returns the selected tab index. This example assumes that some content relies on the **activeTab** piece of State.

`setState` is also important here. In the weather app, whenever you tap a different time of day in the tab bar, the UI rerenders with the weather conditions for that time of day. This action is possible because `setState` tells Flutter to rerender and to display the newly selected tab when it does. The `TabController.index` getter refers the currently active tab.

The last note I'd like to make about the `TabBar` controller is that you don't have to change it directly. This object is used to get information about the tabs and to update which tabs are active. You need only to interact with it, not extend it into a custom class.

4.4.7 TabBar widget in practice

Now that you've been exposed to the functionality of tabs and using the tab bar in Flutter, consider an example from the weather app. Although most of the `TabBar` functionality lives in the controller, developers care about the widget itself and passing it the arguments you pass to it. This is how the `TabBar` is used in the weather app.

Listing 4.22 TabBar widget in build method

```
// weather_app/lib/widget/time_picker_row.dart
@override
Widget build(BuildContext context) {
  return TabBar(
    labelColor: Colors.black,
    unselectedLabelColor: Colors.black38,
    unselectedLabelStyle:
      Theme.of(context).textTheme.caption.copyWith(fontSize: 10.0),
    labelStyle: Theme.of(context).textTheme.caption.copyWith(fontSize: 12.0),
    indicatorColor: Colors.transparent,
    labelPadding: EdgeInsets.symmetric(horizontal: 48.0, vertical: 8.0),
    controller: _tabController,
    tabs: widget.tabItems.map((t) => Text(t)).toList(),
    isScrollable: true,
  );
}
```

The **TabController**, which was created in the parent widget's constructor, is passed into the widget from the parent.

By default, tabs aren't scrollable. This argument allows them to be scrollable.

The **TabBar** widget comes with many configuration options, which are aptly named. Here, **labelColor**, **unselectedLabelColor**, and every property down to **labelPadding** are configuration options that define styles.

TabItems are passed in from the **ForecastPage** and happen to be displaying **Text**. It could be any widget, though. Icons are common. This is another instance of using Dart code to programmatically create the widgets. It iterates through every **String** from **tabItems** and returns a **Text** widget for each one.

At this point, you've seen all the moving parts of the `TabBar`. It's a lot, but the important takeaways are this:

- Using tabs requires a `TabController` and children widgets. The children are the widgets that will be displayed and are tappable.

- The functionality required to switch tabs when a widget in the `TabBar` is tapped is done via a callback. This callback should use the properties exposed by the `TabController` to tell Flutter when to render a new tab.

This mental paradigm is common in Flutter.

4.5 Working with *ListView* and *builder*

`ListView` is arguably the most important widget in the book, which is apparent by the sheer length of its documentation page on the Flutter website.⁴ It's not only used frequently, but also introduces some patterns and ideas that are crucial in writing an effective Flutter app.

The `ListView` widget is like a column or row in that it displays its children widgets in a line. It's important that it's scrollable. This widget is commonly used in the case of an unknown number of children. You could use `ListView` in a to-do app to display all your to-dos. You could have 0 or many to-dos. `ListView` provides a way to say “Hey, for each of these pieces of information, create a widget and add it to this list.”

In the weather app (figure 4.10), a `ListView` widget is used in the `SettingsPage` widget in `lib/page/settings_page.dart`. It uses (fake, generated) data to build a scrollable list that lets you select which cities the user of the weather app cares about.

Weather App settings page

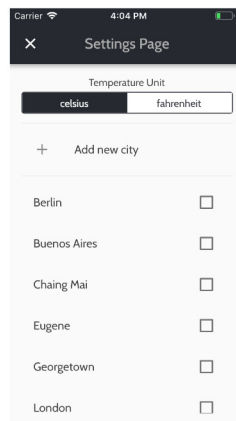


Figure 4.10 Weather app settings page

According to the docs, a `ListView` is a “scrollable list of widgets arranged linearly.” In human English, it’s a scrollable row or column, depending on what axis you tell it to lay out on. The power of `ListView` is how flexible it is. It has a couple of constructors that let you make choices based on the content of the list. If you have a static, small number of items to display, you can create a `ListView` with the default constructor,

⁴ The documentation for `ListView` is at <https://api.flutter.dev/flutter/widgets/ListView-class.html>.

and it's created with code similar to a row or column. This option is the most performant one, but it may not be ideal if you have tens or hundreds of items to put in the list or an unknown number of items.

What I want to focus on here, though, is the builder pattern in Flutter. The builder pattern is used all over Flutter; it essentially tells Flutter to create widgets *as needed*. In the default `ListView` constructor, Flutter builds all the children at the same time and then renders. The `ListView.builder` constructor takes a callback in the `itemBuilder` property, and that callback returns a widget. In this case, the callback returns a `ListTile` widget. This builder makes Flutter smarter about rendering items if you have a huge (or infinite) number of list items to display in your list. Flutter renders only the items that are visible onscreen.

Imagine a social media app like Twitter, which is an infinite list of tweets. It wouldn't be possible to render all the tweets in that list every time some state changes because the number is infinite. Instead, the app renders as needed as the user scrolls the tweets into view. This practice is common in UI, and Flutter provides `ListView` as a built-in solution to this problem.

Here's an example in the weather app. `ListView` is used on the `SettingsPage`.

Listing 4.23 `ListView` builder code in `SettingsPage`

`ListView` expands on its main axis to be as big as possible. Because it's the child of a column in this app, it would expand infinitely if `Expanded` didn't constrain it.

`shrinkWrap` is another way to protect against infinite size. It tells `ListView` to try to be the size of its children.

This `itemBuilder` property takes a callback that will be passed a build context and the index of the item in the list. This function is the builder function, and functions like it are used frequently in Flutter.

A builder must know how many total items it will create.

The `CheckboxListTile` is a convenience widget built into Flutter that displays a check box as children in `ListView` widgets.

```
// weather_app/lib/page/settings_page.dart -- line ~58
Expanded( 1)
  child: ListView.builder(
    shrinkWrap: true,
    itemCount: allCities.length,
    itemBuilder: (BuildContext context, int index) {
      var city = allCities[index];
      return CheckboxListTile(
        value: widget.settings.selectedCities[city],
        title: Text(city),
        onChanged: (bool b) => _handleCityActiveChange(b, city),
      );
    },
  );
```

The `onChanged` argument is used to control the check box. The function you pass it is called whenever the item is checked. The `_handleCityActiveChange` method is a method I wrote to make sure that the app knows which cities are active.

`ListView` probably seems to be more complicated than many of the other widgets discussed in this chapter. The important piece of this example is builder. The `ListView.builder` is a simple way to create a scrolling list with potentially infinite items. That's often what the builder pattern is used for in Flutter: to create widgets that display unknown data.

It's worth noting that `ListView` has a couple of other constructors:

- `ListView.separated` is similar to `ListView.builder` but takes two builder methods: one to create the list items and a second to create a separator that will be placed between list items.
- `ListView.custom` is the last constructor. As the name suggests, it allows you to create a `ListView` with custom children. This isn't quite as simple as updating the builder. Suppose that you have a `ListView` in which some list items should be a certain widget, and other list items are an entirely different widget. This situation is where the custom `ListView` comes into play, because it gives you fine-grained control of all aspects of how the `ListView` renders its children.

Sincerely, `ListView` is one of the widgets that beautifully represents Flutter as a whole. It's clean, functional, but highly useful. The API is simple enough but doesn't pigeon-hole you into a certain paradigm.

Summary

- Flutter includes a ton of convenience structural widgets, such as `MaterialApp`, `Scaffold`, and `AppBar`. These widgets give you an incredible amount for free: navigation, menu drawers, theming, and more.
- Use the `SystemChrome` class to manipulate features of the device itself, such as forcing the app to be in landscape or portrait mode.
- Use `MediaQuery` to get information about the screen size. This widget is useful if you want to size widgets in a way that ensures that they scale by screen size.
- Use `Theme` to set style properties that affect nearly every widget in your app.
- Use the `Stack` widget to overlap widgets anywhere on the screen.
- Use the `Table` widget to lay out widgets in a table.
- `ListView` and its builder constructor give you a fast, performant way to create lists with infinite items.

A

- Android operating system
 - building apps on 50–52
 - running apps on 50–52
- app layer 36
- app structure, Scaffold widget and 63
- AppBar widget 86
 - properties of 65
- AppBar.automaticallyImplyLeading property 66
- AppBar.leading property 66
- appKey argument 4
- AppRegistry method 4
- apps
 - building and running 49–52
 - on Android 50–52
 - on iOS 52
- asynchronous programming 61

B

- backgroundColor property 16
- binding layer 34, 36
- BoxConstraint widget 58
- build method, MaterialApp and 62–63
- BuildContext, adding Themes in Flutter apps 69
- builder constructor 86
- builder pattern 58, 85
- business logic layer 33, 37
- Button component 17

C

- children argument, working with tables and 75
- class libraries, .NET Standard 38–40

- code

- cross-platform 37–40
 - .NET Standard class libraries 38–40
- column widths, defining, working with tables and 75
- components
 - designing apps using 3
- console.log() function 8
- cross-platform apps
 - creating using MVVM 33–37, 53–54
 - creating solutions 40–52
 - cross-platform code 37–40
 - UI design patterns 32–33

D

- Dart
 - applications and pubspec.yaml file 59
 - Future class 61
 - List.generate() constructor and generating widgets 77–80
 - main.dart as an entry point 60
 - named imports in 67
- Debug JS Remotely option 10
- defaultColumnWidth, working with tables and 75
- deleteTodo function 20
- design patterns, UI 32–33
- designing apps 3
- Dev Settings option 11
- developer menu
 - opening 9–10
 - overview of 10

E

emulators 10
 Enable Hot Reloading option 11
 Enable Live Reload option 11
 entry point 60

F

FirstViewModel class 53
 flexed columns 76
 Flutter
 and constraints of widgets' sizes 66
 and diagram of tab-related widgets in 80
 as a complete SDK 57
 built-in widgets 57
 Dart code 77
 importing iOS icons 60
 logical pixel 69
 main.dart 60
 of method 70
 specific configuration items 59
 styling in 67
 SystemChrome and 60–61
 TextEditingController 81
 weather app example 57–86
 background of 72
 file structure 59
 main function 60
 table code from 77
 flutter_in_action repository 59
 ForecastPage widget, weather app example 65
 Future, and asynchronous programming in
 Dart 61

G

getVisibleTodos function 27

H

hot reloading 11

I

Input component 12
 inputChange function 8, 13
 inputValue property 5, 13
 iOS operating system, building and running apps
 on 52

K

keyboardShouldPersistTaps prop 5

L

laying out apps 3
 leading action, AppBar widget property 66
 List.generate() constructor 77–80
 listeners 82
 ListView widget 58, 84–86
 builder pattern 85
 described 84
 flexibility 84
 frequent use of 84
 Twitter 85

M

main function 60
 MaterialApp widget 86
 helpful functionality and 62
 Material widget library 62
 WidgetsApp and 62
 MediaQuery class 58
 MediaQuery widget 86
 of method 69
 MediaQuery.of(context).size, static method 69
 mobile platforms 41–42
 MVVM (model-view-view model) design pattern
 creating cross-platform apps 53–54
 creating solutions 40–52
 cross-platform code 37–40
 UI design patterns 32–33
 overview of 33–37

N

Name property 35
 .NET Standard specification
 class libraries 38–40
 New Solution dialog box 44
 Number property 36

O

onPress method 24

P

PCLs (portable class libraries) 38
 placeholderTextColor 8

Positioned widget 73
 PreferredSize widget, and providing accurate
 sizing for 71
 profiles 38
 pubspec.yaml file, Dart applications and 59

R

React Native apps
 creating 2–3, 9
 designing 3
 react-native run-android 7
 react-native run-ios 7
 Reload option 10

S

Scaffold widget 86
 and app structure 63
 and providing functionality 63
 bottom sheet 63
 constructor method and configuring optional
 features 64
 drawer 63
 optional features 64
 properties of 64
 Scaffold.appBar property 66
 screen-aware sizing methods 70–71
 ScrollView component 5
 selectionColor 8
 setType function 25, 27
 Show Perf Monitor option 11
 simulators 9
 Stack widget 58, 72–74, 86
 default behavior 72
 nonpositioned children 72
 Start Systrace option 11
 State object 81
 styles, color-related, controlled by Theme widget 67
 styles.border 26
 styles.selected 26
 submitTodo function 15, 17, 20
 SystemChrome 60–61
 SystemChrome class 86
 SystemChrome.setPreferredOrientations 60
 Systrace 11

T

TabBar component 25
 TabBar widget 58
 in practice 83–84

 scrollable horizontal view of its children 80
 TabController 81–83
 TabBarItem component 25
 TabController 81–83
 getters 82
 listeners 82
 setState 83
 table row, equal number of children and 76
 Table widget 58, 79, 86
 displaying data in a readable manner 74
 table-layout algorithm 74
 TableCell
 in children's subwidget trees 76
 working with tables and 77
 tables, working with 75–77
 Text property 34
 TextInput component 5, 8, 13
 Theme widget 86
 accessibility 67
 and applying styles automatically 67
 arguments and 68
 BuildContext and 69
 implementing in Flutter apps 68
 multiple Themes 68
 Theme.primaryColor, and changing the color of
 all widgets 68
 ThemeData class 68
 then function 61
 this.submitTodo 17
 TimePickerRow widget 81
 todo app, creating 2
 design 3
 developer menu, opening 9
 Todo component 18, 20, 23
 TodoButton component 23
 todoIndex function 15
 TodoList component 18, 20, 27
 TodoMVC site 3
 todos variable 28
 Toggle Inspector option 11
 toggleComplete function 20, 25
 TouchableHighlight component 16, 26
 Twitter, as infinite list of tweets 85

U

UI (user interface)
 designing, design patterns 32–33
 UI layer 33, 36
 UI logic layer 37
 underlayColor property 16

V

versioned packages 48

Visual Studio

for Mac, creating solutions with 42–45

for Windows, creating solutions with 45–47

W

widgets

and of method 69

AppBar 65

building-block widgets 58

built-in, Flutter and 57

configuring structural 61–67

constraints and final size 66

convenience widgets 58

iOS-style 60

layout widgets 58

ListView 84–86

making a widget positioned 73

MaterialApp 62–63

MediaQuery 69–70

PageContainer 63

PreferredSize 66–67

Scaffold 63–65

Stack widget 72–74

structural 58

TabBar 80–84

Table widget 74

Theme 67–69

WidgetsApp 62

