REAL-WORLD XAMARIN.FORMS

SUCCINCTLY

BY ALESSANDRO DEL SOLE

SUCCINCTLY EBOOK SERIES

Syncfusion

Real-World Xamarin.Forms Succinctly

By
Alessandro Del Sole

Foreword by Daniel Jebaraj

2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA

ISBN: 978-1-64200-217-1

Technical Reviewer: James McCaffrey
Copy Editor: Courtney Wright
Acquisitions Coordinator: Tres Watkins, VP of content, Syncfusion, Inc.
Proofreader: Graham High, senior content producer, Syncfusion, Inc.

Table of Contents

The Story Behind the *Succinctly* Series of Books

Daniel Jebaraj, CEO
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Alessandro Del Sole is a Xamarin Certified Mobile Developer and has been a Microsoft MVP since 2008. Awarded MVP of the Year in 2009, 2010, 2011, 2012, and 2014, he is internationally considered a Visual Studio expert and a .NET authority. Alessandro has authored many printed books and ebooks on programming with Visual Studio, including *Visual Studio 2017 Succinctly*, *Visual Basic 2015 Unleashed*, and *Visual Studio Code Succinctly*. He has written tons of technical articles about .NET, Visual Studio, and other Microsoft technologies in Italian and English for many developer portals, including MSDN Magazine and the Visual Basic Developer Center from Microsoft. He is a frequent speaker at Italian conferences, and he has released a number of Windows Store apps. He has also produced a number of instructional videos in both English and Italian. Alessandro works as a senior software engineer for Fresenius Medical Care, focusing on building mobile apps with Xamarin in the healthcare market. You can follow him on Twitter at @progalex.

Introduction

With the book *Xamarin.Forms Succinctly*, I walked through the development platform with an instructional approach. I discussed common views, types, libraries, and patterns with simple examples that explained how to put together the different blocks of the technology from the point of view of the beginner, no matter how experienced in C# programming the reader was.

When it comes to building real applications, developers working with Xamarin.Forms who are not very experienced with this technology will face a number of problems to solve, especially when the app has a more complex design, and especially if they used to work with other platforms, such as desktop or web frameworks. This is natural and happens because developing for mobile devices implies taking care about things you do not need to care about on the desktop or for a website. Checking the battery status to ensure the user does not lose data, ensuring the user knows the app is working with a data plan, accessing files on the device, and notifying the user of a problem with a nice UI are all examples of this different way of thinking.

Developers sometimes expect that everything really works cross-platform at no cost, but unfortunately that's not true. With Xamarin.Forms, this happens in 90 percent of cases, but there is a good 10 percent of situations where you need to write custom renderers or use third-party libraries to make a view have the exact same behavior on the iOS and Android platforms, and that is totally normal because they are two different operating systems, with two different APIs and implementations.

This requires effort. I have been working with Xamarin.Forms on real-world mobile apps for five years, and I have been lucky enough to do this within a quite large agile team with professional designers, lawyers (yes, an app must comply with regulations), businesspeople, and other cool developers. During these years, I have gained a lot of experience in implementing solutions to solve both common problems to any app and custom scenarios, and I realized that sometimes the effort needed to obtain simple things is higher than expected.

With this in mind, I decided to write this new book and collect some of my experience to provide developers with insights and points of view on topics they will need to face sooner or later when working with Xamarin.Forms. This book contains only a part of that experience, and this is due to the nature of books from the *Succinctly* series. However, the topics you will find in this book are of common interest, and the examples will help you be faster and more productive.

The book comes with a companion code repository that contains sample projects grouped by chapter. My suggestion is to read chapters while keeping the sample projects open in Visual Studio, and I will remind you in those chapters where this approach is necessary. Figures in the book are taken from Visual Studio 2019 on a Windows PC, but the source code will also work on macOS and Visual Studio 2019 for Mac with no issues (this is also a reason why the examples target iOS and Android only, and not Universal Windows Platform).

There is certainly much more in Xamarin.Forms to explore in real-world applications, and this book cannot cover every possible scenario, but I am confident that it will help you solve many common problems quickly.

Chapter 1 Cross-Platform Device Capabilities

Mobile apps in the real world need to make sure the user always has control over what's happening, especially with unexpected situations, and they need to respond accordingly. For example, if the network connection drops off, not only must the app intercept the loss of connection, but it also needs to communicate to the user that a problem happened and provide one or more actions. This should be done with a nice user interface, avoiding unnecessary modal dialogs that might be OK for desktop apps, but not on mobile apps where the user should never feel they are blocked.

This chapter will describe how to solve common problems that mobile apps might encounter, in a cross-platform approach. You will not find how to work with sensors, which are usually specific to certain application scenarios; you will find ways of handling scenarios that almost every app will need to face.

Chapter prerequisites

Except for taking pictures and recording videos, all the techniques and scenarios described in this chapter simply require the Xamarin.Essentials library, which Visual Studio automatically adds to a new solution at creation time. Xamarin.Essentials is an open-source library available as a NuGet package. It provides a cross-platform API to access features that are common to all the supported systems, to help you avoid the need to write native, platform-specific code.

In order to follow the examples provided in this chapter, create a new Xamarin.Forms solution by using the **Blank** template, as shown in Figure 1.

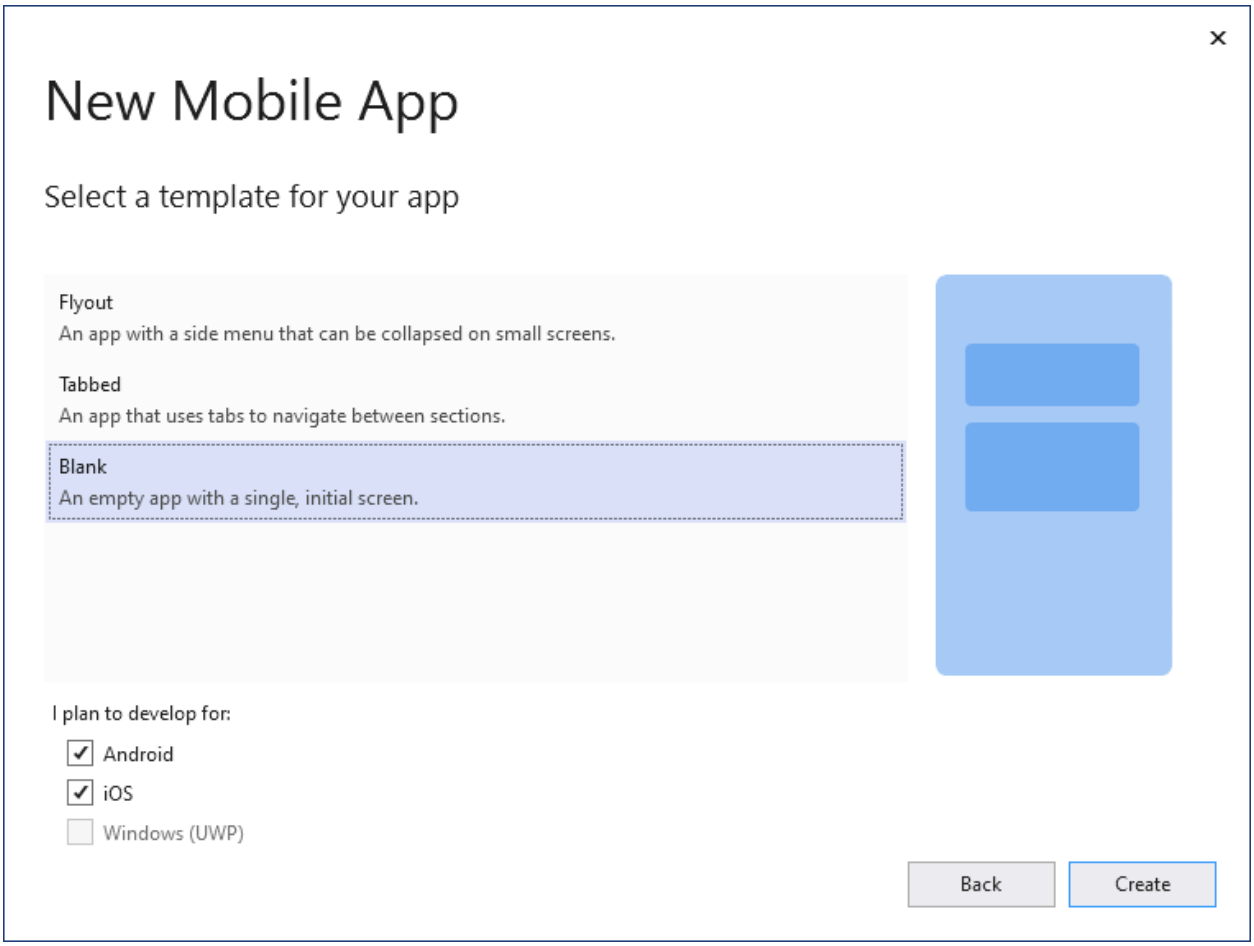

Figure 1: Creating a new Xamarin.Forms blank solution

When the solution is ready, make sure you update the Xamarin.Forms and Xamarin.Essentials packages to the latest version. To do so, right-click the solution name in Solution Explorer, select **Manage NuGet Packages for Solution**, and then open the **Updates** tab. Click the **Select all packages** tab (see Figure 2), and then the **Update** button. In Figure 2, you will see that an update is available only for Xamarin.Forms, which is fine.

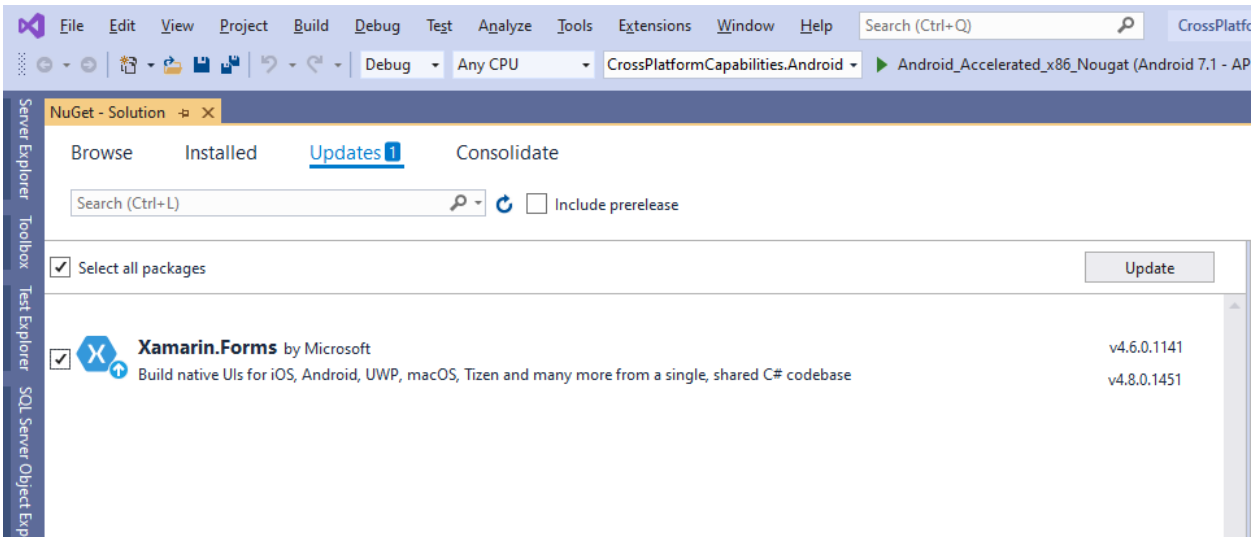

Figure 2: Updating libraries in the Xamarin.Forms solution

You are now ready to write some code (and you can always follow the example by opening the companion source code).

Tip: The steps I just described are the same ones to follow when you start a new chapter and you want to create a new, fresh solution.

Managing the network connections with Snackbar views

Most mobile apps need an internet connection to work properly and you, as the developer, are responsible for responding to changes of the connection status. For example, you need to inform users that the connection dropped and, depending on the scenario, you might also need to offer different actions. You probably already know that handling changes in the status of the network connection is accomplished via the **`Xamarin.Essentials.Connectivity`** class, but in this chapter I will show how to implement a Snackbar view to inform the user. This view does not exist in Xamarin.Forms, but it has become very popular in many applications because it does not block the user interaction while providing information. For a better understanding, consider Figure 3, where you see the final result of the work.

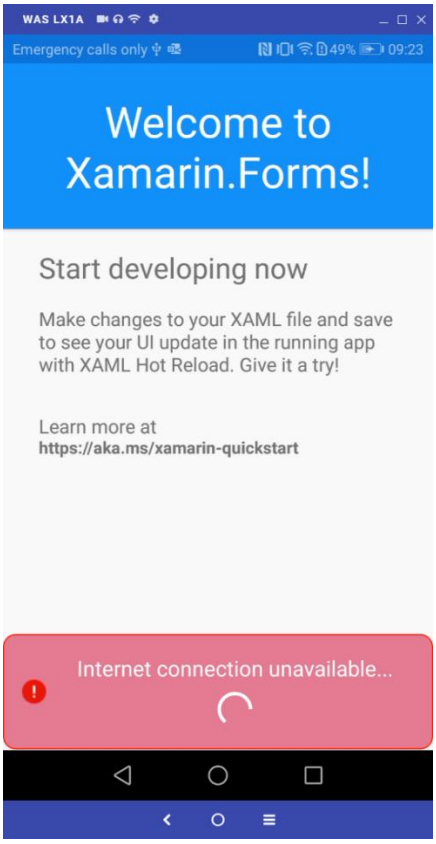

Figure 3: Displaying a Snackbar when the connection drops

As you can see, the user interface displays a nice view that informs the user that the connection dropped, while the other views have been programmatically disabled. Such a view will disappear once the connection is established again, and the other views will be re-enabled. Let's now go through each step required to build this component.

Building a Snackbar view

A Snackbar can basically be thought of as a box that appears as an overlay to provide information to the user. This can include errors, confirmations, warning messages, and so on. The benefit of using a Snackbar is that it does not block the user interaction with the app, and you have complete control of it. For instance, you could animate a Snackbar to make it appear, keep the information visible for a few seconds, and then make it disappear. A typical situation for this is when the user saves some changes and the app provides a confirmation. In the case of the lost network connection, it could be a good idea to keep the Snackbar visible until the connection returns. Obviously, depending on the design requirements, you will change the behavior accordingly.

The implementation of the Snackbar can be done in a separate **`ContentView`** object. Code Listing 1 shows the full code for the Snackbar you see in Figure 3, and the new control is called **`NoNetworkSnackBarView`**.

Code Listing 1

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentView xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="CrossPlatformCapabilities.NoNetworkSnackBarView">
  <ContentView.Content>
        <Grid>
            <Frame
        BackgroundColor="PaleVioletRed"
        BorderColor="Red"
        CornerRadius="10"
        VerticalOptions="End"
        HasShadow="False"
        Padding="16">

                <StackLayout
            Orientation="Horizontal"
            Spacing="16"
            Margin="0"
            Padding="0">
                    <Image
                Source="ExclamationMark.png"
                HorizontalOptions="Start"
                HeightRequest="20"
                WidthRequest="20"
                VerticalOptions="CenterAndExpand" />

                    <StackLayout
                Margin="40,0,0,0"
                VerticalOptions="CenterAndExpand"
                HorizontalOptions="StartAndExpand"
                Orientation="Vertical"
                Spacing="5">
                        <Label TextColor="White"
                               FontSize="Medium"
                               Text="Internet connection unavailable..."
                               HorizontalTextAlignment="Center"/>

                        <ActivityIndicator
                             IsRunning="True" Color="White"
                             VerticalOptions="Center"
                             HeightRequest="35"
                             WidthRequest="35"/>
                </StackLayout>
```

```
                </StackLayout>
            </Frame>
        </Grid>

  </ContentView.Content>
</ContentView>
```

Tip: You can even go a step further and build a completely reusable Snackbar by binding the `Label` properties, such as `Text` and `TextColor`, and the `Frame` color properties to objects in a view model. This is actually how I have my Snackbar views, which also need to be localized. For the sake of simplicity, and because we just need a specific message, in this chapter you see a Snackbar that is tailored for loss of connectivity.

The code is very simple, as it combines **`StackLayout`** panels to organize an icon, a text message, and a running **`ActivityIndicator`** inside a **`Frame`**. There is nothing complex in Code Listing 1—this implementation is for the UI only and does not perform any action, which means no C# code is required.

Handling the status of the network connection

The **`Connectivity`** class exposes an event called **`ConnectivityChanged`**, which is raised every time the status of the network connection changes. Because you might have multiple pages in your app that require a connection, the best place to handle the event is in the **`App`** class. First of all, add the following line right after the invocation to the **`InitializeComponent`** method, inside the constructor:

```
Connectivity.ConnectivityChanged += Connectivity_ConnectivityChanged;
```

The event handler might look like the following:

```
private void Connectivity_ConnectivityChanged(object sender,
             ConnectivityChangedEventArgs e)
{
    MessagingCenter.Send(this, "ConnectionEvent",
              e.NetworkAccess == NetworkAccess.Internet);
}
```

Because the event is handled in the **`App`** class, and this does not have direct access to the user interface of any page, the best way to inform all the subscribers that the network status changed is sending a broadcast message via the **`MessagingCenter`** class and its **`Send`** method. The message includes a **`bool`** parameter, which is **`true`** if an internet connection is available, otherwise **`false`**, whether the device is completely disconnected from any network or connected to a local network or to a network with limited access.

You can further manage the event using the values from the **NetworkAccess** enumeration if you need more granularity of control. There is actually an additional thing to do: the network check will not happen immediately at startup, so you need to handle the **OnStart** method as follows:

```
protected override void OnStart()
{
    MessagingCenter.Send(this, "ConnectionEvent",
             Connectivity.NetworkAccess == NetworkAccess.Internet);
}
```

The same check is done at startup, just once, and the same message is sent to the subscribers. The next step is notifying the app's user interface of the network status change and showing the Snackbar.

Notifying the user interface and showing the Snackbar

All the pages that will want to display the Snackbar when the connection drops off will need to subscribe the **ConnectionEvent** message via the **MessagingCenter.Subscribe** method. In the current example there is only one page, so Code Listing 2 shows the code for the MainPage.xaml.cs file.

Code Listing 2

```
public partial class MainPage : ContentPage
{
    public MainPage()
    {
        InitializeComponent();

        MessagingCenter.Subscribe<App, bool>(this,
            "ConnectionEvent",
            ManageConnection);
    }

    private void ManageConnection(App arg1, bool arg2)
    {
        LayoutRoot.IsEnabled = arg2;
        ConnectionStatusBar.IsVisible = !arg2;
    }
}
```

This code is also simple. The page subscribes for the message sent by the **App** class and expects a **bool** value, which in this case is **true** if the connection is available and of type **Internet**, and otherwise **false**. When the message is raised, the **ManageConnection** method is invoked. **LayoutRoot** is a **Grid** that I added to the layout that Visual Studio autogenerated when creating the solution, and **ConnectionStatusBar** is the instance name for the **NoNetworkSnackBarView** control. Code Listing 3 shows the full XAML markup for the MainPage.xaml file.

Code Listing 3

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns:local="clr-namespace:CrossPlatformCapabilities"
    xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="CrossPlatformCapabilities.MainPage">

    <Grid x:Name="LayoutRoot">
        <StackLayout>
            <Frame BackgroundColor="#2196F3" Padding="24" CornerRadius="0">
                <Label Text="Welcome to Xamarin.Forms!"
                       HorizontalTextAlignment="Center"
                       TextColor="White" FontSize="36"/>
            </Frame>
            <Label Text="Start developing now"
                    FontSize="Title" Padding="30,10,30,10"/>
            <!-- Text cut for spacing reasons, look in Visual Studio -->
            <Label Text="Make changes to your XAML file and save..."
                   FontSize="16" Padding="30,0,30,0"/>
            <Label FontSize="16" Padding="30,24,30,0">
                <Label.FormattedText>
                    <FormattedString>
                        <FormattedString.Spans>
                            <Span Text="Learn more at "/>
                            <Span
                             Text="https://aka.ms/xamarin-quickstart"
                             FontAttributes="Bold"/>
                        </FormattedString.Spans>
                    </FormattedString>
                </Label.FormattedText>
            </Label>
        </StackLayout>

        <local:NoNetworkSnackBarView x:Name="ConnectionStatusBar"/>
    </Grid>
</ContentPage>
```

If you now run the app and disable the internet connection on your device, you will see the Snackbar appear like in Figure 3. When you re-enable the connection, the Snackbar will disappear. Handling changes in the network connection is probably the most common task with mobile app development, but in this chapter, you have seen a step further, which is related to a nice user interface, rather than focusing on the information only.

Checking the battery level

Checking the battery level of the device can be an important task, depending on the type of app you develop. Both iOS and Android inform the user with an alert when the battery level goes down to 20 percent, so if your app only displays data and does not require the user to save information, that might be enough. Generally speaking, it is not a good idea to bore users with additional alerts on the same information already provided by the system. As an alternative, you could consider the implementation of a more discreet Snackbar that disappears after a few seconds if you need to make sure the user saves some information manually. This is a good idea if your app manages sensitive data and it is better to leave to the user the final responsibility of data handling, including data persistence.

Another possible scenario is the need of temporarily persisting some information in a silent way. To accomplish this, I will demonstrate how to use the **`EnergySaverStatusChanged`** event from the **`Xamarin.Essentials.Battery`** class. This event is raised when the device enters energy saver mode, which normally happens when the battery level goes down to 20%, but because it can also be manually enabled by the user, I will also show how to combine this with a check on the actual battery level. Before going into this, let's look at what the **`Battery`** class has to offer. This class exposes the following properties:

- **`ChargeLevel`**, of type **`double`**, which returns the current battery charge level between 0 and 1 (where 0 means completely discharged and 1 means 100 percent charged).
- **`BatteryState`**, of type **`BatteryState`**, which describes the state of the battery and returns the information described in Table 1.
- **`PowerSource`**, of type **`BatteryPowerSource`**, which describes how the device is being powered and returns the values described in Table 2.

Table 1: BatteryState enumeration values

Value	Description
`Charging`	The battery is currently charging.
`Discharging`	The battery is discharging. This is the expected value when the device is disconnected from a power source.
`NotCharging`	The battery is not being charged.
`NotPresent`	A battery is not available in the device (for example, on a desktop computer).

Value	Description
Unknown	The state could not be detected.
Full	The battery is fully charged.

Table 2: BatteryPowerSource enumeration values

Value	Description
Battery	The device is powered by the battery.
AC	The device is powered by an AC unit.
Usb	The device is powered through a USB cable.
Wireless	The device is powered via wireless charging.
Unknown	The power source could not be detected.

When one of the aforementioned properties changes, the **BatteryInfoChanged** event is raised and you will be able to read its status.

Note: On Android, you will need to select the `BATTERY_STATS` permission in the manifest before accessing information on the battery. UWP and iOS do not require additional steps.

The next example will focus on the **EnergySaverStatusChanged** event, for which you specify a handler as follows in the constructor of the **App** class:

```
Battery.EnergySaverStatusChanged += Battery_EnergySaverStatusChanged;
```

The event handler will check if the phone entered into the energy saving status, but because this can be also manually enabled by the user, it will also check if the battery level is equal to or lower than 20 percent. This is accomplished with the following code:

```
private void Battery_EnergySaverStatusChanged(object sender,
        EnergySaverStatusChangedEventArgs e)
{
    MessagingCenter.Send(this, "BatteryEvent",
        e.EnergySaverStatus == EnergySaverStatus.On
        && Battery.ChargeLevel <= 0.2);
}
```

In this case, any tasks related to persisting user settings or storing local information will actually happen only if there is a real need. A message is sent to subscribers, which can then react to the status change.

Reacting to the energy saving status in the user interface

The first thing that a page must do is subscribe to the message sent by the **App** class when the phone enters the energy saving status. In the current example, the following line must be added to the constructor of the MainPage.xaml.cs file:

```
MessagingCenter.Subscribe<App, bool>(this,
                "BatteryEvent", ManageBatteryLevelChanged);
```

The **ManageBatteryLevelChanged** method will be responsible for performing the actual actions. This is its code:

```
private void ManageBatteryLevelChanged(App arg1, bool arg2)
{
    FileHelper.WriteData("test data");
}
```

FileHelper is a program-defined class that must be implemented, and that exposes methods for reading and writing text files locally. Code Listing 4 shows the full code for the class.

Note: For Android, you will need to enable the *READ_EXTERNAL_STORAGE* ***and*** *WRITE_EXTERNAL_STORAGE* ***permissions in the app manifest.***

Code Listing 4

```
using System;
using System.IO;
using System.Threading.Tasks;
using Xamarin.Essentials;

namespace CrossPlatformCapabilities
{
    public static class FileHelper
    {

        public static string ReadData()
        {
            try
            {
                string fileName = Path.Combine(Environment.GetFolderPath(
                       Environment.SpecialFolder.LocalApplicationData),
                       "appdata.txt");

                string text = File.ReadAllText(fileName);

                return text;
```

```
            }
            catch (Exception)
            {
                return null;
            }
        }

        public static bool WriteData(string data)
        {
            try
            {
                string fileName = Path.Combine(Environment.GetFolderPath(
                       Environment.SpecialFolder.LocalApplicationData),
                       "appdata.txt");

                File.WriteAllText(fileName, data);

                return true;
            }
            catch (Exception)
            {
                return false;
            }
        }
    }
}
```

The code for the **ReadData** and **WriteData** methods is very simple: it opens a stream for reading and writing a file, and then it reads/writes the file content as text. The **WriteData** method was invoked by the **ManageBatteryLevelChanged** method in the user interface's code-behind file, which silently saves some user information when the battery level is very low. Such information can then be read back by invoking the **ReadData** method, which returns a string. In the current example, everything happens silently, but you can also consider implementing a Snackbar that gently informs the user about the battery level getting low, and about the automatic information backup. Using a text file for local persistence is obviously the simplest example possible, but you can go further by persisting user and app settings to the local storage or into a SQLite database. App settings and SQLite databases will be discussed in Chapters 2 and 4. Keep the **FileHelper** class in mind, because this will be extended in the next section with methods to work with the file system.

Working with files

Working with files is a very common requirement in mobile apps. Due to the natural system restrictions, what you will commonly be able to do is work with image files, video files, and document files, and with Xamarin.Essentials you will be able to do this cross-platform. This section discusses two topics: accessing data in the local app folders and picking up files.

Accessing files in the local app folders

In terms of cross-platform development, you will be able to access two file locations: the cache directory, where temporary files are stored, and the app data folder, where app data files are usually stored. Notice that the cache directory is not usually visible to the user, and the data it contains can be removed at any time, due to its temporary nature. For a better understanding of how reading files from these locations works, consider the following method that you can add to the **`FileHelper`** class, so that you can build a reusable class for file access:

```
public async static Task<string> ReadDataAsync()
{
    try
    {
        var mainDir = FileSystem.AppDataDirectory;
        string localData;

        using (var stream = await
FileSystem.OpenAppPackageFileAsync("appdata.txt"))
        {
            using (var reader = new StreamReader(stream))
            {
                localData = await reader.ReadToEndAsync();
            }
        }
        return localData;
    }
    catch (Exception)
    {
        return null;
    }
}
```

The **`FileSystem`** class exposes two properties, **`AppDataDirectory`** and **`CacheDirectory`**, which are both self-explanatory. It also exposes a method called **`OpenAppPackageFileAsync`**, which allows for reading the content of a file. With regard to this, there are important considerations to keep in mind:

- Only text files are supported.

- As the name implies, **`OpenAppPackageFileAsync`** allows for opening a file that is part of the app resources. Further, you can only open text files that were previously added to the platform-specific projects.
- Continuing from the previous point, for Android you will need to place the text file under the **Assets** folder and set its **Build Action** property with **AndroidAsset**. On iOS, the text file must be placed under the **Resources** folder and its **Build Action** set with **BundleResource**. On UWP, the file must be placed in the project folder, and its **Build Action** set with **Resource**.

Accessing files in the local app folders is useful, but it is even more useful learning how to select an existing file from the user's device.

Picking up files from the device

If your app needs to allow users to select files from the device, Xamarin.Essentials has the cross-platform solution for you.

Note: You will need Xamarin.Essentials 1.6 for this API. At the time of writing, this has not been yet released as a stable build, but you can install the 1.6.0-pre2 version from NuGet.

Let's start by supposing your app allows users to select and display an image file. You will need a button to select an image, a label to display the selected file name, and an **`Image`** view to display the image. Code Listing 5 shows the XAML for this.

Code Listing 5

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="CrossPlatformCapabilities.FilePickerPage">
    <ContentPage.Content>
        <Grid>
            <Grid.RowDefinitions>
                <RowDefinition Height="Auto" />
                <RowDefinition Height="Auto" />
                <RowDefinition />
            </Grid.RowDefinitions>

            <Button x:Name="PickButton" Clicked="PickButton_Clicked"
                    HorizontalOptions="Center"
                    Text="Pick an image"/>

            <Label x:Name="ImageNameLabel"
                   HorizontalTextAlignment="Center"
                   HorizontalOptions="Center" Grid.Row="1"/>
```

```
            <Image x:Name="PickedImage" Grid.Row="2"/>
        </Grid>
    </ContentPage.Content>
</ContentPage>
```

Nothing complex at all. Because you want to display the image and its file name, you might want to implement a class that stores this information from the selected file:

```
public class FileSelection
{
    public string FileName { get; set; }
    public ImageSource Image { get; set; }
}
```

Now it is time to write a method for file selection. The **Xamarin.Essentials.FilePicker** class exposes a method called **PickAsync**, which opens the system user interface for file selection, and returns an object of type **FileResult**. An instance of **FileResult** will contain the file name, the full path, and a stream object that contains the selected file. Code Listing 6 demonstrates how to invoke **PickAsync** and how to store the result in the **FileSelection** class we created previously.

Code Listing 6

```
private async Task<FileSelection> PickAndShowAsync(PickOptions options)
{
    try
    {
       FileResult result = await FilePicker.PickAsync(options);
       FileSelection fileResult = new FileSelection();
       if (result != null)
       {
           fileResult.FileName = $"File Name: {result.FileName}";
           if (result.FileName.EndsWith("jpg",
               StringComparison.OrdinalIgnoreCase) ||
               result.FileName.EndsWith("png",
               StringComparison.OrdinalIgnoreCase))
           {
               var stream = await result.OpenReadAsync();
               fileResult.Image =
                   ImageSource.FromStream(() => stream);
           }
       }
        return fileResult;
    }
```

```
    catch (Exception ex)
    {
        return null;
        // The user canceled or something went wrong
    }
}
```

The method also takes a parameter of type **PickOptions**, but this is discussed in the next paragraph. Key points to focus on now:

- **PickAsync** returns the selected file as a **FileResult** object.
- Built-in support is for .png and .jpg image files, so other types are filtered out (see the **if** block).
- The file name and the actual image are assigned to an object of type **FileSelection** and returned as the result of the method. The image in particular is returned as a stream.

The final piece is the **Clicked** event handler for the button, which will look like the following:

```
private async void PickButton_Clicked(object sender, EventArgs e)
{
    var result = await PickAndShowAsync(
        new PickOptions { PickerTitle="Pick a file",});
    this.ImageNameLabel.Text = result.FileName;
    this.PickedImage.Source = result.Image;
}
```

The key point here is obviously the invocation of the **PickAndShowAsync** method. Notice how an instance of the **PickOptions** allows for customizing the user interface for file selection. You can actually specify a different text for the picker and, as you will see in the next paragraph, this is also the place where you add support for different file types. If you run the code and select an image, you will see a result similar to Figure 4.

Figure 4: Displaying an image selected from the device

You can support different file types, as described in the next section.

Supporting custom file types

You can add support for file types different from images. This requires some platform-related specifications, but fortunately the picker API makes this simple. Suppose you want the picker to show .doc files. You would change the invocation to our method **`PickAndShowAsync`** as follows:

```
var fileType = new FilePickerFileType(new Dictionary<DevicePlatform,
    IEnumerable<string>>
{
    { DevicePlatform.iOS, new[] { "public.my.doc.extension" } },
    { DevicePlatform.Android, new[] { "application/doc" } },
    { DevicePlatform.UWP, new[] { ".doc", ".doc" } },
});

var result = await PickAndShowAsync(new PickOptions {
    PickerTitle="Pick a file", FileTypes = fileType  });
```

The **PickOptions** object is passed to the **PickAsync** method of the **FilePicker** class. Notice how, for each operating system, you need to supply a different syntax to identify the file type. The easiest are for Android, where you pass the MIME type, and for UWP where you simply pass the file extension. For iOS, I suggest you replace the file extension inside the string you see in the previous example. If you want to try yourself based on this example, you will need to remove the filter on .png and .jpg files.

Note: Of course, it is not enough to allow users to open different types of files. You also need to implement viewers or launch the appropriate app. The "Sharing files and documents" section later in the book will help you with the second option.

Development trick: Running code in the main thread

In order to keep the user interface active and capable of accepting the user input even when an app is busy doing something else, most operating systems (iOS, Android, and UWP are no exception) run the user interface on a dedicated thread—known as the main thread, user interface thread, or UI thread.

Note: For consistency with the API naming, I will refer to this thread as the main thread.

All the code that interacts with elements of the user interface should run within this thread; otherwise, exceptions would be raised because of invalid cross-thread calls. However, this does not mean that an app only runs on a single thread. For example, classes like **Battery** or **Connectivity** that you saw at the beginning of this chapter raise events on a separate thread to avoid a work overload on the main thread. If part of the code that runs on a separate thread needs to interact with elements of the user interface, that piece of code needs to run on the main thread instead.

For instance, an event handler might contain some logic and code that changes properties of UI elements. The logic part can stay on a separate thread, but the piece of code that works against UI elements needs to run on the main thread. Xamarin.Essentials simplifies running code in the main thread via the **MainThread** class. If you want to run code on the main thread, all you do is write something like this:

```
MainThread.BeginInvokeOnMainThread(() =>
{
    // Run your code on the main thread here..
});
```

The body for the **BeginInvokeOnMainThread** method is an **Action** that you can represent with a lambda expression, like in the previous snippet, or by explicitly declaring a method name as follows:

```
MainThread.BeginInvokeOnMainThread(DoSomething);
```

```
…
void DoSomething()
{
     // Run your code on the main thread
}
```

The fact is that you might not know if a piece of code needs to run on the main thread or not. Luckily enough, the **MainThread** class exposes the **IsMainThread** static property, which returns **true** if the current piece of code is already running on the main thread. You use it as follows:

```
if (MainThread.IsMainThread)

    DoSomething();

else

    MainThread.BeginInvokeOnMainThread(DoSomething);
```

The **MainThread** class is extremely useful. It will help you avoid errors and save time detecting which thread your code is running on.

Opening the browser and sending emails

Sometimes you need to open a browser on the device for navigating to a webpage, or your app might need to allow users to send emails. Xamarin.Essentials provides APIs for both scenarios, which are discussed in this section. For the next example, add a new **ContentPage** to the project with the XAML shown in Code Listing 7, which declares two simple buttons:

Code Listing 7

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             Padding="0,20,0,0"
             x:Class="CrossPlatformCapabilities.BrowserEmailSample">
    <ContentPage.Content>
        <StackLayout VerticalOptions="Center">
            <Button Text="Open webpage"
                    x:Name="BrowserButton"
                    Clicked="BrowserButton_Clicked"/>
            <Button Text="Open email client"
                    x:Name="EmailButton"
                    Clicked="EmailButton_Clicked"
                    Margin="0,15,0,0"/>
        </StackLayout>
    </ContentPage.Content>
```

```
</ContentPage>
```

The event handlers of each button will be implemented to work with the two scenarios taken into consideration.

Opening the browser

Xamarin.Essentials provides the **Browser** class, which makes it really simple to open an external browser pointing to the specified URI via the **OpenAsync** method. While you can display web content, including websites, inside the **WebView** control in Xamarin.Forms, there might be several reasons for displaying a website with an external browser. The first reason is certainly the fact that web browsers are more optimized to display websites than the **WebView**. I will explain other reasons with a practical example coming from my daily work. The main project I work on is an app for dialysis patients, and we had the requirement to provide links to official sources about COVID-19 from the World Health Organization (WHO) website. We made the decision to display such webpages inside an external browser for the following reasons:

- If we implemented our browsing system inside the app, the user would have the perception that contents from the WHO were offered by us, which is not true.
- In the unlucky (and unlikely) event the website was hacked, the user would have the perception that it was our fault, which is not true.
- Because of the sensitivity of the information, the user had to understand clearly who was the source of the information, and that would not be possible (or at least very difficult) with in-app browsing.
- Security is key for us, so we thought that delegating this responsibility to the browser was the best approach.

The simplest way to open a website inside a browser is by invoking the **OpenAsync** method as follows:

```
private async void BrowserButton_Clicked(object sender, EventArgs e)
{
    await Browser.OpenAsync("https://www.microsoft.com");
}
```

If multiple browsers are installed on your device, the system will show a user interface where you can select the browser for opening the website. With this simple line of code, a browser will be opened and will run externally. You can also open a web browser instance inside the app, changing the previous line of code as follows:

```
await Browser.OpenAsync("https://www.microsoft.com",
      BrowserLaunchMode.SystemPreferred);
```

The **`BrowserLaunchMode`** enumeration has two values: **`External`**, which is basically the default and works exactly like the first example, and **`SystemPreferred`**, which makes the browser run inside the app. Figure 5 shows an example based on iOS where on the left the browser is running externally. You can see a shortcut to go back to the app at the top-left corner, and on the right, the browser is running inside the app.

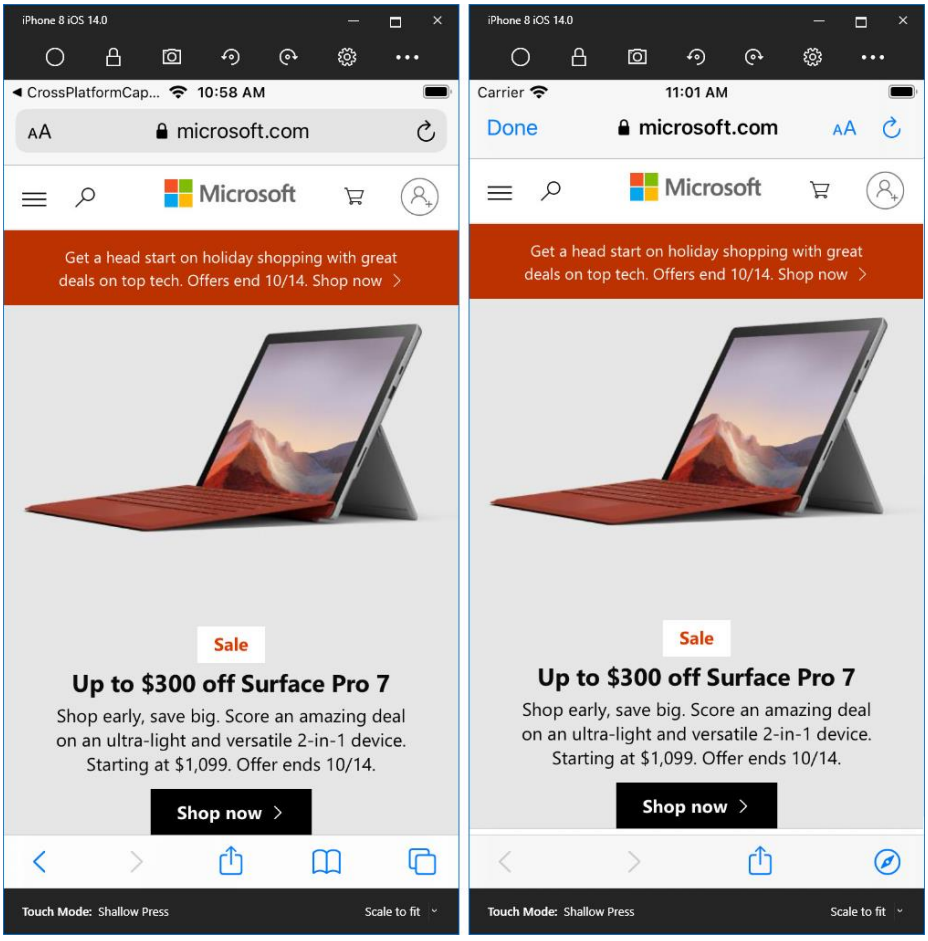

Figure 5: Opening the browser outside and inside the app

As you can see, when the browser runs externally you will have the browser controls available. When it runs inside the app, you will have fewer controls available, but the user does not need to switch between apps.

Sending emails

Similar to browsing external websites is sending emails. Creating a dedicated user interface would take time, and you would need to care about a lot of information. Xamarin.Essentials provides the **`Email`** class, which exposes the **`ComposeAsync`** method which allows for opening one of the installed email clients on your device. In its simplest form, the method works like this:

```
await Email.ComposeAsync();
```

This will open an email client on the device, and the user will need to supply all the email information manually. Another method overload allows for specifying a minimum set of information as follows:

```
await Email.ComposeAsync("Support request",
      "We have problems with the internet connection",
      "support@onecompany.com");
```

The first argument is the email subject, the second argument is the email body, and the third argument is an array of **string** containing the list of recipients. However, there is a third method overload that is much more powerful and gives you complete control of the email. This overload takes an argument of type **EmailMessage** and works as shown in the following example (which is also the event handler for the button in the sample app):

```
 private async void EmailButton_Clicked(object sender, EventArgs e)
 {
     EmailMessage message = new EmailMessage();
     message.Subject = "Support request";
     message.To = new List<string> { "support@onecompany.com" };
     message.Cc = new List<string> { "myboss@mycompany.com" };
     message.BodyFormat = EmailBodyFormat.PlainText;
     message.Body = "We have problems with the internet connection";

     await Email.ComposeAsync(message);
 }
```

As you can see, the **EmailMessage** class exposes self-explanatory properties and allows you to specify if the body should be plain text (**EmailBodyFormat.PlainText**) or HTML (**EmailBodyFormat.HTML**). You can define multiple recipients in the **To**, **Cc**, and **Bcc** properties via **List<string>** objects. The **EmailMessage** class also exposes the **Attachments** properties, which allows you to programmatically attach files to the email message.

The following code demonstrates how to add an attachment:

```
string attachmentPath =
    Path.Combine(Environment.GetFolderPath(
        Environment.SpecialFolder.MyPictures),
        "myimage.png");
EmailAttachment attachment = new EmailAttachment(attachmentPath);

message.Attachments = new List<EmailAttachment>();
message.Attachments.Add(attachment);
```

Before running the example, you need to add the following XML markup to the Android manifest if you plan for the app to run on Android 11 devices:

```
<queries>
  <intent>
    <action android:name="android.intent.action.SENDTO" />
```

```
    <data android:scheme="mailto" />
  </intent>
</queries>
```

Tip: For iOS, the `Email` class is only supported on physical devices. In the simulator, it will throw a `FeatureNotSupportedException`.

If you run the example, you will first see the operating system asking which email client you want to use (if there is more than one on the system), and then you will see a new email message with all the required information supplied and ready to be sent. Figure 6 shows an example from my device, where on the left there is the client selection, and on the right the email message.

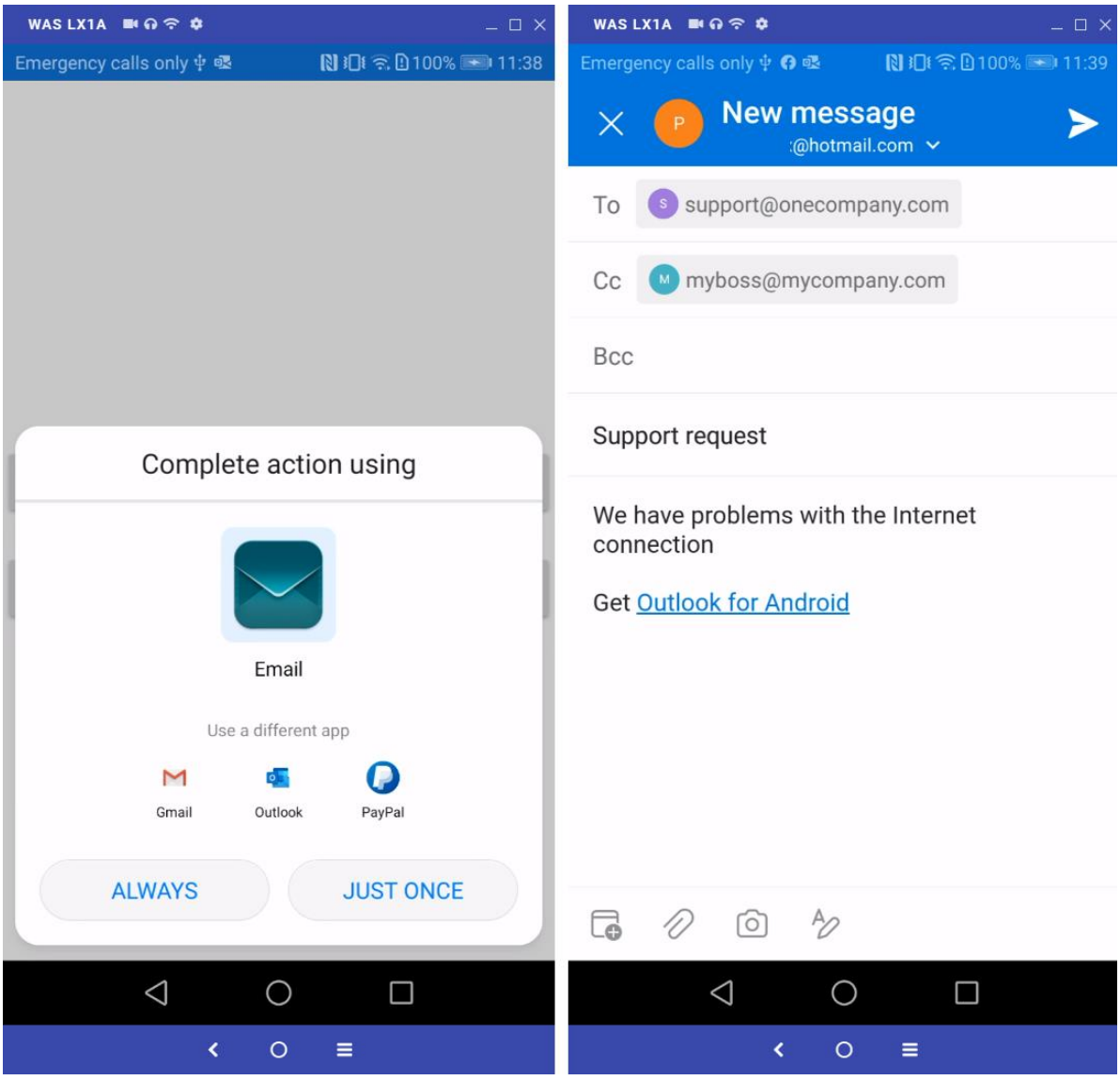

Figure 6: Email client selection and an email message ready to be sent

With a few lines of code, you can make your users able to send emails without the need of implementing a custom user interface and logic.

Chapter summary

Mobile apps in the real world often need to implement features that need to work cross-platform. In this chapter, you have seen how to build a Snackbar to inform users about the loss of network connection, and you have seen how to check for the battery level to make sure the app can persist important information. You have seen how to work with local files, and how you can open the browser and send emails using system resources that avoid the need of custom user interfaces and additional effort for developers. All these capabilities are provided by the Xamarin.Essentials library, and in the next chapter, you will learn other cross-platform capabilities from this library related to the app settings and secure storage.

Chapter 2 Local Settings: Preferences and Secure Storage

Storing user preferences and app settings locally is a very common requirement with mobile applications. There are tons of examples, which might include (but are not limited to) storing the last access date and time, the consent to use a cellular network when Wi-Fi is not available, color scheme preferences, and so on. These options are not generally required to be secure, but sometimes you might need to store an encrypted password or information that identifies the user. In this case, you do need to secure preferences and settings. The Xamarin.Essentials library provides types that make it extremely easy to store local settings in both ways. This chapter describes both scenarios and provides details on the classes, and their methods, you can use for these purposes.

Managing local preferences

Local preferences in Xamarin.Forms can be easily managed via the **`Preferences`** class from the **`Xamarin.Essentials`** namespace. The **`Preferences`** class stores preferences in the native local storages, more specifically:

- For Android, in the SharedPreferences.
- For iOS, in the NSUserDefaults.
- For UWP, in the ApplicationDataContainer.

On each platform, preferences are key/value pairs. The value can be of one of the primitive .NET types. Before understanding how preferences work, it is important to list the methods exposed by the **`Preferences`** class, as summarized in Table 3.

Table 3: Methods exposed by the Preferences class

Method	Description
`Set`	Adds a preference to the local settings.
`Get`	Gets a preference value from the local settings.
`Remove`	Removes the specified preference.
`Clear`	Deletes all the preferences for the current app.
`ContainsKey`	Detects if the specified preference exists in the storage given its key.

Tip: Uninstalling the app will also remove all local preferences.

Let's now start with an example. Suppose you want to store the last date and time the app was used. You can do this in the **OnSleep** method of the **App.xaml.cs** file as follows:

```
protected override void OnSleep()
{
    Preferences.Set("TimeOfLastUsage", DateTime.Now);
}
```

TimeOfLastUsage is the key, and the value is a **DateTime** object. Keys are always of type **string**. Retrieving a value from the local settings is easy. In the **OnStart** method, you can write something like the following:

```
internal static DateTime timeOfLastUsage;
protected override void OnStart()
{
    timeOfLastUsage = Preferences.Get("TimeOfLastUsage",
        DateTime.Now);
}
```

The generic **Get** method returns the value for the specified key and stores the result in a variable. **Get** also requires you to specify a default value in case the key does not exist in the storage, and this is true for all the supported data types. In this example, local preferences are used to store information that can be useful for the app logic at runtime, so it's not really a user preference.

Let's look at another example that will check if the phone is connected to a cellular network only, asking the user permission for using it and storing the result. Code Listing 8 contains the code that you want to add to the **MainPage.xaml.cs** file.

Code Listing 8

```
        private bool CheckCellularConnection()
        {
            var profiles = Connectivity.ConnectionProfiles;
            return profiles.Count() == 1 &&
                   profiles.Contains(ConnectionProfile.Cellular);
        }

        private bool useCellularNetwork;
        protected override async void OnAppearing()
        {
            if(CheckCellularConnection())
            {
```

```
                if (!Preferences.ContainsKey("UseCellularNetwork"))
                {
                    bool result = await DisplayAlert("Warning",
                        "Do you agree to using cellular data when Wi-
Fi is not available?",
                        "Yes", "No");
                    Preferences.Set("UseCellularNetwork", result);
                    useCellularNetwork = result;
                }
                else
                    useCellularNetwork =
                         Preferences.Get("UseCellularNetwork", false);
            }
        }
```

The **CheckCellularConnection** method checks if only one connection is available, and if this is a cellular network. The remaining code is in the **OnAppearing** method for convenience so that the code is executed at the app startup, but you will likely create a separate method that will be invoked when the **ConnectivityChanged** event of the **Connectivity** class is raised.

In this second example, the code invokes the **Preferences.ContainsKey** method to detect if the preference was already saved, and then it still invokes the **Set** and **Get** methods to store and retrieve the preference value. The **Preferences** class is intended to store small pieces of information represented by primitive .NET types. If you need to locally save and retrieve more complex and structured data, a good option is to use a SQLite database, as discussed in Chapter 4.

Securing preferences and settings: the secure storage

Sometimes you need to store information locally, but you also need the information to be secured. Each of the supported operating systems offers a secure place to store encrypted information and the Xamarin.Essentials library provides the **SecureStorage** class, which exposes methods to work with secure storages in a cross-platform way. There are a few differences from the non-secured preferences discussed in the previous section:

- While preferences are stored inside the app's own cache, the secure storage can be accessed by multiple applications.
- As an implication, data you save in the secure storage is not removed when you uninstall an app. This is extremely important to consider.
- You still work with key/value pairs, but the value can be of type **string** only.

The native implementation for the secure storage can be summarized as follows:

- For iOS, it is located in the **KeyChain**.
- For Android, it is located in the **KeyStore**.

- For UWP, data is encrypted with the **`DataProtectionProvider`** algorithms.

When working with iOS, if you want to test your apps on a simulator, you will need to perform a couple more steps:

- Double-click the **Entitlements.plist** file in Solution Explorer.
- When the editor opens, locate the **KeyChain** element and enable it.
- In the iOS project properties, select the **iOS Bundle Signing** tab and assign the **Entitlements.plist** file as the source for **Custom Entitlements**.

These entitlements should be removed when working on physical devices, and before distribution. The biggest benefit of the secure storage is that it automatically encrypts the information with the highest levels of security possible offered by each operating system.

The **`SecureStorage`** class exposes two simple methods: **`SetAsync`** and **`GetAsync`**. The first method saves the specified key and value, and the second method retrieves the value for the specified key. If the key does not exist, it returns **`null`**, but you cannot specify a default value like you do with local preferences. To understand how the class works, suppose you have the following user interface where the user can enter a password:

```
<StackLayout VerticalOptions="CenterAndExpand"
             HorizontalOptions="FillAndExpand"
             Orientation="Vertical">
    <Label Text="Enter your password: "
           FontSize="Large"
           Margin="0,20,0,0"/>
    <Entry IsPassword="True"
           x:Name="PasswordEntry"
           Margin="0,10,0,0"/>
    <Button Text="OK" Margin="50,20,50,0" x:Name="OkButton"
            Clicked="OkButton_Clicked"/>
</StackLayout>
```

The **`Clicked`** event handler for the button contains the code that stores the password in the local storage:

```
private async void OkButton_Clicked(object sender, System.EventArgs e)
{
    // Perform password validation here...

    await SecureStorage.SetAsync("Password", PasswordEntry.Text);
}
```

Obviously, you will implement your logic for password validation, but for the sake of simplicity, the code stores the password directly. The key points are the two string parameters—key and value—for the **`SetAsync`** method, and the usage of the **`await`** operator, as this is an asynchronous call. The code you need to write to retrieve information from the secure storage looks like the following, related to the current example:

```
// Assuming you have declared a bool field called hasPassword
string password = await SecureStorage.GetAsync("Password");
if (password != null)
    hasPassword = true;
```

GetAsync returns the value for the specified key if the key exists in the secure storage; otherwise, it returns **null**, so a null check is necessary to avoid exceptions. As you can see, reading and writing data against the secure storage is very straightforward, though I recommend you use this feature only with very sensitive information.

Secure storage tips and tricks

As I mentioned previously, when you uninstall an app that stores information in the secure storage, such information is not removed because the secure storage is at the operating system level, not at the app level (though the OS knows what app is associated to the key). You cannot control when an app is uninstalled, but you can control if the app is running on the device for the first time (including after reinstall), checking if a key already exists in the secure storage so that it can be removed, and providing a fresh environment to the user. The following method demonstrates how to accomplish this:

```
private async Task CheckAppFirstRun()
{
    if (VersionTracking.IsFirstLaunchEver)
    {
        string password = await SecureStorage.GetAsync("password");
        if (password != null)
            SecureStorage.Remove("password");
    }
}
```

The **VersionTracking** class from the Xamarin.Essentials library provides properties and methods that help working with the app's versions and builds. **IsFirstLaunchEver** returns **true** if the app is running for the first time on the device. The **SecureStorage.Remove** method deletes the key and its value from the secure storage. You can avoid this code if you instead want to reuse the value of an existing key.

Note: Remember to invoke `VersionTracking.Track` the very first time; otherwise, the class will not be able to generate version information.

Chapter summary

Storing preferences and user settings is a very common task in mobile applications, and the Xamarin.Essentials library provides the **`Preferences`** class, which allows saving key/value pairs in the app's local storage. Supported data types are .NET primitive types. If the information you need to store locally requires encryption, you can leverage the operating system's secure storage, for which the Xamarin.Essentials library offers the **`SecureStorage`** class, which supports key/value pairs of type **`string`**.

In the next chapter, you will see a very practical example of using the secure storage, which is the implementation of biometric authentication.

Chapter 3 Implementing Biometric Authentication

Biometric authentication is the option to log into an application using the fingerprint sensor or the facial recognition feature, also known as face ID, without the need to type a username and password every time. This option is very common and popular in modern applications that require authentication, and you will likely need to implement it yourself in your apps. There are several reasons for doing this. The first reason is simplifying the user experience by leveraging sensors on the device. The second reason, which is basically connected to the first, is that typing credentials every time on a small keyboard might be annoying, and the user might quickly become tired of using your app—even if it is the best app on the market. This assertion might seem strange, but it's not. It actually comes from research on analytics and user experience. With the incredible number of apps available, you might build a killer app on a certain topic, but if it's not easy, fast, and fun to use, people will move to another app on the same topic. Login features are the user's first contact with an app, and starting the app must be as simple, fast, and smooth as possible. You will agree that using the fingerprint or face ID is much faster than typing a username and password every time. This chapter explains how to implement biometric authentication in Xamarin.Forms, adding some steps for local password storage, with some logic that you will be able to reuse with web services, custom APIs, or even Azure.

Setting up libraries for biometric authentication

Xamarin.Forms does not include a built-in API that allows for implementing biometric authentication in a cross-platform way, so in theory, you should write platform-specific code. However, within the large ecosystem of plugins for Xamarin.Forms, you can use the Biometric/Fingerprint plugin for Xamarin, an open-source project by Sven-Michael Stübe, which is free and easy to use. It offers a cross-platform approach to leveraging all the biometric authentication features on even the most recent devices.

With this in mind, create a new Xamarin.Forms project using the **Blank App** template. Open the NuGet Package Manager user interface, and then locate and install the **Plugin.Fingerprint** package to all the projects in the solution (see Figure 7).

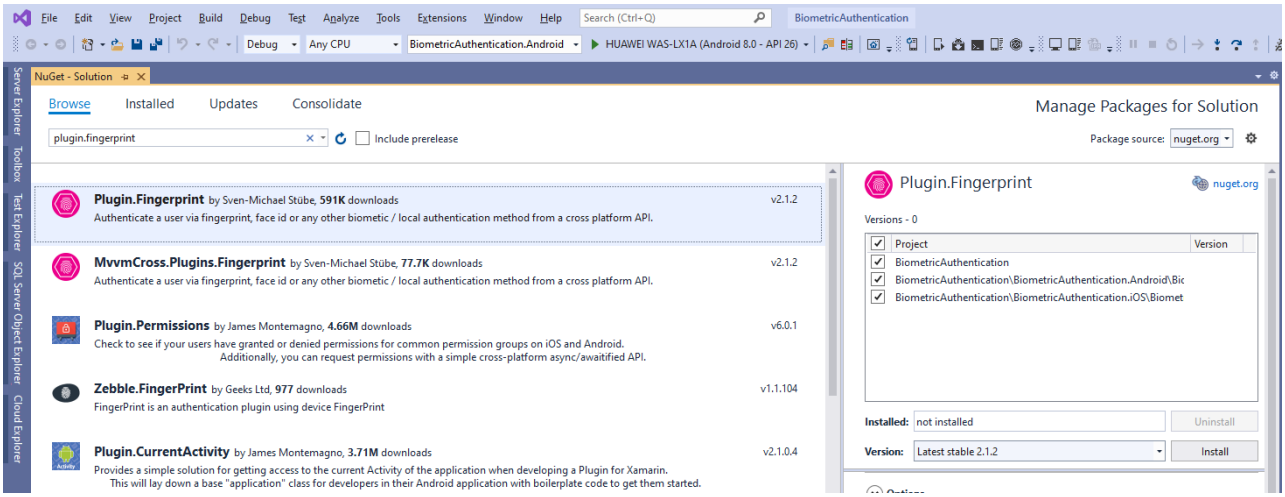

Figure 7: Adding the Plugin.Fingerprint library to the solution

Additional setup for Android

For Android, you will need to install the following additional NuGet packages to the project:

- Xamarin.AndroidX.Browser
- Xamarin.AndroidX.Legacy.Support.V4
- Xamarin.AndroidX.Lifecycle.LiveData
- Xamarin.Google.Android.Material

Each package will install its own dependencies. This is required by the Plugin.Fingerprint library, which relies on the most recent Android support libraries. Additionally, Android API supports biometric authentication since Level 23 (Android 6.0), which requires explicit user permission.

Open the Android project properties and select the **Android Manifest** tab. Make sure the **Minimum Android Version** is set to **Android 6.0** (API Level 23, Marshmallow) or later, as shown in Figure 8. Obviously, later versions are OK, but earlier versions do not support biometric authentication.

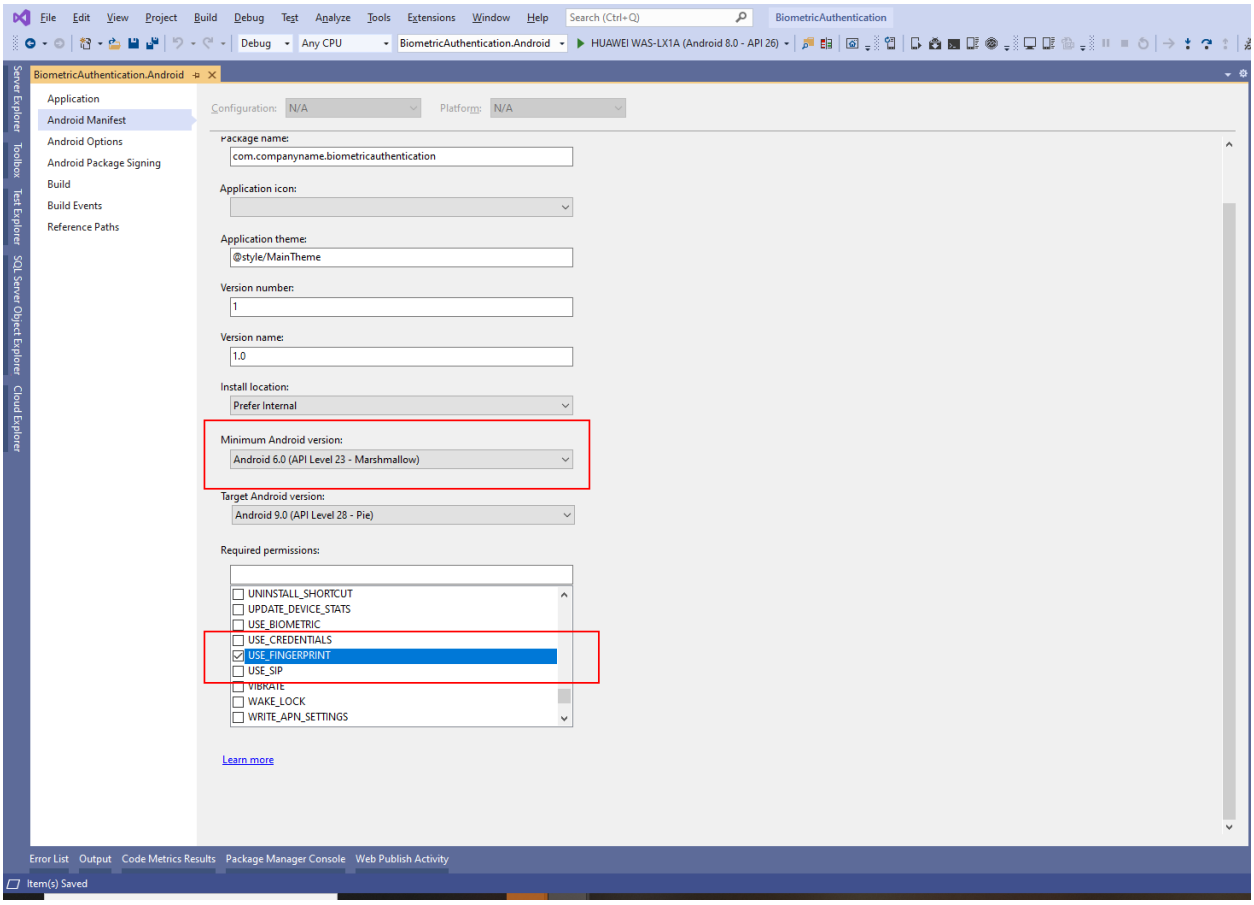

Figure 8: Setting Android API and permissions

In the **Required permissions** list, select the **USE_FINGERPRINT** permission (see Figure 8).

The logic of the sample project

The biometric authentication APIs in each system tell developers if a user is recognized by the device based on the fingerprint or face ID. Being recognized by the device does not mean being recognized, authenticated, and authorized by your application or your web services. In real-world development, the flow would be the following:

1. The user registers, providing a username and a password. These credentials are usually validated against a web service and stored inside a database, on-premises, or in-cloud, and will be passed by the app every time it needs to access secured information.
2. The credentials supplied at registration are also usually encrypted and stored locally on the device.
3. After registration, when the user wants to manually log in, the app looks for a secured password on the device. If not found, the user is not registered. If found, the app makes a first match between the password typed at login with the secured, local one. If they match, the password is included in the calls to APIs, web services, or any remote service and undergoes a server-side validation before getting the information.

4. With biometric authentication, the app looks for a secured password on the device. If not found, the user is not registered. If found, it means the user is registered, so the biometric authentication user interface is shown. If the device recognizes the user, the local, secure password is sent to any API calls, web services, or any remote service and undergoes a server-side validation before getting the information.

There is nearly an infinite number of scenarios and web services, APIs, and cloud services you might be using in your daily development, so the perfect server-side example does not exist, and would be outside the scope of this book. For this reason, the sample application will simulate the inclusion of the password to be sent to a service, but will still show the entire flow of registration, login, and biometric authentication with password management.

Creating a sample user interface

Based on the considerations of the previous paragraph, the user interface of the sample application consists of three pages:

- The auto-generated main page will serve as the welcome page and will offer two buttons: one for registration, and one for login.
- One page for registration, where the user will be able to supply and save a password. For the sake of simplicity, no username will be required.
- One page for the login part, with both the options to log in by typing the password manually and with biometric authentication.

Now, add two **ContentPage** items to the .NET Standard project: one called **RegisterPage.xaml**, and one called **LoginPage.xaml**.

Note: As you will see in the next code snippets and listing, the example is focusing on features, not on the beauty of the user interface. A professional designer will provide you with specific icons for biometric authentication features that you can use instead of a regular button.

For the **MainPage.xaml** file, the code is the following:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="BiometricAuthentication.MainPage" Padding="0,20,0,0">
    <StackLayout>
        <Button Text="Register" x:Name="RegisterButton"
                Clicked="RegisterButton_Clicked"/>
        <Button Text="Login" x:Name="LoginButton"
                Clicked="LoginButton_Clicked"/>
    </StackLayout>
</ContentPage>
```

The event handlers for the two buttons are extremely simple:

```
private async void RegisterButton_Clicked(object sender, EventArgs e)
{
    await Navigation.PushAsync(new RegisterPage());
}

private async void LoginButton_Clicked(object sender, EventArgs e)
{
    await Navigation.PushAsync(new LoginPage());
}
```

This assumes that you wrap the assignment of the **MainPage** property with a **NavigationPage**, inside the constructor of the **App** class, as follows:

```
MainPage = new NavigationPage(new MainPage());
```

In the **RegisterPage.xaml** file, replace the content of the **StackLayout** with the following:

```
<Label Text="Set your password:"/>
<Entry IsPassword="True" x:Name="PasswordEntry" />
<Button Text="Save password" x:Name="SavePasswordButton"
        Clicked="SavePasswordButton_Clicked"/>
```

In the **LoginPage.xaml** file, replace the content of the **StackLayout** with the following:

```
<Label Text="Enter your password:" />
<Entry x:Name="PasswordEntry" IsPassword="True" />
<Button x:Name="PasswordLoginButton"
        Text="Login with password"
        Clicked="PasswordLoginButton_Clicked" />
<Button x:Name="BiometricLoginButton"
        Text="Login with biometric authentication"
        Clicked="BiometricLoginButton_Clicked"/>
```

There is really nothing complex in the XAML for both pages, and you are totally free to rearrange the layout inside the page. The C# event handlers for both pages are discussed in the next sections, as part of the implementation logic.

Enforcing requirements for biometric authentication

As I mentioned previously, biometric authentication is useful for understanding if the device recognizes the user, but does not guarantee that the user is also registered or authenticated on an application. So it's a good idea to enforce the requirements for the user to be able to use biometric authentication. To accomplish this, when the user registers with an app, the credentials can be stored inside the device's secure storage. As you learned in Chapter 2, the secure storage provides a high level of encryption and security, but nothing prevents you from encrypting the password with an algorithm and storing the resulting string in the secure storage. (This is not done in this chapter.)

Note: Because the logic relies on the secure storage, if you work with an iOS simulator, remember to set up the keychain entitlement, as you learned in Chapter 2.

You might argue that saving a password in the local secure storage is not a good idea. Most apps that both require authentication and offer biometric authentication options do this, and the reason is simple: if the user does not enter credentials manually, but they are required by API calls or web services, the app needs to take these from somewhere, even for the very first call.

Open the **RegistrationPage.xaml.cs** file and add the following event handler for the **`SavePasswordButton`** object:

```
private async void SavePasswordButton_Clicked(object sender, EventArgs e)
{
    // Add more validation logic here...
    if (!string.IsNullOrEmpty(PasswordEntry.Text))
    {
        await SecureStorage.SetAsync("P", PasswordEntry.Text);
        await DisplayAlert("Success", "Password saved", "OK");
        await Navigation.PopAsync();
    }
}
```

You can perform your own validation logic on the entered password, such as length or minimum security requirements. It is saved inside the secure storage with a key that is not very readable, and this is intentional. We trust the secure storage, but more security is better. So, calling the key **`P`** instead of **`Password`** can be a good idea. When done, an alert is shown, and the code navigates back to the welcome page. This was the easiest part. Now let's go into more complex logic.

Implementing biometric authentication

It is not possible to provide only biometric authentication options. Older devices might not have sensors, or the user might not have configured the hardware. If the sensor is broken, for example, or the user wears gloves (in the case of fingerprint), they must have an option to log in manually. So, the example will show how to work with both manual login and biometric login.

Open **LoginPage.xaml.cs**. There is some code common to both scenarios. For example, the following code detects if a password was securely stored previously, which means the user had registered:

```
private async Task<bool> IsPasswordSetAsync()
{
    string result = await SecureStorage.GetAsync("P");
    return result != null;
}
```

This will be invoked in the next paragraphs.

Authentication with password

In the case of a login scenario where the user types their credentials manually, you will need to validate the entered password and then send it to a web API or web service. In the **LoginPage.xaml.cs** file, add the following method:

```
private async Task<bool> IsLocalPasswordValidationPassing(string password)
{
    string localPassword = await SecureStorage.GetAsync("P");
    return localPassword == password;
}
```

Given the specified password, the method checks if it matches the password stored in the secure storage and returns the result of the comparison.

Note: A password, as well as regular data, should be validated on the front end, on the back end, and on the data store—something that is not possible to demonstrate here, but certainly to keep in mind.

This method is invoked by the event handler for the **PasswordLoginButton** object, whose code is the following:

```
private async void PasswordLoginButton_Clicked(object sender, EventArgs e)
{
    bool isPasswordSet = await IsPasswordSetAsync();
    if (!isPasswordSet)
    {
        await DisplayAlert("Error", "Password not set, register first",
                           "OK");
        await Navigation.PopAsync();
        return;
    }

    if (!string.IsNullOrEmpty(PasswordEntry.Text))
    {
        bool localValidation =
            await IsLocalPasswordValidationPassing(PasswordEntry.Text);
        if (localValidation)
        {
            await DisplayAlert("Success", "Authenticated!", "OK");
            // Do login here...
        }
    }
}
```

If the password is not set, as per the **IsPasswordSetAsync** method explained in the previous section, it means the user is not registered, so the code returns to the calling page. Otherwise, if a formal validation of the password passes, **IsLocalPasswordValidationPassing** checks if the entered password matches the one stored in the secure storage. If they match, the password is accepted and sent to any real login service.

Authentication with fingerprint and face ID

About biometric authentication: the **CrossFingerprint** singleton class, through its **Current** property, provides methods and types that make it very easy to work with biometric hardware. The first thing to do is check if this is available on the device. Availability depends on several factors, such as hardware sensors available on the device, sensors enabled and configured, and API support. Checking availability is accomplished via a method called **GetAvailabilityAsync**, which returns an object of type **FingerprintAvailability**. The following method uses both to detect if biometric authentication is available:

```
private async Task<bool> CheckIfBiometricAuthIsAvailableAsync()
{
    FingerprintAvailability isBiometricAuthAvailable =
        await CrossFingerprint.Current.GetAvailabilityAsync();
    return isBiometricAuthAvailable == FingerprintAvailability.Available;
}
```

Tip: The `FingerprintAvailability` enumeration also allows you to understand if face ID is available, not just fingerprint. Do not get confused by its name.

GetAvailabilityAsync checks if the operating system's API supports biometric authentication, and then it checks if the user has given permissions, if the device has sensors, and finally, if the user has enabled and configured the sensor. Table 4 lists values from the **FingerprintAvailability** enumeration.

Table 4: FingerprintAvailability enumeration

Value	Description
Available	Biometric authentication can be used.
NoImplementation	The plugin has no implementation for the current platform.
NoApi	The operating system's API does not support biometric authentication.
NoPermission	The app is not allowed to access biometric hardware.
NoSensor	The device has no biometric hardware.
NoFingerprint	Biometric authentication has not been set up.

Value	Description
NoFallback	The fallback user interface has not been set up.
Unknown	An error occurred and could not be mapped to any of the other values.
Denied	The user has denied the usage of the biometric authentication.

The next step is writing the code for the **BiometricLoginButton** object, which is the following:

```
private async void BiometricLoginButton_Clicked(object sender, EventArgs e)
{
    bool isPasswordSet = await IsPasswordSetAsync();
    if (!isPasswordSet)
    {
        await DisplayAlert("Error", "Password not set, register first",
            "OK");
        await Navigation.PopAsync();
        return;
    }

    bool biometricAuthAvailability = await
         CheckIfBiometricAuthIsAvailableAsync();
    if (!biometricAuthAvailability)
    {
        await DisplayAlert("Error",
              "Biometric authentication is not available.", "OK");
        return;
    }

    await AuthenticateAsync();
}
```

Like for the manual login button, a first check is made to see if the user is registered; otherwise, the app goes back to the main page. If the user is registered, the code checks if biometric authentication is available by invoking the **CheckIfBiometricAuthIsAvailableAsync** method explained previously. If available, a new method called **BiometricAuthenticationAsync** is invoked. The following is the code for this method:

```
private async Task BiometricAuthenticationAsync()
{
    var authRequest = new AuthenticationRequestConfiguration
        ("Biometric authentication", "Log in with fingerprint or face ID");
    FingerprintAuthenticationResult result =
        await CrossFingerprint.Current.AuthenticateAsync(authRequest);
    if (result.Authenticated)
```

```
    {
        await DisplayAlert("Success", "Authenticated!", "OK");
        // Do login here...
    }
    else
        await DisplayAlert("Error", $"Reason: {result.ErrorMessage}", "OK");
}
```

The **CrossFingerprint.AuthenticateAsync** method is invoked to prompt the user with the system user interface for biometric authentication. Depending on the hardware installed on your device, the API will know which sensor to invoke. This method takes an object of type **AuthenticationRequestConfiguration**, whose constructor needs you to specify the title and text to be displayed in the user interface, where supported. This object allows for further authentication configuration, but most of the time you will not need to do this.

AuthenticateAsync returns the result of the biometric authentication with an object of type **FingerprintAuthenticationResullt**. The **Authenticated** property, of type **bool**, is true if the user was recognized. If not, you can get a description of the error through the **ErrorMessage** property, of type **string**, and the reason behind the error via a property called **Status**, of type **FingerprintAuthenticationResultStatus**. Values might be **Failed**, **Canceled**, **TooManyAttempts**, or **Denied**. **Status** is set with **Succeeded** when **Authenticated** is **true**.

Detecting the available hardware

If you need to understand what kind of hardware is supported on the device, you can invoke the **GetAuthenticationTypeAsync** method as follows:

```
AuthenticationType authTypes =
      await CrossFingerprint.Current.GetAuthenticationTypeAsync();
```

It returns an object of type **AuthenticationType**, an enumeration whose values can be **None**, **Fingerprint**, and **Face**.

Running the sample app

After a bit of work, you can run the sample app and see how it looks. First, you will need to set and save a password in the registration page, or an error message will be displayed. Next, if you try to log in with biometric authentication, you will see the operating user interface for this feature. Figure 9 shows a fingerprint example based on an Android device.

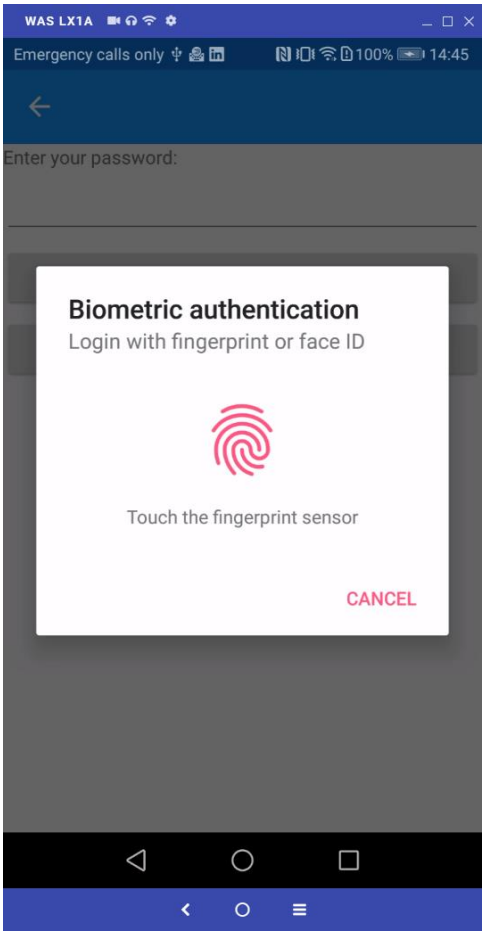

Figure 9: Biometric authentication on Android

Remember that the biometric authentication user interface can vary depending on your operating system version and system theme, especially on Android.

Chapter summary

Biometric authentication improves the user experience by leveraging the device hardware and avoiding the need to manually type user credentials every time. With the Plugin.Fingerprint library, it's easy to implement biometric authentication using the **`CrossFingerprint`** class and its methods, such as **`GetAvailabilityAsync`** and **`AuthenticateAsync`**. The complexity of the logic depends on your specific scenario, but now you know how to cover a topic that Xamarin.Forms does not natively.

The secure storage, widely used in this chapter, is not the only secure place where you can store passwords and data in general. In the next chapter, you will learn how to work with secured SQLite local databases.

Chapter 4 Local Data Access with SQLite

More often than not, applications work with data—and mobile apps are no exception. In many cases, mobile apps exchange data over networks and take advantage of cloud storage and services, such as push notifications. However, there are situations in which mobile apps only need to store data locally. For app settings, you can use the techniques described in Chapter 2. In the case of complex, structured data, applications need a different way to store information.

The good news is that you can easily include local databases in your mobile app using SQLite. This is an open-source, lightweight, serverless database engine that makes it simple to create local databases and perform operations on data. Information is stored inside tables, and data operations can be performed by writing C# code and LINQ queries. SQLite perfectly suits cross-platform development because it's a portable database engine, it's pre-installed on both iOS and Android, and it can be easily deployed to Windows.

In this chapter I will describe how to implement local data access with SQLite, but with a step further: securing the database with a password.

> ***Note: The code example provided in this chapter is based on the Model-View-ViewModel pattern. If you are not familiar with it, you can read an introduction from the Xamarin.Forms Succinctly ebook. This is important advice, because I will assume you already know some concepts.***

Using SQLite on Universal Windows Platform

While the SQLite engine is pre-installed on Android and iOS systems, it needs to be deployed to Universal Windows Platform systems. This is actually a very easy task. All you need to do is install a Visual Studio extension called **SQLite for Universal Windows Platform**. In Visual Studio 2019, select **Extensions** > **Manage Extensions**, and then search for **SQLite**. Figure 10 shows the extension you need to install.

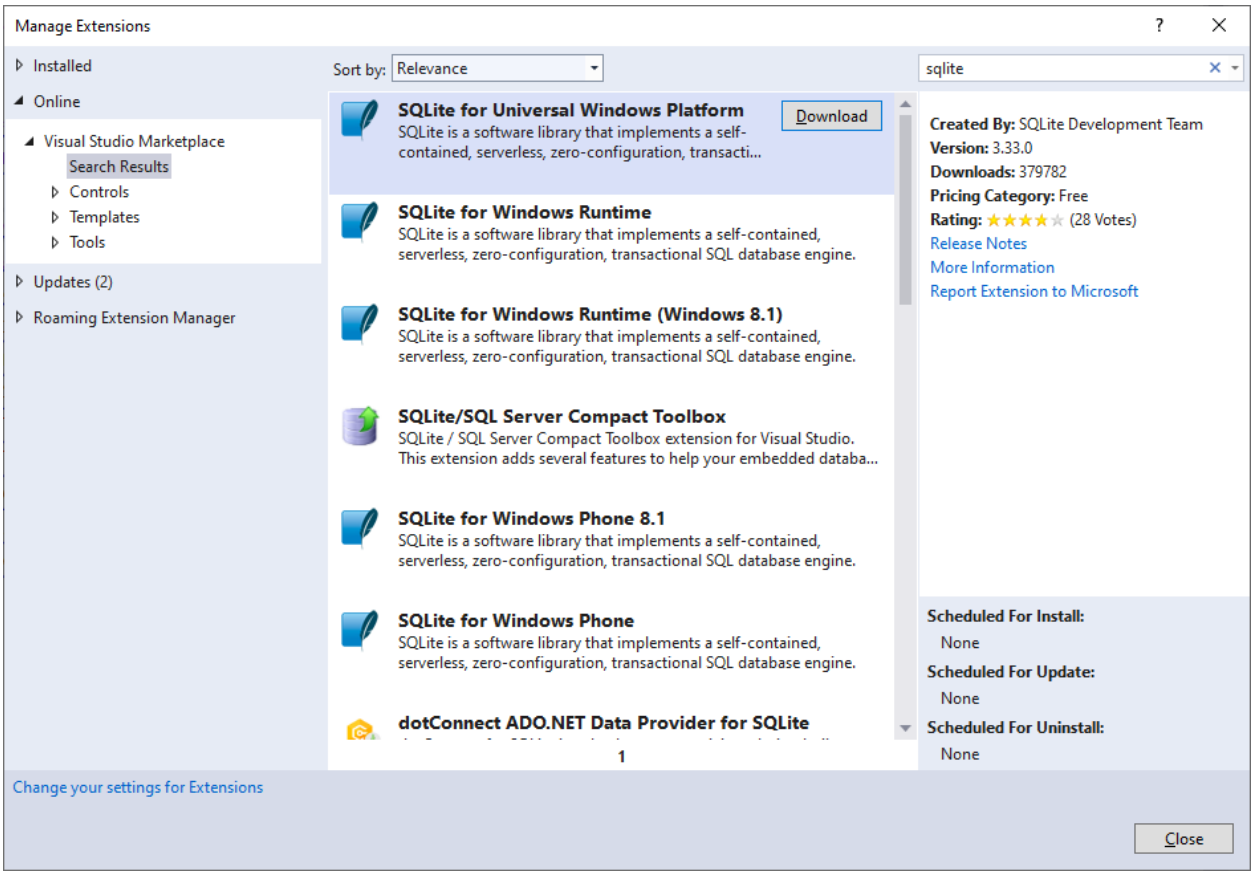

Figure 10: The SQLite extension for UWP

Other versions of the extension can be ignored, as they are not intended to work with the latest versions of Xamarin.Forms. Once the extension is installed, it will make sure to package and deploy the SQLite engine to UWP along with the app binaries.

Choosing and installing the SQLite libraries

SQLite libraries targeting C# and .NET for your apps are available through NuGet. There are several editions, each targeting different platforms. For Xamarin.Forms, you have a couple of options:

- **sqlite-net-pcl**: A cross-platform, open-source library that enables creating databases, reading data, and writing data. Don't be confused by the "pcl" (portable class library) literal; the library works with .NET Standard projects as well.
- **sqlite-net-sqlcipher**, also known as SQLCipher: Works like **sqlite-net-pcl**, but also adds the ability to secure the database with a password.

Securing the local database is a very good idea, especially if you store sensitive data, or simply because your company might want to add an additional level of security. Therefore, the sample project will be based on the **sqlite-net-sqlcipher** library.

Note: SQLCipher used to be a paid library and has been recently open-sourced. A paid version still exists, and the subscription offers technical support, custom builds, and additional options for the enterprise.

Assuming you have created a new blank Xamarin.Forms project, open the NuGet package manager in Visual Studio 2019 and install the **sqlite-net-sqlcipher** NuGet package to all the projects in your solution. NuGet will also install a number of dependencies, which are not necessary to cover in this chapter.

Tip: All the code that you will see in the next sections works exactly the same for both SQLite and SQLCipher libraries. The difference is that the latter allows for securing databases with a password, as you will see shortly. This means that you can reuse the code even if you do not need a secured database.

Setting up the database connection

Your code will access a SQLite database through a connection string, exactly like it would do with any other database. So, the connection string is the first thing you need to build. Because a SQLite database is a file that resides in a local folder, constructing the connection string requires the database pathname.

Though most of the code you will write is shared across different platforms, the way Android, iOS, and Windows handle pathnames is different, so building the connection string requires platform-specific code. You then invoke the connection string via the Xamarin.Forms dependency service. Because the example discusses SQLCipher, the connection string is also the place where you supply a password.

In the .NET Standard project, add a new interface called **IDatabase-Connection.cs** and write the following code:

```
public interface IDatabaseConnection
{
    SQLite.SQLiteConnection DbConnection();
}
```

This interface exposes a method called **`DbConnection`**, which will be implemented in each platform project, and will return the proper connection string. The next step is adding a **`DatabaseConnection`** (or a name of your choice) class to each platform project that implements the interface and returns the proper connection string, based on a sample database called **CustomersDatabase.db3**, where .db3 is the extension that identifies SQLite databases.

Code Listing 9 contains the code for Android, Code Listing 10 contains the code for iOS, and Code Listing 11 contains the code for UWP. The necessary comments will follow the listings.

Code Listing 9

```
using LocalDataAccess.Droid;
using SQLite;

[assembly: Xamarin.Forms.Dependency(typeof(DatabaseConnection))]
namespace LocalDataAccess.Droid
{
    public class DatabaseConnection : IDatabaseConnection
    {
        public SQLiteConnection DbConnection()
        {

            string dbName = "CustomersDatabase.db3";

            string path = Path.Combine(System.Environment.
              GetFolderPath(System.Environment.
              SpecialFolder.Personal), dbName);

            SQLiteConnectionString connectionString =
                new SQLiteConnectionString(path, false, "p@$$w0rd");
            return new SQLiteConnection(connectionString);
        }
    }
}
```

Code Listing 10

```
using Foundation;
using LocalDataAccess.iOS;
using SQLite;
using System;
using System.IO;

[assembly: Xamarin.Forms.Dependency(typeof(DatabaseConnection))]
namespace LocalDataAccess.iOS
{
    public class DatabaseConnection : IDatabaseConnection
    {
        public SQLiteConnection DbConnection()
        {
            string dbName = "CustomersDatabase.db3";
            string documentsPath = NSFileManager.DefaultManager.GetUrls(NSS
earchPathDirectory.DocumentDirectory, NSSearchPathDomain.User)[0].Path;
```

```
            var path = Path.Combine(documentsPath, dbName);

            SQLiteConnectionString connectionString =
                new SQLiteConnectionString(path, false, "p@$$w0rd");

            return new SQLiteConnection(connectionString);
        }
    }
}
```

Code Listing 11

```
using SQLite;
using Xamarin.Forms;
using LocalDataAccess.UWP;
using Windows.Storage;
using System.IO;
[assembly: Dependency(typeof(DatabaseConnection))]
namespace LocalDataAccess.UWP
{
  public class DatabaseConnection: IDatabaseConnection
  {
    public SQLiteConnection DbConnection()
    {
      string dbName = "CustomersDatabase.db3";
      string path = Path.Combine(ApplicationData.
        Current.LocalFolder.Path, dbName);


      SQLiteConnectionString connectionString =
        new SQLiteConnectionString(path, false, "p@$$w0rd");
        return new SQLiteConnection(connectionString);
    }
  }
}
```

What the code does on each platform project is obtain the path for a local folder where the database will be created. For Android, this is the user's personal folder, and it is retrieved via the **`System.Environment.GetFolderPath`** method, passing the **`Personal`** value of the **`SpecialFolder`** enumeration.

For iOS, it is the user's documents folder, obtained via the **`NSFileManager.GetUrls`** method (the arguments are self-explanatory). For UWP, it's the local app folder, and is retrieved via the **`ApplicationData.Current.LocalFolder.Path`** object.

The last part of the code is common to all projects, and is about the connection string, which is of type **SQLiteConnectionString**. The constructor of this type takes the pathname, a **bool** parameter that specifies if the database file must be re-created (false in this case) if already existing, and a password that secures the database. This parameter is only available with SQLCipher. If you wanted to install the SQLite packages without security features, you would just write the following statement:

```
return new SQLiteConnection(path);
```

Remember the importance of the **assembly** attribute at the namespace level, with its **Dependency** argument that makes sure the runtime knows which implementation of the **IDatabaseConnection** interface it must use, depending on which platform the app is running on. Failure to add this attribute will result in a **NullReferenceException** when invoking the **DependencyService.Get** method.

Creating a data model

Let's imagine the app provides a piece of user interface to manage a list of your customers. First of all, you need a **Customer** class to model your data. Code Listing 12 demonstrates this in a way that satisfies the needs of SQLite.

Code Listing 12

```
using SQLite;
using System.ComponentModel;
namespace LocalDataAccess
{
    [Table("Customers")]
    public class Customer : INotifyPropertyChanged
    {
        private int _id;
        [PrimaryKey, AutoIncrement]
        public int Id
        {
            get
            {
                return _id;
            }
            set
            {
                this._id = value;
                OnPropertyChanged(nameof(Id));
            }
        }
        private string _companyName;
```

```
        [NotNull]
        public string CompanyName
        {
            get
            {
                return _companyName;
            }
            set
            {
                this._companyName = value;
                OnPropertyChanged(nameof(CompanyName));
            }
        }
        private string _physicalAddress;
        [MaxLength(50)]
        public string PhysicalAddress
        {
            get
            {
                return _physicalAddress;
            }
            set
            {
                this._physicalAddress = value;
                OnPropertyChanged(nameof(PhysicalAddress));
            }
        }
        private string _country;
        public string Country
        {
            get
            {
                return _country;
            }
            set
            {
                _country = value;
                OnPropertyChanged(nameof(Country));
            }
        }
        public event PropertyChangedEventHandler PropertyChanged;
        private void OnPropertyChanged(string propertyName)
        {
            this.PropertyChanged?.Invoke(this,
```

```
                new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

The first thing to note is the **Table** attribute at the class level, which maps the class to a table name. Because the SQLite libraries act as an ORM (object-relational mapping), the logic is that the library maps class properties to table columns in the database. So, at property levels, you will notice annotations such as **Autoincrement** and **PrimaryKey** for the **id** property (which maps the same-named column) and other annotations that apply constraints, such as **NotNull** and **MaxLength**.

The implementation of the **INotifyPropertyChanged** interface is useful to notify the user interface of changes on the data, but will be ignored by the SQLite library, so this interface implementation and the data annotation can coexist.

Now that you have a data model, it is a good idea to create a class that performs operations against data.

Reading, filtering, and writing data

The SQLite and SQLCipher libraries offer all the methods you need to read and write data, and you can use LINQ to filter your query results. Add a class called **DataService** to the .NET Standard project. The first lines of code declare a field of type **SQLiteConnection** that represents the connection to the database, and a static field of type **object** that is used to create a lock over the connection in order to avoid conflicts:

```
private SQLiteConnection database;

private static object collisionLock = new object();
```

In the constructor, you get the instance of the connection and create the table of **Customer** objects:

```
public DataService()
{
    database =
        DependencyService.Get<IDatabaseConnection>().
        DbConnection();
    database.CreateTable<Customer>();
}
```

The instance of the connection is retrieved via the **DependencyService.Get** method, which retrieves the proper implementation of the **IDatabaseConnection** interface, depending on the platform the app is running on. The **SQLiteConnection.CreateTable<T>** method creates a table based on the data model. The table is created only if it does not already exist.

Now it is time to write methods that perform operations against data.

Reading data

Let's imagine we want to read the full list of records from the **Customers** table, and we want to retrieve an individual record. The first task is accomplished via the following simple method:

```
public IEnumerable<Customer> GetCustomers()
{
    lock (collisionLock)
    {
        return database.Table<Customer>().AsEnumerable();
    }
}
```

The **Table<T>** method returns a **TableQuery<T>** object, which is a collection that implements the **IEnumerable** interface. However, an explicit conversion via **AsEnumerable** is done to ensure maximum compatibility outside of the SQLite environment. Notice how a **lock** block surrounds the **return** statement, making sure other callers cannot access the database at the same time.

Retrieving an individual record is still simple, and it can be accomplished with the following code:

```
public Customer GetCustomer(int id)
{
    lock (collisionLock)
    {
        return database.Table<Customer>().
          FirstOrDefault(customer => customer.Id == id);
    }
}
```

Here you get a first example of data filtering with LINQ. The **FirstOrDefault** method is invoked on the collection to retrieve the first record that matches the supplied ID. More complex LINQ queries can be performed over the result of **Table<T>** to filter the list based on a given criterion.

The following method demonstrates how to filter the list of customers based on their country of residence:

```
public IEnumerable<Customer> GetFilteredCustomers(string countryName)
{
    lock (collisionLock)
    {
        var query = from cust in database.Table<Customer>()
                    where cust.Country == countryName
                    select cust;
```

```
        return query.AsEnumerable();
    }
}
```

It is worth mentioning that SQLite also supports the SQL language directly. For example, you could rewrite the previous method as follows:

```
public IEnumerable<Customer> GetFilteredCustomers(string countryName)
{
    lock (collisionLock)
    {
        return database.Query<Customer>(
          $"SELECT * FROM Item WHERE Country =
'{countryName}'").AsEnumerable();
    }
}
```

Tip: Using the SQL language directly has the disadvantage of being more susceptible to injection attacks. You might want to use the managed approach where possible.

Writing data

SQLite supports both insert and update operations. You can easily detect if an object instance must be inserted or updated if its **`Id`** is equal to zero, like in the following method that saves an individual **`Customer`** instance:

```
public int SaveCustomer(Customer customerInstance)
{
    lock (collisionLock)
    {
        if (customerInstance.Id != 0)
        {
            database.Update(customerInstance);
            return customerInstance.Id;
        }
        else
        {
            database.Insert(customerInstance);
            return customerInstance.Id;
        }
    }
}
```

The code is very simple and does not need any further explanation. The **SQLiteConnection** class also exposes the **InsertAll** and **UpdateAll** methods, which allow for inserting and updating all the objects inside an **IEnumerable<T>** collection.

Though this is very convenient, a problem arises when you have a mixed list of new and updated objects. The solution is then represented with the following method, whose purpose is saving an entire **IEnumerable<Customer>** collection:

```
public void SaveAllCustomers(IEnumerable<Customer> customerCollection)
{
    lock (collisionLock)
    {
        foreach (var customerInstance in customerCollection)
        {
            if (customerInstance.Id != 0)
            {
                database.Update(customerInstance);
            }
            else
            {
                database.Insert(customerInstance);
            }
        }
    }
}
```

Basically, you apply the same logic of the previous method inside a **foreach** loop.

Deleting data

Deleting data is also an easy task, keeping in mind it's a one-way operation that cannot be reverted. The following code deletes an individual record from the table:

```
public int DeleteCustomer(Customer customerInstance)
{
    var id = customerInstance.Id;
    if (id != 0)
    {
        lock (collisionLock)
        {
            database.Delete<Customer>(id);
        }
    }

    return id;
}
```

The **Delete** method requires the ID of the record you want to delete. Another method called **DeleteAll** allows for deleting all the objects in a table. You can also completely delete an entire table via the **DropTable** method. You will need to invoke the **CreateTable** method shown at the beginning to create the table again.

If your app handles very sensitive data, I would recommend you add a Boolean column to the table called **IsDeleted** or something similar, which you set as **true** if you want to mark the record as deleted, and then you update the record instead of physically deleting it. This allows you to keep a history of the data, even when some records will not be used anymore.

Note: There is more in the SQLite database engine. For example, it supports transactions, indexes, and backups, and the `SQLiteConnection` class also exposes the `TableChanged` event that is raised when a table is changed. For further details, have a look at the GitHub repository for the library.

Implementing Model-View-ViewModel logic

A good approach when working with databases is implementing separation between data models, data access, business logic, and user interface. The Model-View-ViewModel pattern is perfect for accomplishing this. In the previous section you wrote a data model and a class for data access. Now, it's time to write a view model.

In summary, the view model needs to expose the following objects that will be data-bound to the user interface:

- A collection of **Customer** objects.
- An individual **Customer** object (useful to represent the selection in a list).
- **Command** objects that invoke methods in the **DataAccess** class and execute operations against data.

Code Listing 13 shows the full code for the **ViewModel**, and comments on the relevant points will follow.

Code Listing 13

```
using System.Collections.ObjectModel;
using System.ComponentModel;
using System.Linq;
using Xamarin.Forms;

namespace LocalDataAccess
{
    public class CustomerViewModel : INotifyPropertyChanged
    {
        private ObservableCollection<Customer> _customers;
        public ObservableCollection<Customer> Customers
```

```
        {
            get
            {
                return _customers;
            }
            set
            {
                _customers = value;
                OnPropertyChanged(nameof(Customers));
            }
        }

        private Customer _selectedCustomer;
        public Customer SelectedCustomer
        {
            get
            {
                return _selectedCustomer;
            }
            set
            {
                _selectedCustomer = value;
                OnPropertyChanged(nameof(SelectedCustomer));
            }
        }


        private DataService dataAccess;

        public CustomerViewModel()
        {
            dataAccess = new DataService();

            var customers = dataAccess.GetCustomers();

            Customers =
              new ObservableCollection<Customer>(customers);

            if (!Customers.Any())
                AddCustomer();
        }

        private void AddCustomer()
        {
```

```
            var newCustomer = new Customer
            {
                CompanyName = "Company name...",
                PhysicalAddress = "Address...",
                Country = "Country..."
            };

            Customers.Add(newCustomer);
            dataAccess.SaveCustomer(newCustomer);
        }

        public Command SaveAllCommand
        {
            get
            {
                return new Command(() =>
                dataAccess.SaveAllCustomers(Customers));
            }
        }

        public Command AddNewCommand
        {
            get
            {
                return new Command(() =>
                AddCustomer());
            }
        }

        public Command DeleteCommand
        {
            get
            {
                return new Command(() =>
                {
                    if (SelectedCustomer != null)
                    {
                        Customers.Remove(SelectedCustomer);
                        dataAccess.DeleteCustomer(SelectedCustomer);
                    }
                });
            }
        }
```

```
        public event PropertyChangedEventHandler PropertyChanged;
        private void OnPropertyChanged(string propertyName)
        {
            PropertyChanged?.Invoke(this,
              new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

The relevant points in Code Listing 13 are:

- The implementation of the **INotifyPropertyChanged** interface allows for notifying the bound views in the user interface so that they can automatically reflect changes.
- The **AddCustomer** method adds a new **Customer** object to the **Customers** collection and inserts the new object into the database at the same time. This demonstrates how updating an existing **Customer** works.
- **Command** objects, which will be bound to buttons in the user interface, call methods in the **DataService** class, according to the MVVM principles.

The final step is displaying a user interface. This is going to be very simple, but with all the pieces required to work with the database.

Creating the user interface with data binding

The user interface for the sample app is defined inside the **MainPage.xaml** file. A good choice to display the list of customers is the **CollectionView** control with a simple **DataTemplate** where a **Label** displays the value of the **Customer.Id** property, so that it cannot be edited, and where **Entry** views allows for editing the value for the other properties. Code Listing 14 shows the full XAML code.

Code Listing 24

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="LocalDataAccess.MainPage" Padding="0,20,0,0">

    <Grid Margin="10">
        <Grid.RowDefinitions>
            <RowDefinition />
            <RowDefinition Height="40"/>
        </Grid.RowDefinitions>

        <CollectionView x:Name="CustomersView"
```

```
                    SelectedItem="{Binding SelectedCustomer,
                                  Mode=TwoWay}"
                    ItemsSource="{Binding Customers}"
                    SelectionMode="Single">
            <CollectionView.ItemTemplate>
                <DataTemplate>
                    <StackLayout Orientation="Vertical"
                                 Margin="0,10,0,0">
                        <Label Text="{Binding Id}"
                               FontSize="Medium"/>
                        <Entry Text="{Binding CompanyName}"
                               FontSize="Medium" />
                        <Entry Text="{Binding PhysicalAddress}"
                               FontSize="Medium"/>
                        <Entry Text="{Binding Country}"
                               FontSize="Medium"/>
                    </StackLayout>
                </DataTemplate>
            </CollectionView.ItemTemplate>
        </CollectionView>

        <StackLayout Orientation="Horizontal"
                     Spacing="5" Grid.Row="1">
            <Button Text="Add"
                    Command="{Binding AddNewCommand}"
                    WidthRequest="90"/>
            <Button Text="Remove"
                    Command="{Binding DeleteCommand}"
                    WidthRequest="90"/>
            <Button Text="Save all"
                    Command="{Binding SaveAllCommand}"
                    WidthRequest="90"/>
        </StackLayout>
    </Grid>
</ContentPage>
```

About the **`CollectionView`**: the **`SelectedItem`** property is bound to the **`SelectedCustomer`** property of the view model; **`ItemsSource`** is bound to the **`Customers`** property, representing the full list of records coming from the database; and **`SelectionMode`** is **`Single`** to allow for one object selection.

The **Label** and the **Entry** views in the data template are bound to related properties in each instance of the **Customer** object. Finally, there are three buttons, each bound to a command in the **ViewModel**, that allow for performing operations against the data with an approach based on layer separation. The final step is instantiating the **ViewModel** and using this as the page's data context. Code Listing 15 shows the C# code that you will add to the code-behind file for the page.

Code Listing 15

```
public partial class MainPage : ContentPage
{
    private CustomerViewModel ViewModel { get; set; }
    public MainPage()
    {
        InitializeComponent();
        ViewModel = new CustomerViewModel();
        BindingContext = ViewModel;
    }
}
```

Now you can run the sample project and see the result of the work you've done so far.

Running the example

When you run the app for the first time, there will be no records in the database, so the code adds a new **Customer** instance, which is added to the **Customers** collection and automatically saved to the **Customers** table in the SQLite database. You can edit the new instance and add new instances, as demonstrated in Figure 11.

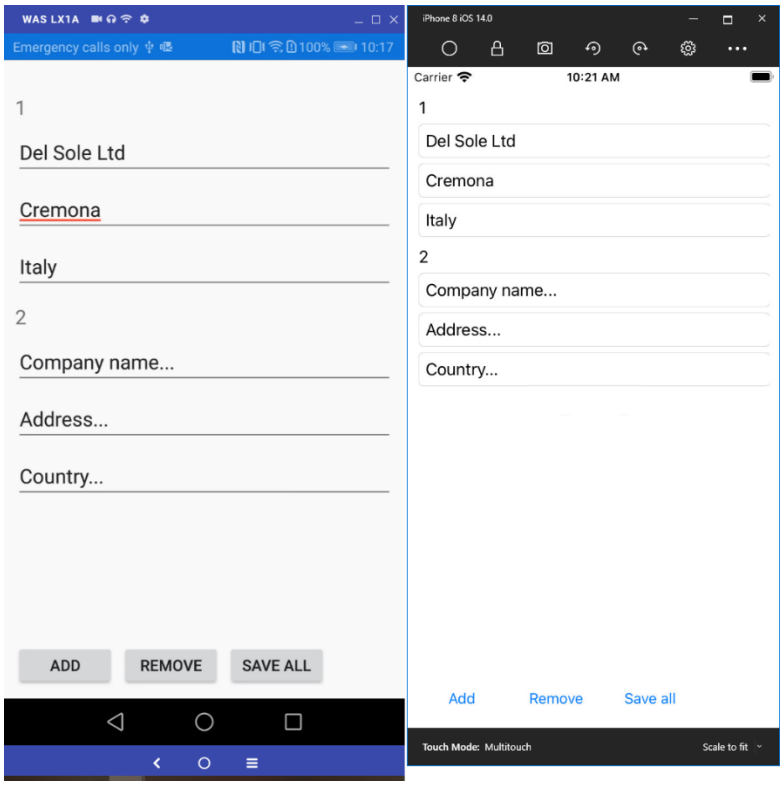

Figure 11: The sample app running and showing data from the database

Based on the code, when a new instance is added, it is automatically inserted to the database, but if you make changes to the values, you will need to save the data manually. This will perform and update the operation.

Chapter summary

SQLite is an open-source, serverless database that mobile apps written with Xamarin.Forms can leverage to locally store complex data within tables. In this chapter, you have seen how to secure the database with SQLCipher, create tables and read/write data, and display the tables in the user interface with data binding. The Model-View-ViewModel pattern has been used for true separation between data, data access, business logic, and user interface, which is the approach you should also use in the real world for better code maintenance.

There are certainly points for improvement which depend on your needs, but you have seen how simple it is to implement local data access securely. Managing data is clearly important, and data does not only mean databases. There are other sources for data in mobile apps, such as web APIs and JSON, which are discussed in the next chapter.

Chapter 5 Working with Web API and JSON

It's very common for mobile apps to exchange data with remote services, either on-premises or on the cloud. Xamarin.Forms supports a number of communication protocols and data exchange formats, but the most common scenario is to work with data in a platform-independent approach.

For instance, a web service might serve different types of applications (desktop, mobile, web clients), and in the ideal world, it should not know in advance what the caller application is and with which development platform it has been written, so data exchange should happen with standardized, cross-platform, and cross-device formats. From the point of view of development with Microsoft technologies, this can be done by developing web API solutions that expose data as JSON or XML, and that can be consumed by any kind of application, including mobile apps created with Xamarin.Forms.

This chapter explains how to consume a web API, and how to exchange data with a web API service using JSON as the standard format.

Assumptions for this chapter

The companion solution for this chapter is more complex, and therefore, more thoroughly explained than the other chapters in the book. In fact, it includes a complete ASP.NET Core web API project and a Xamarin.Forms project based on the Model-View-ViewModel pattern. For this reason, I assume you open the sample solution for Chapter 5 in Visual Studio 2019, and that you will follow the reading while looking at the source code in the IDE. I will only highlight and list the code snippets that are relevant to the topic of this chapter. Based on this consideration, some knowledge of the Model-View-ViewModel (MVVM) pattern is also required. You can read an introduction in my book *Xamarin.Forms Succinctly*. If you wish to fully try the solution and you want to publish the web API project to Azure, I recommend you do the following:

- Create an Azure account. A free account is available for 12 months (please read the terms of use carefully).
- Read the documentation about creating a SQL database on Azure.
- Read the documentation about publishing an ASP.NET Core project to Azure.

Once you have set this up, you are ready to walk through the sample solution and this chapter.

Understanding the sample solution

The companion example for this chapter is made of three main blocks:

- A SQL database called **Books** hosted on Azure. This database contains a simplified list of book information.

- A web API project, deployed to Azure, that exposes calls to read, write, update, and delete books from the database.
- A Xamarin.Forms project that displays the list of books and allows for adding new books to the database.

These will be described in more detail in the next sections.

Creating the database and publishing the web API project to Azure has been the perfect choice for the current example, since I do not own a physical web server and, at the same time, I could work with a real remote environment rather than showing local debugging. Let's understand a bit more of the individual blocks of the architecture.

Understanding the database

The companion solution for this chapter works with a database called **Books**, which contains a simplified list of books. For each book, the database contains the title, the author, and the ISBN code. You can easily re-create the database on Azure using SQL Server Management Studio or Visual Studio 2019, connecting to the Azure database engine, and running the script shown in Code Listing 16.

Notice how a Basic profile is selected, which allows for creating a free database, limited to 1 GB of storage.

Code Listing 16

```
CREATE DATABASE [Books]
  (EDITION = 'Basic',
   SERVICE_OBJECTIVE = 'Basic',
   MAXSIZE = 1 GB)
   WITH CATALOG_COLLATION = SQL_Latin1_General_CP1_CI_AS;
GO

ALTER DATABASE [Books] SET COMPATIBILITY_LEVEL = 100
GO

ALTER DATABASE [Books] SET ANSI_NULL_DEFAULT OFF
GO

ALTER DATABASE [Books] SET ANSI_NULLS OFF
GO

ALTER DATABASE [Books] SET ANSI_PADDING OFF
GO

ALTER DATABASE [Books] SET ANSI_WARNINGS OFF
GO
```

```
ALTER DATABASE [Books] SET ARITHABORT OFF
GO

ALTER DATABASE [Books] SET AUTO_SHRINK OFF
GO

ALTER DATABASE [Books] SET AUTO_UPDATE_STATISTICS ON
GO

ALTER DATABASE [Books] SET CURSOR_CLOSE_ON_COMMIT OFF
GO

ALTER DATABASE [Books] SET CONCAT_NULL_YIELDS_NULL OFF
GO

ALTER DATABASE [Books] SET NUMERIC_ROUNDABORT OFF
GO

ALTER DATABASE [Books] SET QUOTED_IDENTIFIER OFF
GO

ALTER DATABASE [Books] SET RECURSIVE_TRIGGERS OFF
GO

ALTER DATABASE [Books] SET AUTO_UPDATE_STATISTICS_ASYNC OFF
GO

ALTER DATABASE [Books] SET DATE_CORRELATION_OPTIMIZATION OFF
GO

ALTER DATABASE [Books] SET ALLOW_SNAPSHOT_ISOLATION ON
GO

ALTER DATABASE [Books] SET PARAMETERIZATION SIMPLE
GO

ALTER DATABASE [Books] SET READ_COMMITTED_SNAPSHOT ON
GO

ALTER DATABASE [Books] SET  MULTI_USER
GO

ALTER DATABASE [Books] SET QUERY_STORE = ON
GO
```

```
ALTER DATABASE [Books] SET QUERY_STORE
  (OPERATION_MODE = READ_WRITE, CLEANUP_POLICY =
  (STALE_QUERY_THRESHOLD_DAYS = 7),
  DATA_FLUSH_INTERVAL_SECONDS = 900,
  INTERVAL_LENGTH_MINUTES = 60,
  MAX_STORAGE_SIZE_MB = 10,
  QUERY_CAPTURE_MODE = AUTO,
  SIZE_BASED_CLEANUP_MODE = AUTO)
GO

ALTER DATABASE [Books] SET  READ_WRITE
GO
```

You can enter some sample data manually, or you could run the script shown in Code Listing 17 to add information for two books to the table.

Code Listing 17

```
USE Books
GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[Books](
  [Id] [int] IDENTITY(1,1) NOT NULL,
  [Title] [nvarchar](50) NOT NULL,
  [Author] [nvarchar](50) NOT NULL,
  [ISBN] [nvarchar](50) NULL,
 CONSTRAINT [PK_Books] PRIMARY KEY CLUSTERED
(
  [Id] ASC
)WITH (PAD_INDEX = OFF,
STATISTICS_NORECOMPUTE = OFF,
IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS = ON,
ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY]
GO
SET IDENTITY_INSERT [dbo].[Books] ON
GO
INSERT [dbo].[Books] ([Id], [Title], [Author], [ISBN])
  VALUES (1, N'Xamarin.Forms Succinctly',
```

```
  N'Alessandro Del Sole', N'978-1-64200-175-4')
GO
INSERT [dbo].[Books] ([Id], [Title], [Author], [ISBN])
  VALUES (2, N'Visual Studio 2019 for Mac Succinctly',
  N'Alessandro Del Sole', N'978-1-64200-177-8')
GO
SET IDENTITY_INSERT [dbo].[Books] OFF
GO
```

You can improve the table by adding the publication date or a Boolean column to mark a record as logically deleted, without actually removing the record from the database, but this is left to you as an exercise.

Understanding the web API project

The web API project is responsible for working against the database using Entity Framework, and it returns JSON responses, depending on the request. For example, it will return a JSON array when a **`GET`** call is made to retrieve the list of books.

The project has been created following this very detailed article published by Syncfusion, and I recommend you to do the same if you want to re-create the sample project on your own. One important note is about securing the API, which I will also recall in the next paragraphs, when appropriate. The article explains how to implement authorization and authentication based on JWT tokens, where JWT stands for JSON Web Token, and is a way to include an authorization code inside the API calls.

Hints about authorization and authentication

Authorization and authentication implementations can be very different between companies' policies, customer requirements, and app architecture. For example, your requirements might be implementing custom authentication with username and password, or you might need an Active Directory authentication system if your application works within a domain, or you might work with bearer tokens. For this reason, the sample web API service does not require any authentication. In this way, you will be able to implement your own system based on your requirements. I will point out, where appropriate, how to add tokens and user credentials inside the API calls.

The companion project also implements the **`TokenController`** class discussed in the Syncfusion article for your convenience, so it will be very easy for you to implement JWT tokens discussed in the article itself. The next paragraphs highlight the most relevant parts of the project.

Data model implementation

The data model consists of a **`Book`** class that maps the **`Books`** table in the database, and looks like the following:

```
public partial class Book
{
    public int Id { get; set; }
    public string Title { get; set; }
    public string Author { get; set; }
    public string Isbn { get; set; }
}
```

A specialized instance of the **DbContext** class is then needed to connect the database to the data model, and is represented in Code Listing 18.

Code Listing 18

```
    public partial class BooksContext : DbContext
    {

        public BooksContext(DbContextOptions<BooksContext> options)
            : base(options)
        {
        }

        public virtual DbSet<Book> Books { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {

            modelBuilder.Entity<Book>(entity =>
            {
                entity.Property(e => e.Author)
                    .IsRequired()
                    .HasMaxLength(50);

                entity.Property(e => e.Isbn)
                    .HasColumnName("ISBN")
                    .HasMaxLength(50);

                entity.Property(e => e.Title)
                    .IsRequired()
                    .HasMaxLength(50);
            });

            OnModelCreatingPartial(modelBuilder);
        }

        partial void OnModelCreatingPartial(ModelBuilder modelBuilder);
```

```
    }
```

As you can see, the **OnModelCreating** method performs the actual binding between the table and the data model, including some validations.

Tip: You will need to replace the sample connection string in the** appsettings.json **file with the proper one.

API call implementation and testing

The API calls are implemented inside a controller class called **BooksController**. The first thing to do in the controller is implement a constructor that will receive an instance of the context class via dependency injection:

```
private readonly BooksContext _context;
public BooksController(BooksContext context)
{
    _context = context;
}
```

The next piece of code is related to retrieving the list of books from the database. There are two **GetBooks** methods: one to get the full list, and one to get an individual book based on its ID:

```
[HttpGet]
public async
    Task<ActionResult<IEnumerable<Book>>> GetBooks()
{
    return await _context.Books.ToListAsync();
}

[HttpGet("{id}")]
public async Task<ActionResult<Book>> GetBooks(int id)
{
    var books = await _context.Books.FindAsync(id);

    if (books == null)
    {
        return NotFound();
    }
    return books;
}
```

The code is executed asynchronously and data is obtained via methods from the **DbContext** class: **ToListAsync** for the full list, and **FindAsync** for the individual object. Adding a new **Book** object to the database is done via the **POST** verb and the **PostBook** method, as follows:

```
[HttpPost]
public async Task<ActionResult<Book>> PostBook(Book book)
{
    _context.Books.Add(book);
    await _context.SaveChangesAsync();

    return CreatedAtAction("GetBooks",
        new { id = book.Id }, book);
}
```

The **Book** object that will be added to the database is received by the API call in the form of a JSON object, and you will see how to package this in Xamarin.Forms shortly.

Notice how the API call returns a **CreatedAtAction** object, which matches the 201 HTTP status code (**Created**) with the ID of the new record.

You can update existing objects by invoking the **HTTP PUT** verb. In this example, updating a **Book** object can be performed via the **PutBook** method, which leverages the **PUT** verb behind the scenes, and works as follows:

```
public async Task<IActionResult> PutBook(int id, Book book)
{
    if (id != book.Id)
    {
        return BadRequest();
    }
    _context.Entry(book).State = EntityState.Modified;
    try
    {
        await _context.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        if (!BooksExists(id))
        {
            return NotFound();
        }
        else
        {
            throw;
        }
    }
    return NoContent();
}
```

```
private bool BooksExists(int id)
{
    return _context.Books.Any(e => e.Id == id);
}
```

As with adding an object, the **Book** entity instance is still received from the caller clients. One difference is that the entity state changes to **EntityState.Modified** and the code checks for concurrency exceptions. This method returns a **NoContent** response, which matches the 204 HTTP status code.

The very last method is called **DeleteBook**, which receives only the ID for the **Book** object you want to delete:

```
[HttpDelete("{id}")]
public async Task<ActionResult<Book>> DeleteBook(int id)
{
    var books = await _context.Books.FindAsync(id);
    if (books == null)
    {
        return NotFound();
    }
    _context.Books.Remove(books);
    await _context.SaveChangesAsync();
    return books;
}
```

The code is very easy, and this is a standard implementation. However, if you want to store the full history of the data, I recommend you add a **BIT** column called **IsDeleted** to the database, add a same-named property of type **bool** to the **Book** class, and replace the **DbContext.Remove** invocation as follows:

```
Book.IsDeleted = true;
```

In this way, the code will not physically remove the record from the database; it will mark it as logically deleted. If you go for this option, you might also want to change the **GetBooks** method body as follows:

```
return await _context.Books.Where(b=>b.IsDeleted == false).ToListAsync();
```

Once you understand the most relevant pieces of code, it's time to test the API calls. You can do this by starting the web API project for debugging on the local machine, or you can deploy it to Azure (or to an on-premises server) for the most realistic experience. Whichever option you choose, you need a tool to make and test API calls.

One of the most popular and free tools is Postman. With Postman you can make API calls with any of the HTTP verbs, and you can include all the information needed for the request through a simple user interface. Whichever option you go with, take note of the web API address.

In Postman, open a new request to make a **GET** call, and paste the URL of your web API service and the **/api/books** suffix. When you're ready, simply click **Send**. After a few seconds, you will see the list of books in the results window, in the form of a JSON array, as shown in Figure 12.

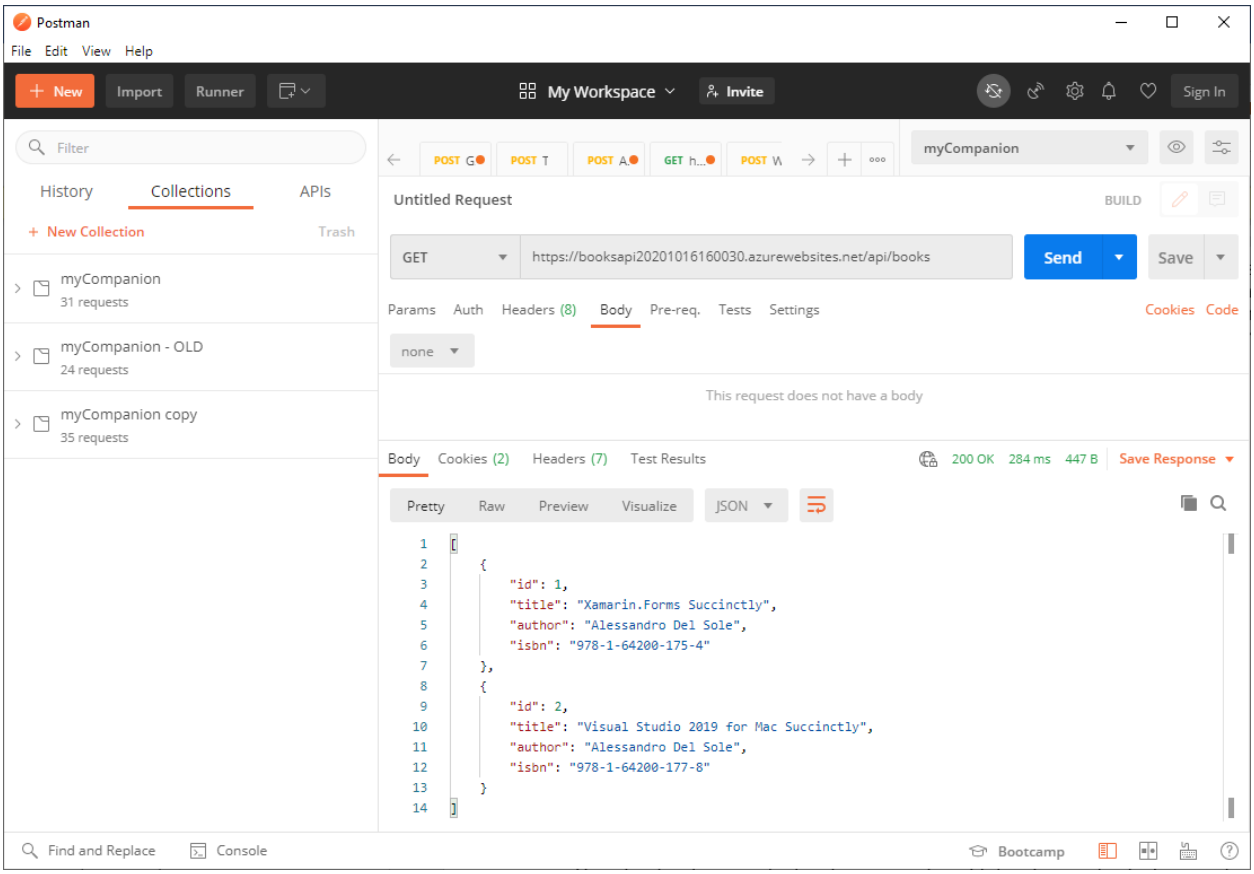

Figure 22: Getting the list of books with Postman

Adding a new book can be done by changing the HTTP verb to **POST** and specifying the data in the **Body** tab. If you select the **raw** view, you can write the JSON directly, as shown in Figure 13.

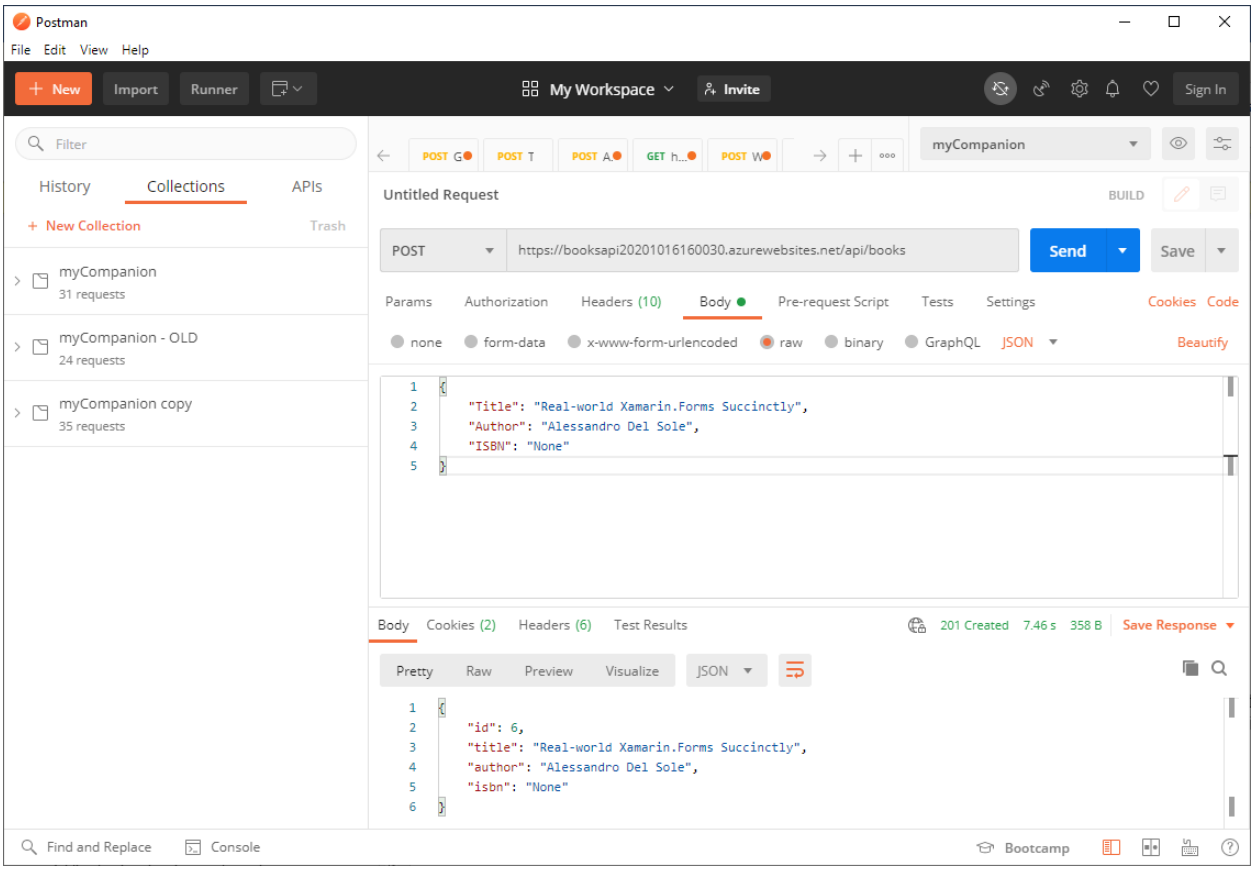

Figure 33: Adding a new object with a POST request

Another required step is opening the **Headers** tab and adding a new header tag of type **`Content-Type`** in the **KEY** column, with a value of **`application/json`**. When you click **Send**, the **`PostBook`** API call will be invoked, and the new object will be written into the database. The response contains the full properties for the newly added object, and the most important one is the new record ID.

> ***Tip: If the web API implements a custom authentication mechanism with username and password, these will be included in the JSON sent to the API. If the implementation relies on a token, this will instead be included in the Authorization tab, where you will find a dropdown list of the most popular token types.***

If you want to double-check, make the **`GET`** call again to see the list of books, and you will see that the new one is available. Updating a book will work very similarly: you will need to write the full list of **Book** properties and change the HTTP verb from **`POST`** to **`PUT`**. The URL of the API call will also need to include the book ID in the following form: **/api/books/1**, where 1 is the record ID.

Deleting a book instance is also easy. You must change the HTTP verb to **`DELETE`** and pass the record ID into the URL like you did for the **`PUT`** call. Then, in the **Body** tab, you select **none**, as the request has no body. Once you have ensured the API calls work outside of any application and development environment, you can understand how to consume the API in Xamarin.Forms.

Understanding the Xamarin.Forms project

The Xamarin.Forms project provides the following points of interest:

- The user interface displays a list of books with a **`CollectionView`** and allows for loading, adding, and deleting books with specific buttons. All these views are bound to properties and commands in a view model called **`BookViewModel`**.
- The view model calls methods to work with data in a service class called **`WebApiService`** and to send broadcast messages to the user interface, which will react by informing the user about the result of data operations.
- Data with the API is exchanged by serializing and deserializing JSON objects. This is done via the Newtonsoft.Json library, the *de facto* standard library for .NET and C#. Actually, starting with .NET Standard 2.0, Microsoft introduced the System.Text.Json API, which provides built-in objects to work with JSON data, without the need for third-party libraries. You are totally free to use this API instead of Newtonsoft.Json, but in the real world, you will still find millions of projects that use the latter. If you are starting a new project, you can still make a choice, but if you are maintaining an existing project, there's a higher probability that you will work with Newtonsoft.Json.

Tip: Due to the popularity of Newtonsoft.Json, Microsoft has created some documentation to help you migrate to System.Text.Json.

The final result of the work is shown in Figure 14, where you can see a list of books, and in Figure 15, where you can see the insertion of a new book. You can keep both figures as a reference since I will not list the XAML code (which you can instead follow through the companion source code).

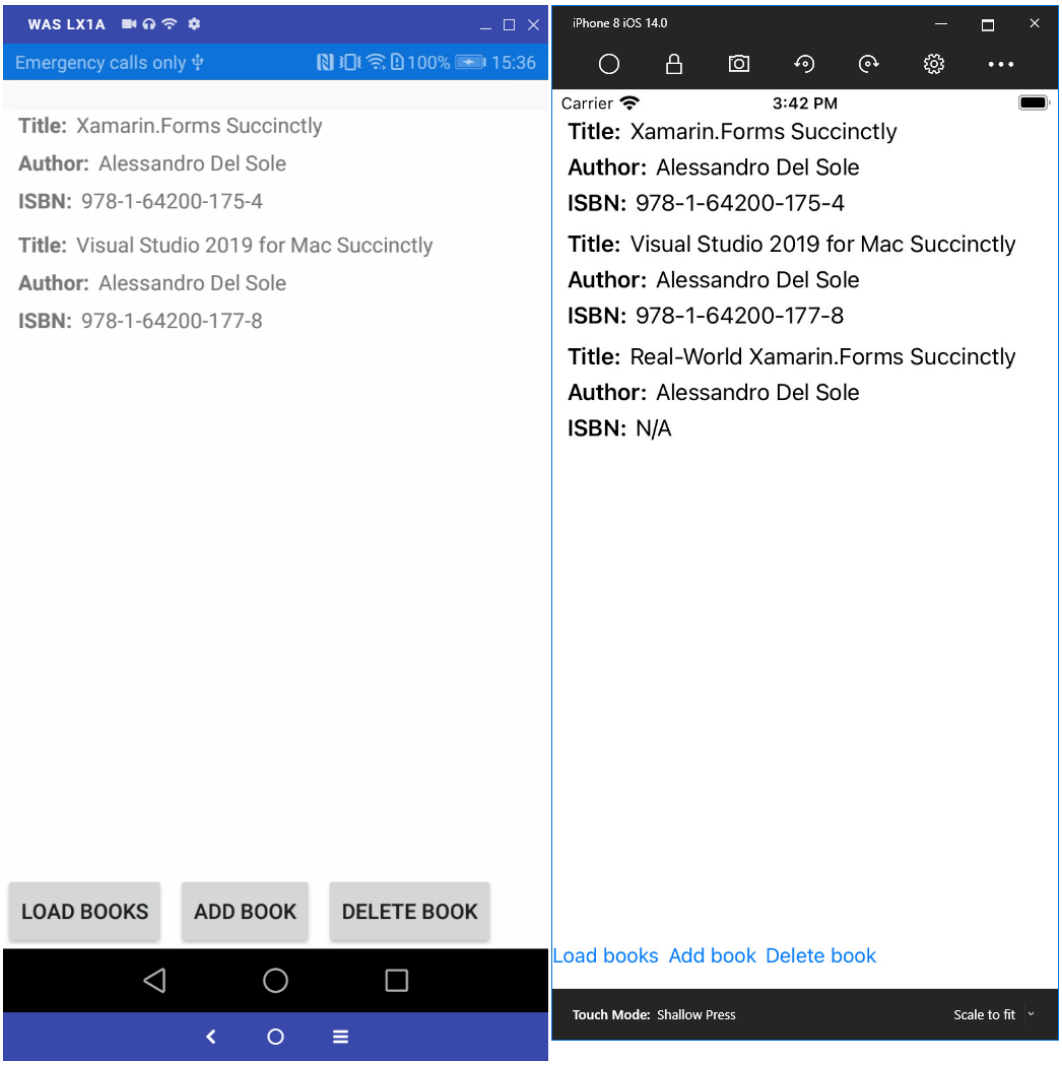

Figure 14: The sample app displays a list of books

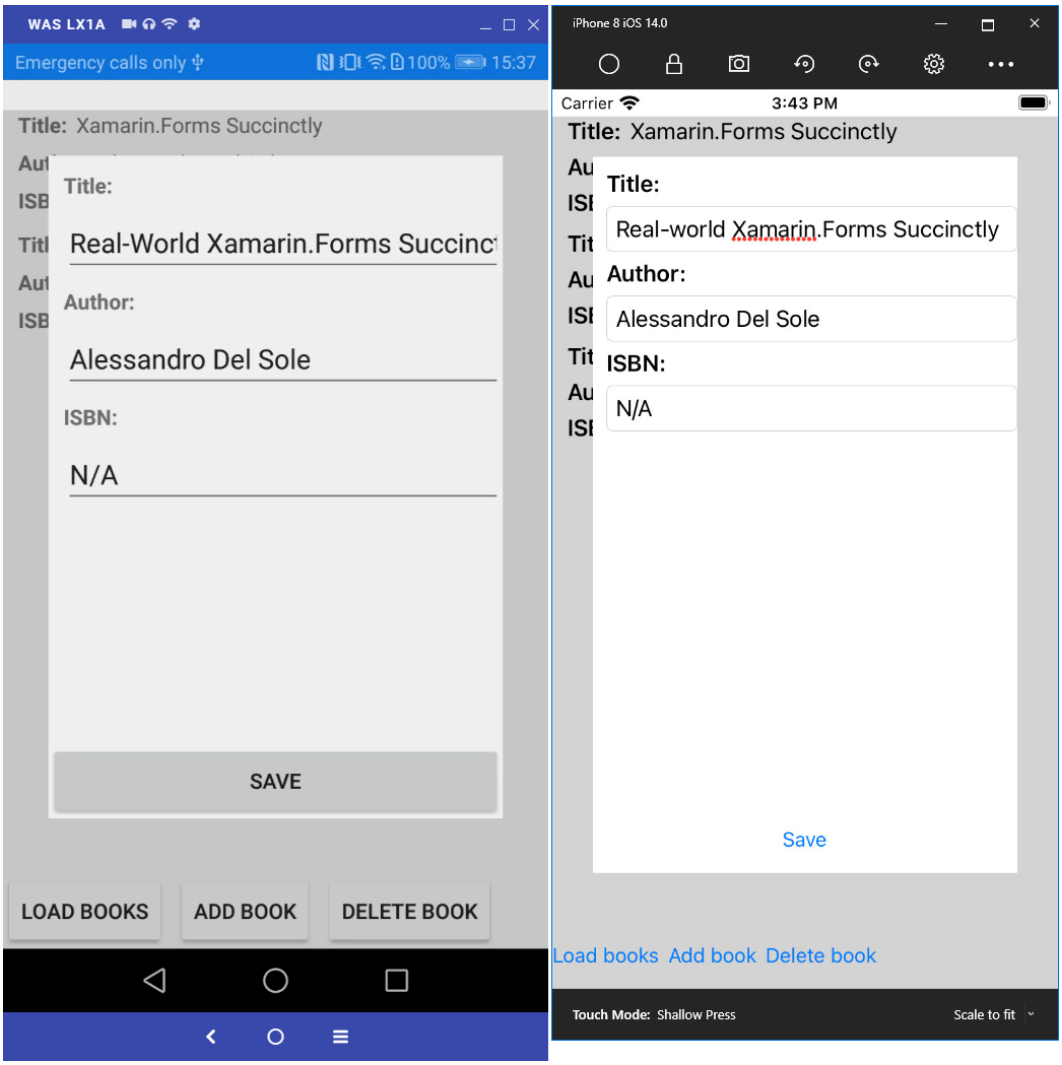

Figure 45: The sample app allows adding and saving a new book

Calling web APIs from Xamarin.Forms

Calling web APIs from Xamarin.Forms is not difficult, and is accomplished via the **`HttpClient`** class and its methods. It is good practice to group methods that work with a web API into a separate class. In the sample project, it's called **`WebApiService`**, and it exposes methods to read, write, and delete data. Another good practice is to write reusable methods, avoiding code that targets only a specific data type.

A method called **`GetDataAsync`** sends a **`GET`** request to the specified endpoint, and is implemented as follows:

```
public static async Task<HttpResponseMessage> GetDataAsync(string url,
       string id = null)
{
```

```
    HttpResponseMessage response;
    try
    {
        using (var client = new HttpClient())
        {
            if (id != null) url = $"{url}/{id}";
            response = await client.GetAsync(url);
        }
    }
    catch (HttpRequestException)
    {
        response = new
HttpResponseMessage(HttpStatusCode.ServiceUnavailable);
    }
    catch
    {
        response = new
HttpResponseMessage(HttpStatusCode.InternalServerError);
    }
    return response;
}
```

The reason for the **id** optional parameter is that, with a single method, you can obtain either the full list of objects from the specified endpoint (in our example a list of books), or an individual object from the specific API overload. The actual API call is done by the **HttpClient.GetAsync** method, and the resulting response is returned as it is to the caller (the **BooksViewModel** class in our case), which will be responsible for data processing. Multiple **catch** statements return different HTTP errors. This is to demonstrate how you can control in your logic the different error types that you might get from a web API service.

As I mentioned previously, the companion source code provides all the necessary code files to implement JWT tokens to secure the API calls, but this is left to you as an exercise, and API calls can be currently invoked with the so-called anonymous authentication. This is because it is not possible to demonstrate all the possible authentication solutions and, at the same time, to offer a working example. However, a few hints can be provided. If your web API requires a token, you can add the following line after instantiating the **HttpClient** class:

```
client.DefaultRequestHeaders.Authorization =

       new AuthenticationHeaderValue("Bearer", "Your Oauth token");
```

The constructor of the **AuthenticationHeaderValue** receives the **Bearer** literal and the authorization token, and is assigned to the **Authorization** property of the request's headers. If your web API services rely on Windows domain authentication, you can create an instance of the **HttpClient** class, specifying that it needs to use the default Windows user credentials:

```
var client =

    new HttpClient(new HttpClientHandler() { UseDefaultCredentials = true });
```

The following method invokes a web API to delete data, and is very similar to the previous method:

```
public static async Task<HttpResponseMessage>
       DeleteDataAsync(string url, int id)
{

    HttpResponseMessage response;
    try
    {
        using (var client = new HttpClient())
        {
            string fullUri = $"{url}/{id}";
            response = await client.DeleteAsync(fullUri);
        }
    }
    catch (HttpRequestException)
    {
        response = new
HttpResponseMessage(HttpStatusCode.ServiceUnavailable);
    }
    catch
    {
        response = new
HttpResponseMessage(HttpStatusCode.InternalServerError);
    }
    return response;
}
```

The only difference is that this method invokes **HttpClient.DeleteAsync** to send a **DELETE** request. The next method, called **WriteDataAsync**, is very interesting. Its purpose is to send a **POST** request to a web API, and its implementation takes advantage of .NET generics, so it can be reused:

```
 public static async Task<HttpResponseMessage>

     WriteDataAsync<T>(T data, string url, string id = null)

 {

     HttpResponseMessage response;

     try

     {

         using (var client = new HttpClient())

         {

             client.DefaultRequestHeaders.Accept.Clear();

             client.DefaultRequestHeaders.Accept.Add(
```

```
                new MediaTypeWithQualityHeaderValue("application/json"));

            string json = JsonConvert.SerializeObject(data);
            var content =
                new StringContent(json, Encoding.UTF8, "application/json");
            if (id != null) url = $"{url}/{id}";
            response = await client.PostAsync(url, content);
        }
    }
    catch (HttpRequestException)
    {
        response = new
HttpResponseMessage(HttpStatusCode.ServiceUnavailable);
    }
    catch
    {
        response = new
HttpResponseMessage(HttpStatusCode.InternalServerError);
    }

    return response;
 }
```

For a better understanding of the code, you can focus on two steps you did manually in Postman. The first step was adding the application/JSON content-type information to the request's headers. In the code, this is done by adding to the **`DefaultRequestHeaders`** collection a new object of type **`MediaTypeWithQualityHeaderValue`**, which receives the specified MIME type as an argument.

Serializing objects

The second step you made in Postman was specifying the JSON markup for the new book you wanted to add to the database via API. In the code, this takes two lines. The first line serializes the object you want to write into JSON via the **`JsonConvert.SerializeObject`** method. The second line creates an instance of the **`StringContent`** class based on the serialized data, the encoding, and the MIME type. In fact, it's not enough for the API to receive the serialized object; it is also necessary for the API to know which text encoding and data format is used. Finally, the API call is made by invoking **`HttpClient.PostAsync`**.

Hints about editing existing data

You can update existing data by sending an HTTP **`PUT`** request to the API via the **`HttpClient.PutAsync`** method. You have different options:

- You can create a method that works like **`WriteDataAsync`** but invokes **`PutAsync`** instead.
- You can keep everything in one method, adding a bool parameter to **`WriteDataAsync`**, such as **`IsUpdate`**, and invoking **`PutAsync`** if **`IsUpdate`** is **`true`**, or **`PostAsync`** if it is **`false`**.
- You could handle the update scenario inside the **`Post`** method of the API controller (in our case it is **`PostBook`**). The client could still send a **`POST`** request, and then the API would decide how to dispatch the request, which would need the **`WriteDataAsync`** method to send a bool additional parameter.

I would personally choose the second option, which allows for having less code, and does not create more risk of confusion in the API code.

Deserializing JSON responses

All the methods in the **`WebApiService`** class return an object of type **`HttpResponseMessage`**. Among the others, this object exposes a **`StatusCode`** property, of type **`HttpStatusCode`**, which contains the code for the result of the API call, and the **`Content`** property, of type **`HttpContent`**, which contains the actual response from the API.

Each view model will check the status code and the HTTP response. In the sample app, one view model called **`BooksViewModel`** is available. Here, a method called **`LoadBooksAsync`** is invoked by a command called **`LoadBooksCommand`**, and gets the lists of books from the web API as follows:

```
private async Task LoadBooksAsync()
{
    // Replace with the URL of your Web API…
    string url =
           "https://yourwebapi.azurewebsites.net/api/books";
    var result = await WebApiService.GetDataAsync(url);
    switch (result.StatusCode)
```

```
    {
        case HttpStatusCode.OK:
            string resultString =
                await result.Content.ReadAsStringAsync();
            var deserialized =
                JsonConvert.DeserializeObject<List<Book>>(resultString);
            Books = new ObservableCollection<Book>(deserialized);
            return;
        default:
            MessagingCenter.
                Send(this, "ServerError",
                $"{result.StatusCode} {result.ReasonPhrase}");
            return;
    }
}
```

As you can see, the view model handles the status code returned by the **GetDataAsync** method and, in case of errors, sends a broadcast message that the user interface will catch to display a warning to the user. (See the **MainPage.xaml.cs** file in the companion source code for the implementation). If it's successful, the response of the API call is first converted into a plain string.

Then, because the code expects a list of books as the response, the **JsonConvert.DeserializeObject** converts the obtained string into a specialized **List<Book>** object. The next line of code creates an **ObservableCollection<Book>** instance, because this type of collection is used to data-bind collections to the user interface. The behavior is similar for the **AddBookAsync** method, which invokes the **WebApiService.PostAsync** method to write a new book into the database:

```
private async Task AddBookAsync()
{
    if (NewBook != null)
    {
        string url = "https://yourwebapi.azurewebsites.net/api/books";
        var result = await WebApiService.WriteDataAsync(NewBook, url);

        switch (result.StatusCode)
        {
            case HttpStatusCode.OK:
            case HttpStatusCode.Created:
                string resultString = await
                       result.Content.ReadAsStringAsync();
                Book deserializedBook =
                    JsonConvert.DeserializeObject<Book>(resultString);
                Books.Add(deserializedBook);
                MessagingCenter.Send(this, "BookSaved");
                return;
```

```
            default:
                MessagingCenter.Send(this, "ServerError",
                    $"{result.StatusCode} {result.ReasonPhrase}");
                return;
        }
    }
}
```

This behavior is similar to the **LoadBooksAsync** method. Here the code also handles the **Created** HTTP status code. In this case, **JsonConvert.DeserializeObject** returns the individual **Book** instance that has been added to the database, and this will be added to the **Books** collection of the view model so that the user interface can update accordingly.

The **DeleteBookAsync** in the view model is also similar:

```
private async Task DeleteBookAsync()
{
    string url =
        "https://yourwebapi.azurewebsites.net/api/books";
    var result =
        await WebApiService.DeleteDataAsync(url, SelectedBook.Id);
    switch (result.StatusCode)
    {
        case HttpStatusCode.OK:
            string resultString = await result.Content.ReadAsStringAsync();
            // Do anything you need with the deleted object...
            Book deserializedBook =
                JsonConvert.DeserializeObject<Book>(resultString);
            Books.Remove(SelectedBook);
            MessagingCenter.Send(this, "BookDeleted");
            return;
        default:
            MessagingCenter.
                Send(this, "ServerError",
                $"{result.StatusCode} {result.ReasonPhrase}");
            return;
    }
}
```

The **WebApiService.DeleteDataAsync** method returns a **Book** object representing the deleted one, in case you need to display information to the user. The corresponding instance represented by the **SelectedBook** property is also removed from the **Books** collection.

Working with web APIs and JSON is really easy and fast in Xamarin.Forms, and with a bit of knowledge of the Model-View-ViewModel pattern, you can create a very flexible and solid architecture.

Chapter summary

This chapter discussed a key topic in real-world development, which is invoking web APIs from a Xamarin.Forms project and deserializing JSON responses, based on a SQL database.

You have seen how web APIs make it easy to work with data, taking advantage of the Entity Framework and of the implementation of HTTP verbs (**`GET`**, **`POST`**, **`PUT`**, and **`DELETE`**). You have seen how easy it is to invoke a web API service from Xamarin.Forms, via the **`HttpClient`** class, and how to customize the request.

Finally, you have seen how to deserialize a response using the *de facto* standard library called Newtonsoft.Json, and how to turn the returned JSON into a .NET data model.

The communication between the mobile app and the web API shown in this chapter relies on the HTTPS protocol, which offers a high-level of security. However, this might not be enough—you might be required to go a step further by detecting certificate pinning attacks. This is covered in the next chapter.

Chapter 6 Securing Communication with Certificate Pinning

Securing communications between applications and services is extremely important, and mobile apps are no exception. Even if you use an encrypted channel based on HTTPS, you should never completely trust the identity of the target. For example, an attacker could easily discover the URL your application is pointing to, and put a fake certificate in the middle of the communication between an application and the server, thus intercepting the communication.

This is extremely dangerous—especially if the application handles sensitive data. To avoid this, you can use a technique called *certificate pinning* to dramatically reduce the risk of this kind of man-in-the-middle attack. This chapter describes how to implement certificate pinning in Xamarin.Forms, making your mobile apps more secure.

Describing the certificate pinning implementation

You should implement certificate pinning every time you build apps that handle sensitive data and call HTTPS URLs. Additionally, many enterprises make strict security checks before validating and distributing an app, even if for internal use only, including penetration tests. Penetration tests search for security holes in an application; a common test scenario is simulating a man-in-the-middle attack with a fake certificate.

By implementing certificate pinning, you avoid the risk of certificate replacement, and this penetration test will pass. In order to implement certificate pinning, you will need the valid certificate's public key. This can be provided by your system administrator. For demonstration or development purposes, you can also create a self-signed certificate and retrieve the public key on your own. This scenario will not be covered in this chapter since the documentation from Microsoft already provides excellent coverage.

Certificate pinning in Xamarin.Forms

As you saw in Chapter 5, in Xamarin.Forms you typically use the **`System.Net.Http.HttpClient`** class to send requests over the network, using methods such as **`GetAsync`**, **`PostAsync`**, **`PutAsync`**, and **`DeleteAsync`**. Under the hood, **`HttpClient`** relies on the **`System.Net.HttpWebRequest`** class. The behavior of the latter can be influenced by working with the **`System.Net.ServicePointManager`** class, which can be instructed to check what kind of security protocol is being used and to validate the certificate at every web request.

For a better understanding, create a new Xamarin.Forms project in Visual Studio 2019. When you're ready, add a new file called **EndpointConfiguration.cs**. Code Listing 19 shows the content for the new class.

Code Listing 19

```
namespace CertificatePinning
{
    public class EndpointConfiguration
    {
        // Replace with the public key of your company's certificate
        public const string
            PUBKEY = "Y O U R  V A L I D  K E Y  G O E S  H E R E";

        // Replace with a fake key you want to use for testing
        public const string
            PUBKEYFAKE = "Y O U R  F A K E  K E Y  G O E S  H E R E";
    }
}
```

In this class, you can store both the valid and fake public keys. You can also consider encrypting the real public key for further security, or you might consider other options for storing it, such as the secure storage.

The next step is setting up the **`ServicePointManager`** class. In **App.xaml.cs**, add the following method (which requires a **`using System.Net`** directive):

```
public static void SetupCertificatePinningCheck()
{
       ServicePointManager.SecurityProtocol = SecurityProtocolType.Tls12;
       ServicePointManager.ServerCertificateValidationCallback =
                            ValidateServerCertificate;
}
```

This code assigns the **`ServicePointManager.SecurityProtocol`** with the type of protocol you want to check (Transport Layer Security v1.2 in the previous code), while **`ServerCertificateValidationCallback`** represents the action that must be executed to check if the certificate is valid. The following code demonstrates this:

```
// Requires the following using directives:
// using System.Net.Security;
// using System.Security.Cryptography.X509Certificates;
private static bool ValidateServerCertificate(object sender,
                    X509Certificate certificate,
                    X509Chain chain, SslPolicyErrors sslPolicyErrors)
{
     return EndpointConfiguration.PUBKEY.Replace(" ", null)
            .ToUpper() == certificate?.GetPublicKeyString();
}
```

The certificate is represented by an instance of the **X509Certificate** class and is inferred by the underlying **HttpWebRequest** instance that is making the network call. The **GetPublicKeyString** method returns the public key of the certificate, which you must compare to the key that has been stored inside the **EndpointConfiguration** class.

The next step is invoking the **SetupCertificatePinningCheck** method. An appropriate place for the invocation is the **App** class constructor so that the configuration is applied at startup. Every time you use methods from the **HttpClient** class, the **ValidateServerCertificate** method will be also called, and it will return **true** if the certificate's public key is the key you are expecting.

For example, the following code will cause the previous method to be executed before **GetAsync** can actually access the resource:

```
var client = new HttpClient();
client.BaseAddress =
       new Uri("https://www.mycompanywebsite.com", UriKind.Absolute);
// The following call will cause ValidateServerCertificate to be executed
// before accessing the resource
var response = await client.GetAsync("/mydata");
```

At this point, the **ValidateServerCertificate** method returns **false** because the certificate check failed and caused an **HttpRequestException** to be thrown. However, this specific exception can happen for many reasons, so it is not enough to understand if it was thrown because of the certificate validation failure, or because of other network issues.

Luckily, you can check its **InnerException**; if this is of type **WebException** and the value of its **Status** property is **TrustFailure**, it means that a server certificate could not be validated; therefore, it can perfectly fit in the current implementation.

Code Listing 20 demonstrates how to write code that performs a network call and intercepts different types of exceptions, including a trust failure.

Code Listing 20

```
var client = new HttpClient();
client.BaseAddress = new Uri("https://www.mycompanywebsite.com",
                     UriKind.Absolute);

try
{
      // The following call causes ValidateServerCertificate to be executed

      // before accessing the resource
      var response = await client.GetAsync("/mydata");
}
catch (HttpRequestException ex)
{
```

```
    if (ex.InnerException is WebException e && e.Status ==
        WebExceptionStatus.TrustFailure)
    {
        // The server certificate validation failed: potential attack
    }
    else
    {
        // Other network issues
    }
}
catch (Exception)
{
    // Other exceptions
}
```

In my projects, I have a service class that contains static methods to make invocations to web services and APIs. My view models call these static methods and, in case of trust failure, they send messages to the user interface through the **`MessagingCenter`** class. When notified, the user interface will show appropriate error handling options.

Considerations for debugging

You might want to avoid checks for certificate pinning while debugging, at least once you have made sure and tested that it works with your app. There are several reasons for doing this. For example, your development and production environments have different certificates, and it's not useful to check for the development certificate every time, or you want to avoid overhead while debugging other features.

I disable these checks while debugging with a simple **`#if`** preprocessor directive that is added to the **`ValidateServerCertificate`** method as follows:

```
private static bool ValidateServerCertificate(object sender,
                X509Certificate certificate,
                X509Chain chain, SslPolicyErrors sslPolicyErrors)
{
#if DEBUG
      return true;
#endif
      return EndpointConfiguration.PUBKEY.Replace(" ", null)
             .ToUpper() == certificate?.GetPublicKeyString();
}
```

This simple addition will simplify the way you do debugging, especially on simulators, which might not support all the protocols.

Chapter summary

When your app works with sensitive data, security is even more important. Implementing checks for certificate pinning is as easy as it is important. In this chapter, you have seen how to leverage the **`X509Certificate`** class and its methods to make this check, and you have also seen how to avoid checks while debugging to avoid overhead.

Security is an important topic for everyone in the team—certainly for developers, but also for the businesspeople that need to promote or sell apps. But security is not the only important topic, especially for the business. Another important topic is driving decisions and investments based on app usage. This is the topic of the next chapter.

Chapter 7 App Center: Drive Business Decisions with Analytics

Every project depends on a budget. Decision-makers in a company invest budget on a project to make money. That's how business works, said in two sentences. Projects are an offer to customers, and for stakeholders, investors, and decision-makers, it's extremely important to understand how to improve the quality of the product (an app) and make customers happy.

People who invest money in the project you are working on will often ask about how many active users the app has, how much time users spend on it, what the most and least used features are, and how reliable the app is. Generally speaking, they will ask you about numbers. This is where analytics come in.

Analytics allow developers to collect information about the usage of an app and will help you understand what features are most used and how reliable the app is. The Microsoft App Center offers an extremely powerful analytics engine that will help you provide answers to decision-makers and investors, so that they can drive their business decisions in the direction of success. This chapter explains what the Microsoft offering is for analytics, and how to set up both the App Center and apps to collect analytic data about app usage, while keeping privacy considerations in mind.

 ***Note: This chapter describes everything that is useful for analyzing app usage, but App Center offers much more than this. You can bookmark the documentation for further studies.***

Analytics and diagnostic data

Among all the possibilities offered by Microsoft's Visual Studio App Center, this chapter focuses on two features:

- **Analytics**: This allows you to understand a user's app usage by collecting events. An event can be a tap gesture, navigation to a page, and so on. Analytics also include the phone model of the user, the operating system, the OS version, the country of the user (based on the phone carrier), the number of sessions per day and week, and much more (see the section called "Understanding analytic data" later in this chapter).
- **Diagnostics**: This tracks errors and crashes that happened during the application lifetime, and stores the .NET exception information for a full understanding of the problem, such as reading the stack trace.

App Center also collects common information that is independent from analytics and diagnostics. For example, it generates a session every time a device connects to the analytics engine, assigning a GUID to the session to make this identifiable. It also collects other information such as the IP address, the session duration and, in general, information that helps you understand how much time the user spent on the app without any details on the actions taken.

Setting up a sample project

Because the focus of this chapter is analytics and collecting usage data, the sample application will be very simple. Create a new Xamarin.Forms blank project and add the XAML shown in Code Listing 21 to the **MainPage.xaml** file.

Code Listing 21

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="AnalyzingActions.MainPage">

    <StackLayout VerticalOptions="Center">
        <Button x:Name="FirstPageButton"
                Text="Open first page"
                Clicked="FirstPageButton_Clicked"/>

        <Button x:Name="SecondPageButton"
                Text="Open second page"
                Clicked="SecondPageButton_Clicked"/>

        <Button x:Name="HelpButton"
                Text="Help"
                Clicked="HelpButton_Clicked"/>
    </StackLayout>
</ContentPage>
```

The app will simply allow for navigating between two pages and will allow you to show a help dialog. The reason behind this is that we want to track page navigation, how many times a page is opened instead of another one, and how many times the user is tapping a button (in this case to ask for help).

Next, add two content pages to the project: one called **FirstPage.xaml**, and one called **SecondPage.xaml**, leaving the autogenerated XAML markup. In **App.xaml.cs**, enable navigation by changing the assignment of the **`MainPage`** property as follows:

```
MainPage = new NavigationPage(new MainPage());
```

You will need to add other pieces, such as event handlers for the buttons, but this will be done after setting up App Center.

Configuring the App Center

App Center is a complete solution for build automation and continuous integration, automated tests, app distribution, analytics, and diagnostics. In this chapter, I will demonstrate only the analytics and diagnostics features, but it's really worth mentioning that you can take full advantage of the App Center to automate builds from a code repository, with automated testing and distribution of the app to the selected people for testing and creating different kinds of releases, such as alpha, beta, production, and enterprise.

You will first need to log into the App Center. You can register using a Microsoft, GitHub, Facebook, or Google account. When logged in, you will see the welcome message shown in Figure 16.

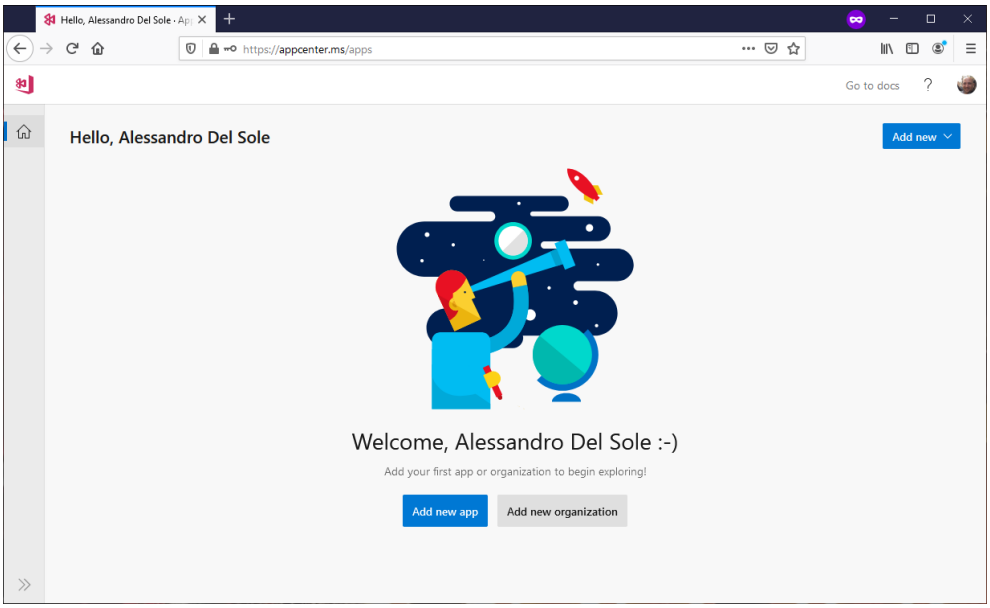

Figure 16: Logging into the App Center

Click **Add new app**. As you can see in Figure 17, you will be asked to enter an app name, the target operating system, and the platform used for development. In the case of a Xamarin.Forms project, the platform will always be **Xamarin**. For the operating system, you will need to select which one you want to target, for example iOS, like in Figure 17.

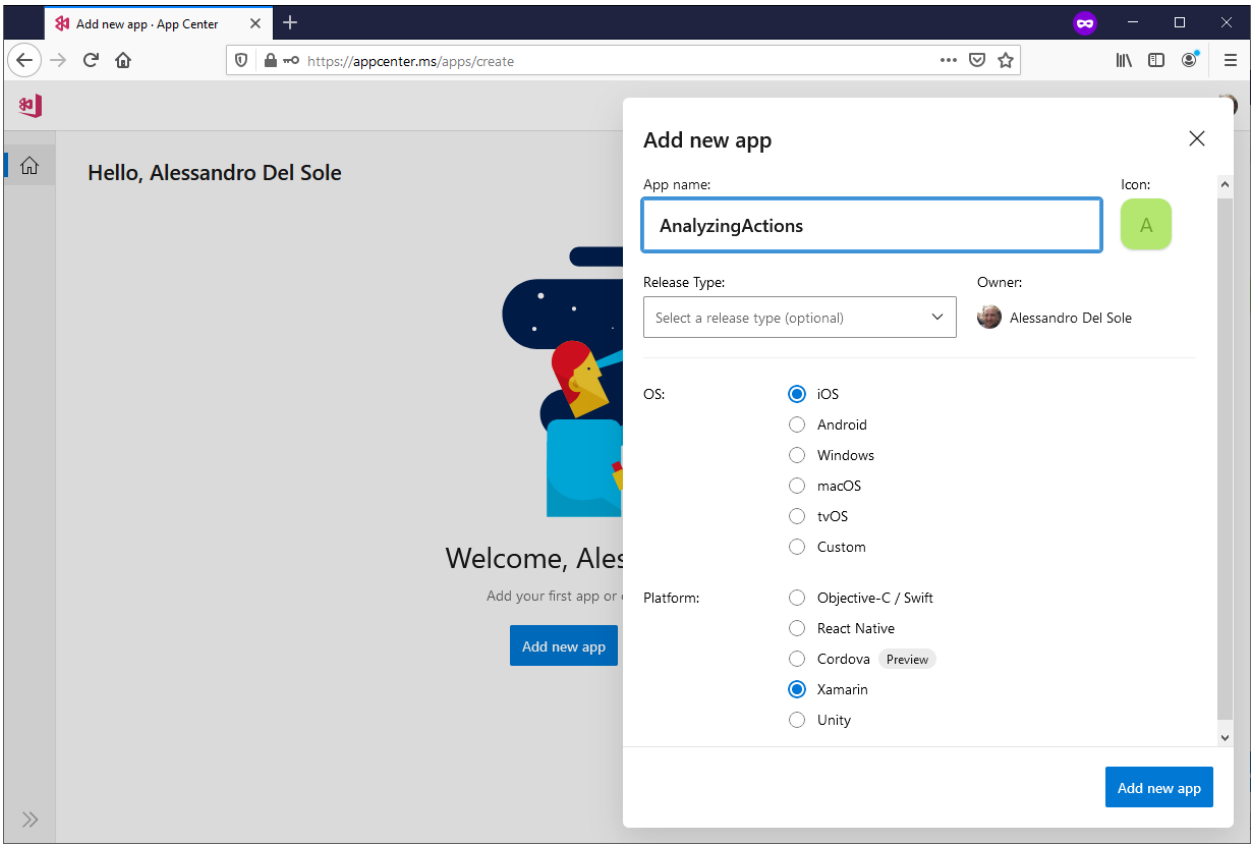

Figure 17: Adding an app to App Center

You will need to add an app for each OS you want to target, so if you are targeting iOS and Android, you will repeat the same steps to add an app, specifying **Android** as the OS and **Xamarin** as the platform. In summary, you will have at least two apps; and you can have more if you target, for example, UWP.

When the app has been added, App Center should automatically display the instructions to set up a Xamarin project. If this does not happen, you can click the **Overview** button (the one below the app icon on the toolbar at the left side of the page). Figure 18 shows the instructions to set up a Xamarin.Forms project to work with analytics.

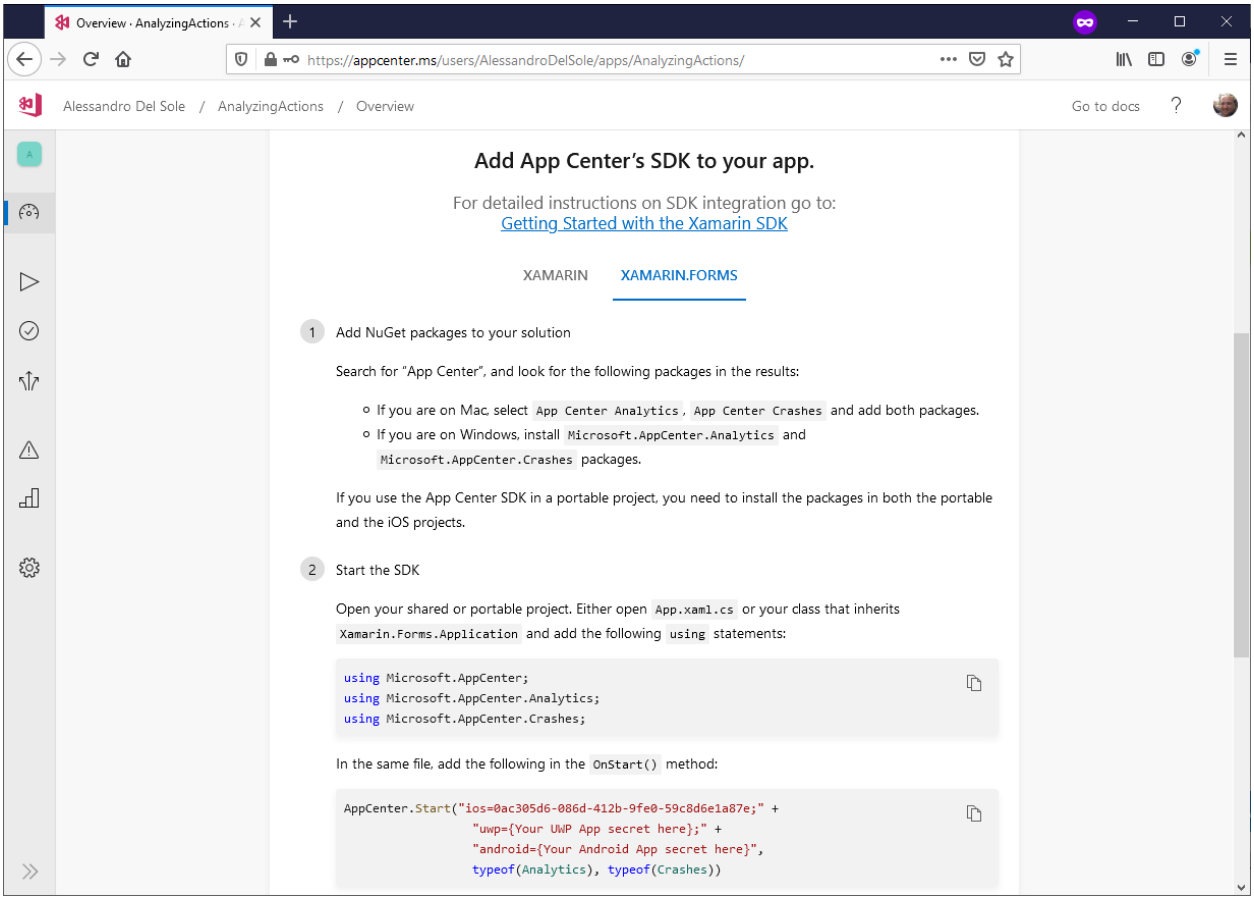

Figure 18: The instructions to set up a Xamarin.Forms project to work with analytics

As you can see, the instructions explain how to include both analytics and diagnostics information in the code. The next paragraph walks through the complete steps to set up the project.

Adding analytics and privacy consent to the app

Before writing code, you need to install a few NuGet packages. More specifically, you need the following (see Figure 19):

- Microsoft.AppCenter
- Microsoft.AppCenter.Analytics
- Microsoft.AppCenter.Crashes

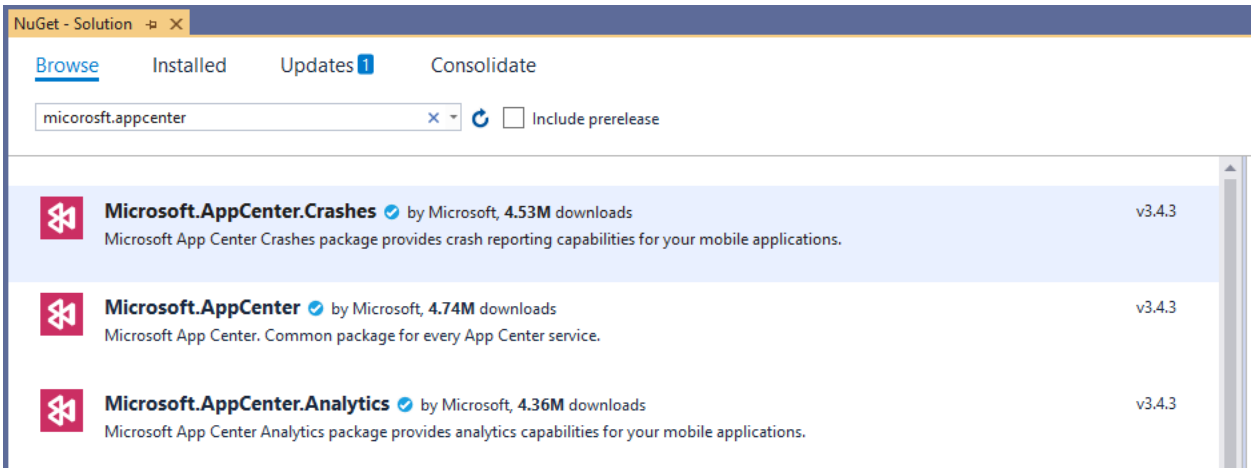

Figure 19: Analytics NuGet packages

Make sure that these NuGet packages are installed to all the projects in the solution.

> ***Tip: Because Microsoft.AppCenter is a dependency for the other packages, you can start installing one between Microsoft.AppCenter.Analytics and Microsoft.AppCenter.Crashes, and Microsoft.AppCenter will be automatically installed.***

When you created the iOS and Android apps in App Center, an *app secret* was created. An app secret is a GUID that connects your app to your App Center account, and one has been generated for all the apps you added. You can find the app secret in the code presented in the instructions inside App Center.

Once you have taken all the app secrets, you can add the following code in the **`OnStart`** method of the **`App`** class, replacing the app secrets with yours:

```
protected override void OnStart()
{
    AppCenter.Start("android=ANDROID_SECRET_GOES_HERE;" +
                    "ios=iOS_SECRET_GOES_HERE;",
                    "uwp=UWP_SECRET_GOES_HERE;",
                    typeof(Analytics), typeof(Crashes));
}
```

An app secret is connected to a specific release. In the current example, I only got a default release when I added the apps, but if you create more app releases (such as alpha, beta, and production), you will get a different app secret for each kind of release. As an implication, you will need to add as many **`AppCenter.Start`** invocations as releases you have (or you might implement your own detection mechanism for the release and switch between app secrets with an **`if`** or **`switch`** block).

Tracking events and errors

With analytics, you can track events and .NET exceptions occurring during the app lifetime. Events can be any actions the user takes while using the app, such as tapping buttons, opening pages, scrolling lists, and so on. Events can be tracked using the **`Microsoft.AppCenter.Analytics`** class, which exposes the **`TrackEvent`** method. In order to understand how it works, add the following event handler for the Help button in the MainPage.xaml.cs file:

```
private async void HelpButton_Clicked(object sender, EventArgs e)
{
    await DisplayAlert("Help",
        "This app will help you understand how App Center Analytics work",
        "OK");
    Analytics.TrackEvent("Help button pressed");
}
```

As you can see, when the user presses a button, in addition to performing a task (in this case showing an alert), an event is sent to the analytics engine in a very detailed way. **`TrackEvent`** is very flexible, as it allows you to specify additional, custom information to the event. This is done by adding **`Dictionary<string, string>`** as an argument, like in the following example, which demonstrates how to group events by category:

```
Analytics.TrackEvent("Help button pressed",

           new Dictionary<string, string> { { "Category", "UI" } });
```

In this way, you could divide events by category and filter them more easily in App Center. Let's keep the simpler way for now. First, add the following event handlers for the buttons that open the two pages:

```
 private async void FirstPageButton_Clicked(object sender, EventArgs e)
 {
     await Navigation.PushAsync(new FirstPage());
 }

 private async void SecondPageButton_Clicked(object sender, EventArgs e)
 {
     await Navigation.PushAsync(new SecondPage());
 }
```

In the **FirstPage.xaml.cs** file, add the following code:

```
protected override void OnAppearing()
{
    base.OnAppearing();
    Analytics.TrackEvent("First Page opened");
}
```

```
protected override void OnDisappearing()
{
    base.OnDisappearing();
    Analytics.TrackEvent("First Page closed");
}
```

It is useful to track both the opening and closing events to understand how much time the user spends on a page. In the **SecondPage.xaml.cs file**, the code will also simulate an exception for error tracking:

```
protected override void OnAppearing()
{
    base.OnAppearing();
    Analytics.TrackEvent("Second page opened");
}
protected override void OnDisappearing()
{
    try
    {
        base.OnDisappearing();
        Analytics.TrackEvent("Second page closed");
        throw new InvalidOperationException();
    }
    catch (Exception ex)
    {
        Crashes.TrackError(ex);
    }
}
```

The **`Microsoft.AppCenter.Crashes.Crashes`** class exposes the **`TrackError`** method, which receives an object of type **`Exception`** (or derived) as an argument. **`TrackError`** will send the full exception information, including the stack trace, to the analytics engine, and this will be crucial to help developers debug any possible error. You will shortly see the result of event and error tracking in the App Center website, but an important piece is still missing: the user consent for analytics.

Managing the user consent for analytics

You cannot use analytics and track the user's behavior without the user's explicit permission. This is required by laws and regulations, and it's something you cannot forget. You should not even track any information that could explicitly identify the user. This info isn't necessary for the purpose of analytics, which is intended for usage and behavior analysis only.

Note: Privacy is a crucial topic for mobile apps in general, and especially if an app uses analytics. I would recommend you ask a lawyer or specialized companies about privacy consent before implementing analytics, and before writing and presenting to the user any privacy policy text and analytics consent text. It is important that you do not underestimate this point.

In the current example, the user's explicit permission is asked in a very simple way through an alert. In your apps, you will want to present the full text of the privacy policy and store the acceptance or rejection inside the app settings or a database.

Add the following code to the **MainPage.xaml.cs** file:

```
bool? isAnalyticsConsentSet = null;
protected override async void OnAppearing()
{
    base.OnAppearing();
    if (!isAnalyticsConsentSet.HasValue)
    {
        bool result = await DisplayAlert("Consent",
            "Do you provide your consent for analytics?", "Yes", "No");
        await Analytics.SetEnabledAsync(result);
        await Crashes.SetEnabledAsync(result);
        isAnalyticsConsentSet = result;
    }
}
```

The user's choice is stored inside an in-memory variable for the sake of simplicity, but do not forget the aforementioned considerations. The key point here is the **`SetEnabledAsync`** method exposed by both the **`Analytics`** and **`Crashes`** classes. You can enable or disable analytics and crash tracking by simply passing **`true`** (enabled) or **`false`** (disabled) to this method and leave all the existing tracking code you wrote unchanged.

Running the app and doing actions

If you run the sample app, the first thing you will see is the analytics consent request, as shown in Figure 20.

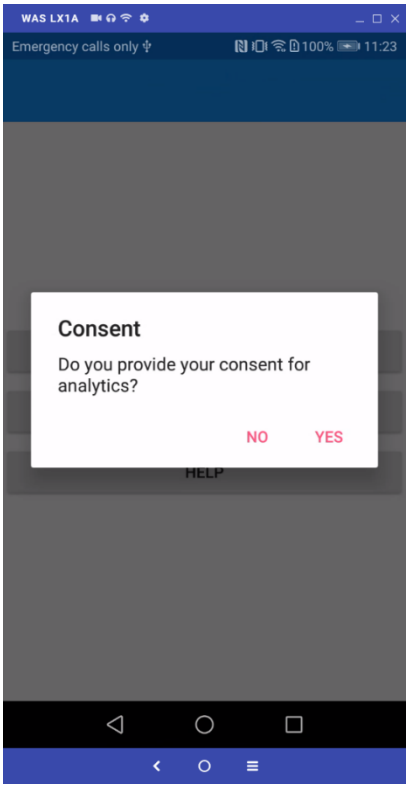

Figure 20: Asking user's permission for analytics

Tap **Yes**, then alternately open the two pages, go back, and tap the **Help** button several times. This will generate a number of analytics events.

> ***Tip: A nice way to collect sample analytics data would be to use the app on different devices, in different days, and setting different languages on the devices. This will help with creating more structured views in App Center.***

After running the app a few times, some analytics data should be available. The next section describes analytics data that you can find in App Center.

Understanding analytics data

All the analytics data is available in App Center. Select one of the apps you created previously (Android or iOS), and then click **Analytics** in the toolbar at the left side of the screen. The first piece of information you get is about users. More specifically, unique active users, shown in a graph that shows the development of user access over time. **Daily sessions per user** shows how many times a user entered the app every day, **Session duration** shows how much time the user spent in the app every time they used it, and **Top devices** shows a list of the most used devices for the selected operating system. Figure 21 shows this information.

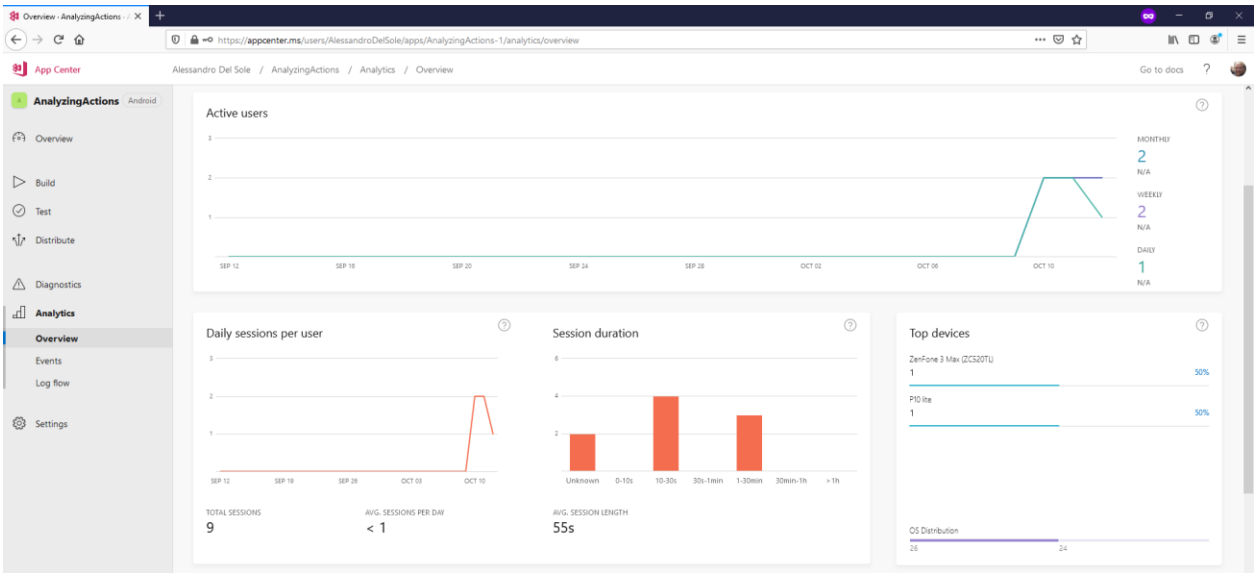

Figure 21: Analyzing user access

Scrolling down the page (see Figure 22), you will see additional information about countries and languages. Sometimes it takes time for the analytics engine to collect information, so the **Countries** area might stay empty for a while, like in Figure 22, but here you will see where your users come from, and what language they have set on their device.

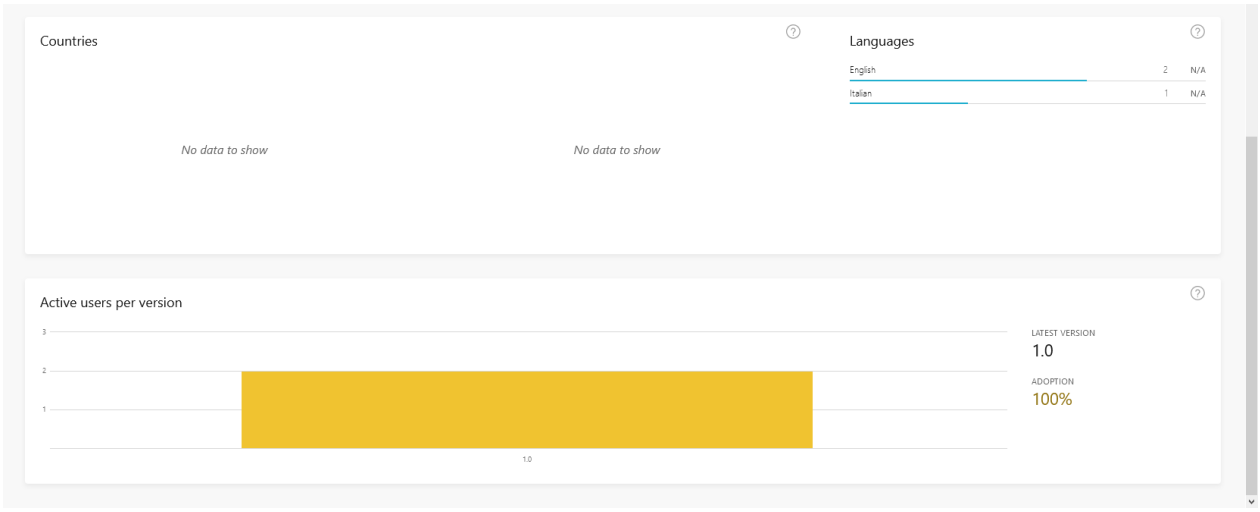

Figure 22: Countries and language information

If multiple versions of your app are currently available, the **Active users per session** area will show a report about how many users are using each version. The next group of information is about events, and probably is the most interesting when it comes to deciding in which features to invest more (or less). Click **Events** in the toolbar at the left. You will see the full list of events that you tracked invoking the **`Analytics.TrackEvent`** method, as shown in Figure 23.

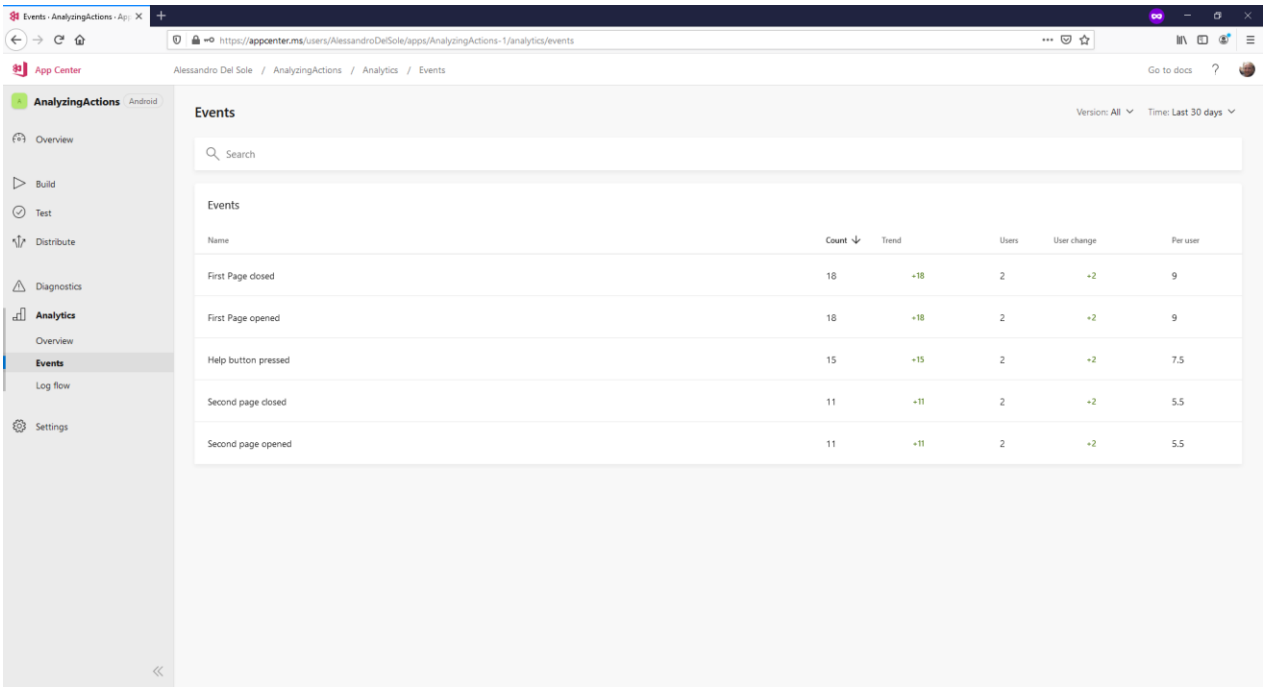

Figure 23: Analyzing events

For each event you will see the total count, the positive or negative trend, how many users are involved, and the count of events per user. The default timeframe is the last 30 days, but you can select a different interval with the dropdown at the upper-right corner. Here you basically get a summary of how much features are used. You can also click individual events to see more detailed information, and Figure 24 shows an example.

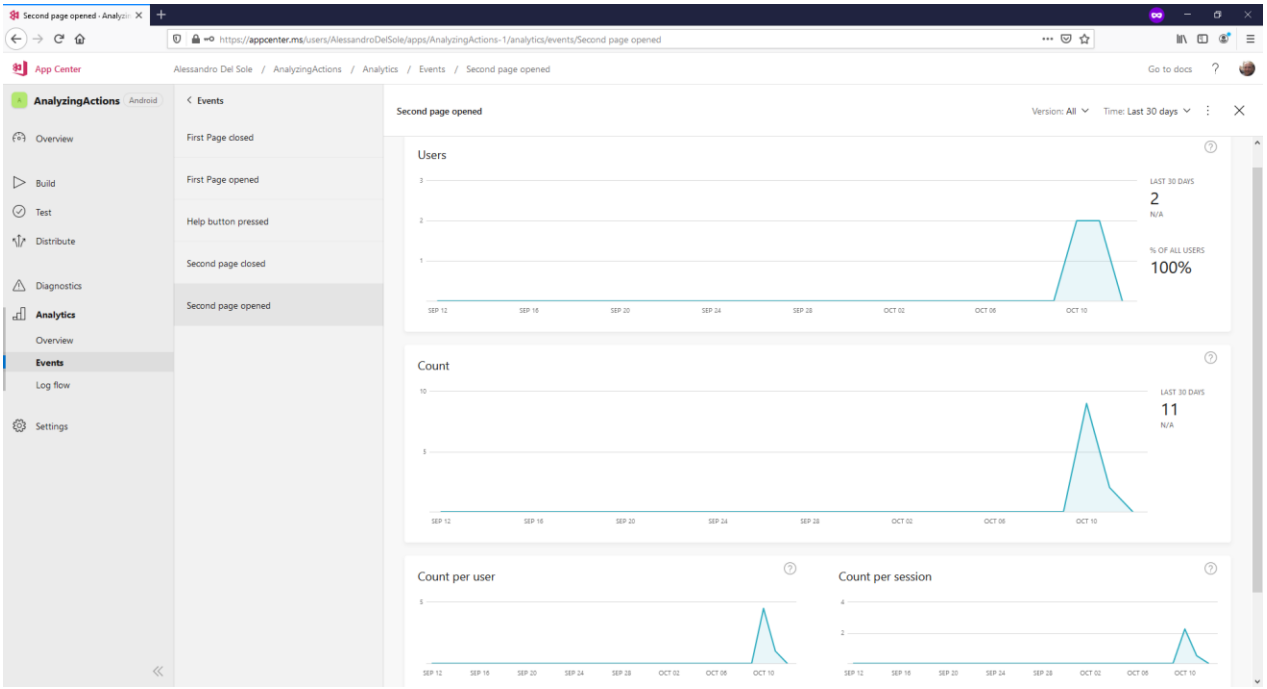

Figure 24: Event details per user

In this detailed view, you can see graphs that show how users interacted with the feature that raised the event over time. This is another important way to analyze the app usage; for example, it gives you an idea of how a feature is used since its release, and it can help you understand if the feature was easy to find, easy to use, useful for the user, and so on.

The last piece of analytic data to walk through is diagnostic data. Click **Diagnostics** in the toolbar at the left. When the page is opened, you will see issues grouped by crashes and errors. In the current example, there are errors raised intentionally by the app and tracked. Figure 25 shows what the page looks like.

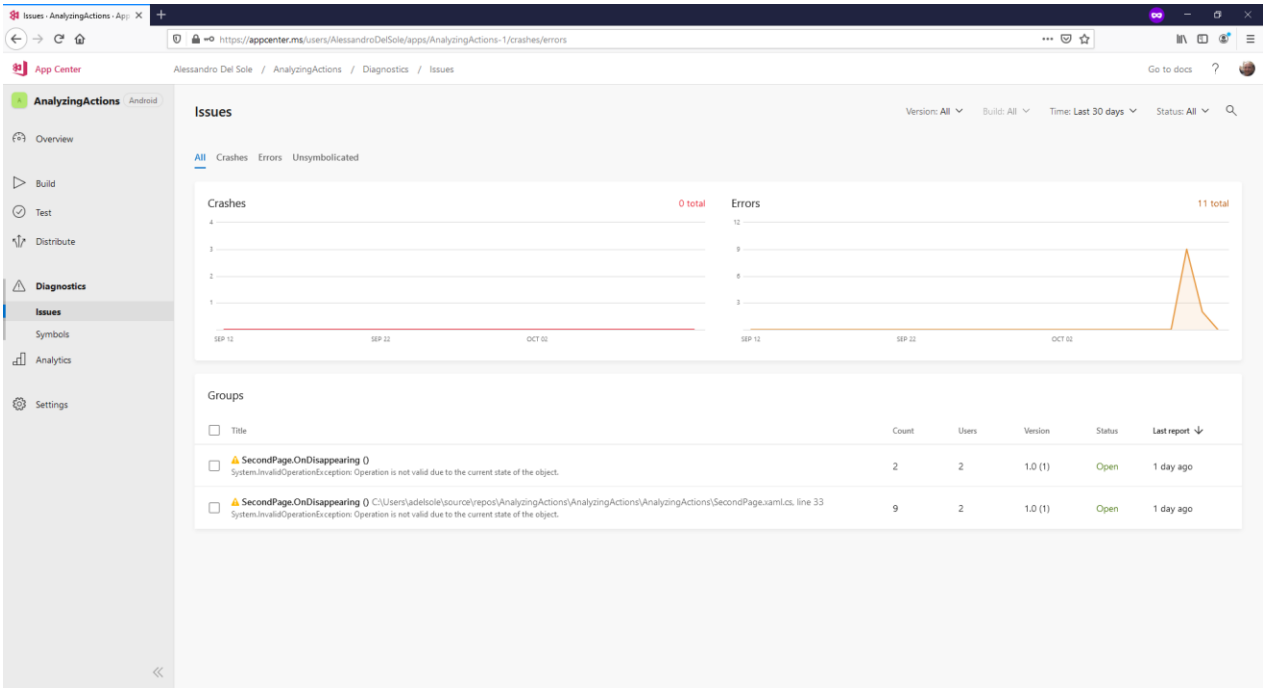

Figure 25: Diagnostic information

You have graphs to see crashes and errors over time, but the most useful information here is probably the list of .NET errors at the bottom of the page. In the current example, these have been raised intentionally to make the analytics engine generate reports, but in your apps, it will be important to add **`Crashes.TrackError`** invocations inside **`Try..Catch`** blocks. If you click an error, you will get a detailed view that includes the stack trace, as shown in Figure 26.

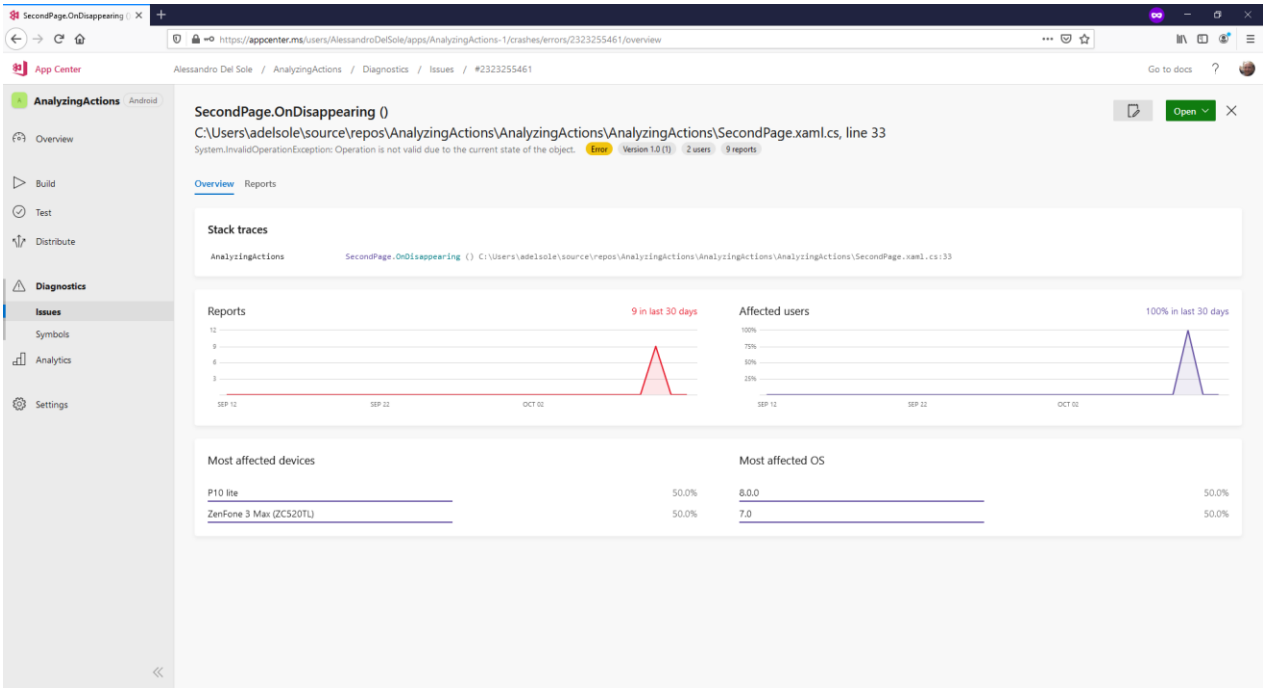

Figure 26: Detailed view of .NET exceptions

Diagnostic information here includes the stack trace, including the line of code and file that caused the error, a graph that shows how many times it happened, a graph that shows the number of users affected, and a summary of the most affected devices and operating systems.

In my experience, Android 9 is an operating system that needs more attention than its predecessors and successors due to its API, so having information on the OS version has been crucial. With all this information in hand, all the roles involved in the project will have an option to increase the quality and success of an app—from stakeholders and investors that will spend more budget on features that are useful for the customers, to developers and designers who will better understand how to fix problems and improve the interaction.

Hints about Application Insights

The way App Center collects information is certainly clear and useful for most of the roles in the project, but for further queries, data grouping, and advanced analysis, the best way is connecting your App Center account to Application Insights in Microsoft Azure. Application Insights also offers a structured query language to write powerful, custom queries. This is not in the scope of this chapter, but it is worth a mention. You can find more information in the documentation.

Chapter summary

Decision-makers need to understand where to drive the budget for a project, and analytics can help. With analytics, you can track events and errors that happen in your app, understanding what features users like most, how much time they spend on a specific part, and more. In this way, all roles involved in the project can give their best contributions to improve the product. As a developer, you are interested in writing code, and the **`Analytics`** and **`Crashes`** classes, with their **`TrackEvent`** and **`TrackError`** methods, make it extremely easy to send analytics information.

Last but not least, these classes also make it simple to handle the user's explicit permission to collect analytics, by passing true or false to the **`SetEnabledAsync`** method that allows you to leave the rest of the code unchanged. Improving the product quality is also possible by providing the user the best experience, which also comes through a nice user interface. This can be easily enriched with Lottie animations, which are discussed in the next chapter.

Chapter 8 Displaying After Effects Animations with Lottie

Animated images and illustrations are an important way to enrich the look and feel of mobile applications to make them more appealing. In my daily work, I use animated illustrations connected to user interactions to provide educational content. In the last few years, an interesting animation format has become extremely popular among designers and developers, and it is a format produced by the famous Adobe After Effects design suite. This format is not actually a proprietary one, rather it is JSON.

Adobe After Effects can produce files in a specialized JSON format containing the design for animated illustrations. In order to render these JSON animations, several libraries exist, but the most important is called Lottie, which is offered by Airbnb. One of its major advantages is that Lottie files are generally much smaller than an equivalent animated .gif file.

For Xamarin.Forms, a Lottie implementation also exists and allows you to play beautiful animated JSON illustrations in your mobile apps. This is the topic of this chapter.

Xamarin.Forms and Lottie

As I mentioned in the introduction, there are several implementations for Lottie that target many platforms, including native Xamarin.iOS and Xamarin.Android. Luckily, an open-source implementation also exists for Xamarin.Forms, and it is very easy to use.

For the next example, create a new, blank Xamarin.Forms project and open the NuGet Package Manager user interface. The NuGet package you need to install is called **Com.AirBnb.Xamarin.Forms.Lottie**, as demonstrated in Figure 27.

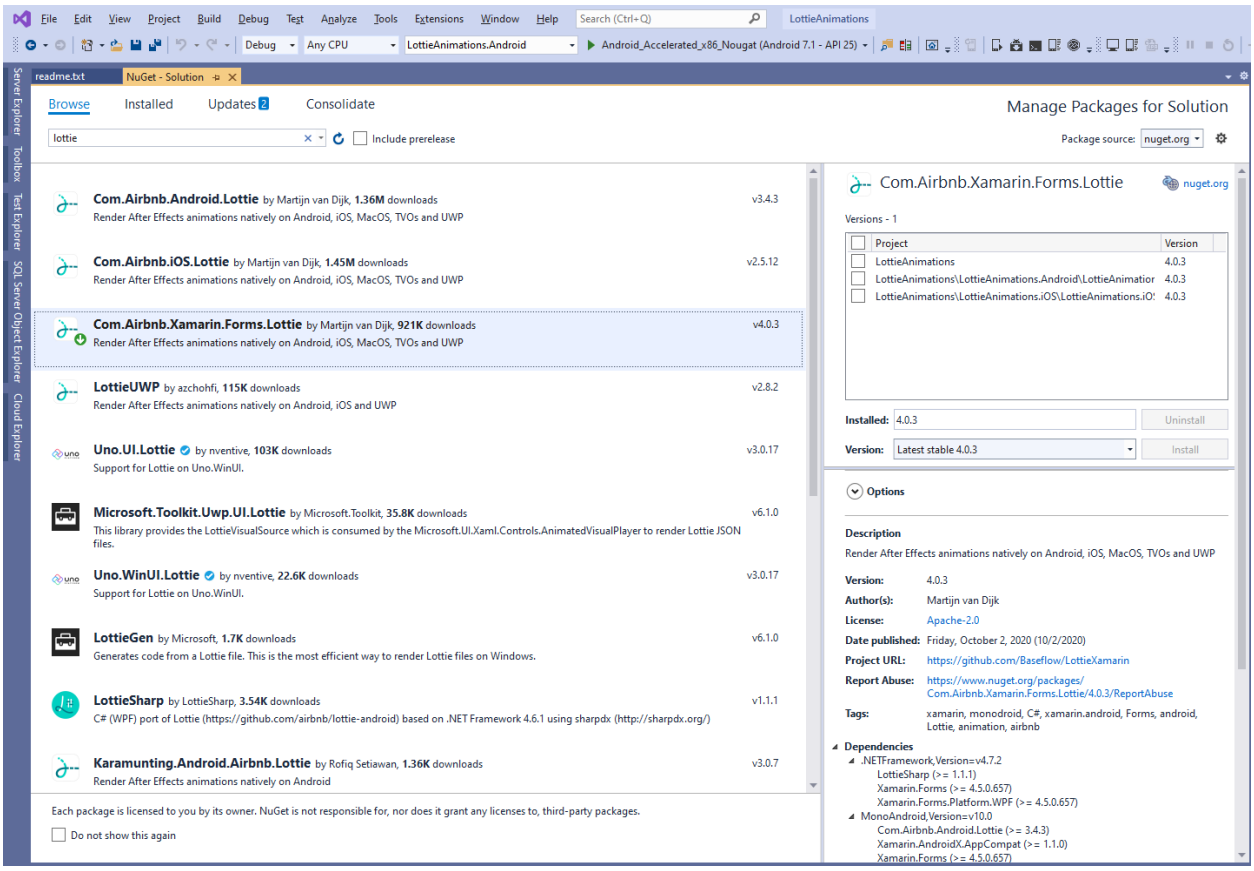

Figure 27: Installing the Lottie NuGet package

This package must be installed to all the projects in the solution. As you will see shortly, the library offers a specific view that is very easy to use, but before using it, you need some sample animations.

Adding Lottie animations to the project

The way your app consumes Lottie animations depends on the source of the animations themselves. If your team has designers that produce Lottie JSON files, you will add such files into the solution as follows:

- For Android, your JSON files must be added into the **Assets** folder, and the **Build Action** must be set as **AndroidResource**.
- For iOS, your JSON files must be added into the **Resources** folder, and the **Build Action** must be set as **BundleResource**.

If Lottie animations are instead hosted on a web server, either public or private, the animation can be simply consumed by passing its URI to the Lottie view. A very nice and popular source for Lottie animations is the Lottiefiles.com website. On this site, you can find both free and paid animations, illustrations, icons, and much more. For the current example, I will use the JSON animation included in the Lottie library repository, which is already included inside the companion code for your convenience.

Note: The JSON for Lottie animations can be very long and complex. The goal of this chapter is not to explain how it is structured, so it is left to you for further study.

You are certainly free to use a different animation, in which case you will decide how to consume it based on what you learn in the next sections.

Displaying Lottie animations

The Lottie library provides a new view called **`AnimationView`**. Before you can use it in your XAML, you need to add the following XML namespace declaration at the **`ContentPage`** level:

```
xmlns:lottie="clr-namespace:Lottie.Forms;assembly=Lottie.Forms"
```

In its simplest form, the **`AnimationView`** can be declared as follows:

```
<lottie:AnimationView
    HorizontalOptions="FillAndExpand"
    VerticalOptions="FillAndExpand"
    x:Name="animationView"
    Animation="lottielogo.json"
    AnimationSource="AssetOrBundle"
    AutoPlay="True" RepeatMode="Restart"
    RepeatCount="3"/>
```

The **`AnimationSource`** property specifies where the JSON animation comes from. In this case, **`AssetOrBundle`** means the animation is a local file. Other options can be **`Url`**, **`Stream`**, and **`Json`**. The first two are self-explanatory; the latter allows you to bind a **`string`** object containing the JSON. With **`AssetOrBundle`**, **`Url`**, and **`Stream`**, you use the **`Animation`** property to specify the animation file name; with **`Json`**, you need to assign a property called **`AnimationJson`**. In the example, **`AutoPlay`** is set as **`True`**, and the animation will restart (**`RepeatMode`**) three times (**`RepeatCount`**). **`RepeatMode`** can also be set with **`Reverse`** and **`Infinite`**, and the latter helps you avoid the need of setting the **`RepeatCount`** property.

If you run the sample app, you will see the Lottie animation being played on your device. Figure 28 shows a static frame of the animation, but it will be moving when running.

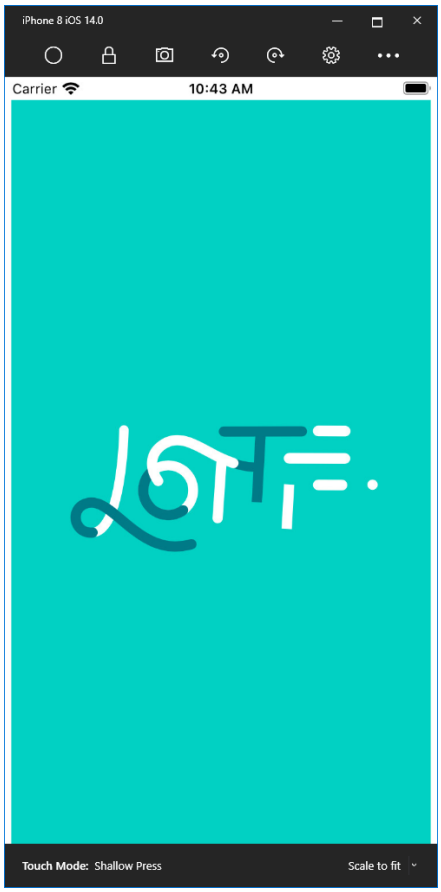

Figure 28: Lottie animation running in Xamarin.Forms

The Lottie library allows you to control with more granularity the way the animation is played, and this is discussed in the next section.

Controlling the animation

The Lottie library provides methods and properties that allow developers to control the flow of the animations. Methods that you can use to manage the animations are **`PlayAnimation`**, **`PauseAnimation`**, and **`StopAnimation`**, which are self-explanatory. There are also events that can be useful to understand the status of playing, such as:

- **`OnAnimationLoaded`**, raised when the animation has been loaded by the **`AnimationView`**.
- **`OnAnimationUpdated`**, raised when playing moves on.
- **`OnFailure`**, raised when loading or playing the application fails.
- A series of self-explanatory events that are raised in the different phases of playing the animation, such as **`OnPlayAnimation`**, **`OnPauseAnimation`**, **`OnFinishedAnimation`**, **`OnRepeatAnimation`**, **`OnResumeAnimation`**, and **`OnStopAnimation`**.

You can also control the user interaction via the **Clicked** event, which is raised when the user taps the animation, and with the **Command** property, which you use instead of the **Clicked** event if your implementation is based on the Model-View-ViewModel pattern.

Especially when your animations are loaded from a web resource via URI, failures in loading the animation can happen, and the **OnFailure** event is raised. Think of connectivity issues, or of an animation that is not compatible with the After Effects JSON structure. In this case, the **AnimationView.FallbackSource** property comes to help. It is of type **ImageSource** and can be assigned with a static image that will be displayed in case of errors. Additional properties of interest are:

- **Duration**, of type **long**, which represents the duration of the animation expressed in frames / frame rate * 1,000.
- **Progress**, **MaxProgress**, and **MinProgress**, all of type **float**, which respectively represent the progress of the animation, the final point at which the animation must stop playing, and the starting point at which the animation should start playing.
- **Frame**, of type **int**, which allows for setting the progress of the animation at an exact frame.
- **MaxFrame** and **MinFrame**, both of type **int**, which respectively allow setting the exact frame where the animation is set to stop and start.
- **IsAnimating**, of type **bool**, which returns true if the animation is being played.
- **Speed**, of type **int**, which allows setting the speed of the animation. The default is 1.

The following XAML shows a simple example in which you prepare the **AnimationView** for programmatic control:

```
<StackLayout>
    <lottie:AnimationView x:Name="animationView"
     Animation="lottielogo.json" AnimationSource="AssetOrBundle"
      FallbackResource="Error.png" AutoPlay="False" />

  <StackLayout Orientation="Horizontal" Margin="20,0,0,0">
     <Button x:Name="PlayButton" Clicked="PlayButton_Clicked" Text="Play" />
     <Button x:Name="PauseButton" Clicked="PauseButton_Clicked"
             Text="Pause" />
     <Button x:Name="StopButton" Clicked="StopButton_Clicked" Text="Stop" />
  </StackLayout>
</StackLayout>
```

Notice how the **FallbackResource** property is assigned with an image file that will be shown in case the animation fails to load. As you would expect, the event handlers that control playing the animation are very easy and look like the following:

```
private void PlayButton_Clicked(object sender, EventArgs e)
{
    animationView.PlayAnimation();
}
```

```
private void PauseButton_Clicked(object sender, EventArgs e)
{
    animationView.PauseAnimation();
}

private void StopButton_Clicked(object sender, EventArgs e)
{
    animationView.StopAnimation();
}
```

In this way, you can satisfy even the more complex design requirements, while at the same time enriching the application behavior.

Chapter summary

Lottie is a library from Airbnb that allows for rendering Adobe After Effects animations, created in a specialized JSON format. An implementation for Xamarin.Forms exists and offers the **`AnimationView`** control, which makes it easy to show animations with both a very basic implementation and more granularity of control with methods, properties, and events.

Lottie animations are certainly an important addition for making mobile apps more beautiful, but they are not the only pieces of content your app might display. An important part is made of documents, which are discussed in the next chapter.

Chapter 9 Displaying and Sharing Documents

Mobile apps offer the great benefit of bringing our documents everywhere with us, no matter if they are stored locally on the device or on a cloud service. This is a great possibility because you might need to receive, send, or share documents with other people, even when you are not at your desk working on a computer. The most popular file format for sharing documents is PDF, so the focus of this chapter is going to be on this file format, but at least for the sharing part, you will be able to reuse the described techniques with any format.

In terms of mobile app development with Xamarin.Forms, there is no built-in view that allows for displaying PDF documents, but luckily, there are third-party controls that make this easy. Sharing files is instead provided by the Xamarin.Essentials library, which simplifies development with yet another scenario. In the next paragraphs, you will learn how to display PDF documents in your mobile apps using Syncfusion's PDF viewer, and how to share PDF documents by leveraging the Xamarin.Essentials API.

Note: A valid subscription for Syncfusion's Essential Studio for Xamarin is required. Actually, you can use a trial version of the libraries described in this chapter by simply downloading the NuGet packages, and the app will show a message about the trial status. If you are an individual developer or work for a small business, you can also consider the Community License. Make sure you read the linked page to see if you are eligible.

Chapter focus and sample project setup

The goal of this chapter is explaining how to display PDF documents in your mobile apps built with Xamarin.Forms, and how to share them using the operating system's built-in sharing mechanism, invoked by the Xamarin.Essentials cross-platform API. To provide you with an example that is working locally, I have included a sample PDF document as an embedded resource of the companion project. In your real project, you will be free to choose where the documents will be opened from, and you will be able to use the techniques described in the "Working with files" section of Chapter 1 and usual Xamarin.Forms file handling techniques.

Create a new blank project. Displaying PDF documents is achieved using the **`SfPdfViewer`** control by Syncfusion, which is part of their Essential Studio for Xamarin suite available as a NuGet package.

Note: You need to add the following URL to the NuGet Package Manager's sources to be able to download Syncfusion's packages for Xamarin:
http://nuget.syncfusion.com/nuget_xamarin/nuget/getsyncfusionpackages/xamarin.

When you're ready, browse NuGet for the **Syncfusion.Xamarin.SfPdfViewer** package (see Figure 29) and install it to all the projects in the solution.

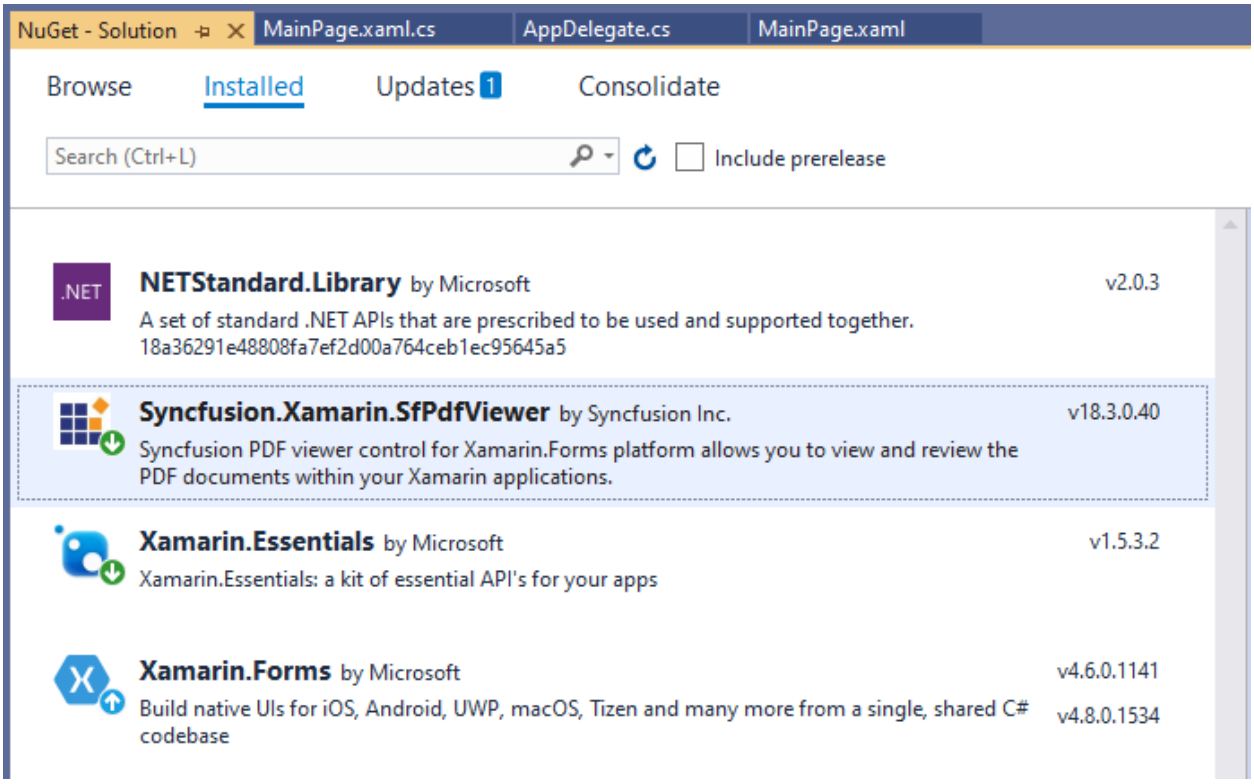

Figure 29: Installing the Syncfusion.Xamarin.SfPdfViewer package

When the package has been installed, you need an additional step for iOS. In the **AppDelegate.cs** file, add the following lines of code to the **`FinishedLaunching`** method right before the invocation to the **`LoadApplication`** method:

```
Syncfusion.SfPdfViewer.XForms.iOS.SfPdfDocumentViewRenderer.Init();

Syncfusion.SfRangeSlider.XForms.iOS.SfRangeSliderRenderer.Init();
```

These are required to initialize the PDF viewer.

The user interface for the main page is very simple. It includes the PDF viewer and two buttons: one for loading a document, and one for sharing the document. Code Listing 22 contains the XAML code for this.

Code Listing 22

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="WorkingWithDocuments.MainPage"
             xmlns:syncfusion="clr-
                   namespace:Syncfusion.SfPdfViewer.XForms;
```

```
                    assembly=Syncfusion.SfPdfViewer.XForms">

    <Grid x:Name="pdfViewGrid">
        <Grid.RowDefinitions>
            <RowDefinition />
            <RowDefinition Height="40" />
        </Grid.RowDefinitions>
        <syncfusion:SfPdfViewer
            x:Name="PdfViewerControl"
            IsToolbarVisible="True" />

        <StackLayout Grid.Row="1" Orientation="Horizontal">
            <Button x:Name="OpenButton"
                    Text="Open"
                    Clicked="OpenButton_Clicked" />
            <Button x:Name="ShareButton"
                    Text="Share"
                    Clicked="ShareButton_Clicked" />
        </StackLayout>
    </Grid>
</ContentPage>
```

For now, only focus on the XML namespace added to use the **`SfPdfViewer`** control. Specific considerations are coming in the next section, together with the C# code that enables the two buttons. As I mentioned previously, I added a sample PDF document in the companion solution, but if you are working on a brand new one, add a PDF file to the root folder of the .NET Standard project and set its **Build Action** as **EmbeddedResource**.

The last step for setup is specifying the Syncfusion license key, which you specify by adding the following line of code in the constructor of the **App.Xaml.cs** file, before the assignment of the **`MainPage`** object:

```
Syncfusion.Licensing.SyncfusionLicenseProvider.

            RegisterLicense("YOUR-LICENSE-KEY-GOES-HERE");
```

If you don't add the license key, libraries will work in trial mode (which is fine for the current example, but not for production purposes).

Introducing the SfPdfViewer control

Syncfusion's **`SfPdfViewer`** is a very powerful and versatile control that can display, annotate, and edit PDF documents. In Code Listing 22, it is declared as follows:

```
<syncfusion:SfPdfViewer x:Name="PdfViewerControl" IsToolbarVisible="True" />
```

The **IsToolbarVisible** property assigned with **true** enables a built-in toolbar that allows for performing many common operations on documents, such as navigating pages, searching for text, annotating text with highlights and comments, printing, and saving a copy. Because the **SfPdfViewer** exposes a lot of bindable properties, you can certainly create your own toolbar and bind your own commands to properties, but I can assure you that most of the time it's not needed. You load PDF documents into the **SfPdfViewer** via its **LoadDocument** method, which receives the file to open as a **Stream** object. You can also invoke the **LoadDocumentAsync** method for asynchronous loading. Both methods can optionally receive a **string** argument representing the document password, if the PDF is protected. With this in mind, the event handler for the **OpenButton** will look like the following:

```
private void OpenButton_Clicked(object sender, EventArgs e)

{

    var fileStream = typeof(App).GetTypeInfo().Assembly.

        GetManifestResourceStream("WorkingWithDocuments.SampleDoc.pdf");


    PdfViewerControl.LoadDocument(fileStream);

}
```

The first line of code reads a file from the embedded resources and returns a **Stream**, and the name has the following structure: *ProjectName.FileName.FileExtension*. If the file was placed in a project subfolder, the structure would be *ProjectName.FolderName.FileName.FileExtension*. In my case, the project is called **WorkingWithDocuments** (and so is the companion project). The second line loads the stream into the **SfPdfViewer** control. If you run the app as it is, you will see the result shown in Figure 30.

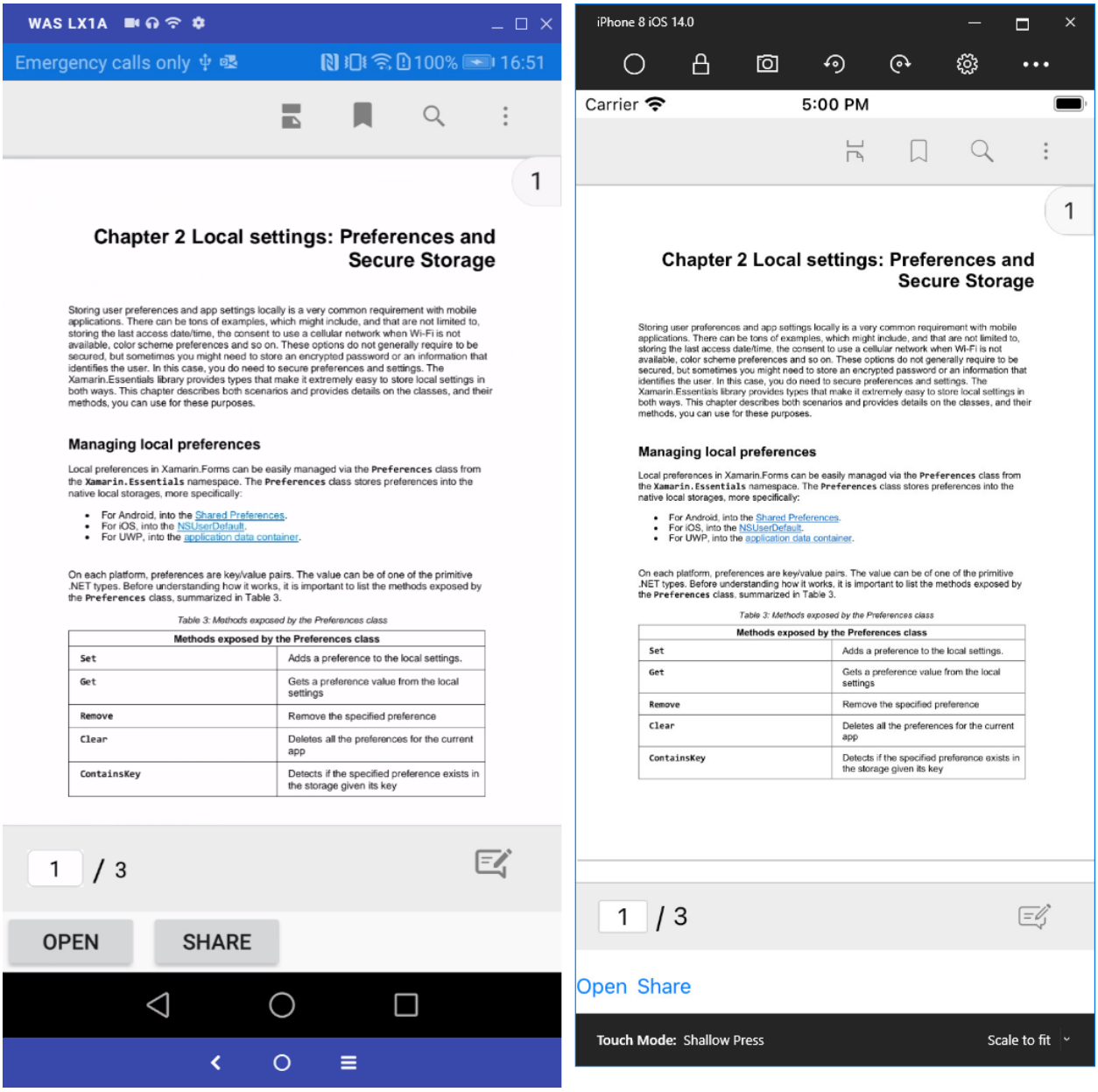

Figure 30: Displaying a PDF document in Xamarin.Forms

As you can see, the PDF document is shown in the user interface in a convenient way. The built-in toolbar is designed to be consistent with the operating system and offers everything you need to work with this kind of document. In particular:

- The navigation button (first from the left) allows for switching between single-page and multiple-page navigation.
- The Bookmarks button allows for managing bookmarks in the document.
- The Search tool makes it possible to search for specified text.
- Shortcuts for undoing and redoing actions, and saving and printing the document are accessible via the last button on the right.

- You can browse pages by writing the page number in the box at the bottom.
- You can add annotations and even digital signatures to the document by accessing the annotation tools with the button at the far right of the bottom bar.

Figure 31 shows an example of annotations added to the PDF document using only the built-in tools of the **`SfPdfViewer`** control.

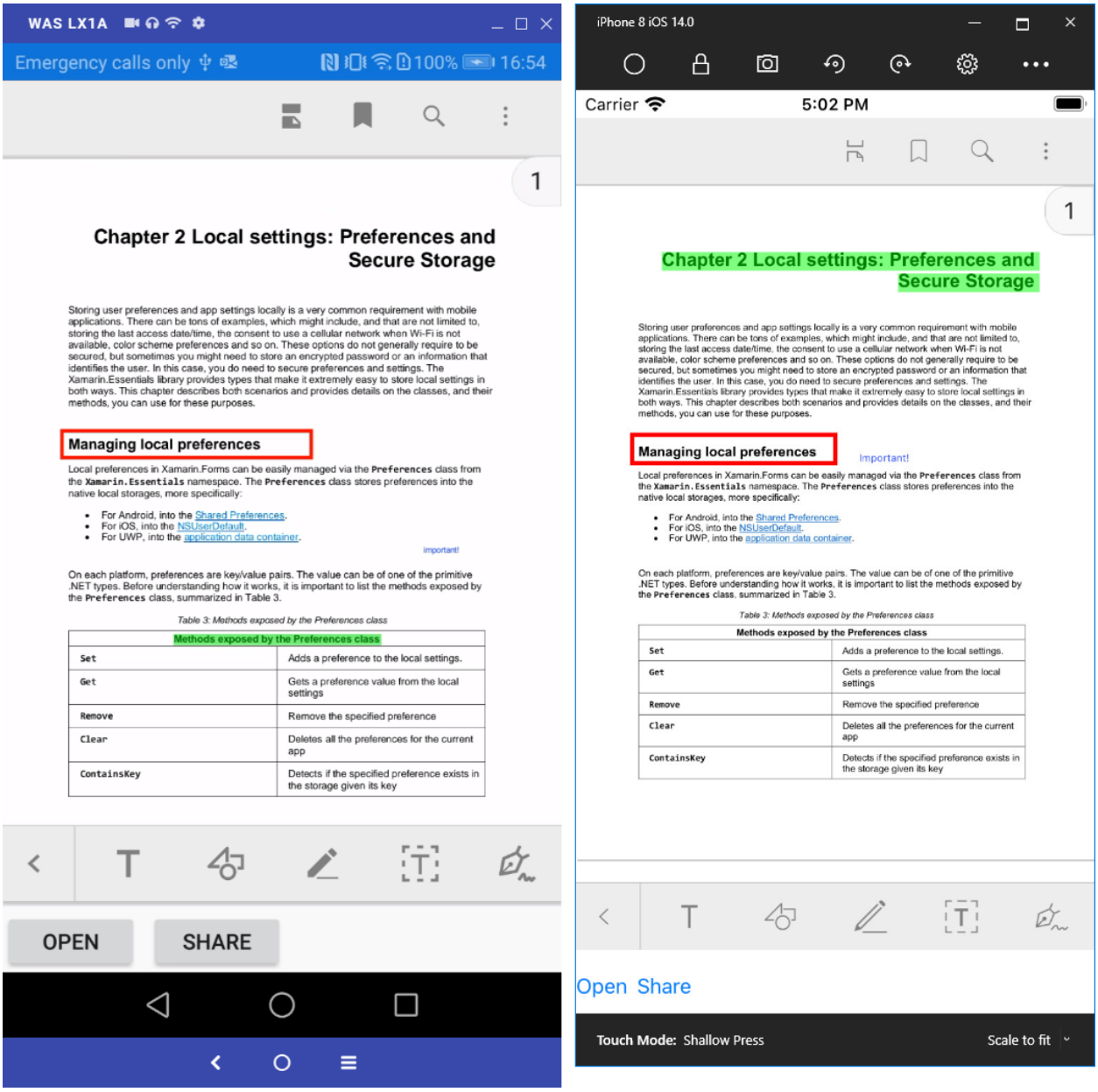

Figure 31: Annotating a PDF document with built-in tools

You can now probably imagine why I suggested you avoid too many customizations on the control. Its user interface is designed to be simple and easy to access, but powerful at the same time, and these features are what you would expect from a PDF viewer in a mobile app. Make sure you visit the official documentation for further information on the control's customization, localization, and other features.

Sharing files and documents

Sharing content in mobile apps is one of the most important features in terms of app usability, and goes beyond the concept of sharing on social media. For example, if you receive a PDF document via email and you want to open it inside a PDF viewer app like Adobe Reader, you are actually sharing the document from the mail client with the PDF viewer. Similarly, you can share files on your device, such as pictures and videos, in a WhatsApp chat. What you are doing is asking the operating system to open another app, which is open enough to receive contents from other apps. In a mobile app that works with data, a common scenario is to receive a document via web API, display the document in-app, and have the option to send it via email to some recipients. So, at a higher level, by sharing, we generally mean sharing content with another app.

Sharing content in Xamarin.Forms is made easy by the **`Share`** class exposed by the Xamarin.Essentials library. This class exposes a static method called **`RequestAsync`**, which receives an argument that can be of the following types:

- **`string`**, for sharing plain text.
- **`ShareTextRequest`**, for sharing URIs or text.
- **`ShareFileRequest`**, for sharing files.

Working with the first two is not covered in this chapter, but we will focus on the third one. The **`ShareFileRequest`** class points to the local pathname of a file. In the current example, the PDF document is stored as an embedded resource, so an additional step is required to get the file from the resources, copy it into the app's directory, and then point to the obtained pathname. This is accomplished by the following method, called **`ShareAsync`**:

```
private async Task ShareAsync()
{
    var fileStream = typeof(App).GetTypeInfo().Assembly.
        GetManifestResourceStream("WorkingWithDocuments.SampleDoc.pdf");

    var cacheFile = Path.Combine(FileSystem.CacheDirectory,
        "SampleDoc.pdf");

    using (var file = new FileStream(cacheFile, FileMode.Create,
                      FileAccess.Write))
    {
        fileStream.CopyTo(file);
    }
    var request = new ShareFileRequest();
```

```
    request.Title = "Share document";
    request.File = new ShareFile(cacheFile);
    await Share.RequestAsync(request);
}
```

As in the first example about loading the document, the first line gets the file stored inside the embedded resources as a **Stream**. The **cacheFile** variable combines the path for the app's cache directory with the file name into one pathname. The **using** block creates an instance of the **FileStream** class pointing to the original file's stream, creating a copy in the target directory. The **using** block ensures the stream resources are disposed after usage. The **ShareFileRequest** object can point to the local pathname via the **File** property. The **Title** property allows for specifying the title for the sharing user interface, if supported by the OS. The **ShareAsync** method is invoked by the **Clicked** event handler for the **ShareButton** as follows:

```
private async void ShareButton_Clicked(object sender, EventArgs e)
{
    await ShareAsync();
}
```

If you now run the app and click **Share** after opening the document, you will get a result similar to Figure 32.

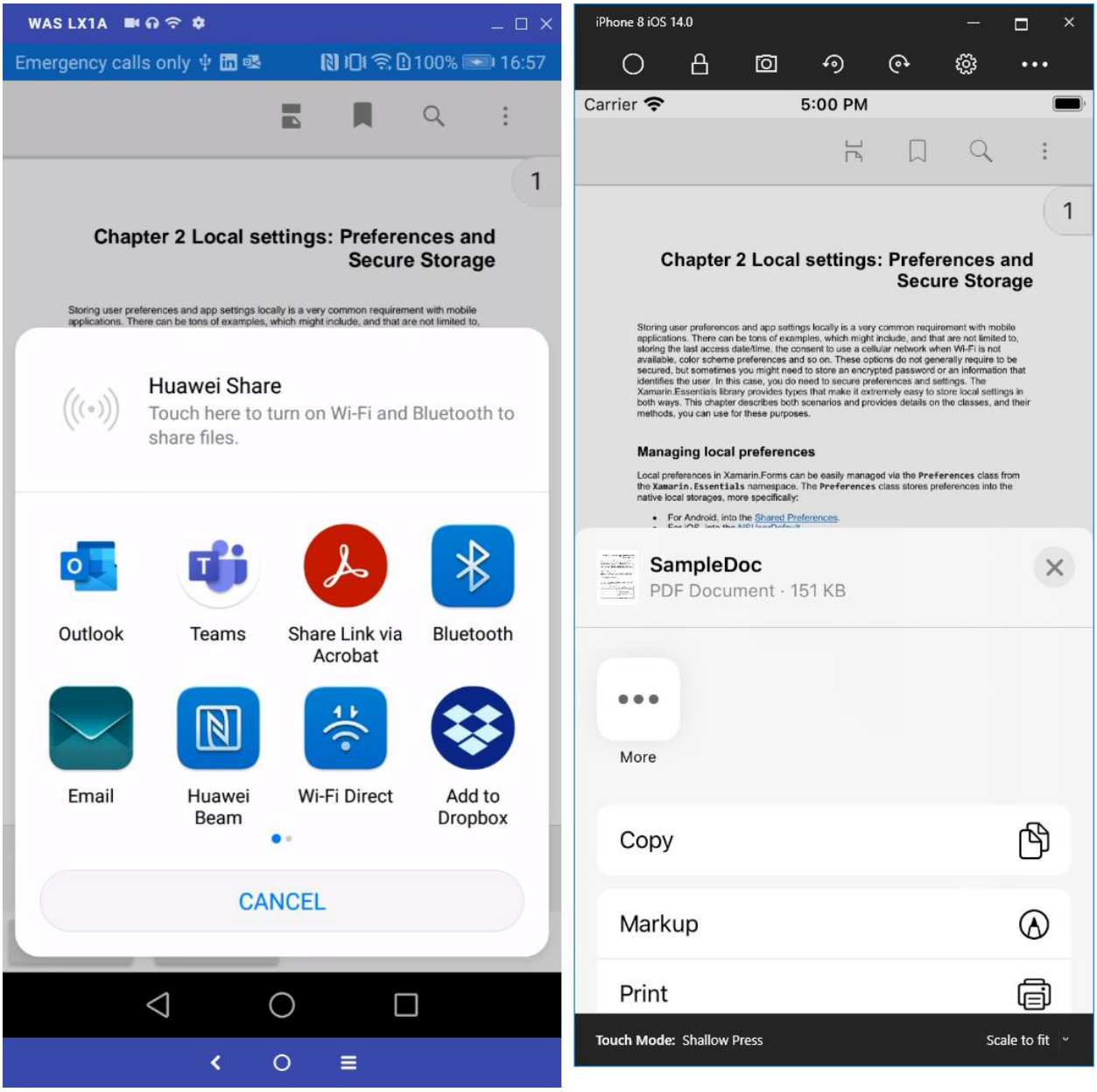

Figure 32: Sharing a document

The result will almost certainly vary on your device because it depends on how many and what apps you have installed to support PDF documents. (In my case, for iOS I'm using the simulator, which has no PDF apps installed, and therefore, it only shows device options.) You will just need to select an app enabled to receive PDF documents, and the file will be shared using the system interface, without the need to write platform-specific code—which is an incredible benefit.

The app that receives the document will then know how to handle it. For example, if you select an email client for sharing, the document will be automatically added as an attachment to a new email message. Actually, you could share multiple files at the same time with a similar approach, using the **`ShareMultipleFilesRequest`** class. This is not covered here, but the documentation provides an appropriate description.

Chapter summary

Mobile apps might need to work with PDF documents for several reasons, such as displaying reports, forms, or simply documentation. Xamarin.Forms does not have a built-in view to display PDF documents, but Syncfusion provides the **`SfPdfViewer`** control, which is simple to use and includes a toolbar with all the commands and tools required to annotate, comment, and sign documents.

When it comes to sharing, the **`Share`** class from the Xamarin.Essentials library makes it easy to send files to other apps by simply invoking the **`RequestAsync`** method. Usually an in-app PDF viewer takes all the space possible on a page, and actually this is something that many other views do. However, especially if you work with iPhone devices, there is something you need to know about handling space with iPhone X and later: the safe area. This is discussed in the next chapter.

Chapter 10 Managing the Screen on New iPhones

The most recent iPhone devices, such as the iPhone X, XS, XR, and 11, don't have the system button anymore, and provide a larger screen area. When you build apps for these devices, you must consider this larger screen area to ensure your contents properly fit into the page. The area of the screen where your contents should be placed is called the *safe area*, and this chapter explains simple ways to handle it.

Introducing the safe area

The most recent iPhones have larger screens and, consequently, a larger area that app developers can leverage to build even more beautiful user interfaces. The Apple layout guidelines divide the screen area on the new iPhones in two parts. The first part is the full-screen area as a whole. The second part is called the safe area, and it is made of a rectangle that represents the same screen area available on older iPhones. Figure 33, which is credited to Apple and taken from the Adaptivity and Layout documentation page, provides a clean representation of how the screen area is organized.

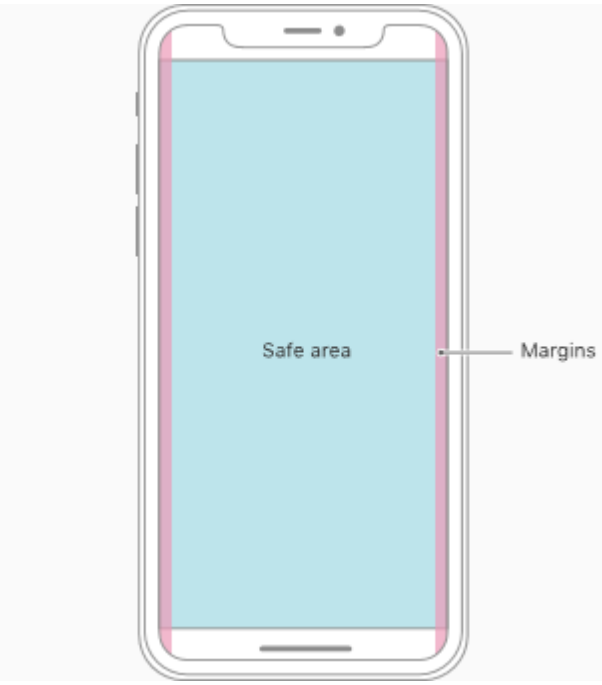

Figure 33: The safe area on iPhone devices. Source: Apple, Inc.

When you work with Xamarin.iOS and Xamarin.Forms, your app will automatically fill the entire screen area. However, there might be exceptions and reasons to avoid filling the entire screen and use only the safe area. For example:

- You want to make sure your app runs properly on older iPhones.
- You build for both Android and iOS, and you don't want to write additional platform-specific code for your UI.
- Your app adheres to a design that is thought to be cross-platform, or that is built upon older iPhone devices and cannot be changed.

By default your app will fill the entire screen when running on iOS, so if you want it to run only inside the safe area, you have to take responsibility to handle this scenario in code. Luckily, with Xamarin.Forms it's very quick and simple to make your apps use the safe area only. This is discussed in the next section.

Handling the safe area in Xamarin.Forms

In order to make your app use the safe area, the first thing you need to know is what device model the app is running on. This information is returned by the **`Model`** property of the **`Xamarin.Essentials.DeviceInfo`** class. Once you have this information, you can invoke the **`SetUseSafeArea`** extension method that is injected to the **`Page`** object by the **`Xamarin.Forms.PlatformConfiguration.iOSSpecifc`** namespace. Because these steps should be repeated for every page in your application, it's a good idea to create an extension method that does the job.

Create a new Xamarin.Forms blank project and add a new code file called **Extensions.cs**. Code Listing 23 contains the code for the extension method implementation, including the necessary **`using`** directives.

Code Listing 23

```
using Xamarin.Essentials;
using Xamarin.Forms;
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.iOSSpecific;

namespace SafeArea
{
    public static class Extensions
    {
        public static void SetIPhoneSafeArea(this ContentPage page)
        {
            string phoneModel = DeviceInfo.Model;

            //DeviceInfo.Model => Real device Model
            //iPhone7,1  => iPhone 6+
            //iPhone7,2  => iPhone 6
            //iPhone8,1  => iPhone 6S
            //iPhone8,2  => iPhone 6S+
```

```
            //iPhone8,4  => iPhone SE
            //iPhone9,1  => iPhone 7
            //iPhone9,2  => iPhone 7+
            //iPhone9,3  => iPhone 7
            //iPhone9,4  => iPhone 7+
            //iPhone10,1 => iPhone 8
            //iPhone10,2 => iPhone 8+
            //iPhone10,3 => iPhone X
            //iPhone11,2 => iPhone XS
            //iPhone11,4 => iPhone XS Max
            //iPhone11,8 => iPhone XR
            //iPhone12,1 => iPhone 11
            //iPhone12,3 => iPhone 11 Pro
            //iPhone12,5 => iPhone 11 Pro Max
            //iPhone12,8 => iPhone SE (2nd generation)

            if (phoneModel.ToLower().Contains("iphone11") ||
                phoneModel.ToLower() == "iphone10,3" ||
                phoneModel.ToLower().Contains("iphone12"))
                page.On<iOS>().SetUseSafeArea(true);
        }
    }
}
```

For your convenience, the comments in the code list the majority of the iPhone models so that you can decide which devices to include or exclude from using the safe area.

Tip: The `DeviceInfo.Model` property will return `x86_64` when the app is running inside the iOS Simulator.

In this particular case, the code simply applies to use the safe area on iPhone X and later devices. As you can see, the **`SetIPhoneSafeArea`** method is extending **`ContentPage`** objects. It's not possible to extend the **`Page`** class, which is an abstract class—you need to work on derived types. At this point, you only need to add the following line of code after the invocation to **`InitializeComponent`** in the constructor of your pages:

```
this.SetIPhoneSafeArea();
```

You do not need to check if the code is running on Android, because this will simply be ignored since the implementation is based on iOS-platform specifics. Figure 34, which is based on the iPhone 11 simulator, shows the difference between handling the safe area or not (in which case you will need to take care in padding and margins more precisely).

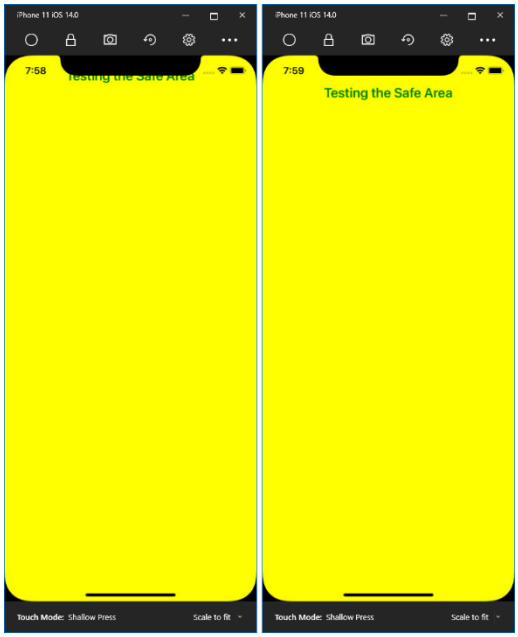

Figure 34: The app running inside the Safe Area

In my daily job, I had to implement this solution for a combination of all three reasons to avoid the full screen that I mentioned previously.

Chapter summary

The most recent iPhones have a larger screen area that is available by default to Xamarin projects. However, there might be reasons not to use it, and make the app run inside the safe area. If this is the case for you, the Xamarin.Forms code base has iOS-platform specifics that make this possible with one method invocation.

Conclusion

Xamarin.Forms offers everything you need to create powerful, native mobile applications. The purpose of this book is to provide guidance on how to take the most out of Xamarin.Forms to implement common requirements in modern mobile apps, from user experience requirements to runtime requirements.

In several cases, you have seen how to use the Xamarin.Essentials library to satisfy basic device requirements in a cross-platform approach, such as checking for network connection and battery level, without the need to write platform-specific code as you would have needed to do in the past. You have seen how to secure data locally and through communications over networks, and also how to secure access to your app via biometric authentication. You have seen how analytics can improve business decisions to drive investments based on the app usage, which is critical to stakeholders.

You have seen how to improve the look and feel of an app with Lottie animations and by handling the safe area on newer iPhone devices, and you have also seen how to implement a PDF viewer to improve the user productivity with documents. You also got some tips about legal notices that you need to provide, after getting advice from a lawyer specialized in IT, because this is more important than the entire app's content.

In your daily job as a mobile developer with Xamarin.Forms, there will certainly be more and different requirements and implementations, but for what is not included in this book, you can always bookmark the official documentation.

I have shared practical solutions that I have been implementing in the last five years, with continuous updates as the technology has evolved, and normally discussed with other very cool developers that share the office with me. Because sharing knowledge is the key to technical growth for every developer, I am happy to receive feedback on the book and about the techniques described here. If reading these chapters makes you think about more efficient solutions that you want to share, that would be beneficial to all of the developer community.