UNO PLATFORM

SUCCINCTLY

BY **ED FREITAS**

SUCCINCTLY EBOOK SERIES

Syncfusion

Uno Platform Succinctly

By
Ed Freitas

Foreword by Daniel Jebaraj

2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA

ISBN: 978-1-64200-215-7

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

Ed Freitas is a cloud software architect, business automation specialist, and customer success manager who loves writing about technology. He enjoys family time, learning, traveling, and outdoor sports.

You can reach him at https://edfreitas.me.

Acknowledgments

Thanks to everyone that contributed to this book. The fantastic Syncfusion team helped this book become a reality—especially Jacqueline Bieringer, Tres Watkins, and Graham High.

The manuscript manager and technical editor thoroughly reviewed the book's organization, code quality, and overall accuracy—Graham High from Syncfusion and James McCaffrey from Microsoft Research. Thank you.

I dedicate this book to *Chelin*, *Puntico*, *Lala*, *Mama*, and *Papa*—for everything you did, for everyone you loved.

Uno Platform

The Uno Platform is an open-source, cross-platform toolkit that allows developers to create applications for different systems and platforms, such as Windows, WebAssembly, iOS, macOS, Android, and Linux, using a codebase that is sharable between all the platforms.

Uno targets all operating systems and browsers by creating multiplatform applications using C# and XAML, allowing for code reusability and sharing business logic among different platforms.

Internally, the way the platform works is by acting as a UWP bridge that allows UWP-based applications, written with C# and XAML, to execute on different operating systems, independently of the type of device used.

The Uno Platform offers a straightforward way for developers with C# and XAML knowledge to leverage their existing skillset and write an application once—and deploy it on other platforms—with barely any adjustments or changes. If that sounds unique, it's because it is!

The Uno Platform works because UWP code can natively run in other platforms where it traditionally wouldn't be able to execute. By leveraging XAML, the Uno Platform can render an application's user interface on all platforms.

In other words, Uno takes the Universal Windows Platform and renders it on other platforms seamlessly.

Uno relies extensively on Microsoft's Xamarin technology that brings C# to different platforms and allows the same code to run everywhere.

Under the hood, UWP code runs natively on Windows; however, UWP executes on the Uno Platform, allowing it to run on iOS, Android, WebAssembly, and macOS by using Xamarin, .NET, and Mono, respectively.

The Uno Platform is an exciting technology because of the potential of code re-utilization that allows developers to target multiple platforms. It enables C# and UWP developers to leverage their existing skills to build applications that can run on various devices and operating systems—beyond Windows.

Without further ado, let's begin this journey to build single-source desktop, mobile, and WebAssembly applications using Uno.

Chapter 1 Setup and Basics

Overview

Throughout this chapter, we will explore how to set up the development environment to work with the Uno Platform.

You will need to install Visual Studio 2019. If you have a professional license, that's great. If you do not, then it is acceptable to use the Community edition, but make sure you install version 16.3 or later.

Uno applications will run natively on Windows; however, installing additional packages is required to run applications built with Uno on other platforms, which Visual Studio installs for you.

On my machine, I will be using Windows 10 and Visual Studio Community 2019 version 16.9.4.

Getting started

Let's set up the development environment. You are going to need Visual Studio 2019 version 16.3 or later if you would like to have a setup similar to mine.

The Uno documentation describes the prerequisites and the steps involved in setting up the development environment, depending on the editor used. For Visual Studio 2019 (which is what I've used), you can find the steps, requirements, and actions here.

There is also the possibility to use other editors or IDEs with the Uno Platform, such as Visual Studio Code or Visual Studio for Mac.

You are free to use the option you feel most comfortable with; however, I would recommend using a setup similar to mine to easily follow along with the examples covered throughout the book.

When I first tried Uno, I downloaded one of the example applications and opened the Visual Studio solution to run it. I found out that even though I had followed the steps to install all the prerequisites for using Visual Studio on Windows as my editor of choice, I was missing a few components required to run the solution on some platforms, such as Android. I was able to find this out because Visual Studio was smart enough to indicate that several components were missing, as seen in the following figure.

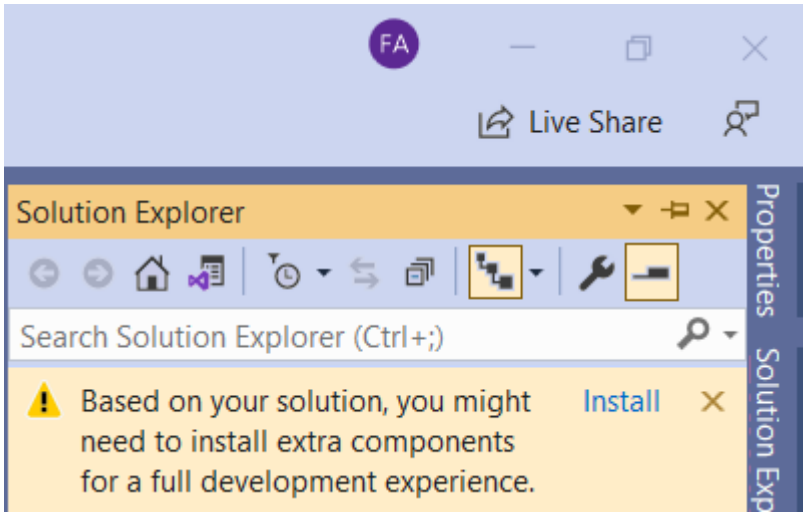

Figure 1-a: Visual Studio Solution Explorer – Missing Components Message

The solution to address these missing components is to install them, which you can do by clicking the **Install** option seen on the upper-right corner of the message.

In my case, after I did that, the Visual Studio 2019 installer popped up on the screen, indicating that the following components were missing.

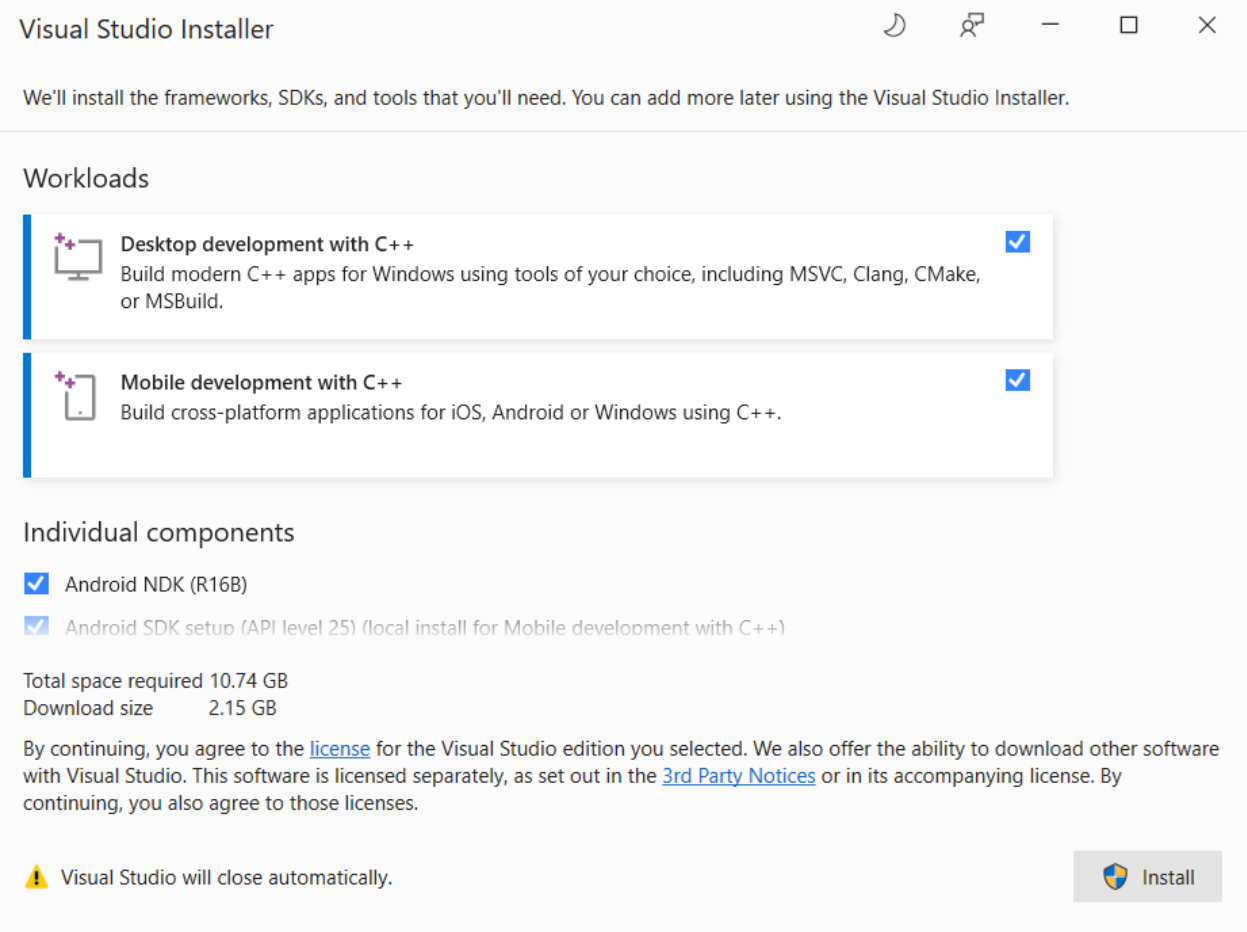

Figure 1-b: Visual Studio Installer, Missing Components

I clicked the **Install** button to resolve this and install all the remaining components that the example application required.

After finalizing the installation process, I downloaded and installed the suggested emulator from the list shown in the **Android Device Manager**.

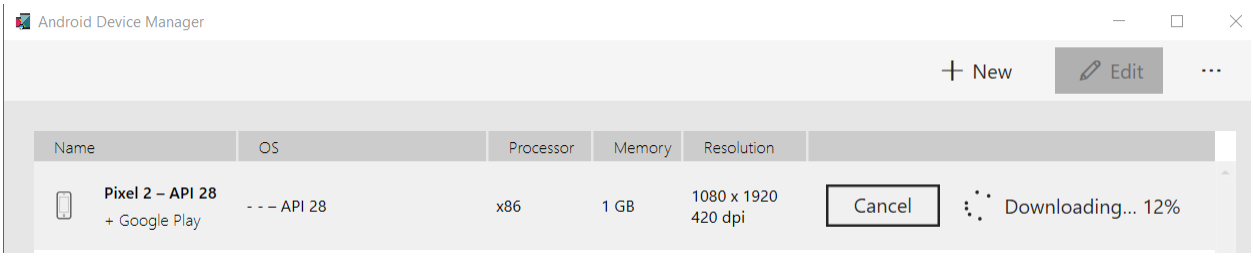

Figure 1-c: Android Device Manager – Emulator Installation

Once the emulator was installed, I was able to run the downloaded example application from the Uno website successfully.

Something important to mention is that the installation experience you might have could be different from what I went through. Depending on which editor you use or what example Uno application you download and run, your installation experience could vary from what I have described. The key is to make sure you follow all the steps for the editor or IDE you choose to use. In the case of Visual Studio 2019 on Windows, follow these instructions.

With Visual Studio Code, use these instructions, and with Visual Studio for Mac, use these instructions.

Uno Platform extension

Once all the required components are installed, the next step is to install the Uno Platform extension for Visual Studio (also known as Uno Platform Solution Templates). This extension will allow you to create an Uno project seamlessly.

There are two ways to install the extension. One way is to download it and execute the extension installer.

Another way is to install it from Visual Studio 2019 directly—which is my preferred way. All you need to do is launch Visual Studio 2019 and click **Continue without code**.

Once Visual Studio 2019 is open, nagivate to **Extensions** > **Manage Extensions** from the menu bar.

You will see the **Manage Extensions** window appear. Search for **uno** and select the **Uno Platform Solution Templates** option.

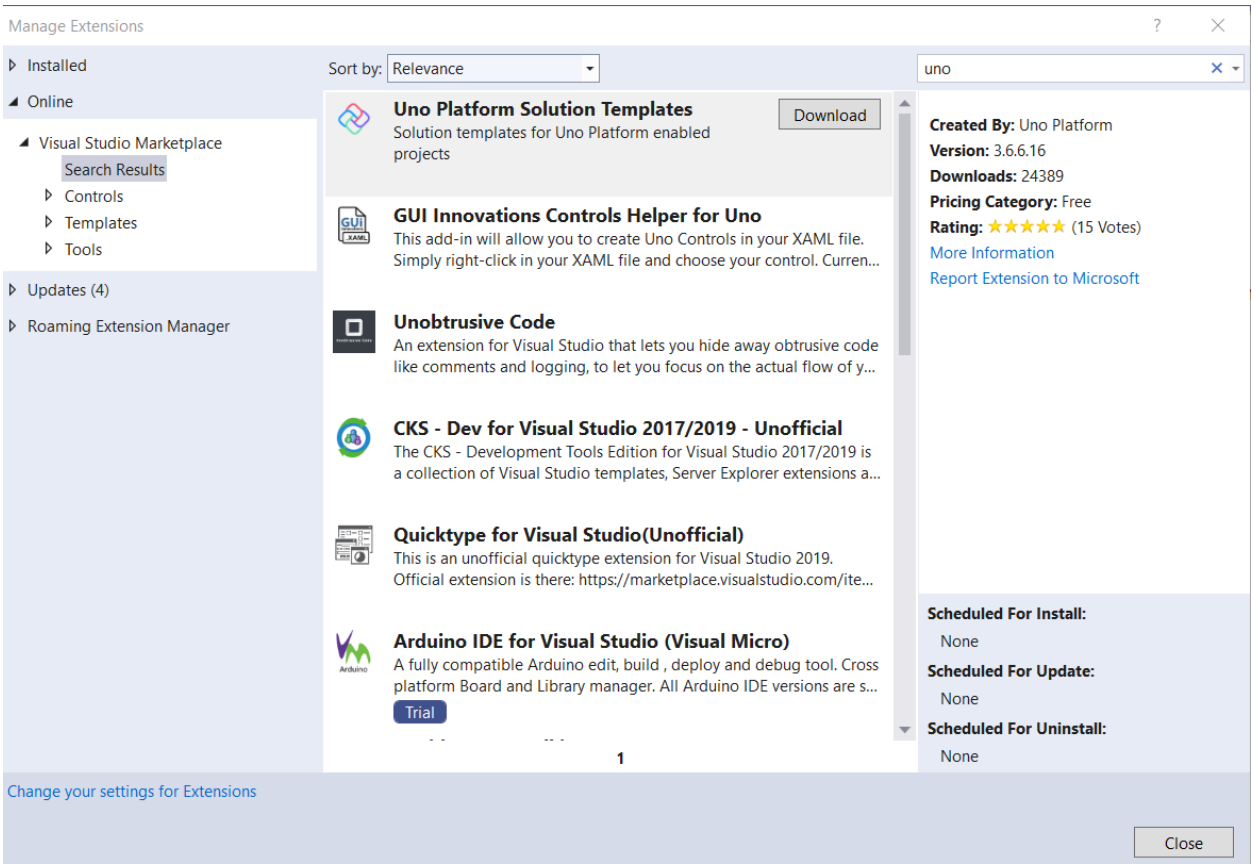

Figure 1-d: Manage Extensions Window – Uno Platform Solution Templates

This process will run the extension installer.

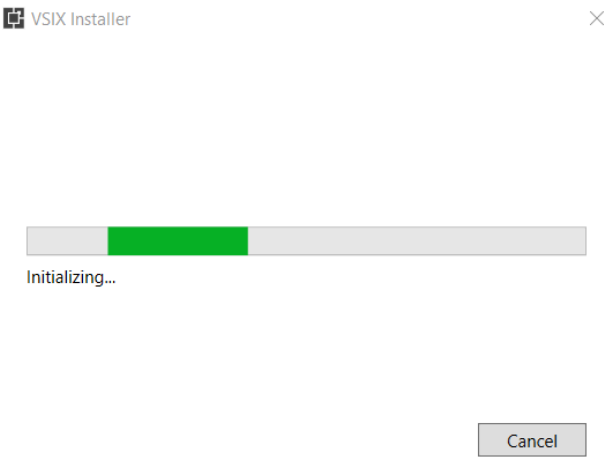

Figure 1-e: Installing the Uno Platform Solution Templates

Once the installation process has ended, we can restart Visual Studio 2019 and create a new Uno Visual Studio solution using a template.

Creating an Uno solution

After Visual Studio 2019 has restarted, you will see the **Create a new project** window. Search for **uno** and select the **Cross-Platform App (Uno Platform)** option. Then, click **Next**.

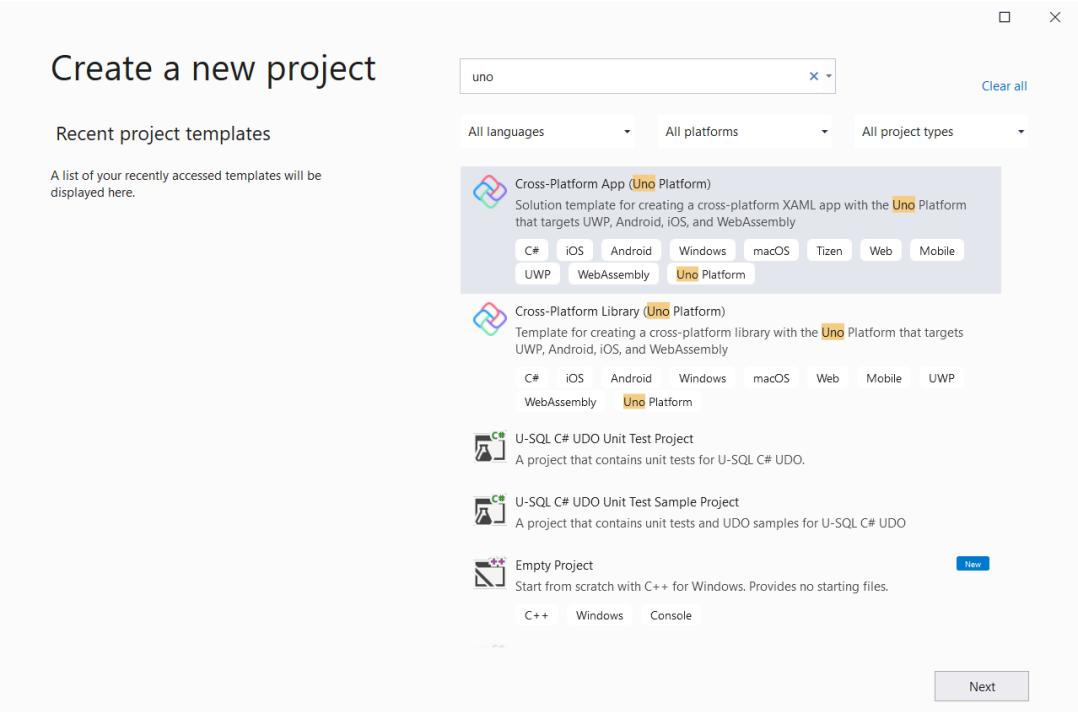

Figure 1-f: Creating a New Uno Solution with Visual Studio 2019

After doing that, you will have to indicate the project name and the location.

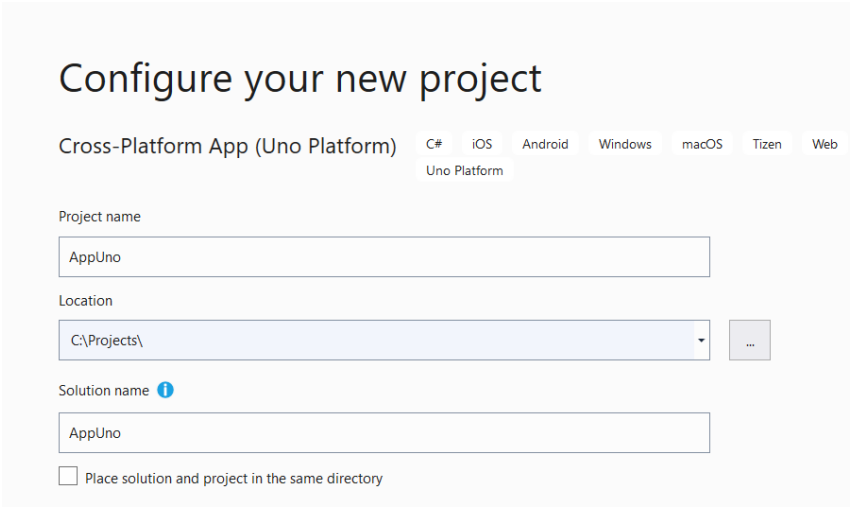

Figure 1-g: Choosing the Uno Solution Name and Location

Click **Next** (not visible in the preceding screenshot) to continue, which will take you to the following screen.

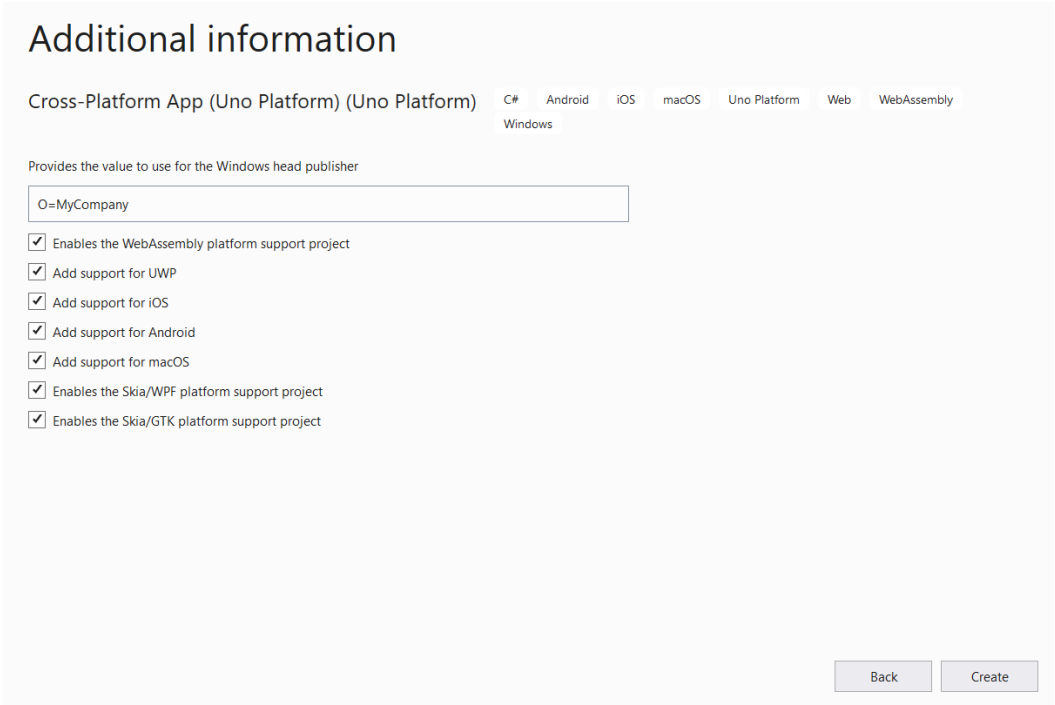

Figure 1-h: Uno Solution – Additional Information

In this step, you can choose which platforms you want your Uno project to support. You can clear any of the options that you do not want your project to target.

Next, click **Create**. This will create the Uno Visual Studio 2019 solution, containing various projects, each targeting a different platform.

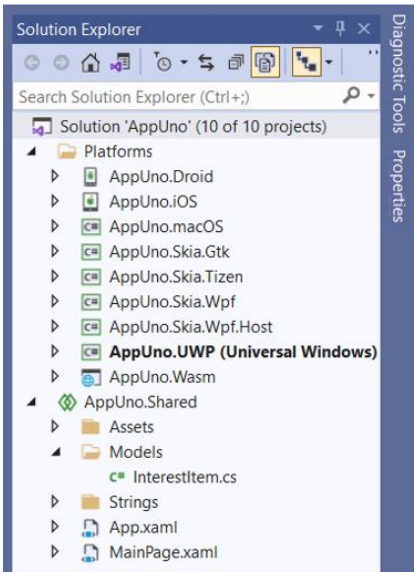

Figure 1-i: Uno Solution with Various Projects (Each Targeting a Different Platform)

In addition to having a different project for each platform, a shared project is also part of the solution. This shared project is where the code will reside, shared among the other projects within the solution.

The shared project is where you will spend most of your time working with an Uno solution, and where you will aim to write most of your logic.

Running the app

Now that we have the project ready, let's explore some of the basics of how the Uno solution works. Using the **Solution Explorer** within Visual Studio 2019, go to the shared project—which in my case is called **AppUno.Shared**—and open the **MainPage.xaml** file.

Listing 1-a: The MainPage.xaml File – AppUno.Shared Project

```
<Page
    x:Class="AppUno.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:AppUno"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">

    <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
        <TextBlock Text="Hello, world!" Margin="20" FontSize="30" />
    </Grid>
</Page>
```

As you can see, we have this XAML file that declares a **`Page`** that contains a **`Grid`**, and that includes a **`TextBlock`**, which renders a **`Hello, world!`** message on the screen when the application runs.

To see this in action, we need first to set the **AppUno.UWP (Universal Windows)** project as the solution's startup project, which you can do by right-clicking it within the **Solution Explorer** and choosing the **Set as Startup Project** option.

Next, click the **Run** button in Visual Studio 2019 to execute the UWP application.

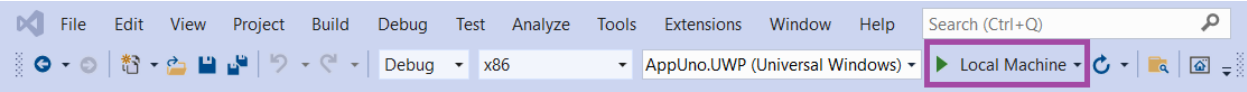

Figure 1-j: The Run Button in Visual Studio 2019 (for UWP Apps)

You will then see the UWP version of the application running as a native Windows app, as follows.

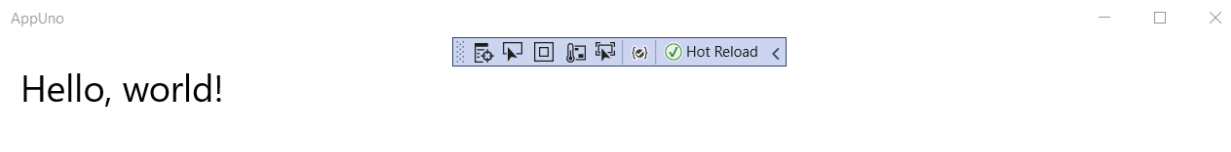

Figure 1-k: The UWP App Running

Next, let's run the application in the browser. We can do this by setting **AppUno.Wasm** as the startup project by right-clicking it within the **Solution Explorer** and choosing **Set as Startup Project**. Then, click the **Run** button in Visual Studio 2019 to execute the WebAssembly application.

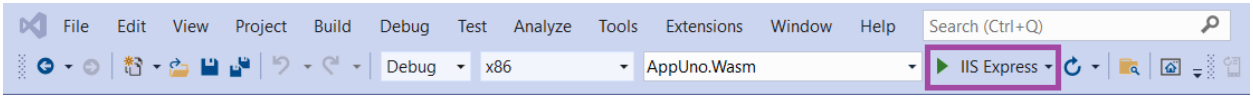

Figure 1-l: The Run Button in Visual Studio 2019 (For WebAssembly Apps)

The compilation process takes a bit longer than when building the UWP version of the application.

After the building process is finished, you will see the WebAssembly version of the browser's application running in the browser, as follows.

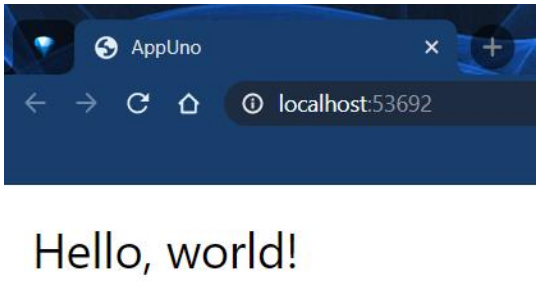

Figure 1-m: The WebAssembly App Running

Next, let's try to run the Android version of the application. We can do this by setting **AppUno.Droid** as the startup project by right-clicking it within the **Solution Explorer** and choosing the **Set as Startup Project** option.

However, before clicking on the Run button, we will need to execute the Android emulator. To do that, go to **Tools** > **Android** > **Android Device Manager**.

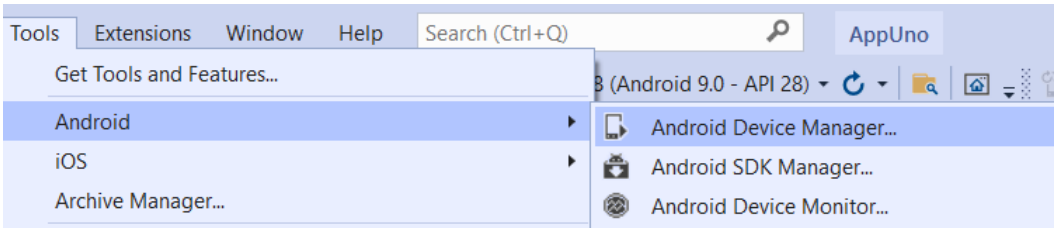

Figure 1-n: How to Open the Android Device Manager within Visual Studio 2019

The Android Device Manager will then open, and you'll be able to use any Android emulators available or create new ones. On my machine, I have the following one.

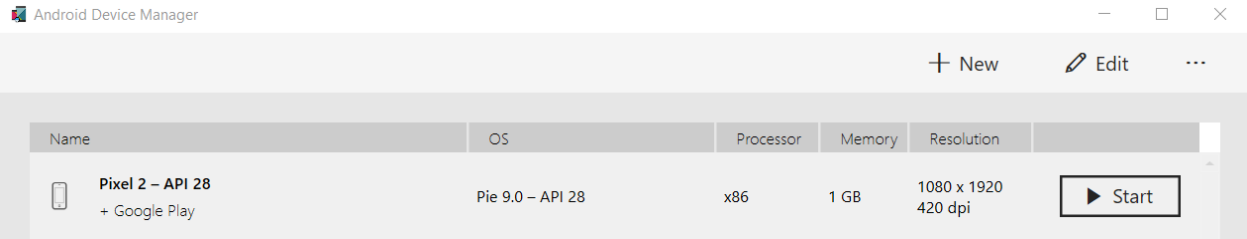

Figure 1-o: Android Device Manager (Launched from Visual Studio 2019)

To launch the Android emulator (if you have one), click **Start**. If you don't have an emulator created, you can create a new one by clicking **New**.

We can also change the settings of the existing emulator by clicking **Edit**. In my case, I've started the emulator I have, which looks as follows.

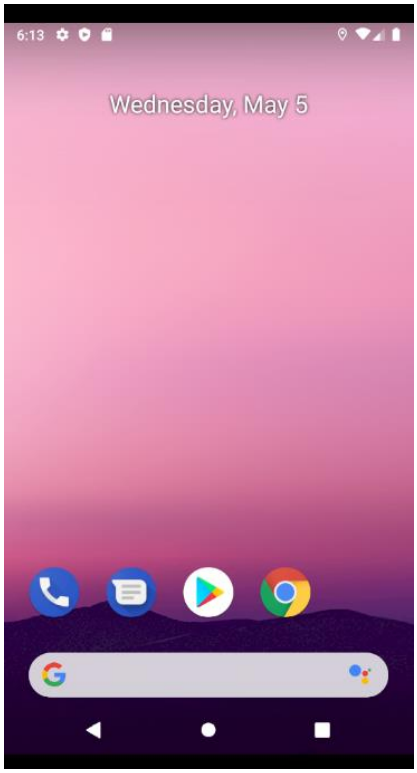

Figure 1-p: Android Emulator Running

Once the emulator is running, we can click the Run button in Visual Studio 2019 to execute the **AppUno.Droid** project. This will deploy the application to the emulator and run it.

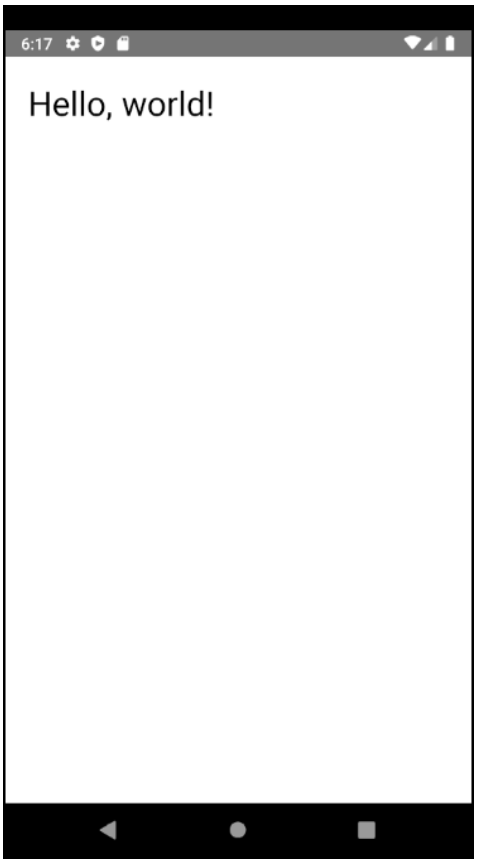

Figure 1-q: The Android App Running

Excellent—we can see the application is running on Android. It's pretty amazing that with the same codebase, we've been able to run the application as a native Windows (UWP) application, as a web application using WebAssembly, and as a mobile app running on Android.

I'm not aware of any other platform that can offer such a wide range of cross-platform support using C# and XAML at the time of writing this book.

Summary

Throughout this chapter, we've explored how to get started with the Uno Platform by explicitly looking at the setup, creating a project from scratch, and running an application targeting different platforms.

Next, we will explore how the solution is structured and then build upon that to learn even more about Uno.

Chapter 2 Solution Structure

Overview

Now that we have our Uno environment ready, let's focus on how the solution is structured, which this chapter is all about. Let's dive right in.

Solution structure

The Uno solution we created in the previous chapter consists of various projects targeting a different platform, except for the **AppUno.Shared** project, which is where most of the code resides.

The projects (besides the **AppUno.Shared** project) that are part of the solution are:

- **AppUno.Droid**: This project targets Android.
- **AppUno.iOS**: This project targets iOS.
- **AppUno.macOS**: This project targets macOS.
- **AppUno.Skia.Gtk**: This project targets the GTK widget toolkit for Linux using the Skia graphics engine.
- **AppUno.Skia.Tizen**: This project targets the Tizen mobile-based Linux operating system using the Skia graphics engine.
- **AppUno.Skia.Wpf** and **AppUno.Skia.Wpf.Host**: These projects target the Windows Presentation Foundation graphical subsystem using the Skia graphics engine.
- **AppUno.UWP (Universal Windows)**: This project targets the Universal Windows Platform on Windows 10.
- **AppUno.Wasm**: This project targets the WebAssembly runtime using a browser.

Let's explore each of these projects individually.

AppUno.Droid

The **AppUno.Droid** project is the boilerplate that Uno uses for building the Android application. It has the following structure.

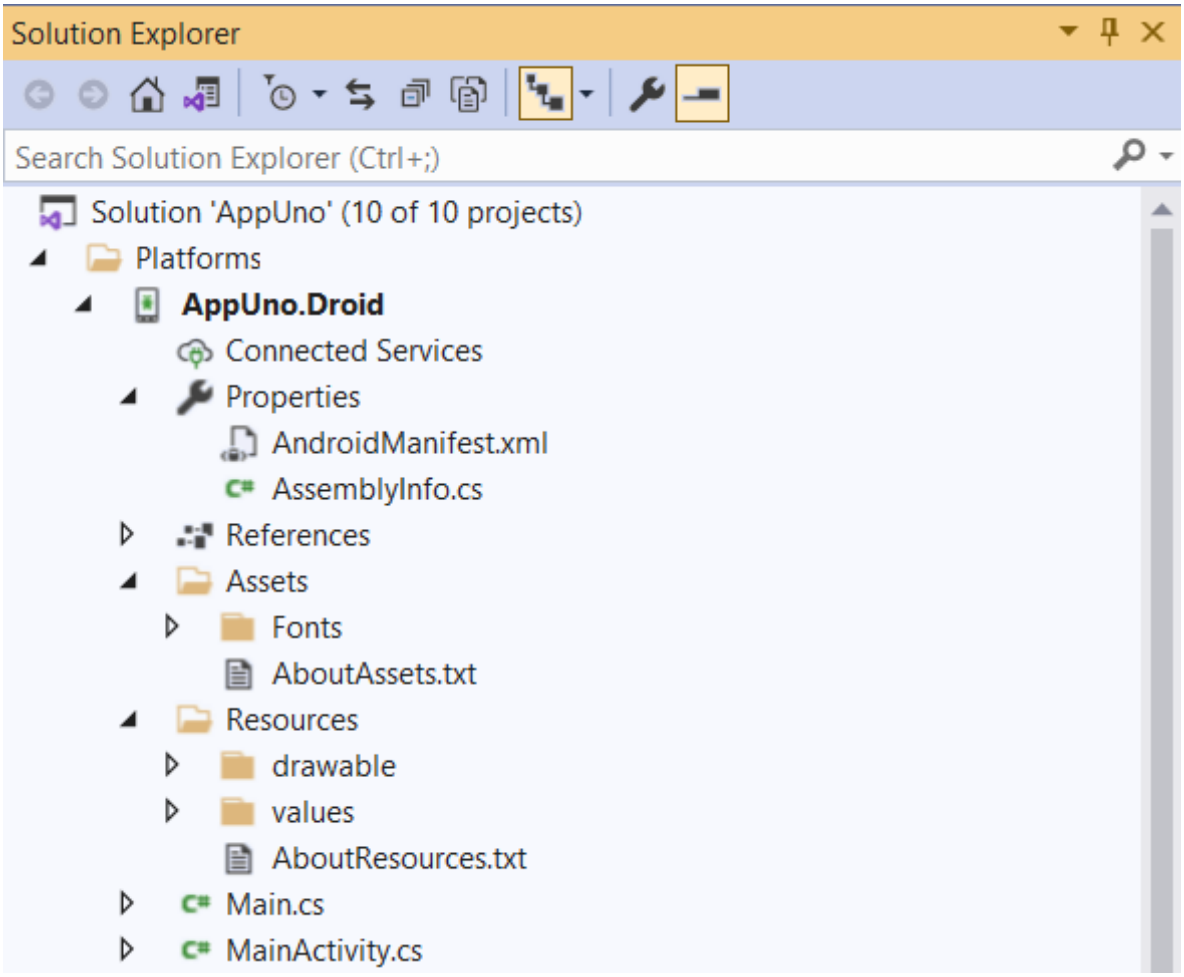

Figure 2-a: The AppUno.Droid Project Structure

First, we find the project's **Properties** folder containing the Android application manifest (**AndroidManifest.xml**) file that includes information of the package, including application components such as activities, services, receivers, and content providers.

The **AssemblyInfo.cs** file has information about the assembly, such as the name of the file, description, and version. If the **AssemblyInfo.cs** file gets removed after compilation, it will contain no information. This means that if you inspect the Details tab of the file properties, you will see no name, no description, and no version.

Then we find the project **References**, the libraries, and the NuGet packages that the project relies on, which are shown in the following figure. **AppUno.Shared** is the project's first reference.

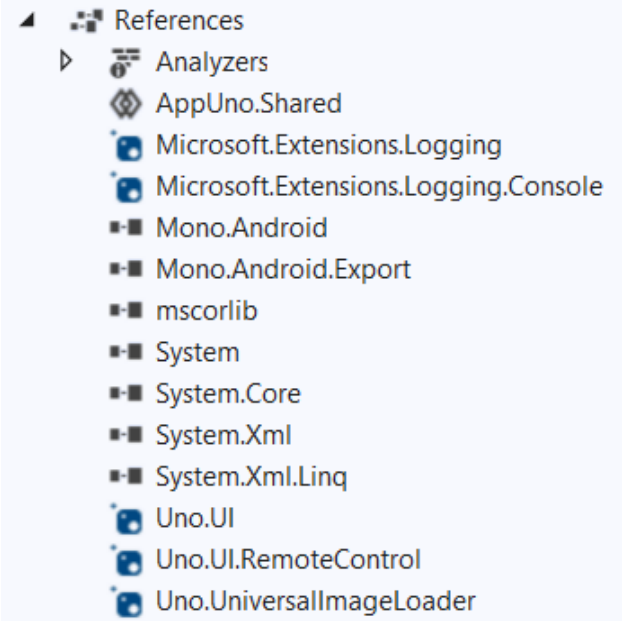

Figure 2-b: The AppUno.Droid Project References

Notice the **Mono.Android** and **Mono.Android.Export** libraries (part of the Mono framework). These are required to run .NET code on Android.

Beyond that, notice the references to the standard .NET libraries such as **mscorlib**, **System**, **System.Core**, **System.Xml**, and **System.Xml.Linq**.

We can see the references to the **Uno.UI** library—which is the core Uno module, **Uno.UI.RemoteControl**—used for XAML hot reload and **Uno.UniversalImageLoader**, which is a Xamarin.Android binding library for **UniversalImageLoader**.

Then we find the **Assets** and **Resources** folders. The **Assets** folder is used for any raw assets you want to deploy with your application.

On the other hand, the **Resources** folder includes images, layout descriptions, binary blobs, and string dictionaries that your application will use.

Next, we find the **Main.cs** and **MainActivity.cs** files. The **Main.cs** file is the **AppUno.Droid** main file, which is the entry point for executing the Android application, and **MainActivity.cs** corresponds to the Android App's main activity.

AppUno.iOS

The **AppUno.iOS** project is the boilerplate that Uno uses for building the iOS application. The files contained within this project are specific to creating an iOS runtime. Let's explore these files.

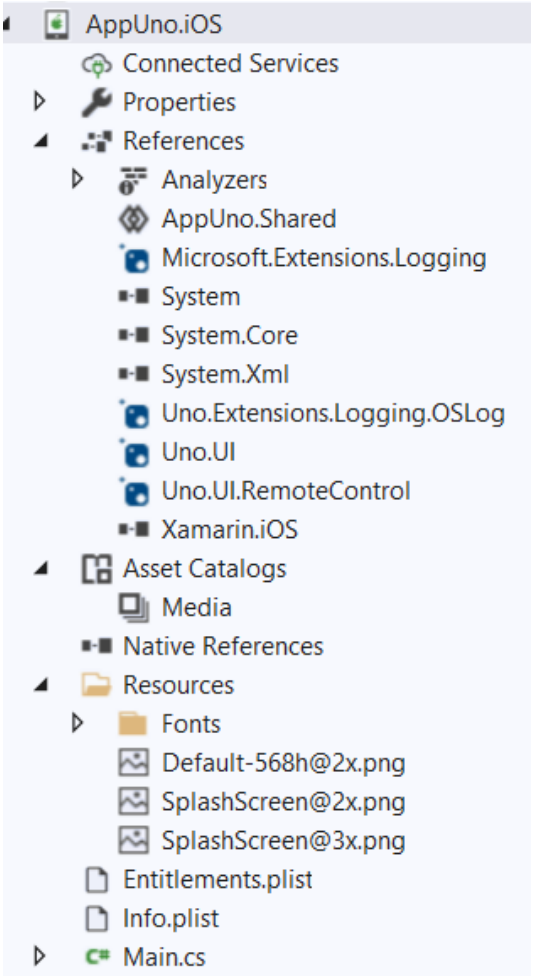

Figure 2-c: The AppUno.iOS Project Structure

As we can see, the project **References** are similar to those used by the **AppUno.Droid** project, and **AppUno.Shared** is also the first reference of the project.

There are also references to the **Uno.UI** and **Uno.UI.RemoteControl** libraries, as well as .NET libraries like **System** and **System.Core**. There's nothing odd about that. However, the difference is that this project uses the **Xamarin.iOS** library, which is Xamarin framework implementation for iOS.

Within the project structure, we find the **Asset Catalogs**, including any iOS application assets.

The **Resources** folder includes the **Fonts** folder that contains the fonts and splash screens used by the application.

The **Entitlements.plist** ("properties list") file defines the capabilities of the iOS application, and the **Info.plist** file is for storing configuration data in a place where the system can easily access it.

Finally, we have the **Main.cs** file that is the application's main entry point.

AppUno.macOS

The **AppUno.macOS** project is the boilerplate that Uno uses for building the macOS application.

The files contained within this project are specific to creating a macOS runtime. Let's explore the file structure of this project.

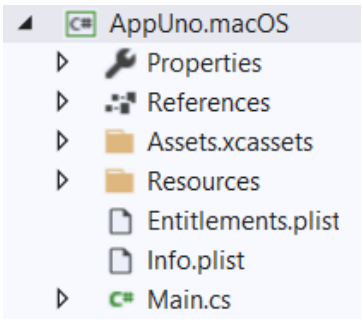

Figure 2-d: The AppUno.macOS Project Structure

Like with the **AppUno.iOS** project, we have a folder dedicated to application assets called **Assets.xcassets**.

There's also a **Resource** folder, which contains a **Fonts** folder that includes the fonts that the application will use.

Like with the **AppUno.iOS** project, we can find the **Entitlements.plist**, **Info.plist**, and **Main.cs** files—these have the same functionalities.

The libraries included in the **References,** though, are slightly different. Let's have a look at them.

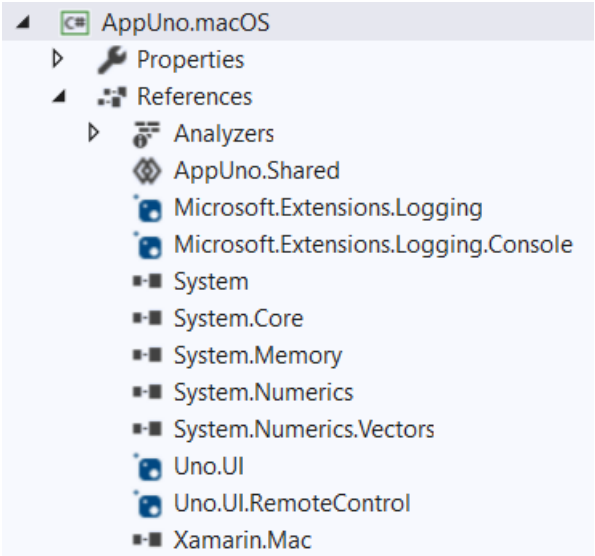

Figure 2-e: The AppUno.macOS Project References

There are references to the **AppUno.Shared** project and **Uno.UI**, **Uno.UI.RemoteControl**, and .NET libraries, such as **System** and **System.Core**.

The difference is that the **Xamarin.Mac** library is used. This is the Xamarin framework implementation for targeting macOS.

AppUno.Skia.Gtk

The **AppUno.Skia.Gtk** project is the boilerplate that Uno uses for building the application that targets GTK using the Skia graphics engine. The structure of this project is as follows.

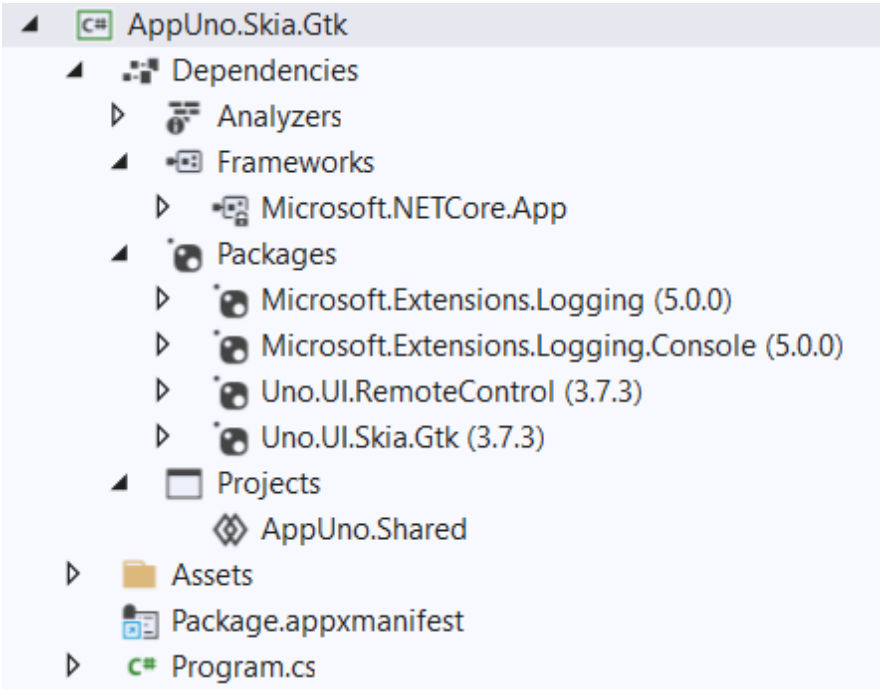

Figure 2-f: The AppUno.Skia.Gtk Project Structure

As for the Uno dependencies, we have only the **Uno.UI.Skia.Gtk** and **Uno.UI.RemoteControl** libraries.

The **AppUno.Skia.Gtk** project also references the **AppUno.Shared** project, and there is also an **Assets** folder.

The **Package.appxmanifest** file is nothing more than a glorified XML settings file containing information about the application, such as the **Name**, **Publisher**, **Version**, and other details.

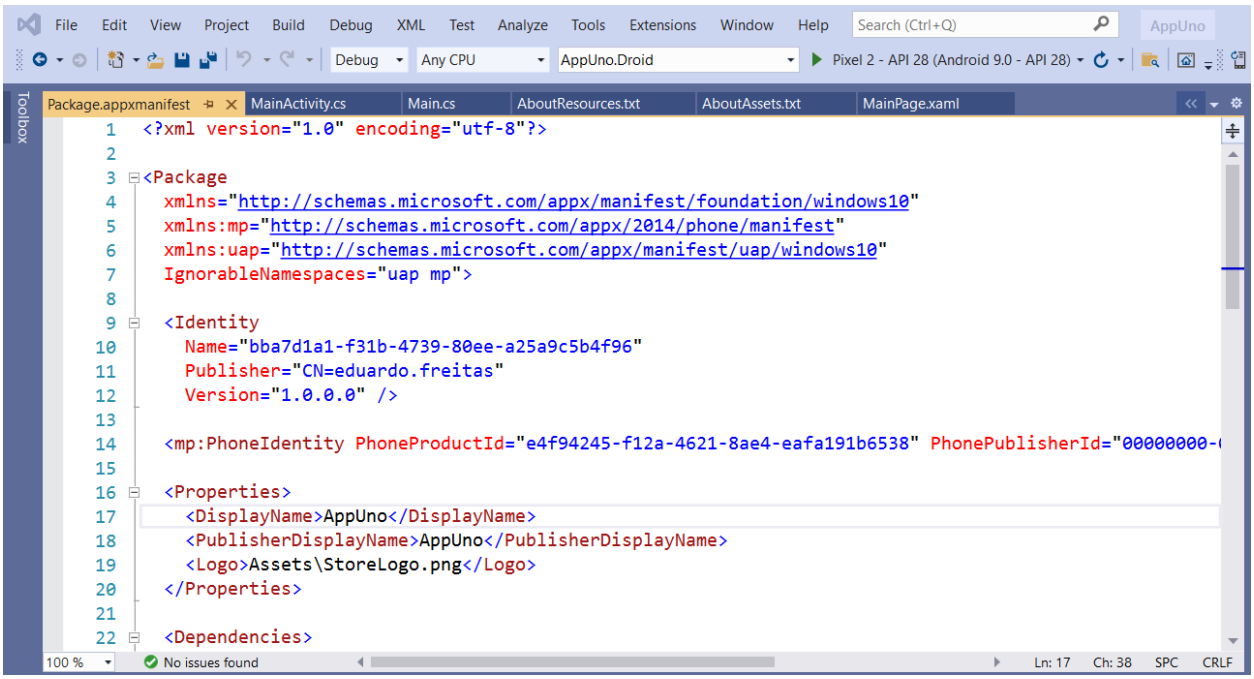

Figure 2-g: The Package.appxmanifest File

For the **AppUno.Skia.Gtk** project, the application's main entry point is the **Program.cs** file instead of **Main.cs**.

AppUno.Skia.Tizen

The **AppUno.Skia.Tizen** project is the boilerplate Uno uses for building the application that targets Tizen using the Skia graphics engine. The structure of this project is as follows.

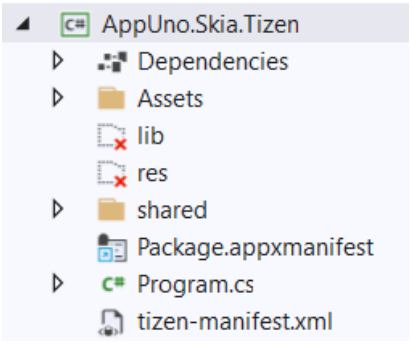

Figure 2-h: The AppUno.Skia.Tizen Project Structure

We can see that we have the project **Dependencies**, which we will look at shortly, an **Assets** folder, and the **Package.appxmanifest** and **Program.cs** files.

A noticeable difference is a **tizen-manifest.xml** file. This file contains the manifest details that Tizen will use to identify the application. The file contains the following information.

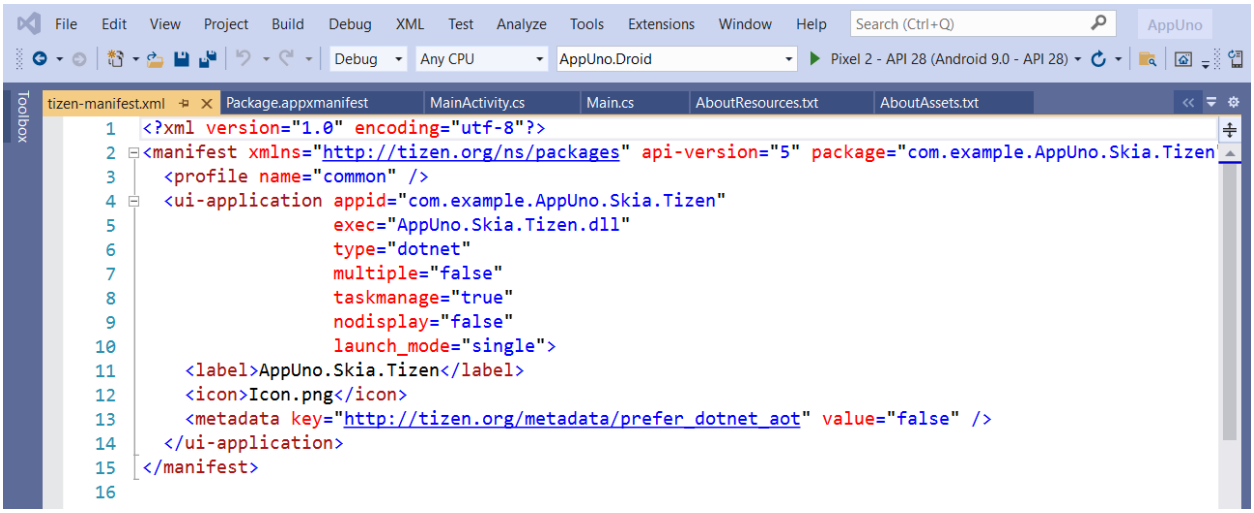

Figure 2-i: The tizen-manifest.xml File

As for the project **Dependencies**, we can see the **SkiaSharp.Views** library, which includes platform-specific views and controls for drawing on the screen.

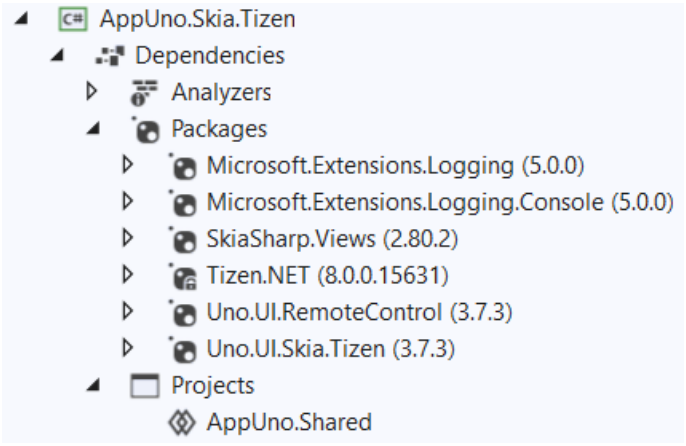

Figure 2-j: The AppUno.Skia.Tizen Project Dependencies

There is also a reference to the **Tizen.NET** library within the **Dependencies**, which allows .NET code to run on Tizen.

There are also references to the **Uno.UI.Skia.Tizen** and **Uno.UI.RemoteControl** libraries, and finally, to the **AppUno.Shared** project.

AppUno.Skia.Wpf and AppUno.Skia.Wpf.Host

The **AppUno.Skia.Wpf** and **AppUno.Skia.Wpf.Host** projects are related to each other, and these execute a Windows Presentation Foundation (WPF) application using Uno. We can see their structure in the following figure.

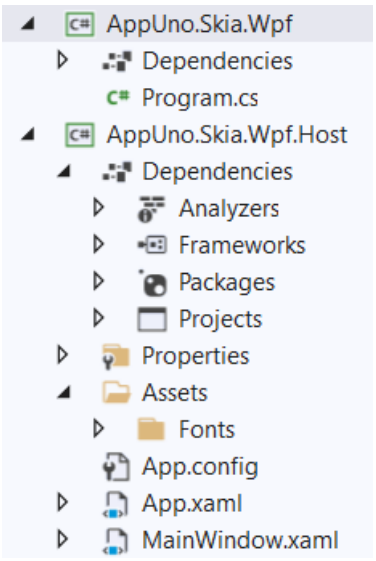

Figure 2-k: The AppUno.Skia.Wpf and AppUno.Skia.Wpf.Host Projects

The **AppUno.Skia.Wpf** project has a straightforward structure. As you can see, it only contains **Dependencies** and the **Program.cs** file, which is the main entry point of the WPF application.

As for the **Dependencies** of the project, we find the following.

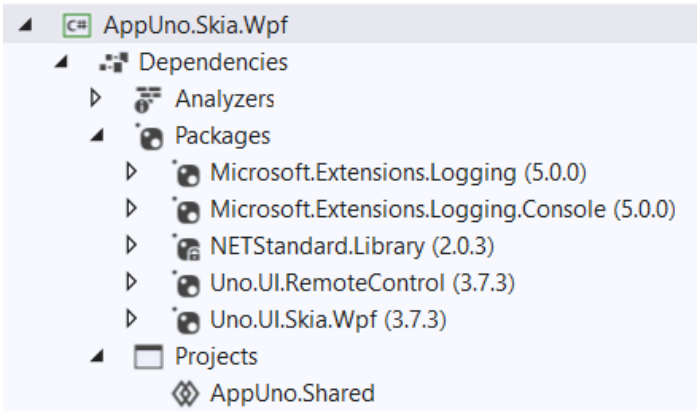

Figure 2-l: The AppUno.Skia.Wpf Project Dependencies

We can see that the **AppUno.Skia.Wpf** project contains references to the **Uno.UI.Skia.Wpf** and **Uno.UI.RemoteControl** libraries. Uno uses the **Uno.UI.Skia.Wpf** library to execute the WPF application and render it using the Skia graphics engine.

Besides that, the .NET Standard library (**NETStandard.Library**) is referenced, and so are the Microsoft logging extension libraries. As with the rest of the projects of the solution, the shared project—**AppUno.Shared**—is referenced as well.

Now, let's inspect the structure of the **AppUno.Skia.Wpf.Host** project.

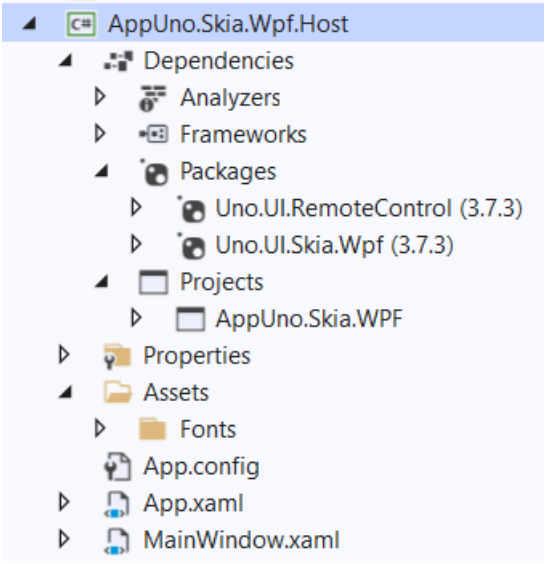

Figure 2-m: The AppUno.Skia.Wpf.Host Project Structure

As for the project **Dependencies**, we find the **Uno.UI.Skia.Wpf** and **Uno.UI.RemoteControl** libraries.

The other references are not needed, given that they are part of the **AppUno.Skia.Wpf** project; this is because the **AppUno.Skia.Wpf** project is included as a reference of **AppUno.Skia.Wpf.Host**.

The references to the **Uno.UI.Skia.Wpf** and **Uno.UI.RemoteControl** libraries are required because the **App.xaml** and **MainWindow.xaml** files are part of the **AppUno.Skia.Wpf.Host** project.

The **Properties** and **Assets** folders contain resources and fonts, respectively, used by the project.

AppUno.UWP (Universal Windows Platform)

Uno uses the **AppUno.UWP** project to create a Universal Windows Platform application. Let's explore its structure.

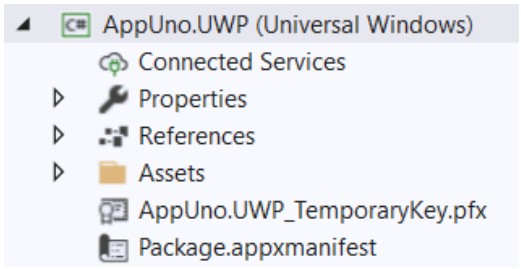

Figure 2-n: The AppUno.UWP Project

We can see that this project has **Properties**, **References**, and **Assets** folders, a certificate file called **AppUno.UWP_TemporaryKey.pfx**, and a **Package.appxmanifest** file. The certificate is for digitally signing the application.

Let's have a look at the **AppUno.UWP** project **References**.

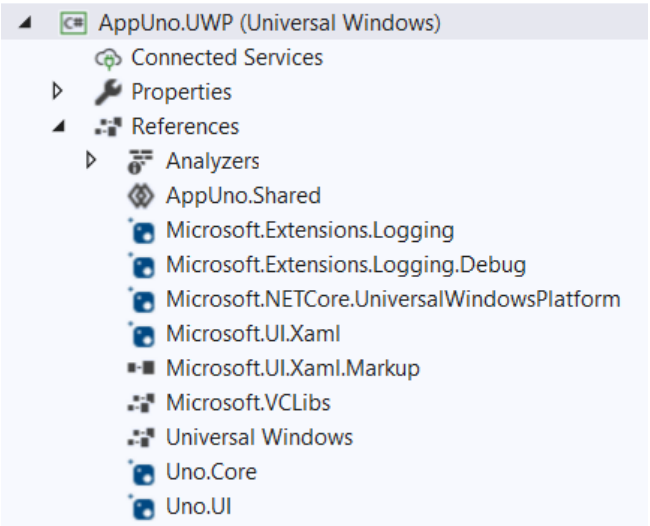

Figure 2-o: The AppUno.UWP Project References

We can see that as part of the **References**, the **AppUno.UWP** project includes some libraries that we have already seen, such as **Microsoft.Extensions.Logging**.

Besides those libraries, the **Microsoft.UI.Xaml** and **Microsoft.UI.Xaml.Markup** libraries are used for rendering the UI using XAML.

It is interesting to notice that this project contains a reference to the **Microsoft.VCLibs** library, which is the C++ Runtime packages for Desktop Bridge. An application running in a Desktop Bridge that uses an incorrect version of the C++ Runtime libraries could fail to access resources.

Besides that, this project includes the **Universal Windows** runtime, and the **Uno.Core** and **Uno.UI** libraries.

Just like all the other projects, this one includes a reference to the **AppUno.Shared** project, which is intended to include the business logic.

AppUno.Wasm

The **AppUno.Wasm** project is intended for Uno to execute the application on web browsers using WebAssembly. Let's look at the project structure.

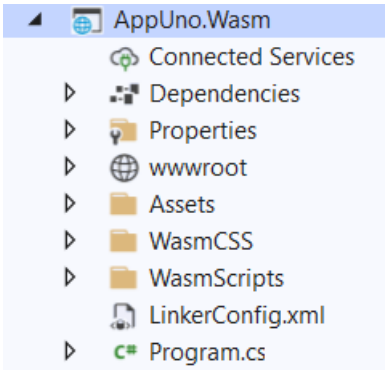

Figure 2-p: The AppUno.Wasm Project Structure

The structure of this project is similar to the organization of web projects. I say this because there is a **wwwroot** folder included as part of the project, and specifically for WebAssembly, we have the **WasmCSS** and **WasmScripts** folders.

The **WasmCSS** folder, as its name implies, is used for keeping the fonts the application will use, as well as CSS files.

The **WasmScripts** folder, as its name implies, is used for storing JavaScript files that the application can use.

The **Assets** folder is primarily used for storing images and fonts, and the **wwwroot** folder contains the **web.config** file.

Regarding the project **Dependencies**, we can find the following.

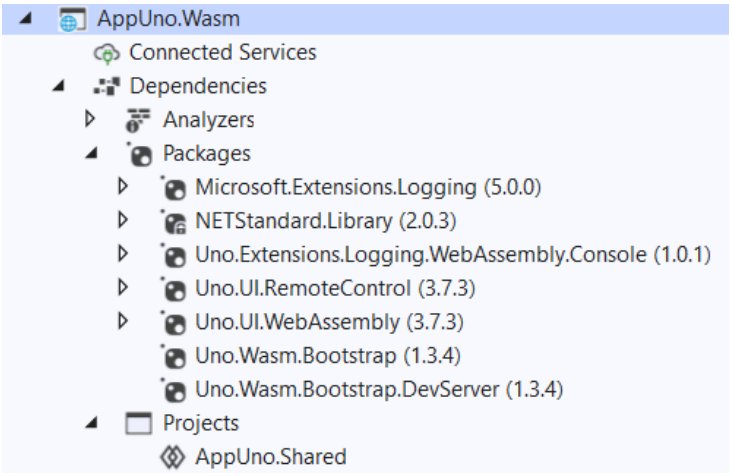

Figure 2-q: The AppUno.Wasm Project Dependencies

This project includes the **Uno.Wasm.Bootstrap** and **Uno.Wasm.Bootstrap.DevServer** libraries, which are the Uno WebAssembly runtimes to execute the application.

On the other hand, the **Uno.UI.WebAssembly** library contains the Uno UI implementation for WebAssembly.

Besides that, the .NET Standard library (**NETStandard.Library**) is referenced, and so is the Microsoft logging extension library.

Like with the rest of the projects, the **AppUno.Shared** project is referenced as well.

AppUno.Shared

The final project of the Uno solution is the **AppUno.Shared** project. Let's explore its structure.

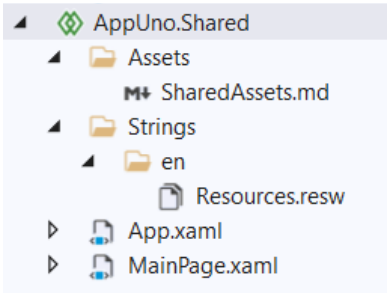

Figure 2-r: The AppUno.Shared Project Structure

The project contains an **Assets** and a **Strings** folder. The former is for storing assets shared among and used across all the other projects, as its name implies.

The latter is used for storing string translations, which can also be shared across the rest of the projects. Then we can see the **App.xaml** and **MainPage.xaml** files we explored in the previous chapter.

Something significant to mention is that we are free to add to the **AppUno.Shared** project as many folders, subfolders, and files as our application needs.

The **AppUno.Shared** project is where we as developers have the freedom to be as creative as we can be.

My recommendation would be to keep the default structure for each solution's projects, except for the **AppUno.Shared** project.

Summary

We've reached the end of this chapter—and covered quite a bit of ground. Although going through the structure of each of the solution's projects is not super exciting, in my opinion, this is a fundamental topic to cover, providing context on what type of applications can be built with the same solution.

The next chapter will explore some XAML fundamentals necessary to build any Uno app user interface.

Chapter 3 XAML Fundamentals

Overview

The Extensible Application Markup Language (XAML) is an XML-based markup language that Microsoft maintains and is used for declarative programming similar to HTML.

Some of the UI systems that are based on XAML are UWP, WPF, Xamarin.Forms, and the topic of this book, the Uno Platform. XAML is used for declaring UI elements with data binding.

XAML has some differences across different UI systems, but the XAML flavor used in Uno is closest to UWP XAML, which means most UWP XAML resources can be used with Uno.

All the code changes will be done exclusively on the **AppUno.Shared** project files.

XAML pages

Using the Visual Studio 2019 Solution Explorer, navigate to the **AppUno.Shared** project and double-click the **MainPage.xaml** file. The file looks as follows.

Listing 3-a: The MainPage.xaml File – AppUno.Shared Project

```
<Page
    x:Class="AppUno.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:AppUno"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">

    <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
        <TextBlock Text="Hello, world!" Margin="20" FontSize="30" />
    </Grid>
</Page>
```

The preceding content of **MainPage.xaml** is what a typical XAML page looks like, and it is nothing more than a markup representation of the UI object model.

Note: The name MainPage is the default and most familiar naming convention for an Uno app's main page, found in Uno and XAML tutorials you'll find on the web. So, we'll stick to that naming convention for simplicity. Here's one example of many.

In this example, we can see that the page content has a **`Grid`** component, and inside it is a **`TextBlock`** component.

The **`Grid`** component is not visible when the application runs, and only the **`TextBlock`** component can be seen. The following diagram illustrates this.

To run the application, set the **AppUno.UWP** project as the startup project by right-clicking it and choosing **Set as Startup Project**.

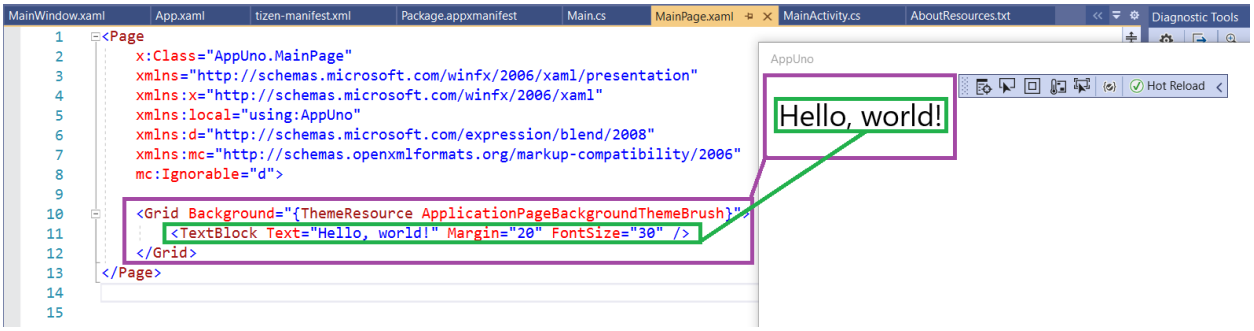

Figure 3-a: The Relationship Between the XAML Component and the App's UI

We can see the **`TextBlock`** component rendered when the application runs, but not the **`Grid`** component.

The **`Grid`** component is a layout panel that arranges the controls inside, depending on the panel's behavior.

The **`TextBlock`** is a component that displays text and is accessible as a variable by giving it a name, as seen in the following listing.

Listing 3-b: The MainPage.xaml File – Giving the TextBlock a Name

```
<Page
    x:Class="AppUno.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:AppUno"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">

    <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
```

```
        <TextBlock Text="Hello, world!" Margin="20" FontSize="30"
         x:Name="tb"/>
    </Grid>
</Page>
```

We have given the **`TextBlock`** component the name **`tb`** by using **`x:Name`**. Now we can reference this variable in the code-behind, which is found on the **MainPage.xaml.cs** file.

Using the code-behind, we can access the variables that reference components we have declared in the XAML and respond to events.

Let's change the value of the **`Text`** property of the **`TextBlock`** component by using the code-behind. To do that, open the **MainPage.xaml.cs** file and replace the existing code as follows. The code change is highlighted in bold.

Listing 3-c: The MainPage.xaml.cs File – Changing the TextBlock Text

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Runtime.InteropServices.WindowsRuntime;
using Windows.Foundation;
using Windows.Foundation.Collections;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Controls.Primitives;
using Windows.UI.Xaml.Data;
using Windows.UI.Xaml.Input;
using Windows.UI.Xaml.Media;
using Windows.UI.Xaml.Navigation;

namespace AppUno
{
    public sealed partial class MainPage : Page
    {
        public MainPage()
        {
            this.InitializeComponent();
            this.tb.Text = "New text";
        }
    }
}
```

Notice how we can access the **Text** property of the **TextBlock** (**tb**) component and assign a new value to it.

If we now run the **AppUno.UWP** project by clicking on the **Run** button, the text displayed by the application is different, modified by the code-behind.

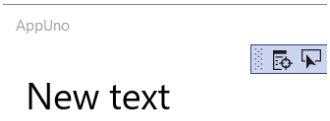

Figure 3-b: AppUno.UWP Running (Changed Text)

Grid component

With the basics of how XAML works covered, let's talk about arranging items on the screen to create a user interface. The **Grid** component facilitates this.

A **Grid** component is a type of layout panel, and it is used to arrange other UI controls within it as a grid.

Most XAML-based applications rely on using the **Grid** or **StackPanel** components to organize other controls. While the **StackPanel** component organizes controls next to each other (horizontally) or on top of each other (vertically), the **Grid** component organizes controls into cells that belong to a row and column.

We need to add the row and column definitions to split the **Grid** component into rows and columns. The updated **MainPage.xaml.cs** code is highlighted in bold in the following listing.

Listing 3-d: The MainPage.xaml File – Adding a Grid

```
<Page
    x:Class="AppUno.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:AppUno"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">

    <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
        <Grid.RowDefinitions>
            <RowDefinition Height="1*" />
            <RowDefinition Height="1*" />
        </Grid.RowDefinitions>

        <Grid.ColumnDefinitions>
```

```
            <ColumnDefinition Width="1*" />
            <ColumnDefinition Width="1*" />
        </Grid.ColumnDefinitions>

        <TextBlock Text="1" Margin="20" FontSize="30" x:Name="tb"/>
        <TextBlock Text="2" Margin="20" FontSize="30" x:Name="tb2"
         Grid.Row="1"/>
        <TextBlock Text="3" Margin="20" FontSize="30" x:Name="tb3"
         Grid.Column="1"/>
        <TextBlock Text="4" Margin="20" FontSize="30" x:Name="tb4"
         Grid.Row="1" Grid.Column="1"/>
    </Grid>
</Page>
```

What we have done here is first added the row and column definitions. Notice that two rows are defined, contained within **`Grid.RowDefinitions`**. The column definitions are included within **`Grid.ColumnDefinitions`**.

Each **`RowDefinition`** includes a **`Height`** property that has a value of **`1*`**. The asterisk tells the **`Grid`** component to separate the rows by ratio to be equally divided.

Therefore, the **`Grid`** component has two rows, each row taking the total height (which is indicated by the value **`1`**), and each row occupying the same height (which is what **`*`** means).

The column's definition follows the same logic. Instead of the **`Height`** property, the **`Width`** property is used.

Each column definition uses the **`Width`** property with a value of **`1*`**. The asterisk tells the **`Grid`** component to separate the columns by ratio to be equally divided.

Following that, I copied the **`TextBlock`** component three times and renamed each of the **`TextBlock`** component copies. For those copies, I also added the **`Grid.Row`** and **`Grid.Column`** properties to indicate in which row or column each should appear.

Notice that in XAML, rows and columns are zero-based, which means that the first row and column start with zero instead of one. So, **`Grid.Row="1" Grid.Column="1"`** indicates the second row and second column, respectively.

So, before we execute the **AppUno.UWP** application, let's look at the following diagram to understand the relationship between the XAML code and what is seen on the screen when the application runs.

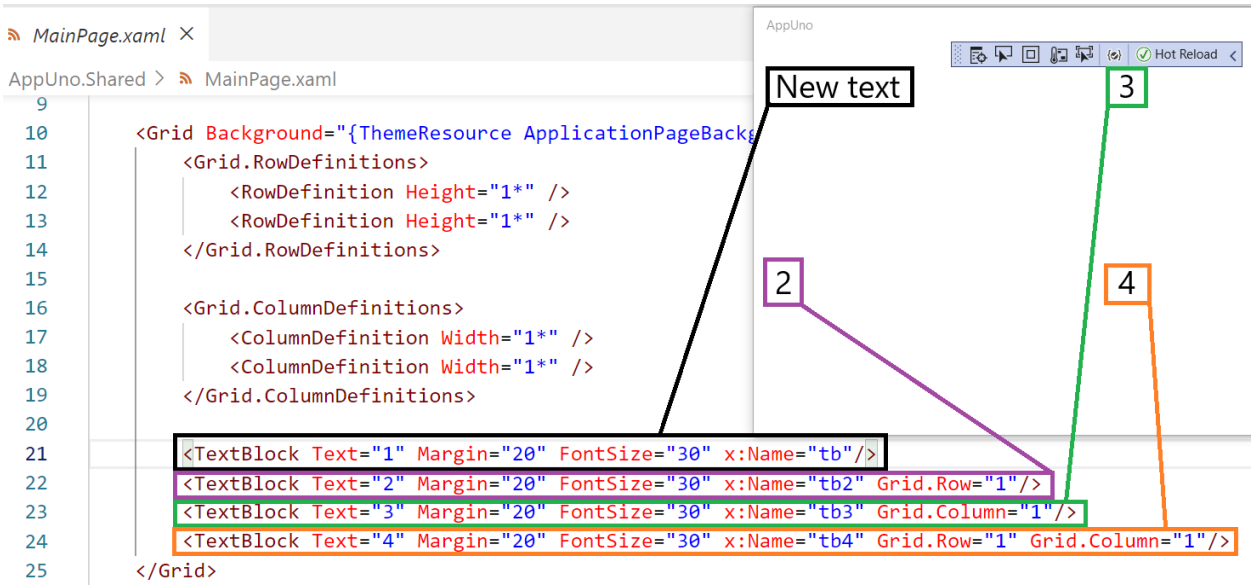

Figure 3-c: Relationship Between the XAML Code and Grid Layout

As we can see, the first **`TextBlock`** component corresponds to the element with **New text**, and then the second **`TextBlock`** corresponds to the part with the value **2**.

The third **`TextBlock`** corresponds to element **3**, and the fourth **`TextBlock`** corresponds to the part with the value **4**.

The order of how these elements are rendered on the screen is controlled by the **`Grid.Row`** and **`Grid.Column`** properties. When either **`Grid.Row`** or **`Grid.Column`** is not specified, the default value is used, which is zero.

If we now run the **AppUno.UWP** application by clicking the Run button, we should see the following result.

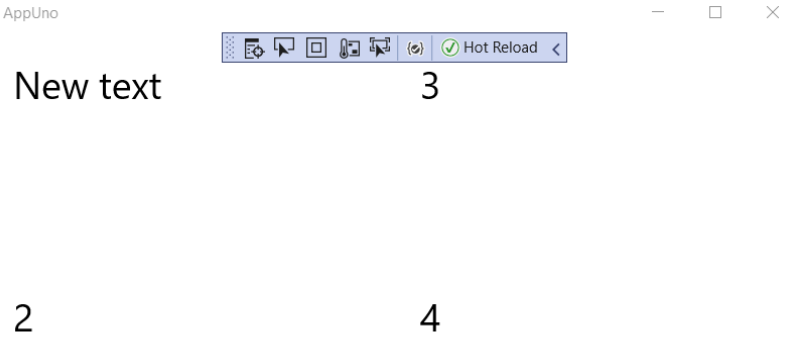

Figure 3-d: The AppUno.UWP Application Running (Using a Grid Component)

Notice that each element is positioned on the top-left corner of each container cell—the default position within a container. It is possible to change this position to another value by using the **`HorizontalAlignment`** and **`VerticalAlignment`** properties and setting their values to **`Center`**.

Let's add these to each of the **`TextBlock`** components to see how the application renders them. The updated code has been highlighted in bold in the following listing.

Listing 3-e: The MainPage.xaml File – Centering Elements Using a Grid

```
<Page
    x:Class="AppUno.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:AppUno"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">

    <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
        <Grid.RowDefinitions>
            <RowDefinition Height="1*" />
            <RowDefinition Height="1*" />
        </Grid.RowDefinitions>

        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="1*" />
            <ColumnDefinition Width="1*" />
        </Grid.ColumnDefinitions>

        <TextBlock Text="1" Margin="20" FontSize="30" x:Name="tb"
         HorizontalAlignment="Center"
         VerticalAlignment="Center"/>
        <TextBlock Text="2" Margin="20" FontSize="30" x:Name="tb2"
         Grid.Row="1"
         HorizontalAlignment="Center"
         VerticalAlignment="Center"/>
        <TextBlock Text="3" Margin="20" FontSize="30" x:Name="tb3"
         Grid.Column="1"
         HorizontalAlignment="Center"
         VerticalAlignment="Center"/>
        <TextBlock Text="4" Margin="20" FontSize="30" x:Name="tb4"
         Grid.Row="1" Grid.Column="1"
         HorizontalAlignment="Center"
         VerticalAlignment="Center"/>
    </Grid>
</Page>
```

If you still have the application running, all you need to do is make the changes to your code and save it.

If the hot reload feature is active, you should see the updates on the application's screen immediately, without having to re-run the application. In my case, this looks as follows.

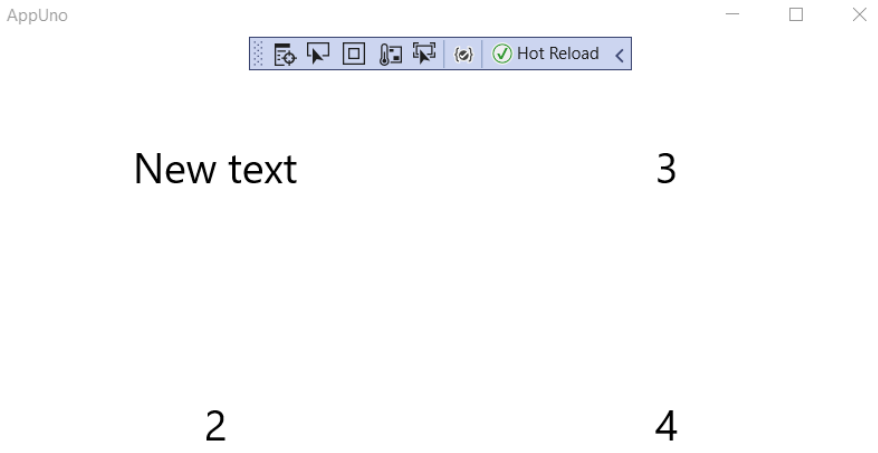

Figure 3-e: The AppUno.UWP Application Running (Using a Grid Component with Centered Elements)

One of the significant features of using the **HorizontalAlignment** and **VerticalAlignment** properties is that even if the application is resized, its components will retain their aspect ratio.

It is also possible to set specific cell sizes in a **Grid** component by indicating a fixed **Width** and **Height** for the **RowDefinition** or **ColumnDefinition** in question, which is done by removing the asterisk from the value.

You can also use the **HorizontalAlignment** and **VerticalAlignment** properties on the **Grid** component itself. Let's give that a try—the changes are highlighted in bold in the following listing.

Listing 3-f: The MainPage.xaml File – Grid Alignment

```
<Page
    x:Class="AppUno.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:AppUno"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">

    <Grid HorizontalAlignment="Center" VerticalAlignment="Center"
     Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
        <Grid.RowDefinitions>
            <RowDefinition Height="1*" />
            <RowDefinition Height="1*" />
        </Grid.RowDefinitions>
```

```
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="1*" />
            <ColumnDefinition Width="1*" />
        </Grid.ColumnDefinitions>

        <TextBlock Text="1" Margin="20" FontSize="30" x:Name="tb"
         HorizontalAlignment="Center"
         VerticalAlignment="Center"/>
        <TextBlock Text="2" Margin="20" FontSize="30" x:Name="tb2"
         Grid.Row="1"
         HorizontalAlignment="Center"
         VerticalAlignment="Center"/>
        <TextBlock Text="3" Margin="20" FontSize="30" x:Name="tb3"
         Grid.Column="1"
         HorizontalAlignment="Center"
         VerticalAlignment="Center"/>
        <TextBlock Text="4" Margin="20" FontSize="30" x:Name="tb4"
         Grid.Row="1" Grid.Column="1"
         HorizontalAlignment="Center"
         VerticalAlignment="Center"/>
    </Grid>
</Page>
```

If you still have the application running, all you need to do is make the changes to your code and save it.

If the hot reload feature is active, you should see the updates on the application's screen immediately without having to re-run the application. In my case, this looks as follows.

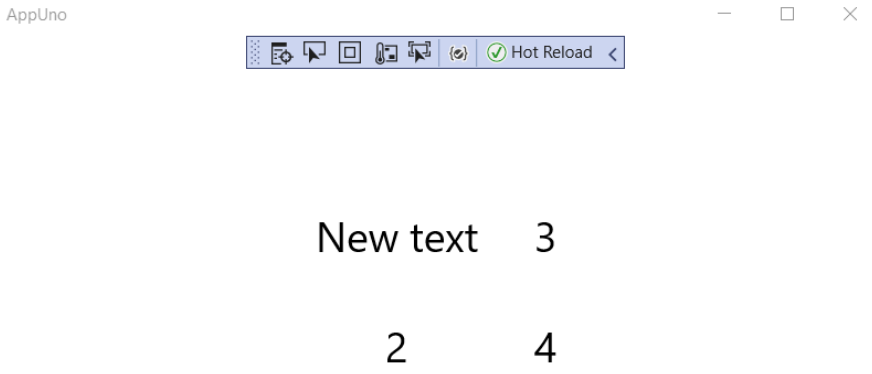

Figure 3-f: The AppUno.UWP Application Running (Using a Centered Grid Component)

Notice how the **`Grid`** component appears centered on the screen of the application, and it no longer adjusts to the dimensions of the screen.

Furthermore, we can add many **RowDefinitions** and **ColumnDefinition** elements as needed to create any grid-like interface required.

StackPanel component

The difference between the **StackPanel** and the **Grid** component is that the **StackPanel** includes controls on top of each other.

The default orientation of the **StackPanel** is vertical. Let's have a look at a basic example.

Listing 3-g: The MainPage.xaml File – StackPanel

```
<Page
    x:Class="AppUno.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:AppUno"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">

    <StackPanel
     Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
        <TextBlock Grid.Row="0" Grid.RowSpan="2" Margin="20" FontSize="30"
         x:Name="tb"
         HorizontalAlignment="Center" VerticalAlignment="Center"/>
        <TextBlock Text="3" Margin="20" FontSize="30" x:Name="tb3"
         Grid.Column="1"
         HorizontalAlignment="Center" VerticalAlignment="Center"/>
        <TextBlock Text="4" Margin="20" FontSize="30" x:Name="tb4"
         Grid.Row="1" Grid.Column="1"
         HorizontalAlignment="Center" VerticalAlignment="Center"/>
    </StackPanel>
</Page>
```

If you still have the application running, all you need to do is make the changes to your code and then save it. You should see the updates on the application's screen. In my case, this looks as follows.

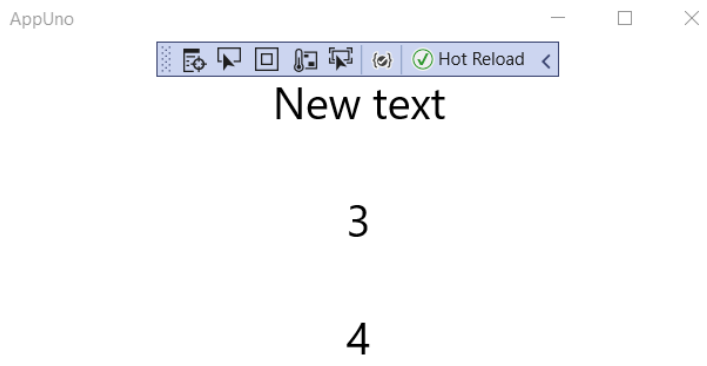

Figure 3-g: The AppUno.UWP Application Running (Using StackPanel Component)

We can see that the **`TextBlock`** controls defined within the **`StackPanel`** are now stacked, one after the other.

Notice as well that row or column definitions are no longer required. However, we can change the default orientation, so let's do that. The code change is highlighted in bold in the following listing.

Listing 3-h: The MainPage.xaml File – StackPanel (Horizontal Orientation)

```
<Page
    x:Class="AppUno.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:AppUno"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">

    <StackPanel Orientation="Horizontal"
     Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
        <TextBlock Grid.Row="0" Grid.RowSpan="2" Margin="20" FontSize="30"
         x:Name="tb"
         HorizontalAlignment="Center" VerticalAlignment="Center"/>
        <TextBlock Text="3" Margin="20" FontSize="30" x:Name="tb3"
         Grid.Column="1"
         HorizontalAlignment="Center" VerticalAlignment="Center"/>
        <TextBlock Text="4" Margin="20" FontSize="30" x:Name="tb4"
         Grid.Row="1" Grid.Column="1"
         HorizontalAlignment="Center" VerticalAlignment="Center"/>
    </StackPanel>
</Page>
```

If you still have the application running, all you need to do is make and save the changes to your code, and you will see the screen updated. In my case, this looks as follows.

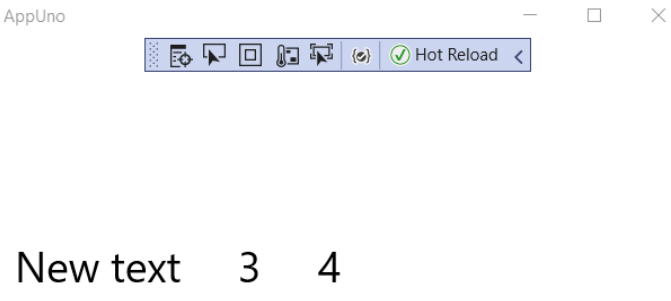

Figure 3-h: The AppUno.UWP Application Running (StackPanel with Horizontal Orientation)

As you have seen, the only change was to add the **`Orientation`** parameter and set its value to **`Horizontal`**.

The **`StackPanel`** component is excellent for aligning controls from the left, making any text quickly read like a book.

It's also possible to nest **`StackPanel`** components and alternatively combine that with the **`Grid`** component. All combinations are technically possible as long as it renders the UI correctly for your application as you intend it to be.

Let's create a layout that includes three input fields and adjust the code to achieve that. The changes are highlighted in bold.

Listing 3-i: The MainPage.xaml File – Nested StackPanels (Data Entry Layout)

```
<Page
    x:Class="AppUno.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:AppUno"
    xmlns:d="http://schemas.microasoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">

    <StackPanel>
        <StackPanel Orientation="Horizontal"
         Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
            <TextBlock Text="Name" Margin="20" FontSize="30"
             x:Name="laName" Grid.Column="1"
             HorizontalAlignment="Center"
             VerticalAlignment="Center"/>
            <TextBox Text="" Margin="20" FontSize="30"
             x:Name="Name" Grid.Row="1" Grid.Column="1"
```

```
            HorizontalAlignment="Center" VerticalAlignment="Center"/>
        </StackPanel>

        <StackPanel Orientation="Horizontal"
         Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
            <TextBlock Text="Age" Margin="20" FontSize="30"
             x:Name="laAge" Grid.Column="1"
             HorizontalAlignment="Center"
             VerticalAlignment="Center"/>
            <TextBox Text="" Margin="20" FontSize="30" x:Name="Age"
             Grid.Row="1" Grid.Column="1"
             HorizontalAlignment="Center"
             VerticalAlignment="Center"/>
        </StackPanel>

        <StackPanel Orientation="Horizontal"
         Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
            <TextBlock Text="Sex" Margin="20" FontSize="30"
             x:Name="laSex" Grid.Column="1"
             HorizontalAlignment="Center" VerticalAlignment="Center"/>
            <TextBox Text="" Margin="20" FontSize="30"
             x:Name="Sex" Grid.Row="1" Grid.Column="1"
             HorizontalAlignment="Center" VerticalAlignment="Center"/>
        </StackPanel>
    </StackPanel>
</Page>
```

Also, open the **MainPage.xaml.cs** file and remove the following line from the **`MainPage`** constructor method, as we no longer need it: **`this.tb.Text = "New text";`**.

If you still have the application running, all you need to do is make the changes to your code and save it, and you will see the screen updated. In my case, this looks as follows.

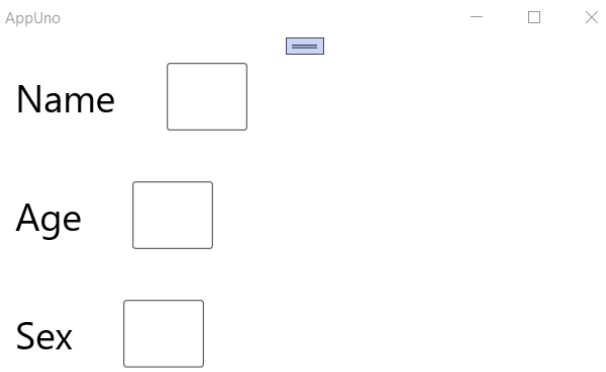

Figure 3-i: The AppUno.UWP Application Running – Nested StackPanels (Data Entry Layout)

As we can see, there are now three input fields that appear vertically stacked, each with its corresponding label next to the input field, horizontally stacked. Let's have a look at how we achieved this.

We first defined a **`StackPanel`** component vertically aligned to its container (the default orientation), the outermost **`StackPanel`**.

Within that outermost **`StackPanel`**, we defined three **`StackPanel`** components horizontally aligned, each of them containing a label (**`TextBlock`**) and an input field (**`TextBox`**). This is why each label and respective input field appear next to each other.

To understand this better, let's look at the following diagram. By looking at how the colors match the code and the finished UI, we can identify the relationship between the XAML markup and controls on the screen.

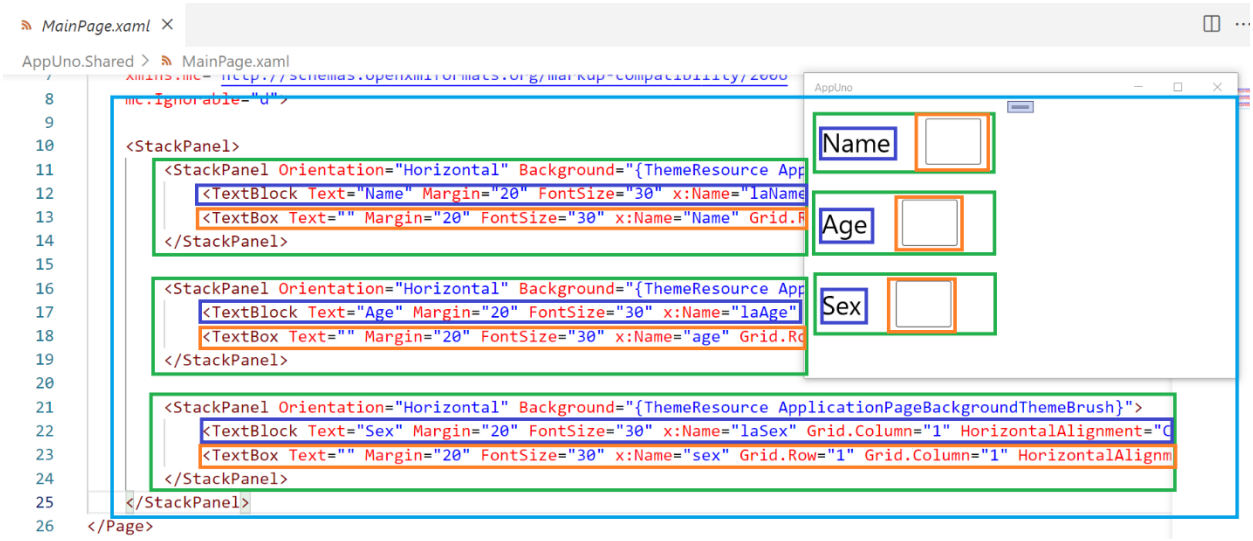

Figure 3-j: The Relationship between the XAML and UI – AppUno.UWP App Running

Notice as well that each of the **`TextBlock`** and **`TextBox`** components have been given a unique name. This is a good practice, in case we need to manipulate those components with the code-behind.

XAML styles

So far, the controls and layout we have rendered on the screen do not use any colors or style. That's not bad; however, all applications look better when they have a bit of styling.

Styles allow you to set properties across multiple controls of the same type instead of assigning those properties one by one for each component. Styles are nothing more than resources that are referenced through the application. We can create a consistent user experience for a specific set of controls used by the application using styles.

To use styles, we need to use setters which set properties on a control, but we need to define the resource before doing that. Let's have a look at how this is done. The code changes are highlighted in bold in the following listing.

Listing 3-j: The MainPage.xaml File – Using a Page Style

```
<Page
    x:Class="AppUno.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:AppUno"
    xmlns:d="http://schemas.microasoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">

    <Page.Resources>
        <Style x:Key="st" TargetType="TextBlock">
            <Style.Setters>
                <Setter Property="Foreground" Value="Blue"/>
            </Style.Setters>
        </Style>
    </Page.Resources>

    <StackPanel>
        <StackPanel Orientation="Horizontal"
         Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
            <TextBlock Style="{StaticResource st}"
             Text="Name" Margin="20" FontSize="30"
             x:Name="laName" Grid.Column="1"
             HorizontalAlignment="Center"
             VerticalAlignment="Center"/>
            <TextBox Text="" Margin="20" FontSize="30"
             x:Name="Name" Grid.Row="1" Grid.Column="1"
             HorizontalAlignment="Center"
             VerticalAlignment="Center"/>
        </StackPanel>

        <StackPanel Orientation="Horizontal"
         Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
            <TextBlock Text="Age" Margin="20" FontSize="30"
             x:Name="laAge" Grid.Column="1"
             HorizontalAlignment="Center" VerticalAlignment="Center"/>
            <TextBox Text="" Margin="20" FontSize="30"
             x:Name="Age" Grid.Row="1" Grid.Column="1"
             HorizontalAlignment="Center" VerticalAlignment="Center"/>
```

```
        </StackPanel>

        <StackPanel Orientation="Horizontal"
         Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
            <TextBlock Text="Sex" Margin="20" FontSize="30"
             x:Name="laSex" Grid.Column="1"
             HorizontalAlignment="Center" VerticalAlignment="Center"/>
            <TextBox Text="" Margin="20" FontSize="30"
             x:Name="Sex" Grid.Row="1" Grid.Column="1"
             HorizontalAlignment="Center" VerticalAlignment="Center"/>
        </StackPanel>
    </StackPanel>
</Page>
```

If the application is running, all you need to do is make the changes to your code and save it to see the screen updated. In my case, this looks as follows.

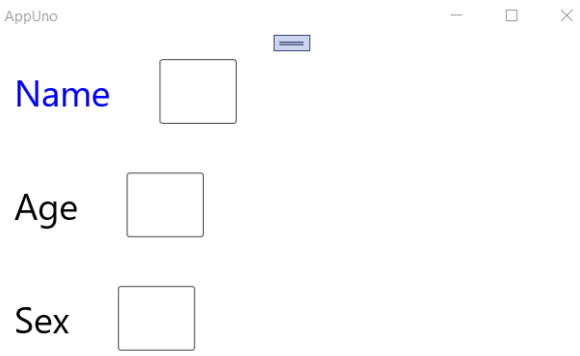

Figure 3-k: Style Applied to a TextBlock Component (Locally and Explicitly)

We have defined a style and then assigned that style to the application's first **`TextBlock`** component using this property: **`Style="{StaticResource st}"`**. The style name is **`st`**, and it is defined with the following code:

```
<Style x:Key="st" TargetType="TextBlock">
  <Style.Setters>
    <Setter Property="Foreground" Value="Blue"/>
  </Style.Setters>
</Style>
```

We can add more styles between the **`<Page.Resources>`** and **`<Page.Resources/>`** tags. Currently, this style is only applicable to the current page, which corresponds to the **MainPage.xaml** file.

However, we can create styles that apply to all the pages of an application, which is achieved by adding them to the **App.xaml** file of the **AppUno.Shared** project. Let's do that. The changes are highlighted in bold in the following listing.

Listing 3-k: The App.xaml.cs File – Using a Global Explicit Style

```
<Application
    x:Class="AppUno.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:AppUno">

    <Application.Resources>
        <Style x:Key="st" TargetType="TextBlock">
            <Style.Setters>
                <Setter Property="Foreground" Value="Blue"/>
            </Style.Setters>
        </Style>
    </Application.Resources>
</Application>
```

We have moved the style resource from the **MainPage.xaml** file to the **App.xaml** file. The old content was removed, and the new style from **MainPage.xaml** was added between the **<Application.Resources>** and **</Application.Resources>** tags.

After making this adjustment, the **MainPage.xaml** file should now look as follows. As you can see, the content between the **<Page.Resources>** and **<Page.Resources/>** tags and the tags themselves are gone.

Listing 3-l: The Updated MainPage.xaml.cs File (Using a Resource)

```
<Page
    x:Class="AppUno.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:AppUno"
    xmlns:d="http://schemas.microasoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">

    <StackPanel>
        <StackPanel Orientation="Horizontal"
         Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
            <TextBlock Style="{StaticResource st}"
             Text="Name" Margin="20" FontSize="30"
             x:Name="laName" Grid.Column="1"
             HorizontalAlignment="Center"
             VerticalAlignment="Center"/>
            <TextBox Text="" Margin="20" FontSize="30"
```

```
            x:Name="Name" Grid.Row="1" Grid.Column="1"
            HorizontalAlignment="Center"
            VerticalAlignment="Center"/>
        </StackPanel>

        <StackPanel Orientation="Horizontal"
         Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
            <TextBlock Text="Age" Margin="20" FontSize="30"
            x:Name="laAge" Grid.Column="1"
            HorizontalAlignment="Center" VerticalAlignment="Center"/>
            <TextBox Text="" Margin="20" FontSize="30"
            x:Name="Age" Grid.Row="1" Grid.Column="1"
            HorizontalAlignment="Center" VerticalAlignment="Center"/>
        </StackPanel>

        <StackPanel Orientation="Horizontal"
         Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
            <TextBlock Text="Sex" Margin="20" FontSize="30"
            x:Name="laSex" Grid.Column="1"
            HorizontalAlignment="Center" VerticalAlignment="Center"/>
            <TextBox Text="" Margin="20" FontSize="30"
            x:Name="Sex" Grid.Row="1" Grid.Column="1"
            HorizontalAlignment="Center" VerticalAlignment="Center"/>
        </StackPanel>
    </StackPanel>
</Page>
```

If the application is running, all you need to do is save the changes to your code, save both the **App.xaml** and **MainPage.xaml** files, and see the application's UI updated. In my case, this looks as follows.

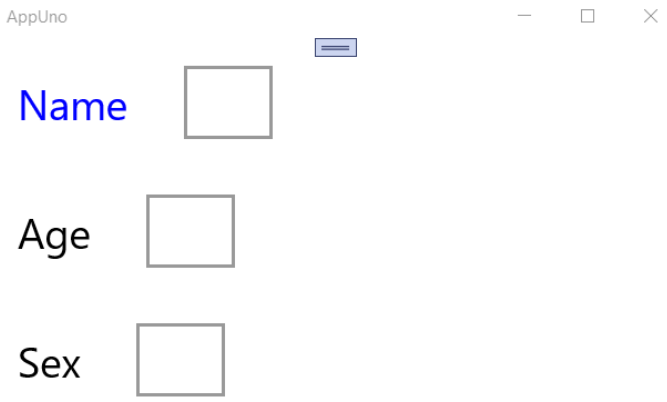

Figure 3-I: Style Applied to a TextBlock Component (Globally and Explicitly)

Referencing a style by its key, in this case **`x:Key="st"`**, is known as explicit styling, and it is specific to the UI element that references that key.

However, if we remove the key, it becomes an implicit style, and as such, the **`Style`** resource applies to all the components of that same type—in this case, the **`TextBlock`** components, because the style's **`TargetType`** value is **`TextBlock`**.

Let's go back to the **App.xaml** file and remove the key from the **`Style`** resource. The updated code follows.

Listing 3-m: The App.xaml.cs File – Using a Global Implicit Style

```
<Application
    x:Class="AppUno.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:AppUno">

    <Application.Resources>
        <Style TargetType="TextBlock">
            <Style.Setters>
                <Setter Property="Foreground" Value="Blue"/>
            </Style.Setters>
        </Style>
    </Application.Resources>
</Application>
```

We need to do one more modification, and that is to remove **`Style="{StaticResource st}"`** from the first **`TextBlock`** component on the **MainPage.xaml** file.

Listing 3-n: The Updated MainPage.xaml File (Without the Resource)

```
<Page
    x:Class="AppUno.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:AppUno"
    xmlns:d="http://schemas.microasoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">

    <StackPanel>
        <StackPanel Orientation="Horizontal"
         Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
            <TextBlock Text="Name" Margin="20" FontSize="30"
             x:Name="laName" Grid.Column="1"
```

```
            HorizontalAlignment="Center"
            VerticalAlignment="Center"/>
        <TextBox Text="" Margin="20" FontSize="30"
            x:Name="Name" Grid.Row="1" Grid.Column="1"
            HorizontalAlignment="Center"
            VerticalAlignment="Center"/>
    </StackPanel>

    <StackPanel Orientation="Horizontal"
     Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
        <TextBlock Text="Age" Margin="20" FontSize="30"
            x:Name="laAge" Grid.Column="1"
            HorizontalAlignment="Center" VerticalAlignment="Center"/>
        <TextBox Text="" Margin="20" FontSize="30"
            x:Name="Age" Grid.Row="1" Grid.Column="1"
            HorizontalAlignment="Center" VerticalAlignment="Center"/>
    </StackPanel>

    <StackPanel Orientation="Horizontal"
     Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
        <TextBlock Text="Sex" Margin="20" FontSize="30"
            x:Name="laSex" Grid.Column="1"
            HorizontalAlignment="Center" VerticalAlignment="Center"/>
        <TextBox Text="" Margin="20" FontSize="30"
            x:Name="Sex" Grid.Row="1" Grid.Column="1"
            HorizontalAlignment="Center" VerticalAlignment="Center"/>
    </StackPanel>
  </StackPanel>
</Page>
```

If the application is running, all you need to do is save the **App.xaml** and **MainPage.xaml** file changes, and see the application's UI updated. In my case, this looks as follows.

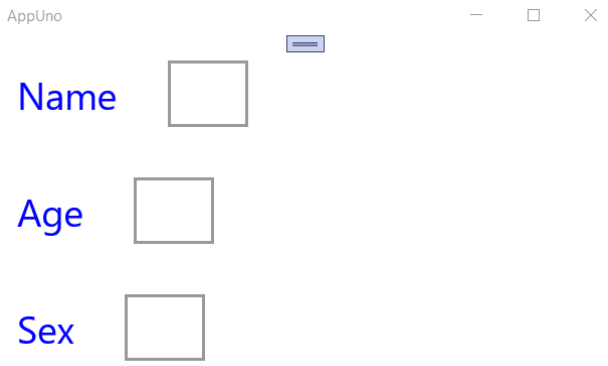

Figure 3-m: Style Applied to all TextBlock Components (Globally and Implicitly)

We can now see that the **`Style`** resource has been applied to all **`TextBlock`** components; therefore, the style is implicit.

If you've used UWP styling before, these concepts will be familiar to you, and you'll feel right at home. However, there are some differences between Uno UI and UWP, so it's good to go over this article to understand those differences.

Data binding: model

Uno applications rely extensively on data binding, which provides the UI a way to retrieve and display data from the application's model. This is called one-way data binding. Uno also can synchronize the data from the UI to the model and vice-versa, known as two-way data binding.

To use data binding, we can use the {x:Bind} or **`{Binding}`** markup extensions, which work primarily the same way; however, there are some slight differences between them.

To see data binding in action, let's add a new C# class file to the **AppUno.Shared** project. To do that, if the application is still running, make sure you stop it first.

> ***Note: Because you will stop the application, I recommended that later you clean and build the complete AppUno solution.***

With Solution Explorer opened, right-click the **AppUno.Shared** project, then click **Add** > **New item**. The following window will open.

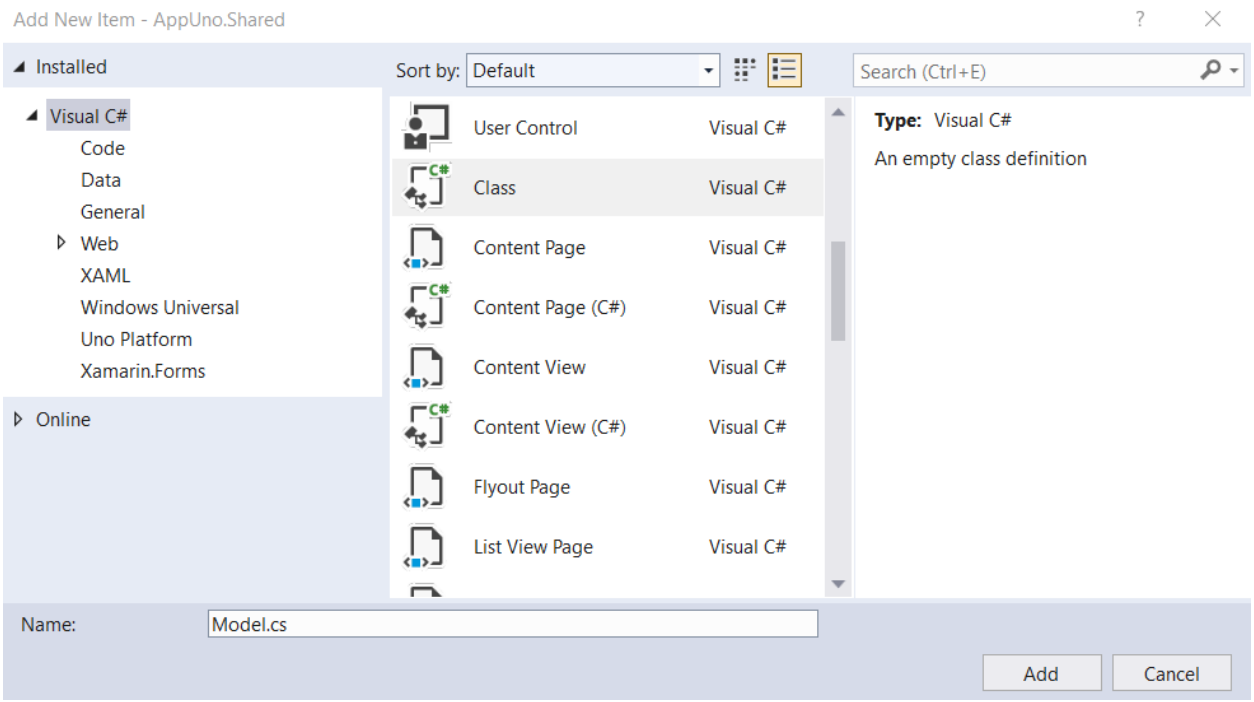

Figure 3-n: The Add New Item Window

Select the **Class** option and give it a name—I'll call the file **Model.cs**. Click **Add**.

Sometimes you might want to create a folder within your project to keep files related to each other. For example, you could have created a **Model** folder and placed the **Model.cs** file within that folder. This is a common and good practice, primarily for code organization.

However, for the sake of simplicity, I will skip creating a folder and instead generate the **Model.cs** file directly. Once the file is created, we can see it under the **AppUno.Shared** project.

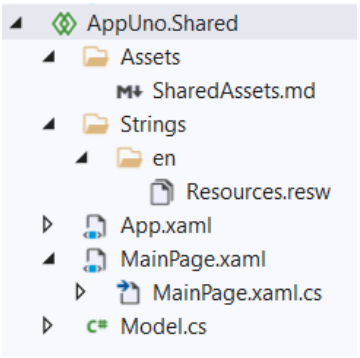

Figure 3-o: The Model.cs File – AppUno.Shared Project

Let's open the **Model.cs** file by double-clicking it within the Solution Explorer, and replace the default code with the following.

Listing 3-o: The Model.cs File

```
using System.ComponentModel;

namespace AppUno
{
    public class Model: INotifyPropertyChanged
    {
        private string name;
        private int age;
        private string sex;

        public string Name
        {
            get => name;
            set
            {
                name = value;
                PropertyChanged?.Invoke(
                  this, new PropertyChangedEventArgs(nameof(Name)));
            }
        }
```

```
        public int Age
        {
            get => age;
            set
            {
                age = value;
                PropertyChanged?.Invoke(
                  this, new PropertyChangedEventArgs(nameof(Age)));
            }
        }

        public string Sex
        {
            get => sex;
            set
            {
                sex = value;
                PropertyChanged?.Invoke(
                  this, new PropertyChangedEventArgs(nameof(Sex)));
            }
        }

        public event PropertyChangedEventHandler PropertyChanged;
    }
}
```

The first thing we did was use the System.ComponentModel namespace, which is used for implementing the behavior of components and controls during runtime.

The **Model** class is declared within the **AppUno** namespace, which is the namespace that is used within the various **AppUno** projects. The **Model** class inherits from the INotifyPropertyChanged interface, which notifies clients subscribed when property values change. This is an essential characteristic of data binding.

Following that, we declared three **private** variables, each corresponding to a UI element: one for **name**, another for **age**, and one for **sex**. These variables are private, meaning they are not exposed and cannot be manipulated or changed outside the class.

These variables are used by the **Name**, **Age**, and **Sex** properties, respectively—which are the ones that will be used to do the data binding in the XAML code, as we will see shortly.

The **Model** class uses the **PropertyChanged** event; a PropertyChangedEventHandler will be raised when the property changes on the component. Therefore, the **Name**, **Age**, and **Sex** properties must invoke this event when a new value is assigned.

The **Name** property retrieves the value from the **name** variable and assigns the new value to the **name** variable, invoking the **PropertyChanged** event:

```
PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(nameof(Name)));
```

The **Age** property retrieves the value from the **age** variable and assigns the new value to the **age** variable, invoking the **PropertyChanged** event:

```
PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(nameof(Age)));
```

Likewise, the **Sex** property retrieves the value from the **sex** variable. It assigns the new value to the **sex** variable, invoking the **PropertyChanged** event:

```
PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(nameof(Sex)));
```

So, with the data binding code in place within the **Model.cs** file, we need to modify the XAML markup to ensure that the **TextBox** components (input fields) bind to these properties.

Data binding: XAML

Let's open the **MainPage.xaml** file and add the bindings to the **TextBox** components. The code changes are highlighted in bold in the following listing.

Listing 3-p: Adding Bindings – MainPage.xaml

```
<Page
    x:Class="AppUno.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:AppUno"
    xmlns:d="http://schemas.microasoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">

    <StackPanel>
        <StackPanel Orientation="Horizontal"
         Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
            <TextBlock
             Text="Name"
             Margin="20" FontSize="30" x:Name="laName" Grid.Column="1"
             HorizontalAlignment="Center" VerticalAlignment="Center"/>
            <TextBox
             Text="{Binding Name}"
             Margin="20" FontSize="30" x:Name="Name"
             Grid.Row="1" Grid.Column="1"
```

```
            HorizontalAlignment="Center" VerticalAlignment="Center"/>
        </StackPanel>

        <StackPanel Orientation="Horizontal"
         Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
            <TextBlock Text="Age" Margin="20" FontSize="30" x:Name="laAge"
             Grid.Column="1"
             HorizontalAlignment="Center"
             VerticalAlignment="Center"/>
            <TextBox
             Text="{Binding Age}"
             Margin="20" FontSize="30" x:Name="Age"
             Grid.Row="1" Grid.Column="1"
             HorizontalAlignment="Center" VerticalAlignment="Center"/>
        </StackPanel>

        <StackPanel Orientation="Horizontal"
         Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
            <TextBlock Text="Sex" Margin="20" FontSize="30" x:Name="laSex"
             Grid.Column="1" HorizontalAlignment="Center"
             VerticalAlignment="Center"/>
            <TextBox
             Text="{Binding Sex}"
             Margin="20" FontSize="30" x:Name="Sex"
             Grid.Row="1" Grid.Column="1"
             HorizontalAlignment="Center" VerticalAlignment="Center"/>
        </StackPanel>
    </StackPanel>
</Page>
```

The changes done are simple. Each **TextBox** component binds to a property of the **Model** class.

On the first **TextBox** component, we bind the **Text** parameter to the **Name** property of the **Model** as follows: **Text="{Binding Name}"**.

On the second **TextBox** component, we bind the **Text** parameter to the **Age** property of the **Model** as follows: **Text="{Binding Age}"**.

On the third **TextBox** component, we bind the **Text** parameter to the **Sex** property of the **Model** as follows: **Text="{Binding Sex}"**.

By doing this, we now have a way to assign the values entered through these **TextBox** (input field) components to the **Model** properties.

But before we can see this working, there's one more modification we need to do, and that is to initialize the **DataContext** object within the **MainPage.xaml.cs** file. This change is highlighted in bold in the following listing.

Listing 3-q: Initializing the DataContext Object – MainPage.xaml.cs

```
using Windows.UI.Xaml.Controls;

namespace AppUno
{
    public sealed partial class MainPage : Page
    {
        public MainPage()
        {
            this.InitializeComponent();
            DataContext = new Model { Name = "Ed", Age = 20, Sex = "M" };
        }
    }
}
```

The initialization of the **DataContext** object is achieved by creating an instance of the **Model** class and assigning values to its properties.

If you now build and run the **AppUno.UWP** project, we should see the following result.

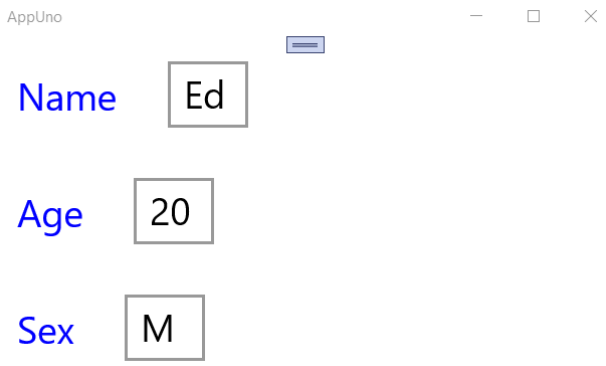

Figure 3-p: AppUno.UWP Running

We can see that by the values assigned to the **DataContext** object and through data binding, each of the **TextBox** components has a value set when the application runs.

Two-way data binding

What we have implemented is one-way data binding. Now, let's implement two-way data binding—which is quickly done by adding the **Mode** parameter and setting its value to **TwoWay**. The changes are highlighted in bold in the following listing.

Listing 3-r: Two-way Data Binding – MainPage.xaml

```
<Page
    x:Class="AppUno.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:AppUno"
    xmlns:d="http://schemas.microasoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">

    <StackPanel>
        <StackPanel Orientation="Horizontal"
         Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
            <TextBlock Text="Name" Margin="20" FontSize="30"
             x:Name="laName" Grid.Column="1"
             HorizontalAlignment="Center" VerticalAlignment="Center"/>
            <TextBox Text="{Binding Name, Mode=TwoWay}" Margin="20"
             FontSize="30" x:Name="Name"
             Grid.Row="1" Grid.Column="1"
             HorizontalAlignment="Center" VerticalAlignment="Center"/>
        </StackPanel>

        <StackPanel Orientation="Horizontal"
         Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
            <TextBlock Text="Age" Margin="20" FontSize="30" x:Name="laAge"
             Grid.Column="1"
             HorizontalAlignment="Center" VerticalAlignment="Center"/>
            <TextBox Text="{Binding Age, Mode=TwoWay}"
             Margin="20" FontSize="30" x:Name="Age"
             Grid.Row="1" Grid.Column="1"
             HorizontalAlignment="Center" VerticalAlignment="Center"/>
        </StackPanel>

        <StackPanel Orientation="Horizontal"
         Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
            <TextBlock Text="Sex" Margin="20" FontSize="30" x:Name="laSex"
             Grid.Column="1"
             HorizontalAlignment="Center" VerticalAlignment="Center"/>
            <TextBox Text="{Binding Sex, Mode=TwoWay}"
             Margin="20" FontSize="30" x:Name="Sex"
             Grid.Row="1" Grid.Column="1"
             HorizontalAlignment="Center" VerticalAlignment="Center"/>
        </StackPanel>
    </StackPanel>
```

```
</Page>
```

As you can see, we have added **Mode=TwoWay** to the binding clause of each **TextBox** component.

To understand this better and see it in action, let's add a **TextBlock** component before the closing tag of the last **StackPanel** component on the **MainPage.xaml** file as follows:

```
<TextBlock Text="{Binding Sex}" Margin="20" FontSize="30" x:Name="laSex2" Grid.Co
lumn="1" HorizontalAlignment="Center" VerticalAlignment="Center"/>
```

Notice that we are binding the **Text** property to the **Sex** property of the **Model**. Therefore, the updated **MainPage.xaml** file should now look as follows. The changes are highlighted in bold.

Listing 3-s: Two-way Data Binding – Updated MainPage.xaml

```
<Page
    x:Class="AppUno.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:AppUno"
    xmlns:d="http://schemas.microasoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">

    <StackPanel>
        <StackPanel Orientation="Horizontal"
         Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
            <TextBlock Text="Name" Margin="20" FontSize="30"
             x:Name="laName" Grid.Column="1"
             HorizontalAlignment="Center" VerticalAlignment="Center"/>
            <TextBox Text="{Binding Name, Mode=TwoWay}" Margin="20"
             FontSize="30" x:Name="Name"
             Grid.Row="1" Grid.Column="1"
             HorizontalAlignment="Center" VerticalAlignment="Center"/>
        </StackPanel>

        <StackPanel Orientation="Horizontal"
         Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
            <TextBlock Text="Age" Margin="20" FontSize="30" x:Name="laAge"
             Grid.Column="1"
             HorizontalAlignment="Center" VerticalAlignment="Center"/>
            <TextBox Text="{Binding Age, Mode=TwoWay}"
             Margin="20" FontSize="30" x:Name="Age"
             Grid.Row="1" Grid.Column="1"
```

```
            HorizontalAlignment="Center" VerticalAlignment="Center"/>
        </StackPanel>

        <StackPanel Orientation="Horizontal"
         Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
            <TextBlock Text="Sex" Margin="20" FontSize="30" x:Name="laSex"
             Grid.Column="1"
             HorizontalAlignment="Center" VerticalAlignment="Center"/>
            <TextBox Text="{Binding Sex, Mode=TwoWay}"
             Margin="20" FontSize="30" x:Name="Sex"
             Grid.Row="1" Grid.Column="1"
             HorizontalAlignment="Center" VerticalAlignment="Center"/>

       <TextBlock Text="{Binding Sex}" Margin="20"

        FontSize="30" x:Name="laSex2" Grid.Column="1"

        HorizontalAlignment="Center" VerticalAlignment="Center"/>

        </StackPanel>
    </StackPanel>
</Page>
```

If you have the application running and save the changes to the **MainPage.xaml** file, you should see the following.

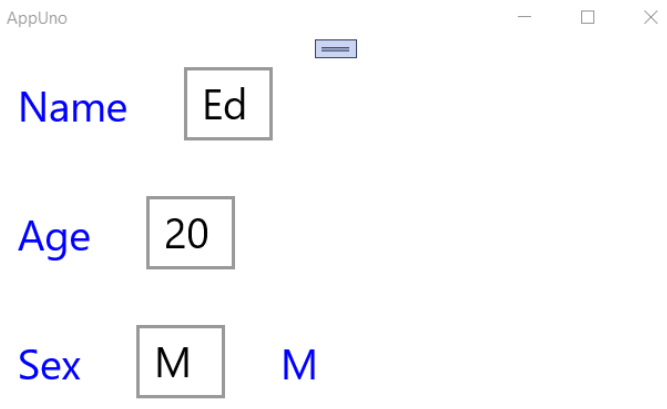

Figure 3-q: Modified AppUno.UWP Running

Notice that there's a new label (**`TextBlock`**) with the value **M** after the **Sex** input field (**`TextBox`**).

If we now change the value of the **Sex** input field to **F** and press Tab to move out of the field, the two-way data binding will update the value of the new label to **F**.

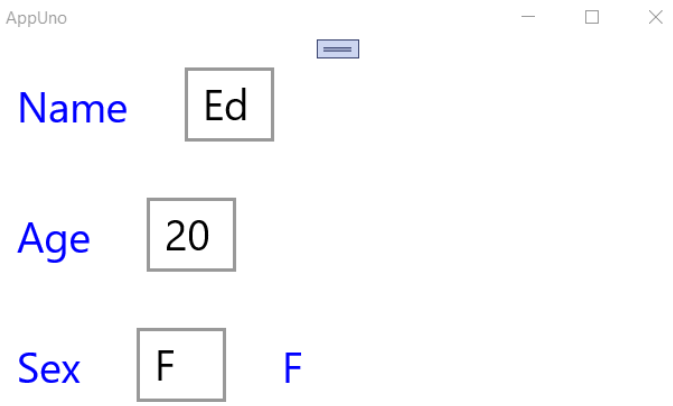

Figure 3-r: Modified AppUno.UWP Running (Updated Value)

Excellent—that worked like a charm!

Summary

Throughout this chapter, we explored some of the essential aspects of XAML and how it can be used to create the UI of an Uno application.

XAML is quite a broad topic, and I could easily write an entire book on it. I invite you to learn more about XAML by reading the official documentation. I also suggest checking out *UWP Succinctly* and *More UWP Succinctly*, which cover how to use XAML for creating UWP apps.

We will explore how to reconstruct a sample Uno single-page application from the ground up in a subsequent chapter.

Chapter 4 Reconstructing an App

Overview

It is now time to reconstruct an application. We will explore the thought process of repurposing an existing application as a potential document tracker single-page application.

This application could help us track the expiration dates of essential documents, such as credit cards, IDs, driver licenses, and passports.

Although it uses some unique features, such as a single XAML component rather than multiple controls and newer library dependencies, this application's code is inspired by the ideas and original code from this repository.

The XAML layout, code organization, solution structure, and views from this repository and this excellent article from the Uno blog were the inspiration for taking that example application and deconstructing, reconstructing, and repurposing it.

So, please check them out once you have read this book, as they are a great way to learn and complement what we will go through.

In general, I invite you to visit GitHub and look at different Uno Platform project repositories, check their code, see what you can learn from them and what you can improve.

Furthermore, the Uno code gallery has practical examples which are great to explore once you have some basic platform knowledge.

Why reconstructing?

I've written many books across different platforms and frameworks using a comprehensive set of technologies, some very different from one another.

However, all those books had one thing in common: a couple of underlying technologies, not more than two or there. Uno is different—it has many aspects, and it targets many other frameworks and platforms.

I've realized that sometimes the best way to learn a comprehensive, new technology—such as Uno—is not constantly building a new application from scratch throughout that writing process.

Uno includes many aspects to learn (such as XAML, MVVM, and targeting different platforms) that are impossible to cover in depth within one short book.

In my opinion, the best way to get acquainted with the basics of Uno is by deconstructing something that exists, rebuilding it, and making some modifications.

So, rather than having a puzzle with a million pieces to fit without the finished picture, we will do the opposite.

We'll take the finished puzzle, look at the entire picture, and then explore each of its constituent parts as if we are building it from scratch—but we'll know where we are heading.

By taking something that exists, pulling it apart, and figuring out how to put it back together—and making some changes in the process—we will cover essential aspects in much more depth than an article.

So, the approach presented here is to stand on the shoulders of giants and explain how the puzzle comes together.

This way, we can cover enough aspects of the platform to understand how it works without being an expert on any underlying technologies. I hope you like the approach and find it fun—because I loved it.

Decluttering

When creating a single-page application using the existing **AppUno** solution, it's best to declutter it and leave only the projects we need.

Tip: If you would like to save time instead of manually decluttering the solution, download the complete decluttered solution template from this GitHub repository.

Using **Solution Explorer** within Visual Studio 2019, remove all the projects within the solution and leave only **AppUno.Wasm** and **AppUno.Shared**. The solution should look as follows.

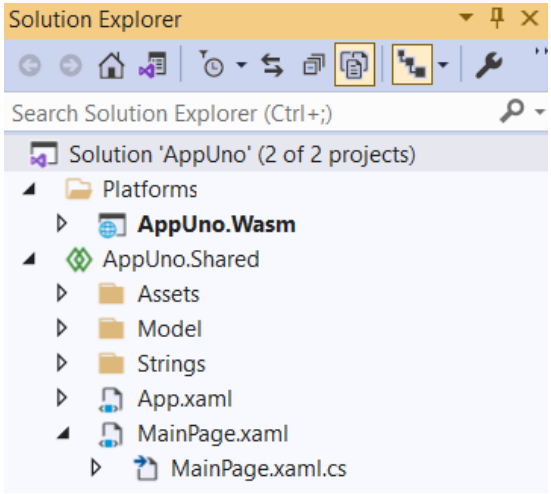

Figure 4-a: The Decluttered VS Solution

Navigate to the folder on your machine where the solution resides. Select and delete the project folders that are no longer needed.

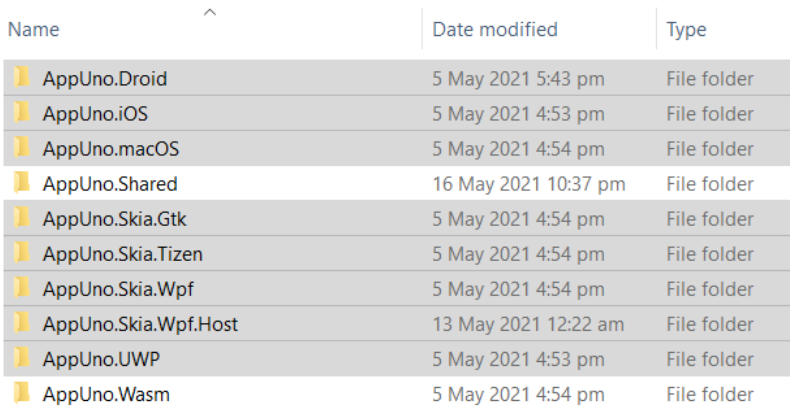

Figure 4-b: Project Folders to Remove

Now our solution is decluttered, and in principle, we are ready to reconstruct the single-page document tracker application.

You might have to restore the NuGet packages for each of the remaining projects, which can sometimes get tricky. Therefore, if you find yourself running into problems resolving dependencies that are no longer part of the solution, don't waste any precious time.

Make sure to fetch the decluttered **AppUno** solution GitHub repository I have already prepared—download and use it instead. If you do use it, click the **Download ZIP** option.

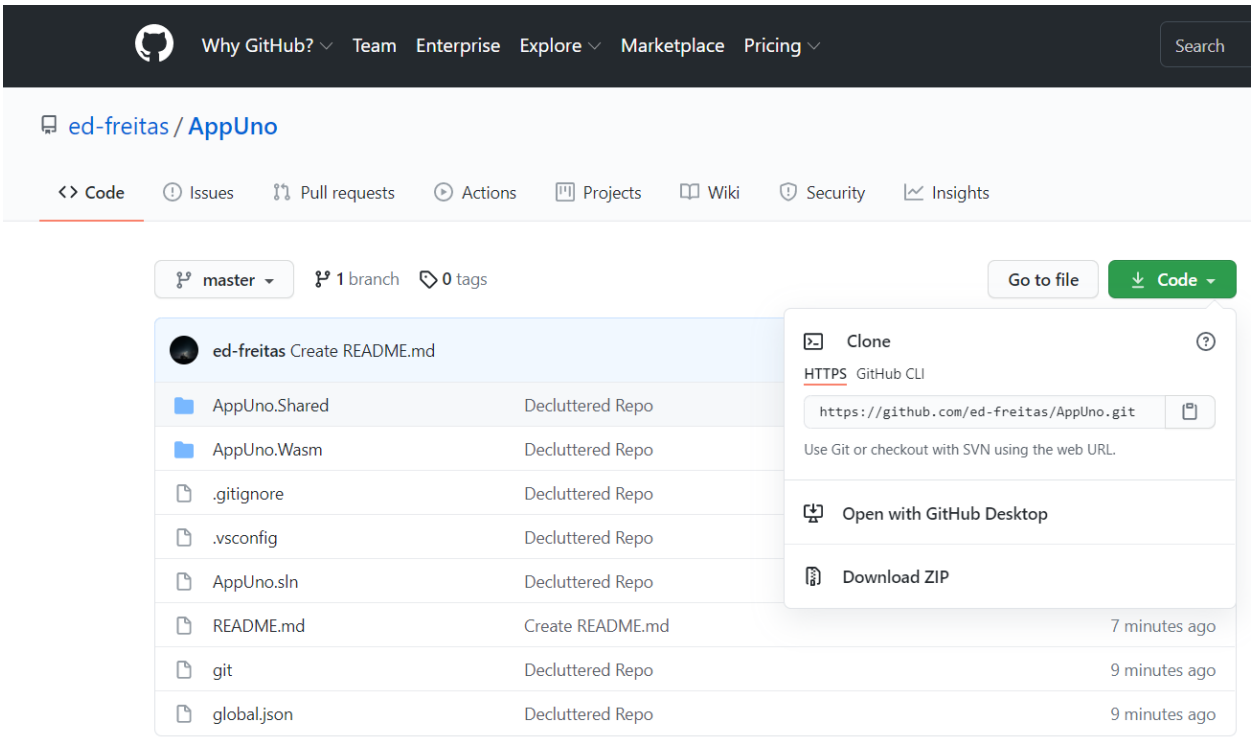

Figure 4-c: The Decluttered AppUno GitHub Repo

Data model

We will begin with the model and define the fields we want to track, such as **`id`**, **`title`**, and **`expiration`** date.

Within the **Solution Explorer**, select the **AppUno.Shared** project, right-click, click **Add** > **New Folder**, and name the folder **Model**.

Drag and drop the **Model.cs** file into the newly created folder. By doing this, we are keeping our code more structured and organized.

Open the **Model.cs** file, and let's update the existing model code we previously created—we can do this by removing the current **Model.cs** code and replacing it with the following.

Listing 4-a: Updated Model.cs File (soon to be renamed Doc.cs)

```
using System;
using System.Globalization;
using Windows.Foundation.Metadata;

namespace AppUno.Model
{
    [CreateFromString(
      MethodName = "AppUno.Model.ModelConverter.CreateDocFromString")]
    public class Doc : Observable
    {
        private int id;
        private string title = "Test";
        private string expiration =
          DateTime.Now.ToString("dd-MMM-yyyy",
          DateTimeFormatInfo.InvariantInfo);

        public int Id
        {
            get => id;
            set
            {
                id = value;
                NotifyPropertyChanged();
            }
        }

        public string Title
        {
            get => title;
            set
```

```
            {
                title = value;
                NotifyPropertyChanged();
            }
        }

        public string Expiration
        {
            get => expiration;
            set
            {
                expiration = value;
                NotifyPropertyChanged();
            }
        }
    }
}
```

Something to notice is that the model's namespace has been renamed from **AppUno** to **AppUno.Model**. Also, the class has been renamed from **Model** to **Doc**.

To keep consistency, let's rename the **Model.cs** file to **Doc.cs**, which we can do by right-clicking the file name in Solution Explorer. It's a good practice to have the model's class and the file that hosts it named the same.

The model structure remains the same, except that class is now called **Doc**, and it contains the variables **id**, **title**, and **expiration**, and the properties **Id**, **Title**, and **Expiration**.

The model now inherits from the **Observable** class (yet to be created), which inherits from the **INotifyPropertyChanged** interface.

Each of the properties of the model invokes the **PropertyChanged** event, but now through the **NotifyPropertyChanged** method.

Another difference is that the **title** and **expiration** variables have default values assigned. Those values are visible when a new document is added to the list using the application.

There is also a reference to the **CreateDocFromString** method (yet to be created), which adds a new document using a string instead of instantiating the **Doc** (model) class.

Observable

The **Observable** class is just a wrapper that inherits from the **INotifyPropertyChanged** interface and will handle raising the **PropertyChanged** event, so the code of the **Doc** class is easier to read and understand.

So, using **Solution Explorer**, right-click the **Model** folder, and select **Add** > **New Item**. The following window will appear.

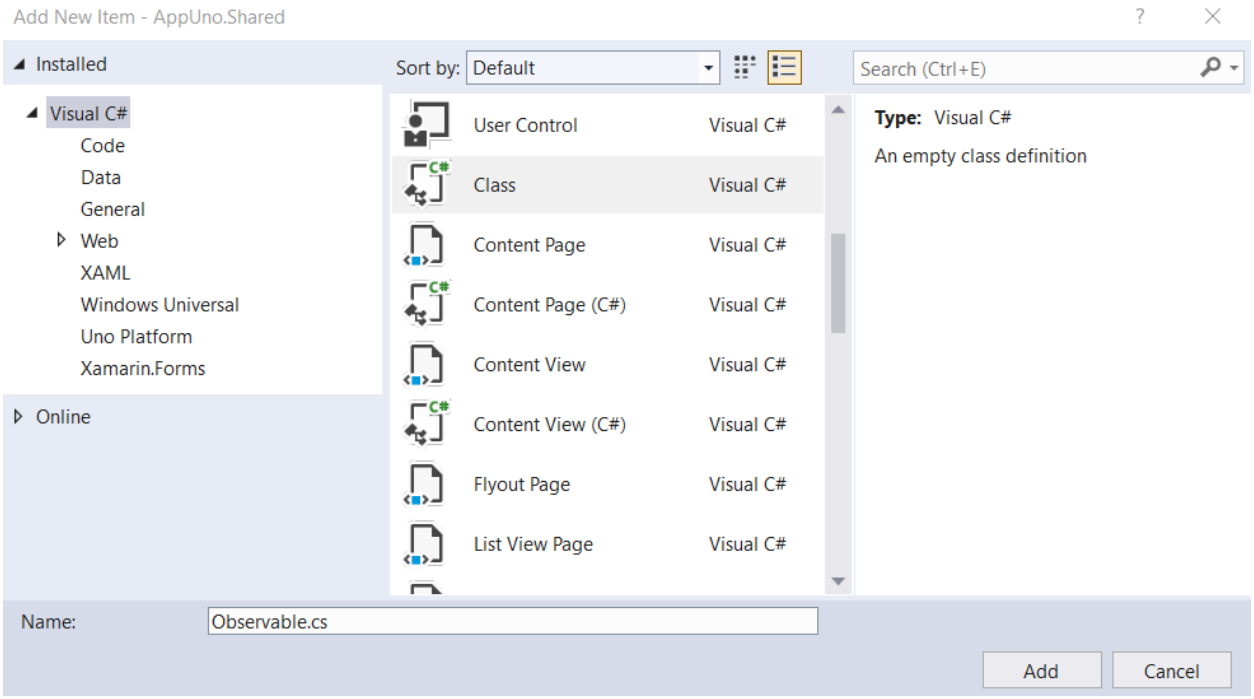

Figure 4-d: Add New Item – Observable.cs File

Once the file has been created, replace the default code with the following code.

Listing 4-b: Updated Observable.cs File

```
using System.ComponentModel;
using System.Runtime.CompilerServices;

namespace AppUno.Model
{
    public class Observable : INotifyPropertyChanged
    {
        public event PropertyChangedEventHandler PropertyChanged;

        protected virtual void NotifyPropertyChanged(
          [CallerMemberName] string propertyName = null)
        {
            PropertyChanged?.Invoke(this,
              new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

As we can see, the **Observable** class simply inherits from the **INotifyPropertyChanged** interface, and it declares the **PropertyChanged** event, which is what the model previously had.

The **Observable** class implements the **NotifyPropertyChanged** method, which is marked as **virtual**, allowing classes that inherit from **Observable** to override the method with a different implementation.

The **Doc** (model) and **Observable** classes belong to the same namespace (**AppUno.Model**); this doesn't require either of them to reference the other with a **using** statement.

If the method is not overridden in the descending class, then the inherited method implementation is used.

The implementation of the **NotifyPropertyChanged** method is straightforward. All it does is call the **Invoke** method of **PropertyChanged**, passing the name of the property being modified as a parameter of **PropertyChangedEventArgs**.

This logic is invoked within the setter of every model's property (as seen in the old **Model** class, **Model.cs**).

By implementing this logic instead within the **Observable** class, we can decouple and refactor this code.

Model converter

The **ModelConverter** class is used for creating a **Doc** instance using a string. We won't use it, but it is good to have it if you would like to make your application more dynamic in the future.

Using **Solution Explorer**, right-click the **Model** folder, and then click **Add** > **New Item**. The following window will appear.

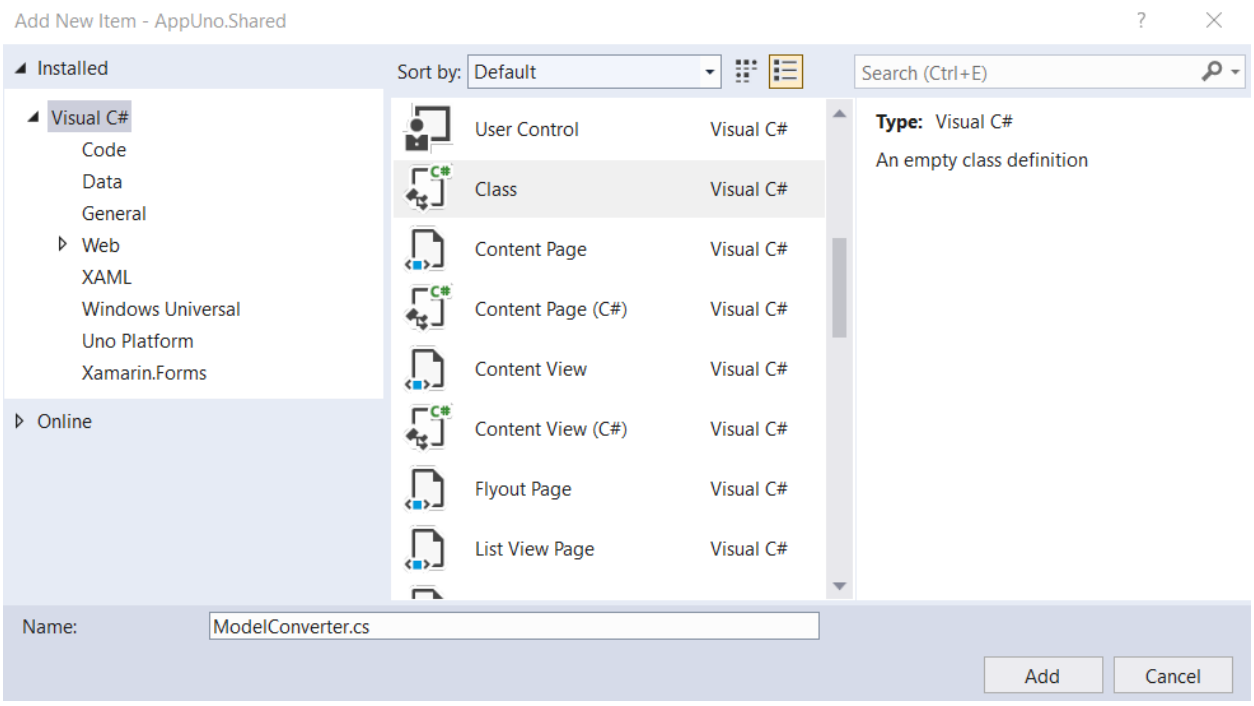

Figure 4-e: Add New Item – ModelConverter.cs

Name the file **ModelConverter.cs** and click **Add**. Once the file has been created, replace the default code with the following.

Listing 4-c: Updated ModelConveter.cs File

```
using System;

namespace AppUno.Model
{
  public static class ModelConverter
  {
    public static Doc CreateDocFromString(string str)
    {
      var values = str.Split(',');
      return new Doc
      {
        Id = Convert.ToInt32(values[0]),
        Title = values[1],
        Expiration = values[2]
      };
    }
  }
}
```

As we can see, the code is straightforward. The **`ModelConverter`** class is part of the same namespace as the **`Doc`** (model) and **`Observable`** classes—**`AppUno.Model`**—this makes it easy for this class to be referenced.

The **`CreateDocFromString`** method simply takes a string (**`str`**) as a parameter and splits its content, where each part (**`value[]`**) is assigned to its corresponding property within a **`Doc`** instance.

View model

During your developer career, it's very likely that you have heard of or come across the MVC pattern.

When developing Uno applications, we use a similar pattern to MVC called MVVM, where the view model plays an essential role. The view model acts as a glue between the data layer or model and the UI by using methods that can perform operations on the model.

Within the **Solution Explorer**, right-click the **Model** folder under the **AppUno.Shared** project and select **Add** > **New Item**. The following window will appear.

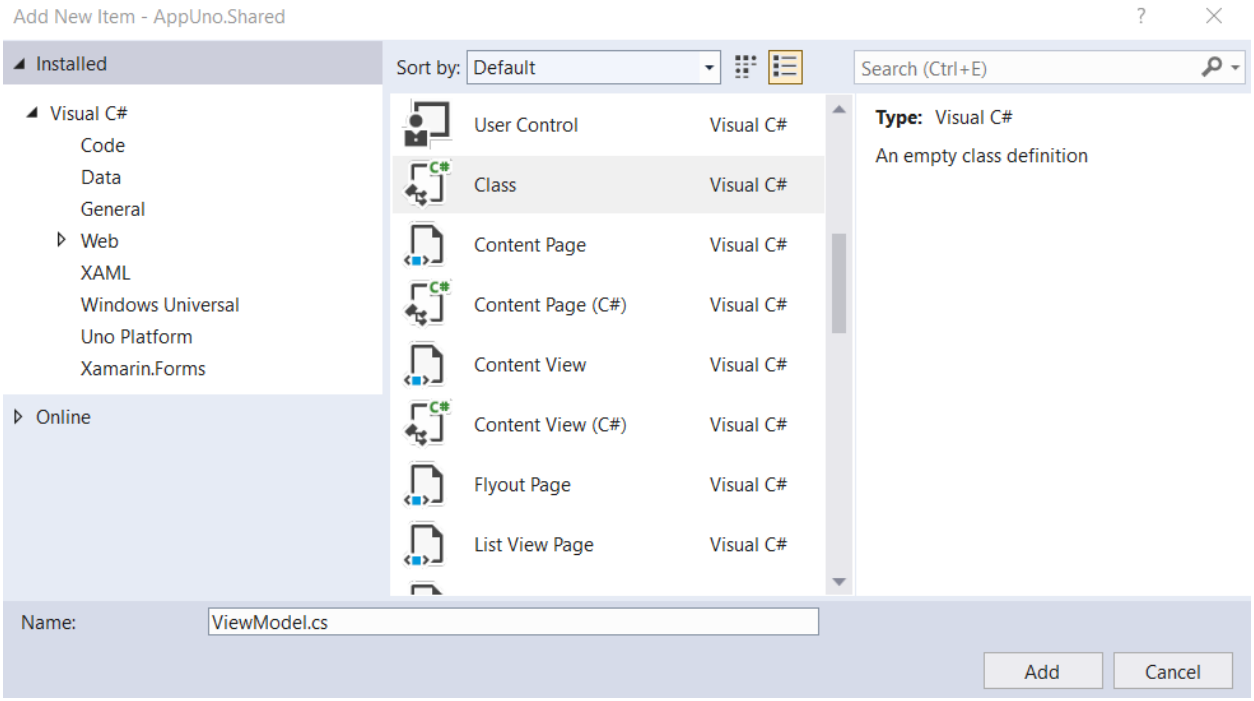

Figure 4-f: Add New Item – ViewModel.cs

Once the file has been created, it will appear under the **Model** folder of the **AppUno.Shared** project. Replace the existing code with the following.

Listing 4-d: Updated ViewModel.cs File

```
using System;
using System.Collections.ObjectModel;
using System.Globalization;

namespace AppUno.Model
{
    public class ViewModel : Observable
    {
        private Doc _selectedDoc;
        public bool IsDocSelected => SelectedDoc != null;
        public ObservableCollection<Doc> Docs { get; }

        public ViewModel()
        {
            string now = DateTime.Now.
              ToString("dd-MMM-yyyy", DateTimeFormatInfo.InvariantInfo);

            Docs = new ObservableCollection<Doc>{
              new Doc { Id=1, Title="ID Card", Expiration=now },
              new Doc { Id=2, Title="Passport", Expiration=now },
              new Doc { Id=3, Title="License", Expiration=now },
            };
        }

        public Doc SelectedDoc
        {
          get { return _selectedDoc; }
          set
          {
            if (_selectedDoc != value)
            {
              _selectedDoc = value;
              NotifyPropertyChanged();
              NotifyPropertyChanged(nameof(IsDocSelected));
            }
          }
        }

        public void AddDoc()
        {
            int id = new Random().Next();
            var doc = new Doc { Id = id };
            Docs.Add(doc);
```

```
            SelectedDoc = doc;
        }

        public void DeleteDoc()
        {
            var doc = SelectedDoc;
            if (doc != null)
            {
                Docs.Remove(doc);
                SelectedDoc = null;
            }
        }
    }
}
```

Let's go over what we have done here. The first thing to notice is that the **ViewModel** class also inherits from the **Observable** class, and both are part of the **AppUno.Model** namespace.

Within the **ViewModel** class, we find three variables. The first is the **_selectedDoc** object that indicates which is the currently selected document.

Next, we have the **IsDocSelected** variable, which indicates whether a particular document is selected.

Then, we find the **Docs** object, an **ObservableCollection** list of **Doc** instances. This list will contain all the documents that our application will display.

After that, we find the **ViewModel** constructor. We initialize the **now** object within the constructor, representing the current date and time, using the **dd-MMM-yyyy** format.

Following that, within the **ViewModel** constructor, we add three documents (each a **Doc** instance) to the **ObservableCollection**, using the following code:

```
Docs = new ObservableCollection<Doc>{
  new Doc { Id=1, Title="ID Card", Expiration=now },
  new Doc { Id=2, Title="Passport", Expiration=now },
  new Doc { Id=3, Title="License", Expiration=now }
};
```

Next, we find the **SelectedDoc** method, which returns the currently selected document within the application (**_selectedDoc**). This method also sets the value of the **_selectedDoc** object when the user selects another document, which triggers **NotifyPropertyChanged**, used for data binding.

The **AddDoc** method, as its name implies, is used by the application when a new document is added.

Each document (**doc**) has a unique identifier, an **id**, assigned a random integer (**int**) value. The new document is added to the **ObservableCollection** (**Docs**), and the document added is set as the selected one (**SelectedDoc**).

The **DeleteDoc** method, as its name implies, retrieves the value of the selected document (**SelectedDoc**). If the document's value is different from **null**, then the document is removed from the list.

As you can see, the **ViewModel** logic focuses on handling (adding, removing, and selecting) the documents within the list of documents—**ObservableCollection**.

App.xaml

Now that we have all the files in the **Model** folder ready, let's move our attention to the user interface. We'll begin with the **App.xaml** file.

Open the existing **App.xaml** file in the **AppUno.Shared** project and replace the current code with the following.

Listing 4-e: Updated App.xaml File

```
<Application
    x:Class="AppUno.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:AppUno">

</Application>
```

If you downloaded the decluttered **AppUno** solution template from GitHub, you probably noticed that the existing **App.xaml** code is identical to the preceding listing.

App.xaml.cs

Next, let's look at the **App.xaml.cs** (code-behind) file from the **AppUno.Shared** project. Replace the existing code with the following.

Listing 4-f: Updated App.xaml.cs File

```
using Microsoft.Extensions.Logging;
using System;
using Windows.ApplicationModel;
using Windows.ApplicationModel.Activation;
using Windows.UI.Xaml;
```

```
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Navigation;

namespace AppUno
{
    /// <summary>
    /// Provides application-
    /// specific behavior to supplement
    /// the default Application class.
    /// </summary>
    public sealed partial class App : Application
    {
        /// <summary>
        /// Initializes the singleton application object.
        /// This is the first line of authored code
        /// executed, and as such is the logical equivalent of
        /// main() or WinMain().
        /// </summary>
        public App()
        {
            ConfigureFilters(
            global::Uno.Extensions.LogExtensionPoint.AmbientLoggerFactory);

            this.InitializeComponent();
            this.Suspending += OnSuspending;
        }

        /// <summary>
        /// Invoked when the application is launched
        /// normally by the end user. Other entry points
        /// will be used such as when the application is launched
        /// to open a specific file.
        /// </summary>
        /// <param name="e">Details about the launch
        /// request and process.</param>
        protected override void OnLaunched(LaunchActivatedEventArgs e)
        {
            var window = Windows.UI.Xaml.Window.Current;

            Frame rootFrame = window.Content as Frame;

            // Do not repeat app initialization
            // when the window already has content,
            // just ensure that the window is active
```

```
            if (rootFrame == null)
            {
                // Create a Frame to act as the navigation
                // context and navigate to the first page
                rootFrame = new Frame();

                rootFrame.NavigationFailed += OnNavigationFailed;

                if (e.PreviousExecutionState ==
                    ApplicationExecutionState.Terminated)
                {
                    //TODO: Load state from previously
                    //suspended application
                }

                // Place the frame in the current Window
                window.Content = rootFrame;
            }

            if (e.PrelaunchActivated == false)
            {
                if (rootFrame.Content == null)
                {
                    // When the navigation stack isn't
                    // restored, navigate to the first page,
                    // configuring the new page by passing
                    // required information as a navigation
                    // parameter
                    rootFrame.Navigate(typeof(MainPage), e.Arguments);
                }
                // Ensure the current window is active
                window.Activate();
            }
        }

        /// <summary>
        /// Invoked when navigation to a certain page fails
        /// </summary>
        /// <param name="sender">The Frame that failed navigation</param>
        /// <param name="e">Details about the navigation failure</param>
        void OnNavigationFailed(object sender, NavigationFailedEventArgs e)
        {
            throw new Exception(
            $"Failed to load {e.SourcePageType.FullName}: {e.Exception}");
```

```
        }

        /// <summary>
        /// Invoked when application execution is being
        /// suspended. Application state is saved
        /// without knowing whether the application will be
        /// terminated or resumed with the contents
        /// of memory still intact.
        /// </summary>
        /// <param name="sender">The source of the suspend request.</param>
        /// <param name="e">Details about the suspend request.</param>
        private void OnSuspending(object sender, SuspendingEventArgs e)
        {
            var deferral = e.SuspendingOperation.GetDeferral();
            //TODO: Save application state and stop any background activity
            deferral.Complete();
        }


        /// <summary>
        /// Configures global logging
        /// </summary>
        /// <param name="factory"></param>
        static void ConfigureFilters(ILoggerFactory factory)
        {
        }
    }
}
```

Let's briefly go over this code—even though we can see many explanatory comments that should provide you with a good understanding.

The **App** class inherits from the **Application** class. The constructor method initializes the logger (what the **ConfigureFilters** method does), the components, and the **OnSuspending** event.

The **App** class overrides the **OnLaunched** method, which gets triggered when the application starts.

All it does is retrieve the **window** content where the application will run, then creates a **Frame** instance containing the rest of the controls that the application will use, and ensures it is active.

Then, we find the **OnNavigationFailed** method, which is invoked when navigation to a particular page fails.

After that, we find the **`OnSuspending`** method invoked when the application execution is being suspended.

As you can see, it looks like a lot of code, but since most of it consists of comments, it's actually not much at all.

MainPage.xaml

There's just one piece of this puzzle remaining, and that's the application's UI, which we need to write using XAML.

Let's open the **MainPage.xaml** file within the **AppUno.Shared** project and replace the existing code with the following.

Listing 4-g: Updated MainPage.xaml File

```
<Page x:Class="AppUno.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:model="using:AppUno.Model"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc=
    "http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    Background="{ThemeResource ApplicationPageBackgroundThemeBrush}"
    d:DesignWidth="600" d:DesignHeight="400">

  <Grid>
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="Auto"/>
      <ColumnDefinition/>
      <ColumnDefinition Width="Auto"/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
      <RowDefinition Height="Auto"/>
      <RowDefinition/>
    </Grid.RowDefinitions>

    <StackPanel Grid.ColumnSpan="2" Background="#4d74d2"
     Orientation="Horizontal">
      <TextBlock Text="Doc Tracker" FontSize="30" FontWeight="ExtraBold"
       Foreground="White" VerticalAlignment="Bottom" Margin="20"/>
      <Button Margin="10" Click="{x:Bind ViewModel.AddDoc}">
        <StackPanel Orientation="Horizontal">
          <SymbolIcon Symbol="Add"/>
```

```
        <TextBlock Text="Add" Margin="5 0 0 0"/>
      </StackPanel>
    </Button>
    <Button Margin="10" Click="{x:Bind ViewModel.DeleteDoc}"
     IsEnabled="{x:Bind ViewModel.IsDocSelected,Mode=OneWay}">
      <StackPanel Orientation="Horizontal">
        <SymbolIcon Symbol="Delete"/>
        <TextBlock Text="Delete" Margin="5 0 0 0"/>
      </StackPanel>
    </Button>
  </StackPanel>

  <Grid Grid.Row="1" x:Name="docListGrid" Background="#e5ebf8">
    <Grid.RowDefinitions>
      <RowDefinition Height="Auto"/>
      <RowDefinition/>
    </Grid.RowDefinitions>

    <ListView Grid.Row="1"
     ItemsSource="{x:Bind ViewModel.Docs,Mode=OneWay}"
     SelectedItem="{x:Bind ViewModel.SelectedDoc,Mode=TwoWay}">
      <ListView.ItemTemplate>
        <DataTemplate x:DataType="model:Doc">
          <StackPanel Orientation="Horizontal">
            <TextBlock Text="{x:Bind Title,Mode=OneWay}"
             FontWeight="Bold"/>
            <TextBlock Text="{x:Bind Expiration,Mode=OneWay}"
             Margin="5 0 0 0"/>
          </StackPanel>
        </DataTemplate>
      </ListView.ItemTemplate>
    </ListView>
  </Grid>

  <StackPanel Grid.Row="1" Grid.Column="1"
   Visibility="{x:Bind ViewModel.IsDocSelected,Mode=OneWay}">
    <TextBox Header="Id" Margin="10"
     Text="{x:Bind ViewModel.SelectedDoc.Id,Mode=TwoWay,
     UpdateSourceTrigger=PropertyChanged}"/>
    <TextBox Header="Title" Margin="10"
     Text="{x:Bind ViewModel.SelectedDoc.Title,Mode=TwoWay,
     UpdateSourceTrigger=PropertyChanged}"/>
    <TextBox Header="Expiration" Margin="10"
     Text="{x:Bind ViewModel.SelectedDoc.Expiration,Mode=TwoWay,
```

```
        UpdateSourceTrigger=PropertyChanged}" />
    </StackPanel>
  </Grid>
</Page>
```

There's quite a bit of XAML markup here. To understand this better, let's look at the following diagrams to have a visual representation that establishes the relationship between the markup and the finished UI.

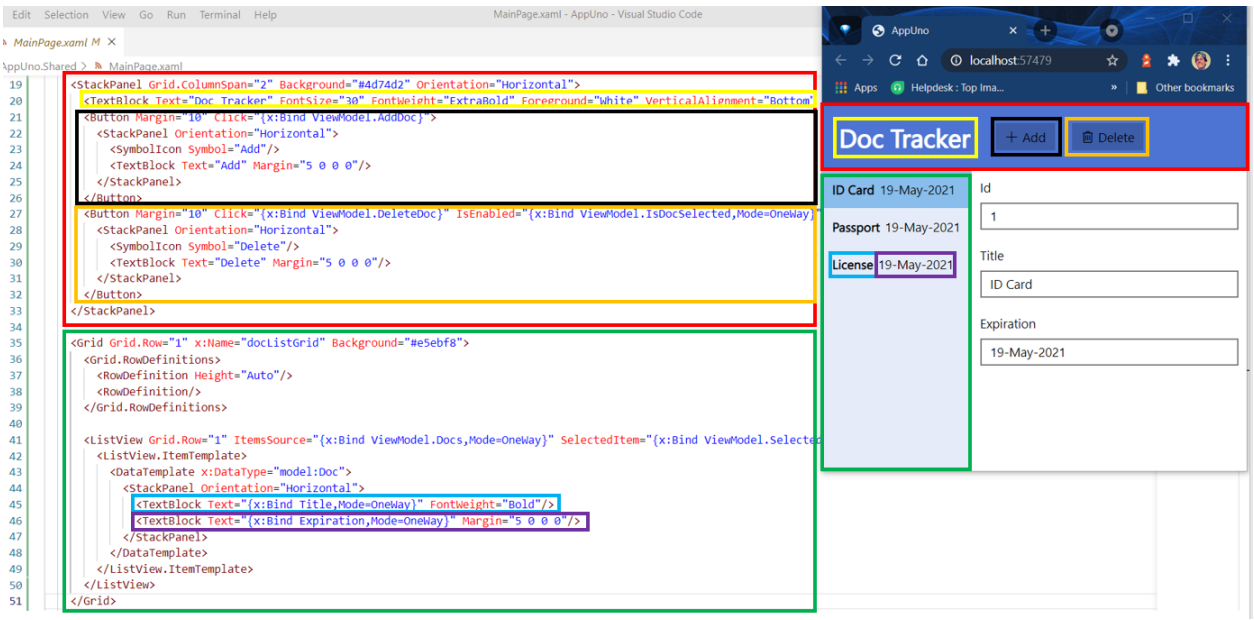

Figure 4-g: Relationship Between the XAML Markup and Finished UI (Part 1)

We can understand the relationship by looking at the XAML and UI and matching the exact colors.

The red part corresponds to the **`StackPanel`** component that contains the application's title and each of the buttons.

The light-yellow corresponds to the application's title (**`TextBlock`**), the black area corresponds to the **Add `Button`** component, and the dark-yellow corresponds to the **Delete `Button`** component.

The green area corresponds to the left-hand side panel (**`Grid`**) that contains the list of documents (**`ListView`**) to track. The blue area corresponds to the **`TextBlock`** that indicates the document's name, whereas the purple area (a **`TextBlock`**) corresponds to the date the document expires.

From the relationship described in the preceding diagram, we can see that all the UI parts except the white area (right-hand side of the application's screen) have been matched to the XAML markup. The reason is that this part of the code wouldn't fit in the preceding screenshot—so, let's look at that last bit.

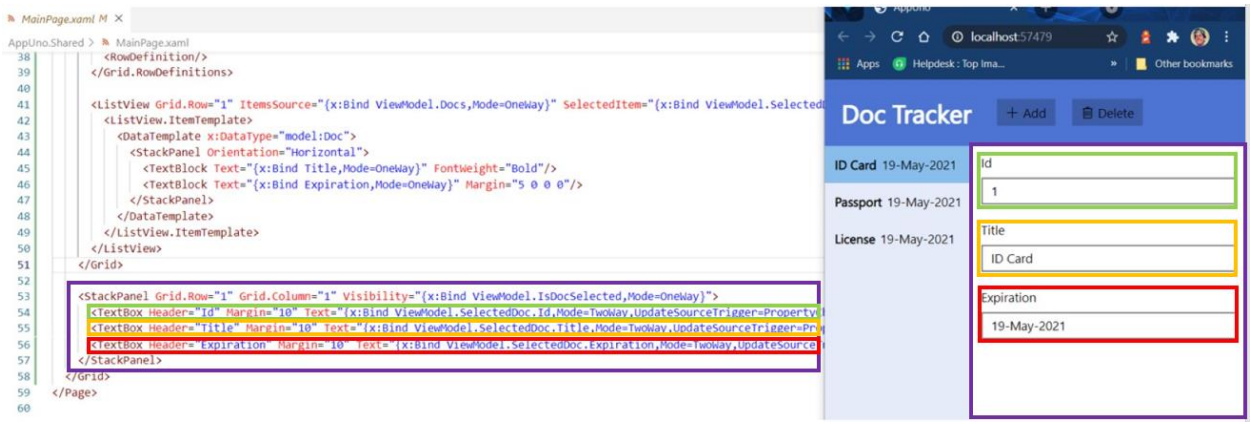

Figure 4-h: Relationship Between the XAML Markup and Finished UI (Part 2)

The preceding diagram shows that the purple area corresponds to the **`StackPanel`** that contains all the fields.

Then, we have the green area that corresponds to the **Id** field (**`TextBlock`**). We can also see the dark-yellow section that corresponds to the **Title** field (**`TextBlock`**).

Finally, the red area corresponds to the **Expiration** field (**`TextBlock`**). Now that we know how the markup relates to the finished UI, let's look at the bindings.

MainPage bindings

Having the UI ready is just one part of the story—the application works due to the bindings between the components and the model.

The first binding we have is for the **Add** button, which binds to the **`AddDoc`** method of the **`ViewModel`**. When the button is clicked, the **`AddDoc`** method is triggered; this is a **`OneWay`** binding operation as it is just used for executing the event.

The second binding we find is for the **Delete** button, which binds to the **`DeleteDoc`** method of the **`ViewModel`**. When the button is clicked, the **`DeleteDoc`** method is triggered; this is also a **`OneWay`** binding operation as it is just used for executing the event.

Something interesting to notice is that by default the **Delete** button is disabled, and only becomes enabled when one of the items from the **`ListView`** component is selected. This is because the button has a property called **`IsSelected`**, which binds to **`IsDocSelected`** of the **`ViewModel`**.

Here's an example of how the application looks when the **Delete** button is disabled by default. In this case, none of the items of the **`ListView`** component are selected.

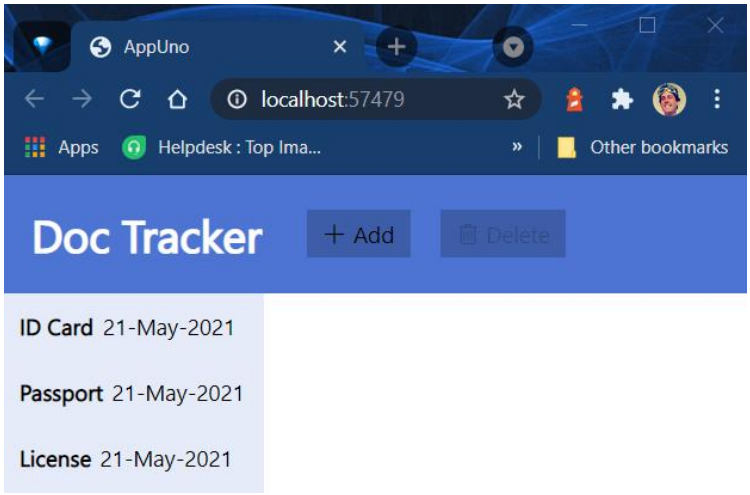

Figure 4-i: The App Running (Delete Button Disabled)

This behavior makes sense because to delete an item, you need to have an element selected. After an item has been selected, the application looks as follows.

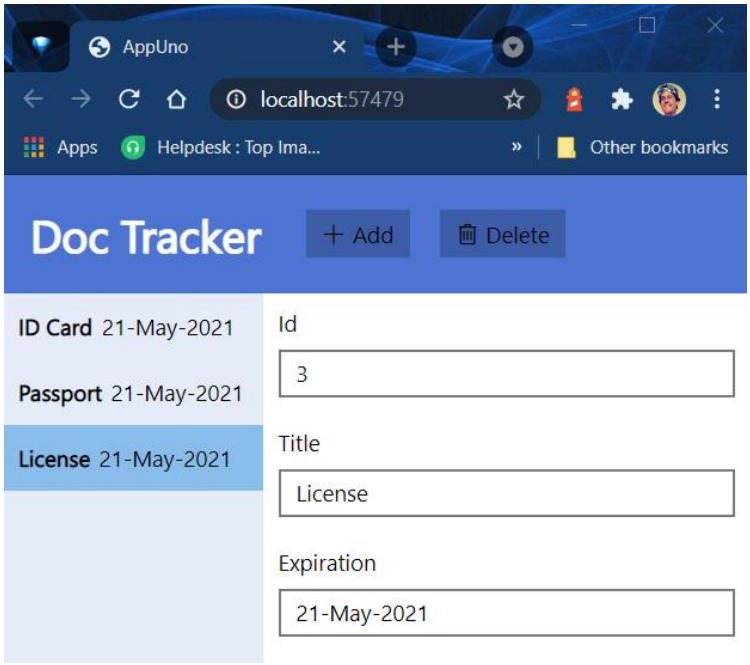

Figure 4-j: The App Running (The Delete Button Enabled)

As expected, when one item has been selected, the **Delete** button is enabled—that's the magic of data binding.

Next, we have the **`ListView`** component, which has two bindings that are indispensable. The first data binding provides a way for the **`ListView`** to load its various child elements—this is a **`OneWay`** binding (**`ViewModel.Docs`**). Its sole purpose is to retrieve data items from the model.

The second binding (**ViewModel.SelectedDoc**) of the **ListView** component is responsible for selecting an item. This is a **TwoWay** binding, which not only selects an object, but also updates the model's corresponding value.

Within the **ListView** component, we have a **TextBlock** component that displays the **Title** property, and another **TextBlock** component that displays the item's **Expiration** property.

Each of these fields uses **OneWay** binding because the binding is only used to retrieve (read) these values.

Finally, we have the three **TextBox** components that contain the values of each of the items of the **ListView**; these are contained within a **StackPanel**.

These three components use **TwoWay** binding, meaning that the values can be read and written to the model.

The first **TextBox** component (**Id**) binds to the property with the same name through **ViewModel.SelectedDoc**—which leads to the currently selected item from the **ListView**.

The second **TextBox** component (**Title**) binds to the property with the same name, also through **ViewModel.SelectedDoc**—which also leads to the currently selected item from the **ListView**.

Finally, the third **TextBox** component (**Expiration**) binds to the property with the same name, also through **ViewModel.SelectedDoc**—which also leads to the currently selected item from the **ListView**.

As you might have noticed, these **TextBox** components all have their **UpdateSourceTrigger** attribute set to the **PropertyChanged** method—which executes when the value of the **TextBox** changes.

MainPage.xaml.cs

Now that we have explored the XAML markup and bindings used in the **MainPage.xaml** file, let's have a quick look at the code-behind file. As you'll see, it's straightforward.

Listing 4-h: MainPage.xaml.cs File

```
using Windows.UI.Xaml.Controls;

using AppUno.Model;

namespace AppUno
{
    public sealed partial class MainPage : Page
    {
        public ViewModel ViewModel { get; }
```

```
        public MainPage()
        {
            this.InitializeComponent();
            ViewModel = new ViewModel();
        }
    }
}
```

All we have is a class called **`MainPage`**, which inherits from **`Page`**. This class contains a **`ViewModel`** property.

The constructor method creates an instance of the **`ViewModel`** class, and this **`ViewModel`** instance makes the binding work.

Project references

The final part is to look at the references required for the solution to run—specifically, the references of the **AppUno.Wasm** project.

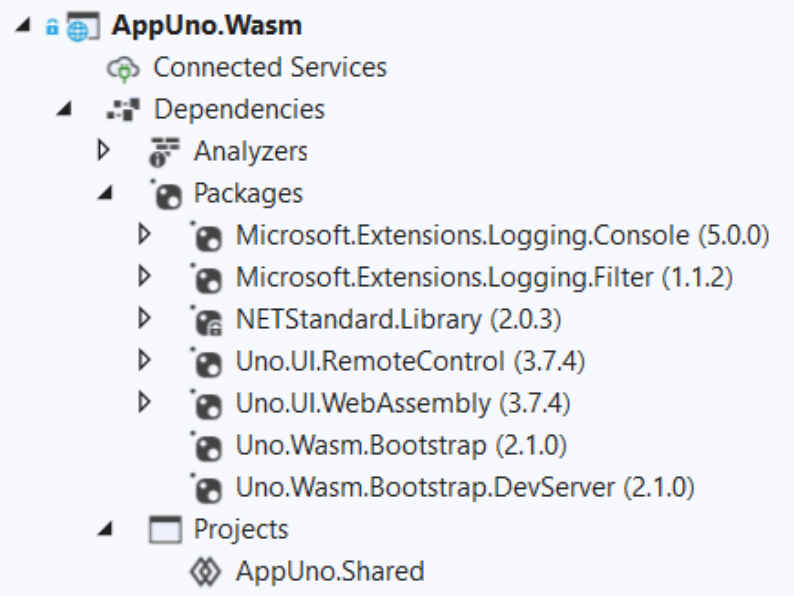

Figure 4-k: The AppUno.Wasm Project References

You'll find the **Manage NuGet Packages** option by right-clicking **Packages**. This displays the window shown in Figure 4-l.

The main packages used by the project are Uno.UI.RemoteControl, Uno.UI.WebAssembly, Uno.Wasm.Bootstrap, Uno.Wasm.Bootstrap.DevServer, and NETStandard.Library.

If you don’t have all these packages installed within your project, click the **Browse** tab as seen in the screenshot that follows, and enter the library's name in the **Search** box.

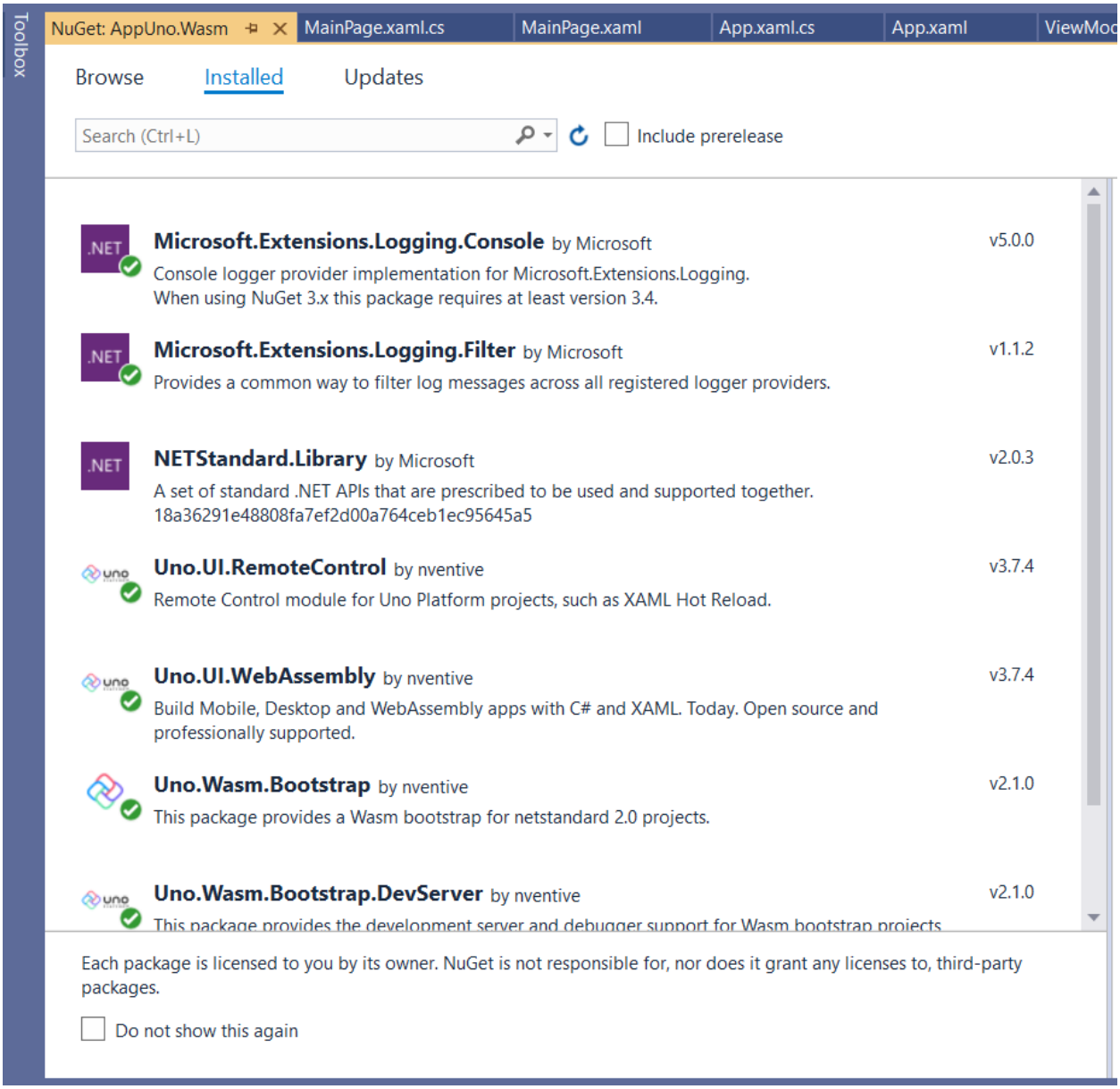

Figure 4-l: The NuGet Packages Window

Once you have these packages installed, you should be ready to run your application. Let’s go ahead and build the project.

Go to the **Solution Explorer**, right-click the solution, and click **Build Solution**. Hopefully, you should not see any compilation or building errors.

Tip: At this stage, if you run into compilation or building errors, you can download the finished repository.

If you execute the application, you should see a screen similar to the following one. First, you'll see the application loading.

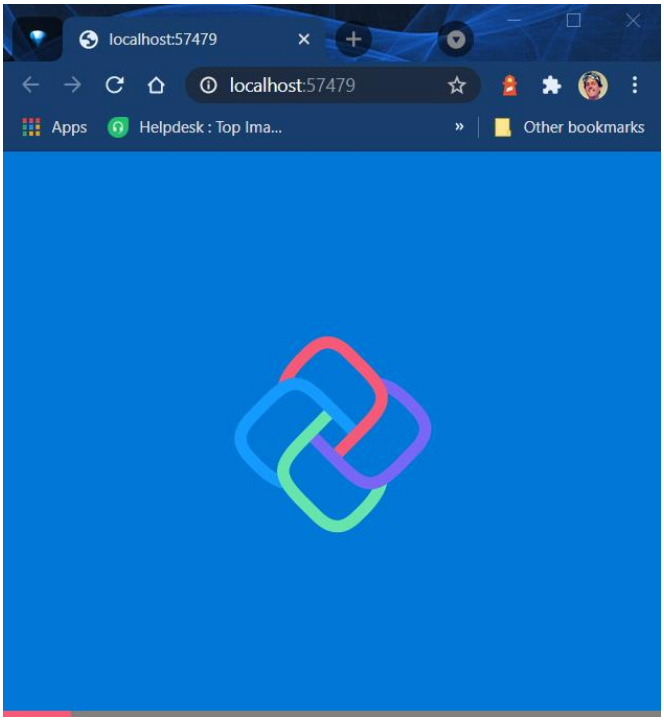

Figure 4-m: The Finished App Loading

Then, after the application loads, you'll see the application running.

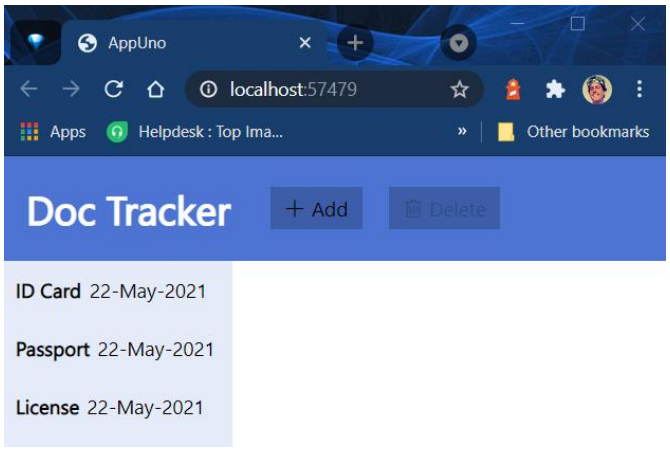

Figure 4-n: The Finished App Running

Excellent—you've done a great job! We have managed to reconstruct an Uno application, learn about its composition and parts, and made some adjustments along the way.

By default, when the application loads, the loading splash screen that Uno uses is white. However, we can see the loading screen's color is blue in the previous screenshots. You might have noticed this if you didn't download and use the decluttered repository.

The reason is that I specifically changed the color of the loading splash screen from white to blue. I was able to do this by editing the **AppManifest.js** file found under the **WasmScripts** folder of the **AppUno.Wasm** project.

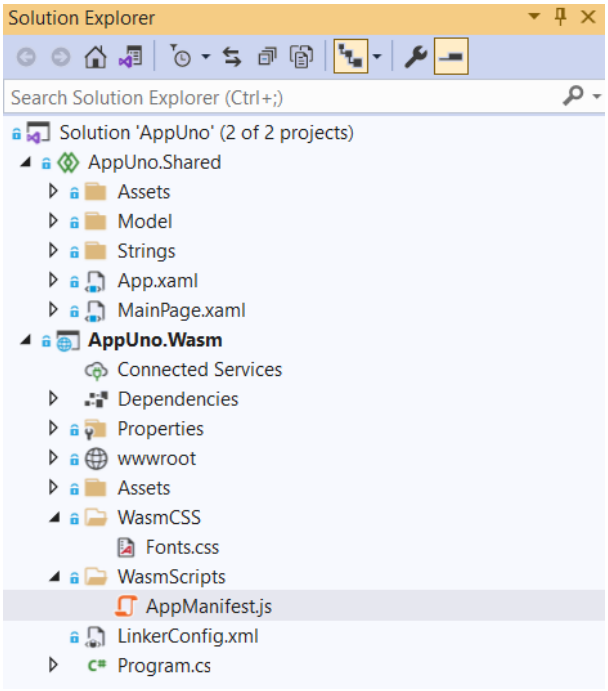

Figure 4-o: The AppManifest.js File

If you open this file, you should see the following content. The attribute used to change the loading screen color is **splashScreenColor**.

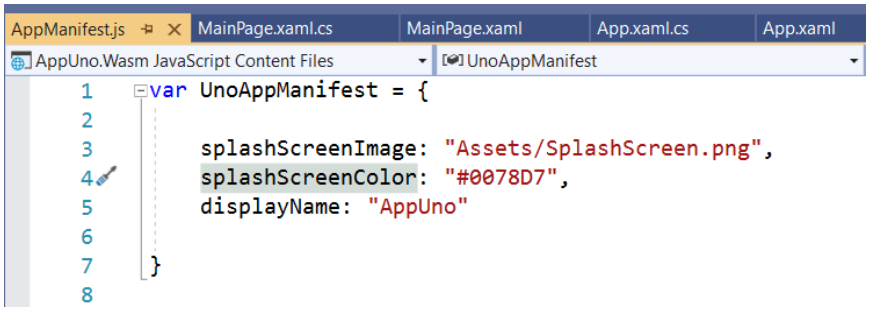

Figure 4-p: The splashScreenColor Attribute

You can change the value of this attribute to set your custom loading color. You can use any hexadecimal color code value.

Summary

We've reached the end of the chapter—and also this book. The Uno Platform is an excellent framework for creating cross-platform applications. It adds a layer of abstraction that allows code written with XAML and C# to target multiple platforms rather seamlessly.

It's essential to have some good knowledge of XAML and C# before jumping into Uno. If you have worked with HTML, it should be relatively easy to pick up XAML and get acquainted with it.

On the other hand, if you don't have a solid foundation for creating user interfaces using a markup language, it's best to get some grounding with XAML before diving into Uno.

If you come from a WPF or UWP development background, you will be used to XAML, and you'll feel right at home with Uno.

One of the critical aspects of working with Uno is writing as much shared code as possible. Using shared code is the key to targeting multiple platforms without reimplementing the same or similar application functionalities across different platforms.

Cross-platform development using the same code base has been the holy grail of software development for a long time, and during this time, many platforms have come and gone. Many others have promised instant cross-platform glory.

Uno seems to be onto something, and from the platforms I have tried, tested, and used, it has the broadest range of cross-platform compatibility I've come across so far—spanning between desktop, mobile, web, and other platforms.

Whether the cross-platform promise will be finally fulfilled with Uno and become ubiquitous is yet to be seen. Still, one thing is sure: they have done a fantastic job, and are on a great path to be one of those platforms that hopefully will remain in developers' lives for years to come.

I hope you have enjoyed this book, and that it has given you some foundations and insight on this technology. I invite you to continue to explore it. Until next time—take care, and do great things!