

# Learn WinUI 3.0

---

Leverage the power of WinUI, the future of native Windows application development

Alvin Ashcraft



# Learn WinUI 3.0

Leverage the power of WinUI, the future of native Windows application development

**Alvin Ashcraft**

**Packt**

BIRMINGHAM—MUMBAI

# Learn WinUI 3.0

Copyright © 2021 Packt Publishing

*All rights reserved.* No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

**Group Product Manager:** Aaron Lazar

**Publishing Product Manager:** Denim Pinto

**Senior Editor:** Nitee Shetty

**Content Development Editor:** Ruvika Rao

**Technical Editor:** Rashmi Subhash Choudhari

**Copy Editor:** Safis Editing

**Project Coordinator:** Deeksha Thakkar

**Proofreader:** Safis Editing

**Indexer:** Priyanka Dhadke

**Production Designer:** Alishon Mendonca

First published: March 2021

Production reference: 1260321

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-80020-866-7

[www.packt.com](http://www.packt.com)

*To my wife, Stelene, and to my three daughters, Anna, Luci, and Rubi, for their patience and for supporting me through the writing process. Thank you to the WinUI development team for your openness and communication with the Windows developer community as WinUI 3 evolved and matured.*

# Contributors

## About the author

**Alvin Ashcraft** is a software engineer and developer community champion with over 25 years of experience in software development. Working primarily with Microsoft Windows, web, and cloud technologies, his career has focused primarily on the healthcare industry. He has been awarded as a Microsoft MVP 11 times, most recently as a Windows Dev MVP.

Alvin works in the Philadelphia area for Allscripts, a global healthcare software company, as a principal software engineer. He is also a board member of the TechBash Foundation, where he helps organize the annual TechBash developer conference. He has previously worked for companies such as Oracle, Genzeon, CSC, and ITG Pathfinders.

Originally from the Allentown, PA area, Alvin currently resides in West Grove, PA with his wife and three daughters.

*I want to thank the people who have been close to me and supported me,  
especially my wife, Stelene, and my three daughters.*

## About the reviewer

**Nick Randolph** currently runs Built to Roam, which focuses on building rich mobile applications. Nick has been identified as a Microsoft MVP in recognition of his work with and expertise in Microsoft application platforms.

Nick has been invited to present at a variety of events, including Tech Ed and Ignite Australia & NZ, DDD, NDC, and local user groups. He has also authored multiple books on Visual Studio and Windows development and helped judge multiple world finals for the Imagine Cup. Nick has worked on numerous mobile applications and has helped hundreds of developers build their own mobile applications. Nick has been involved with applications for well-known brands such as Domain.com.au, ninemsn, AFL, NRL, Qantas, JB Hi-Fi, NAB, Stan, and Boost Juice.



# Table of Contents

## Preface

---

## Section 1: Introduction to WinUI and Windows Applications

### 1

#### Introduction to WinUI

---

Technical requirements	4	What is WinUI?	21
Before UWP – Windows 8 XAML applications	5	The first WinUI release	22
		The road to WinUI 3.0	23
Windows application UI design	6	What's new in WinUI 3.0?	26
Windows Runtime (WinRT)	6	Goodbye UWP?	26
User backlash and the path forward to Windows 10	7	New features for WinUI 3.0	28
		Project Reunion and WinUI	28
Windows 10 and UWP application development	7	WinUI compared to other Windows development frameworks	29
Language choice with UWP development	8	WinUI versus UWP	29
Lifting app restrictions	9	WinUI versus WPF	29
UWP backward compatibility	10	WinUI versus Windows Forms (WinForms)	31
What is XAML?	11	Summary	31
Creating an adaptive UI for any device	14	Questions	32
Powerful data binding	15		
Styling your UI with XAML	18		
Separating presentation from business logic	20		

## 2

### Configuring the Development Environment and Creating the Project

---

Technical requirements	34	XAML basics	46
Installing Visual Studio and Windows development workloads	35	Building the model	46
Adding the WinUI app templates	36	Creating sample data	50
Introducing the application idea	38	Building the initial UI	52
Reviewing the application features	38	Completing the data-binding initialization	53
WinUI in UWP versus WinUI in Desktop projects	38	Creating the DataTemplate and binding the UI	55
Creating your first WinUI project	39	Understanding WinUI and UWP	57
Anatomy of a WinUI in UWP project	42	Understanding the UWP app model	59
Reviewing App.xaml	42	Working with WinUI controls, properties, and events	60
Reviewing App.xaml.cs	43	Adding a ListView header	61
Reviewing MainPage.xaml	44	Creating the ComboBox filter	62
Reviewing MainPage.xaml.cs	45	Adding a new item button	68
Reviewing the project references	45	Summary	71
Reviewing the project properties	45	Questions	71

## 3

### MVVM for Maintainability and Testability

---

Technical requirements	74	Choosing a framework for WinUI applications	77
Understanding MVVM	74	Understanding data binding in WinUI	77
MVVM – the big picture	75	What are markup extensions?	78
MVVM libraries for WinUI	76	Binding markup extension	78
Windows Community Toolkit MVVM library	76	x:Bind markup extension	79
Prism Library	76	Updating View data with INotifyPropertyChanged	80
MVVMCross	77		

Updating collection data with INotifyCollectionChanged	81	Implementing ICommand	86
Implementing MVVM in WinUI applications	82	Using commands in the ViewModel	88
Working with events and commands	86	Updating the View	90
		Choosing a unit test framework	92
		Summary	92
		Questions	93

## 4

### Advanced MVVM Concepts

---

Technical requirements	96	Creating a data service	109
Understanding the basics of DI	96	Increasing maintainability by consuming services	113
Using DI with ViewModel classes	97	Handling parameters in ItemDetailsPage	116
Leveraging x:Bind with events	100	Creating the ItemDetailsViewModel class	116
Page navigation with MVVM and DI	102	Summary	121
Adding ItemDetailsPage	102	Questions	121
Adding new interfaces and services	106		
Creating a navigation service	106		

## 5

### Exploring WinUI Controls

---

Technical requirements	124	Reviewing what's new in WinUI 3.0	133
Understanding what WinUI offers developers	124	Backward compatibility	134
Animated visual player (Lottie)	125	Visual Studio tooling	134
Navigation view	126	Input validation	134
Parallax view	128	A new WebView	135
Rating control	129		
Two-pane view	129	Exploring the XamlDirect APIs for	
		middleware authors	137
Exploring the XAML Controls Gallery Windows app	130	Adding some new controls to the project	139
Learning about the ScrollViewer control	132		

Using the SplitButton control	140	Summary	144
Adding a TeachingTip to the save button	142	Questions	145
		Further reading	145

## 6

### Leveraging Data and Services

---

Technical requirements	148	Leveraging a Micro ORM to simplify data access	158
Managing application state with app life cycle events	148	Adding Dapper to the project	159
Exploring Windows application life cycle events	148	Updating the data service's initialization	163
Life cycle events of WinUI applications	149		
Creating a SQLite data store	154	Retrieving data via services	166
What is SQLite?	154	Performing data validation with MVVM	172
Adding SQLite to DataService	154	Summary	174
		Questions	175

## Section 2: Extending WinUI and Modernizing Applications

## 7

### Fluent Design System for Windows Applications

---

Technical requirements	180	Incorporating Fluent Design in WinUI applications	186
What is the Fluent Design System?	180	Updating the title bar	186
Exploring Fluent Design for Windows Controls	181	Changing the style of MainPage	188
Patterns	182	Changing the style of ItemDetailsPage	193
Layout	183		
Input	184	Using the Fluent XAML Theme Editor	194
Style	184	Colors	196
	185	Shapes	197

---

Using the UWP Resources		for Fluent Design	201
Gallery	199	Summary	202
Design resources and toolkits		Questions	202

**8****Building WinUI Apps with .NET 5**


---

Technical requirements	204	Exploring the Packaging project	211
Creating a WinUI project with .NET 5	204	Visual assets in the Manifest Designer	213
What is WinUI in Desktop?	205	Referencing .NET 5 Libraries from your project	215
Creating a new WinUI in Desktop project	206	Sharing the .NET 5 library with a WPF application	218
Exploring the Desktop project structure	208	Creating a WinUI control library	223
Adding the WebView2 control	209	Summary	227
		Questions	228

**9****Enhancing Applications with the Windows Community Toolkit**


---

Technical requirements	230	Referencing the WCT packages	244
Introducing the WCT	230	Adding data to the DataGrid	244
Origins of the WCT	231	Adding controls to the MainWindow	247
Reviewing recent toolkit releases	231	<b>Exploring the toolkit's helpers, services, and extensions</b>	250
Exploring the Windows Community Toolkit Sample App	232	Helpers	250
Installing and launching the sample app	233	Services	252
Controls	234	MVVM	254
WPF and WinForms controls	238	Extensions	254
Using controls from the toolkit	240	<b>Summary</b>	255
Creating the WinUI in Desktop project	241	<b>Questions</b>	256

## 10

### Modernizing Existing Win32 Applications with XAML Islands

---

Technical requirements	258	Using the UWP MapControl in WPF	273
What is XAML Islands?	259	Using the WebViewCompatible browser control in WPF	278
Modernizing a WinForms application with XAML Islands	261	Working with the WebView2 browser control in WinForms	281
Creating a shared class library project	261	Summary	283
Creating the WinForms host project	266	Questions	284
Modernizing a WPF application with XAML Islands	270		

## Section 3: Build and Deploy on Windows and Beyond

## 11

### Debugging WinUI Applications with Visual Studio

---

Technical requirements	288	Using the XAML Binding Failures window	304
Debugging in Visual Studio	289	Debugging live data with Live Visual Tree and Live Property Explorer	306
Debugging local applications	289	Coding with XAML Hot Reload	307
Debugging remote applications	295	Debugging with Live Visual Tree and Live Property Explorer	308
Common XAML layout mistakes	297	Summary	313
Improving your XAML with static code analysis	299	Questions	313
Learning to pinpoint data binding errors	301		
Common mistakes in data binding	301		

## 12

### Hosting an ASP.NET Core Blazor Application in WinUI

---

Technical requirements	316	Exploring some history of ASP.NET and ASP.NET Core	316
Getting started with ASP.NET Core and Blazor	316	What is Blazor?	318

WebAssembly and client-side .NET development	319	<b>Publishing Blazor to Azure Static Web Apps hosting</b>	<b>330</b>
<b>Creating a Blazor Wasm application</b>	<b>320</b>	Pushing the project to GitHub	330
Building a simple application for tracking tasks	323	Creating an Azure Static Web Apps resource	333
<b>Exploring Blazor Wasm deployment options</b>	<b>328</b>	Publishing an application with GitHub Actions	336
Deployment options for Blazor Wasm projects	328	<b>Hosting your Blazor application in the WinUI WebView2</b>	<b>338</b>
		<b>Summary</b>	<b>340</b>
		<b>Questions</b>	<b>340</b>

## 13

### **Building, Releasing, and Monitoring Applications with Visual Studio App Center**

---

Technical requirements	342	<b>App Center</b>	<b>354</b>
<b>Getting started with Visual Studio App Center</b>	<b>342</b>	Creating early releases of your application or beta testers	354
Creating an App Center account	343	<b>Application monitoring and analytics</b>	<b>357</b>
Creating your first App Center application	347	Instrumenting your code	358
<b>Setting up builds in App Center</b>	<b>349</b>	<b>Gathering and analyzing App Center crash reports</b>	<b>361</b>
Integrating App Center with a GitHub repository	351	<b>Summary</b>	<b>364</b>
<b>Deploying your application with</b>		<b>Questions</b>	<b>364</b>

## 14

### **Packaging and Deploying WinUI Applications**

---

Technical requirements	366	<b>Getting started with application packaging in Visual Studio</b>	<b>371</b>
<b>Discovering application packaging and MSIX basics</b>	<b>366</b>	<b>Deploying applications with Windows Package Manager</b>	<b>376</b>
What is MSIX?	367	Adding a package to the community repository	376
Reviewing MSIX tools and resources	370		

Using WinGet for package management	379		
<b>Distributing applications with the Microsoft Store</b>	<b>381</b>	<b>Sideload WinUI applications with MSIX</b>	<b>390</b>
Preparing a free application for the Microsoft Store	382	Creating an MSIX package for sideloading	390
Uploading a package to the Store	386	Sideload an MSIX package	392
		<b>Summary</b>	<b>395</b>
		<b>Questions</b>	<b>395</b>

## **Assessments**

---

Chapter 1	397	Chapter 9	400
Chapter 2	397	Chapter 10	400
Chapter 3	398	Chapter 11	400
Chapter 4	398	Chapter 12	401
Chapter 5	398	Chapter 13	401
Chapter 6	399	Chapter 14	401
Chapter 7	399	Why subscribe?	403
Chapter 8	399		

---

## **Other Books You May Enjoy**

---

## **Index**

---

# Preface

WinUI 3.0 is Microsoft's first step toward a unified Windows development platform. This unification effort is called Project Reunion and is an attempt to bring UWP, WPF, and other desktop UI frameworks to one platform. WinUI allows developers to quickly build Windows applications with styles that adapt themselves to the platform. As the WinUI platform matures, developers will have the ability to target desktop Windows machines, Xbox, HoloLens, Surface Hub, and more.

Win32 application developers can also leverage WinUI controls to modernize their existing applications by using XAML Islands controls from the Windows Community Toolkit. This open source toolkit provides dozens of controls and other helper libraries for WinUI, UWP, and Win32 application developers. You will learn how to find the right controls for your applications and share them across multiple projects.

In this book, you will learn how to develop, debug, build, and deploy applications using Visual Studio and cloud tools from Microsoft Azure and GitHub. You will discover deployment options to get your WinUI application into the hands of consumer and enterprise Windows users. By the end of this book, you will have a foundational understanding of how to create, modernize, and distribute Windows applications with WinUI 3.0.

## Who this book is for

This book is for anyone who wants to develop Windows 10 applications with WinUI 3.0. Readers who are familiar with UWP, WPF, and WinForms will also find this book useful to deepen their knowledge of Windows development and to modernize their existing applications. To benefit from this book, some familiarity with C# and .NET is expected but no previous knowledge of WinUI, UWP, or XAML UI development is assumed.

## What this book covers

*Chapter 1, Introduction to WinUI*, examines the history of UI frameworks in Windows and the origins of WinUI. It also walks you through creating your very first WinUI 3.0 project in Visual Studio 2019.

*Chapter 2, Configuring the Development Environment and Creating the Project*, introduces you to the project used throughout much of the book. The chapter explains the anatomy of a WinUI project and covers platform basics including the application life cycle and data binding.

*Chapter 3, MVVM for Maintainability and Testability*, explains the **Model-View-ViewModel (MVVM)** pattern, a design pattern that is fundamental to WinUI application development. You will also learn the basics of unit testing WinUI projects in this chapter.

*Chapter 4, Advanced MVVM Concepts*, continues to examine MVVM concepts in WinUI by adding a **dependency injection (DI)** NuGet package and service classes for page navigation and data retrieval.

*Chapter 5, Exploring WinUI Controls*, explores the controls available in WinUI 3.0 by introducing you to the XAML Controls Gallery application. This open source application helps developers discover controls with sample code that can be used in WinUI projects. You will then add a couple of new controls from the gallery to your project.

*Chapter 6, Leveraging Data and Services*, expands upon the data services created in *Chapter 4, Advanced MVVM Concepts*, to add a SQLite data store for persisting user data in the sample application. Data validation concepts will also be introduced to ensure invalid data cannot be entered by users.

*Chapter 7, Fluent Design System for Windows Applications*, covers Microsoft's cross-platform Fluent Design System for applications. The history and evolution of Windows design concepts will be explored. The chapter will also introduce the Fluent XAML Theme Editor for creating a custom UI theme for a WinUI application.

*Chapter 8, Building WinUI Applications with .NET 5*, explains how to create a WinUI for Desktop project that targets the .NET 5 platform instead of the UWP app platform. The chapter also illustrates how to create a .NET 5 control library to share controls across multiple WinUI desktop projects.

*Chapter 9, Enhancing Applications with the Windows Community Toolkit*, introduces you to the open source Windows Community Toolkit and its sample application for Windows. Controls, helpers, and other libraries from the toolkit will be explored and added to the sample projects in this chapter.

*Chapter 10, Modernizing Existing Win32 Applications with XAML Islands*, covers the XAML Islands controls from the Windows Community Toolkit and explains how to integrate them into WPF and WinForms applications as the first step toward application modernization and WinUI adoption.

*Chapter 11, Debugging WinUI Applications with Visual Studio*, takes a deep dive into the XAML debugging tools available to Windows developers in Visual Studio 2019. These tools allow WinUI developers to optimize and troubleshoot their XAML-based user interface code and data binding issues.

*Chapter 12, Hosting an ASP.NET Core Blazor Application in WinUI*, describes how to create and deploy a Blazor **single-page application (SPA)** to Azure with Visual Studio Code and GitHub Actions. The deployed Blazor application will then be hosted within a Windows application by leveraging the new WebView2 WinUI control.

*Chapter 13, Building, Releasing, and Monitoring Applications with Visual Studio App Center*, examines the features available in Visual Studio App Center to build, deploy, and monitor Windows applications. The chapter will help you understand how to instrument your code to get real-time analytics and crash data from your production applications.

*Chapter 14, Packaging and Deploying WinUI Applications*, talks about the different methods for deploying WinUI applications to consumers. The chapter explores how to use Visual Studio, the Microsoft Partner Center, the Microsoft Store, and WinGet to deploy your applications.

## To get the most out of this book

You will need to install Visual Studio 2019 version 16.9 or later (<https://visualstudio.microsoft.com/downloads/>) with the following workloads and the latest WinUI 3.0 NuGet package (<https://marketplace.visualstudio.com/items?itemName=Microsoft-WinUI.WinUIProjectTemplates>):

- Universal Windows Platform Development
- .NET Desktop Development (includes .NET 5)

The code and instructions should also work with newer versions of the recommended software. The latest WinUI 3.0 pre-requisites are available on Microsoft Docs: <https://docs.microsoft.com/en-us/windows/apps/winui/winui3/#install-winui-3-preview-4>.

Software/hardware covered in the book	OS requirements
Visual Studio 2019 (version 16.9)	Windows 10 version 1803 or later
WinUI 3.0 templates	
Visual Studio Code	Windows 10, macOS, or Linux

**Note**

As WinUI 3 and Project Reunion are in active development in 2021, it is possible that the names of some projects, packages, and libraries referenced in the book may change in future releases. XAML Islands will not be available and UWP clients will not be fully supported initially. For a full list of what is planned for the first stable release of WinUI 3.0, you can reference the team's roadmap on GitHub: <https://github.com/microsoft/microsoft-ui-xaml/blob/master/docs/roadmap.md#winui-3-0-feature-roadmap>

*For the chapter on Blazor web development, it is recommended to download Visual Studio Code (<https://code.visualstudio.com/>). If you have not already installed the .NET Desktop Development workload for Visual Studio, the .NET 5 SDK will also be needed for Blazor development (<https://dotnet.microsoft.com/download/dotnet/5.0>).*

If you are using the digital version of this book, we advise you to type the code yourself or access the code via the GitHub repository (link available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

*For additional background on Windows development concepts, you can explore the Develop Windows 10 applications Learning Path on Microsoft Learn (<https://docs.microsoft.com/en-us/learn/paths/develop-windows10-apps/>).*

## Download the example code files

You can download the example code files for this book from your account at [www.packtpub.com](http://www.packtpub.com). If you purchased this book elsewhere, you can visit [www.packtpub.com/support](http://www.packtpub.com/support) and register to have the files emailed directly to you.

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/-Learn-WinUI-3.0>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

## Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: [https://static.packt-cdn.com/downloads/9781800208667\\_ColorImages.pdf](https://static.packt-cdn.com/downloads/9781800208667_ColorImages.pdf).

## Conventions used

There are a number of text conventions used throughout this book.

**Code in text:** Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "The way to intercept the launch event is to override the `OnLaunched` method on the application class."

A block of code is set as follows:

```
while (query.Read())
{
    var medium = new Medium
    {
        Id = query.GetInt32(0),
        Name = query.GetString(1),
        MediaType = (ItemType)query.GetInt32(2)
    };
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
private void OnSuspending(object sender, SuspendingEventArgs e)
{
    var deferral = e.SuspendingOperation.GetDeferral();
    deferral.Complete();
}
```

**Bold:** Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Now when you run the app, you will see that the new style has been applied to both the **Submit** and **Cancel** buttons without adding any styling directly to each control."

**Tips or important notes**

Appear like this.

## Get in touch

Feedback from our readers is always welcome.

**General feedback:** If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at [customercare@packtpub.com](mailto:customercare@packtpub.com).

**Errata:** Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit [www.packtpub.com/support/errata](http://www.packtpub.com/support/errata), selecting your book, clicking on the Errata Submission Form link, and entering the details.

**Piracy:** If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at [copyright@packt.com](mailto:copyright@packt.com) with a link to the material.

**If you are interested in becoming an author:** If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit [authors.packtpub.com](http://authors.packtpub.com).

## Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit [packt.com](http://packt.com).

# Section 1: Introduction to WinUI and Windows Applications

WinUI 3.0 is Microsoft's new UI framework for Windows developers. This section will start by exploring the recent history of XAML and Windows UI frameworks and introduce readers to WinUI. Throughout the chapters of this section, you will learn about WinUI concepts by building a simple project from scratch, adding controls and features by following design patterns and best practices. These patterns and practices include the **Model-View-ViewModel (MVVM)** design pattern, unit testing WinUI projects, and using **dependency injection (DI)** to inject service dependencies into the application components.

This section includes the following chapters:

- *Chapter 1, Introduction to WinUI*
- *Chapter 2, Configuring the Development Environment and Creating the Project*
- *Chapter 3, MVVM for Maintainability and Testability*
- *Chapter 4, Advanced MVVM Concepts*
- *Chapter 5, Exploring WinUI Controls*
- *Chapter 6, Leveraging Data and Services*



# 1

# Introduction to WinUI

WinUI is a set of open source controls and libraries that Windows developers can leverage in their **Universal Windows Platform (UWP)** and Win32 applications. UWP developers use the Windows **software development kit (SDK)** to build their applications and are required to select a target SDK version in a project's properties. By extracting the UWP controls and **user interface (UI)** components from the Windows SDK and releasing them as a set of open source libraries under the name WinUI, Microsoft is able to release versions at a faster cadence than Windows itself (as Windows SDK versions are linked to those of Windows). This separation also enables the controls to be used on older versions of Windows 10. While building UWP and Win32 applications with WinUI is the current recommendation, it is important to learn where WinUI and UWP fit in the larger Windows development landscape.

In this book, you will learn how to build applications for Windows with the WinUI 3.0 libraries. Throughout the course of the book, we will build a real-world application using recommended patterns and practices for Windows application development.

Before we start building our WinUI app, it's important to have a good foundation in Windows client development, the different types of **Extensible Application Markup Language (XAML)** UI markup, and how WinUI compares to other Windows desktop development frameworks. Therefore, in this first chapter, you will start by learning some background on UWP and WinUI.

In this chapter, we will learn about the following topics:

- What UWP is and why Microsoft created yet another application framework
- How XAML can be leveraged to create great UIs on many device sizes and families
- Why WinUI was created and how it relates to UWP
- Where WinUI fits in the Windows developer landscape
- What WinUI 3.0 brings to the table

Don't worry! It won't take very long to cover the background stuff, and it will help provide some context as you start building your WinUI app. In the next chapter, you will get your hands on some code when you create your first WinUI project.

## Technical requirements

To follow along with the examples in this chapter, the following software is required:

- Windows 10 version 1803 or newer. You can find your version of Windows in **Settings | About**.
- Visual Studio 2019 version 16.9 or newer with the following workloads: .NET Desktop Development and UWP Development.
- WinUI 3.0 project templates—at the time of this writing, the templates can be downloaded from Visual Studio Marketplace at <https://marketplace.visualstudio.com/items?itemName=Microsoft-WinUI>. `WinUIProjectTemplates`. After WinUI 3.0 is released, the templates will likely be included with Visual Studio.

The source code for this chapter is available on GitHub at this URL: <https://github.com/PacktPublishing/-Learn-WinUI-3.0/tree/master/Chapter01>.

**Note**

The WinUI 3.0 site on Microsoft Docs has up-to-date guidance on setting up a developer workstation for WinUI development: <https://docs.microsoft.com/en-us/uwp/toolkits/winui3/>.

## Before UWP – Windows 8 XAML applications

Before UWP applications were launched with Windows 10 in 2015, there were XAML applications for Windows 8 and 8.1. The XAML syntax and many of the **application programming interfaces (APIs)** were the same, and they were Microsoft's next step in an attempt to achieve universal app development across desktop, mobile, and other platforms (Xbox, mixed reality, and so on). A XAML app could be written for Windows 8 and Windows Phone. These projects would generate separate sets of binaries that could be installed on a PC or a Windows Phone.

These apps had many other limitations that modern UWP apps do not. For instance, they only ran fullscreen, as shown in the following screenshot:

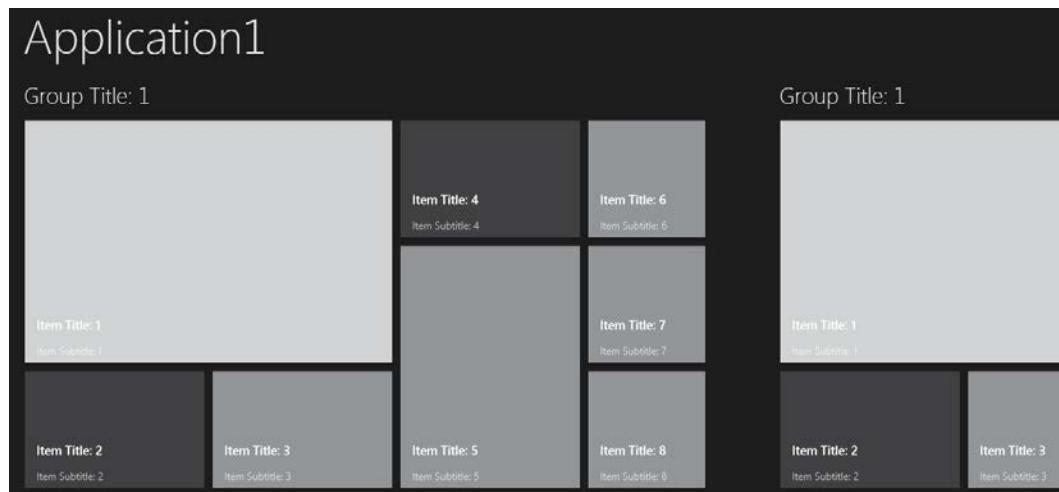


Figure 1.1 – Windows 8 fullscreen app (sourced from Stack Overflow; reproduced under CC BY-SA 4.0 – <https://creativecommons.org/licenses/by-sa/4.0/>)

Many other early restrictions on Windows 8 apps have been lessened or completely removed in UWP app development. *Figure 1.2*, which follows, documents these changes:

	<b>Windows 8 XAML App</b>	<b>Windows 10 UWP App</b>
<b>Window Type</b>	Full screen only	Resizable window
<b>Device Type</b>	Runs on PC only	Multiple Windows 10 device types
<b>Number of Instances</b>	1	1 (default) or Multiple
<b>Console App Supported</b>	No	Yes
<b>File System Access</b>	Sandboxed - local storage only	Sandboxed by default  App can request additional access to user folders and removable devices

Figure 1.2 – Windows 8 and Windows 10 app comparison table

## Windows application UI design

The term *Metro Style* was used to define the design and layout of Windows 8 apps. Metro Style apps were designed to be usable with touch input, mouse and keyboard, or a stylus. Microsoft's introduction of the first Windows Phone was a driving factor for Metro Style design. Metro Style later became Modern UI design, with the introduction of Surface devices. Aspects of Metro live on today, in UWP apps and Windows 10.

Live Tiles were born with Metro Style. These tiles on the user's Windows 8 home screen and Windows 10 Start menu can update to display live updates to users without having to open the app. Most of Microsoft's own apps for Windows support Live Tiles. The Weather app can show live updates to current weather conditions on the tile, based on the user's current location. You will explore Live Tiles further in *Chapter 5, Exploring WinUI Controls and Libraries*.

## Windows Runtime (WinRT)

Another term that has its roots in Windows 8 app development is WinRT. The letters RT became a source of great confusion. WinRT was short for Windows Runtime, the underlying APIs used by Windows XAML apps. There was also a version of Windows 8 called Windows RT that supported **Advanced RISC Machines (ARM)** processors. The first Surface PC was the Surface RT, which ran the Windows 8 RT operating system.

Although WinRT can still be used today to define the WinRT APIs consumed by UWP apps, you will not see the term as often. We will also avoid using WinRT in this book and instead refer to the APIs as the UWP or Windows APIs.

## User backlash and the path forward to Windows 10

While Microsoft pushed hard to win over users with Modern UI design, a new app model, Surface PCs, and Windows 8 and 8.1, the idea of a fullscreen, touch-first app experience and a deemphasized Windows desktop was never embraced by customers. It turns out that Windows users really like the start menu experience they used for years with Windows XP and Windows 7.

The next step in Windows app development was a big one—so big, in fact, that Microsoft decided to skip a number in their versioning, jumping straight from Windows 8.1 to Windows 10.

## Windows 10 and UWP application development

While taking a leap forward with the launch of Windows 10, Microsoft also blended the best of what worked in previous versions of Windows. They brought back the start menu, but its contents look an awful lot like the Windows 8 home screen experience. In addition to an alphabetized list of all installed apps, there is a resizable area for pinned app tiles. In fact, when running Windows in Tablet Mode, the start menu can transform into the Windows 8-style home screen experience for better usability on a touchscreen.

When Microsoft launched Windows 10, they also introduced UWP applications to Windows developers. While UWP apps have their roots in the XAML apps of Windows 8, there are some key differences that give developers some major advantages when building apps for the platform.

A key advantage is in the Universal aspect of these apps. Microsoft builds versions of Windows 10 to run on different device families, listed as follows:

- Desktop (PC)
- Xbox
- Mobile (Windows Phone)
- HoloLens
- IoT
- IoT Headless
- Team (Surface Hub)

UWP developers can build apps to target any of these devices. There is a single base set of Windows APIs shared across all these targets, and specialized SDKs available for the device-specific APIs of some families—for example, there is a Mixed Reality Toolkit and SDK for HoloLens development. With UWP, it is possible to create a single project to target many device families—for instance, you can create a project that creates apps for Desktop, Xbox, and Team families.

Because the UWP XAML for building the app's UI is the same, the learning curve for cross-device development is lowered and code reusability is very high. The nature of XAML provides a UI flexibility to adapt to different device sizes and aspect ratios.

## Language choice with UWP development

While the underlying UWP APIs were written in C++, UWP developers can choose from a number of programming languages when building apps for Windows. UWP projects can be created with any of these popular languages:

- C#
- C++
- F#
- **Visual Basic .NET (VB.NET)**
- **JavaScript**

You may be surprised to see JavaScript on the list. During the Windows 8.x days, developers could create JavaScript apps with APIs known as WinJS apps. Today, Microsoft has created a branch of React Native for Windows developers, known as React Native for Windows. These JavaScript client apps have full access to the same Windows APIs as other UWP apps and can be packaged and deployed through the Windows Store.

**Note**

React Native for Windows is an open source project hosted by Microsoft on GitHub at <https://github.com/Microsoft/react-native-windows>.

While many of the UWP apps developed for Windows 10 by Microsoft are created with C++, most other developers choose C#. We will also use C# when building our app throughout the course of this book.

## Lifting app restrictions

As discussed earlier, apps built for Windows 8 had a number of restrictions that have been either removed or relaxed with UWP.

First and foremost, today's UWP apps can run in resizable windows, just like any other Windows desktop application. The trade-off is that developers now need to test for and handle the resizing of their app to almost any size. The dynamic nature of XAML can handle much of the resizing very well, but below a certain minimum size, scroll bars will need to be employed.

For end users, one of the benefits of using UWP apps is the inherent security they provide due to the limited access of apps to the PC's filesystem. By default, each app can only access its own local storage. In 2018, the Windows developer team announced a new feature for UWP developers. By adding some app configuration declaring which additional types of access the app requires, applications can request access to additional parts of the filesystem. Among them are the following:

- User libraries, including Documents, Pictures, Music, and Videos
- Downloads
- Removable devices

### Note

There are additional filesystem permissions that can be requested. See the Microsoft documentation for an entire list: <https://docs.microsoft.com/en-us/windows/uwp/files/file-access-permissions>.

Any additional permissions requested will be declared on the app's listing on the Windows Store.

Some less common scenarios are now available to UWP apps on Windows 10. Developers can add some configuration and startup code to enable multiple instances of their app to launch. While it would seem that the point of a UWP app is the XAML UI, it is now possible to create a UWP console app. The app will run at the command line and have access to Universal C runtime calls. Developers who want to get started with console apps can find project templates on Visual Studio Marketplace, at <https://marketplace.visualstudio.com/items?itemName=AndrewWhitechapelMSFT.ConsoleAppUniversal>.

## UWP backward compatibility

No UWP app is compatible with any version of Windows before Windows 10. Beyond this, each UWP app must declare a **Target Version** and a **Minimum Version** of Windows with which it is compatible. The target version is your recommended version, which will enable all of an app's features and functionality. The minimum version is, unsurprisingly, the minimum version of Windows that users must have to be able to install an app from the Windows Store.

Visual Studio will prompt you to select these versions when creating a new UWP project. If the two are the same, it keeps things simple. You will have all of the APIs of that SDK version available to the app. If the target version is greater than the minimum version, you need to add some conditional code to light up the features of any versions greater than the minimum. The app must still be useful to users running the minimum version; otherwise, it is advisable to increase the minimum. If any of the newer APIs or controls are fundamental to the app, it is also recommended that the minimum version be increased to one where those are available.

**Note**

For more information on writing the conditional or version adaptive code, see the Microsoft documentation here: <https://docs.microsoft.com/en-us/windows/uwp/debug-test-perf/version-adaptive-code>.

If you are creating .NET libraries that will be referenced by your UWP project and you would like to share them across other platforms, perhaps by a Xamarin mobile app, a .NET Standard version should be targeted by the shared library project. The most common .NET Standard version today is .NET Standard 2.0. To reference a .NET Standard 2.0 project from a UWP project, the target version of the UWP project should be 16299 or later.

The primary benefit of WinUI over UWP is that it lessens the dependency of Windows apps on a particular version of Windows. Instead, the controls, styles, and APIs are maintained outside of the Windows SDK. As of this writing, the minimum version required for a WinUI 3.0 app is 17134 or higher, and the target version must be set to 18362 or higher. Check the latest WinUI documentation for the current minimum requirements.

The hope for WinUI is to bring a greater number of controls and features to more supported versions of Windows 10 as the project matures.

## What is XAML?

XAML is based on **Extensible Markup Language (XML)**. This would seem like a great thing as XML is a flexible markup language familiar to most developers. It is indeed flexible and powerful, but it has some drawbacks.

The primary problem with Microsoft's implementations of XAML is that there have been so many variations of the XAML language created for different development platforms over the years. Currently, UWP, **Windows Presentation Foundation (WPF)**, and Xamarin.Forms applications all use XAML as their UI markup language. However, each of these uses a different XAML implementation or schema, and the markup cannot be shared across the platforms. In the past, Windows 8, Silverlight, and Windows Phone apps also had other different XAML schemas.

If you have never worked with XAML before, you're probably ready to see an example of some UI markup. The following XAML is a fragment that defines a `Grid` containing several other of the basic WinUI controls (you can download the code for this chapter from GitHub here: <https://github.com/PacktPublishing/-Learn-WinUI-3.0/tree/master/Chapter01>):

```
<Grid Width="400" Height="250" Padding="2"
    HorizontalAlignment="Center"
    VerticalAlignment="Center">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="*"/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto"/>
        <ColumnDefinition Width="*"/>
    </Grid.ColumnDefinitions>

    <TextBlock Grid.Row="0" Grid.Column="0"
        Text="Name :"
        Margin="0,0,2,0"
        VerticalAlignment="Center"/>
    <TextBox Grid.Row="0" Grid.Column="1"
        Text="" />
    <Button Grid.Row="1" Grid.Column="1" Margin="0,4,0,0"
        HorizontalAlignment="Right"
```

```
    VerticalAlignment="Top"
    Content="Submit"/>
</Grid>
```

Let's break down the XAML here. The top level of a UWP window is `Page`. UWP app navigation is page-based, and this top-level navigation happens within a root `Frame` container in the `App.xaml` file in the project. You will learn more about page navigation in *Chapter 4, Advanced MVVM Concepts*. A `Page` must contain only one child, usually some type of layout panel such as a `Grid` or `StackPanel`. By default, a `Grid` is inserted as that child. We will discuss other types of panels that serve as a good parent container in the next chapter. I made a few modifications to the `Grid`.

`Height` and `Width` properties provide a static size for the example, and `HorizontalAlignment` and `VerticalAlignment` properties will center the `Grid` on the `Page`. Fixed sizes are uncommon at this level of the XAML and limit the flexibility of the layout, but they illustrate some of the available attributes.

A `Grid` is a layout panel that allows developers to define rows and columns to arrange its elements. The rows and columns can have their sizes defined as fixed, relative to each other, or auto-sized based on their contents. For more information, you can read the Microsoft Docs article *Responsive layouts with XAML*: <https://docs.microsoft.com/en-us/windows/uwp/design/layout/layouts-with-xaml>.

The `Grid.RowDefinitions` block defines the number and behavior of the grid's rows. Our grid will have two rows. The first one has `Height="Auto"`, which means it will resize itself to fit its contents, provided enough space is available. The second row has `Height="*"`, which means the rest of the grid's vertical space will be allocated to this row. If multiple rows have their height defined like this, they will evenly split the available space. We will discuss additional sizing options in the next chapter.

The `Grid.ColumnDefinitions` block does for the grid's columns what `RowDefinitions` did for the rows. Our grid has two columns defined. The first `ColumnDefinition` has its `Height` set to `Auto`, and the second has `Height="*"`.

`TextBlock` defines a label in the first `Grid.Row` and `Grid.Column`. When working with XAML, all indexes are 0-based. In this case, the first `Row` and `Column` are both at position 0. The `Text` property conveniently defines the text to display, and the `VerticalAlignment` in this case will vertically center the text for us. The default `VerticalAlignment` for a `TextBlock` is `Top`. The `Margin` property adds some padding around the outside of the control. A margin with the same amount of padding on all sides can be set as a single numeric value. In our case, we only want to add a couple of pixels to the right side of the control to separate it from `TextBox`. The format for entering these numeric values is "`<LEFT>,<TOP>,<RIGHT>,<BOTTOM>`", or "`0,0,2,0`" here.

The `TextBox` is a text entry field defined in the second column of the grid's first row.

Finally, we've added a `Button` control to the second column of the grid's second row. A few pixels of top margin are added to separate it from the controls above. The `VerticalAlignment` is set to `Top` (the default is `Center`) and `HorizontalAlignment` is set to `Right` (the default is `Center`). To set the text of the `Button`, you don't use the `Text` property like we did with the `TextBlock`, as you might think. In fact, there is no `Text` property. The `Content` property of the `Button` is used here. `Content` is a special property that we will discuss in more detail in the next chapter. For now, just know that a `Content` property can contain any other control: text, an `Image`, or even a `Grid` control containing multiple other children. The possibilities are virtually endless.

Here is the UI that gets rendered by the preceding markup:



Figure 1.3 – WinUI XAML rendered

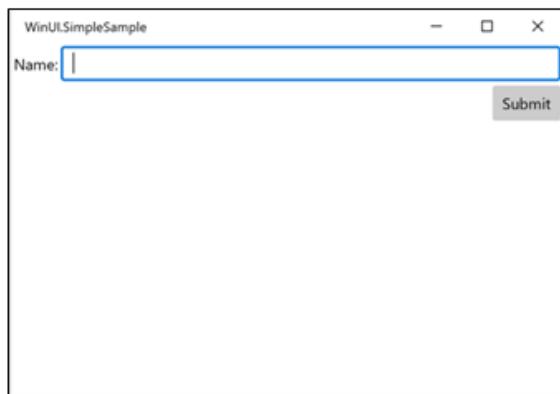
This is a very simple example to give you a first taste of what can be created with XAML. As we move ahead, you will learn how powerful the language can be.

## Creating an adaptive UI for any device

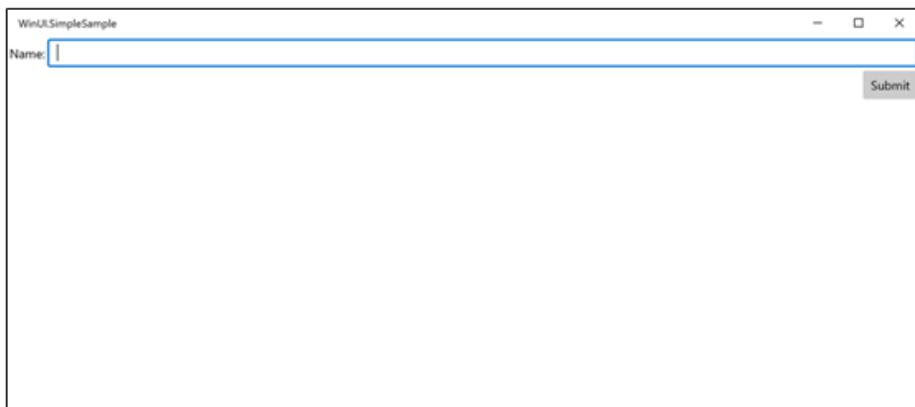
In the previous example, the `Grid` had fixed `Height` and `Width` properties. I mentioned that setting fixed sizes can limit a UI's flexibility. Let's remove the fixed size properties and use the alignment properties to guide the UI elements, to render how we want them to at different sizes and aspect ratios, as follows:

```
<Grid VerticalAlignment="Top" HorizontalAlignment="Stretch"
    Padding="2">
```

The rest of the markup remains unchanged. The result is a `TextBox` that resizes to fit the width of the window, and the `Button` remains anchored to the right of the window as it resizes. See the window resized a couple of different ways here:



A narrower window



A wider window

Figure 1.4 – Resized windows

If you were using this app on a tablet PC, the contents would resize themselves to fit in the available space. That is the power of XAML's adaptive nature. When building a UI, you will usually want to choose relative and adaptive properties such as alignment to fixed sizes and positions.

It's this adaptive layout that makes XAML work so well on mobile devices with Xamarin, and this is why WPF developers have loved using it since its launch with Windows Vista.

## Powerful data binding

Another reason why UWP and other XAML-based frameworks are so popular is the ease and power of their data-binding capabilities. Nearly all properties on UWP controls can be data-bound. The source of the data can be an object or a list of objects on the data source. In most cases, that source will be a `ViewModel` class. Let's have a very quick look at using UWP's `Binding` syntax for data binding to a property on a `ViewModel` class, as follows:

1. First, we will create a simple `MainViewModel` class with a `Name` property, like this:

```
public class MainViewModel : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler
        PropertyChanged;
    private string _name;
    public MainViewModel()
    {
        _name = "Bob Jones";
    }
    public string Name
    {
        get
        {
            return _name;
        }
        set
        {
            if (_name == value) return;
            _name = value;
        }
    }
}
```

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

The `MainViewModel` class implements an interface called `INotifyPropertyChanged`. This interface is key to the UI receiving updates when data-bound properties have changed. This interface implementation is typically wrapped either by a **Model-View-ViewModel (MVVM)** framework, such as Prism or MvvmCross, or with your own `ViewModelBase` class. For now, we will directly invoke a `PropertyChanged` event inside the `Name` property's setter. We will learn more about ViewModels and the `INotifyPropertyChanged` interface in *Chapter 3, MVVM for Maintainability and Testability*.

2. The next step is to create an instance of the `MainViewModel` class and set it as the `ViewModel` for our `MainPage`. This happens in the code-behind file for the page, `MainPage.xaml.cs`, as illustrated in the following code snippet:

```
public sealed partial class MainPage : Page
{
    public MainPage()
    {
        this.InitializeComponent();
        this.ViewModel = new MainViewModel();
    }
    public MainViewModel ViewModel { get; set; }
}
```

We have added a `ViewModel` property to the `MainPage` and set it to a new instance of our `MainViewModel` class in the constructor.

**Tip**

Any code added to a page's constructor should be added after the call to `InitializeComponent()`.

- Now, it's time to add the data-binding code to the XAML markup for the TextBox, as follows:

```
<TextBox Grid.Row="0" Grid.Column="1" Text="{x:Bind Path=ViewModel.Name, Mode=TwoWay}" />
```

Some markup has been added to set the Text property using the `x:Bind` markup extension. The data-binding Path is set to the Name property on the `ViewModel`, which has been assigned in the code-behind file in the preceding *Step 2*. By setting the data-binding mode to `TwoWay`, updates in the `ViewModel` will display in the UI, and any updates by the user in the UI will also be persisted in the Name property of the `MainViewModel` class. Now, running the app will automatically populate the name that was set in the constructor of the `ViewModel`, as illustrated in the following screenshot:

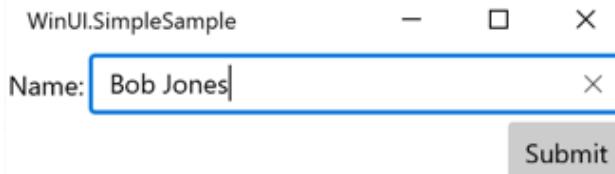


Figure 1.5 – Data binding the TextBox

- To illustrate data binding to another property on another UI element on the page, we will first modify the grid to add a name, as follows:

```
<Grid x:Name="ParentGrid" VerticalAlignment="Top" HorizontalAlignment="Stretch" Padding="2">
```

- Now add another `RowDefinition` to the `Grid` to fit the new UI element in the page:

```
<Grid.RowDefinitions>
  <RowDefinition Height="Auto" />
  <RowDefinition Height="Auto" />
  <RowDefinition Height="*" />
</Grid.RowDefinitions>
```

6. Next, add a `TextBlock` element and use the `Binding` markup extension to bind its `Text` property to the `ActualWidth` of the `ElementName` set to `ParentGrid`. We are also adding a `TextBlock` to label this as the **Actual Width**:

```
<TextBlock Grid.Row="1" Grid.Column="0"
           Text="Actual Width:"
           Margin="0,0,2,0"
           VerticalAlignment="Center"/>
<TextBlock Grid.Row="1" Grid.Column="1"
           Text="{Binding ElementName=ParentGrid,
           Path=ActualWidth}" />
```

7. Next, update the **Submit** Button to appear in `Grid.Row 2`.
8. Now the new `TextBlock` control displays the width of the `ParentGrid` when the page is loaded. Note that it will not update the value if you resize the window. The `ActualWidth` property does not raise a property change notification. This is documented in the `FrameworkElement.ActualWidth` docs: <https://docs.microsoft.com/en-us/uwp/api/windows.ui.xaml.frameworkelement.actualwidth>:



Figure 1.6 – Data binding to another element

The **Submit** button does not function yet. You will learn how to work with **Events** and **Commands** with MVVM in *Chapter 5, Exploring WinUI Controls and Libraries*.

## Styling your UI with XAML

When working with XAML, styles can be defined and applied at almost any scope, global to the application in `App.xaml`, in the current `Page` inside a `Page.Resources` declaration, or inside any level or nested control on the page. The `Style` property specifies a `TargetType` property, which is the data type of the elements to be targeted by the style. It can optionally have a `Key` property defined as a unique identifier, similar to a class identifier in **Cascading Style Sheets (CSS)**. That `Key` property can be used to apply the style to only selected elements of that type. Only one `Key` property can be assigned to an element, unlike with CSS classes.

In the next example, we will modify the page to define a **Style** property for all buttons on the page, as follows:

1. Start by moving the **Submit** button to be nested inside a **StackPanel** element. A **StackPanel** element stacks all child elements in a horizontal or vertical orientation, with vertical being the default orientation. Some of the button's properties will need to be moved to the **StackPanel** element, as it is now the direct child of the **Grid**. After adding a second button to the **StackPanel** element to act as a **Cancel** button, the code for the **StackPanel** and **Button** elements should look like this:

```
<StackPanel Grid.Row="1" Grid.Column="1"
            Margin="0,4,0,0"
            HorizontalAlignment="Right"
            VerticalAlignment="Top"
            Orientation="Horizontal">
    <Button Content="Submit" Margin="0,0,4,0"/>
    <Button Content="Cancel"/>
</StackPanel>
```

A new margin has been added to the first button to add some space between the elements.

2. Next, we will add a **Style** block to the **Page.Resources** section to style the buttons. Because no **Key** is assigned to the **Style** block, it will apply to all **Button** elements that do not have their styles overridden in an inner scope. This is known as an *implicit style*. The code for this is shown here:

```
<Page.Resources>
    <Style TargetType="Button">
        <Setter Property="BorderThickness"
               Value="2" />
        <Setter Property="Foreground"
               Value="LightGray" />
        <Setter Property="BorderBrush"
               Value="GhostWhite" />
        <Setter Property="Background"
               Value="DarkBlue" />
    </Style>
</Page.Resources>
```

3. Now, when you run the app, you will see that the new style has been applied to both the **Submit** and **Cancel** buttons without adding any styling directly to each control, as illustrated in the following screenshot:



Figure 1.7 – Styled buttons

If we moved the `Style` block to the `Application.Resources` section, the defined style would get applied to every button in the entire app unless the developer had individually overridden some of the properties in the style. For instance, if the **Submit** button had a `Background` property set to `DarkGreen`, only the **Cancel** button would appear as dark blue.

We will spend more time on styles and design in *Chapter 7, Windows Fluent UI Design*.

## Separating presentation from business logic

We looked briefly at the MVVM pattern in the earlier section on data binding. MVVM is key to the separation of presentation logic from business logic in UWP application development. The XAML elements only need to know that there is a property with a particular name somewhere in its data context. The `ViewModel` classes have no knowledge of the `View` (our XAML file).

This separation provides several benefits. First, `ViewModels` can be unit tested independently of the UI. If any UWP elements are referenced by the system under test, the UI thread is needed. This will cause tests to fail when they're running on background threads locally or on a **Continuous Integration (CI)** server. See *Chapter 3, MVVM for Maintainability and Testability* for more information on unit testing WinUI applications.

The next benefit of `View/ViewModel` separation is that businesses with dedicated **user experience (UX)** experts will sometimes work on designing the XAML markup for an app while other developers are building the `ViewModels`. When it is time to sync up the two, the developer can add in the necessary data-binding properties to the XAML, or perhaps the UX designer and developer have already agreed upon the names of the properties in the shared data context. Visual Studio includes another tool geared toward designers in this workflow, called Blend for Visual Studio. Blend was first released by Microsoft in 2006 as Microsoft Expression Blend, as a tool for designers to create UIs for WPF. Support was later added for other XAML languages such as Silverlight and UWP. Blend is still included with the UWP development workload when installing Visual Studio.

A final benefit we will discuss here is that a good separation of concerns between any layers of your application will always lead to better maintainability. If there are multiple components involved in a single responsibility or if logic is duplicated in multiple places, this leads to buggy code and unreliable applications. Follow good design patterns, and you will save yourself a lot of work down the road.

Now that you have a good understanding of the history of UWP applications, it's time to look at WinUI: what it is, and why it was created.

## What is WinUI?

The WinUI library is a set of controls and UI components that have been extracted from the Windows SDK. After this separation, many controls have been enhanced and others have been added. The libraries have been made available as open source on GitHub and are maintained by Microsoft and the Windows developer community.

So, if these WinUI libraries came from UWP libraries in the Windows SDK, you may be wondering why you should choose WinUI as your UI framework instead of UWP. UWP has been around since the launch of Windows 10 and is quite robust and stable. There are actually several very good reasons to consider WinUI.

Choosing WinUI brings with it all the benefits of open source. **Open source software (OSS)** is typically very reliable. When software is developed in the open by an active developer community, issues are found and resolved quickly. In fact, if you find an issue with an open source package, you can fix it yourself and submit a pull request to have the fix made available to the rest of the community. Open source projects can iterate quickly without having to remain in sync with product groups in a large enterprise such as the Windows team. Windows releases feature updates on a regular cadence now, but this is still less frequent than with a typical control library.

The best reason to use WinUI is its backward compatibility. When using a UWP control, the features and fixes in a specific version of the control cannot be deployed in apps to older versions of Windows. With WinUI, so long as you are targeting the minimum version of Windows supported by WinUI as a whole, you can use those new controls and features in multiple Windows versions. Controls not previously available to UWP developers on one version of Windows are now available there as WinUI controls.

For instance, Microsoft did not introduce the Fluent UI design to Windows until the Fall 2017 release (version 16299). However, WinUI controls can be included in apps targeting a minimum Windows version of 10.0.15063.0, the Spring 2017 release. The controls in WinUI support Fluent UI styles. WinUI adds controls and other features that are not available at all in UWP and the Windows SDK.

## The first WinUI release

The first version of WinUI was released in July 2018 as a preview release for Windows developers. It was released as the following two NuGet packages:

- `Microsoft.UI.Xaml`: The WinUI controls and Fluent UI styles.
- `Microsoft.UI.Xaml.Core.Direct`: Components for middleware developers to access the `XamlDirect` API.

3 months later, WinUI 2.0 was released. Despite the version number, it was the first production release of WinUI. The release included more than 20 controls and brushes. A few notable controls included the following:

- `TreeView`: A staple of any UI library.
- `ColorPicker`: A rich visual color picker with a color spectrum.
- `DropDownButton`: A button with the ability to open a menu.
- `PersonPicture`: An image control for displaying an avatar. It has the ability to fall back to displaying initials or a generic placeholder image.
- `RatingControl`: Allows users to enter star ratings for items.

Let's add a few of these controls to our WinUI project and see how they look. Change the contents of the `StackPanel` to look like this:

```
<StackPanel Grid.Row="1" Grid.Column="1" Margin="0,4,0,0">
    <PersonPicture Initials="MS" Margin="0,0,8,0"/>
    <DropDownButton Content="Submit" Margin="0,0,4,0">
        <DropDownButton.Flyout>
            <MenuFlyout Placement="Bottom">
                <MenuFlyoutItem Text="Submit + Print"/>
                <MenuFlyoutItem Text="Submit + Email"/>
            </MenuFlyout>
        </DropDownButton.Flyout>
    </DropDownButton>
    <Button Content="Cancel"/>
</StackPanel>
```

A `PersonPicture` control with the initials `MS` has been added as the first item in the `StackPanel`, and the first of the two buttons has been replaced by a `DropDownButton` control. The `Submit` `DropDownButton` control has a `FlyoutMenu` serving as a drop-down list, and there are two `MenuItem` elements. Now, users can simply click the `Submit` button, or they can select `Submit + Print` or `Submit + Email` from the drop-down list.

**Note**

`DropDownButton` is only available in *Windows 10 version 1809 and later*. If you use this control in a production application, you should set this as your minimum version for the project.

This is how the new window appears with the `DropDownButton` menu shown:

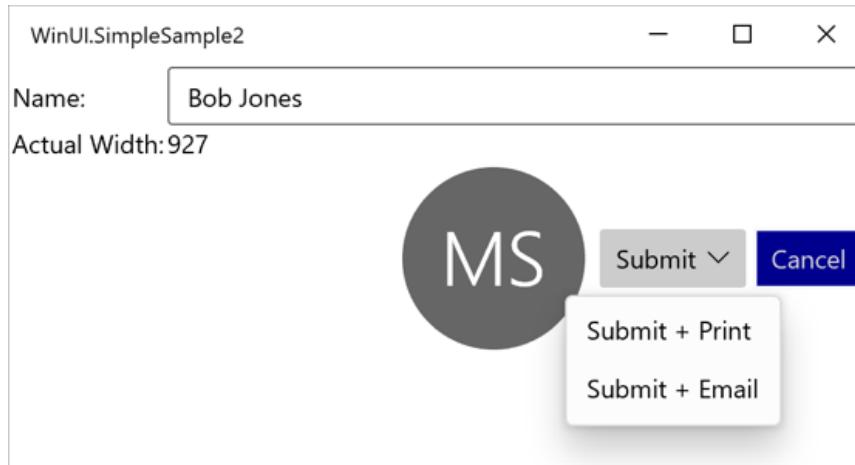


Figure 1.8 – Adding a `PersonPicture` and `DropDownButton` control

We're only scratching the surface of what the first release can do for Windows developers. Don't worry, as we will dive much deeper in the chapters ahead. Let's briefly look at what was added in subsequent versions leading to WinUI 3.0.

## The road to WinUI 3.0

There have been five additional minor releases of WinUI following v2.0, in addition to many incremental bug fixes and prerelease versions.

## WinUI 2.1

The WinUI 2.1 release brought a number of new controls and features to the library. These are some highlights:

- **TeachingTip:** Think of TeachingTip as a rich, context-sensitive tooltip. It is linked to another element on the page and displays informative details about the target element to help guide users with non-intrusive content as needed.
- **AnimatedVisualPlayer:** This hosts Lottie animations. Lottie files are a popular animation format created in **Adobe After Effects** used by designers across Windows, the web, and mobile platforms. There are libraries available to host Lottie animations for most modern development frameworks.

### Note

Get more information about Lottie files on their website at <https://airbnb.design/lottie/> and check out this great repository of Lottie animation files: <https://lottiefiles.com/>

- **CompactDensity:** Adding this resource dictionary to your app can provide the ability to switch between *Compact* and *Normal* display modes. CompactDensity will reduce the spacing within and between elements on the page, providing up to 33% more visible content to users. This Fluent UI design concept was introduced to developers at Microsoft's Build 2018 conference.

## WinUI 2.2

This release brought many enhancements to existing features. However, the single new control added to the library is one that many Windows developers will find useful.

The TabView control creates a familiar tabbed user experience on the screen. Each tab can host a page in your WinUI project.

### WinUI 2.2 enhancements

A few of the notable updated controls and libraries in version 2.2 are listed here:

- **NavigationView:** The NavigationView control was enhanced to allow the back button to remain visible when the panel is collapsed. Other visual updates maximize the viewable content of the control.

- **Visual Styles:** Many of the WinUI visual styles were updated, including `CornerRadius`, `BorderThickness`, `CheckBox`, and `RadioButton`. The updates all make the WinUI visuals more consistent and in line with Fluent UI design guidelines.

## WinUI 2.3

In the WinUI 2.3 release, the `ProgressBar` received some updates, and a couple of new controls were added to the library.

There are now two modes available when creating a **Progress Bar** in a WinUI application: **Determinate** and **Indeterminate**. A determinate progress bar has a known amount of the task to complete and a known current state of the task. An indeterminate control indicates that a task is ongoing without a known completion time. Its purpose is similar to that of a busy indicator.

## New controls in WinUI 2.3

The following are a few new controls in this update:

- `NumberBox`: A `NumberBox` control is an input editor that makes it easy to support numeric formatting, up/down incrementing buttons, and inline mathematical calculations. It is a seemingly simple but practical and powerful control.
- `RadioButtons`: You might be thinking: *Radio buttons have always been available. How is this a new control?* `RadioButtons` is actually a control that groups a set of `RadioButton` (*singular*) controls, making it easier to work with them as a single unit.

## WinUI 2.4

When it was released in May 2020, two new features were made available in WinUI 2.4: a `RadialGradientBrush` visual and a `ProgressRing` control.

The brush is similar in use to the `RadialGradientBrush` used by WPF developers. It makes it easy to add a gradient to a visual element that radiates out from a central point.

The `ProgressRing` control, as it sounds, recreates progress bar functionality in a circular format. The control is available with a determinate state and an indeterminate state in version 2.4. An indeterminate `ProgressRing` control displays a repeating animation and is the default state of the control.

Several controls were updated in version 2.4. The TabView control was updated to provide more control over how tabs are rendered, including **Compact**, **Equal**, and **Size to Content** modes. TextBox controls received a *dark mode* enhancement to keep the content area of the control dark, with white text by default. Finally, the NavigationView control was updated with hierarchical navigation, with **Left**, **Top**, and **LeftCompact** modes.

## WinUI 2.5

WinUI 2.5 was released in December 2020 and included a new **InfoBar** control. Several control enhancements and bug fixes were also included in the release.

The **InfoBar** control provides a way to display important status messages to users. The control can display an alert or informational icon, a status message, and a link or button allowing users to take action on a message. There is also an option to display a close button to the right of the message. By default, the control includes an icon, message, and close button. Microsoft Docs provides usage guidelines for this new control: <https://docs.microsoft.com/en-us/windows/uwp/design/controls-and-patterns/infobar>.

Several updates are also available in version 2.5. The **ProgressRing** control received enhancements to the determinate state of the control. The **NavigationView** control was updated to provide customizable **FooterMenuItem**s. In previous versions of the **NavigationView** control, the footer area could be shown or hidden but not customized.

We've seen what was available to UWP developers in WinUI 2.x. Now, let's see what you get with WinUI 3.0.

## What's new in WinUI 3.0?

Unlike WinUI 2.0 and the incremental versions that followed, WinUI 3.0 is a major update featuring more than new and improved controls and libraries to use with UWP and .NET 5 apps. In fact, the primary goal of WinUI 3.0 was not to add new controls and features beyond its current UWP counterparts. For version 3.0, the team has made WinUI a complete UI framework that can sit atop the UWP or Win32 application platforms.

## Goodbye UWP?

So, what is happening to UWP? Will my UWP apps stop working?

As previously mentioned, the plan for the UWP UI libraries is to keep providing important security updates, but they will not receive any new features going forward. All new features and updates will be developed for WinUI. New WinUI projects will support all types of Windows UI client. For existing Win32 applications, developers can incrementally upgrade parts of an application to WinUI with the **Xaml Islands** interop control. New applications will be developed in WinUI with either .NET Core, written in C# or VB, or with native C++. These clients will sit on top of either the Win32 platform or UWP. This is all possible because WinUI is developed completely in C++.

#### Note

Some features discussed in this book will not be available in the first stable **Release To Manufacturing (RTM)** version of WinUI 3.0, coming in March 2021 with Project Reunion v0.5. XAML Islands will not be available and UWP clients will not be fully supported initially. For a full list of what is planned for the first stable release of WinUI 3.0, you can reference the team's roadmap on GitHub: <https://github.com/microsoft/microsoft-ui-xaml/blob/master/docs/roadmap.md#winui-3-0-feature-roadmap>.

The fact that WinUI is developed in C++ enables React Native for Windows client apps to interoperate with the WinUI platform. Between React Native and the Uno Platform, WinUI has some great cross-platform potential.

Let's look at this new app model in the following screenshot:

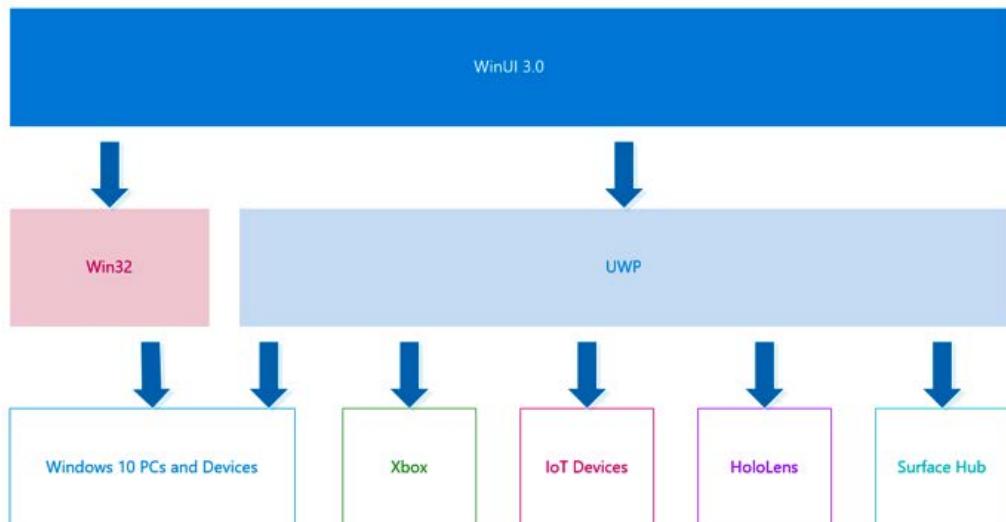


Figure 1.9 – The WinUI 3.0 app model

As you can see, there will be multiple paths available for developers to create apps for Windows PCs and tablet devices such as the dual-screen Surface Duo. Other Windows devices, such as Xbox and HoloLens, will need to follow the UWP app model under the WinUI layer.

## New features for WinUI 3.0

Are there any new features in WinUI 3.0?

While it sounded like the team was so busy creating a UI framework to replace the UWP UI libraries, they did find some time to add a few new features. The major new control available in version 3.0 is the new `WebView2` control based on the Microsoft Edge Chromium browser. Compatibility is also a feature. All XAML and Composition features available in the Spring 2019 Windows SDK will be backward-compatible, back to the Windows Creators update and later.

## Project Reunion and WinUI

WinUI 3.0 is bringing UWP and Win32 application developers together on a single set of UI libraries, but that is only the beginning. At Microsoft's Build 2020 conference, the Windows team announced **Project Reunion**, a long-term plan for bringing all Windows developers together on a single platform. WinUI 3.0 is focused on the UI layer, while Project Reunion will encompass WinUI and the entire Windows developer platform. In 2021, Microsoft will release three versions of Project Reunion and WinUI 3.x, as follows:

- *March 2021*: The first Project Reunion release, version 0.5, will feature the first *supported* release of WinUI 3.
- *May 2021*: This Project Reunion version 0.8 will include several enhancements to WinUI 3.
- *October 2021*: This will be the Project Reunion 1.0 release, with more improvements to WinUI 3.

To read more about Project Reunion and to follow its progress, you can check out their GitHub repository at <https://github.com/microsoft/ProjectReunion>. Now, let's see how WinUI compares to other Windows development frameworks.

# WinUI compared to other Windows development frameworks

Where does WinUI fit in the overall landscape of Microsoft's Windows development frameworks? Let's draw some comparisons to help answer that question, starting with those that are most similar to WinUI.

## WinUI versus UWP

This is a tricky comparison because most WinUI apps today are UWP apps at their core. In fact, WinUI 2.x are controls for UWP applications. When considering WinUI 3.0, think of WinUI as the UI framework and UWP as the application platform layer. They share the same XAML schema, base visuals, and underlying Windows APIs. Any UWP app that has the same minimum and target versions of Windows specified can add the WinUI 2.x libraries to leverage the new and updated features.

A key difference between apps that use WinUI versus traditional UWP apps is access to new and updated controls and other visual elements without requiring an updated Windows SDK. This enables developers to bring apps with the same look and features to more users across multiple versions of Windows 10. This differentiator makes for happier developers and users.

## WinUI versus WPF

WinUI and WPF have many similarities. Both are application frameworks, and both types of apps rely on XAML to define UI elements. This means that they both offer the same separation of UI and business logic when implementing the MVVM pattern. WPF XAML has the same concepts of styles, resources, data binding, and adaptiveness of the UI layout.

## WinUI advantages

A significant performance advantage of WinUI—and UWP apps in general—is the availability of **compiled bindings**. We will discuss compiled bindings in more detail later. For now, just know that using them offers a performance boost over traditional data binding in both UWP and WPF. Compiled bindings are identified by the use of the `x:Bind` syntax in XAML, rather than `Binding`.

Another advantage of WinUI over WPF is that it is more secure by default. Sandboxed access to users' filesystems and devices can give users a sense of ease, knowing that malicious behavior has limits to which hardware and data it can access. WPF apps' access is only limited by the configuration of Windows **User Account Control (UAC)** on the PC.

## WPF advantages

The primary advantage of WPF applications is the fact that they are not directly tied to minimum versions of Windows 10. WPF apps target a .NET Framework or .NET Core version. Any version of Windows that supports the target .NET version can run that WPF app. This significantly increases the potential user base of WPF apps. In fact, WPF apps can be deployed and run on Windows 7, something not possible with UWP or WinUI.

### Note

There is a project called **Uno Platform** that enables WinUI XAML to run on iOS and Android, leveraging Xamarin and on the web with **WebAssembly**. These WinUI web apps can run in the browser on previous versions of Windows, including Windows 7. The Uno Platform goal and tagline is *WinUI Everywhere*.

Learn more about Uno Platform at <https://platform.uno/>.

Learn more about WebAssembly at <https://webassembly.org/>.

The secure-by-default nature of WinUI apps provides WPF apps with a greater degree of flexibility and power. Many types of applications will require full access to the Windows filesystem or to particular devices and drivers—for example, it would be much easier to create a custom file manager application with complete access to a computer's filesystem. While WinUI apps can now request this access in newer versions of Windows 10, this is available by default in any version with WPF.

A new WPF advantage emerged with the releases of .NET Core 3.x and .NET 5. .NET developers can now create WPF apps with .NET Core, bringing performance and deployment advantages of .NET Core to WPF developers. For instance, applications targeting different versions of .NET Core can be deployed side by side on a machine without creating version conflicts.

The difference in deployment models can be debated as to which framework has an advantage. The easiest way to deploy a WinUI app is through the Windows Store. The easiest way to deploy a WPF app with .NET Framework is via an installer package. WPF apps can be deployed through the Store through Windows Containers, and WinUI apps can be deployed without the Store with MSIX installers. WinUI deployment will be covered in detail in *Chapter 12, Build, Release, and Monitor Apps with Visual Studio App Center* and *Chapter 13, Packaging and Deployment Options and the Windows Store*.

## WinUI versus Windows Forms (WinForms)

WinForms is a .NET UI framework that was introduced with .NET Framework 1.0. Developers can easily create a WinForms UI with the visual design surface in Visual Studio, which generates C# or VB code that creates the UI at runtime. Most of the advantages and disadvantages of WPF also apply to WinForms: security, deployment, and .NET Core—WinForms apps can also be created with .NET Core 3 and later.

### WinUI advantages

Similarities between WinUI and WPF are their primary advantages over WinForms: data binding, adaptive layout, and a flexible styling model. These advantages all stem from the use of XAML for UI layout. Another advantage of XAML is offloading render processing from the **central processing unit (CPU)** to the **graphics processing unit (GPU)**. WinUI controls inherit the Windows 10 styles by default and have a more modern appearance than WinForms controls. WinUI applications also handle **dots per inch (DPI)** scaling and touch input well. The WinForms UI framework matured before touch input and DPI scaling were a concern for Windows developers.

A primary advantage of WinUI over WPF also applies to WinForms: security by default.

### WinForms advantages

In addition to the advantages that WinForms shares with WPF over WinUI—greater access to Windows, .NET Core apps, and Windows compatibility—WinForms also has a well- deserved reputation for rapid UI development. If you need to create a simple Windows application in a minimal amount of time, the drag-and-drop WinForms designer is easy and intuitive. Many experienced Windows developers still default to WinForms when tasked with creating a simple utility or UI test harness for a .NET library.

## Summary

We have covered a lot of the history of Windows application development in this chapter. We learned about the origins of UWP and its roots in Windows 8 apps, and learned of the benefits of XAML when building Windows UIs. We had a taste of what some simple WinUI app code and UIs look like. Finally, we examined the recent history of WinUI versions and how the new version 3.0 is a complete replacement for the UWP UI libraries and a viable option for WPF developers going forward.

This will give you a good foundation of what's to come as we start building an app with WinUI in the chapters ahead. In the next chapter, you will set up your development environment, learn about the app project that we will create throughout the book, and create your first WinUI 3.0 project. When we get to *Chapter 3, MVVM for Maintainability and Testability*, we will refactor the app to use the MVVM pattern. This will set us up with a solid, maintainable design as we later add to and extend the app throughout the rest of the book.

## Questions

1. Which version of Windows first introduced UWP apps to developers?
2. What is the name of the pattern commonly used by WinUI and other XAML developers to separate UI logic from business logic?
3. WinUI and WPF apps can share the same XAML. True or false?
4. Which was the first Microsoft UI framework to use XAML to define the UI?
5. What was the version number of the first WinUI release?
6. What is one of the benefits of developing with WinUI over WinForms?
7. Can WinUI apps only be developed with .NET languages?
8. Challenge: Create a Style that will apply to `Button` elements.

# 2

# Configuring the Development Environment and Creating the Project

To get started with WinUI development, it is important to install and configure Visual Studio for Windows development. A WinUI developer must also understand the basics of **Universal Windows Platform (UWP)** development with **Extensible Application Markup Language (XAML)** and C#, which we started learning in *Chapter 1, Introduction to WinUI*. However, the best way to understand the development concept is to get your hands on a real project. We will do that here in this chapter.

After setting up your Visual Studio environment, you will create the beginnings of a project that we will be building throughout the rest of the book.

In this chapter, you will learn the following topics:

- How to set up a new Visual Studio installation for Windows application development
- How to create a new WinUI project, add a few controls, and run the project for the first time
- The anatomy of a new WinUI project and why each part is important
- How XAML can be used to build flexible, performant **user interfaces (UIs)**
- How WinUI fits with UWP and the role of each layer in the overall application architecture
- How to work with WinUI controls and customize them through changes in the XAML markup or C# code
- How to handle some basic UI events

If you are new to WinUI and other XAML-based development platforms, by the end of this chapter you should be starting to feel comfortable working with WinUI projects.

## Technical requirements

To follow along with the examples in this chapter, the following software is required:

- Windows 10 version 1803 (build 17134) or newer
- Visual Studio 2019 version 16.9 or newer with the following workloads: .NET desktop development and UWP development
- WinUI 3.0 project templates

The source code for this chapter is available on GitHub at this URL: <https://github.com/PacktPublishing/-Learn-WinUI-3.0/tree/master/Chapter02>.

### Note

At the time of writing, the templates are available as an extension in **Visual Studio Marketplace**. After WinUI 3.0 has been released, the templates will probably be included with Visual Studio.

# Installing Visual Studio and Windows development workloads

The first step to follow when starting with WinUI development is to install Microsoft's Visual Studio **integrated development environment (IDE)**. You can download the current version of Visual Studio 2019 from <https://visualstudio.microsoft.com/downloads/>. Visual Studio Community 2019 edition is free for personal use and has all the features you will need to build WinUI applications.

## Tip

If you want to try new Visual Studio features before they are released, you can install the Visual Studio Preview version from <https://visualstudio.microsoft.com/vs/preview/>. The Preview version is not recommended for development of production applications as some features are unstable.

During installation, you can select workloads for any type of application that you want to create. For WinUI development, you must select these three workloads:

- **.NET desktop development**
- **.NET Core cross-platform development**
- **Universal Windows Platform development**

An overview of the **Workloads** section is shown in the following screenshot:

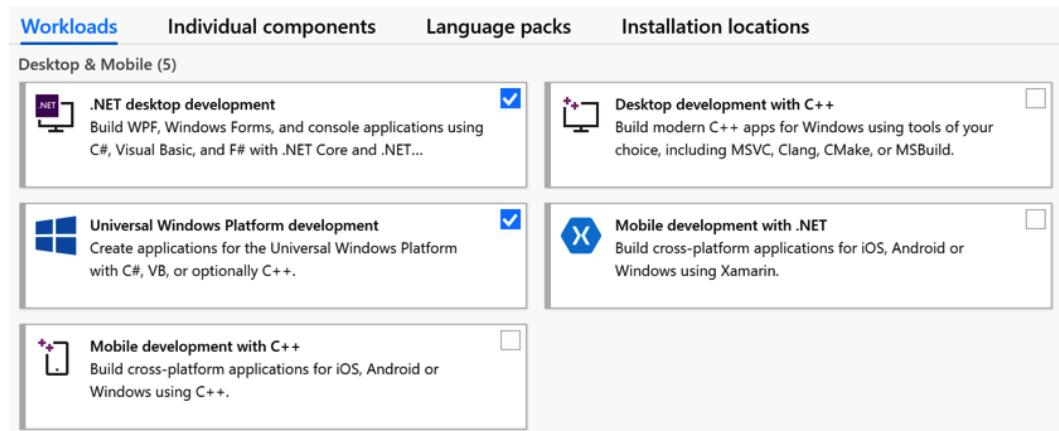


Figure 2.1 – Visual Studio Installer workload selection, showing the first two workloads to select

**Note**

If you are a C++ developer and would like to build WinUI apps with C++, you must select the **Desktop development with C++** workload and the optional **C++ (v142) Universal Windows Platform tools** component within the **Universal Windows Platform development** workload. However, WinUI C++ applications are beyond the scope of this book.

When continuing to the next step, the Visual Studio Installer will download and install the selected workloads and components. When setup is complete, launch Visual Studio. You will be prompted to sign in with a Microsoft account. Linking Visual Studio to your account will enable Visual Studio to sync your settings, link any available licenses, and link your Windows Store account after it has been created. We will discuss more about the Windows Store and application distribution in *Chapter 13, Packaging and Deployment Options and the Windows Store*.

## Adding the WinUI app templates

It's time to install the WinUI 3.0 templates for Visual Studio:

**Note**

This step may not be necessary by the time WinUI 3.0 has launched. At the time of this writing, the project was in preview release and some additional installation steps were required to install the WinUI templates in Visual Studio. While it is likely that WinUI will be installed through Visual Studio's **Manage Extensions** window, it may come bundled with Project Reunion in another extension or installer.

1. In Visual Studio, open the **Manage Extensions** window by selecting **Extensions | Manage Extensions**. The **Manage Extensions** window is shown in the following screenshot:

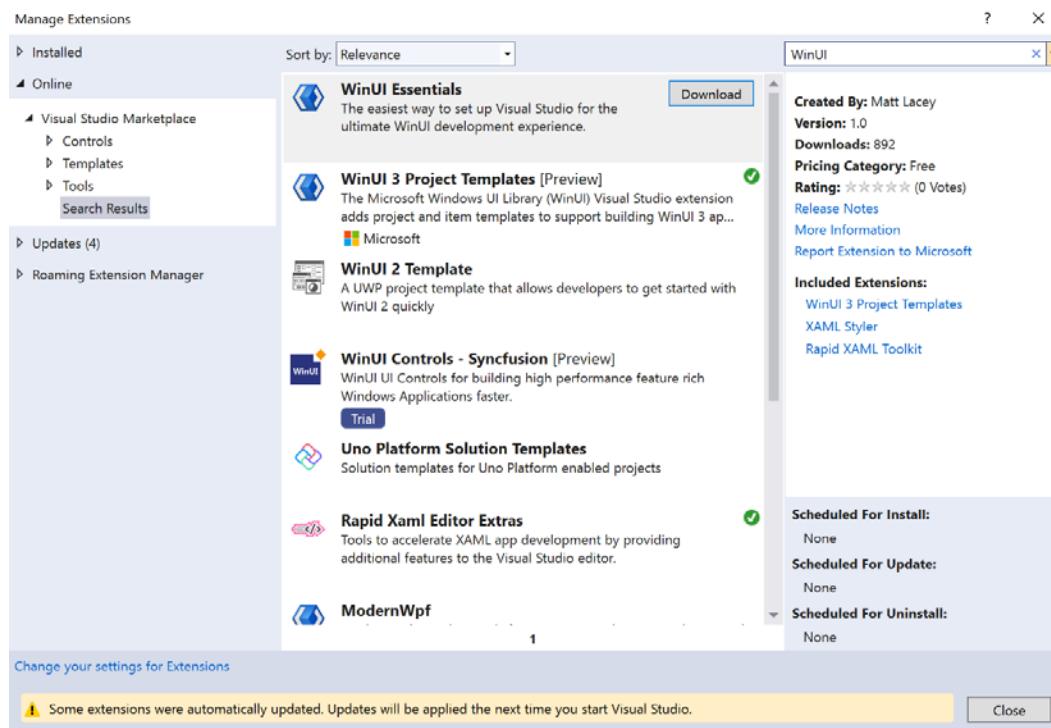


Figure 2.2 – Installing the WinUI 3 project templates

2. Enter **WinUI** in the search field and select **WinUI 3 Project Templates** from the results.
3. Click the **Download** button and restart Visual Studio to complete the installation.

You can alternatively download the templates from Visual Studio Marketplace at <https://marketplace.visualstudio.com/items?itemName=Microsoft-WinUI.WinUIProjectTemplates>. After downloading, double-click the downloaded .vsix file to install it on Visual Studio.

**Note**

At the time of this writing, if you want to enable UI debugging with WinUI projects in Visual Studio, you must select the **Enable UI Debugging Tooling in WinUI 3 Projects** option in **Tools | Options | Preview Features**.

Now, it's time to start building our WinUI application.

## Introducing the application idea

The application we're going to build is a tool called **My Media Collection** that catalogs your media library. Because digital media is becoming more popular with the passage of time, we can design the application to include both physical and digital media. The application will be able to catalog different types of media, including music, video, and books. We will add some features that will only light up for some of the media types. Physical media (books, DVDs, and CDs) are often loaned to friends. The application will help you remember who borrowed your favorite book at your last family party.

## Reviewing the application features

Before we dive in and create the new project, let's get organized. It helps to understand what you're going to build so that you can track your progress while progressing through each chapter. Let's start with a high-level look at the application's features, as follows:

- View all media
- Filter the media library by the following: **Media Type (Music, Video, or Book)**, **Medium** (This will vary by media type. Examples include **CD, Record, DVD, Blu-Ray, Hardcover, Paperback, Digital**), or **Location (In Collection or Loaned)**
- Add a new media item
- Edit a media item
- Mark an item as loaned or returned
- Sign in to the application
- Back up (or restore) the catalog to OneDrive

The application will use features of the **Windows Community Toolkit**, which can simplify things such as login and reading/writing files on OneDrive. The application's data will be stored in a local SQLite database, allowing for online or offline access to the media collection. You'll even be able to send email reminders if one of your friends is taking a little too long in returning one of the items in your collection.

## WinUI in UWP versus WinUI in Desktop projects

In this chapter, we will be building an application with the **WinUI in UWP** project template. The other option is to use the **WinUI in Desktop** project template. The primary difference between the two project types in WinUI 3.0 is the app model.

The UWP app model is based on the `CoreApplication` class (see Microsoft Docs for more about `CoreApplication`: <https://docs.microsoft.com/en-us/uwp/api/windows.applicationmodel.core.coreapplication>). Building on top of `CoreApplication` and **Windows Runtime (WinRT)** provides access to changes in the application state, UI frameworks (UWP), and window management.

A **WinUI in UWP** project uses .NET Native, while a **WinUI in Desktop** project targets .NET 5. In a future WinUI release, **WinUI in UWP** projects will also target .NET 5 (or 6, depending on the implementation timeline). The WinUI team did not have time to port the UWP project type over to .NET 5 for the WinUI 3.0 release.

The other difference in a **WinUI in Desktop** project is that a **Windows Application Packaging Project** is included in a newly created project. This provides the same modern packaging solution that is native to a **UWP** or a **WinUI in UWP** project. We will learn more about packaging WinUI applications in *Chapter 13, Packaging and Deployment Options and the Windows Store*. Now, let's get started with our first project.

## Creating your first WinUI project

It's time to start building the project. To do so, proceed as follows:

1. Launch Visual Studio, and from the opening screen select **Create a new project**, as illustrated in the following screenshot:

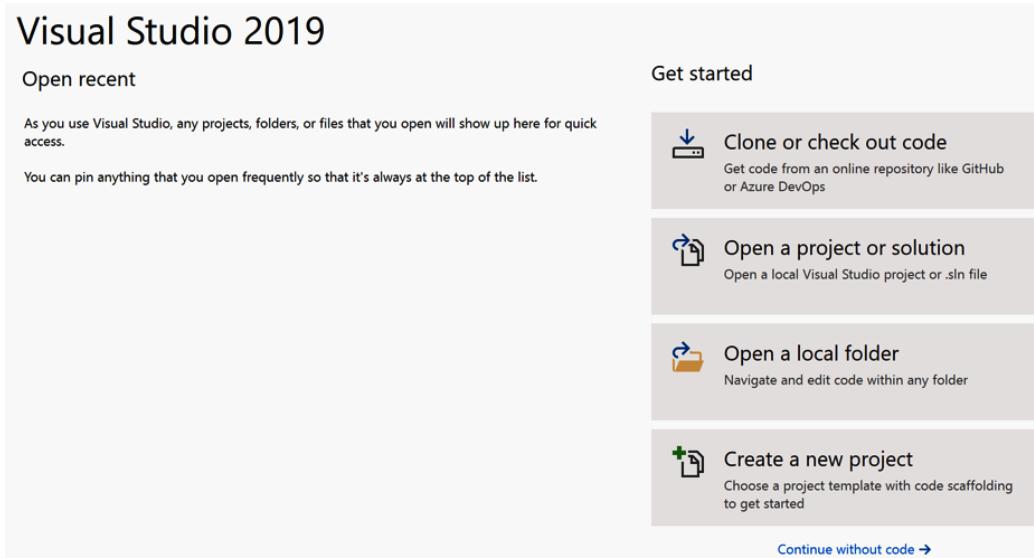


Figure 2.3 – The Visual Studio initial dialog

2. On the **Create a new project** screen, enter **winui** in the **Search for templates** field, select the **Blank App (WinUI in UWP)** C# template, and click **Next**, as illustrated in the following screenshot:

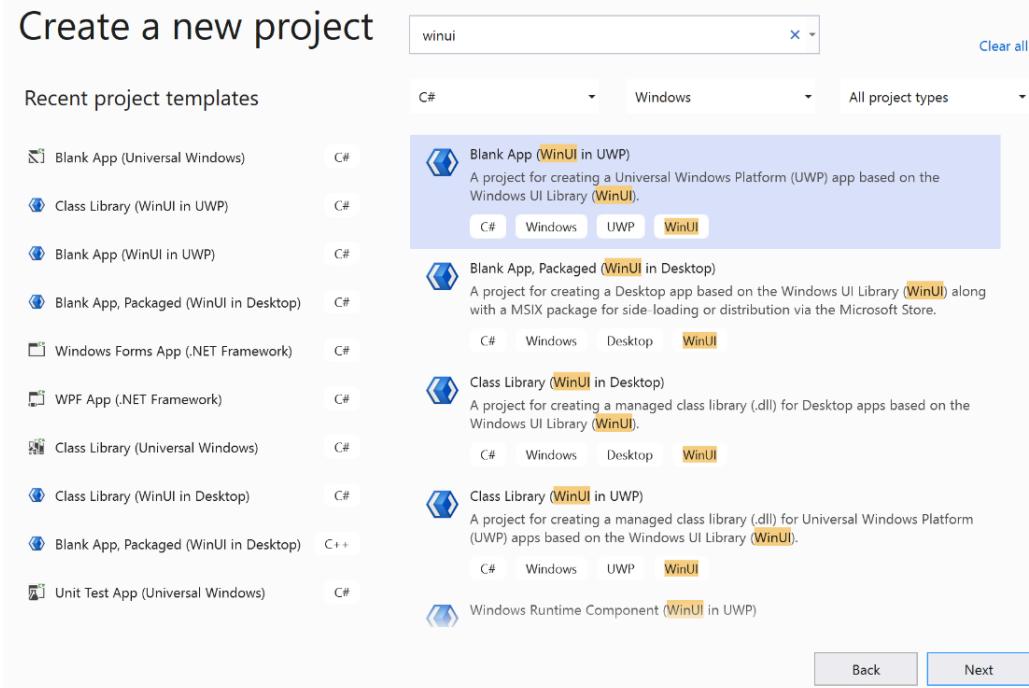


Figure 2.4 – Selecting the project template

**Tip**

Be sure to select the **C#** project template and not **C++**. You can filter the project types to show only C# projects by changing the language filter from **All languages** to **C#**.

3. Name the project `MyMediaCollection`, leave the rest of the fields set to their default values, and click **Create**. You will be prompted to select the Windows versions to target with your app. WinUI 3.0 requires that the following minimum versions be selected:

**Target Version:** Build 18362 or higher

**Minimum Version:** Build 17134 or higher

**Note**

It is up to you if you want to select newer versions for your app, but you will be preventing users of these older Windows 10 versions from being able to install the app. The **Target** version must be equal to or greater than the **Minimum** version. If you are unsure of what to choose, it is okay to stick with the default values.

4. Now, the project has been created and Visual Studio has loaded. Run the application and see what the template has provided for `MainPage`. You should see an empty window hosting the page. Next, we will see what Visual Studio has created to get us started.

**Note**

To run and debug WinUI apps on Windows 10, you must update your Windows settings to enable **Developer Mode**. To do this, follow these steps:

1. Open **Settings** from the Start menu.
2. Type **Developer** in the search bar and select **Developer settings** in the search results area.
3. On the **For developers** page that appears, switch the **Developer Mode** toggle switch on, if it is not already turned on. Enabling this allows developers to sideload, run, and debug unsigned apps, and enables some other developer-focused Windows settings. You can get more information here: <https://docs.microsoft.com/en-us/windows/uwp/get-started/enable-your-device-for-development>.

## Anatomy of a WinUI in UWP project

Now that we have a new empty WinUI project loaded in Visual Studio, let's examine the different pieces. In **Solution Explorer**, you will see two XAML files named `App.xaml` and `MainPage.xaml`. We will start by discussing the purpose of each of these. Both files can be seen in the following screenshot:

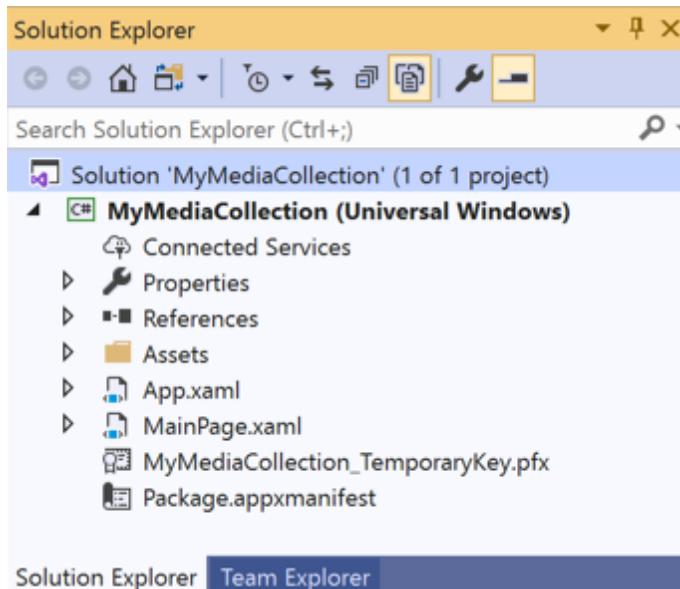


Figure 2.5 – The new WinUI app in Solution Explorer

## Reviewing App.xaml

The `App.xaml` file, as its name implies, stores resources available across an entire application. If you have any templates or styles that will need to be used across multiple pages, they should be added at the Application level.

The new project's `App.xaml` file will contain some markup to start out, as illustrated in the following code snippet:

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

```
<ResourceDictionary>
    <ResourceDictionary.MergedDictionaries>
        <XamlControlsResources
            xmlns="using:Microsoft.UI
                .Xaml.Controls" />
        <!-- Other merged dictionaries here -->
    </ResourceDictionary.MergedDictionaries>
    <!-- Other app resources here -->
</ResourceDictionary>
</Application.Resources>
</Application>
```

We will cover some XAML basics in the next section. For now, you should know that the `Application.Resources` section will contain all of the resources to be shared across an application. Within this section, the `ResourceDictionary.MergedDictionaries` section contains references to other XAML files whose resources will be shared with the entire application. This allows developers to organize shared content into multiple resource files, leading to better organized and more maintainable code. This also enables the sharing of third-party resources across the application. For example, this file merges `XamlControlsResources` from the `Microsoft.UI.Xaml.Controls` namespace. These are the resources for the WinUI controls that we will be using to build our app.

## Reviewing App.xaml.cs

In **Solution Explorer**, expand the `App.xaml` file's node, and you will see there is another file named `App.xaml.cs` nested underneath. This is referred to as a **code-behind file**. It is a partial C# class that, in tandem with the XAML file, defines the `App` class. If you open the C# file, you will see it contains quite a bit of code. This is where you will handle any application-wide events. These are the event handlers added by default:

- `OnLaunched`: If you need to execute any specific logic when your app is first launched, it should be added here. This is also where any application arguments passed to the app can be handled, including instructions to navigate to a specific page on launch.
- `OnNavigationFailed`: If any page cannot be loaded, this handler will fire. There is some default exception handling added here.
- `OnSuspended`: When your app is suspended by Windows, this handler will be invoked. If you need to save any application state, do it here.

## Reviewing MainPage.xaml

The `MainPage.xaml` file contains the `MainPage` WinUI page that will be loaded when the application launches. You can see this in the `OnLaunched` event handler in `App.xaml.cs`, as illustrated in the following code snippet:

```
rootFrame.Navigate(typeof(MainPage), e.Arguments);
```

In a new blank WinUI app, the `Page` will contain an empty `Grid` layout control. Try replacing the `Button` control with `TextBlock` with a `Text` property of `Media` and replace the `StackPanel` with a `Grid`. The result should look like this:

```
<Page
    x:Class="MyMediaCollection.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
        presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:MyMediaCollection"
    xmlns:d="http://schemas.microsoft.com/expression/blend/
        2008"
    xmlns:mc="http://schemas.openxmlformats.org/
        markup-compatibility/2006"
    mc:Ignorable="d"
    Background="{ThemeResource
    ApplicationPageBackgroundThemeBrush}">
    <Grid>
        <TextBlock Text="Media"/>
    </Grid>
</Page>
```

Run the app with the **Debug | Start Debugging** menu item or the **Start Debugging** button on the Visual Studio toolbar, and you should see something like this:



Figure 2.6 – MainPage with a TextBlock added

We'll make this page a bit more functional later in the chapter. For now, let's finish our review of the project structure.

## Reviewing MainPage.xaml.cs

The code-behind file for the `MainPage.xaml.cs` page contains only a call to `InitializeComponent()` in the constructor. Later, we will add some code here to populate some sample data and handle events on the page. In a well-designed MVVM app, the code-behind files for your pages will have very little code. Most of the code will reside in the `ViewModel` classes.

## Reviewing the project references

All of your project's references to NuGet packages, **dynamic link libraries (DLLs)**, and other projects in the solution will appear in the **References** folder in **Solution Explorer**. Your WinUI project will reference these two NuGet packages:

- `Microsoft.NETCore.UniversalWindowsPlatform`: The UWP app platform components required by WinUI.
- `Microsoft.WinUI`: The WinUI components for the application's UI layer. This replaces the default UWP library that brings in the controls to our project.

There are also three DLL binary references to some required C++ and UWP components. Do not modify or remove any of these references.

## Reviewing the project properties

If you right-click the project in **Solution Explorer** and select **Properties** from the context menu, you can view and modify the project properties. You won't often need to make any changes here. These are a few of the properties you might need to modify from time to time:

- **Assembly Name**: You can change the name of the output assembly that is compiled by the project.
- **Min Version and Target Version**: The Windows versions that were selected when creating the project can be modified here.
- **.NET Native settings**: There are two .NET Native-related settings that can be selected. These can provide some performance optimizations.

### Tip

For more information about .NET Native in Windows apps, you can read this article on Microsoft Docs: <https://docs.microsoft.com/en-us/dotnet/framework/net-native/net-native-and-compilation>.

The primary difference between a new UWP project and a new WinUI 3.0 project is that the controls and other objects referenced in the `Windows.UI.Xaml.*` namespaces are now referenced in WinUI's `Microsoft.UI.Xaml.*` namespaces. One other difference is that in the `App.xaml` file of the UWP project, it's not necessary to import the controls' resources. The remaining differences are mostly hidden from app developers in the project file.

Now that you've become familiar with the WinUI project, let's start building the UI for the `MainPage`. We will start with some of the more common XAML controls and concepts.

## XAML basics

It's time to start building the main screen of the **My Media Collection** application. The focal point of the application will be the media items in the collection. To display that list, we are going to need a few things, as follows:

- A **Model** class that defines an item in the collection
- Some code to bind the collection of items to the UI
- A XAML control to display the items

## Building the model

We will start by building the model for the **My Media Collection** application. A **model** defines an entity and its attributes. Earlier in the chapter, we discussed some of the items' attributes we want to display in the UI. In order to display and (eventually) persist this information, we must create the model.

The initial version of our model will consist of two **enums** (`ItemType` and `LocationType`) in an `Enums` folder and two **classes** (`Medium` and `MediaItem`) in a `Model` folder, as illustrated in the following screenshot:

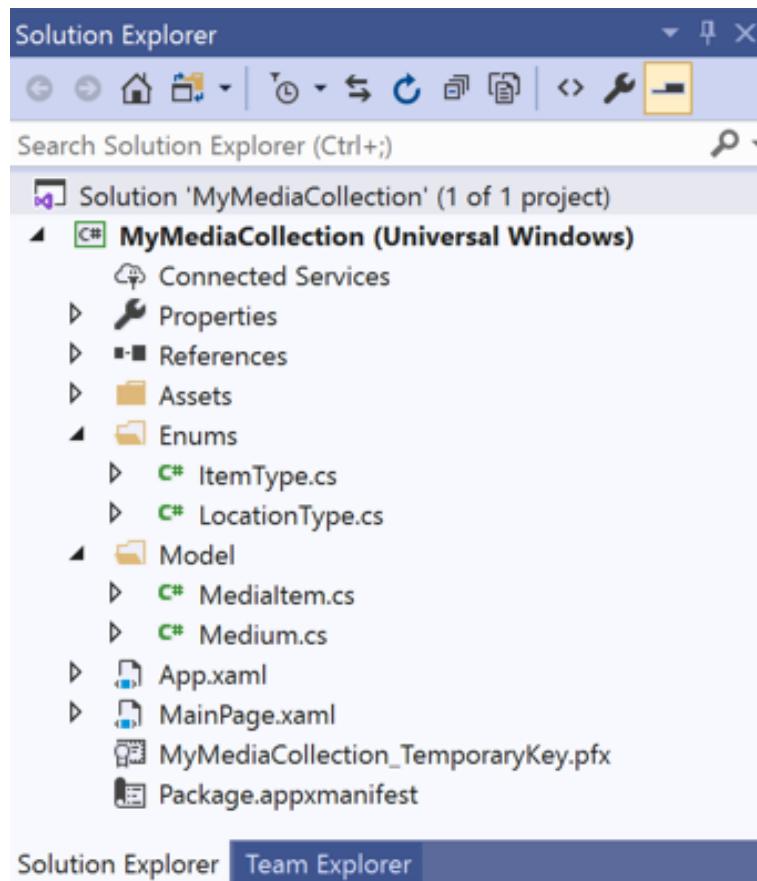


Figure 2.7 – Solution Explorer displays the new model and enum files

To add a new folder to the project, right-click on the **MyMediaCollection** project in **Solution Explorer** and select **Add | New Folder**. Enter **Enums** as the folder name and hit *Enter*. Repeat this process to add the **Model** folder.

Next, right-click the project file and select **Add | Class**. In the **Add New Item** dialog that appears, the **Class** file type will be selected. Because there is no template for a new enum, keep this selection, change the name to `ItemType`, and click the **Add** button, as illustrated in the following screenshot:

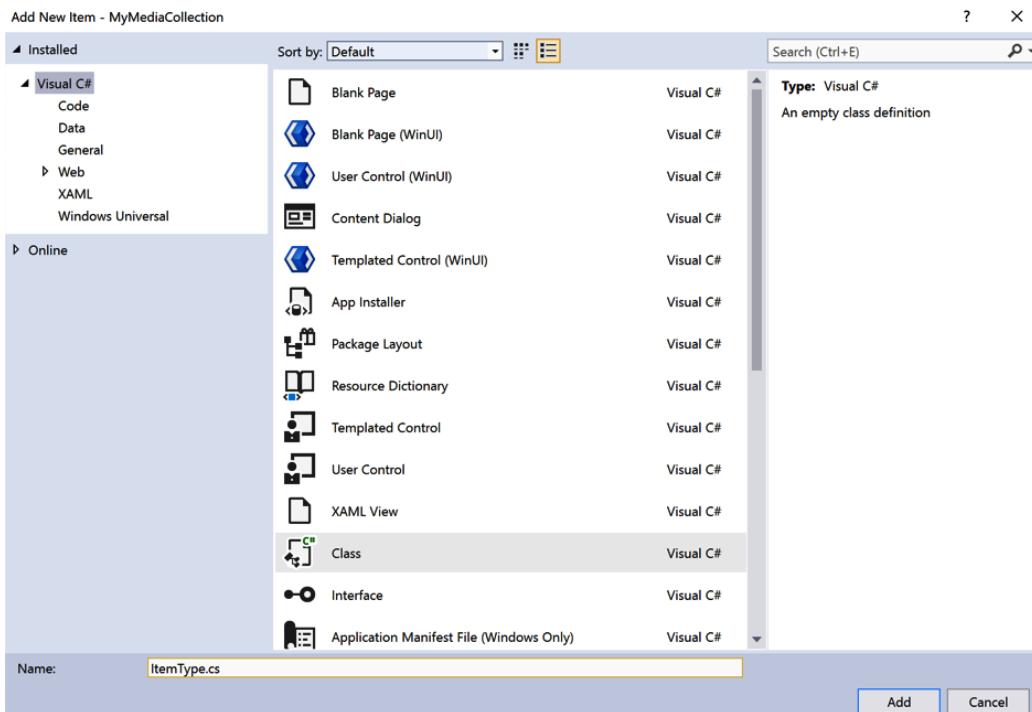


Figure 2.8 – The Add New Item dialog

The code for `ItemType` is created and displayed. Change the `class` definition to `enum` and add the `public` keyword, preceding the `enum` keyword. The `ItemType` enum contains three items named `Music`, `Video`, and `Book`. When you are finished, the definition of the `ItemType` enum will look like this:

```
public enum ItemType
{
    Music,
    Video,
    Book
}
```

Follow the same steps to create a `LocationType` enum, defined as follows:

```
public enum LocationType
{
    InCollection,
    Loaned
}
```

Using what you have learned, create two classes named `Medium` and `MediaItem` in the `Model` folder. The `Medium` class represents a specific medium such as *hardcover* or *paperback*, while the `MediaType` property assigns an `ItemType` to which the `Medium` class belongs. For these, the valid `ItemType` would be `Book`. When you are finished, the two new classes will look like this:

```
public class Medium
{
    public int Id { get; set; }
    public string Name { get; set; }
    public ItemType MediaType { get; set; }
}

public class MediaItem
{
    public int Id { get; set; }
    public string Name { get; set; }
    public ItemType MediaType { get; set; }
    public Medium MediumInfo { get; set; }
    public LocationType Location { get; set; }
}
```

**Note**

In order to reference the enum types in the `Enums` folder from the two classes, a `using` declaration will need to be added at the top of each new class, like this: `using MyMediaCollection.Enums;`.

In addition to the properties discussed earlier in the chapter, an `Id` property has been added to the `MediaItem` class to uniquely identify each item in the collection. This will be useful later when we start persisting data.

## Creating sample data

With the model classes in place, we're ready to add some code that will create three media items to display in the UI. This will help to visualize things as we start creating our list of items on the main screen. When this step is complete and we're ready to move on to the ability to add items through the app, this code will be removed.

For now, we are going to add this code in the `MainPage.xaml.cs` code-behind file. Later, in *Chapter 3, MVVM for Maintainability and Testability*, this type of code will be added in a `ViewModel` file. The `MainPage` will contain only presentation logic and will not be responsible for creating or fetching the data that populates the UI:

1. First, open `MainPage.xaml.cs` and create a method called `PopulateData()`. This method will contain the code that creates three `MediaItem` objects (a CD, a book, and a Blu-ray) and adds them to a `List` named `_items`, as illustrated in the following code snippet:

```
public void PopulateData()
{
    if (_isLoaded) return;
    _isLoaded = true;
    var cd = new MediaItem
    {
        Id = 1,
        Name = "Classical Favorites",
        MediaType = ItemType.Music,
        MediumInfo = new Medium { Id = 1,
            MediaType = ItemType.Music, Name = "CD" }
    };
    var book = new MediaItem
    {
        Id = 2,
        Name = "Classic Fairy Tales",
        MediaType = ItemType.Book,
        MediumInfo = new Medium { Id = 2,
            MediaType = ItemType.Book, Name = "Book" }
    };
    var bluRay = new MediaItem
    {
```

```

        Id = 3,
        Name = "The Mummy",
        MediaType = ItemType.Video,
        MediumInfo = new Medium { Id = 3,
            MediaType = ItemType.Video, Name = "Blu Ray" }
    };
    _items = new List<MediaItem>
    {
        cd,
        book,
        bluRay
    };
}

```

You will need to add using statements to `MainPage.xaml.cs` for the `MyMediaCollection.Model` and `MyMediaCollection.Enums` namespaces.

2. Define `_items` `IList` and `isLoading` `bool` as a private class member. We will later change the collection of items to be an `ObservableCollection`. An `ObservableCollection` is a special collection that notifies data-bound items on the UI when items have been added or removed from the collection. For now, an `IList` will suit our needs. The code can be seen in the following snippet:

```

private IList<MediaItem> _items { get; set; }
private bool _isLoading;

```

3. Next, add a line of code to the `MainPage` constructor, after the call to `InitializeComponent()` to call `PopulateData()`. Any code added to the `Page` or `UserControl` constructors must be added after this initialization code. This is where all of the XAML code in `MainPage.xaml` is initialized. If you attempt to reference any of those elements before the call to `InitializeComponent`, it will result in an error. The constructor should now look like this:

```

public MainPage()
{
    this.InitializeComponent();
    PopulateData();
}

```

We will return to add some data-binding logic to this file after creating some UI components in the XAML file. Let's go and do that now.

## Building the initial UI

Earlier in the chapter, we added a single `TextBlock` to the `MainPage.xaml` file with the `Text` property set to `Media`. We are going to add a `ListView` control beneath the `TextBlock` that we added earlier. A `ListView` control is a powerful and flexible control to display a list of items in a vertical list. It is similar to the **WinForms** `ListBox` control in basic list functionality (item selection, multi-selection, and automatic scroll bars), but each list item can be templated to display in just about any way imaginable:

**Note**

For more information about the `ListView` class, documentation with sample code and markup is available on Microsoft Docs here: <https://docs.microsoft.com/en-us/uwp/api/Windows.UI.Xaml.Controls.ListView>.

1. First, add some markup to create two rows inside the top-level `Grid` control. The `RowDefinitions` should be added before the existing `TextBlock` element. The `TextBlock` will remain in the first row, and we'll add the `ListView` control to the second row, as illustrated in the following code snippet:

```
<Grid.RowDefinitions>
  <RowDefinition Height="Auto"/>
  <RowDefinition Height="*"/>
</Grid.RowDefinitions>
```

As you can see, we are auto-sizing the first `RowDefinition` to fit to the size of the `TextBlock` and allocating the remaining space inside the `Grid` to the `ListView` control's `RowDefinition` by using `"*"`. If you assign more than one row with `"*"`, those rows will split the remaining available space equally. This is the default value for the `Height` attribute. It would be sized the same if it were omitted, but most developers explicitly include it for completeness and improved readability.

**Note**

For more information about row sizing, see the following Microsoft Docs page: <https://docs.microsoft.com/en-us/uwp/api/windows.ui.xaml.controls.rowdefinition.height>.

2. Next, add a `ListView` control to the `Grid`, assign it to `Grid.Row 1` (row numbering is 0-based, meaning the first row is row 0), and name it `ItemList`. It is not necessary to name your XAML elements. This is only required if you either want to reference them by `ElementName` in XAML data binding or as a variable in the code-behind file. Naming a control causes a variable to get created in the `InitializeComponent` call, and you should only name your controls if you need to reference them elsewhere. The code can be seen in the following snippet:

```
<ListView Grid.Row="1" x:Name="ItemList"
          Background="Aqua" />
```

By also setting the background color to `Aqua`, we can see the `ListView` in the application's main window before we populate any data. Run the app, and it should look something like this:



Figure 2.9 – A ListView added to the UI

Let's return to the code-behind file to start wiring up the data binding.

#### Note

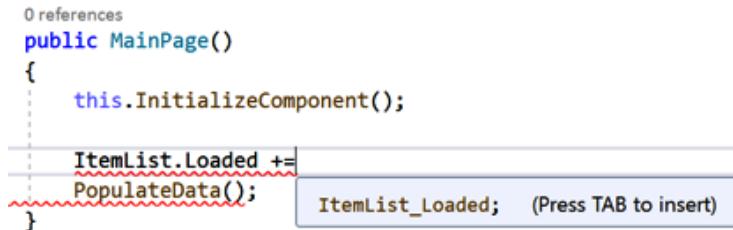
If you are new to the concept of data binding in UI programming, Microsoft Docs has a great overview of data-binding concepts in Windows development at the following web page: <https://docs.microsoft.com/en-us/windows/uwp/data-binding/>.

## Completing the data-binding initialization

Return to the `MainPage.xaml.cs` file and add a line of code before the `PopulateData()` method call, as follows:

```
ItemList.Loaded += ItemList_Loaded;
```

After typing `+=`, Visual Studio **IntelliSense** will prompt you to press `<TAB>` to automatically create the `ItemList_Loaded` event handler, as illustrated in the following screenshot:



The screenshot shows a code editor with the following C# code:

```
0 references
public MainPage()
{
    this.InitializeComponent();

    ItemList.Loaded +=
        PopulateData();
}
```

The line `ItemList.Loaded +=` is underlined with a red wavy line, indicating a potential error. A tooltip box appears over the `+=` operator with the text `ItemList_Loaded; (Press TAB to insert)`.

Figure 2.10 – Inserting the `ItemList_Loaded` event handler

Alternatively, if you have typed the entire line, press `Ctrl + .` you'll be prompted to create the new method.

To keep the `MainPage` constructor simple and keep our data-loading code together, move the `PopulateData` call to the `ItemList_Loaded` method. Then, add the two other lines of code to the new event handler, as follows:

```
private void ItemList_Loaded(object sender,
    Microsoft.UI.Xaml.RoutedEventArgs e)
{
    var listView = (ListView)sender;
    PopulateData();
    listView.ItemsSource = _items;
}
```

This code is called after the `ItemsList` control has completed loading in the UI. We're getting the instance of the control from the `sender` parameter and setting `ItemsSource` of the list to the `_items` collection that was loaded in `PopulateData()`. Now, we have data in the list, but things don't look quite right, as can be seen from the following screenshot:

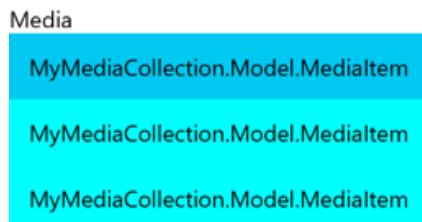


Figure 2.11 – A `ListView` with three rows of data

The `ListView` is displaying three rows for our three sample data items, but it's displaying the data type for each item instead of the data. That's because we haven't told the `ListView` which properties it should display from the items in the collection. By default, the list will display whatever is returned by an object's `ToString()` method. If `ToString()` is not overridden, the data type name of the class is returned.

## Creating the DataTemplate and binding the UI

Let's return to `MainPage.xaml` and tell the `ListView` which data we want to see for each item in the list. Try making this change with the application still running. If you have enabled UI debugging for WinUI 3 projects, you should see the reload of the UI without having to restart the debugging session. Customizing the appearance of each `ListView` item is accomplished by defining a `DataTemplate` inside the `ListView`. `ItemTemplate`. A `DataTemplate` can contain any WinUI controls we need to lay out each item. Let's keep it simple for now and add a `Grid` containing two columns. Each column will contain a `TextBlock`. The entire parent grid should now look like this:

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="*"/>
    </Grid.RowDefinitions>
    <TextBlock Text="Media"/>
    <ListView Grid.Row="1"
              x:Name="ItemList"
              Background="Aqua">
        <ListView.ItemTemplate>
            <DataTemplate x:DataType="model:MediaItem">
                <Grid>
                    <Grid.ColumnDefinitions>
                        <ColumnDefinition Width="100"/>
                        <ColumnDefinition Width="*"/>
                    </Grid.ColumnDefinitions>
                    <TextBlock Text="{x:Bind
Path=MediumInfo.Name}"/>
                    <TextBlock Grid.Column="1"
                              Text="{x:Bind Path=Name}"/>
                </Grid>
            </DataTemplate>
        </ListView.ItemTemplate>
    </ListView>
</Grid>
```

```
</DataTemplate>
</ListView.ItemTemplate>
</ListView>
</Grid>
```

We are beginning to create some more complex XAML and need to keep formatting in mind to optimize its readability.

**Note**

To quickly reformat your XAML, you can use the *Ctrl + K | D* keyboard shortcut. There is also an extension on **Visual Studio Marketplace** called **XAML Styler**. You can search for it in the **Manage Extensions** dialog or get more info on the Marketplace here: <https://marketplace.visualstudio.com/items?itemName=TeamXavalon.XAMLStyler>.

To enable each `TextBlock` to bind to the properties of `MediaItem`, we have to set a `x:DataType` property on the `DataTemplate`. To resolve the `MediaItem`, a namespace declaration needs to be added to the `Page` definition, like this:

```
xmlns:models="using:MyMediaCollection.Model"
```

A shortcut to add this `using` statement is to place your cursor in the `x:DataType` and press *Ctrl + .* Visual Studio will suggest adding the missing namespace to the file. We now have access to objects in the `MyMediaCollection.Model` namespace by using the `model` prefix, and `x:DataType="model:MediaItem"` will resolve the `MediaItem` when we build and run the app.

Each `TextBlock` has its `Text` property bound to a property of `MediaItem`, using the `x:Bind` **markup extension**.

**Note**

Using `x:Bind` instead of the `Binding` markup extension to bind data to the UI has the benefit of compile-time validation and increased performance. The previously noted data-binding overview on Microsoft Docs covers the differences in depth. I prefer to use `x:Bind` where possible. One important difference between `Binding` and `x:Bind` you should note is that while `Binding` defaults to `OneWay` mode, `x:Bind` defaults to `OneTime`. This change to the default binding behavior was made for performance considerations. `OneWay` binding requires more code behind the scenes to wire up the change detection needed for monitoring changes to the source value. You can still explicitly update your `x:Bind` usages to be `OneWay` or `TwoWay`. For more information about `x:Bind`, see this Microsoft article: <https://docs.microsoft.com/en-us/windows/uwp/xaml-platform/x-bind-markup-extension>.

For more information on **markup extensions** in XAML, you can read this **Windows Presentation Foundation (WPF)** article: <https://docs.microsoft.com/en-us/dotnet/framework/wpf/advanced/markup-extensions-and-wpf-xaml>.

Now, when you run the app, you can see the `MediumType` Name and the `Item Name` for each item in the `ListView`, as illustrated in the following screenshot:

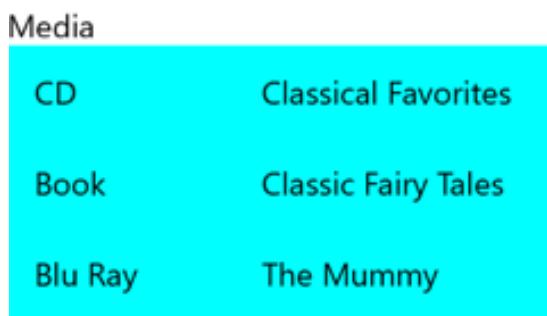


Figure 2.12 – The `ListView` with two columns of sample data

That's some pretty good progress! Before we expand on the functionality, let's talk a little about how WinUI and UWP fit together in terms of the app development process.

## Understanding WinUI and UWP

Let's review the WinUI controls available to use in our project and see how they can help us build the **My Media Collection** application. The `Microsoft.WinUI` package we saw in **Solution Explorer** earlier in the chapter contains these controls, and much more.

To view the contents of this package, open the **Object Browser** window from Visual Studio's **View** menu. The controls will be listed here under **Microsoft.UI** | **Microsoft.UI.Xaml.Controls**, as illustrated in the following screenshot:

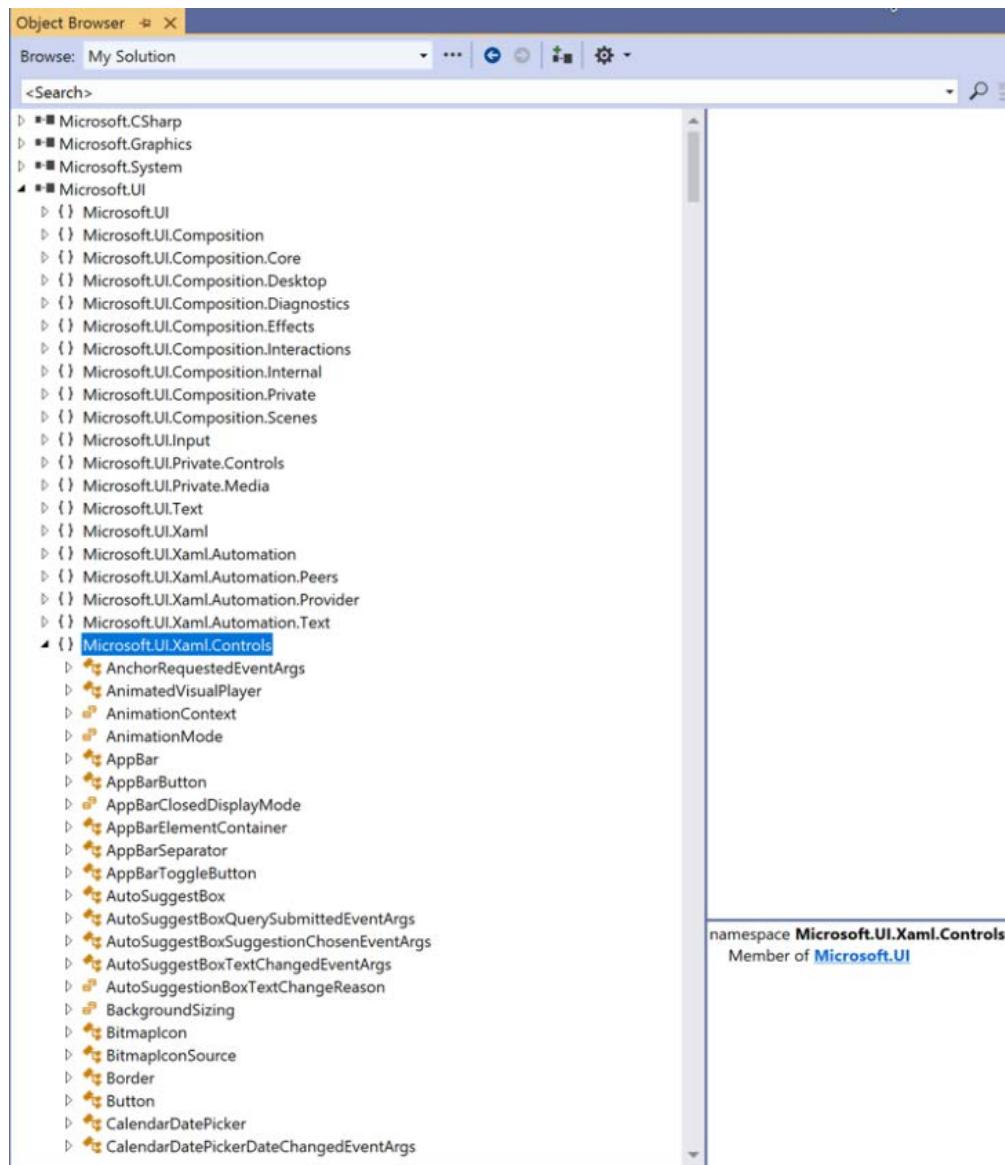


Figure 2.13 – WinUI in the Object Browser

All of the available WinUI controls can be found inside the **Microsoft.UI.Xaml.Controls** namespace, along with other related classes and interfaces.

So far, we have used in our application the `Grid`, `TextBlock`, and `ListView` controls. Open the **Object Browser** window from the **View** menu. Then, find the `ListView` class, expand `Microsoft.UI.Xaml.Controls.ListView` | **Base Types**, and select the `ListViewBase` class. This base class contains the methods, properties, and events available to the `ListView`. The members of `ListViewBase` will display in the right pane. Take some time to review these members and see if you recognize any of them from your use of the control so far.

Clear `ListView` from your search, scroll down in the left pane of the **Object Browser**, and find the `TextBlock` control. Select it and, in the right pane, find and select the `Text` property. The bottom-right pane displays details of the property, as illustrated in the following screenshot:



```
public string Text { get; set; }  
Member of Microsoft.UI.Xaml.Controls.TextBlock
```

**Summary:**

Gets or sets the text contents of a `TextBlock`.

**Returns:**

A string that specifies the text contents of this `TextBlock`. The default is an empty string.

Figure 2.14 – Details of the `TextBlock.Text` property in the Object Browser

The **Object Browser** window can be a valuable resource when familiarizing yourself with a new library or project. All referenced projects, NuGet packages, and other references will appear here.

The controls and other components you have reviewed here make up the UI layer of WinUI applications. The underlaying application platform for our app is UWP.

## Understanding the UWP app model

You may have been hearing that the underlying app model for WinUI apps can be Win32 for C++, or .NET for desktop apps, or UWP for .NET Native apps. So, what exactly is the **UWP app model**?

The UWP app model in WinUI apps describes how apps are packaged and deployed. It also defines the following behaviors and capabilities:

- Data storage
- State management
- Life cycle events (**Startup**, **Suspend**, **Resume**, and **Shutdown**)

- Background processing and multitasking
- Resource management
- Inter-app communication

Although the WinUI libraries are decoupled from the Windows SDK, the underlying UWP app platform is still dependent on it. Selecting a target and minimum Windows version is one of the side effects of the dependency.

The UWP controls from its UI layer still exist in the Windows SDK but will no longer be evolved or enhanced. There is also some uncertainty for the UWP app platform. The platform will probably continue to be developed, but only so far as to support the needs of WinUI and Project Reunion. It is also possible that parts of the UWP app platform that are not needed by Project Reunion could be deprecated in the future. It will be interesting to see how Project Reunion influences the UWP and Win32 platforms over the next few years.

**Note**

Microsoft has a good overview of UWP applications on this Docs page:  
<https://docs.microsoft.com/en-us/windows/uwp/get-started/universal-application-platform-guide>.

Now that you have a better understanding of WinUI controls and how they relate to the app platform, let's use a few more of them in our application.

## Working with WinUI controls, properties, and events

It's time to enhance the UI of the application. Currently, the main page only consists of a `Media` label over a `ListView`, with columns for the media type and the name of the media item. The following are the enhancements we will add in this section:

- A header row for the `ListView`
- A `ComboBox` filter to filter the rows based on the media type
- A `Button` to add a new item to the collection

We will start by enhancing the `ListView` for our media collection.

## Adding a ListView header

Before we create the header, let's change the background color of the `ListView`. The Aqua color worked well to highlight the control, but it would be distracting when the application is actually used. We will discuss WinUI theme brushes and understanding **Fluent Design** concepts later, in *Chapter 7, Windows Fluent UI Design*. For now, just remove `Background="Aqua"` from the `ListView` definition in the  `MainPage.xaml` file.

Creating the header row for the media collection is relatively simple. To define the rows for each item, we created a `ListView.ItemTemplate` block containing a `DataTemplate`. To create the header, we do the same inside a `ListView.HeaderTemplate` block.

Just as with the item rows, the header row will consist of a `Grid` with two columns, with the same `Width` definitions. We again want to use two `TextBlock` controls inside the `Grid`, but to add some separation between the header and the items, we will add `Border` controls. Let's take a look at the markup for the header and then discuss the differences in more detail. Have a look at the following code block:

```
<ListView.HeaderTemplate>
  <DataTemplate>
    <Grid>
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="100"/>
        <ColumnDefinition Width="*"/>
      </Grid.ColumnDefinitions>

      <Border BorderBrush="BlueViolet"
              BorderThickness="0,0,0,1">
        <TextBlock Text="Medium"
                  Margin="5,0,0,0"
                  FontWeight="Bold"/>
      </Border>
      <Border Grid.Column="1"
              BorderBrush="BlueViolet"
              BorderThickness="0,0,0,1">
        <TextBlock Text="Title"
                  Margin="5,0,0,0"
                  FontWeight="Bold"/>
      </Border>
    </Grid>
  </DataTemplate>
</ListView.HeaderTemplate>
```

```
</Border>
</Grid>
</DataTemplate>
</ListView.HeaderTemplate>
```

As you can see, each `TextBlock` is nested inside a `Border` element. This will wrap the text in a border with a `BlueViolet` color. However, by setting the `BorderThickness="0, 0, 0, 1"`, the border color will only appear on the bottom of the header row items. Here is how that appears in the application:

Medium	Title
CD	Classical Favorites
Book	Classic Fairy Tales
Blu Ray	The Mummy

Figure 2.15 – The ListView with a header row added

#### Note

The same bottom border could be achieved by nesting the entire `Grid` inside a `Border` instead of putting one around each header item. However, by doing it this way, we have more control over the appearance of each column's border style. When we implement sorting later, the border's color can be modified to highlight the column on which sorting has been applied.

You probably also noticed that the header row text stands out from the rows in the grid. The `FontWeight="Bold"` property set inside each `TextBlock` in the `HeaderTemplate` helps to highlight the header row.

## Creating the ComboBox filter

One of the requirements for the application is to allow users to filter on several of the collection items' properties. Let's start simple, by adding a filter only on the medium (**Book**, **Music**, or **Movie**). The list also needs an "All" option, which will be the default selection when users open the application:

1. First, add the XAML to `MainPage` to add a filter to the right of the **Media** label. Replace the "**Media**" `TextBlock` with the following markup:

```
<Grid>
<Grid.ColumnDefinitions>
```

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

<TextBlock Text="Media Collection"
Margin="4"
FontWeight="Bold"
VerticalAlignment="Center"/>
<StackPanel Grid.Column="1"
Orientation="Horizontal"
HorizontalAlignment="Right">
<TextBlock Text="Media Type:" Margin="4"
FontWeight="Bold"
VerticalAlignment="Center"/>
<ComboBox x:Name="ItemFilter"
MinWidth="120" Margin="0,2,6,4"/>
</StackPanel>
</Grid>
```

The single `TextBlock` label has been replaced with a two-column grid. The first column contains the `TextBlock`, with a few modifications. First, the `Text` property has been updated to "Media Collection". The `FontWeight` has been changed to "Bold" and some margin has been added. Finally, the element is centered vertically.

The second column contains a new `StackPanel`. A `StackPanel` is a container control that stacks its contents horizontally or vertically. The default orientation is `Vertical`. In our case, we want a horizontal stack, which is why the `Orientation` property has been set.

The `StackPanel` contains a `TextBlock` label and a `ComboBox` filter for the filter selection. The `ComboBox` filter has been given an `x:Name` so that we can reference it from the C# code-behind file when initializing its contents. We have also configured a `MinWidth` setting of 120. If the contents of the `ComboBox` filter require more than 120px (pixels), it has the ability to grow larger, but its width cannot be less than the value set here.

2. In the `MainPage.xaml.cs` file, add a new variable to hold the list of mediums, as follows:

```
private IList<string> _mediums { get; set; }
```

This collection can be an `IList` rather than an `ObservableCollection` because we don't expect its contents to change while the application is running.

3. Inside the `PopulateData()` method, add some code at the end of the method to populate the `_mediums` list, as follows:

```
_mediums = new List<string>
{
    "All",
    nameof(ItemType.Book),
    nameof(ItemType.Music),
    nameof(ItemType.Video)
};
```

We're adding an item to the collection for each of the possible values in our `ItemType` enum, plus the default "All" value.

4. The `ComboBox` filter will be bound to the collection after it has loaded, so add a `Loaded` event handler in the `MainPage` constructor, as we did for the `ItemList` earlier, like this:

```
ItemFilter.Loaded += ItemFilter_Loaded;
```

5. The `ItemFilter_Loaded` event handler will look similar to the `ItemList_Loaded` handler. Use the following code:

```
private void ItemFilter_Loaded(object sender,
    Microsoft.UI.Xaml.RoutedEventArgs e)
{
    var filterCombo = (ComboBox)sender;
    PopulateData();
    filterCombo.ItemsSource = _mediums;
    filterCombo.SelectedIndex = 0;
}
```

The code casts the sender to the `ComboBox` type and sets its `ItemSource` to the list we populated in the last step. Finally, an additional step is needed to default the `ComboBox` filter to default to the "All" item. This is accomplished by setting the `SelectedIndex` to 0.

Let's run the application and see how it looks now. You can see the result in the following screenshot:

Media Collection		Media Type:
Medium	Title	
CD	Classical Favorites	
Book	Classic Fairy Tales	
Blu Ray	The Mummy	

Figure 2.16 – Media Collection with the Media Type filter added

Pretty sharp! If you click the **Media Type** filter, you can see the four values available for selection, as illustrated here:

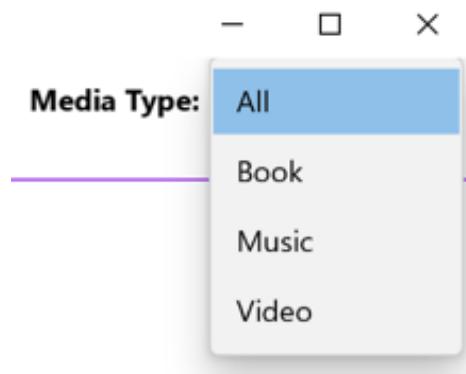


Figure 2.17 – Media Type values

Select one of the other values in the list and see what happens. Nothing! That's because we haven't added any code to do the filtering when the selection changes on the filter. We can add a little bit of extra code to fix that, as follows:

1. First, create a new `_allItems` collection to keep a list of all of the media items, regardless of the current filter, like this:

```
private IList<MediaItem> _allItems { get; set; }
```

2. Next, in the `PopulateData()` method, after populating the `_items` collection, add the same items to `_allItems`, like this:

```
_allItems = new List<MediaItem>
{
    cd,
    book,
    bluRay
};
```

3. Now, we need to do some filtering when the filter selection changes. We want to handle the `SelectionChanged` event on the `ComboBox` filter, but we don't want to hook it up until after the entire page has loaded. This will prevent the event from being handled while the `ComboBox` filter is initially populated.

Add an event handler for the page's `Loaded` event at the end of the `MainPage` constructor, like this:

```
Loaded += MainPage_Loaded;
```

4. In the implementation of `MainPage_Loaded`, add an event handler for the `SelectionChanged` event on the `ComboBox` filter, like this:

```
private void MainPage_Loaded(object sender,
    Microsoft.UI.Xaml.RoutedEventArgs e)
{
    ItemFilter.SelectionChanged +=
        ItemFilter_SelectionChanged;
}
```

5. In the new `ItemFilter_SelectionChanged` event handler, we will iterate through the `_allItems` list and determine which of the items to include in the filtered list, based on their `MediaType` property, as follows:

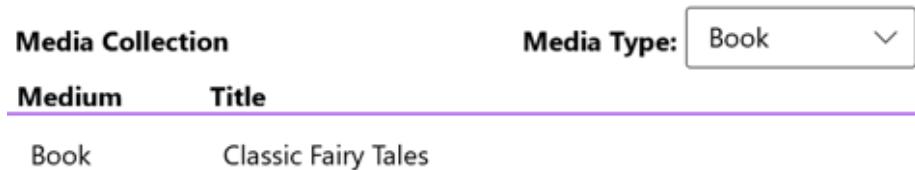
```
private void ItemFilter_SelectionChanged(
    object sender, Microsoft.UI.Xaml.Controls.
    SelectionChangedEventArgs e)
{
    var updatedItems =
        (from item in _allItems
        where
            string.IsNullOrWhiteSpace(ItemFilter.
                SelectedValue.ToString()) ||
                ItemFilter.SelectedValue.ToString() ==
                "All" ||
                ItemFilter.SelectedValue.ToString() ==
                item.MediaType.ToString())
        select item).ToList();
    ItemList.ItemsSource = updatedItems;
}
```

If the filter value is empty or "All" is selected, we want to include the item, regardless of its `MediaType`. Otherwise, we check if the `MediaType` matches the selection in the `ItemFilter` `ComboBox`. When there is a match, we add it to the `updatedItems` list. Then, we set the `updatedItems` as the `ItemsSource` on the `ListView`.

**Note**

A filter should never be empty unless there is an error while initializing the data. This condition is only a safeguard for unforeseen scenarios.

Now, run the app again and select **Book** in the filter, as illustrated in the following screenshot:



Media Collection		Media Type:
Medium	Title	
Book	Classic Fairy Tales	

Figure 2.18 – Media Collection filtered to display only books

That takes care of the filter implementation for the time being. Let's finish up this part of the UI design with a **Button**.

## Adding a new item button

We are not quite ready to start working with multiple pages or navigation yet. You should have some understanding of the **MVVM** pattern before we put too much logic into the app. This will minimize our code refactoring in later chapters. However, we can add a **Button** to the current page and add some code to ensure everything is hooked up correctly.

Open the `MainPage.xaml` file and add a third `RowDefinition` to the first-level `Grid` on the `Page`, as follows:

```
<Grid.RowDefinitions>
  <RowDefinition Height="Auto"/>
  <RowDefinition Height="*"/>
  <RowDefinition Height="Auto"/>
</Grid.RowDefinitions>
```

The new row will have a `Height` of `Auto` so that it sizes itself to fit the `Button`. We still want the `ListView` to take up the majority of the screen.

Now, after the closing tag of the `ListView` control, add the new `Button`, as follows:

```
<Button x:Name="AddButton" Content="Add Item"
  HorizontalAlignment="Right" Grid.Row="2" Margin="8"/>
```

As discussed in the previous chapter, the `Button` control does not have a `Text` property. Instead, if you only want a `Button` control to contain text, you assign it to the `Content` property. We are also assigning the `Button` control to the third row of the `Grid`, setting the `Margin`, and aligning it to the right side of the `Grid`.

Let's see how the app looks now, as follows:

Media Collection		Media Type:	All
Medium	Title		
CD	Classical Favorites		
Book	Classic Fairy Tales		
Blu Ray	The Mummy		

**Add Item**

Figure 2.19 – My Media Collection with an Add Item button

The button doesn't do anything yet. Because we're not yet ready to add an additional page to the application to add items, let's open a message dialog to inform the user that this function is not available.

In `MainPage.xaml`, wire up a new event handler inside the `AddButton`. You can also remove the `x:Name` attribute. We're able to remove the name because we do not need to reference it in the code-behind file. The code is shown in the following snippet:

```
<Button Content="Add Item"
        HorizontalAlignment="Right"
        Grid.Row="2" Margin="8" Click="AddButton_Click"/>
```

You can create the event handler by placing the cursor on the name of the handler, `AddButton_Click`, and pressing *F12*. This will create the handler and navigate to it in the `MainPage.xaml.cs` file. Inside the `AddButton_Click` event handler, we will create a new `MessageDialog` with the message we want to display to the user.

**Note**

The call to `dialog.ShowAsync()` must be awaited, so remember to add the `async` directive to the event handler, as shown next.

The `async` directive is illustrated in the following code snippet:

```
private async void AddButton_Click(object sender,
        Microsoft.UI.Xaml.RoutedEventArgs e)
{
```

```
var dialog = new MessageDialog("Adding items to the
    collection is not yet available.",
    "My Media Collection");
await dialog.ShowAsync();
}
```

Two using directives must also be added to `MainPage.xaml.cs` in order to compile the latest changes, as illustrated in the following code snippet:

```
using System;
using Windows.UI.Popups;
```

Now, run the application again and click the **Add Item** button, as illustrated in the following screenshot:

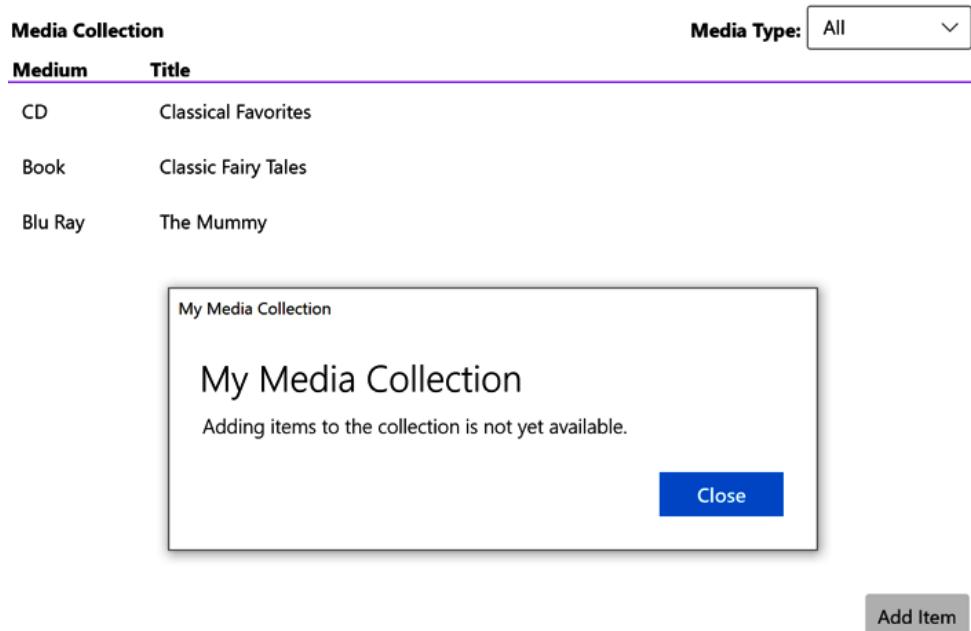


Figure 2.20 – Displaying a message dialog

That's all there is to adding a functional button to a WinUI page. As we have discussed, some of this code will change and be moved to a `ViewModel` in the next chapter, but you should now have a good idea of how to work with some basic `Button` properties and events.

## Summary

We have made some great progress on the **My Media Collection** application in this chapter. Along the way, you have learned to use several common WinUI controls. You have also learned how to change the appearance, layout, and behavior of WinUI controls by using different layout controls and updating control properties in XAML. Finally, you saw how to leverage data binding and events to add and update data displayed to the user.

Next, we will learn how to decouple some of the logic we have been writing in the code-behind files in order to build testable and maintainable applications.

## Questions

1. How do you add or remove features from Visual Studio?
2. What is the lowest minimum version of Windows that must be targeted when creating a new WinUI 3.0 project?
3. Where can you add XAML resources that can be shared by components in a whole application?
4. What is the default name of the first page loaded in a new WinUI app?
5. Which XAML container control allows you to define rows and columns to lay out its contents?
6. Which XAML container control stacks its contents horizontally or vertically?
7. What is the name of the message box that WinUI apps can use to display simple messages to users?
8. Challenge: What type of layout panel in WinUI allows its contents to be positioned absolutely?



# 3

# MVVM for Maintainability and Testability

When building **Extensible Application Markup Language (XAML)**-based applications, one of the most important design patterns to learn is the **Model-View-ViewModel (MVVM)** pattern. MVVM provides a clear separation of concerns between the XAML markup in the view and the C# code in the ViewModel, through the use of data binding. With this separation comes ease of maintenance and testability. The ViewModel classes can be unit tested without any dependency on the underlying **user interface (UI)** platform. For large teams, another benefit of this separation is that changing the XAML enables UI designers to work on the UI independently of developers who specialize in writing the business logic and the backend of the application.

In this chapter, you will learn about the following concepts:

- Fundamentals of the MVVM design pattern
- Popular MVVM frameworks
- Implementing MVVM in WinUI applications
- Handling ViewModel changes in the View
- Event handling in MVVM
- Popular unit test frameworks for .NET projects

By the end of the chapter, you will understand the basics of the MVVM design pattern, have some familiarity with some popular MVVM frameworks available to developers, and know how to implement MVVM in a WinUI application. We will wrap up by discussing the value of MVVM and unit testing.

## Technical requirements

To follow along with the examples in this chapter, please refer to the *Technical requirements* section of *Chapter 2, Configuring the Development Environment and Creating the Project*.

You will find the code files of this chapter here: <https://github.com/PacktPublishing/-Learn-WinUI-3.0/tree/master/Chapter03>

## Understanding MVVM

MVVM was introduced by Microsoft in 2005 and gained popularity with developers following the launch of **Windows Presentation Foundation (WPF)** and Silverlight because it lends itself so well to building applications with XAML. It is similar to the **Presentation Model** pattern, which was created by Martin Fowler, one of the most influential proponents of design patterns.

The MVVM pattern consists of the following three layers:

- **Model:** The Model layer contains the application's business logic and should perform all of the data access operations. The View Model communicates with the Model to retrieve and save the application's data.

- **View:** The View layer is only responsible for the presentation of data in the application. The layout or structure is defined here, along with style definitions. This is the layer responsible for interacting with the user and receiving input events and data. The View is aware of the View Model only through data-binding expressions.
- **ViewModel:** The View Model (or ViewModel) layer is responsible for maintaining the state of data for the View. It has a set of properties that provide the data to the View through data binding and a set of commands invoked by the View in response to user input events. View Model classes have no knowledge of their corresponding Views.

## MVVM – the big picture

Let's look at how the components of MVVM fit into the overall architecture of an application implementing the pattern, depicted in the following diagram:

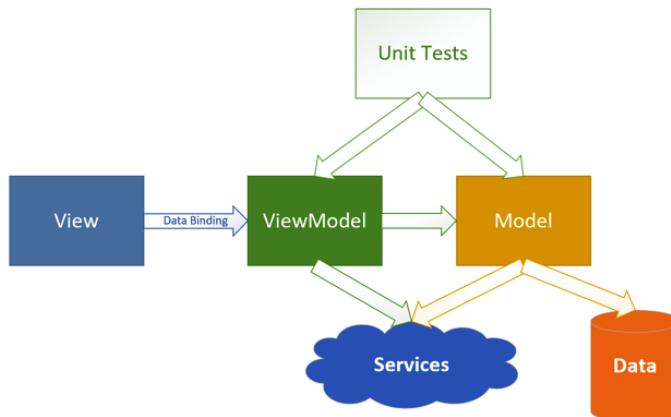


Figure 3.1 – The MVVM pattern in use

In *Figure 3.1*, you can see a representation of the MVVM pattern along with other parts of the application: services, data, and unit tests. The arrows in the diagram represent dependencies, not data flow. You can imagine that data would need to move both ways along most of these pathways to create a functional application.

As with many design patterns, the MVVM pattern is meant to be a guide for developers to create reliable, maintainable applications. However, not all developers implement the pattern in exactly the same way. The differences usually lie in the implementation of the Model. Some developers will create a domain model in the **Domain-Driven Design (DDD)** style. This makes sense for large applications with complex business logic. For simpler applications, the Model may only be a simple data access layer, residing either on the client or behind a service layer. In some of these cases, the **Services** cloud in the preceding diagram would move between the View Model and Model layers.

The point of MVVM is to help guide you in building the best app for your users. As you gain experience with MVVM and WinUI, you will find the right implementation for your applications. A good way to get started is to find some frameworks to make it easier to work with MVVM.

## MVVM libraries for WinUI

When working with MVVM in WinUI applications, you must create a little infrastructure code to facilitate data binding between Views and View Models. You can either write this yourself or choose a framework that abstracts this plumbing code away from your application. Although we will be writing the plumbing code for our application in the next section, let's review some popular MVVM frameworks for WinUI before that.

### Windows Community Toolkit MVVM library

We will discuss the **Windows Community Toolkit (WCT)** in more detail in *Chapter 9, Enhancing Applications with the Windows Community Toolkit*, but one of the libraries included in this open source toolkit is the MVVM library. You can get the `Microsoft.Toolkit.Mvvm` NuGet package through the **NuGet Package Manager** in Visual Studio or view its details on the NuGet website, at <https://www.nuget.org/packages/Microsoft.Toolkit.Mvvm/7.0.0>.

The library includes base classes to support `INotifyPropertyChanged`, `IMessenger`, and `ICommand`. It also includes other messaging and **Inversion of Control (IoC)** helper classes.

### Prism Library

**Prism** started out as a library created and maintained by Microsoft. I was part of the **Patterns & Practices** guidance, reference architectures, and libraries that Microsoft's developer division used to maintain.

#### Note

The remaining Patterns & Practices projects still maintained by Microsoft are available on GitHub here: <https://github.com/mspnp>.

Microsoft decided to open source the **Prism Library** and transferred ownership to the community. The project is hosted on GitHub and can be found on the web at <https://prismlibrary.com/>. Prism has packages available for WPF, Xamarin, and **Universal Windows Platform (UWP)**/WinUI projects.

Prism is much more than an MVVM framework. It also includes **application programming interfaces (APIs)** that help developers implement **dependency injection (DI)**, commands, and EventAggregator for loosely coupled application messaging. Prism can be added to a project through NuGet. There is also an installer available on the Prism site to add a Visual Studio project and item templates.

## MVVMCross

**MVVMCross** is a newer MVVM framework that was first created for Xamarin developers. It now has NuGet packages available for Xamarin, WPF, and UWP/WinUI. As with Prism, MVVMCross does much more than facilitate data binding in WinUI applications. It has helpers for the following:

- Data binding
- Navigation
- Logging/tracing
- DI and IoC
- Unit testing

There are some additional libraries but most are specific to Xamarin applications. The MVVMCross package can also be added to your project via NuGet. For more information about MVVMCross use with WinUI, check out their site: <https://www.mvvmcross.com/>.

## Choosing a framework for WinUI applications

Using a third-party framework for building production XAML applications is a great choice. These provide built-in support for things such as logging and DI. For the WinUI application in this book, we will not use any third-party helper libraries for data binding. This will help you understand the underlying mechanisms of data binding, DI, and other core concepts relating to the MVVM pattern.

Before jumping into the code, you should have some background on exactly how data binding works in WinUI.

## Understanding data binding in WinUI

In the previous chapter, you saw some simple examples of data binding, using both the `Binding` and newer `x:Bind` markup extensions. Let's dissect some of the pieces that allow the View to receive updates when the ViewModel data changes.

## What are markup extensions?

An in-depth discussion of markup extensions is beyond the scope of this introductory book. In brief, they are a class that executes some logic to return a value to the XAML parser. You can identify their use in XAML by looking for some markup inside curly braces. Take this example of `x:Bind` in the `Text` property of `TextBlock`:

```
<TextBlock Text="{x:Bind Path=Name, Mode=TwoWay}" />
```

From this, you can derive that there is a markup extension class named `Binding` and that two of its properties are `Path` and `Mode`. This markup extension takes these properties, resolves a value, and returns it to the XAML parser for display in the application's View.

Some XAML markup languages allow developers to write their own custom markup extensions. WPF and Xamarin have custom markup extensions, but UWP and WinUI do not.

Now, let's learn more about the `Binding` markup extension in WinUI.

## Binding markup extension

As you briefly saw, the `Binding` markup extension maps data from the binding source, the `ViewModel` in MVVM, and provides it to the View. These are all the properties of the `Binding` markup extension:

- `Path`: The path to the value in the data-binding source. For our application, this will be the property name on the `ViewModel`.
- `Converter`: If the data type of the source property does not match the data type of the control's property in the View, a `Converter` property is used to define the mapping between the two data types.
- `ConverterLanguage`: If a `Converter` property is specified, a `ConverterLanguage` property can also be set to support internationalization.
- `ConverterParameter`: If a `Converter` property takes parameters, use the `ConverterParameter` property to pass them. It is not common to use a `ConverterParameter` property, and they are usually `string` values.
- `ElementName`: This parameter is used when binding to the attribute of another element in the View.
- `FallBackValue`: If the data binding fails for any reason, you can specify a `FallBackValue` property to display in the View.

- **Mode:** Defines whether the data binding is `OneTime` (sets the value only when the XAML is first parsed), `OneWay` (fetches the value from the ViewModel when changes are detected), or `TwoWay` (value flows both ways between the View and ViewModel). The default `Mode` setting for `Binding` is `OneWay` in most cases.
- **RelativeSource:** This is used to define a data-binding source that is relative to the current control. This is usually used with control templates that get their data through a parent element.
- **Source:** Specifies the data-binding source. This is usually defined at the `Page` level in WinUI as the `ViewModel`. However, it is possible for controls within `Page` to set a different `Source`. The `Source` value defined at any level in the View will be inherited by all child elements.
- **TargetNullValue:** Specifies a default value to display if the data-binding source is resolved but has a null value.
- **UpdateSourceTrigger:** Specifies the frequency with which to update `TwoWay` binding sources. The options are `PropertyChanged`, `Explicit`, and `LostFocus`. The default frequency is `PropertyChanged`.

`Path` is the default property and is assumed when no property name is given for a parameter. The earlier `TextBlock` example could also be written as follows:

```
<TextBlock Text="{Binding Name, Mode=TwoWay}" />
```

Here, `Name` is assumed to be the `Path` property. Providing two parameters without specifying which parameter they are will result in an XAML parser error.

The `Binding` markup extension is found in every XAML language. The other option for data binding, `x:Bind`, is not. It is only an option with UWP and WinUI.

## x:Bind markup extension

`x:Bind` is an alternative markup extension for WinUI. It is faster and uses less memory than `Binding` and has better debugging support. It achieves this performance gain by generating code at compile time, to be used during binding at runtime. By contrast, the `Binding` markup extension is executed by the XAML parser at runtime, which incurs additional overhead. Compile-time bindings also result in incorrect data-binding expressions being caught when compiling, rather than generating data-binding failures at runtime.

An important distinction between `Binding` and `x:Bind` is that `Binding` requires `DataContext` to be set. Data is bound to properties of objects within `DataContext`. When using `x:Bind`, you are binding directly to properties on `Page` or `UserControl`. You can also bind events directly to event handlers on the page with `x:Bind`.

While most of the properties of `x:Bind` are the same as those in `Binding`, let's highlight those that differ, as follows:

- `ElementName`: Not available in `x:Bind`. You must use `Binding` to data-bind to other XAML element attributes. If your application must bind to other elements, `x:Bind` and `Binding` can be used in the same View.
- `Mode`: The only difference here is that the default `Mode` for `x:Bind` is `OneTime`, not `OneWay`.
- `RelativeSource`: Not available in `x:Bind`.
- `Source`: Not available in `x:Bind`. Instead, you will typically define a `ViewModel` property in the code-behind file of each View with a data type of the corresponding `ViewModel` class. You could also create a domain-specific name for the property, such as `MediaItems` for our application.
- `BindBack`: This property is unique to `x:Bind`. It allows a custom function to be called in `TwoWay` binding when reverse data binding is invoked. This is not commonly used, and we will not use it in our application.

`x:Bind` is a powerful and complex markup extension. For more information about it, you can read this page in Microsoft Docs: <https://docs.microsoft.com/en-us/windows/uwp/xaml-platform/x-bind-markup-extension>.

Next, let's discuss `INotifyPropertyChanged`, the interface that enables changes in data-bound properties in the `ViewModel` to be reflected in the view.

## Updating View data with `INotifyPropertyChanged`

So, how does the View get notified when data changes in the `ViewModel`? That magic lies within the `Microsoft.UI.Xaml.Data.INotifyPropertyChanged` interface. That interface consists of a single member, as shown here:

```
public interface INotifyPropertyChanged
{
    event PropertyChangedEventHandler PropertyChanged;
}
```

Every ViewModel class must implement this interface and raise the `PropertyChanged` event to update the View. Indicate which property is changed by passing its name in the `PropertyChangedEventArgs` parameter. To refresh all properties, pass `null` or `string.Empty` as the property name.

## Updating collection data with `INotifyCollectionChanged`

`INotifyPropertyChanged` works great for most properties, but it will not update the View if items from a collection have been added or removed. This is where the `Microsoft.UI.Xaml.Interop.INotifyCollectionChanged` interface is used. Again, this interface only has a single member, as shown here:

```
public interface INotifyCollectionChanged
{
    event NotifyCollectionChangedEventHandler
    CollectionChanged;
}
```

None of the collections or collection interfaces commonly used in .NET (`List<T>`, `IEnumerable<T>`, and so on) implement this interface. You can create your own collection derived from an existing list type and implement `INotifyCollectionChanged` yourself, but it's much easier to use the `ObservableCollection<T>` list type that is already available to WinUI developers. This is a read-only collection that will update the View when items are added or removed, or when the contents are completely refreshed.

`ObservableCollection` is read-only, thus its `Items` property cannot be directly set. You can add items by passing a `List<T>` or `IEnumerable<T>` to the constructor when creating `ObservableCollection` or by using its `Add` or `Insert` methods (there is no `AddRange` method). You can remove items with the `Remove`, `RemoveAt`, or `Clear` methods.

**Note**

The `INotifyCollectionChanged` interface currently has issues with **WinUI in UWP** projects. The code for this chapter has a custom `ObservableCollection` implementation to work around one of the issues. For more details and workarounds, see the current list of known issues in WinUI 3.0 on Microsoft Docs: <https://docs.microsoft.com/en-us/windows/apps/winui3/#known-issues>.

In the next section, when we implement the MVVM pattern for ourselves, you will see these concepts in practice.

## Implementing MVVM in WinUI applications

It's time to start converting our project to use MVVM. As we previously learned, we can best leverage the power and performance of WinUI bindings if we build our own MVVM framework. For most applications, it's not much more than a single base class:

1. Start by adding a `ViewModels` folder to the project. If you are using the code from [GitHub](#), you can either continue with your project from the previous chapter or use the Start project in the folder for this chapter.
2. Next, add a new class to the `ViewModels` folder and name it `BindableBase`. This will be the base class for all of our View Model classes in the project. It will be responsible for notifying the corresponding views of any property changes. This is accomplished by implementing the `INotifyPropertyChanged` interface. Let's review the `BindableBase` class code, as follows:

```
public class BindableBase : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;
    protected void OnPropertyChanged([CallerMemberName] string propertyName = null)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }
    protected bool SetProperty<T>(ref T originalValue, T newValue, [CallerMemberName] string propertyName = null)
    {
```

```

        if (Equals(originalValue, newValue))
        {
            return false;
        }
        originalValue = newValue;
        OnPropertyChanged(propertyName);
        return true;
    }
}

```

By using this as the base class of our View Models, they will have two new methods available to use, as follows:

- **OnPropertyChanged**: Use this to simply trigger a `PropertyChanged` event for the View to handle.
- **SetProperty**: This method will set the value of a property only if the value has been changed and will then call `OnPropertyChanged`.

**Note**

Be sure to add these two `using` directives at the top of the `BindableBase` class file:

```

using Microsoft.UI.Xaml.Data;
using System.Runtime.CompilerServices;

```

Now that we have a base class, let's add our first View Model to the project. Right-click the `ViewModels` folder and add a new class named `MainViewModel`. This View Model is going to replace most of the code in the `MainPage.xaml.cs` code-behind file for our `MainPage`. The following code is part of the revised class. Please refer to `MainViewModel.txt` in the GitHub repository for the chapter (<https://github.com/PacktPublishing/-Learn-WinUI-3.0/tree/master/Chapter03/Complete/MyMediaCollection/ViewModels>) for the current version of the complete class:

```

public class MainViewModel : BindableBase
{
    private string selectedMedium;
    private ObservableCollection<MediaItem> items;
    private ObservableCollection<MediaItem> allItems;
    private IList<string> mediums;
}

```

```
public MainViewModel()
{
    PopulateData();
}

...
public IList<string> Mediums
{
    get
    {
        return mediums;
    }
    set
    {
        SetProperty(ref mediums, value);
    }
}
...
}
```

You may have noticed that I updated the code to use the new `BindableBase`.  
`SetProperty()` method inside each property's `Set` block. This ensures that the UI will be notified when the data has been changed.

3. Now, we'll need to make the `MainViewModel` class available to the `MainPage` view. Because there will be a single instance of this `ViewModel` used throughout the lifetime of the app, we will add a static read-only property to the `App.xaml.cs` file to make it available to the application, as follows:

```
public static MainViewModel ViewModel { get; } = new
MainViewModel();
```

4. We can now strip out all of the code from `MainPage.xaml.cs` that we copied to the `MainViewModel` class. In addition, add a property to make `App.ViewModel` available to `MainPage` for data binding, as follows:

```
public sealed partial class MainPage : Page
{
    public MainPage()
    {
        this.InitializeComponent();
    }
}
```

```

        Loaded += MainPage_Loaded;
    }

    public MainViewModel ViewModel => App.ViewModel;
    private async void AddButton_Click(object sender,
    Microsoft.UI.Xaml.RoutedEventArgs e)
    {
        var dialog = new MessageDialog("Adding items
        to the collection is not yet available.", "My Media
        Collection");
        await dialog.ShowAsync();
    }
}

```

The only other code we need to keep in the code-behind file right now is for the **Add** button, which we will update in the next section.

Finally, it's time to update the `MainPage.xaml` file to bind to the data of the `MainViewModel`. There are only two changes required to handle the new data source, detailed here:

1. First, update `ComboBox` to remove the `x:Name` and add `x:Bind` data binding for the `ItemsSource` and `SelectedItem` properties. The `SelectedItem` binding needs to be set to `TwoWay`. This will make sure that `MainViewModel` is updated when the user changes `SelectedMedium` in the UI. The code can be seen in the following snippet:

```

<ComboBox ItemsSource="{x:Bind ViewModel.Mediums}"
    SelectedItem="{x:Bind ViewModel.SelectedMedium,
    Mode=TwoWay}" HorizontalAlignment="Right" MinWidth="120"
    Margin="0,2,6,4" />

```

2. Now, update `ListView` to remove `x:Name` and add an `ItemsSource` `x:Bind` data binding, as follows:

```

<ListView Grid.Row="1" ItemsSource="{x:Bind ViewModel.
    Items}">

```

The assigned names for these controls are no longer needed because we are not referencing them in the code-behind file.

**Note**

Assigning names to XAML elements allocates additional resources. It is recommended to only name elements when the elements must be referenced directly from code-behind files or by other View elements data binding via `ElementName`.

Now, run the application and try changing the `Medium` filter with `ComboBox`. It should behave exactly as it did before, but now we have decoupled the `ViewModel` data from the UI, making it easier to test. We'll get to some unit tests right after we take care of that button `Click` event we left in the `MainPage.xaml.cs` file.

## Working with events and commands

It's time to update our code to move the event handling code to `MainViewModel`. By the end of this section, you will have removed all the code that was added to the `MainPage.xaml.cs` file, except for the `ViewModel` property. This will be great for separation of concerns, as well as for the maintainability and testability of the project.

We could simply use the same method of hooking up events with the `Add` button's `Click` event and connect it to a method on the `MainViewModel` class. There are two problems with this approach, detailed here:

- The View and View Model layers become more tightly coupled, reducing maintainability.
- UI concerns are injected into the view model, reducing the testability of the class.

Let's take another route to tackle it. The MVVM pattern has the concept of **Commands** to handle events. Instead of adding a handler to the event of our view element, we will bind that event to a property on the view model. The `Command` properties all expect a type of `Microsoft.UI.Xaml.Input ICommand`.

## Implementing `ICommand`

To use commands in the project, you must start by creating an implementation of `ICommand`.

**Note**

Another advantage of using an MVVM framework such as **Prism** or **MVVMCross** is that it provides implementations of `ICommand`.

Add a new class to the `ViewModel` folder in the project and name it `RelayCommand`. This class will implement the `ICommand` interface. The `RelayCommand` class will look like this:

```
public class RelayCommand : ICommand
{
    private readonly Action action;
    private readonly Func<bool> canExecute;
    public RelayCommand(Action action)
        : this(action, null)
    {
    }
    public RelayCommand(Action action, Func<bool> canExecute)
    {
        if (action == null)
            throw new ArgumentNullException(nameof(action));
        this.action = action;
        this.canExecute = canExecute;
    }

    public bool CanExecute(object parameter) => canExecute == null || canExecute();
    public void Execute(object parameter) => action();
    public event EventHandler<object> CanExecuteChanged;
    public void RaiseCanExecuteChanged() => CanExecuteChanged?.Invoke(this, EventArgs.Empty);
}
```

`RelayCommand` has two constructors that both take an `Action` that will get invoked when the command is executed. One of them also takes a `Func<bool>`, which will allow us to enable or disable UI actions based on the return value of `CanExecute()`. We will use this to enable a **Delete** button only when a media item is selected in the list.

**Note**

The following `using` statement is also required in `RelayCommand`:

```
using Microsoft.UI.Xaml.Input.
```

## Using commands in the ViewModel

Now, it's time to update `MainViewModel` to handle **Add** and **Delete** operations. In the next chapter, we will enhance the **Add** operation to **Add** or **Edit** items, so let's name the command and method accordingly, as follows:

1. First, add two new private variables to the `MainViewModel` class, like this:

```
private MediaItem selectedMediaItem;  
private int additionalItemCount = 1;
```

The `additionalItemCount` variable is a temporary variable we will use to track how many new items we have added to the list. The counter will help to generate unique IDs and names for each new media item. `selectedMediaItem` is a backing variable for the new `SelectedMediaItem` property.

2. Add the `SelectedMediaItem` property next, as follows:

```
public MediaItem SelectedMediaItem  
{  
    get => selectedMediaItem;  
    set  
    {  
        SetProperty(ref selectedMediaItem, value);  
        ((RelayCommand)DeleteCommand).  
        RaiseCanExecuteChanged();  
    }  
}
```

In addition to calling `SetProperty` to notify the UI that `SelectedMediaItem` has changed, we also need to call `RaiseCanExecuteChanged` on a new `DeleteCommand`.

3. Let's add `DeleteCommand` as well as `AddEditCommand` and their corresponding actions next, as follows:

```
public ICommand AddEditCommand { get; set; }  
public void AddOrEditItem()  
{  
    // Note this is temporary until  
    // we use a real data source for items.  
    const int startingItemCount = 3;  
    var newItem = new MediaItem
```

```

    {
        Id = startingItemCount + additionalItemCount,
        Location = LocationType.InCollection,
        MediaType = ItemType.Music,
        MediumInfo = new Medium { Id = 1, MediaType = ItemType.Music, Name = "CD" },
        Name = $"CD {additionalItemCount}"
    };
    allItems.Add(newItem);
    Items.Add(newItem);
    additionalItemCount++;
}
public ICommand DeleteCommand { get; set; }
private void DeleteItem()
{
    allItems.Remove(SelectedMediaItem);
    Items.Remove(SelectedMediaItem);
}
private bool CanDeleteItem() => selectedMediaItem != null;

```

There is an `ICommand` property for each UI operation (`AddEditCommand` and `DeleteCommand`) and new methods to execute for each command (`AddOrEditItem` and `DeleteItem`). There is also a `CanDeleteItem` method that returns a `bool` value to indicate whether a media item has been selected by the user.

4. At the end of the `MainViewModel` constructor, add these two lines of code to link the commands to the actions:

```

DeleteCommand = new RelayCommand(DeleteItem,
    CanDeleteItem);
// No CanExecute param is needed for this command
// because you can always add or edit items.
AddEditCommand = new RelayCommand(AddOrEditItem);

```

5. To resolve the `ICommand` interface, add a `using` directive to the file, as follows:

```
using Microsoft.UI.Xaml.Input;
```

Our ViewModel has been updated to use commands. Next, we will update the view to bind to them.

## Updating the View

Our view model is ready to go. It's now safe to remove all the event handling code from the `MainPage` code-behind file. It should look like this when you're finished:

```
public sealed partial class MainPage : Page
{
    public MainPage()
    {
        this.InitializeComponent();
    }
    public MainViewModel ViewModel => App.ViewModel;
}
```

The `MainPage.xaml` file will need a few updates to have the **Add** and **Delete** features fully functional with the temporary test data, as follows:

1. Update `ListView` to bind the `SelectedItem` property to `SelectedMediaItem` in the view model, as follows:

```
<ListView Grid.Row="1" ItemsSource="{x:Bind
ViewModel.Items}" SelectedItem="{x:Bind ViewModel.
SelectedMediaItem, Mode=TwoWay}">
```

TwoWay data binding is required to allow the UI to update the view model.

2. Next, move the **Add Item** button inside the first column of a new two-column grid. Then, create a **Delete Item** button in the second column. Remove the `Click` event handler and set the properties of the grid and two buttons to match the following snippet:

```
<StackPanel Grid.Row="2"
HorizontalAlignment="Right"
Orientation="Horizontal">
<Button Command="{x:Bind ViewModel.AddEditCommand}"
Content="Add Item"
Margin="8,8,0,8"/>
<Button Command="{x:Bind ViewModel.DeleteCommand}">
```

```

Content="Delete Item"
Margin="8" />
</StackPanel>

```

Each button's `Command` property will be bound to the new `ICommand` properties in the view model. The `Command` property of a button will be invoked when it is clicked by the user.

3. We're now done updating the application to use MVVM. Run the application to see how it works.
4. When it first loads, the **Delete Item** button will be disabled. Select an item in the list and notice that the button automatically enables. If you click **Delete Item**, the selected item is removed from the list and the button is disabled again. Finally, click **Add Item** a few times to see how new items are created and added to the list. Each new item has a unique name using the counter we created in the view model, as illustrated in the following screenshot:

Media Collection		Media Type:	All
Medium	Title		
CD	Classical Favorites		
Blu Ray	The Mummy		
CD	CD 1		
CD	CD 2		
CD	CD 3		
CD	CD 4		

Add Item Delete Item

Figure 3.2 – My Media Collection after adding and removing a few items

MainPage now has a view model that is completely decoupled from any UI concerns. This will allow us to maximize unit test coverage on the project.

## Choosing a unit test framework

WinUI 3 does not currently support any of the .NET unit testing frameworks. However, following the MVVM pattern will make it simple to add unit tests to your application when support is added to WinUI.

There are three popular unit test frameworks for .NET developers today, as follows:

- **MSTest:** This framework was created by Microsoft and has been included with Visual Studio for many years. Documentation on using MSTest with C# can be found here: <https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-with-mstest>.
- **NUnit:** This open source framework was the first of the three mentioned here to be created for .NET developers. It was modeled after the JUnit Java unit testing framework. Learn more about the NUnit framework here: <https://nunit.org/>.
- **xUnit:** xUnit has been quickly growing in popularity since its introduction. The xUnit team's aim was to create a framework that is modern and extensible and that aligns with the features now available in .NET. Learn more about the team's original goals at <https://xunit.net/docs/why-did-we-build-xunit-1.0.html>.

## Summary

We've made quite a bit of progress with the application in this chapter. While it's not yet connected to a live data source, we have methods in place to add and remove items from the media collection in memory. In addition, the project has been refactored to use the **MVVM** pattern, moving all of the existing view logic from the `MainPage` code-behind file to a new `MainViewModel` class. The new `MainViewModel` class has no dependencies on the UI. This decoupling allowed us to effectively unit test our application logic. We added a unit test project to the solution with a suite of five unit tests, covering a majority of the `MainViewModel` logic. These good software design habits will serve us well in the chapters ahead as we build on more functionality to the project.

In the next chapter, we will continue learning how to use the MVVM pattern to write robust, maintainable WinUI applications. We will cover some more advanced MVVM topics and learn some additional techniques for testing our WinUI project.

## Questions

1. What does MVVM stand for?
2. Which layer typically defines the business entities in the MVVM pattern?
3. Name one of the popular MVVM frameworks discussed in the chapter.
4. Which interface must every View Model class implement in an MVVM application?
5. Which special collection type in .NET notifies the UI of changes to the collection, via data binding?
6. Which control property of ComboBox and ListView is used to get or set the currently selected item in the control?
7. Which interface is implemented to create commands for event binding?



# 4

# Advanced MVVM Concepts

After learning the basics of the MVVM pattern and its implementation in WinUI, it's now time to build on that knowledge base to handle some more advanced techniques. Now you will learn how to keep components loosely coupled and testable when adding new dependencies to the project.

Few modern applications have only a single page or window. There are MVVM techniques that can be leveraged to navigate between pages from a `ViewModel` command without being coupled with the UI layer.

In this chapter, you will learn the following concepts:

- Understanding the basics of **Dependency Injection (DI)**
- Leveraging DI to expose view model classes to WinUI views
- Using MVVM and `x:Bind` to handle more UI events with event handlers in the view model
- Navigating between pages with MVVM and DI

By the end of this chapter, you will have a deeper understanding of the MVVM pattern and will know how to decouple your view models from any external dependencies.

## Technical requirements

To follow along with the examples in this chapter, please reference the technical requirements in *Chapter 2, Configuring the Development Environment and Creating the Project*.

You will find the code files of this chapter here: <https://github.com/PacktPublishing/-Learn-WinUI-3.0/tree/master/Chapter04>.

## Understanding the basics of DI

Before starting down the path of using DI in our project, we should take some time to understand what DI is and why it is fundamental for building modern applications. You will often see DI referenced with another related concept, **Inversion of Control (IoC)**. Let's discuss these two concepts, clarify the relationship between them, and prepare you to use DI properly in this chapter.

DI is used by method developers to inject dependent objects into a class rather than creating instances of the objects inside of the class. There are different ways to inject those objects:

- **Method injection:** Objects are passed as parameters to a method in the class.
- **Property injection:** Objects are set through properties.
- **Constructor injection:** Objects are passed as constructor parameters.

The most common method of DI is constructor injection. In this chapter, we will be using both property injection and constructor injection. Method injection will not be used because it is not common to use methods to set a single object's value in .NET projects. Most developers use properties for this purpose.

IoC is the concept that a class should not be responsible for (or have knowledge of) the creation of its dependencies. You're inverting control over object creation. This sounds a bit like DI, doesn't it? Well, DI is one method of achieving this IoC in your code. There are other ways to implement IoC, including the following:

- **Delegate:** This holds a reference to a method that can be used to create and return an object.
- **Event:** Similar to delegates, they are typically used in association with user input or other outside actions.
- **Service Locator Pattern:** This is used to inject the implementation of a service at runtime.

When you separate the responsibilities of object creation and use, it facilitates code reuse and increases testability.

The classes that will be taking advantage of DI in this chapter are views and view models. So, if we will not be creating instances of objects in those classes, where will they be created? Aren't we just moving the tight coupling somewhere else? In a way, that is true, but the coupling will be minimized by centralizing it to one part of the project, the `App.xaml.cs` file. If you remember from the previous chapter, the `App` class is where we handle application-wide actions and data.

We are going to use a **DI container** in the `App` class to manage the application's dependencies. A DI container is responsible for creating and maintaining the lifetime of the objects it manages. The object's lifetime in the container is usually either *per instance* (each object request returns a new instance of the object) or a *singleton* (every object request returns the same instance of the object). The container is set up in the `App` class, and it makes instances available to other classes in the application.

It is time to see DI and DI containers in practice.

## Using DI with ViewModel classes

All of the popular MVVM frameworks include a DI container to manage dependencies. Because we are handing MVVM ourselves, we will use a DI container that isn't bundled with any MVVM framework. Microsoft has included its own DI container in **ASP.NET Core** that is lightweight and easy to use. Luckily, this container is also available to other types of .NET projects, via a **NuGet** package.

In the MyMediaCollection project, open **NuGet Package Manager** and search for `Microsoft.Extensions.DependencyInjection`:

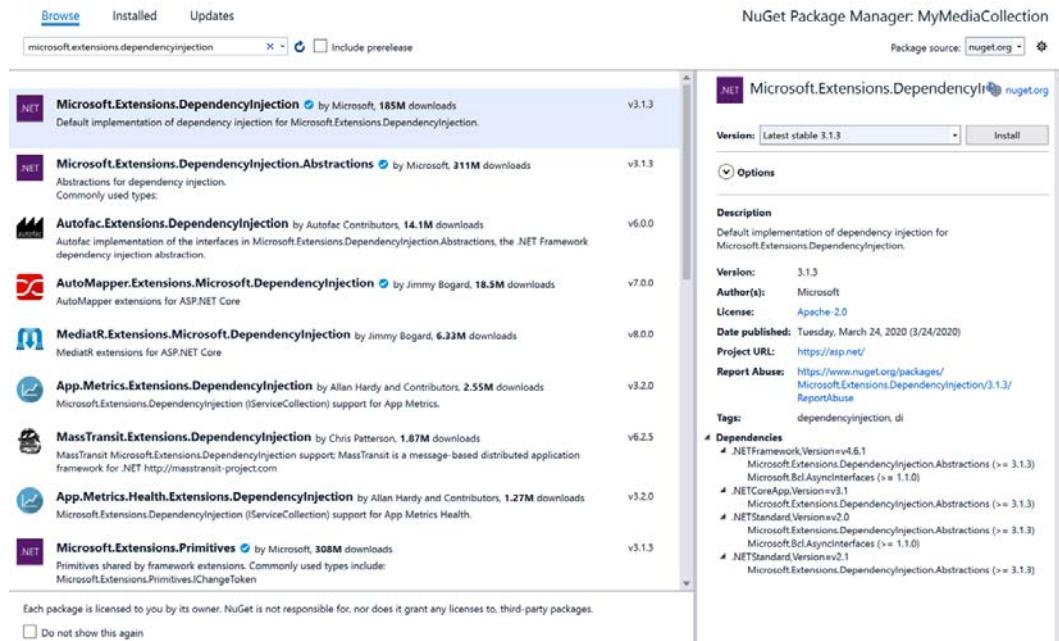


Figure 4.1 – Microsoft's DI NuGet package

Select the package and install the latest available version. After the installation completes, close the **NuGet Package Manager** tab and open `App.xaml.cs`. We have to make a few changes here to start using the DI container.

Microsoft implements a DI container through a class called `ServiceCollection`. As the name implies, it is intended to create a collection of services for the application. However, we can add any type of class to the container. Its use is not restricted to services. `ServiceCollection` builds the container, implementing the `IServiceProvider` interface. In the following steps, you will add support for DI to the application:

1. The first thing you should do is add a public property to the `App` class that makes the container available to the project:

```
/// <summary>
/// This DI container manages the project's
/// dependencies.
/// </summary>
public IServiceProvider Container { get; private set; }
```

Here, `get` is public, but the property has a `private` `set` accessor. This restricts the creation of the container to the `App` class. Don't forget to add a `using` statement to the class:

```
using Microsoft.Extensions.DependencyInjection;
```

2. The next step is to create a new method that initializes the container and to add our first dependency:

```
/// <summary>
/// Initializes the DI container.
/// </summary>
/// <returns>An instance implementing
/// IServiceProvider.</returns>
private IServiceProvider RegisterServices()
{
    var services = new ServiceCollection();
    services.AddTransient<MainViewModel>();
    return services.BuildServiceProvider();
}
```

In the `RegisterServices` method, we are creating `ServiceCollection`, registering `MainViewModel` as a **transient (per instance)** object and using the `BuildServiceProvider` method of the `ServiceCollection` class to create and return the DI container.

3. Finally, you will call `RegisterServices` from the `App.OnLaunched` event handler:

```
protected override void
OnLaunched(LaunchActivatedEventArgs e)
{
    Container = RegisterServices();

    Frame rootFrame = Window.Current.Content as Frame;
    ...
}
```

That's all of the code needed to create and expose the DI container to the application. Now that we are delegating the creation of `MainViewModel` to the container, you can remove the property of the `App` class, exposing the instance of `MainViewModel`.

Using the view model controlled by the container is simple. Go ahead and open `MainPage.xaml.cs` and update the `ViewModel` property to use the container's `GetService<T>` method:

```
public MainViewModel ViewModel { get; } =  
    (Application.Current as  
     App).Container.GetService<MainViewModel>();
```

Don't forget to add a `using` statement for the DI package:

```
using Microsoft.Extensions.DependencyInjection;
```

If you build and run the application now, it will work just as it did before. However, now all of our object instances will be registered in the `App` class and managed by the container. As new view models, services, and other dependencies are added to the project, they will be added to the `RegisterServices` method.

We will be adding a second page to the app later in this chapter. First, let's discuss the *event-to-command* pattern.

## Leveraging `x:Bind` with events

In the previous chapter, we bound view model commands to the `Command` properties of the **Add** and **Delete** buttons. This works great and keeps the view model decoupled from the UI, but what happens if you need to handle an event that isn't exposed through a `Command` property? For this scenario, you have two options:

- Use `Binding` in the view to bind to a command on the view model.
- Use `x:Bind` in the view to bind directly to an event handler on the view model.

If you are planning to share view models between **WinUI** and **WPF**, binding to commands is recommended because both app models support the `Binding` syntax. If your view models are only going to be used in a WinUI application, you should use `x:Bind`. This option will provide compile-time type checking and added performance.

We want to give users of the My Media Collection application the option to double-click (or double-tap) a row on the list to view or edit its details. The new **Item Details** page will be added in the next section. Until then, double-clicking an item will invoke the same code as the **Add** button, as this will become the **Add/Edit** button later:

1. Start by adding an `ItemRowDoubleTapped` event handler to the `MainViewModel` class that calls the existing `AddOrEditItem` method:

```
public void ListViewDoubleTapped(object sender,
    DoubleTappedRoutedEventArgs e)
{
    AddOrEditItem();
}
```

2. Next, bind the `ListView.DoubleTapped` event to the view model:

```
<ListView Grid.Row="1" ItemsSource="{x:Bind
    ViewModel.Items}"
    SelectedItem="{x:Bind
        ViewModel.SelectedMediaItem,
        Mode=TwoWay}"
    DoubleTapped="{x:Bind
        ViewModel.ListViewDoubleTapped}">
```

3. Finally, to ensure that the double-clicked row is also selected, modify `Grid` inside `ListView.ItemTemplate` to set the `IsHitTestVisible` property:

```
<ListView.ItemTemplate>
    <DataTemplate x:DataType="model:MediaItem">
        <Grid IsHitTestVisible="False">
            ...
        </Grid>
    </DataTemplate>
</ListView.ItemTemplate>
```

Now when you run the application, you can either click the **Add** button or double-click a row in the list to add new items. In the next section, you will update the **Add** button to be an **Add/Edit** button.

## Page navigation with MVVM and DI

Until this point, the application has consisted of only a single page. It is now time to add a second page to handle adding new items or editing existing items. The page will be accessible from the **Add/Edit** button or by double-clicking on an item in the list.

### Adding ItemDetailsPage

The full `ItemDetailsPage.xaml` code can be found on GitHub (<https://github.com/PacktPublishing/-Learn-WinUI-3.0/blob/master/Chapter04/Complete/MyMediaCollection/Views/ItemDetailsPage.xaml>). You can follow along with the steps in this section or review the final code on GitHub.

#### Note

The project will not compile successfully until we have added the new view model to the project and added it to the DI container for consumption by the view. Before we add the view model, we need to create a couple of services to enable page navigation and data persistence between pages.

To add `ItemDetailsPage`, let's follow these steps:

1. Add a new folder to the project named `Views`.
2. Right-click the new folder and select **Add | New Item**.
3. On the new item dialog, select **Blank Page (WinUI)** and name the page `ItemDetailsPage`:

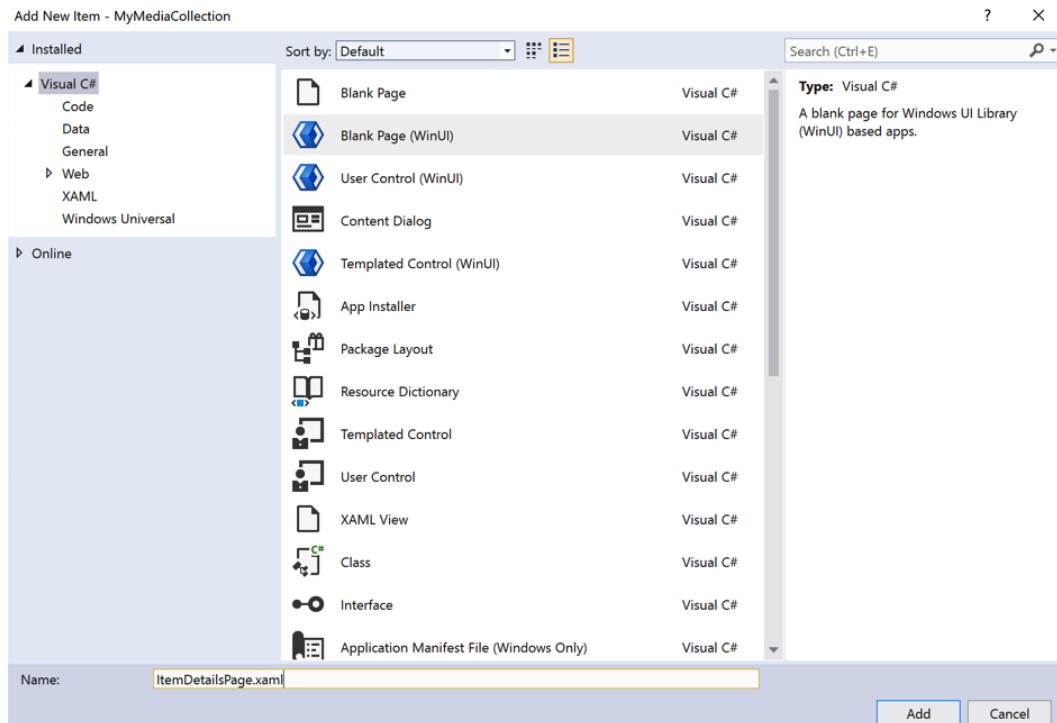


Figure 4.2 – Create an Item Details page

4. There are going to be several input controls with some common attributes on the page. Start by adding three styles to a `Page.Resources` section just before the top-level `Grid` control:

```

<Page.Resources>
    <Style x:Key="AttributeTitleStyle"
        TargetType="TextBlock">
        <Setter Property="HorizontalAlignment"
            Value="Right"/>
        <Setter Property="VerticalAlignment"
            Value="Center"/>
    </Style>
    <Style x:Key="AttributeValueStyle"
        TargetType="TextBox">
        <Setter Property="HorizontalAlignment"
            Value="Stretch"/>
        <Setter Property="Margin" Value="8"/>
    </Style>

```

```
</Style>
<Style x:Key="AttributeComboValueStyle"
       TargetType="ComboBox">
    <Setter Property="HorizontalAlignment"
            Value="Stretch"/>
    <Setter Property="Margin" Value="8" />
</Style>
</Page.Resources>
```

In the next step, we can assign `AttributeTitleStyle` to each `TextBlock`, `AttributeValueStyle` to each `TextBox`, and `AttributeComboValueStyle` to each `ComboBox`. If you need to add any other attributes to input labels later, you only have to update `AttributeTitleStyle` to have the attributes apply to every applicable `TextBlock`.

5. The top-level `Grid` will contain three child `Grid` controls to partition the view into three areas—a header, the input controls, and the **Save** and **Cancel** buttons at the bottom. The input area will be given the bulk of the available space, so define `Grid.RowDefinitions` like this:

```
<Grid.RowDefinitions>
    <RowDefinition Height="Auto"/>
    <RowDefinition Height="*"/>
    <RowDefinition Height="Auto"/>
</Grid.RowDefinitions>
```

The header area will contain only a `TextBlock`. You are welcome to design this area however you like:

```
<TextBlock Text="Item Details" FontSize="18"
           Margin="8" />
```

The input area contains a `Grid` with four `RowDefinitions` and two `ColumnDefinitions`, for the labels and input controls for the four fields that users can currently edit:

```
<Grid Grid.Row="1">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
```

```
</Grid.RowDefinitions>
<Grid.ColumnDefinitions>
    <ColumnDefinition Width="200"/>
    <ColumnDefinition Width="*"/>
</Grid.ColumnDefinitions>
<TextBlock Text="Name:" Style="{StaticResource
    AttributeTitleStyle}"/>
<TextBox Grid.Column="1" Style="{StaticResource
    AttributeValueStyle}" Text="{x:Bind
        ViewModel.ItemName, Mode=TwoWay,
        UpdateSourceTrigger=PropertyChanged}"/>
<TextBlock Text="Media Type:" Grid.Row="1"
    Style="{StaticResource AttributeTitleStyle}"/>
<ComboBox Grid.Row="1" Grid.Column="1"
    Style="{StaticResource
        AttributeComboxValueStyle}"
    ItemsSource="{x:Bind ViewModel.ItemTypes}"
    SelectedValue="{x:Bind ViewModel.
        SelectedItemType, Mode=TwoWay}"/>
<TextBlock Text="Medium:" Grid.Row="2"
    Style="{StaticResource AttributeTitleStyle}"/>
<ComboBox Grid.Row="2" Grid.Column="1"
    Style="{StaticResource
        AttributeComboxValueStyle}"
    ItemsSource="{x:Bind ViewModel.Mediums}"
    SelectedValue="{x:Bind ViewModel.
        SelectedMedium, Mode=TwoWay}"/>
<TextBlock Text="Location:" Grid.Row="3"
    Style="{StaticResource AttributeTitleStyle}"/>
<ComboBox Grid.Row="3" Grid.Column="1"
    Style="{StaticResource
        AttributeComboxValueStyle}"
    ItemsSource="{x:Bind
        ViewModel.LocationTypes}"
    SelectedValue="{x:Bind
        ViewModel.SelectedLocation,"
```

```
        Mode=TwoWay } "/>
    </Grid>
```

6. The item's Name is a free-text entry field, while the others are ComboBox controls to allow the user to pick values from lists bound to ItemsSource. The final child element of the top-level Grid is a right-aligned horizontal StackPanel containing the **Save** and **Cancel** buttons:

```
<StackPanel Orientation="Horizontal"
    Grid.Row="2" HorizontalAlignment="Right">
    <Button Content="Save"
        Margin="8,8,0,8"
        Command="{x:Bind ViewModel.SaveCommand}" />
    <Button Content="Cancel"
        Margin="8"
        Command="{x:Bind ViewModel.CancelCommand}" />
</StackPanel>
```

The first step after this is to add interfaces and services, so let's check this next.

## Adding new interfaces and services

Now that we have more than a single page to manage in the application, we need a couple of services to centralize that management and abstract the details from the view model code. Start by creating **Services** and **Interfaces** folders in the project. Each service will implement an interface. This interface will be used for DI and later when we add new unit tests to the test project.

## Creating a navigation service

The first service we need is a **navigation service**. Start by defining the **INavigationService** interface in the **Interfaces** folder. The interface defines methods to get the current page name, navigate to a specific page, or navigate back to the previous page:

```
public interface INavigationService
{
    string CurrentPage { get; }
    void NavigateTo(string page);
    void NavigateTo(string page, object parameter);
```

```
    void GoBack();
}
```

Now, create a `NavigationService` class in the `Services` folder. In the class definition, make sure that `NavigationService` implements the `INavigationService` interface. The full class can be viewed on [GitHub](https://github.com/PacktPublishing/-Learn-WinUI-3.0/blob/master/Chapter04/Complete/MyMediaCollection/Services/NavigationService.cs) (<https://github.com/PacktPublishing/-Learn-WinUI-3.0/blob/master/Chapter04/Complete/MyMediaCollection/Services/NavigationService.cs>). Let's discuss a few highlights.

The purpose of a navigation service in MVVM is to store a collection of available pages in the application so when its `NavigateTo` method is called, the service can find a page that matches the requested Name or Type and navigate to it.

The collection of pages will be stored in a `ConcurrentDictionary<T>` collection. The `ConcurrentDictionary<T>` functions like the standard `Dictionary<T>`, but it can automatically add locks to prevent changes to the dictionary simultaneously across multiple threads:

```
private readonly IDictionary<string, Type> _pages = new
    ConcurrentDictionary<string, Type>();
```

The `Configure` method will be called when you create `NavigationService` before adding it to the DI container. This method is not a part of the `INavigationService` interface and will not be available to classes that consume the service from the container. There is a check here to ensure views are only added to the service once. We check the dictionary to determine whether any pages of the same data type exist. If this condition is true, then the page has already been registered:

```
public void Configure(string page, Type type)
{
    if (_pages.Values.Any(v => v == type))
    {
        throw new ArgumentException($"The
            {type.Name} view has already been
            registered under another name.");
    }
    _pages[page] = type;
}
```

These are the implementations of the three navigation methods in the service. The two `NavigateTo` methods navigate to a specific page, with the second providing the ability to pass a parameter to the page. The third is `GoBack`, which does what you would think. It navigates to the previous page in the application. They wrap the `Frame` navigation calls to abstract the UI implementation from the view models that will be consuming this service:

```
public void NavigateTo(string page)
{
    NavigateTo(page, null);
}

public void NavigateTo(string page, object parameter)
{
    if (!_pages.ContainsKey(page))
    {
        throw new ArgumentException($"Unable to
            find a page registered with the name {page}.");
    }
    AppFrame.Navigate(_pages[page], parameter);
}

public void GoBack()
{
    if (AppFrame?.CanGoBack == true)
    {
        AppFrame.GoBack();
    }
}
```

We're ready to start using `NavigationService`, but first, let's create a data service for the application.

**Note**

You can jump ahead to implementing the services in the next section if you like. The `DataService` and `IDataService` code is available in the completed solution on GitHub: <https://github.com/PacktPublishing/-Learn-WinUI-3.0/tree/master/Chapter04/Complete/MyMediaCollection>.

## Creating a data service

The data on MainPage of My Media Collection currently consists of a few sample records created and stored in MainViewModel. This isn't going to work very well across multiple pages. By using a data service, the view models will not need to know how the data is created or stored.

For now, the data will still be sample records that are not saved in between sessions. Later, we can update the data service to save and load data from a database without any changes to the view models that use the data.

The first step is to add an interface named `IDataService` to the `Interfaces` folder:

```
public interface IDataService
{
    IList<MediaItem> GetItems();
    MediaItem GetItem(int id);
    int AddItem(MediaItem item);
    void UpdateItem(MediaItem item);
    IList<ItemType> GetItemTypes();
    Medium GetMedium(string name);
    IList<Medium> GetMediums();
    IList<Medium> GetMediums(ItemType itemType);
    IList<LocationType> GetLocationTypes();
    int SelectedItemId { get; set; }
}
```

These methods should look familiar to your from previous chapters, but let's briefly review the purpose of each:

- `GetItems`: Returns all of the available media items
- `GetItem`: Finds a media item with the provided `id`
- `AddItem`: Adds a new media item to the collection
- `UpdateItem`: Updates a media item in the collection
- `GetItemTypes`: Gets the list of media item types
- `GetMedium`: Gets a `Medium` with the provided name
- `GetMediums`: These two methods either get all available mediums or any available for the provided `ItemType`

- `GetLocationTypes`: Gets all of the available media locations
- `SelectedItemId`: Persists the ID of the selected item on `MainPage`

Now, create the `DataService` class in the `Services` folder. Make sure that `DataService` implements `IDataService` in the class definition.

Again, we will only review parts of the code. You can review the entire implementation on GitHub (<https://github.com/PacktPublishing/-Learn-WinUI-3.0/blob/master/Chapter04/Complete/MyMediaCollection/Services/DataService.cs>). The data in `DataService` will be persisted in four lists and the `SelectedItemId` property:

```
private IList<MediaItem> _items;
private IList<ItemType> _itemTypes;
private IList<Medium> _mediums;
private IList<LocationType> _locationTypes;
public int SelectedItemId { get; set; }
```

Copy the `PopulateItems` method from `MainViewModel` and modify it to use `List<T>` collections and to add the `Location` property assignment to each item.

Start by creating the three `MediaItem` objects:

```
var cd = new MediaItem
{
    Id = 1,
    Name = "Classical Favorites",
    MediaType = ItemType.Music,
    MediumInfo = _mediums.FirstOrDefault(m => m.Name ==
        "CD"),
    Location = LocationType.InCollection
};
var book = new MediaItem
{
    Id = 2,
    Name = "Classic Fairy Tales",
    MediaType = ItemType.Book,
    MediumInfo = _mediums.FirstOrDefault(m => m.Name ==
        "Hardcover"),
    Location = LocationType.InCollection
}
```

```

} ;

var bluRay = new MediaItem
{
    Id = 3,
    Name = "The Mummy",
    MediaType = ItemType.Video,
    MediumInfo = _mediums.FirstOrDefault(m => m.Name ==
        "Blu Ray"),
    Location = LocationType.InCollection
};

```

Then initialize the `_items` list and add the three `MediaItem` objects you just created:

```

_items = new List<MediaItem>
{
    cd,
    book,
    bluRay
};

```

There are three other methods to pre-populate the sample data: `PopulateMediums`, `PopulateItemTypes`, and `PopulateLocationTypes`. All of these are called from the `DataService` constructor. These methods will be updated later to use a *SQLite* data store for data persistence.

Most of the `Get` method implementations are very straightforward. The `GetMediums (ItemType itemType)` method uses **Language Integrated Query (LINQ)** to find all `Medium` objects for the selected `ItemType`:

```

public IList<Medium> GetMediums (ItemType itemType)
{
    return _mediums
        .Where(m => m.MediaType == itemType)
        .ToList () ;
}

```

#### Note

If you are not familiar with LINQ expressions, Microsoft has some good documentation on the topic: <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/>.

The `AddItem` and `UpdateItems` methods are also fairly simple, adding to or updating the `_items` collection:

```
public int AddItem(MediaItem item)
{
    item.Id = _items.Max(i => i.Id) + 1;
    _items.Add(item);
    return item.Id;
}

public void UpdateItem(MediaItem item)
{
    var idx = -1;
    var matchedItem =
        (from x in _items
         let ind = idx++
         where x.Id == item.Id
         select ind).FirstOrDefault();

    if (idx == -1)
    {
        throw new Exception("Unable to update item. Item
            not found in collection.");
    }

    _items[idx] = item;
}
```

The `AddItem` method has some basic logic to find the highest `Id` and increment it by 1 to use as the new item's `Id`. `Id` is also returned to the calling method in case the caller needs the information.

The services are all created. It is time to set them up when the application launches and consume them in the view models.

## Increasing maintainability by consuming services

Before using the services in the view models, open the `RegisterServices` method in `App.xaml.cs` and add the following code to register the new services in the DI container, register a new `ItemDetailsViewModel` (yet to be created), and register the two views with `NavigationService`:

```
private IServiceProvider RegisterServices()
{
    var services = new ServiceCollection();
    var navigationService = new NavigationService();
    navigationService.Configure(nameof(MainPage),
        typeof(MainPage));
    navigationService.Configure(nameof(ItemDetailsPage),
        typeof(ItemDetailsPage));
    services.AddSingleton<INavigationService>(navigationService
    );
    services.AddSingleton<IDataService, DataService>();
    services.AddTransient<MainViewModel>();
    services.AddTransient<ItemDetailsViewModel>();

    return services.BuildServiceProvider();
}
```

Both `INavigationService` and `IDataService` are registered as **singletons**. This means that there will be a single instance of each stored in the container. Any state held in these services is shared across all classes that consume them.

You will notice that when we're registering `INavigationService`, we are passing the instance we already created to the constructor. This is a feature of Microsoft's DI container and most other DI containers, to allow for initialization and configuration of instances before they are added.

We need to make a few changes to `BindableBase` and `MainViewModel` to consume `IDataService` and `INavigationService`, update the `PopulateData` method, and navigate to `ItemDetailsPage` when `AddEditCommand` is invoked:

1. Start by adding protected properties to `BindableBase` for `INavigationService` and `IDataService` so they are available to every `ViewModel`:

```
protected INavigationService _navigationService;
protected IDataService _dataService;
```

Don't forget to add a `using` statement for `MyMediaCollection.Interfaces`.

2. Next, update `MainViewModel` to receive and store the services:

```
public MainViewModel(INavigationService
    navigationService, IDataService dataService)
{
    _navigationService = navigationService;
    _dataService = dataService;
    PopulateData();
    DeleteCommand = new RelayCommand(DeleteItem,
        CanDeleteItem);
    AddEditCommand = new RelayCommand(AddOrEditItem);
}
```

Wait, we've added two parameters to the constructor but haven't changed the code that adds them to the DI container. How does that work? Well, the container is smart enough to pass them because both of those interfaces are also registered. Pretty cool!

3. Next, update `PopulateData` to get the data the view model needs from `_dataService`:

```
public void PopulateData()
{
    items.Clear();
    foreach(var item in _dataService.GetItems())
    {
        items.Add(item);
    }
}
```

```

allItems = new
    ObservableCollection<MediaItem>(Items);
mediums = new ObservableCollection<string>
{
    AllMediums
};
foreach(var itemType in
    _dataService.GetItemTypes())
{
    mediums.Add(itemType.ToString());
}
selectedMedium = Mediums[0];
}

```

You have to add the `AllMediums` string constant with a value of "All" to the `mediums` collection because it is not part of the persisted data. It is only needed for the UI filter. Be sure to add this constant definition to the `ViewModel`.

4. Finally, when `AddEditCommand` calls the `AddOrEditItem` method, instead of adding hardcoded items to the collection, you will pass `selectedItemId` as a parameter when navigating to `ItemDetailsPage`:

```

private void AddOrEditItem()
{
    var selectedItemId = -1;
    if (SelectedMediaItem != null)
    {
        selectedItemId = SelectedMediaItem.Id;
    }
    _navigationService.NavigateTo("ItemDetailsPage",
        selectedItemId);
}

```

That's it for `MainViewModel`. The only change you need to make in the `MainPage.xaml` file is to change `Content` of the **Add** button to `Add/Edit Item`.

## Handling parameters in ItemDetailsPage

To accept a parameter passed from another page during navigation, you must override the `OnNavigatedTo` method in `ItemDetailsPage.xaml.cs`. The `NavigationEventArgs` parameter contains a property named `Parameter`. In our case, we passed an `int` containing the selected item's `Id`. Cast this `Parameter` property to `int` and pass it to a method on the `ViewModel` named `InitializeItemDetailData`, which will be created in the next section:

```
protected override void OnNavigatedTo(NavigationEventArgs
e)
{
    base.OnNavigatedTo(e);
    var selectedItemId = (int)e.Parameter;
    if (selectedItemId > 0)
    {
        ViewModel.InitializeItemDetailData(selectedItemId);
    }
}
```

In the next section, you will add the final piece of the puzzle, the `ItemDetailsViewModel` class.

## Creating the ItemDetailsViewModel class

To add or edit items in the application, you will need a view model to bind to `ItemDetailsPage`. Right-click the `ViewModels` folder in **Solution Explorer** and add a new class named `ItemDetailsViewModel`.

The class will inherit from `BindableBase` like `MainViewModel`. The full class can be found on GitHub at <https://github.com/PacktPublishing/-Learn-WinUI-3.0/blob/master/Chapter04/Complete/MyMediaCollection/ViewModels/ItemDetailsViewModel.cs>. Let's review some of the important members of the class.

The constructor receives the two services from the container and initializes commands:

```
public ItemDetailsViewModel(INavigationService
navigationService, IDataService dataService)
{
    _navigationService = navigationService;
```

```

    _dataService = dataService;
    SaveCommand = new RelayCommand(SaveItem, CanSaveItem);
    CancelCommand = new RelayCommand(Cancel);
    PopulateLists();
    PopulateExistingItem(dataService);
    IsDirty = false;
}

```

A public method named `InitializeItemDetailData` will accept the `selectedItemId` parameter passed by `ItemDetailsPage.OnNavigatedTo`. It will call methods to populate the lists and initializes an `_isDirty` flag to help to enable `SaveCommand`:

```

public void InitializeItemDetailData(int selectedItemId)
{
    _selectedItemId = selectedItemId;
    PopulateLists();
    PopulateExistingItem(_dataService);
    IsDirty = false;
}

```

The two populate methods set up the list data for the three `ComboBox` controls and add existing item data if the page is in edit mode:

```

private void PopulateExistingItem(IDataService dataService)
{
    if (_selectedItemId > 0)
    {
        var item = _dataService.GetItem(_selectedItemId);
        Mediums.Clear();
        foreach (string medium in
            dataService.GetMediums(item.MediaType) .
                Select(m => m.Name))
            Mediums.Add(medium);

        _itemId = item.Id;
        ItemName = item.Name;
        SelectedMedium = item.MediumInfo.Name;
    }
}

```

```
        SelectedLocation = item.Location.ToString();
        SelectedItemType = item.MediaType.ToString();
    }
}

private void PopulateLists()
{
    ItemTypes.Clear();
    foreach (string iType in
        Enum.GetNames(typeof(ItemType)))
        ItemTypes.Add(iType);
    LocationTypes.Clear();
    foreach (string lType in
        Enum.GetNames(typeof(LocationType)))
        LocationTypes.Add(lType);
    Mediums = new TestObservableCollection<string>();
}
```

Most of this view model's properties are pretty simple, but `SelectedItemType` has some logic to repopulate the list of `Mediums` based on the `ItemType` selected. For instance, if you are adding a book to the collection, there's no need to see the DVD or CD mediums in the selection list:

```
public string SelectedItemType
{
    get => _selectedItemType;
    set
    {
        if (! SetProperty(ref _selectedItemType, value,
            nameof(SelectedItemType)))
            return;
        IsDirty = true;
        Mediums.Clear();
        if (!string.IsNullOrWhiteSpace(value))
        {
            foreach (string med in
                _dataService.GetMediums((ItemType)Enum.
                    Parse(typeof(ItemType), SelectedItemType)).
```

```
        Select(m => m.Name) )
        Mediums.Add(med) ;
    }
}
}
```

Lastly, let's look at the code that `SaveCommand` and `CancelCommand` will invoke to save and navigate back to `MainPage`:

```
private void SaveItem()
{
    MediaItem item;
    if (_itemId > 0)
    {
        item = _dataService.GetItem(_itemId);
        item.Name = ItemName;
        item.Location = (LocationType)Enum.Parse(typeof
            (LocationType), SelectedLocation);
        item.MediaType =
            (ItemType)Enum.Parse(typeof(ItemType),
                SelectedItemType);
        item.MediumInfo =
            _dataService.GetMedium(SelectedMedium);
        _dataService.UpdateItem(item);
    }
    else
    {
        item = new MediaItem
        {
            Name = ItemName,
            Location = (LocationType)Enum.Parse(typeof
                (LocationType), SelectedLocation),
            MediaType =
                (ItemType)Enum.Parse(typeof(ItemType)
                    , SelectedItemType),
            MediumInfo =
                _dataService.GetMedium(SelectedMedium)
        };
    }
}
```

```
    };
    _dataService.AddItem(item);
}
_navigationService.GoBack();
}
private void Cancel()
{
    _navigationService.GoBack();
}
```

The other change needed before you run the application to test the new page is to consume `ItemDetailsViewModel` from `ItemDetailsPage.xaml.cs`:

```
public ItemDetailsViewModel ViewModel { get; } =
(Application.Current as
App)?.Container.GetService<ItemDetailsViewModel>();
```

Now, run the app and try to add or edit an item—you should see the new page. If you are editing, you should also see the existing item data in the controls:

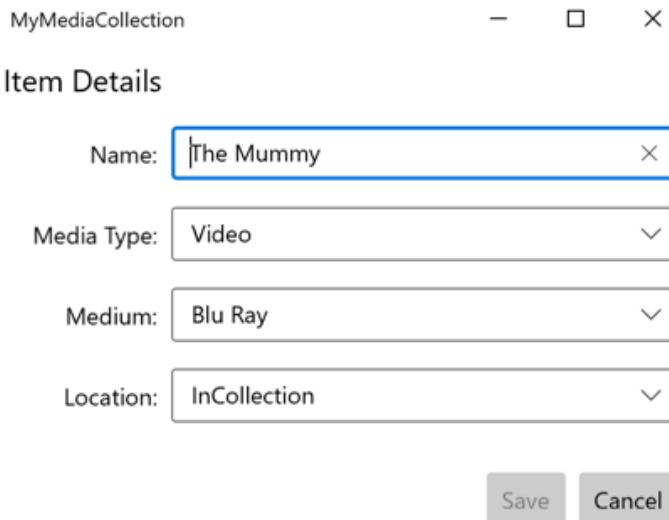


Figure 4.3 – The Item Details page with edit data populated

Great! Now when you save, you should see any added records or edited data appear on `MainPage`. Things are really starting to take shape in our project. Let's review what we have learned about WinUI and MVVM in this chapter.

## Summary

You have learned quite a bit about MVVM and WinUI page navigation in this chapter. You also learned how to create and consume services in your application, and you leveraged DI and DI containers to keep your ViewModels and services loosely coupled. Understanding and using DI is key to build testable, maintainable code. At this point, you should have enough knowledge to create a robust, testable WinUI application.

In the next chapter, you will learn about more of the available controls and libraries in WinUI 3.0.

## Questions

1. How do DI and IoC relate?
2. How do you navigate to the previous page in a WinUI application?
3. What object do we use to manage dependencies?
4. With Microsoft's DI container, what method can you call to get an object instance?
5. What is the name of the framework that queries objects in memory?
6. What event argument property can you access to get a parameter passed to a new Page?
7. Which dictionary type is safe to use across threads?



# 5

# Exploring WinUI Controls

WinUI offers a number of controls and APIs for developers building apps for Windows. The controls include new controls not previously available to Windows developers, as well as updated controls that were already available in WinUI 2.x or UWP. Using these new and updated controls with WinUI 3 enables their use in older versions of Windows 10 that did not previously support this full suite of components. Developers can also leverage **XamlDirect** APIs to get low-level access to XAML APIs.

In this chapter, we will cover the following topics:

- Learning more about the controls that are available in WinUI
- Exploring the **XAML Controls Gallery** application to learn about the WinUI controls
- How the **XamlDirect** APIs provide middleware authors with a performant way to declaratively create and manipulate XAML types in their code
- How to implement the `SplitButton` and `TeachingTip` controls in your application

By the end of this chapter, you will have a greater understanding of the controls and libraries in WinUI 3. You should also feel comfortable using the **XAML Controls Gallery** Windows application to explore the controls and find samples that demonstrate how to use them.

## Technical requirements

To follow along with the examples in this chapter, please reference the technical requirements in *Chapter 2, Configuring the Development Environment and Creating the Project*.

The source code for this chapter is placed here: <https://github.com/PacktPublishing/-Learn-WinUI-3.0/tree/master/Chapter05>.

## Understanding what WinUI offers developers

In *Chapter 1, Introduction to WinUI*, you learned a good amount of background about the origins of WinUI and UWP. That chapter also covered some of the controls available in the various releases of WinUI. Now, it's time to explore a few of these in more detail. Let's start by looking at a complete list of the controls available to developers in WinUI 3:

Animated visual player (Lottie) (WinUI)	Parallax view (WinUI)
Auto-suggest box	Password box
Button	Person picture (WinUI)
Calendar date picker	Pivot
Calendar view	Progress bar (WinUI)
Checkbox	Progress ring (WinUI)
Color picker (WinUI)	Radial gradient brush (WinUI)
Combo box	Radio button
Command bar	Rating control (WinUI)
Command bar flyout (WinUI)	Repeat button
Contact card	Rich edit box
Content dialog	Rich text block
Content link	Scroll viewer
Context menu	Search
Date picker	Semantic zoom
Dialogs and flyouts	Shapes
Drop down button (WinUI)	Slider
Flip view	Split button (WinUI)
Flyout	Split view
Forms	Swipe control (WinUI)
Grid view	Tab view (WinUI)
Hyperlink	Teaching tip (WinUI)
Hyperlink button	Text block
Images and image brushes	Text box
Inking controls	Time picker
List view	Toggle switch
Map control	Toggle button
Master/details	Toggle split button
Media playback	Tooltips
Menu bar (WinUI)	Tree view (WinUI)
Menu flyout	Two-pane view (WinUI)
Navigation view (WinUI)	Web view
Number box (WinUI)	Web view (Chromium-based MS Edge) (WinUI)

Figure 5.1 – The list of WinUI 3.0 controls

The items notated with *WinUI* in parentheses are WinUI controls that were not previously available in UWP applications. This is quite an extensive list of controls available to developers out of the box.

**Tip**

For an up-to-date list of the available controls, you can check this page on Microsoft Docs: <https://docs.microsoft.com/windows/uwp/design/controls-and-patterns/>.

If you have developed Windows applications before, most of these control names probably look familiar to you. In the next few sections, I will give you an overview of some of the controls that you may not have seen before.

## Animated visual player (Lottie)

The **AnimatedVisualPlayer** control is a WinUI control that can display **Lottie animations**. Lottie is an open source library that can parse and display animations on Windows, the web, iOS, and Android. These animations are created by designers in **Adobe After Effects** and exported in JSON format. You can learn more about Lottie animations on their website, <http://airbnb.io/lottie/#/>:

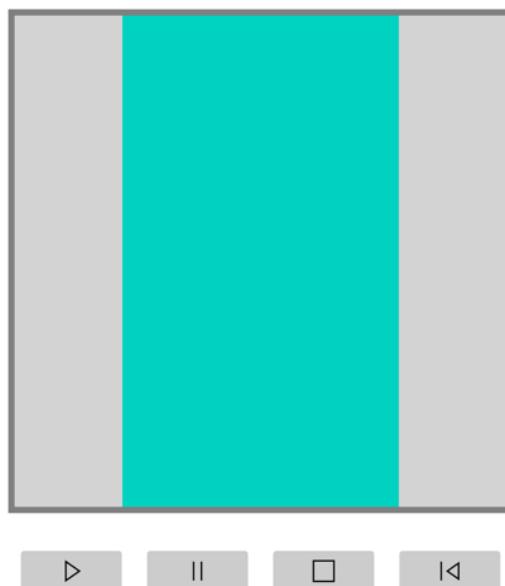


Figure 5.2 – The AnimatedVisualPlayer control

## NavigationView

**NavigationView** provides a user-friendly page navigation system. Use it to give users quick access to all your application's top-level pages. **NavigationView** can be configured to appear as a menu at the top of the application, with each page's link appearing like a tab across the top of the page:

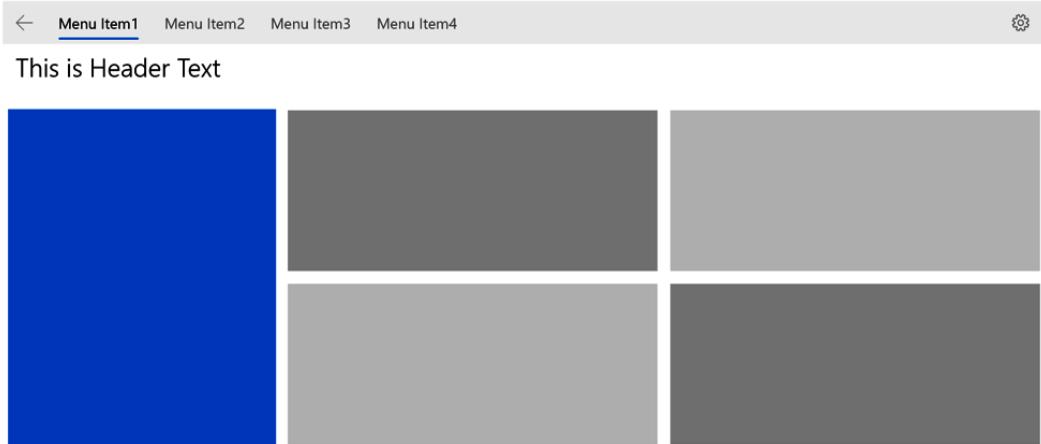


Figure 5.3 – NavigationView configured at the top of a page

**NavigationView** can also be configured to appear on the left-hand side of the page. This is a view that should be familiar to Windows and Android users. This menu format is commonly known as a **hamburger menu**. This is how the view appears when the menu is in a collapsed state:

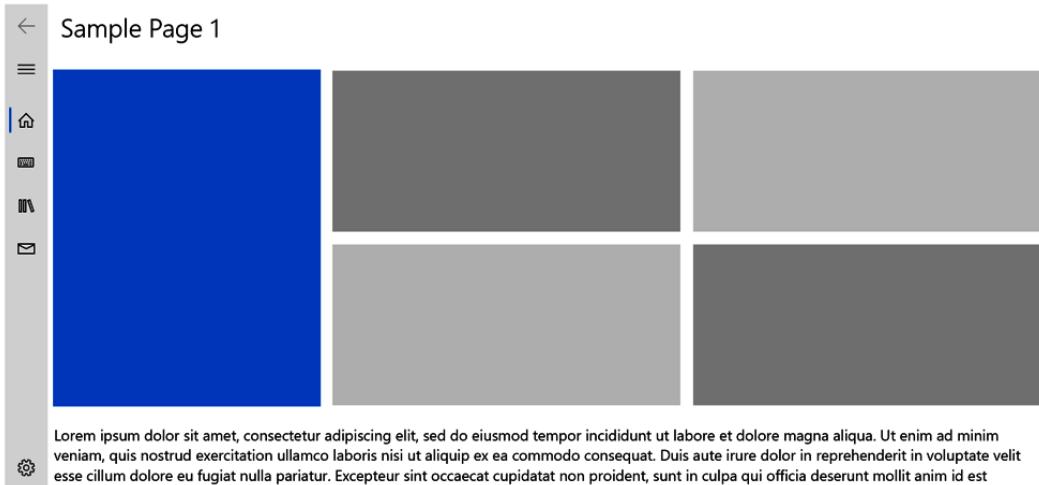


Figure 5.4 – A collapsed left NavigationView control

In either configuration, **NavigationView** can hide or show a back arrow to navigate to the previous page, and also has a settings menu item to show the application's settings page. If your application does not have a settings page, this item should be hidden. When the left menu is expanded by clicking the **hamburger** icon below the back arrow, the menu text is displayed, along with the respective icons:

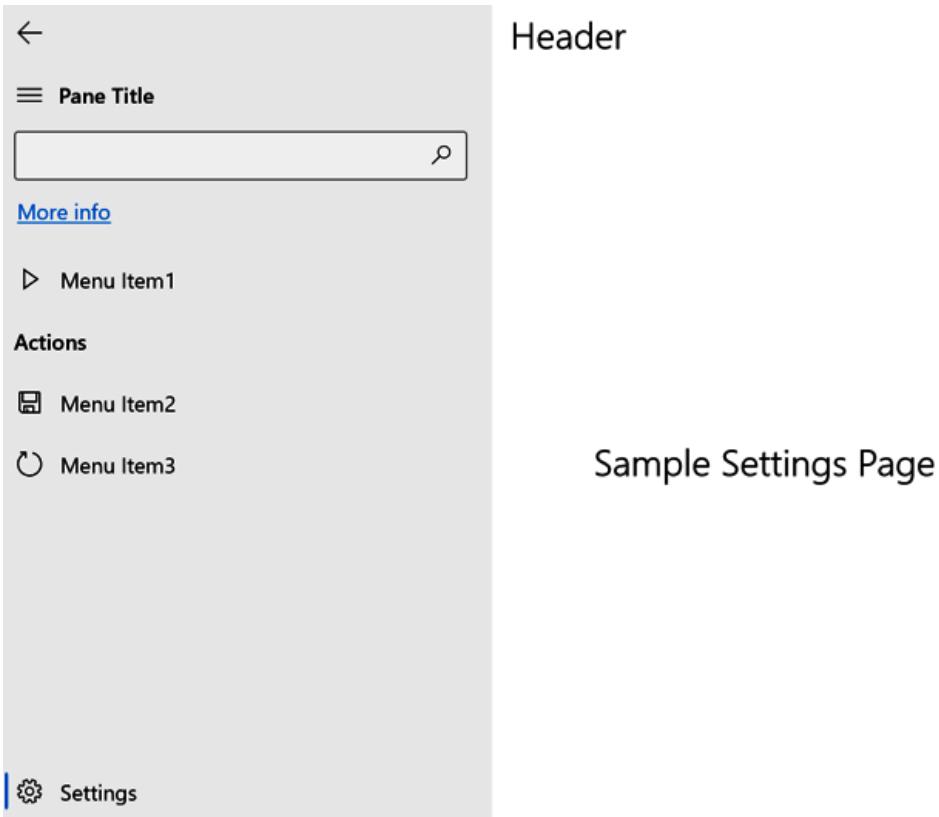


Figure 5.5 – The expanded left NavigationView control

Everything on the menu is configurable. You can group pages with the **Actions** section. You can hide or show the search box, the **More Info** link, and, as we previously mentioned, the **Settings** item. If your application has multiple top-level pages, you should consider using **NavigationView**.

## Parallax view

**Parallax scrolling** is a design concept that links scrolling a list or web page to scrolling a background image or other animations. A great example of a website with parallax scrolling is *History of the Web* (<https://webflow.com/ix2>). The `ParallaxView` control brings this concept to your WinUI applications. You link `ParallaxView` to `ListView` and a background image, and it will provide a parallax effect when `ListView` is scrolled. There are settings to control the relationship between scrolling the list and the amount the image scrolls. This effect has a great impact on users when it is not overused:

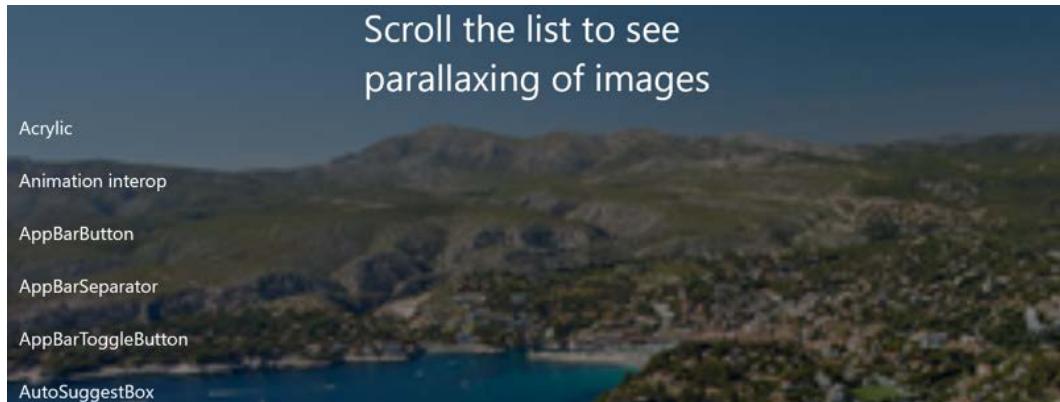


Figure 5.6 – The `ParallaxView` control scrolled to the top of a list



Figure 5.7 – The `ParallaxView` control scrolled partially through a list

## Rating control

Everyone is familiar with rating controls. You see them on shopping websites, streaming apps, and in online surveys. The WinUI RatingControl allows users to rate items in your application from 1 to 5 stars:



Figure 5.8 – The RatingControl displaying the user's rating

The control can also allow you to clear a rating by swiping left on the control, and can show a placeholder value before the user has provided their own rating. Applications typically use the placeholder value as a means of showing users the average rating given by other users.

## Two-pane view

The TwoPaneView control allows developers to target **Windows 10x** dual-screen devices with their applications. This control is recommended for applications targeting dual-screen devices when you have two content areas to display simultaneously, and you want them to automatically resize to fit the current layout and orientation of the device. Our **My Media Collection** application can be configured to display the list of media items on one screen and the currently selected item's details on the other:

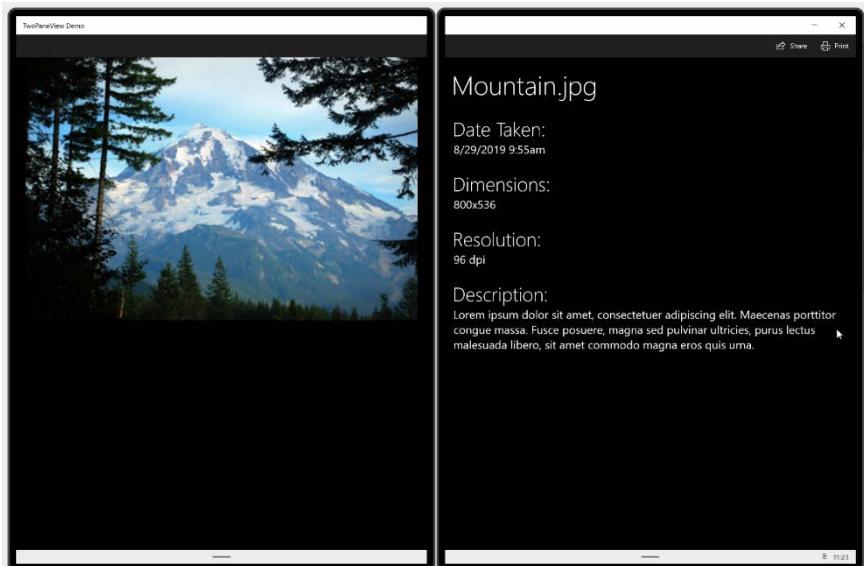


Figure 5.9 – The TwoPaneView control running on Windows 10x (Source: Microsoft Docs)

To learn more about Windows 10x and dual-screen devices, check out Microsoft's blog post announcing upcoming dual-screen Windows devices: <https://blogs.windows.com/windowsexperience/2019/10/02/introducing-windows-10x-enabling-dual-screen-pcs-in-2020/>.

**Note**

At the time of writing, **Windows 10x** and **TwoPaneView** were both in pre-release states. This information may change once this book has been published.

Now that we have explored a few of the controls that were added with WinUI, let's dive into a Windows application that makes it easy to explore them on your own.

## Exploring the XAML Controls Gallery Windows app

Because there are so many powerful and configurable WinUI controls available, the WinUI team at Microsoft decided to create an application where Windows developers can explore and even try the controls. The **XAML Controls Gallery** is a great tool to get familiar with the controls, decide which ones are a fit for your application, and get some sample code.

To install the XAML Controls Gallery, you can visit its Microsoft Store page on the web (<https://www.microsoft.com/p/xaml-controls-gallery/9msvh128x2zt>) or launch the Microsoft Store Windows app and search for *XAML Controls Gallery*. The Gallery app itself is open source. You can browse the code to learn more about it on **GitHub**: <https://github.com/microsoft/Xaml-Controls-Gallery/tree/winui3preview>.

Once it has been downloaded and installed, launch the application:



Figure 5.10 – The XAML Controls Gallery application

On the **What's New** page of the application, you can quickly see which control samples have been recently added or updated. The application uses a navigation system that should look familiar. The left-hand side of the page has a **NavigationView** control for quickly browsing or searching the different controls in the gallery.

#### Note

If you search for the controls shown in the previous section, you may notice that the screenshots provided for the controls in this chapter were taken from the XAML Controls Gallery application.

## Learning about the ScrollViewer control

Suppose you were thinking of adding scrolling capability to part of a page on your application. In the gallery application, you can click **Scrolling** on the left navigation menu and then click on the **ScrollViewer** card on the **Scrolling** page. That will bring you to the details page for the WinUI **ScrollViewer** control:

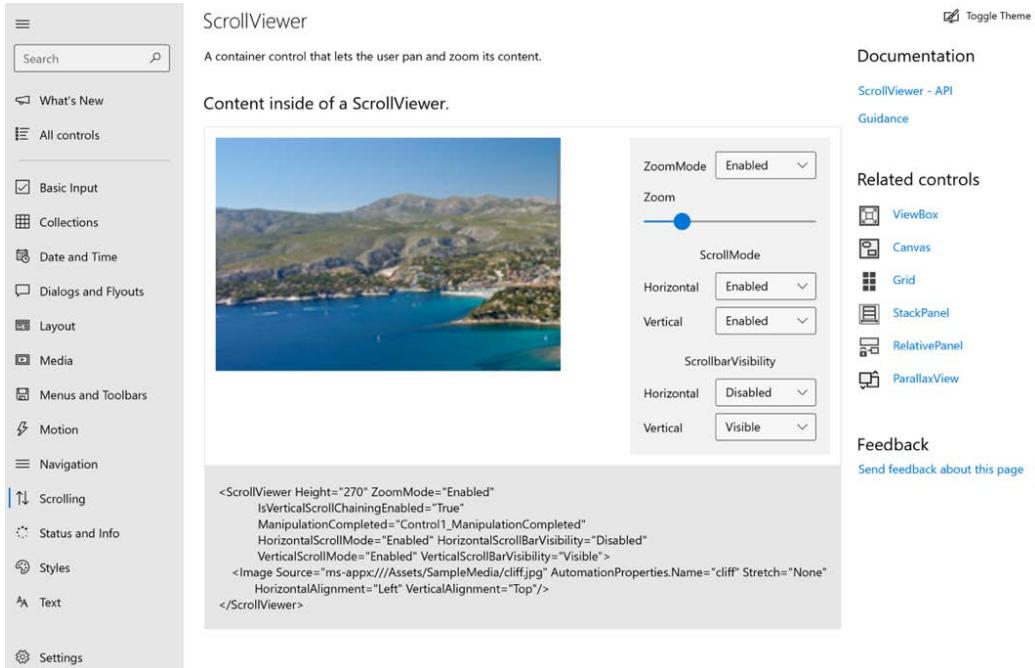


Figure 5.11 – The ScrollViewer control detail page in the XAML Controls Gallery

The control's detail page includes several sections. The header area provides a brief description of the control and its purpose. The right pane provides useful links to online documentation, other related controls in the gallery, and a link to provide feedback on the current gallery page.

The middle area of the page itself contains three sections: the rendered, usable control, and a properties panel. Using the properties panel, you can update some of the properties of **ScrollViewer** (and see the control immediately update). In the bottom center, you will find a panel containing the source code for the rendered control. The source control pane is very handy for copying the code to use as a starting point in your own project.

The gallery application's design also responds well to being resized. If you drag the right-hand side of the window to make it as narrow as possible, you will see the left and right panels collapse and the center area will realign to a single vertical column:

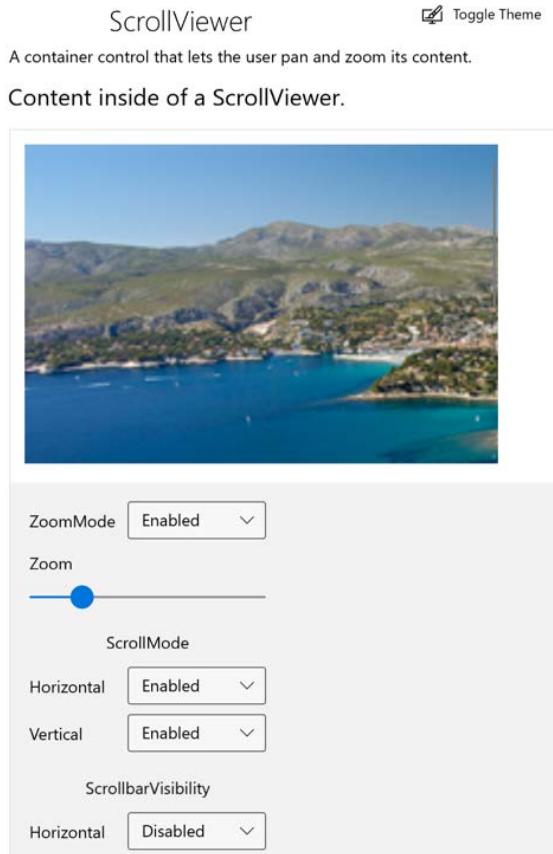


Figure 5.12 – The gallery application resized horizontally

You can imagine this view would fit well on a small Windows tablet.

Take some time to explore the controls in the gallery. Some of the code samples can be quite lengthy. Try changing some control properties and notice how the XAML code updates to reflect the new property values. This is a great way to learn XAML and familiarize yourself with the WinUI controls.

Now, let's take a tour of the new features in WinUI 3.0.

## Reviewing what's new in WinUI 3.0

Although WinUI 3.0 is a major release, the number of new features is not large, and only one new control has been added to the library: `WebView2`. That may be surprising to many people, but the first feature was quite an undertaking. We'll look at all of them in the following subsections.

## Backward compatibility

To make WinUI applications compatible with any version of Windows (starting with the Creator's Update, released in spring 2017), the WinUI team had to extract all of the UWP controls from the Windows SDK and move them to the new `Microsoft.UI.Xaml` libraries. The result of this work not only creates compatibility with more versions of Windows, but it also enables developers to consume WinUI, regardless of whether they are using UWP or Win32 as the underlying platform. C# and Visual Basic developers can build .NET 5 apps with WinUI for Desktop projects and .NET Native apps with WinUI for UWP projects, and C++ developers can consume WinUI on the Win32 platform.

Meanwhile, developers who maintain WPF, WinForms, and MFC applications will be able to extend them with WinUI components in XAML Islands using a Fluent UI. We will be covering Fluent UI in depth in *Chapter 7, Fluent Design System for Windows Applications*, while XAML Islands will be covered in *Chapter 10, Modernizing Existing Win32 Applications with WinUI and XAML Islands*. These Win32 applications hosting WinUI components will be able to run on the same versions of Windows as pure WinUI applications.

## Visual Studio tooling

Visual Studio can now add WinUI project templates via an extension on the Visual Studio Marketplace for the different combinations of platforms and languages discussed in the previous section. Starting a new project with WinUI in Visual Studio is literally as easy as going to **File | New Project**.

## Input validation

One of the features missing from UWP was built-in support for input validation with data binding. Validation is something most Windows and web developers take for granted today. Well, WinUI 3.0 has finally filled this gap for modern Windows applications. WinUI now has built-in support for data validation through the `INotifyDataErrorInfo` interface, which has three members:

```
public interface INotifyDataErrorInfo
{
    bool HasErrors { get; }
    event EventHandler<DataErrorsChangedEventArgs>
    ErrorsChanged;
    IEnumerable GetErrors(string propertyName);
}
```

There is an event handler to notify us when the collection of validation errors has changed, a Boolean property to indicate if the implementing class has any validation errors, and a method to get the collection of errors.

The implementing classes decorate members with attributes to define validation rules for the data. For example, the `LastName` property has attributes to indicate that it is a required item, and it cannot exceed 30 characters:

```
private string _lastName;
[Required]
[StringLength(30)]
public string LastName
{
    get { return _lastName; }
    set
    {
        _lastName = value;
        base.ValidateProperty(value);
        base.NotifyPropertyChanged(nameof(LastName));
    }
}
```

This snippet assumes that your base class implements the `INotifyDataErrorInfo` class and has a `ValidateProperty` method to handle the validation logic. The implementation is outside the scope of this chapter. Implementing data validation with WinUI will be covered in *Chapter 6, Leveraging Data and Services*.

## A new WebView

The one new control available to WinUI developers in version 3.0 is `WebView2`. This new version of `WebView` is built on the new, Chromium-based Microsoft Edge web browser. If you need to embed some web content into your app, `WebView2` is the control you should use to ensure maximum compatibility with modern web standards.

### Note

Using `WebView2` requires installing the runtime or any non-stable version of the new Microsoft Edge browser. For a full list of prerequisites, you can reference this page on Microsoft Docs: <https://docs.microsoft.com/en-us/microsoft-edge/webview2/gettingstarted/winui#prerequisites>.

Here is a screenshot of **WebView2** in **XAML Controls Gallery**:

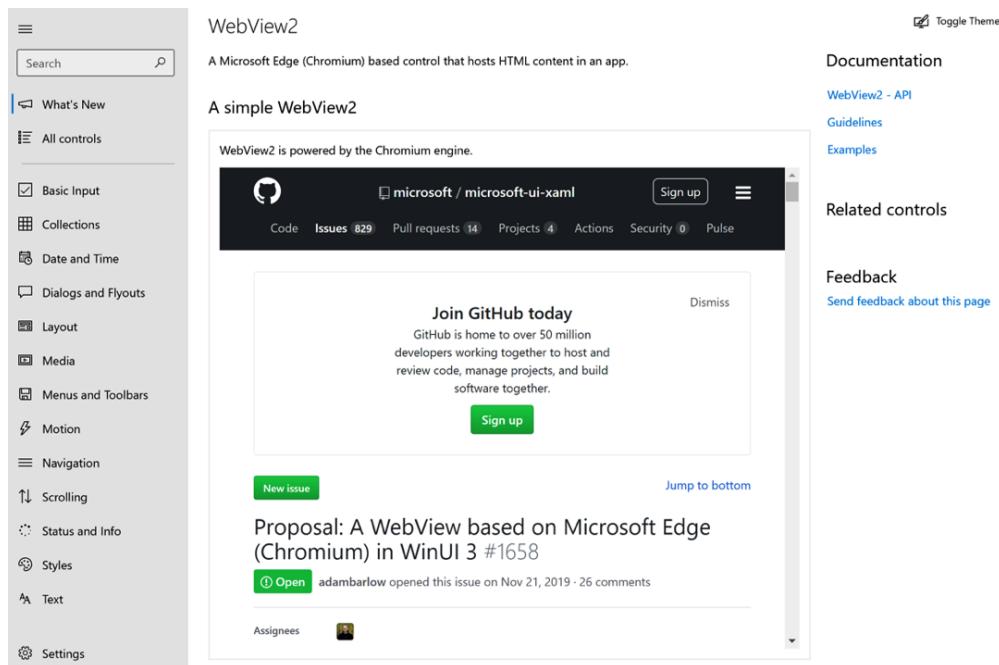


Figure 5.13 –WebView2 running in the Gallery application

Web content can be loaded into the control from the web or local network, from files in local storage, or from files embedded into the application's binaries. Here are some examples of loading into WebView2 from each type of source:

```

<!-- Load a website. -->
<WebView2 x:Name="WebView_Web" Source="https://www.packtpub.com"/>

<!-- Load web files from local storage. -->
<WebView2 x:Name="WebView_Local" Source="ms-appdata:///local/site/index.html"/>

<!-- Load web files embedded from the app package. -->
<WebView2 x:Name="WebView_EMBEDDED" Source="ms-appx-web:///web/index.html"/>

```

If you have some existing application written for the web, then this is a great way to integrate it into your new WinUI client application.

Let's shift gears for a moment and talk about working with UI components inside application middleware.

# Exploring the XamlDirect APIs for middleware authors

There are times when developers must create some UI elements at runtime. Maybe you need to dynamically add a section to a page and the content of that section is data dependent, meaning you cannot create the structure for the UI in XAML at design time. One way to handle creating a dynamic UI is by simply creating new instances of the necessary objects, and then adding them to whichever layout element their parent will be in. This will work, but Microsoft has created a more performant way to achieve this.

The **XamlDirect** APIs are a part of the `Microsoft.UI.Xaml.Core.Direct` namespace. These APIs provide greater performance when working with UI objects by providing access to XAML at a more primitive level. To demonstrate how XamlDirect can be used, let's create the same three child elements of a `StackPanel` in three different ways:

- Created in XAML
- Instantiated in code
- Created with XamlDirect

We are going to add a `TextBlock` and two `Button` elements to `StackPanel`. First, create them with XAML:

```
<StackPanel x:Name="ParentPanel">
    <TextBlock Text="Actions:" Margin="0,0,6,0"/>
    <Button Content="Save"/>
    <Button Content="Cancel"/>
</StackPanel>
```

This is how we have been creating our elements in the `My Media Collection` project throughout this book. Now, let's assume that we only have `ParentPanel` defined in XAML, and we want to add the three children in the C# code-behind file:

```
// Create the TextBlock
var ButtonsLabel = new TextBlock
{
    Text = "Actions:",
    Margin = new Thickness { Left = 0, Top = 0, Right = 6,
    Bottom = 0 }
};
```

```
// Create the Save button
var SaveButton = new Button
{
    Content = "Save"
};

// Create the Cancel button
var CancelButton = new Button
{
    Content = "Cancel"
};

// Add the controls to the StackPanel
ParentPanel.Children.Add(ButtonsLabel);
ParentPanel.Children.Add(SaveButton);
ParentPanel.Children.Add(CancelButton);
```

This is also straightforward. We create an instance of each new control, set some properties, and add them as children of `ParentPanel`. This will have the same result as if they were defined in the XAML. However, we can do the same thing with `XamlDirect` and make it perform better than this example. This third example uses `XamlDirect`:

```
// Get the XamlDirect instance
XamlDirect xd = XamlDirect.GetDefault();

// Create the TextBlock with a right margin
IXamlDirectObject ButtonsLabel =
    xd.CreateInstance(XamlTypeIndex.TextBlock);
xd.SetThicknessProperty(ButtonsLabel, XamlPropertyIndex.
    FrameworkElement_Margin, new Thickness { Left = 0, Top = 0,
    Right = 6, Bottom = 0 });
xd.SetStringProperty(ButtonsLabel, XamlPropertyIndex.TextBlock_
    Text, "Actions:");

// Create the Save button
IXamlDirectObject SaveButton = xd.CreateInstance(XamlTypeIndex.
    Button);
xd.SetObjectProperty(SaveButton, XamlPropertyIndex.
    ContentControl_Content, "Save");

// Create the Cancel button
IXamlDirectObject CancelButton =
    xd.CreateInstance(XamlTypeIndex.Button);
xd.SetObjectProperty(CancelButton, XamlPropertyIndex.
```

```
ContentControl_Content, "Cancel");  
// Add the controls to the StackPanel  
ParentPanel.Children.Add((UIElement)  
xd.GetObject(ButtonsLabel));  
ParentPanel.Children.Add((UIElement)xd.GetObject(SaveButton));  
ParentPanel.Children.Add((UIElement)  
xd.GetObject(CancelButton));
```

This code is faster, but it is not as intuitive to read or write. The `XamlDirect` object has methods to create control instances, set properties of different data types, and get `UIElement` instances from a corresponding `IXamlDirectObject` instance.

For an example like this, the performance difference will be barely measurable. However, if your middleware is creating hundreds of controls, and each control requires dozens of properties to be set, you will want to use `XamlDirect` to improve the application's performance.

If you want to read more about how to use `XamlDirect`, Microsoft Docs has a complete API reference guide: <https://docs.microsoft.com/en-us/uwp/api/windows.ui.xaml.core.direct>.

Now that we've learned about the new controls, let's get back to our project and add a couple of them to it.

## Adding some new controls to the project

In this section, we are going to use two controls that are only available to Windows applications with WinUI. We are going to change the `Save` button to `SplitButton` to allow users to save and return to a list of items, or save and continue adding another item to the item details page. Then, we will add a `TeachingTip` control to inform users of the new saving capabilities. To follow along with these steps, you can use the `Start` project on GitHub (<https://github.com/PacktPublishing/-Learn-WinUI-3.0/tree/master/Chapter05/Start>). Let's start by updating the `Save` button.

## Using the SplitButton control

Follow these steps:

1. First, in `ItemDetailsViewModel`, add a new `SaveItemAndContinue` method to be bound to the `Click` event of our new `SplitButton`:

```
public void SaveItemAndContinue()
{
    SaveItem();
    _itemId = 0;
    ItemName = "";
    SelectedMedium = null;
    SelectedLocation = null;
    SelectedItemType = null;
    IsDirty = false;
}
```

In the `SaveItemAndContinue` method, we are calling `SaveItem` and then resetting all the item state data so that it's ready for a new item to be entered. The one problem here is that `SaveItem` currently navigates back to the previous page. Let's fix that.

2. To remove the call from `SaveItem` (to return to the previous page), we need a new method for the Save button to use. Create a new method named `SaveItemAndReturn`:

```
public void SaveItemAndReturn()
{
    SaveItem();
    _navigationService.GoBack();
}
```

Here, we are calling `SaveItem` and then navigating back to the previous page. The call to `_navigationService.GoBack` can now be removed from `SaveItem`.

3. We are now using `x:Bind` to bind directly to the save methods instead of using `ICommand`. So, you can remove the `SaveCommand` property and the `CanSaveItem` method from `ViewModel`. You can also remove this line from the `ItemDetailsViewModel` constructor:

```
SaveCommand = new RelayCommand(SaveItemAndReturn,
    CanSaveItem);
```

4. Finally, open `ItemDetailsPage` and update the save button to be a `SplitButton` instead:

```
<SplitButton x:Name="SaveButton"
             Content="Save and Return"
             Margin="8,8,0,8"
             Click="{x:Bind ViewModel.SaveItemAndReturn}"
             IsEnabled="{x:Bind ViewModel.IsDirty,
             Mode=OneWay}"
             <SplitButton.Flyout>
               <Flyout>
                 <Button Content="Save and Create New"
                        Click="{x:Bind ViewModel.
                        SaveItemAndContinue}"
                        IsEnabled="{x:Bind ViewModel.IsDirty,
                        Model=OneWay}"
                        Background="Transparent"/>
                 <Button Content="Save and Return"
                        Click="{x:Bind ViewModel.
                        SaveItemAndReturn}"
                        IsEnabled="{x:Bind ViewModel.IsDirty,
                        Mode=OneWay}"
                        Background="Transparent"/>
               </Flyout>
             </SplitButton.Flyout>
           </SplitButton>
```

Here, the content of `SplitButton` is updated to `Save and Return`, updating the binding to use the `Click` event to invoke the action, and the `IsEnabled` property with `IsDirty`. A new child `Flyout` item is also added. `Flyout` contains a `StackPanel` with `Button` controls for both the `Save and Return` and the new `Save and Create New` action, with the `Click` event invoking `SaveItemAndContinue`.

That's it! Now, run the application and try out this new feature. Here's how the new button looks when you click the drop-down arrow:

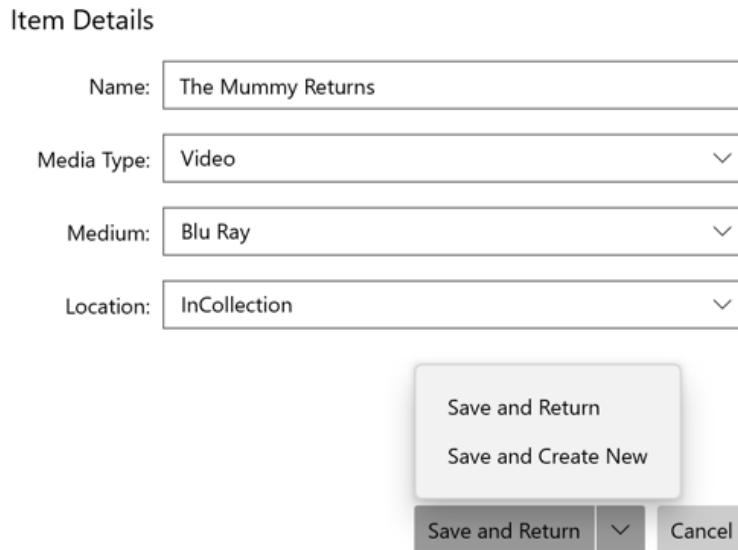


Figure 5.14 – Using the new SplitButton to save items

## Adding a TeachingTip to the save button

A `TeachingTip` control is a great way to educate users about features in your application. It is a small popup with header text and content text. You may have seen them in the Windows applications you use.

`TeachingTip` can either be linked directly to a control on the page or it can be placed directly on the page with an optional `PreferredPlacement` property, controlling where it appears on the page. It can be configured to be dismissed by the user with a close button or automatically when the user starts interacting with the page.

To add a `TeachingTip` for our save `SplitButton`, add it to the control's `Resources` in `ItemDetailsPage`:

```
<SplitButton x:Name="SaveButton" Content="Save and Return"  
Margin="8,8,0,8" Command="{x:Bind ViewModel.SaveCommand}">  
  <SplitButton.Flyout>  
    <Flyout>  
      <Button Content="Save and Create New"  
Command="{x:Bind ViewModel.SaveAndContinueCommand}"  
Background="Transparent"/>
```

```
</Flyout>
</SplitButton.Flyout>
<SplitButton.Resources>
    <TeachingTip x:Name="SavingTip"
        Target="{x:Bind SaveButton}"
        Title="Save and create new"
        Subtitle="Use the dropdown button
option to save your item and create another.">
    </TeachingTip>
</SplitButton.Resources>
</SplitButton>
```

In `TeachingTip`, we're binding `Target` to `SaveButton` and setting `Title` and `Subtitle` to educate users about the new *Save and create new* feature.

There's an additional call needed in the `ItemDetailsPage` constructor to make the tip appear:

```
SavingTip.IsOpen = true;
```

If you run the application now, `TeachingTip` is going to appear every time the user opens `ItemDetailsPage`. This is going to quickly annoy our users. We can add a little bit of code to `ItemDetailsPage.xaml.cs` to save a user setting indicating that the current user has already seen this `TeachingTip`. Then, the next time we load the page, we'll check this setting and skip the code that loads the tip.

We're going to leverage Windows local storage to save and load this user setting:

```
Windows.Storage.ApplicationDataContainer localSettings =
Windows.Storage.ApplicationData.Current.LocalSettings;
// Load the user setting
string haveExplainedSaveSetting = localSettings.
Values[nameOf(SavingTip)] as string;
// If the user has not seen the save tip, display it
if (!bool.TryParse(haveExplainedSaveSetting, out bool result)
|| !result)
{
    SavingTip.IsOpen = true;
    // Save the teaching tip setting
    localSettings.Values[nameOf(SavingTip)] = "true";
}
```

Now, the user will only see this tip the first time they load `ItemDetailsPage`.

Let's see `TeachingTip` in action. Run the application and click **Add/Edit Item** to open `ItemDetailsPage`:

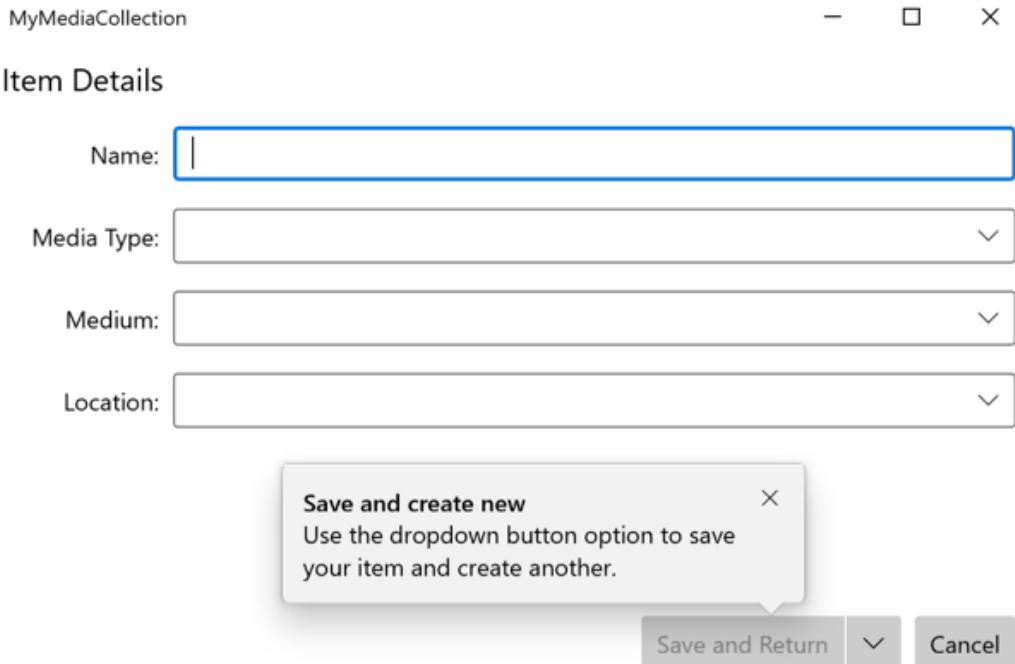


Figure 5.15 – Using the new `TeachingTip` control

Now, the application has a new feature and a great way to inform users of how to use it. Let's wrap up and discuss what we have learned in this chapter.

## Summary

In this chapter, we explored many of the controls available in WinUI 3.0. We learned that the **XAML Controls Gallery** application is a great tool for exploring the controls available to WinUI developers. If you build applications that will be loading many controls at runtime, then it is a great option for you to explore the **XamlDirect** APIs. At the end, we learned about, and added, a couple of new WinUI controls to our application.

In the next chapter, we will learn more about services and will start persisting the data for our media items between sessions.

## Questions

1. Which WinUI control can display Lottie animations?
2. Which WinUI control can display HTML content with the new Chromium-based Microsoft Edge browser?
3. Which WinUI APIs can create controls very quickly?
4. What control would you use to educate users about a new feature?
5. What application can you download from the Microsoft Store to learn about all the WinUI controls?
6. What type can be used to save and load user settings between sessions?
7. What is the earliest version of Windows that can run a WinUI 3.0 application?

## Further reading

**Two-pane view control:** <https://docs.microsoft.com/en-us/windows/uwp/design/controls-and-patterns/two-pane-view>.



# 6

# Leveraging Data and Services

Managing live data is a core part of what applications do. Learning how to load and save that data is an important aspect of WinUI development. Some of the key points of data management are using concepts such as state management, the service locator pattern, and data validation. We will be covering these concepts and putting them to use in our application.

In this chapter, we will cover the following topics:

- Understanding the WinUI application life cycle
- Learning to use SQLite to store application data
- Learning to use Dapper to quickly map objects in your data service
- Exploring the service locator pattern further and working with our data service
- Implementing data validation to provide useful feedback to users about invalid data before it is saved

By the end of this chapter, you will have a working understanding of the WinUI application life cycle and will know how to manage data and state in your projects.

## Technical requirements

To follow along with the examples in this chapter, the following software is required:

- Windows 10 version 1803 (version 17134) or newer
- Visual Studio 2019 Version 16.9 or newer with the following workloads: .NET Desktop Development and Universal Windows Platform Development

The source code for this chapter is available on GitHub at <https://github.com/PacktPublishing/-Learn-WinUI-3.0/tree/master/Chapter06>.

## Managing application state with app life cycle events

Before working with data in any application, it is important to understand the application life cycle for the target application platform. We have touched on these concepts briefly, but now, it's time to take a deeper dive into the Windows application life cycle for **WinUI on UWP** applications.

### Exploring Windows application life cycle events

**WinUI on UWP** applications have a different set of life cycle events than Win32 applications. Win32 applications are either running or they're not. There are several events that occur while launching and shutting down WPF and WinForms applications:

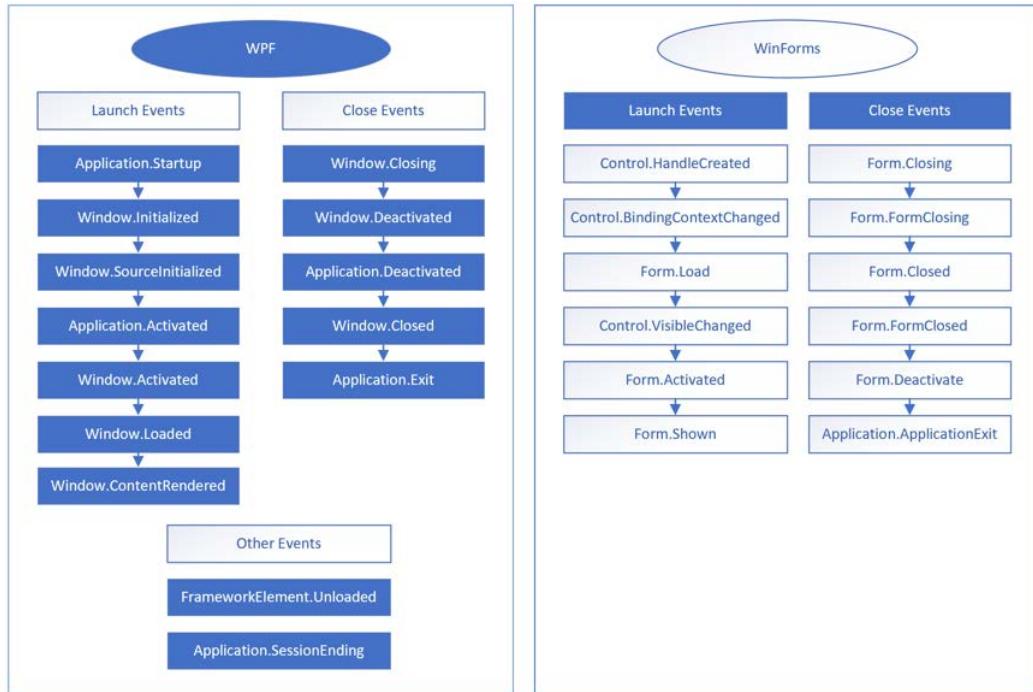


Figure 6.1 – Win32 application life cycle events

### Note

We won't go into the details here, as our primary focus is building **WinUI on UWP** applications. However, the two WPF events that fall outside of the launch and close their sequences are as follows:

- FrameworkElement.Unloaded:** This event fires when an element is removed from the WPF **Visual Tree**. It does not fire during application shutdown.
- Application.SessionEnding:** This event fires when the current Windows user logs off or shuts Windows down. In the event handler, you can request that Windows cancel the process by setting the `SessionEndingCancelEventArgs.Cancel` property to `true`.

## Life cycle events of WinUI applications

Let's talk about the WinUI life cycle. Life cycle events give you a chance to initialize any data and state when your application starts or resumes execution, and also allows you to clean up or save the state when the application is closed or suspended.

**Note**

When writing WinUI apps for the UWP platform, the WinUI life cycle and UWP life cycle are one and the same. We will refer to this as the WinUI life cycle in this chapter. This will be different when we discuss building WinUI on Desktop projects in *Chapter 8, Building WinUI Applications with .NET 5*. For some additional background on UWP application life cycle events and their execution state, you can read this Microsoft Docs page: <https://docs.microsoft.com/en-us/windows/uwp/launch-resume/app-lifecycle>.

Now, let's examine the WinUI life cycle events in a little more detail. Before discussing the specific events that you can handle in your WinUI application, you should understand the different application execution states:

- **NotRunning:** The application is not running yet. When it is launched by the user or Windows, it will go through initialization events.
- **Running:** The application is currently running. It could enter this state from the NotRunning or Suspended states. It could be running in the foreground or the background. An application moves to the background when it is minimized by the user in Windows 10, or when the user switches to another app on Windows 10 running in tablet mode or on other platforms, such as Xbox. Apps running in the background will be moved to the Suspended state by Windows after a few seconds.
- **Suspended:** The application was suspended by Windows after it moved to the background. The state of objects in memory is retained and the navigation stack is preserved. References to external resources such as files are lost and must be acquired again after the application reactivates. The application will return to the Running state after the user switches back to it.
- **Terminated:** The application can be terminated by Windows if it has been suspended and Windows needs to reclaim resources. It is up to the application to save any data in memory that is needed for the next session.
- **ClosedByUser:** The user explicitly closed the application. The application goes through the same closing life cycle events as if it were terminated by Windows.

A **WinUI on UWP** application goes through a set of life cycle events, similar to Win32 applications, only there are more possible application events, as we will see in the following subsections.

## OnLaunched

You can intercept the launch event by overriding the `OnLaunched` method of the application class. It is important to remember that this event may not mean that the app has been launched by the user. Windows will prelaunch frequently used applications when there are available resources. Your application should use this event to begin loading data that will be needed when it is activated. It should not do something such as play a video or music. Interactive things like these should be handled when the main page is loaded.

### Note

Applications must opt in to be considered for prelaunch by Windows. To learn more about opting in and handling prelaunch, read this Microsoft Docs article: <https://docs.microsoft.com/en-us/windows/uwp/launch-resume/handle-app-prelaunch>.

You should check the `PreviousApplicationState` event argument to determine the proper steps to take to launch the app, based on its previous state. Once the launch process has completed, the application will be in the `Running` state.

## OnActivated

As we discussed previously, application activation is invoked by a *process* rather than directly by the user. Your application should handle the `OnActivated` event to determine its `ActivationKind` and take the appropriate action. A few examples of `ActivationKind` are as follows:

- **ShareTarget:** Your app is registered as a **share target**, and another application wants to share some data with yours.
- **File:** Your app is associated with a file type that was launched by the user.
- **Device:** Your application is registered with AutoPlay for a device that was inserted.
- **CommandLineLaunch:** The application was launched from the Windows command line.
- **Terminated:** The app was terminated by Windows after being suspended and is now being launched again.

### Note

To read about all of the `ApplicationKind` values, check out this Microsoft Docs page: <https://docs.microsoft.com/uwp/api/Windows.ApplicationModel.Activation.ActivationKind>.

There are also more specific activation events that should be used for your application's most common activation types. For example, for the `File` option's `ActivationKind` event, there is an `Application.OnFileActivated` method.

Like application launching, the application will enter the `Running` state when it is activated. Both the `OnLaunched` and `OnActivated` events can lead to your app entering a running state. How it gets there, and which events fire, depends on whether the app is launched by the user or activated by the system.

## Running apps, foreground, and background

When an application is running, it can be running in the foreground or the background. The `EnteredBackground` and `LeavingBackground` application events, which were added in version 14393 of the Windows 10 SDK, can be handled to detect these states being switched between. Add any logic to prepare your application's active page to be displayed to the user in the `LeavingBackground` event handler. You can release some application memory when it enters the background by pausing any animations or other UI rendering. If your app needs more time to complete operations when entering the background, it can use the `GetDeferral` method in the `EnteredBackgroundEventArgs` parameter.

### Note

If your app needs to continue running in the background, it can declare this as a capability. See the Microsoft Docs page titled *Run in the background indefinitely* here: <https://docs.microsoft.com/en-us/windows/uwp/launch-resume/run-in-the-background-indefinitely>.

## Suspending

When an app has been in the background for a few seconds or the user locks Windows, the application will enter the `Suspended` state, assuming it has not requested to remain running in the background. Handle the `Application.Suspending` event to save any state that was not saved when the app entered the background. However, there is limited time to perform any activities in the `Suspending` event. To request additional time, use the `SuspendingOperation.GetDeferral` method from the event arguments. This works like `GetDeferral` in `EnteredBackground`, but this was the only way to request a deferral in Windows 10 versions prior to version 14393.

The Visual Studio WinUI template requests a deferral by default in the `App.OnSuspending` event handler:

```
private void OnSuspending(object sender, SuspendingEventArgs e)
{
    var deferral = e.SuspendingOperation.GetDeferral();
    //TODO: Save application state and stop any background
    //activity
    deferral.Complete();
}
```

This is your last chance to save any state and release external resources before the `deferral.Complete` call. It is a best practice to save user data and other application state as it changes so that the code that runs in `OnSuspending` allows your application to suspend quickly. If your application takes too long to suspend, Windows could forcibly terminate it. Beyond the `Suspending` state, there are no other events to handle when the app is closing.

#### Note

To debug the suspend process, and events such as `EnteredBackground`/`LeavingBackground`, from Visual Studio, show the **Debug Location** toolbar from **View | Toolbars** and use the **Suspend** button. Windows will not suspend an application while the Visual Studio debugger is attached.

## Resuming

If an application resumes from a `Suspended` state, it will not have lost any data from memory unless it was released while being suspended by the app itself. External resources such as file handles or network connections will need to be refreshed. This can all be managed in the `Application.Resuming` event handler.

#### Note

For more information about handling life cycle events, read the following Microsoft Docs page: <https://docs.microsoft.com/en-us/windows/uwp/launch-resume/app-lifecycle>.

Now that you have a solid understanding of the WinUI's life cycle, let's start working with some real data that will need to be persisted between sessions.

## Creating a SQLite data store

Until now, the **My Media Collection** project has only been working with data stored inside in-memory collections. This means that every time the app is closed, all the user's data is lost. It has also meant calling a method to populate all the lists with hardcoded seed data with every launch.

In the previous chapter, we took the first step in creating a maintainable data service for the application. By creating a data service class that implements `IDataService`, no changes will be required in the `ViewModel` classes. This section will focus on creating a new `SqliteDataService` class so that you can use SQLite for data access. The starting code for this chapter can be found on GitHub at <https://github.com/PacktPublishing/-Learn-WinUI-3.0/tree/master/Chapter06/Start>.

### What is SQLite?

**SQLite** (found at <https://sqlite.org/>) is a SQL-based database that is frequently used by mobile apps and small desktop applications. It is a popular choice because it is small, fast, and self-contained in a single file. There are SQLite libraries available for virtually every platform. We will be using Microsoft's **Microsoft.Data.Sqlite** ADO.NET provider for SQLite.

#### Note

For more information about Microsoft's SQLite provider, you can read <https://docs.microsoft.com/en-us/dotnet/standard/data/sqlite/>. To learn more about using SQLite with WinUI and UWP projects, check out this Microsoft Docs article: <https://docs.microsoft.com/en-us/windows/uwp/data-access/sqlite-databases>.

## Adding SQLite to DataService

Follow these steps:

1. Start by adding the **Microsoft.Data.Sqlite** NuGet package to the **MyMediaCollection** project by opening **Package Manager Console** from **View | Other Windows | Package Manager Console** and running the following command. Also, make sure the **MyMediaCollection** project is selected in the **Package Manager Console** window's project dropdown before running this command:

```
Install-Package Microsoft.Data.Sqlite
```

Running this command is equivalent to adding the package from the **Manage NuGet Packages** window.

- When the process completes, open the `SqliteDataService` class and start by adding four `using` statements to the top of the file:

```
using Microsoft.Data.Sqlite;
using System.IO;
using System.Threading.Tasks;
using Windows.Storage;
```

The `System.IO` and `Windows.Storage` namespaces will be used when we're initializing the SQLite database file, and we'll need the `System.Threading.Tasks` namespace imported to work with some *async* tasks.

- Next, add a new constant to the class to hold the database's filename:

```
private const string DbName = "mediaCollectionData.db";
```

- Next, let's create a `private` method to create or open the database file, create a `SqliteConnection` to the database, open it, and return it to the caller. This method can be used throughout the class whenever a new database connection is needed. The database file will be created in the user's `LocalFolder`, which means the application's data will be saved with the user's local Windows profile data:

```
private async Task<SqliteConnection>
GetOpenConnectionAsync()
{
    await ApplicationData.Current.LocalFolder.
    CreateFileAsync(DbName, CreationCollisionOption.
    OpenIfExists).AsTask().ConfigureAwait(false);
    string dbPath = Path.Combine(ApplicationData.Current.
    LocalFolder.Path, DbName);
    var cn = new SqliteConnection($"Filename={dbPath}");
    cn.Open();
    return cn;
}
```

Notice that we have declared this method as `async` and that it uses the `await` keyword when opening or creating the file. It is a good practice to use `async/await` when using external resources such as files, network connections, or databases to keep your application responsive.

**Note**

To find out more about `async/await` with C# and .NET, Microsoft Docs has a great article to get you started: <https://docs.microsoft.com/en-us/dotnet/csharp/async>.

5. Next, create two methods in order to create the `MediaItems` and `Mediums` tables in the database. These will be called each time the app launches, but the SQL code only creates the tables if they do not exist. The `SqlCommand` object accepts the `tableCommand` query string and `SqliteConnection`. It has several methods it can use to execute the command, depending on whether any data is expected to be returned by the query. In our case, no return values are expected, so `ExecuteNonQueryAsync` is the best of the `async` options for these two methods:

```
private async Task CreateMediumTableAsync(SqliteConnection db)
{
    string tableCommand = @"CREATE TABLE IF NOT
        EXISTS Mediums (Id INTEGER PRIMARY KEY
        AUTOINCREMENT NOT NULL,
        Name NVARCHAR(30) NOT NULL,
        MediumType INTEGER NOT NULL)";
    var createTable = new SqlCommand(tableCommand,
        db);
    await createTable.ExecuteNonQuery();
}

private async Task
CreateMediaItemTableAsync(SqliteConnection db)
{
    string tableCommand = @"CREATE TABLE IF NOT
        EXISTS MediaItems (Id INTEGER PRIMARY KEY
        AUTOINCREMENT,
        Name NVARCHAR(1000) NOT NULL,
        ItemType INTEGER NOT NULL,
        MediumId INTEGER NOT NULL,
        LocationType INTEGER,
        CONSTRAINT fk_mediums
        FOREIGN KEY(MediumId)
        REFERENCES Mediums(Id))";
```

```
        var createTable = new SqliteCommand(tableCommand,
db);
        await createTable.ExecuteNonQueryAsync();
    }
```

6. Now, we will create two methods for the `Mediums` table – one to insert a row into the table and another to read all the rows from the table:

```
private async Task InsertMediumAsync(SqliteConnection db,
Medium medium)
{
    var insertCommand = new SqliteCommand
    {
        Connection = db,
        CommandText = "INSERT INTO Mediums VALUES (NULL,
@Name, @MediumType);"
    };
    insertCommand.Parameters.AddWithValue($"@nameof(medium.Name)", medium.Name);
    insertCommand.Parameters.AddWithValue("@MediumType",
(int)medium.MediaType);
    await insertCommand.ExecuteNonQuery();
}

private async Task<IList<Medium>>
GetAllMediumsAsync(SqliteConnection db)
{
    IList<Medium> mediums = new List<Medium>();
    var selectCommand = new SqliteCommand("SELECT Id,
Name, MediumType from Mediums", db);
    SqlDataReader query = await selectCommand.
ExecuteReaderAsync();
    while (query.Read())
    {
        var medium = new Medium
        {
            Id = query.GetInt32(0),
            Name = query.GetString(1),
            MediaType = (ItemType)query.GetInt32(2)
        };
    }
}
```

```
        mediums.Add(medium) ;  
    }  
    return mediums;  
}
```

There's a lot of code needed for these simple operations. The `insert` method needs to add parameters for each property to be saved in the table, and the `select` method has a `while` loop to add each table's record to the collection. Let's see if we can simplify this.

Before we implement the remaining methods for the **Create, Read, Update, Delete (CRUD)** operations, a new library must be added to the project to simplify the data access code we'll be writing.

## Leveraging a Micro ORM to simplify data access

As you saw in the previous section, writing data access code for even the simplest application can take some time.

**Object Relational Mappers (ORMs)** such as **Entity Framework Core (EF Core)** can greatly simplify and reduce the code required, but they can be overkill for a small app with just a handful of tables. Let's look at **Micro ORMs**, which are lightweight frameworks that handle mapping data between objects and data queries.

### Note

EF Core is a popular ORM for .NET developers. If you want to learn more about how to use EF Core with your projects, you can get the **Packt** title *Mastering Entity Framework Core 2.0* at <https://www.packtpub.com/application-development/mastering-entity-framework-core-20>.

The framework we will be using for data access in our project, **Dapper**, is an open source .NET Micro ORM that was created by the developers at **Stack Overflow**. You can learn more about Dapper at <https://stackexchange.github.io/Dapper/> and get the package on NuGet: <https://www.nuget.org/packages/Dapper>.

Dapper is quickly growing in popularity among the .NET community. While it doesn't offer some of the features of EF Core, such as model generation or entity change tracking, it does make it easier to write a fast, slim data layer. When you add the **Dapper Contrib** library (<https://www.nuget.org/packages/Dapper.Contrib>) to the mix, it is even easier to write the CRUD methods for your application.

## Adding Dapper to the project

Let's get started:

1. Let's start by adding Dapper and Dapper.Contrib to the MyMediaCollection project. Open the **Package Manager Console** window again and add the two packages to your project:

```
Install-Package Dapper
Install-Package Dapper.Contrib
```

2. Now, let's revisit the `InsertMediumAsync` method. If we use the `QueryAsync` method offered by Dapper, we can reduce the code from our method to this:

```
private async Task InsertMediumAsync(SqliteConnection db,
Medium medium)
{
    var newIds = await db.QueryAsync<long>(
        $@"INSERT INTO Mediums
        ({nameof(medium.Name)}, MediumType)
        VALUES
        (@{nameof(medium.Name)}, @{nameof(medium.
MediaType)} );
        SELECT last_insert_rowid()", medium);
    medium.Id = (int)newIds.First();
}
```

The code we wrote to set the values of the query parameters is now gone. Dapper maps them for us from the `medium` object, which is passed into its `QueryAsync` method. You must ensure that the parameter names in the SQLite query match the property names on our object for Dapper's automatic mapping to work.

3. As a bonus, we can also get the generated ID back from the `QueryAsync` call by adding the following SQLite code, which will return it after `INSERT` completes:

```
SELECT last_insert_rowid();
```

4. Next, update the code for `GetAllMediumsAsync` to use Dapper:

```
private async Task<IList<Medium>>
GetAllMediumsAsync(SqliteConnection db)
{
    var mediums =
```

```
        await db.QueryAsync<Medium>(@"SELECT Id,
                                         Name,
                                         MediumType
                                         AS MediaType
                                         FROM Mediums");
        return mediums.ToList();
    }
```

We've gone from 14 lines of code to only two. Notice in the highlighted part of the query how we are using an alias of `MediaType` for the `MediumType` field. This is an easy way to map data to an object property that doesn't match the database field name, by simply renaming the field that's returned as part of the SQL `select` statement. Dapper has also helped us by directly returning a list of our `Medium` objects instead of us having to use a C# `while` loop to iterate over the result set.

5. Now, let's create a query that will get all the media items to populate the main `ListView` control. This query is a little more complex because we are joining two tables, `MediaItems` and `Mediums`, on `MediumId` and returning the data to be mapped to two corresponding objects, `item` and `medium`, as indicated by the first two generic types provided to the `QueryAsync` method. To perform this mapping, we must give Dapper a **lambda expression** that tells it to set `medium` as the `MediumInfo` property of `item` for each row that's returned from the query. The type of the returned object is defined by the third generic type provided to the `QueryAsync` method. The rest of the parameters will all be mapped automatically by Dapper, based on their property names:

```
private async Task<List<MediaItem>>
GetAllMediaItemsAsync(SqliteConnection db)
{
    var itemsResult = await db.QueryAsync<MediaItem,
        Medium, MediaItem>
        (
            @"
                SELECT
                    [MediaItems] . [Id] ,
                    [MediaItems] . [Name] ,
                    [MediaItems] . [ItemType] AS MediaType ,
                    [MediaItems] . [LocationType] AS
                    Location ,
                    [Mediums] . [Id] ,
                    [Mediums] . [Name] ,
```

```

[Mediums] . [MediumType] AS MediaType
FROM
[MediaItems]
JOIN
[Mediums]
ON
[Mediums] . [Id] = [MediaItems] .
[MediumId] ",
(item, medium) =>
{
    item.MediumInfo = medium;
    return item;
}
);
return itemsResult.ToList();
}

```

6. Next, we will create the `insert` and `update` methods for our media items:

```

private async Task<int>
InsertMediaItemAsync(SqliteConnection db, MediaItem item)
{
    var newIds = await db.QueryAsync<long>(
        @"INSERT INTO MediaItems
        (Name, ItemType, MediumId, LocationType)
        VALUES
        (@Name, @MediaType, @MediumId, @Location);
        SELECT last_insert_rowid()", item);
    (int)newIds.First();
}

private async Task UpdateMediaItemAsync(SqliteConnection
db, MediaItem item)
{
    await db.QueryAsync(
        @"UPDATE MediaItems
        SET Name = @Name,
        ItemType = @MediaType,
        MediumId = @MediumId,
        LocationType = @LocationType
        WHERE Id = @Id");
}

```

```
        LocationType = @Location
        WHERE Id = @Id;", item);
    }
```

The code in `InsertMediaItemAsync` should look familiar. It's very similar to what we did when we inserted it into the `Mediums` table. The code to update a row in `MediaItems` is only one (wrapped) line, thanks to Dapper.

7. There is one new read-only property that's been added to the `MediaItem` object in our model. This makes it easy to map `MediumId` to the `MediaItems` table:

```
public int MediumId => MediumInfo.Id;
```

8. Now, add the `Computed` attribute to the `MediaItem.MediumInfo` property. This tells Dapper to ignore the property when we're attempting to insert or update rows to the database. We only need to have `MediumId` saved. Users are unable to make changes to the rows in the `Mediums` table:

```
[Computed]
public Medium MediumInfo { get; set; }
```

9. Finally, let's create a method that will delete items from the `MediaItems` table. This code is a little different, thanks to `Dapper.Contrib`. We don't need to write any parameterized SQL in the code, because `Dapper.Contrib` has a `DeleteAsync` method that generates the code to delete from `MediaItems` based on the `Id` property of the `MediaItem` class provided:

```
private async Task DeleteMediaItemAsync(SqliteConnection
    db, int id)
{
    await db.DeleteAsync<MediaItem>(new MediaItem { Id =
        id });
}
```

To make this work, you must decorate the primary key properties of your model classes with `Key` attributes:

```
public class MediaItem
{
    [Key]
    public int Id { get; set; }

    ...
}
```

Make sure that every model class that uses one of the `Dapper.Contrib` attributes adds a `using` statement for `Dapper.Contrib.Extensions`.

Before we update all the public CRUD methods of the `SqliteDataService` class so that they call these private methods, let's complete the code that will initialize the service when the application launches.

## Updating the data service's initialization

Let's get started:

1. First, update the `PopulateMediums` method so that it's `async` and rename it `PopulateMediumsAsync`. This method will be updated so that it fetches the data from SQLite, and it will also create the necessary data if this is the first time the app has been launched for this user:

```
private async Task PopulateMediumsAsync(SqliteConnection db)
{
    _mediums = await GetAllMediumsAsync(db);
    if (_mediums.Count == 0)
    {
        var cd = new Medium { Id = 1, MediaType =
ItemType.Music, Name = "CD" };
        var vinyl = new Medium { Id = 2, MediaType =
ItemType.Music, Name = "Vinyl" };
        var hardcover = new Medium { Id = 3, MediaType =
ItemType.Book, Name = "Hardcover" };
        var paperback = new Medium { Id = 4, MediaType =
ItemType.Book, Name = "Paperback" };
        var dvd = new Medium { Id = 5, MediaType =
ItemType.Video, Name = "DVD" };
    }
}
```

```
    var bluRay = new Medium { Id = 6, MediaType =
Item.Type.Video, Name = "Blu Ray" };
    var mediums = new List<Medium>
    {
        cd,
        vinyl,
        hardcover,
        paperback,
        dvd,
        bluRay
    };
    foreach (var medium in mediums)
    {
        await InsertMediumAsync(db, medium);
    }
    _mediums = await GetAllMediumsAsync(db);
}
}
```

2. Second, remove `PopulateItems` from `SqliteDataService`, `DataService`, and `IDataService`. It will not be needed because we are not maintaining a collection of items in memory. You can also remove the `_items` private variable.
3. Now, take the code from the `SqliteDataService` constructor, move it to a new public method named `InitializeDataAsync`, and update the code so that it uses the new private initialization methods. Don't forget to remove the call to populate the items collection. The `SqliteConnection` object should always be part of a `using` block so that the connection is closed and the object is disposed of:

```
public async Task InitializeDataAsync()
{
    using (var db = await GetOpenConnectionAsync())
    {
        await CreateMediumTableAsync(db);
        await CreateMediaItemTableAsync(db);
        SelectedItemId = -1;
        PopulateItemTypes();
        await PopulateMediumsAsync(db);
    }
}
```

```
        PopulateLocationTypes() ;  
    }  
}
```

4. This new initialization method will need to be added to `IDataService` to make it available through our DI container:

```
public interface IDataService  
{  
    Task InitializeDataAsync();  
    ...  
}
```

5. After changing the location of the code to initialize `SqliteDataService`, the `OnLaunched` method in `App.xaml.cs` will need to be updated to use the new `SqliteDataService` and call `InitializeDataAsync`:

```
protected override async void OnLaunched(Microsoft.  
UI.Xaml.LaunchActivatedEventArgs e)  
{  
    Container = RegisterServices();  
    var dataService = Container.GetService<IDataService,  
    SqliteDataService>();  
    await dataService.InitializeDataAsync();  
    ...  
}
```

Don't forget to update `OnLaunched` so that it's `async`, enabling it to await the call to `dataService.InitializeDataAsync`.

Now that the app is initializing the data service when it launches, it's time to update the public CRUD methods so that they use the `async` private methods we created to fetch data from SQLite.

## Retrieving data via services

Let's start retrieving and saving SQLite data with our service methods. It will only be necessary to update the create, update, and delete pieces of CRUD. All the media items are stored in `List<MediaItem>` in `DataService`, so the public methods used to retrieve items can remain as they were in the previous chapter. Let's get started:

1. Start by updating the create, update, and delete methods for the media items. Each of these will get an open connection to the database from `GetOpenConnectionAsync` and call its corresponding private method asynchronously:

```
public async Task<int> AddItemAsync(MediaItem item)
{
    using (var db = await GetOpenConnectionAsync())
    {
        return await InsertMediaItemAsync(db, item);
    }
}
public async Task UpdateItemAsync(MediaItem item)
{
    using (var db = await GetOpenConnectionAsync())
    {
        await UpdateMediaItemAsync(db, item);
    }
}
public async Task DeleteItemAsync(MediaItem item)
{
    using (var db = await GetOpenConnectionAsync())
    {
        await DeleteMediaItemAsync(db, item.Id);
    }
}
```

2. Update the public methods to fetch items to be `async`:

```
public async Task<MediaItem> GetItemAsync(int id)
{
    IList<MediaItem> mediaItems;
```

```

        using (var db = await GetOpenConnectionAsync())
        {
            mediaItems = await GetAllMediaItemsAsync(db);
        }
        // Filter the list to get the item for our Id.
        return mediaItems.FirstOrDefault(i => i.Id == id);
    }
    public async Task<IList<MediaItem>> GetItemsAsync()
    {
        using (var db = await GetOpenConnectionAsync())
        {
            return await GetAllMediaItemsAsync(db);
        }
    }
}

```

**Note**

If you need to do a lot of filtering when querying data, Entity Framework is a more robust ORM that can provide more extensive options. SQLite is best suited for simpler applications. Note in the preceding code that `GetItemAsync` queries all the items and then filters to the item matching the provided `id` with a **lambda expression**.

3. The names have been updated to include `Async`, each use of the `_items` collection has been removed, and each method has been changed to return `Task`, so update the `IDataService` interface members to reflect the same change. It would be best to anticipate when methods may need to be `async` in the future to prevent making breaking changes to your interfaces:

```

Task<int> AddItemAsync(MediaItem item);
Task UpdateItemAsync(MediaItem item);
Task DeleteItemAsync(MediaItem item);
Task<IList<MediaItem>> GetItemsAsync();
Task<MediaItem> GetItemAsync(int id);

```

4. In `MainViewModel`, the `DeleteItem` method will be updated to use `async/await` with its data service call. Don't forget to rename it `DeleteItemAsync` to follow best practices when naming `async` methods. You will also need to add a `using` statement to the file for the `System.Threading.Tasks` namespace:

```
private async Task DeleteItemAsync()
{
    await _dataService.DeleteItemAsync(SelectedMediaItem);
    Items.Remove(SelectedMediaItem);
    allItems.Remove(SelectedMediaItem);
}
```

5. Update the `PopulateData` method so that it's `PopulateDataAsync` and use the `async` method to get items:

```
public async Task PopulateDataAsync()
{
    items.Clear();
    foreach(var item in await _dataService.
    GetItemsAsync())
    {
        items.Add(item);
    }
    allItems = new ObservableCollection<MediaItem>(Items);
    mediums = new TestObservableCollection<string>
    {
        AllMediums
    };
    foreach(var itemType in _dataService.GetItemTypes())
    {
        mediums.Add(itemType.ToString());
    }
    selectedMedium = Mediums[0];
}
```

6. Now, the code in the `MainViewModel` constructor that creates `RelayCommand` for our `DeleteCommand` must also be updated so that it can work with an `async` method. You will also have to update the `MainViewModel` constructor to call to `PopulateDataAsync` at the end of the constructor:

```
public MainViewModel(INavigationService
navigationService, IDataService dataService)
{
    _navigationService = navigationService;
    _dataService = dataService;
    DeleteCommand = new RelayCommand(async () => await
DeleteItemAsync(), CanDeleteItem);
    AddEditCommand = new RelayCommand(AddOrEditItem);
    PopulateDataAsync();
}
```

7. Some similar changes will be needed in `ItemDetailsViewModel`. Update the `SaveItem` method so that it's `async` and awaits the data service calls to `AddItemAsync`, `GetItemAsync`, and `UpdateItemAsync`. Don't forget to rename `SaveItem` to `SaveItemAsync` and add a `using` statement for the `System.Threading.Tasks` namespace:

```
private async Task SaveItemAsync()
{
    MediaItem item;
    if (_itemId > 0)
    {
        item = await _dataService.GetItemAsync(_itemId);
        item.Name = ItemName;
        item.Location = (LocationType)Enum.
Parse(typeof(LocationType), SelectedLocation);
        item.MediaType = (ItemType)Enum.
Parse(typeof(ItemType), SelectedItemType);
        item.MediumInfo = _dataService.
GetMedium(SelectedMedium);
        await _dataService.UpdateItemAsync(item);
    }
    else
    {
```

```
        item = new MediaItem
        {
            Name = ItemName,
            Location = (LocationType)Enum.
Parse(typeof(LocationType), SelectedLocation),
            MediaType = (ItemType)Enum.
Parse(typeof(ItemType), SelectedItemType),
            MediumInfo = _dataService.
GetMedium(SelectedMedium)
        };
        await _dataService.AddItemAsync(item);
    }
}
```

8. Next, update the `SaveItemAndReturn` and `SaveAndContinue` methods so that they also use `async/await`:

```
private async Task SaveItemAndReturnAsync()
{
    await SaveItemAsync();
    _navigationService.GoBack();
}

private async Task SaveItemAndContinueAsync()
{
    await SaveItemAsync();
    _dataService.SelectedItemId = 0;
    _itemId = 0;
    ItemName = "";
    SelectedMedium = null;
    SelectedLocation = null;
    SelectedItemType = null;
    IsDirty = false;
}
```

9. Finally, update `ItemDetailsViewModel.xaml` so that the save buttons use the `async` methods when they're binding their `Click` methods:

```
<Button Content="Save and Return"
Click="{x:Bind ViewModel.SaveItemAndReturnAsync}"
```

```

    IsEnabled="{x:Bind ViewModel.IsEnabled, Mode=OneWay}"
    Background="Transparent"/>
<Button Content="Save and Create New"
    Click="{x:Bind ViewModel.SaveItemAndContinueAsync}"
    IsEnabled="{x:Bind ViewModel.IsEnabled, Mode=OneWay}"
    Background="Transparent"/>

```

10. That's it. Run the application and see how it works. Since we are no longer creating any dummy data for the media items list, the collection will be empty when the app launches for the first time:

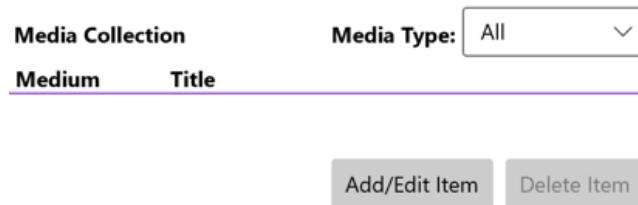


Figure 6.2 – Launching with a database for the first time

Try adding, updating, and removing some items. Then, close the application and run it again. You should see the same items on the list that were there when you closed it:

Medium	Title
Hardcover	War and Peace
DVD	The Mummy
CD	Classical Favorites
Vinyl	Smooth Jazz

Figure 6.3 – Relaunching with saved data

Users can now retain their saved data. If you would like to browse your SQLite data outside your app, there are tools you can use to connect to a local db and inspect it. One of them is **DB Browser for SQLite**. Covering this tool is beyond the scope of this book, but you can explore it for yourself at <https://sqlitebrowser.org/>. The next problem we will tackle is ensuring that the data your users are entering is valid. Let's look at how to implement data input validation in a WinUI project.

## Performing data validation with MVVM

Support for data validation is common in modern development frameworks. If an application or website doesn't alert users about which input fields are invalid, those users will file bug reports, leave bad reviews, or worse... stop using the app/website.

Web developers who use ASP.NET Core, Angular, and React have rich validation support at their disposal. UWP developers have never had built-in support for data validation, but that is changing with the move to WinUI 3.0. Simple validation can be added to our `ViewModel` properties through a little boilerplate code and a few attributes.

A little bit of code needs to be added to our `BindableBase` class, which is what all the `ViewModel` classes will inherit from. This will bring validation support to every `ViewModel`. The code for this can be found in the specifications for WinUI: <https://github.com/microsoft/microsoft-ui-xaml-specs/blob/user/lucashaines/inputvalidation/active/InputValidation/InputValidation.md>.

Here is a snippet of the code that was added to `BindableBase`. You can find the complete code for `BindableBase` on GitHub (<https://github.com/PacktPublishing/-Learn-WinUI-3.0/blob/master/Chapter06/Complete/MyMediaCollection/ViewModels/BindableBase.cs>). In this code, we are implementing `INotifyDataErrorInfo`, which is a standard interface for data validation. The `IValidatable` interface is our own new interface that we will define in a moment:

```
public class BindableBase : INotifyPropertyChanged,
INotifyDataErrorInfo, IValidatable
{
    ...
    public void Validate(string memberName, object value)
    {
        ClearErrors(memberName);
        List<ValidationResult> results = new
        List<ValidationResult>();
        bool result = Validator.TryValidateProperty(
            value,
            new ValidationContext(this, null, null)
        {
            MemberName = memberName
        },
    }
```

```
        results
    ) ;

    if (!result)
    {
        AddErrors(memberName, results);
    }
}
```

The `Validate` method is only one of the members that we've added to our base class. This method takes the property name and its value and performs some validation based on the attributes that we will be adding soon.

An `IValidatable` interface must also be added to the `Interfaces` folder in the project. This interface specifies the signature of the `Validate` method:

```
public interface IValidatable
{
    void Validate(string memberName, object value);
}
```

A XAML control must have `TwoWay` binding specified for validation to work correctly. Our `TextBox` Item Name on `ItemDetailsPage` is all ready to go. The last step is to add some data validation attributes to the `ItemName` property in `ItemDetailsViewModel`.

Let's add the `MinLength` and `MaxLength` attributes, which require that the Item Name is between 2 and 100 characters long. You can also provide a custom `ErrorMessage` for each validation attribute. It is recommended that you add some friendly messages here as the default message is not as intuitive for users:

```
[MinLength(2, ErrorMessage ="Item name must be at least 2
characters.")]
[MaxLength(100, ErrorMessage ="Item name must be 100 characters
or less.")]
public string ItemName
{
    get => _itemName;
    set
    {
```

```
        if (!SetProperty(ref _itemName, value,
nameof (ItemName)))
            return;
        IsDirty = true;
    }
}
```

You will also need to add a `using` statement for the `System.ComponentModel.DataAnnotations` namespace.

Now, run the app again and try to enter only a single character for the item name. The field will have a red border around it, with a red exclamation mark icon to the right. If you hover over the icon, it will provide a tooltip stating our customer `MinLength` `ErrorMessage` text:

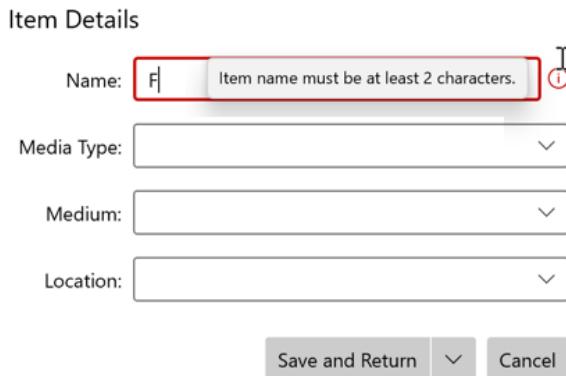


Figure 6.4 – Data input validation on the item name field

It is possible to customize how the validation information is presented to users, but that is beyond the scope of this book. I encourage you to read the WinUI specification linked previously, as it contains additional information about customizing `ValidationErrorMessage`.

## Summary

We have covered a lot of important material in this chapter. You learned how to read and write data to a local **SQLite** database. Then, you learned how to simplify your data access code by leveraging **Dapper**, an **ORM** for .NET developers. Using an **ORM** will save you time creating boilerplate mapping code in your data access layers for WinUI projects (or any other .NET projects). All this data access code was made `async` to keep the UI responsive for the user.

Finally, you saw how to ensure the data going into the application's database is valid by adding data validation to our `ViewModel` properties.

In the next chapter, we will learn how to create a beautiful **Fluent UI** with Microsoft's Fluent UI Design.

## Questions

1. When will Windows put a WinUI app into a suspended state?
2. When should you save application state to ensure it is not lost if Windows suspends the app?
3. What `ApplicationKind` will be passed to the `OnActivated` event handler if your app is launched from the command line?
4. What is a Micro ORM?
5. What is the name of the Dapper package that adds CRUD helpers such as `Delete` and `DeleteAsync`?
6. What is one of the powerful features of some more full-featured ORMs such as Entity Framework?
7. Which data validation attributes will validate string length?



# Section 2: Extending WinUI and Modernizing Applications

In this section, you build on what has been learned about WinUI application development, and expand on it with design concepts, platform options, and open source libraries. The Fluent Design system that is native to WinUI controls provides Windows application users with a familiar look and feel. You will learn how to bring this same design to desktop applications by building WinUI 3.0 applications on the .NET 5 platform. Finally, readers will explore the **Windows Community Toolkit** (WCT), a series of open source packages that expand on the controls available to WinUI developers. The XAML Islands controls in the WCT can even bring native WinUI controls to existing WPF and WinForms desktop applications.

This section comprises the following chapters:

- *Chapter 7, Fluent Design System for Windows Applications*
- *Chapter 8, Building WinUI Applications with .NET 5*
- *Chapter 9, Enhancing Applications with the Windows Community Toolkit*
- *Chapter 10, Modernizing Existing Win32 Applications with XAML Islands*



# 7

# Fluent Design System for Windows Applications

The **Fluent Design System** is a set of application design principles created by Microsoft and implemented across multiple desktop, mobile, and web platforms. The Fluent Design System for Windows is the set of controls, patterns, and styles for applications built for Windows 10 and Windows itself. In fact, it is the implicit styling for all WinUI controls.

It is important to learn the tenets of Fluent Design and how to implement these in your WinUI applications. We will also explore the **Fluent XAML Theme Editor** application for Windows. This application assists developers in creating a theme for their applications, including color schemes and style elements such as borders and corners. Developers can then easily import the resources to implement the theme.

In this chapter, we will cover the following topics:

- Learning the concepts of Fluent Design
- How to find the latest information about Fluent Design
- Incorporating Fluent Design concepts into WinUI applications
- Using the Fluent XAML Theme Editor to customize and use a UI theme

- Using the UWP resources gallery to explore built-in resources available to WinUI applications
- Exploring the Fluent Design toolkits for design applications such as Figma, Sketch, Adobe XD, Photoshop, and Illustrator

By the end of this chapter, you will understand the Fluent Design System for Windows applications. You will also know how to incorporate these design standards into your WinUI applications.

## Technical requirements

To follow along with the examples in this chapter, the following software is required:

- Windows 10 version 1803 (version 17134) or newer
- Visual Studio 2019 version 16.9 or newer with the following workloads: .NET Desktop Development, Universal Windows Platform Development

The source code for this chapter is available on GitHub at this URL: <https://github.com/PacktPublishing/-Learn-WinUI-3.0/tree/master/Chapter07>.

## What is the Fluent Design System?

The Fluent Design System is a cross-platform system to guide developers in creating beautiful, intuitive applications. The website for Fluent Design (<https://www.microsoft.com/design/fluent/>) has dedicated pages with resources for developers on many platforms:

- Android
- iOS
- macOS
- Web
- Windows
- Cross-platform (React Native)

Fluent Design aims to be simple and intuitive. While it maintains its design philosophy across platforms, it also adapts aspects of its design to feel native on every platform. In *Chapter 1, Introduction to WinUI*, we discussed the origins of some of the current Fluent Design concepts in the **Metro** design that was introduced with Windows Phone. While the look and feel of Microsoft's designs have evolved over the years, some of the principles remain. The three core principles of the Fluent Design System are the following:

- **Natural on every device:** Software should adapt to the device where it's running, whether it's a PC, tablet, game console, or AR headset.
- **Intuitive and powerful:** The UI anticipates users' actions and pulls them into the experience while using the app.
- **Engaging and immersive:** The design pulls from real-world elements, using light, shadow, texture, and depth.

The driving philosophy behind the design is to adapt and feel natural. The device and the app should feel comfortable and anticipate the user's actions.

This is a very abstract and high-level explanation, so far. Let's explore the specifics of Fluent Design for Windows in the next section.

## Exploring Fluent Design for Windows

For Windows applications, Fluent Design encompasses several areas. When compared to other design systems, Fluent is more universal. Apple's **Human Interface Guidelines** (<https://developer.apple.com/design/human-interface-guidelines/>) have only been widely adopted on iOS and macOS. Google's **Material Design** (<https://material.io/>) system has seen wider adoption but only has toolkits available for Android, Flutter, and the web.

Fluent Design is most often compared with Material Design, as they share some concepts when it comes to shapes and texture, but Fluent Design uses transparency to much greater effect than Material Design. Choosing Fluent Design provides a rich toolset that you can use in WinUI and any other development platforms.

Let's explore some of these design aspects and how they apply to your WinUI applications.

## Controls

A control equates to a single element of *user input* or *interaction*. We have already explored many of the controls available in WinUI in *Chapter 5, Exploring WinUI Controls and Libraries*. This is what some of the common WinUI controls look like in light and dark modes with the default accent color in Windows 10:

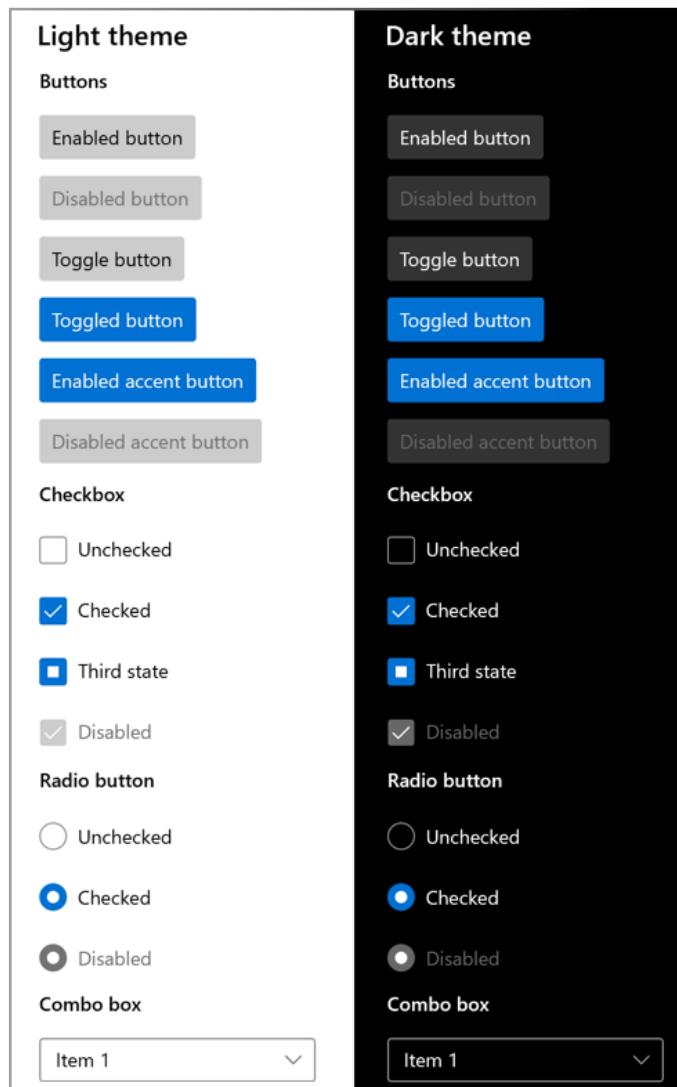


Figure 7.1 – Some common controls in light and dark modes

By default, the WinUI controls make use of Fluent styles. We will see how to override the default Fluent styles in our WinUI controls later in the chapter.

## Patterns

Patterns are groups of related controls or a group of controls that becomes a single new element. This group could be added to a composite control for re-use. Some examples of patterns in WinUI are the following:

- **Search:** In its simplest form, a search pattern needs to have controls for accepting input, invoking the search, and displaying the search results. An additional element may be added for suggesting searches before any input is received. Assistants such as *Microsoft's Cortana* do this based on a user's calendar, contacts, news preferences, and so on. Adding an autosuggest list based on user input is a common feature of modern search controls. You could also integrate chat controls with some **artificial intelligence (AI)** such as Microsoft's **Bot Framework** to ask some follow-up questions based on the initial search parameters (Microsoft Docs example: <https://docs.microsoft.com/en-us/windows/uwp/design/controls-and-patterns/search>).
- **Forms:** Forms are a very common control pattern. They consist of groups of related labels, input controls, and command buttons that collect a related set of data elements. Some common forms with potential for re-use are user account creation forms and forms for collecting user feedback. Forms should follow the Fluent Design guidelines for spacing, flexible layout, and using typography for creating a hierarchy (Microsoft Docs example: <https://docs.microsoft.com/en-us/windows/uwp/design/controls-and-patterns/forms>).
- **List/details:** The list/details control pattern is seen so frequently that we probably don't notice it most of the time. You may think our My Media Collection application follows this pattern (with the items list and a page to view and edit item details), but this pattern describes how to show these views on the same page. This is typically achieved with a **ListView** control and a **SplitView** control to separate the list from the selected item's details. Based on the available width on the page, the two views can either be stacked vertically or displayed side by side (Microsoft Docs example: <https://docs.microsoft.com/en-us/windows/uwp/design/controls-and-patterns/list-details>).

Each of these patterns encapsulates the elements of Fluent Design to create a composite control that can be reused across applications. You may have control patterns in your projects that could be added to a shared control library for ease of reuse. Shared libraries like these can save time and ensure that good design practices are followed across teams.

## Layout

The layout design is important to ensure that an application adapts to any screen size or orientation. Flexibility is a key tenet of a well-designed layout. When a page is resized, the contents can adapt by repositioning controls, adding/removing items, changing the flow of items, replacing controls with others that better fit the current available space, or simply resizing items. This is typically handled in XAML with **Visual States**. You can define a different **VisualState** for each size threshold to which your page must adapt, possibly defined as **Narrow**, **Default**, **Wide**, and **ExtraWide**. Each **VisualState** updates control properties to adapt to the new layout. Microsoft Docs has a great example of this at <https://docs.microsoft.com/en-us/windows/uwp/design/layout/layouts-with-xaml>. WinUI includes several different layout panels that can help developers create the right layout for their pages and respond to changes in size, orientation, and resolution.

## Input

There are Fluent recommendations for responding to user input. There are guidelines for reacting to the traditional mouse and keyboard input that developers have been handling for decades. Modern applications can do things such as pan, zoom, rotate, or scroll based on mouse input. A keyboard may be a physical keyboard or an onscreen keyboard for mobile and touch users.

Input can come in other forms with today's hardware:

- Pen/stylus
- Touch
- Touchpad
- Gamepad/controller
- Remote control
- Surface Dial (see <https://docs.microsoft.com/en-us/windows/uwp/design/input/windows-wheel-interactions>)
- HoloLens
- Voice

User input can also be simulated with the input injection APIs. An example of where this might be useful is creating a **Show Me How** or **Guided Tour** feature in your app. Your code can execute some pre-defined steps, guiding the user through performing some action on the page. This API is beyond the scope of this book. To read about an example of using input injection to intercept mouse input and turn it into touch input, read this article on Microsoft Docs: <https://docs.microsoft.com/en-us/windows/uwp/design/input/input-injection>.

## Style

Style encompasses multiple aspects of Fluent Design:

- **Icons:** Good icons should be simple and convey the application's purpose.
- **Color:** The color choice is important. Allowing users to customize their colors is also a great way to make your app feel personal to them. WinUI makes it easy to adapt the user's light or dark theme choice and highlight color with theme brushes.
- **Typography:** Microsoft recommends that Windows applications all use the **Segoe UI** font. Selecting the font size can help convey a hierarchy within the app, like a book or document layout. To this end, Microsoft defined a **type ramp** (available at <https://docs.microsoft.com/en-us/windows/uwp/design/style/typography#type-ramp>), which has static resources that can be leveraged in WinUI to select the right size for a control's intended use.
- **Spacing:** Spacing between and within controls is important for readability and usability. WinUI controls allow a **Standard** or **Compact** density to be selected. More information about sizing and Fluent densities can be found here: <https://docs.microsoft.com/en-us/windows/uwp/design/style/spacing>.
- **Reveal highlight:** This style element uses illumination to draw focus to important parts of the user interface. An example of this is illuminating a button or checkbox as the mouse pointer moves across it.
- **Reveal focus:** Drawing attention to focusable elements is important for larger displays, such as an **Xbox** or **Surface Hub**. This is achieved through lighting effects with Fluent.
- **Acrylic:** This is a type of WinUI brush that creates texture with transparency. This texture gives a feeling of depth to the user interface.
- **Corner radius:** Fluent Design promotes the idea that rounded corners promote positive feelings within users. WinUI controls have a rounded corner radius consistent with Fluent Design recommendations.

These are just some of the aspects of style defined by Fluent Design. You can read more about them at Microsoft Docs: <https://docs.microsoft.com/en-us/windows/uwp/design/style/>.

**Note**

In 2021, Microsoft is revamping its UI styles in an effort codenamed *Sun Valley*. The new styles have a softer appearance with rounded corners. WinUI controls will be picking up these styles, and in fact, they are already available in WinUI 2.6. To read more about the new visual styles coming to WinUI, check out this blog post by Martin Zikmund: <https://blog.mzikmund.com/2021/02/enabling-the-sun-valley-visual-styles-in-windows-ui-2-6-preview/>.

Many of the aspects of Fluent style are made available to our WinUI apps via XAML styles and other static resources. Next, we will look at how we can update our sample application to respond to changes in a user's Windows theme.

## Incorporating Fluent Design in WinUI applications

It is time to incorporate a few of the Fluent Design principles into the application and polish the UI a little. Most of the WinUI controls are already designed to meet Fluent standards, but there were a few attributes we added without understanding Fluent Design.

### Updating the title bar

Before we even get into the XAML to improve the styles, let's fix the application's title bar. Until now, the title bar always read **MyMediaCollection** without any spaces or indication of the current page:

1. First, to fix the spacing, open `Package.appmanifest` from the **Solution Explorer** window. On the page that opens, update **Display name** to `My Media Collection`. If you like, you can also change **Description**.

Application	Visual Assets	Capabilities	Declarations	Content URIs	Packaging
-------------	---------------	--------------	--------------	--------------	-----------

Use this page to set the properties that identify and describe your app.

Display name:	My Media Collection
Entry point:	MyMediaCollection.App
Default language:	en-US <a href="#">More information</a>
Description:	My Media Collection

Supported rotations: An optional setting that indicates the app's orientation preferences.



Lock screen notifications: (not set)

Resource group:

**Tile Update:**

Updates the app tile by periodically polling a URI. The URI template can contain "(language)" and "(region)" tokens that will be replaced at runtime to generate the URI to poll.

[More information](#)

Recurrence: (not set)

URI Template:

Figure 7.2 – Updating information in Package.appmanifest

Updating **Display name** will change the application's title bar text.

2. Optionally, you can also update the `AssemblyTitle`, `AssemblyDescription`, and `AssemblyProduct` attributes in **Properties |AssemblyInfo.cs**. However, they will not impact the title bar. For .NET assemblies, this information appears in the file properties in **Windows Explorer**:

```
[assembly: AssemblyTitle("My Media Collection")]
[assembly: AssemblyDescription("Catalog and track your
music, movies and books!")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("")]
[assembly: AssemblyProduct("My Media Collection")]
[assembly: AssemblyCopyright("Copyright © 2020")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]
```

3. Next, in `MainPage.xaml.cs`, add an event handler for the `Loaded` event of the page. In the event handler, add some code to set the `Title` of the current window to `Home`. This will prepend the specified text to the title bar, followed by a dash. When we launch the application, the title bar should read "Home - My Media Collection":

```
public MainPage()
{
    this.InitializeComponent();
    Loaded += MainPage_Loaded;
}

private void MainPage_Loaded(object sender,
    RoutedEventArgs e)
{
    Window.Current.Title = "Home";
}
```

4. Finally, make the same changes in `ItemDetailsView.xaml.cs`, but set the window's title to `Item Details`.

Now, when you run the application, you should see the title bar text update as you navigate between the list of items and the item details. Let's make some changes to the styles of `MainPage` next.

## Changing the style of MainPage

Currently, the main page of our application doesn't have many styles. We set the `FontWeight` of a few `TextBlock` controls to `Bold` to set them apart as important items, but this doesn't follow the Fluent Design guidelines for typography. There is also a purple border separating the `ListView` header from its items:



Figure 7.3 – The current main page of MyMediaCollection

Hardcoding colors is not a good practice. Even if you were using custom colors as product branding, you could centralize those in `Application.Resources`:

1. Let's work our way down the XAML file and make some improvements.

First, update the text of the first `TextBlock` from `Media Collection` to `Home`, matching the text in the window's title bar. Wrap it in a horizontally aligned `StackPanel` and add a preceding `SymbolIcon` control to display a `Home` symbol. Finally, remove the hardcoded font size and weight attributes and set the `Style` attribute to import the `SubheaderTextBlockStyle` `StaticResource`. Those changes should look like this:

```
<StackPanel Orientation="Horizontal">
  <SymbolIcon Symbol="Home" Margin="8"/>
  <TextBlock Text="Home"
    Style="{StaticResource
    SubheaderTextBlockStyle}"
    Margin="8"/>
</StackPanel>
```

2. We should also remove the `FontWeight` attribute from the `Media Type` label and use a Fluent style resource:

```
<TextBlock Text="Media Type:" Margin="4"
  Style="{StaticResource
  SubtitleTextBlockStyle}"
  VerticalAlignment="Bottom"/>
```

3. Next, change the surrounding `Grid` to a `StackPanel` and remove the `Grid.ColumnDefinition` definitions. In addition to simplifying the layout, this will set the `Home` symbol and text to sit above the rest of the controls on the page, reinforcing the hierarchy. The full block of code will look like this:

```
<StackPanel>
  <StackPanel Orientation="Horizontal">
    <SymbolIcon Symbol="Home" Margin="8"/>
    <TextBlock Text="Home"
      Style="{StaticResource
      SubheaderTextBlockStyle}"
      Margin="8"/>
  </StackPanel>
```

```
<StackPanel Orientation="Horizontal"
            HorizontalAlignment="Right">
    <TextBlock Text="Media Type:"
               Margin="4"
               Style="{StaticResource
SubTitleTextBlockStyle}"
               VerticalAlignment="Bottom"/>
    <ComboBox ItemsSource="{x:Bind ViewModel.
Mediums}"
               SelectedItem="{x:Bind ViewModel.
SelectedMedium, Mode=TwoWay}"
               MinWidth="120"
               Margin="0,2,6,4"
               VerticalAlignment="Bottom"/>
</StackPanel>
</StackPanel>
```

4. Next, update HeaderTemplate of ListView to replace the purple BorderBrush attributes with SystemAccentColor from ThemeResource. This will make sure that the border's color picks up the user's preferred accent color from their selected Windows theme. Also, change each TextBlock to use a built-in Style instead of setting FontWeight:

```
<ListView.HeaderTemplate>
    <DataTemplate>
        <Grid Margin="4,0,4,0">
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="100"/>
                <ColumnDefinition Width="*"/>
            </Grid.ColumnDefinitions>
            <Border BorderBrush="{ThemeResource
SystemAccentColor}"
                    BorderThickness="0,0,0,1">
                <TextBlock Text="Medium"
                           Margin="4,0,0,0"
                           Style="{StaticResource
TitleTextBlockStyle}"/>
            </Border>
```

```
<Border Grid.Column="1"
        BorderBrush="{ThemeResource
SystemAccentColor}"
        BorderThickness="0,0,0,1">
    <TextBlock Text="Title"
        Margin="4,0,0,0"
        Style="{StaticResource
TitleTextBlockStyle}"/>
</Border>
</Grid>
</DataTemplate>
</ListView.HeaderTemplate>
```

5. Finally, let's define the end of the list area by adding a border between the bottom of ListView and the command buttons. Do this by wrapping the buttons' StackPanel with a Border control, again using SystemAccentColor. Margin= "4,0" is shorthand that is equivalent to Margin= "4,0,4,0":

```
<Border Grid.Row="2"
        BorderBrush="{ThemeResource SystemAccentColor}"
        BorderThickness="0,1,0,0"
        Margin="4,0">
    <StackPanel Orientation="Horizontal"
        HorizontalAlignment="Right">
        <Button Command="{x:Bind ViewModel.
AddEditCommand}">
            Content="Add/Edit Item"
            Margin="8,8,0,8"/>
        <Button Command="{x:Bind ViewModel.
DeleteCommand}">
            Content="Delete Item"
            Grid.Column="1"
            Margin="8"/>
    </StackPanel>
</Border>
```

6. Run the application and check out the restyled user interface. It looks much better. You can now easily see the hierarchy of data, however limited it may be in our simple application:

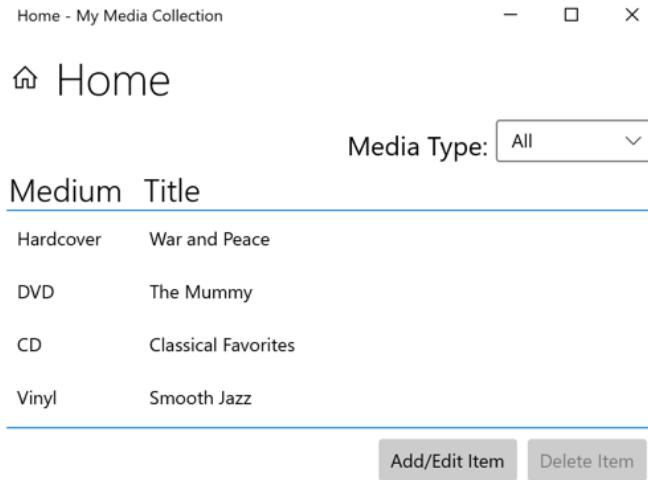


Figure 7.4 – The newly styled My Media Collection home page

Before moving on to the details page, let's see how the page looks if we select the dark mode in Windows. Open **Windows Settings**, go to **Personalization | Colors**, and select **Dark** from the **Choose your color** dropdown (if you normally use **Dark**, try changing it to **Light**):

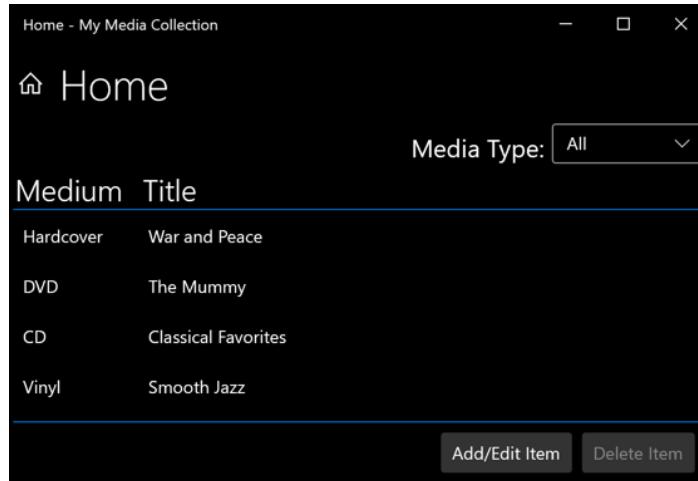


Figure 7.5 – My Media Collection running in dark mode

Everything on the page switches to dark mode without any code changes. Great!

If you have a good reason to keep your application in light or dark mode, you can update `Application.xaml` to add a single attribute to the `Application` element using this command:

```
RequestedTheme="Dark"
```

This will apply the `Dark` theme to the entire application. If you have a reason to only force this theme on part of the application, the `RequestedTheme` attribute can be applied to a `Page` or `Control`. Now, let's apply the same types of styles to the details page.

## Changing the style of `ItemDetailsPage`

We want to update `ItemDetailsPage.xaml` to make sure it has the same overall look and feel as the main page:

1. Open the file and start by updating `Item Details`. Give it the same `SubheaderTextBlockStyle` used on `Home` and wrap it in a horizontally aligned `StackPanel`. Precede `TextBlock` with `SymbolIcon`, which uses the `Edit` symbol:

```
<StackPanel Orientation="Horizontal">
    <SymbolIcon Symbol="Edit" Margin="8"/>
    <TextBlock Text="Item Details"
        Style="{StaticResource
SubheaderTextBlockStyle}"
        Margin="8"/>
</StackPanel>
```

2. Next, modify `Grid` following the new `StackPanel` to have top and bottom borders. Also, modify `Margin` to have 4px on either side of `Grid`:

```
<Grid Grid.Row="1"
    BorderBrush="{ThemeResource SystemAccentColor}"
    BorderThickness="0,1,0,1"
    Margin="4,0,4,8">
```

That's all we need to change on this page. Run the application again and navigate to the details page to see how it looks:

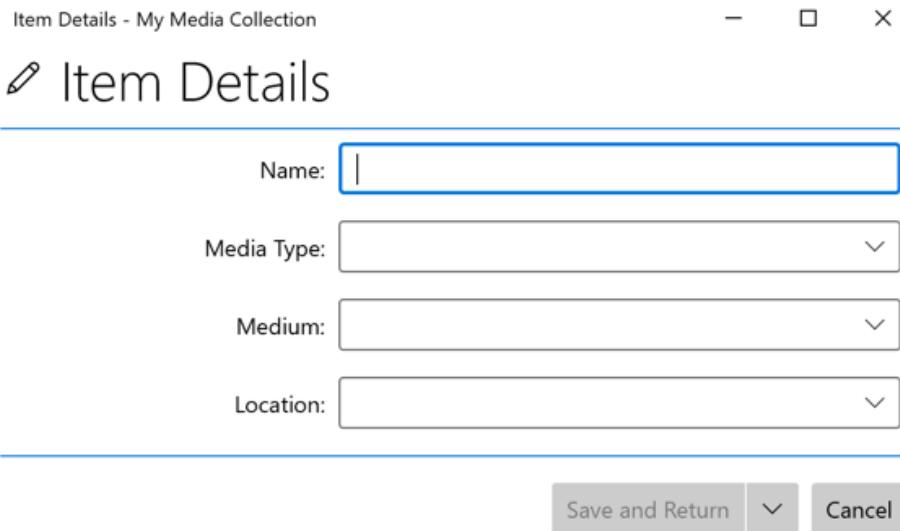


Figure 7.6 – The restyled Item Details page

This looks good. Now the styles of the pages match, and the added border lines match the color of the highlighted active input field.

Let's now shift gears and review a few tools that can help designers and developers when implementing Fluent Design.

## Using the Fluent XAML Theme Editor

We've seen how easy it is to pick up the default color and theme resources from the user's Windows settings, but what if you or your company wants to create a custom theme for an application. Maybe that theme needs to be shared across a suite of applications. You could create a XAML file with `ResourceDictionary` in Visual Studio and code all the markup by hand for a new style. Visual Studio's **IntelliSense** will help in some regards. However, there is an easier way.

Microsoft has created an open source tool called the **Fluent XAML Theme Editor**. It is available on the Windows Store at <https://www.microsoft.com/en-us/p/fluent-xaml-theme-editor/9n2xd3q8x57c>, and the source is available on GitHub here: <https://github.com/Microsoft/fluent-xaml-theme-editor>. This application provides an easy-to-use visual designer to create a `ResourceDictionary` XAML that you can drop into your projects.

**Note**

The Fluent XAML Theme Editor adjusts the styles for the built-in UWP controls, but these same styles will work with our WinUI controls as well.

To install the application, open the **Microsoft Store** app, search for `fluent xaml` in the search field, and you will find **Fluent XAML Theme Editor** in the search results. Click it in the search results to view the product page:

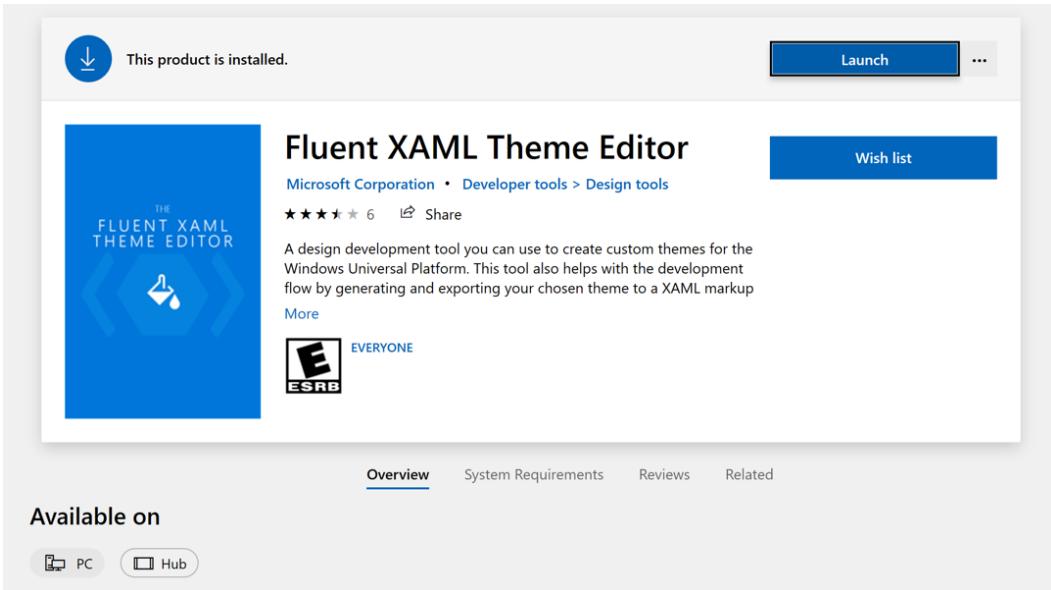


Figure 7.7 – The Fluent XAML Theme Editor page in the Microsoft Store

If you have the app installed, there will be a **Launch** button. If it is not installed, you can click the **Install** button. When it completes its installation, you will find the app in your **Start Menu**.

When you first launch the app, it will launch with the default style, displayed in both light and dark themes. On the right-hand panel, you will find controls for changing the colors, shapes, and typography (coming soon):

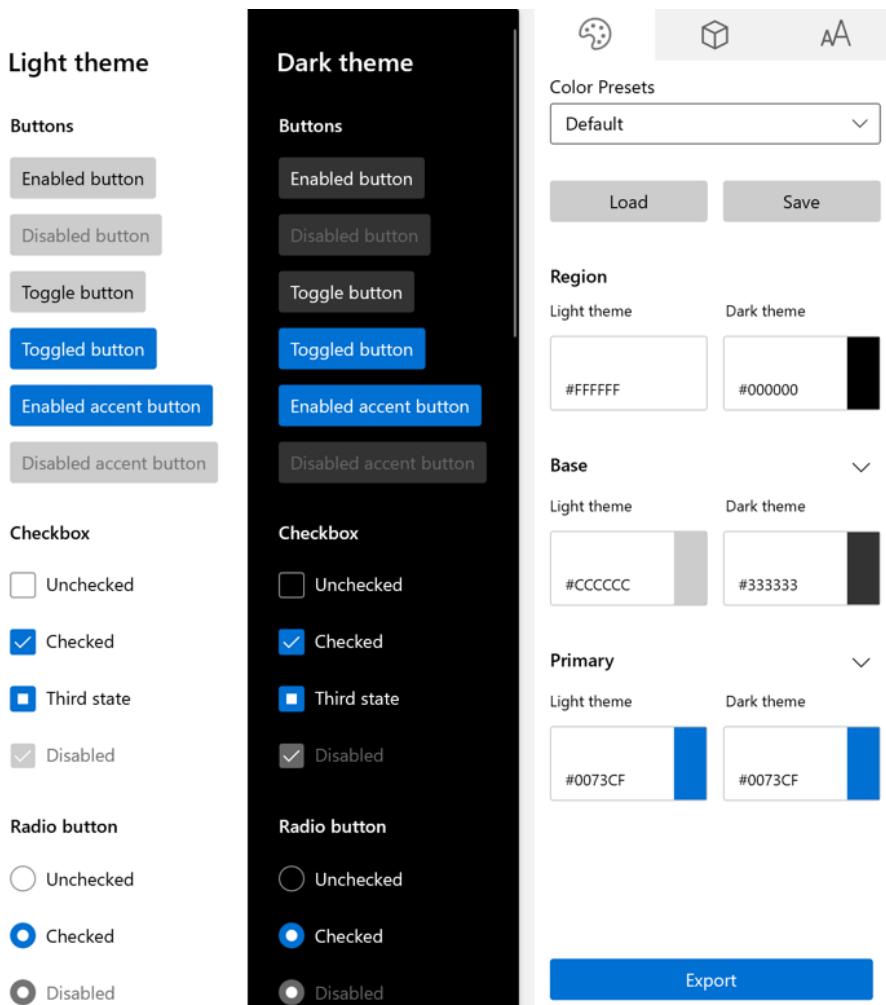


Figure 7.8 – The Fluent XAML Theme Editor for Windows

## Colors

On the **Colors** tab, you can select one of the default profiles from the **Color Presets** dropdown. In addition to the default preset, there is **Lavender**, **Forest**, and **Nighttime**. There are also options to load additional presets or save your current color settings to a new preset. These color presets are saved in JSON format.

### Note

Any colors you specify here will override the Windows system accent color that will get picked up by WinUI applications by default. Unless your application has a good reason to follow another theme, it is usually best to let WinUI use the user's chosen accent colors. Designing a custom theme should be undertaken by an experienced design team.

Clicking on any of the colors in the current preset will launch a color picker window where you can adjust the current color:

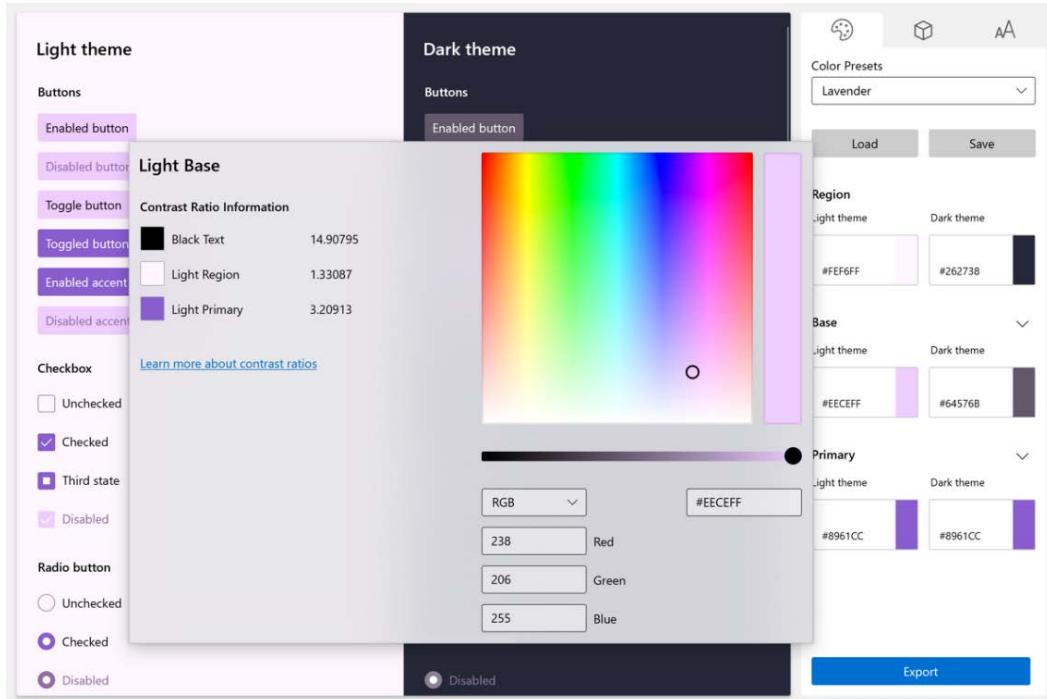


Figure 7.9 – Adjusting a preset color with a color picker

The **Region**, **Base**, and **Primary** colors can be adjusted for light and dark themes independently.

## Shapes

The **Shapes** panel provides controls to adjust the **Corner Radius** of the **Controls** and the **Overlay**, as well as the default **Border Thickness** for the theme.

Shape presets can also be saved and loaded. The app comes with two presets: **Default** and **No Rounding, Thicker Borders**. The difference is subtle but noticeable:

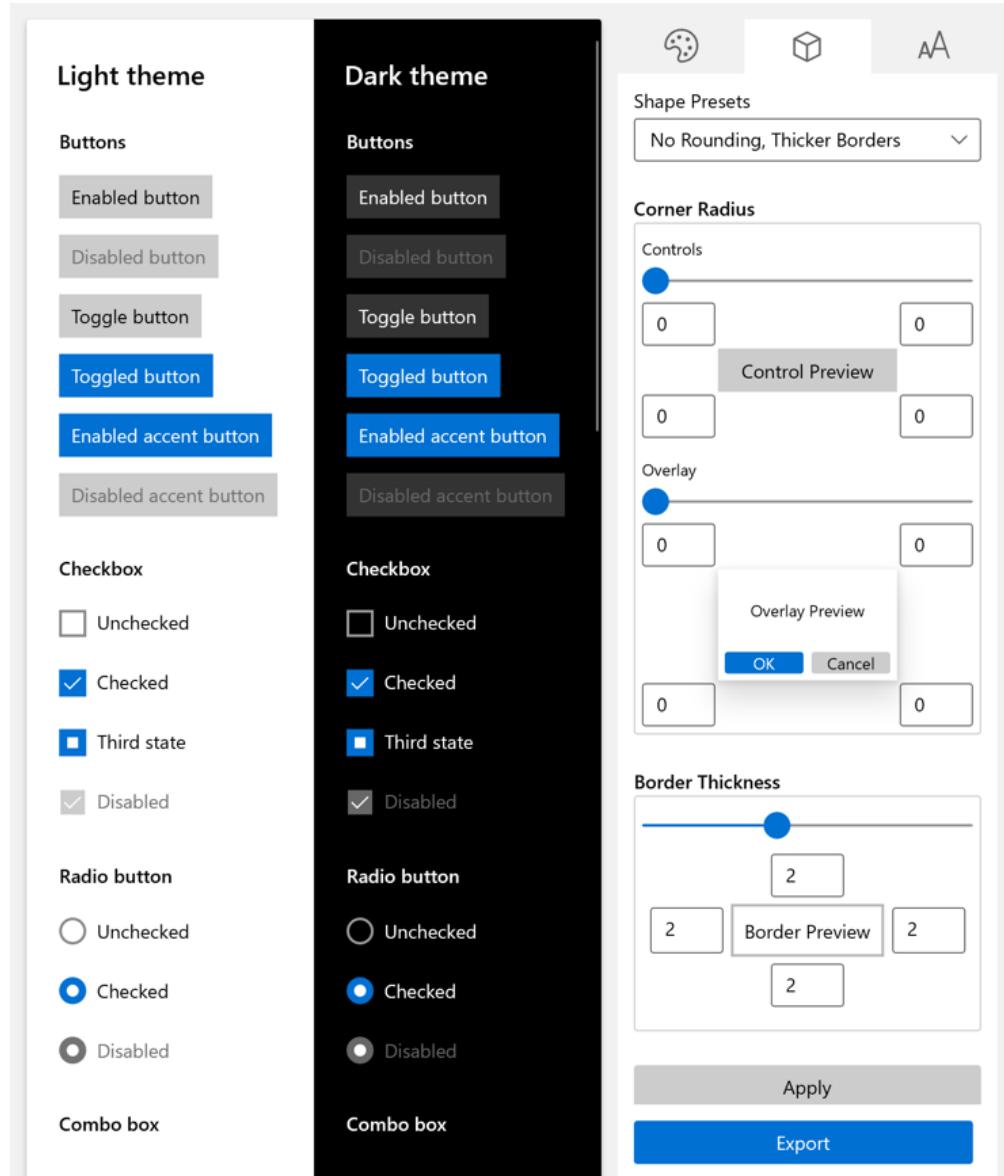


Figure 7.10 – Shapes with No Rounding, Thicker Borders applied

When you are done adjusting the color and shape settings, use the **Export** button to open a new window containing a `ResourceDictionary` with your theme data. You can copy the XAML and paste it into a `Resources` section in your project:

```

<!-- Free Public License 1.0.0 Permission to use, copy, modify, and/or distribute this code for any purpose with or without fee is hereby granted.
<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:Windows10version1809="http://schemas.microsoft.com/winfx/2006/xaml/presentation?IsApiContractPresent(Windows.Foundation.
    xmlns:BelowWindows10version1809="http://schemas.microsoft.com/winfx/2006/xaml/presentation?IsApiContractNotPresent(Windows.
<ResourceDictionary.ThemeDictionaries>
    <ResourceDictionary x:Key="Default">
        <ResourceDictionary.MergedDictionaries>
            <Windows10version1809:ColorPaletteResources Accent="#FFCC4D11" AltHigh="#FF000000" AltLow="#FF000000" AltMedium="#FF000000" Alt
            <ResourceDictionary>
                <ResourceDictionary x:Key="Default">
                    <Windows10version1809:Color x:Key="SystemAccentColor">#FFCC4D11</BelowWindows10version1809:Color>
                    <BelowWindows10version1809:Color x:Key="SystemAltHighColor">#FF000000</BelowWindows10version1809:Color>
                    <BelowWindows10version1809:Color x:Key="SystemAltLowColor">#F0000000</BelowWindows10version1809:Color>
                    <BelowWindows10version1809:Color x:Key="SystemAltMediumColor">#FF000000</BelowWindows10version1809:Color>
                    <BelowWindows10version1809:Color x:Key="SystemAltMediumHighColor">#F0000000</BelowWindows10version1809:Color>
                    <BelowWindows10version1809:Color x:Key="SystemAltMediumLowColor">#F0000000</BelowWindows10version1809:Color>
                    <BelowWindows10version1809:Color x:Key="SystemBaseHighColor">#FFFFFFFF</BelowWindows10version1809:Color>
                    <BelowWindows10version1809:Color x:Key="SystemBaseLowColor">#FF2F7BAD</BelowWindows10version1809:Color>
                    <BelowWindows10version1809:Color x:Key="SystemBaseMediumColor">#FF80BDFD</BelowWindows10version1809:Color>
                    <BelowWindows10version1809:Color x:Key="SystemBaseMediumHighColor">#FA5DEC</BelowWindows10version1809:Color>
                    <BelowWindows10version1809:Color x:Key="SystemBaseMediumLowColor">#FF5E9DC6</BelowWindows10version1809:Color>
                    <BelowWindows10version1809:Color x:Key="SystemChromeAltLowColor">#FA5DEC</BelowWindows10version1809:Color>
                    <BelowWindows10version1809:Color x:Key="SystemChromeBlackHighColor">#F0000000</BelowWindows10version1809:Color>
                    <BelowWindows10version1809:Color x:Key="SystemChromeBlackLowColor">#FA5DEC</BelowWindows10version1809:Color>
                    <BelowWindows10version1809:Color x:Key="SystemChromeBlackMediumColor">#F0000000</BelowWindows10version1809:Color>
                    <BelowWindows10version1809:Color x:Key="SystemChromeBlackMediumLowColor">#F0000000</BelowWindows10version1809:Color>
                    <BelowWindows10version1809:Color x:Key="SystemChromeDisabledHighColor">#F21F7BAD</BelowWindows10version1809:Color>
                    <BelowWindows10version1809:Color x:Key="SystemChromeDisabledLowColor">#FF8DBFDF</BelowWindows10version1809:Color>
                    <BelowWindows10version1809:Color x:Key="SystemChromeGrayColor">#F76AED3</BelowWindows10version1809:Color>
                    <BelowWindows10version1809:Color x:Key="SystemChromeHighColor">#F76AED3</BelowWindows10version1809:Color>
                    <BelowWindows10version1809:Color x:Key="SystemChromeLowColor">#FF993B73</BelowWindows10version1809:Color>
                    <BelowWindows10version1809:Color x:Key="SystemChromeMediumColor">#F134B82</BelowWindows10version1809:Color>
                    <BelowWindows10version1809:Color x:Key="SystemChromeMediumLowColor">#FF26689F</BelowWindows10version1809:Color>
                    <BelowWindows10version1809:Color x:Key="SystemChromeWhiteColor">#FFFFFFFF</BelowWindows10version1809:Color>
                    <BelowWindows10version1809:Color x:Key="SystemListLowColor">#F134B82</BelowWindows10version1809:Color>
                    <BelowWindows10version1809:Color x:Key="SystemListMediumColor">#FF2F7BAD</BelowWindows10version1809:Color>
                    <Color x:Key="SystemChromeAltMediumHighColor">#CC0D2644</Color>
                    <Color x:Key="SystemChromeAltHighColor">#FF2F7BAD</Color>
                    <Color x:Key="SystemRevealListLowColor">#F134B82</Color>
                </ResourceDictionary>
            </ResourceDictionary.MergedDictionaries>
        </ResourceDictionary>
    </ResourceDictionary.ThemeDictionaries>

```

[Learn more about XAML ResourceDictionary](#)

[Copy to Clipboard](#)

Figure 7.11 – Exporting a theme from the Fluent XAML Theme Editor

Next, let's look at a great way to find and discover resources you can use in your WinUI app.

## Using the UWP Resources Gallery

Another great tool to help WinUI developers while they are styling their application is the **UWP Resources Gallery**. While this tool does have UWP and not WinUI in its name, all the resources you can discover in the tool also exist in WinUI 3.0. This app is open source and is available on the Windows Store at <https://www.microsoft.com/en-us/p/uwp-resources-gallery/9pj1433vx9r>. A WinUI developer named **Marcel Wagner** maintains the open source project on GitHub at <https://github.com/chingucoding/UWP-Resources-Gallery>.

After installing the app, you will see that there are several types of resources you can browse or search:

- **Icons:** In this section, you can search for **Segoe MDL2** icons such as the **Home** and **Edit** symbols we used in our application. Select an icon from the list to display details about how it can be used in XAML or C#:

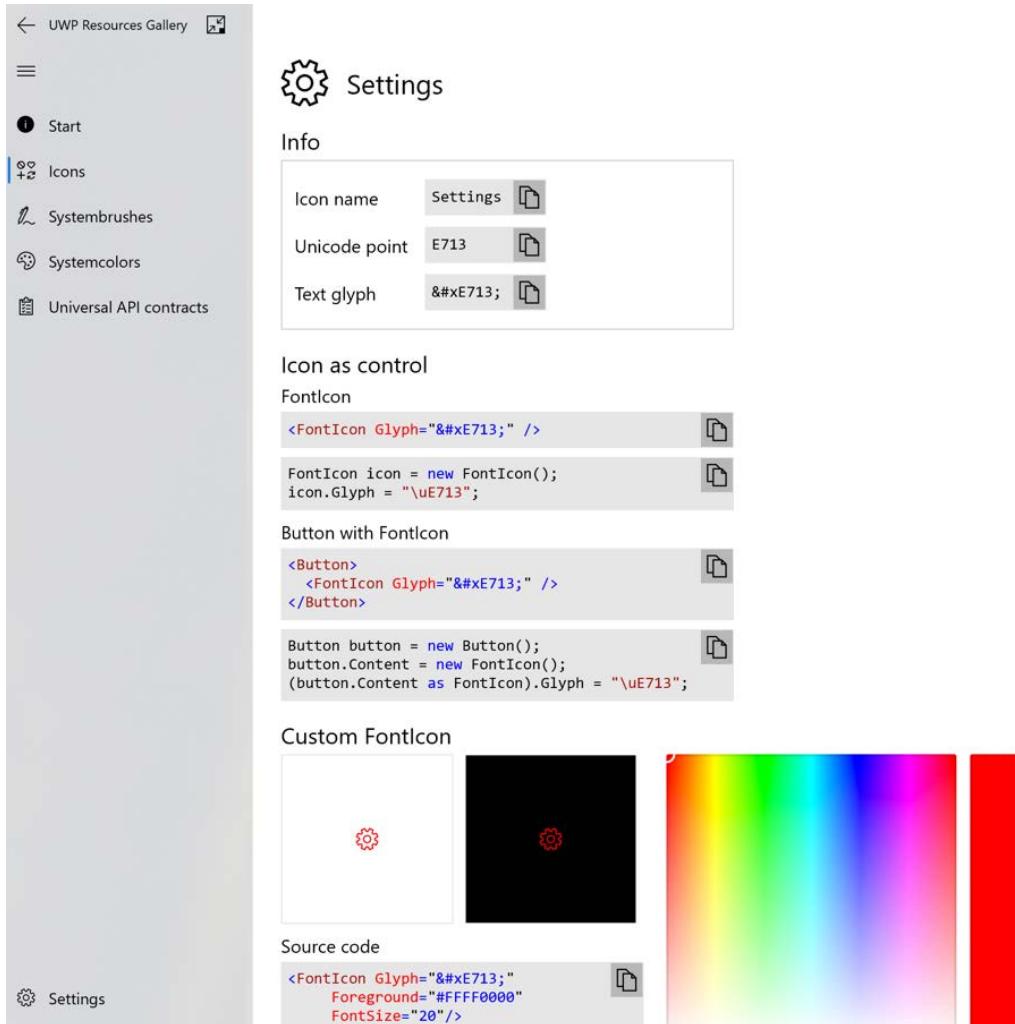


Figure 7.12 – Details about the Settings symbol in UWP Resources Gallery

- **Systembrushes:** In this section, you can search or browse all the system-defined system brushes available in UWP. Click the **Info** icon on any of these brushes to display the XAML definition and the **ThemeResource** snippet for the resource.
- **Systemcolors:** Like the **Systembrushes** section, the **Systemcolors** section provides browsing and searching for all the available color resources. Click on the **Info** icon to get a **ThemeResource** snippet you can copy to your XAML. If there are specific definitions for a color resource for light and dark themes, these are also displayed on the **Info** popup.
- **Universal API contracts:** This section displays the UWP API contracts available for each release of Windows 10. This section is less useful for WinUI 3.0 developers, as the same capabilities are available for all compatible Windows platforms.

Now, we will take a quick look at some additional Fluent Design tools geared toward designers.

## Design resources and toolkits for Fluent Design

While a deep dive into user interface design is beyond the scope of this book, we will briefly review some of the Fluent Design toolkits available for popular design tools. The Fluent Design toolkits for all of these tools can be downloaded from Microsoft Docs at <https://docs.microsoft.com/en-us/windows/uwp/design/downloads/>:

- **Figma:** This is a design and prototyping tool with free and paid options, depending on the team and project size. You can find out more about Figma on its website: <https://www.figma.com/>.
- **Sketch:** This is another popular tool for designing and prototyping applications individually or with teams. There is not a free plan, but Sketch does have a free trial period. Sketch is available at <https://www.sketch.com/>.
- **Adobe XD:** XD is Adobe's design/prototype tool. Like Figma, Adobe XD has free and paid options for designing apps with their tool. Check out XD at <https://www.adobe.com/products/xd.html>.

- **Adobe Illustrator:** This is a powerful vector design tool from Adobe. There is a free trial available. Download the Fluent Design toolkit and get started with Adobe Illustrator at <https://www.adobe.com/products/illustrator.html>. **Inkscape** (<https://inkscape.org/>) is a free vector image editor that can also work with **Adobe Illustrator (AI)** files.
- **Adobe Photoshop:** This is probably one of the best-known raster image editors. Adobe also has a free trial for Photoshop at <https://www.adobe.com/products/photoshop.html>.

The Fluent Design toolkit for Photoshop includes several PSD files. It is also possible to work with PSD files in free image editors such as **GIMP** (<https://www.gimp.org/>) or **Paint.NET** (<https://www.getpaint.net/>). Paint.NET requires an open source plugin, available at <https://www.psdplugin.com/>.

## Summary

We have learned a lot about Fluent Design, design resources, and the tools available to WinUI developers in this chapter. You will be able to use these tools and techniques in your WinUI application design or recommend them to designers at your company. We also updated the My Media Collection app to be more compliant with Fluent Design recommendations.

In the next chapter, we will examine the **Windows Control Toolkit**, an open source bundle of Windows controls for WinUI, UWP, WPF, and WinForms.

## Questions

1. Which platforms have Fluent Design implementations?
2. What is a control pattern?
3. Which font does Microsoft recommend using for Fluent Design?
4. Which aspect of style is specific to devices with larger screens?
5. What are the names of the two spacing densities available in Fluent Design?
6. Which attribute can be set in `Application.xaml` to override a user's light/dark theme selection?
7. Which design tools have Fluent Design toolkits available?

# 8

# Building WinUI Apps with .NET 5

WinUI allows developers to create a new project with C# targeting either the UWP platform or .NET 5. This chapter will explore the new WinUI in Desktop project type built on .NET 5. The WinUI in Desktop project has the same application model as WPF projects. We will learn how it differs from the traditional WinUI in UWP platform project. Projects that target .NET 5 have the advantage of being able to directly reference other .NET 5 libraries. We will explore this and show you how to create your own .NET 5 control library. Finally, we will look at the packaging project that is included with a WinUI in Desktop solution.

In this chapter, we will cover the following topics:

- Creating a new WinUI in Desktop project
- Discovering the differences between WinUI in Desktop and WinUI in UWP projects
- Creating a .NET 5 library and using it from a WinUI in Desktop project
- Exploring the packaging project included with the WinUI in Desktop solution

By the end of this chapter, you will understand how to create a WinUI in Desktop application with .NET 5. You will also understand how they get packaged so that they can be deployed on Windows or distributed via the Microsoft Store.

**Note**

Sometime after WinUI 3.0 RTM is released, the WinUI in UWP projects will also target .NET 5. It is not going to be part of the first stable version. That support and tooling are not yet available to the UWP app platform. We'll be exploring WinUI in Desktop, the project type that currently supports .NET 5.

## Technical requirements

To follow along with the examples in this chapter, the following software is required:

- Windows 10 version 1803 (build 17134) or newer
- Visual Studio 2019 version 16.9 or newer with the following workloads: .NET Desktop Development, Universal Windows Platform Development

The source code for this chapter is available on GitHub at this URL: <https://github.com/PacktPublishing/-Learn-WinUI-3.0/tree/master/Chapter08>.

## Creating a WinUI project with .NET 5

WinUI adds several different project templates to Visual Studio. We have spent most of the book working with the **Blank App (WinUI in UWP)** C# template. This template is also available to C++ developers. These other templates are also available:

- **Blank App, Packaged (WinUI in Desktop)** (C# and C++): Creates a .NET 5 desktop application with support for WinUI controls in the UI.
- **Class Library (WinUI in Desktop)** (C#): Creates a class library project intended to be used with WinUI in Desktop apps and libraries.
- **Class Library (WinUI in UWP)** (C#): Creates a class library project intended for use with WinUI in UWP apps and libraries.
- **Windows Runtime Component (WinUI in UWP)** (C# and C++): Creates a Windows Runtime Component project. This is a packaged component with a .winmd file extension that can be consumed by any language with **Windows Runtime** projections, including C++, C#, Visual Basic, and JavaScript (React Native, for example).

In this section, we will be creating a new **WinUI in Desktop** project in C#. Before we get started, we should examine the difference between the WinUI in UWP, WinUI in Desktop, and WPF projects.

## What is WinUI in Desktop?

The WinUI in Desktop project is very much like a modern WPF desktop project. To help you understand this project type, let's review some of the primary things it has in common with WPF. Both of these desktop project types target the .NET 5 runtime. This enables them to consume other .NET libraries. They can also access WinRT APIs by changing the project's .NET 5 **target framework metadata (TFM)** to include the Windows version info. Here is an example:

```
<TargetFramework>net5.0-
  windows10.0.19041.0</TargetFramework>
```

The same technique is available to other .NET 5 applications to access WinRT APIs. For example, you could add this to a .NET 5 WPF project.

The default WinUI in Desktop template also adds a packaging project for deployment. As you will see in this chapter, the packaging project opens up WinRT access by providing an identity to desktop applications.

The primary difference between WinUI in Desktop and WPF projects is in the XAML markup. WinUI still uses the UWP XAML syntax and markup extensions in desktop projects, which differs from WPF XAML. WPF projects still require XAML Islands to incorporate WinUI controls into their UI. Project Reunion is a broad effort from Microsoft to unify Windows development, and WinUI 3.0 appears to be a big step in that direction.

You might be wondering what makes the desktop project different from its UWP counterpart. There are two primary differences:

- First, the desktop template creates a *solution* with two projects. The second project is a packaging project. The packaging project provides desktop applications with a **package identity** on Windows so they can access specific WinRT APIs that require identity. We will explore this project later in the chapter.
- The second major difference is the default *navigation model*. Up to this point, we have been building apps in a single window and navigated between pages within that window. The WinUI in Desktop project starts with a **MainWindow**, as you would see in a new WPF project. From this window, your desktop app can create and launch other modal and non-modal windows. Building a WinUI in Desktop app is very similar to building a WPF app but with WinUI XAML syntax.
- For the initial release of WinUI 3.0, the two project types target different runtimes. **WinUI in UWP** preview continues to use .NET Native while **WinUI in Desktop** uses .NET 5.

The desktop project has access to all the WinUI libraries we have been using throughout the book, and the target runtime is .NET 5. Let's start building a new application with WinUI in Desktop.

## Creating a new WinUI in Desktop project

Creating a WinUI in Desktop project is very straightforward. Let's step through the process:

1. Start by launching Visual Studio, select **Create a new project**, and click **Next**.
2. Next, on the **Create a new project** page, search for **WinUI in Desktop**, and select the **Blank App, Packaged (WinUI in Desktop)** template in the language of your choice and click **Next** again:

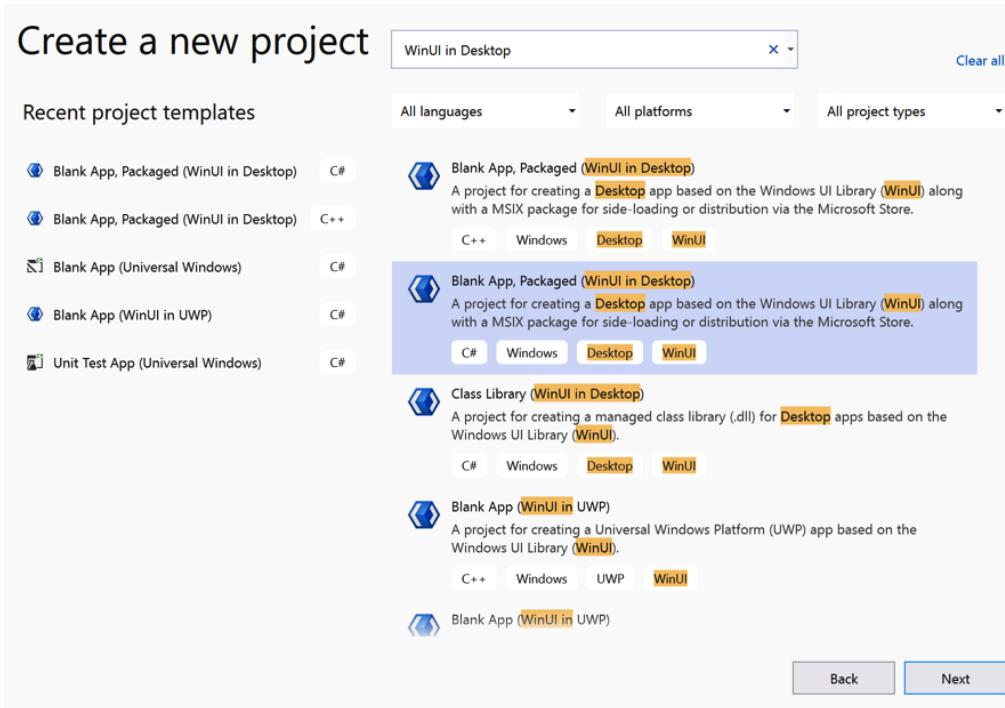


Figure 8.1 – Selecting the WinUI in Desktop template in Visual Studio

3. On the **Configure your new project** page, name the project **WebViewBrowser** and click **Create**. We will be creating a project that uses the new **WebView2** control in WinUI:

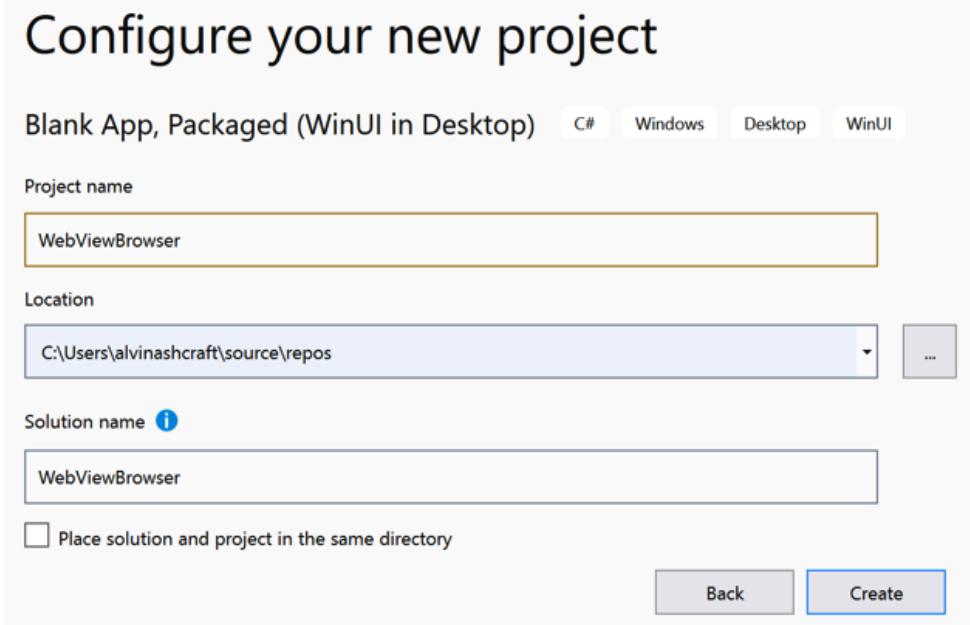


Figure 8.2 – Configuring the WebViewBrowser project

4. Visual Studio will now prompt you to select the **Target version** and the **Minimum version** of Windows your app will target. This dialog still references UWP, but what you are selecting here are the versions of Windows you plan to support in your packaged desktop project. You can leave the default versions selected and click **OK**:



Figure 8.3 – Selecting target and minimum Windows versions

Visual Studio will then generate your new solution and load it into **Solution Explorer**. It is always best to build and run the project before you begin to make any changes. In the next section, we will explore the structure of the **WebViewBrowser** project.

## Exploring the Desktop project structure

In this section, we are going to examine the files in the **WebViewBrowser** desktop project that was generated by the Visual Studio template. Exploring the differences between this project and our earlier WinUI projects will help you understand how the desktop app executes. When Visual Studio first loads the new solution, it displays some information about the packaging project:

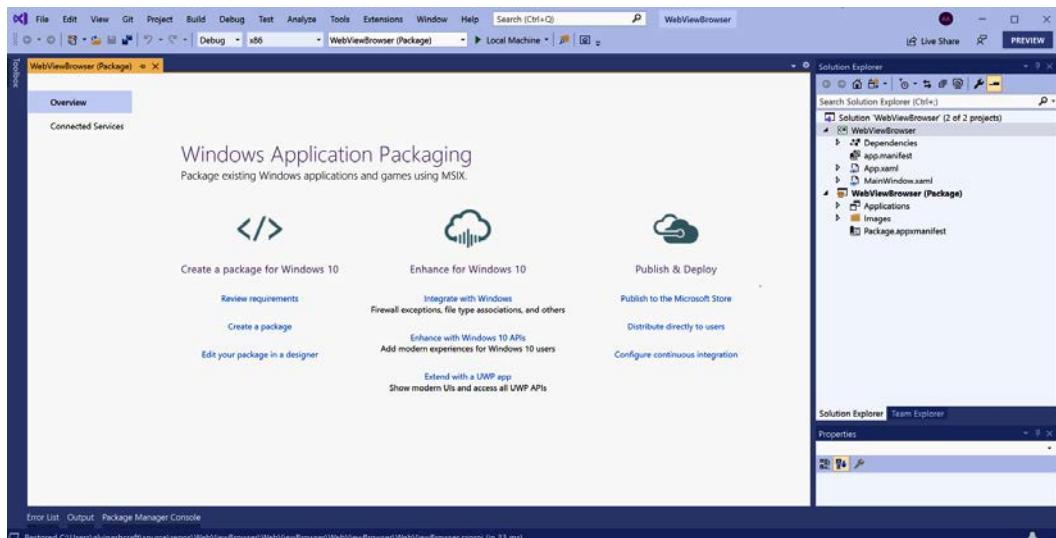


Figure 8.4 – The WebViewBrowser solution in Visual Studio

We will explore the packaging project in the next section. Let's start by looking at some of the properties of the **WebViewBrowser** project. Right-click the project and select **Properties**. On the **Application** tab of the properties, you can see that the **Target framework** is **.NET 5.0** and the **Output type** is **Windows Application**. You would see these same values in a new WPF project's properties.

Now let's open the **App.xaml** file. This looks the same as the **App.xaml** files in our other WinUI on UWP projects. However, opening **App.xaml.cs** is a different story. The **App** constructor is the same, but the **OnLaunched** event handler is very different in the desktop project:

```
protected override void OnLaunched(Microsoft.UI.Xaml.  
LaunchActivatedEventArgs args)  
{  
    m_window = new MainWindow();  
    m_window.Activate();  
}
```

There are no references to frames or navigation because we won't have to handle any page navigation in the desktop app. This is what you also find in a WPF application. While navigating with pages is supported in both WPF and WinUI desktop applications, it is not required. All our project does on launch is to create a new instance of `MainWindow` and call `Activate` on it. These controls and classes are all derived from types in the `Microsoft.UI.Xaml` namespace in WinUI.

Now let's open `MainWindow.xaml`. Again, this looks very much like our other WinUI XAML files, but the root element is a `Window` instead of a `Page`. The `Window` contains a `StackPanel` containing a single `Button` named `myButton`. `myButton` has a `Click` event handler, `myButton_Click`, in the `MainWindow.xaml.cs` file that changes the button's content from **Click Me** to **Clicked**:

```
private void myButton_Click(object sender, RoutedEventArgs
    e)
{
    myButton.Content = "Clicked";
}
```

That's all there is to the initial project. The packaging project is set as the startup project by default. Leave that setting, as it is required for WinUI desktop projects, and run it to make sure everything works as expected before we start making any changes.

## Adding the WebView2 control

Let's explore the `WebView2` control, which is new to WinUI 3. The `WebView2` control is like the original `WebView` control except that it uses the new Chromium-based **Microsoft Edge** to render the web content.

### Note

Chromium is the open source browser on which **Google Chrome** and many other popular browsers are based. You can read more about the Chromium project at <https://www.chromium.org/>.

This new Microsoft Edge browser has several versions available in addition to the stable release:

- **Beta**: Receives updates *every 6 weeks* and is the most stable preview release
- **Dev**: Receives updates *weekly* and may have some minor bugs
- **Canary**: Receives *daily updates* and may sometimes have stability issues

The Edge preview releases are available at <https://www.microsoftedgeinsider.com/en-us/download>, where you can select which channel you would like to download:

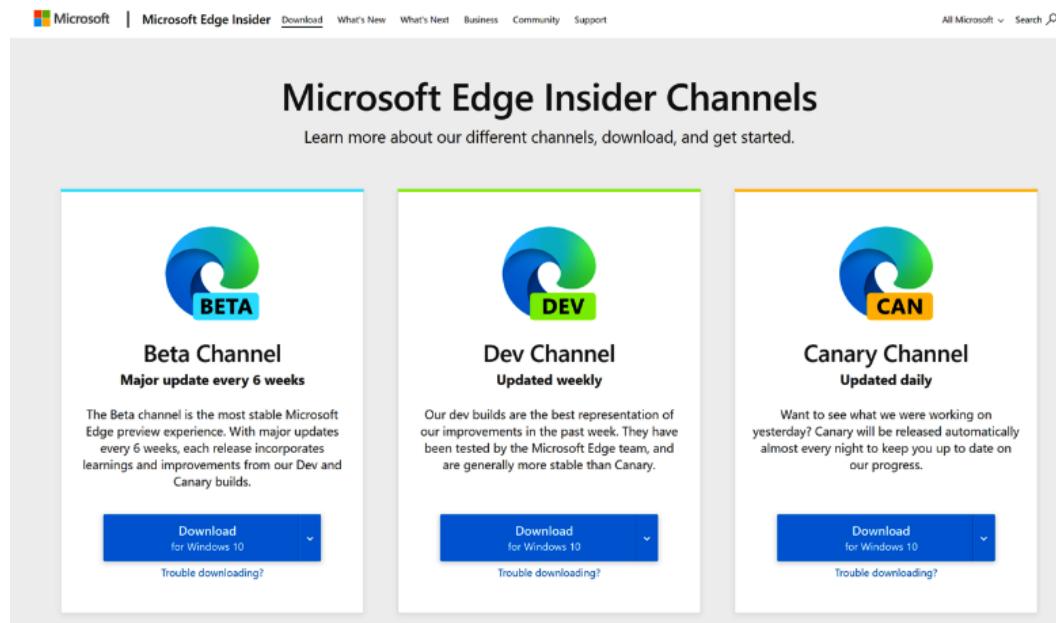


Figure 8.5 – The Microsoft Edge Insider download page

During the WinUI preview release, the Beta or Canary channel of Microsoft Edge is required to run the WebView2 control. Now, we'll add an Edge WebView2 to our WinUI application:

1. First, download and install the Beta release from the Edge Insider page before we make any changes to use WebView2.
2. Next, replace the contents of **Window** in **MainWindow.xaml** with **Grid** containing a **WebView2** control and set the **Source** property to <https://www.packtpub.com/>:

```
<Grid>
    <WebView2 Source="https://www.packtpub.com/" />
</Grid>
```

3. Now, remove the `myButton_Click` event handler from `MainWindow.xaml.cs`.
4. Finally, run the app. It will launch a window displaying the **Packt** home page inside the browser control:

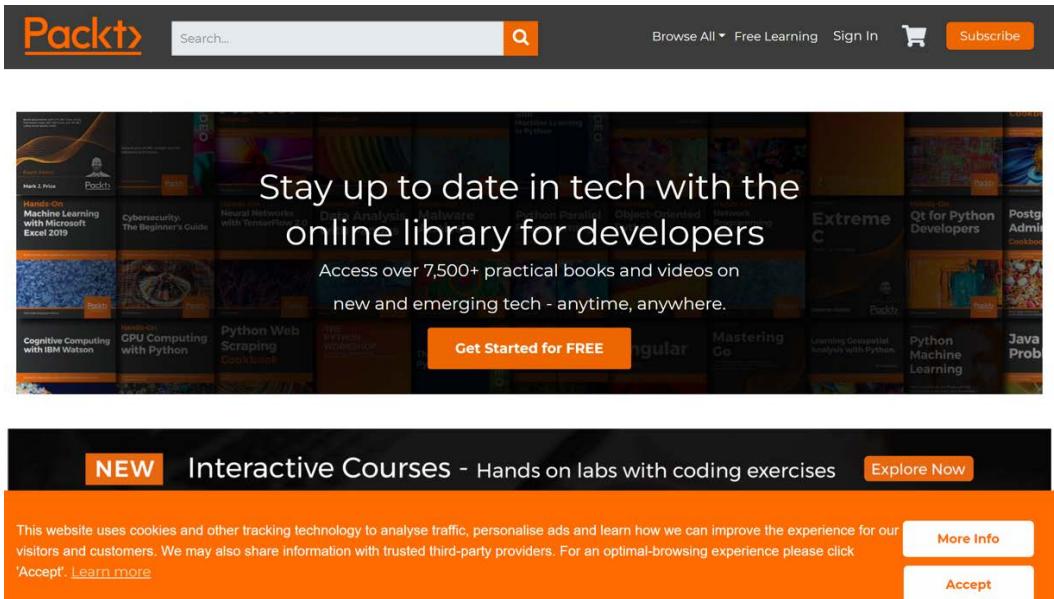


Figure 8.6 – The WebViewBrowser app with the Packt home page loaded

Just like that, we have a browser window, but without any toolbar or other navigation controls. All the navigation is controlled within our code unless we expose our controls to the user.

We should update our `WebView2` app to use MVVM and data binding to set the `Source` URL instead of having it hardcoded in the XAML markup. Before we continue with that change, let's explore the **WebViewBrowser (Package)** project.

## Exploring the Packaging project

The **WebViewBrowser (Package)** project is a **Windows Application Packaging Project** that will create an **MSIX** package for deploying to Windows and the Microsoft Store. The packaging project is required to provide the application with an identity in Windows and to run and debug desktop WinUI projects in Visual Studio.

The packaging identity allows Windows to identify the calling application to provide access to Windows APIs and resources, based on the capabilities granted to that application. In this section, we will see how to configure the project to declare which resources are required by your application. This Microsoft Docs article explains more about how packaged desktop applications run: <https://docs.microsoft.com/en-us/windows/msix/desktop/desktop-to-uwp-behind-the-scenes>.

Packaging projects are not limited to WinUI desktop apps. You can package any Windows desktop application, including C++/Win32, WinForms, and WPF projects. Let's have a closer look at what's included in the project:

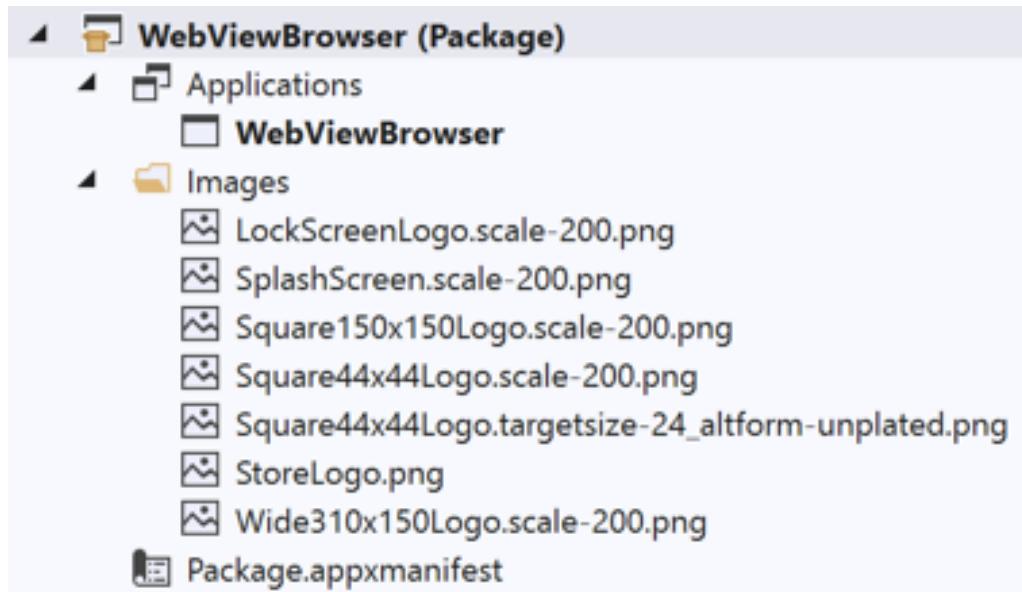


Figure 8.7 – The Windows Application Packaging Project in Solution Explorer

The first node in the project, **Applications**, contains a reference to the **WebViewBrowser** project. You can include multiple projects in a packaging project if your solution contains additional projects. To add another project, right-click the **Application** item and select **Add Reference**. A **Reference Manager** dialog window will appear with a list of projects in the solution that can be added to the package.

The **Images** folder contains images that you would normally find in a new WinUI in UWP project. These are the default images used for your app in the Microsoft Store, in the Windows Start menu, and on your splash screen. You should replace these images with your own before publishing or distributing the app. It sounds like a lot of work creating all these different sized images, but Visual Studio provides a shortcut for developers.

## Visual assets in the Manifest Designer

To generate all the images for your Windows app, you need to start with a single image. Vector images work best (.pdf and .ai formats), but if you have a bitmap type of image (.png, .jpg, .tiff, .bmp, and .gif formats), it should be *400px x 400px*. Take your image and follow these steps to generate all the images for your project:

1. Start by double-clicking the Package.appxmanifest file in **Solution Explorer**. This will open the **Manifest Designer** for your packing project.
2. Switch to the **Visual Assets** tab and find the **Source** field in the **Asset Generator** section near the top of the page. Click the browse button, ..., to the right of the field. Select your image and click **Open**:

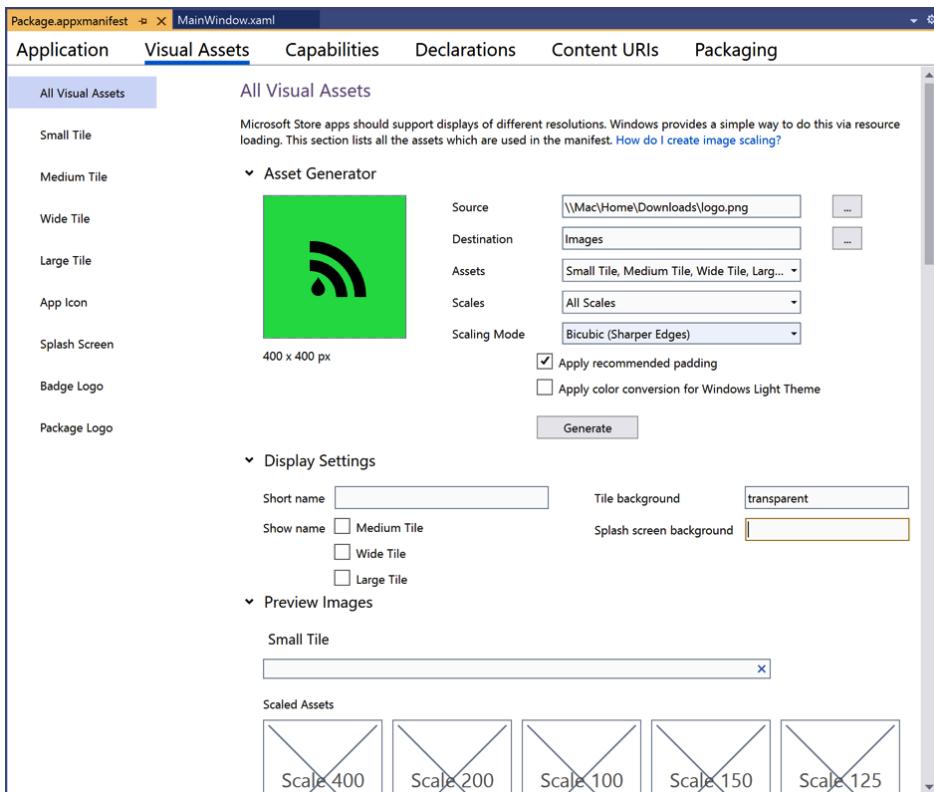


Figure 8.8 – Generating visual assets for an application package in Visual Studio

3. Change any other options you would like and click the **Generate** button. Your application images will be generated, overwriting the default images in the packaging project. Now when you run your app, you will see your logo on the splash screen.

The generator is a handy resource to quickly generate all of the formats and sizes needed, but for a professional-looking application, you should review and edit each image individually to ensure it is pixel-perfect. You can customize or replace any of the generated images in the **Visual Assets** pane.

If you only wanted to generate one type of image, you could select that image type on the left side of the page instead of using the **All Visual Assets** section.

The package manifest allows you to declare other information about your app when it is installed. Let's explore the other areas of the manifest:

- **Application:** This section declares some general information about your app. Here you can define things such as **Display name**, **Description**, **Default language**, and **Supported rotations**. If you want your app to display notifications on the Windows lock screen or use a **Live Tile**, these can be configured here.
- **Capabilities:** If your application needs to request access to any special capabilities, they must be declared in the manifest. This includes things such as access to **Removable Storage**, **Music Library**, **Location**, or the device's **Microphone**.
- **Declarations:** This section allows you to declare capabilities that require additional configuration. Some examples of these include **Share Target**, **Open File Picker**, **Media Playback**, **Lock Screen**, and **Search**. If you declare **Search**, your app will be registered in Windows as a search provider, allowing it to display information from the app in Windows search results.
- **Content URIs:** Adding content **Uniform Resource Identifiers (URIs)** to the manifest allows your app to be declared as the default handler for those URIs. If you add any of these URIs here, you must make sure that your app handles it in the way users would expect when it is activated. Microsoft Docs has additional information about handling URI activation here: <https://docs.microsoft.com/en-us/windows/uwp/launch-resume/handle-uri-activation>.
- **Packaging:** This tab contains general information about the package, including **Package name**, **Version**, and **Publisher display name**.

This information is saved in **Package.appxmanifest** in XML format. We will explore packaging and deployment in more depth in *Chapter 14, Packaging and Deployment Options for Windows*. Next, we will look at how to reference other .NET library projects from a WinUI desktop project.

# Referencing .NET 5 Libraries from your project

A WinUI desktop project is a .NET 5 project, so it's easy to reference any other .NET 5 project, NuGet package, or local DLL. We want to update our **WebViewBrowser** project to use MVVM but using MVVM doesn't require the views and ViewModel classes to be in the same project. It's possible to share ViewModels across multiple projects if they are created in their own .NET assembly.

We are going to add a .NET library to the **WebViewBrowser** solution to hold a **ViewModel** class for the **MainWindow**:

1. Start by right-clicking the solution file in **Solution Explorer** and select **Add | New Project**.
2. On the **Add a new project** window, select **C#** from the **Language** dropdown and **Library** from the **Project Types** dropdown to filter the list of templates. Find and select the **Class Library (.NET Core)** template and click **Next**.

## Note

A .NET Standard library project can also be created for sharing code across multiple projects. The advantage of using .NET Standard over .NET Core is that you can generate binaries for multiple runtimes, including .NET Framework, .NET Core, and Mono. There is no .NET Standard specification that includes .NET 5. This is because .NET 5 is in fact a standard set of libraries for all platforms. To read about the future of .NET Standard, check out this blog post from the .NET team: <https://devblogs.microsoft.com/dotnet/the-future-of-net-standard/>.

To read more about .NET Standard and see which runtimes are compatible with each .NET Standard version, visit this Microsoft Docs page: <https://docs.microsoft.com/en-us/dotnet/standard/net-standard>.

3. Name the new project `WebViewBrowser.Bus` and click **Create**. This will add the new .NET library project to **Solution Explorer**.
4. Remove the `class1.cs` file from the project and add a folder named **ViewModels**. Add two new classes to the **ViewModels** folder named `ViewModelBase` and `MainViewModel`.
5. The `ViewModelBase` class will contain an implementation of `INotifyPropertyChanged` like `BindableBase` in the **MyMediaCollection** project in earlier chapters:

```
public class ViewModelBase : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler
        PropertyChanged;
    protected bool SetProperty<T>(ref T originalValue,
        T newValue, [CallerMemberName]
        string propertyName = null)
    {
        if (!EqualityComparer<T>.Default
            .Equals(originalValue, newValue))
        {
            originalValue = newValue;
            OnPropertyChanged(propertyName, newValue);
            return true;
        }

        return false;
    }
    private void OnPropertyChanged(string
        propertyName,
        object value)
    {
        PropertyChanged?.Invoke(this, new
            PropertyChangedEventArgs(propertyName));
    }
    protected void
        OnPropertyChanged([CallerMemberName] string
            propertyName = null)
```

```
    {
        PropertyChanged?.Invoke(this, new
        PropertyChangedEventArgs(propertyName));
    }
}
```

6. The **MainViewModel** class will inherit from **ViewModelBase** and contain a property for **UrlSource** of our browser:

```
public class MainViewModel : ViewModelBase
{
    private string _urlSource =
        "https://www.packtpub.com/";
    public string UrlSource
    {
        get
        {
            return _urlSource;
        }
        set
        {
            SetProperty(ref _urlSource, value
                , nameof(UrlSource));
        }
    }
}
```

7. Next, right-click on the **Dependencies** node in the **WebViewBrowser** project and select **Add Project Reference**.

8. In the **Reference Manager** dialog, select **WebViewBrowser.Bus** and click **OK**:

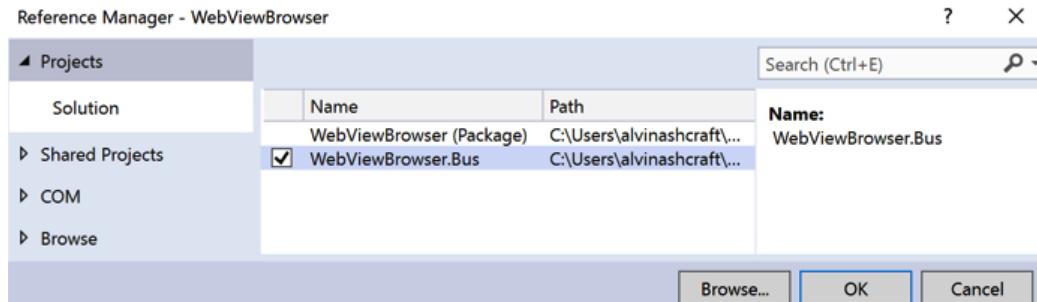


Figure 8.9 – Adding a reference to WebViewBrowser.Bus

9. Next, open **MainWindow.xaml.cs** and add a property named **ViewModel** to use with **x:Bind** in the view. Be sure to add a **using** statement for the **WebViewBrowser.Bus.ViewModels** namespace:

```
public MainViewModel ViewModel { get; } = new MainViewModel();
```

10. Finally, open **MainWindow.xaml** and update the **WebView2** to set the **Source** property with **x:Bind** instead of the hardcoded URL:

```
<WebView2 Source="{x:Bind ViewModel.UrlSource}" />
```

Now when you run the app again, you should still see the Packt home page load into the browser window. However, we're now getting the URL from the value set in the **ViewModel**. In the next section, we'll look at how to bind that same **ViewModel** to a WPF view.

## Sharing the .NET 5 library with a WPF application

Class libraries are a great way to share code across multiple projects. It's also one way you can incrementally migrate applications to a new UI platform like WinUI. If you have your **ViewModels** or other business logic in separate .NET assemblies (or web services), the effort needed to build a new and modern UI is greatly reduced. If your existing desktop apps are single assembly monoliths, refactoring business logic into a separate class library is a great first step in a migration effort.

By defining our ViewModel in a separate .NET class library project, we can easily consume it in multiple UI projects. This separation also helps to ensure that the ViewModels will not have any dependencies on WinUI or other UI frameworks. Let's create a WPF project that also uses a WebView2 control:

1. Start by making sure you have the required version of Microsoft Edge installed. At the time of this writing, the Dev or Canary channel of Edge is required to use WebView2 with a WPF application. Find more info at <https://docs.microsoft.com/en-us/microsoft-edge/webview2/gettingstarted/wpf>.
2. Add another new project to the current solution. On the **Add a new project** dialog, search for `wpf` and select **WPF App (.NET Core)** as the language of choice:

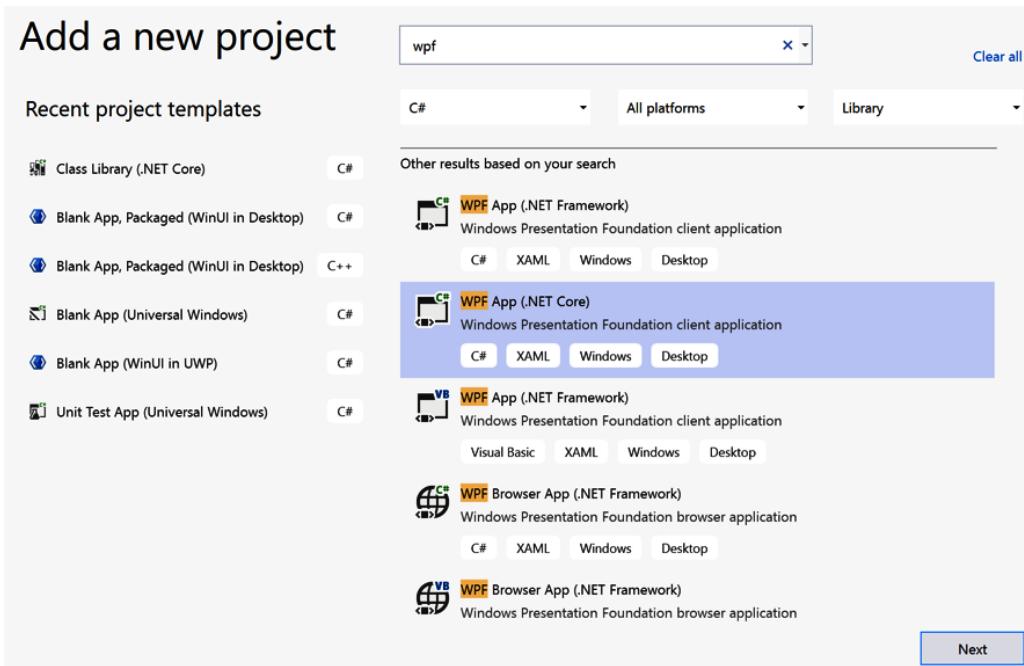


Figure 8.10 – Add a new WPF project to the solution

3. Name the project **WebViewBrowser.WPF** and click **Create**. After creating the project, you can verify that it is targeting .NET 5 on the project properties page:

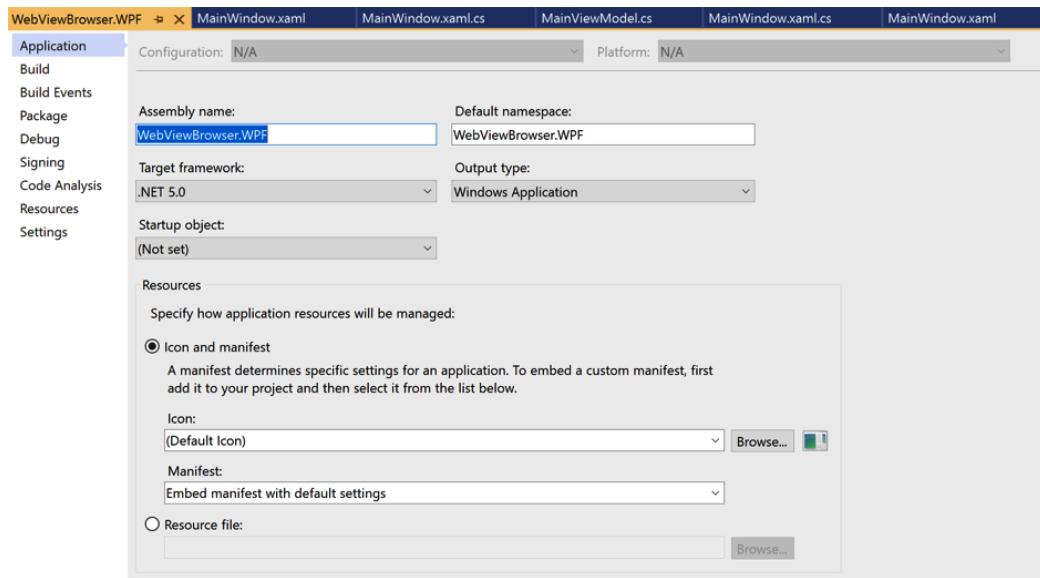


Figure 8.11 – The WPF project's properties designer

4. Right-click the project in **Solution Explorer** and select **Set as Startup Project** to ensure this project launches when we run from Visual Studio. Run the project to make sure it compiles and launches without any errors.
5. Next, open the **NuGet Package Manager** window, check the **Include prerelease** checkbox, and search for **Microsoft.Web.WebView2**. Add the package to the **WebViewBrowser.WPF** project, making sure to select the newest prerelease version:

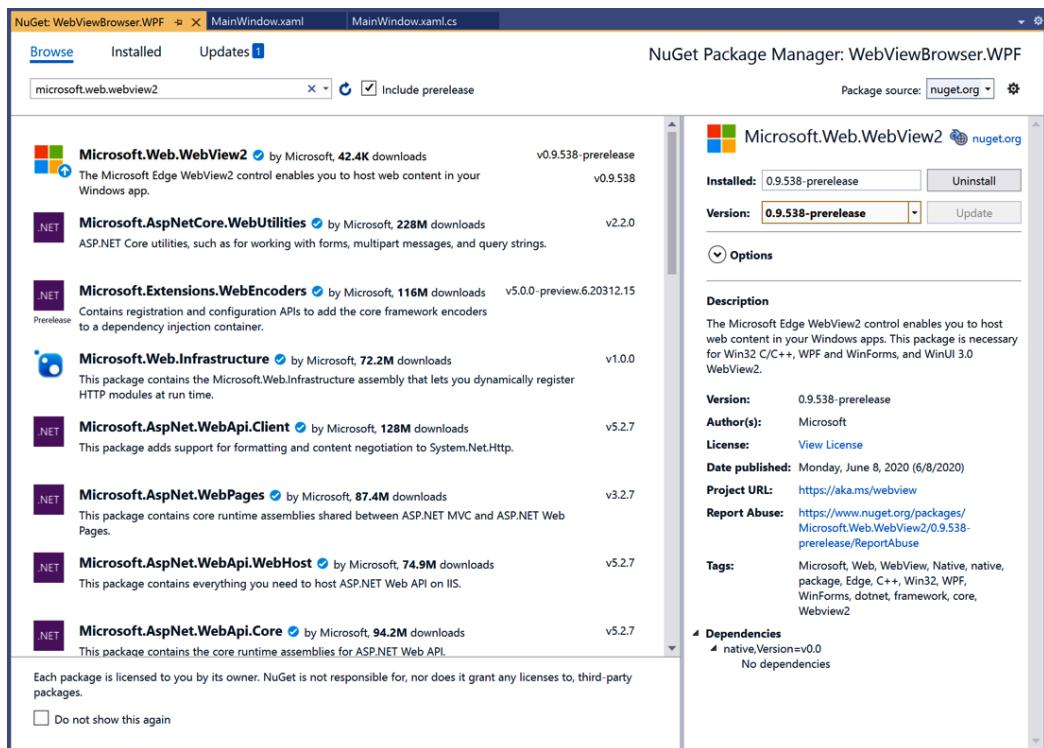


Figure 8.12 – Adding the WebView2 packages to the WPF project

6. Add a project reference to **WebViewBrowser.Bus** in **WebViewBrowser.WPF**.
7. Open **MainWindow.xaml.cs** from the WPF project and set **DataContext** to a new instance of **MainViewModel**. This will enable us to use the members of **MainViewModel** in **Binding** expressions in **MainWindow**. The WPF data binding does not support the **x:Bind** markup extension. So, we need to use the **DataContext** with the **Binding** markup extension:

```
public MainWindow()
{
    InitializeComponent();
    DataContext = new MainViewModel();
}
```

8. In **MainWindow.xaml**, add a namespace reference to the Window for the namespace of `WebView2`:

```
xmlns:wv2="clr-
namespace:Microsoft.Web.WebView2.Wpf;assembly=Microsoft
t.Web.WebView2.Wpf"
```

9. Finally, replace the contents of `Grid` with the `WebView2` control. Use the `Binding` markup extension to bind to the `UrlSource` property in the current `DataContext`:

```
<Grid>
    <wv2:WebView2 Source="{Binding UrlSource}" />
</Grid>
```

10. Now, run the WPF application, and it should look just like the WinUI desktop app:

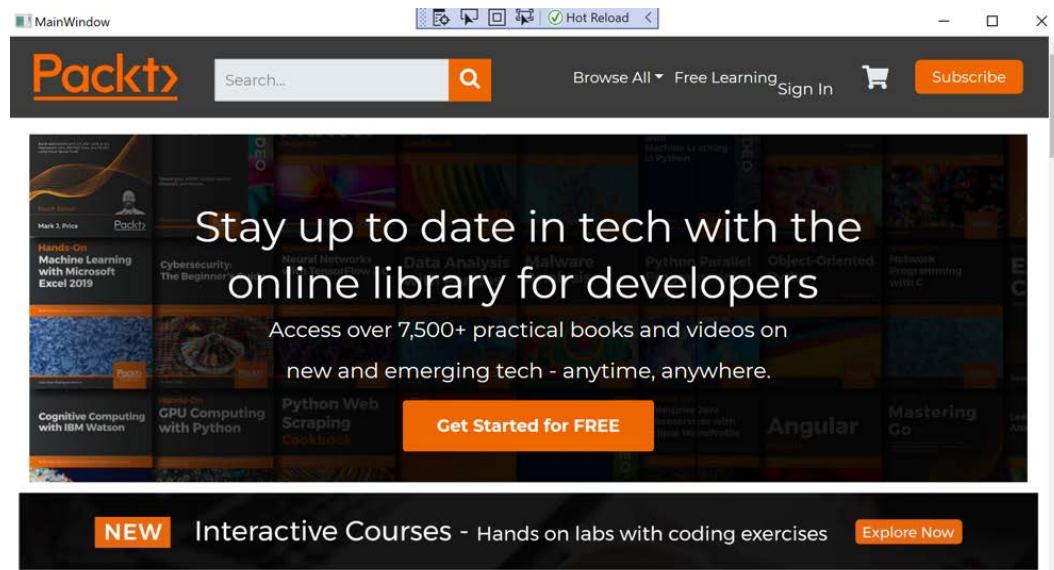


Figure 8.13 – A `WebView2` control running in a WPF application

Now that you have seen how to share libraries between WinUI desktop and WPF apps, let's shift back to WinUI and see how to create a reusable control library.

# Creating a WinUI control library

A **control library** is a great way to reuse your WinUI controls across projects, much in the same way that .NET libraries enable projects to share business logic. This concept is not new to WinUI; you can create control libraries for most UI frameworks. We are going to create a control library with a user control containing two WinUI controls:

- A `TextBox` to enter a URL where our users want that `WebView2` to navigate
- A reload `Button` to reload the current web page in `WebView2`

We will use one event to trigger the reload and another to notify the host that a new URL has been entered:

1. Start by adding a new **Control Library (WinUI in Desktop)** project to the solution in your language of choice. Name the project `WebViewBrowser.Controls`.
2. Remove `Class1.cs` from the project and use **Add | New Item** to add a new **User Control (WinUI)** named `BrowserToolbar` to the project.

#### Note

A **User Control** is best suited to our needs here. We want to define the layout of several WinUI controls in the XAML file. A **Custom Control** would typically be used when you are inheriting from another WinUI control and adding or overriding some of the base control's behaviors.

3. In `BrowserToolbar.xaml`, define a `Grid` with two columns containing a `TextBox` and a `Button`. The `TextBox` will bind its `Text` attribute to a `UrlSource` property. Setting `UpdateSourceTrigger` to `PropertyChanged` will force `INotifyPropertyChanged` each time the text changes, instead of waiting until focus is lost on the `TextBox`. We will be handling a `TextBox.KeyUp` event and a `Button.Click` event in the code-behind file:

```

<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*"/>
        <ColumnDefinition Width="Auto"/>
    </Grid.ColumnDefinitions>
    <TextBox x:Name="urlTextBox"
        KeyUp="urlTextBox_KeyUp"
        Text="{x:Bind UrlSource, Mode=TwoWay,
        UpdateSourceTrigger=PropertyChanged}"/>

```

```
<Button x:Name="reloadButton"
        Grid.Column="1"
        Click="reloadButton_Click">
    <SymbolIcon Symbol="Refresh" />
</Button>
</Grid>
```

4. In the **Toolbar.xaml.cs** file, start by creating two public events to be raised to the hosting app and a constant to hold the initial URL:

```
private const string InitialUrl =
    "https://www.packtpub.com/";
public event RoutedEventHandler ReloadClicked;
public event RoutedEventHandler UrlEntered;
```

5. Next, create a **DependencyProperty** for the **UrlSource**. Using a dependency property makes it available to data binding. We won't be using binding in the host project in this example. We only want the host to refresh the browser when users hit **Enter** in the URL bar. However, the user control should generally support binding on its public properties. When registering a dependency property, you will provide the name of the standard property, its data type, the data type of the containing control, and optionally a default value:

```
public static readonly DependencyProperty
UrlSourceProperty =
DependencyProperty.Register(nameof(UrlSource),
                           typeof(System.Uri),
                           typeof(BrowserToolbar),
                           new
                           PropertyMetadata(null));
public System.Uri UrlSource
{
    get { return
        (System.Uri)GetValue(UrlSourceProperty); }
    set { SetValue(UrlSourceProperty, value); }
}
```

6. Now, add the following code to the constructor to initialize the URL in the control and notify the parent of the change:

```
public BrowserToolbar()
{
    this.InitializeComponent();

    urlTextBox.Text = InitialUrl;
    UrlSource = new System.Uri(InitialUrl);
    UrlEntered?.Invoke(this, new RoutedEventArgs());
}
```

7. The final step in the user control project is to create the event handlers that were referenced in the XAML. These will, in turn, invoke the events to be handled by the hosting app. The `KeyUp` event checks that the *Enter* key was pressed and that `urlTextBox.Text` is not blank. You could add some additional validation to ensure that the URL is in a valid format:

```
public void urlTextBox_KeyUp(object sender,
    KeyRoutedEventArgs e)
{
    if (e.Key == Windows.System.VirtualKey.Enter &&
        !string.IsNullOrWhiteSpace(urlTextBox.Text))
    {
        UrlEntered?.Invoke(this, new
            RoutedEventArgs());
    }
}

private void reloadButton_Click(object sender,
    RoutedEventArgs e)
{
    ReloadClicked?.Invoke(this, new
        RoutedEventArgs());
}
```

8. Now, back in the **WebViewBrowser** project, open the **MainWindow.xaml** file and update the **Grid** to have two rows. The first row will contain the **BrowserToolbar** user control, and the second row will contain **WebView2**. The **ReloadClicked** and **UrlEntered** events of **BrowserToolbar** need to be handled:

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="*"/>
    </Grid.RowDefinitions>
    <ctrl:BrowserToolbar x:Name="browserToolbar"
        ReloadClicked="browserToolbar_ReloadClicked"
        UrlEntered="browserToolbar_UrlEntered"/>
    <WebView2 x:Name="mainWebView"
        Grid.Row="1"/>
</Grid>
```

9. In **MainWindow.xaml.cs**, update the constructor to set the **Source** of the **mainWebView** to the initial value of **browserToolbar.UrlSource**:

```
public MainWindow()
{
    this.InitializeComponent();
    mainWebView.Source = browserToolbar.UrlSource;
}
```

10. Finally, add the two event handlers to update the **mainWebView** when the **UrlSource** is updated or a reload is requested by the user control:

```
private void browserToolbar_ReloadClicked(object
    sender, RoutedEventArgs e)
{
    mainWebView.Reload();
}

private void browserToolbar_UrlEntered(object sender,
    RoutedEventArgs e)
```

```
{
    mainWebView.Source = browserToolbar.UrlSource;
}
```

11. Run the app, and you will see the Packt website load when the window is first launched. Now try the toolbar. You can enter <https://docs.microsoft.com> and press *Enter* to load the Microsoft Docs home page in the browser. Then click **Reload** to test that event:

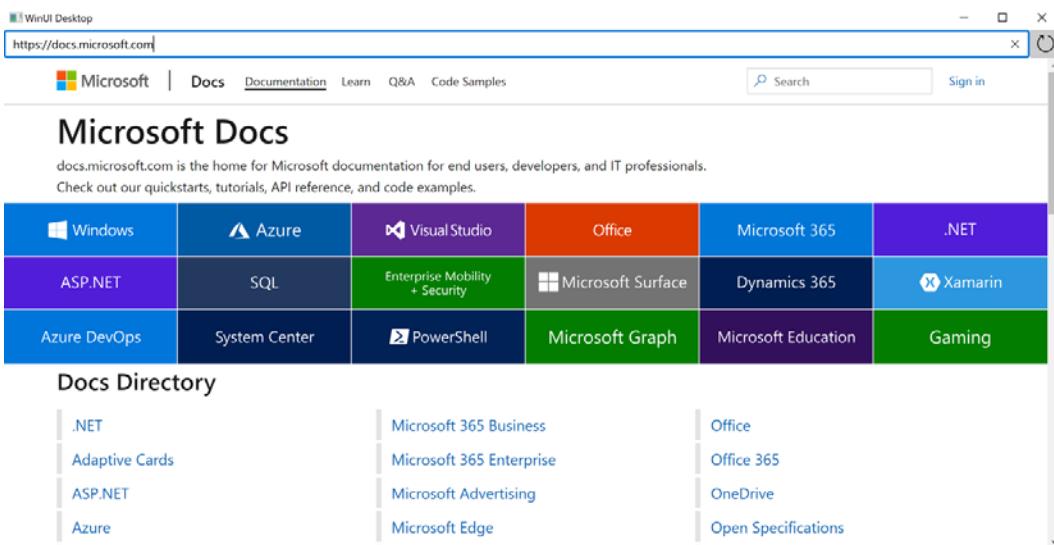


Figure 8.14 – The WebViewBrowser application with a functional toolbar

The browser application is ready for some additional features. I'll leave those up to you. Let's wrap up the chapter with a review of what we've covered.

## Summary

In this chapter, we have created a solution with several different projects. We learned about the WinUI in Desktop project, the WinUI control library, .NET 5 class library, and even a WPF desktop project with .NET 5. We have discussed the advantages of separating controls and logic into multiple projects to promote reuse. We also examined the packaging project, which can be used to package and distribute any kind of Windows desktop application. You will be able to take what you have learned in this chapter to either build new WinUI projects or to start preparing your existing desktop applications for migration to WinUI.

We will continue in this vein in the next chapter when we learn about the Windows Community Toolkit, which includes **XAML Islands** controls with the ability to host WinUI controls in WPF and WinForms applications. You will explore this and many more controls, helpers, and extensions that the toolkit offers Windows developers.

## Questions

1. What is the additional project generated by Visual Studio when creating a new WinUI in Desktop solution?
2. Where would you declare that your application will need access to the user's microphone?
3. How can you share business logic between a WPF application and a WinUI desktop application?
4. What project type allows custom WinUI controls to be shared with multiple desktop projects?
5. What type of property should you create in your user control to allow it to participate in data binding?
6. Where is `MainWindow` instantiated in a WinUI in Desktop project?
7. Where can you define the application icons and images to be used by your WinUI in Desktop app?

# 9

# Enhancing Applications with the Windows Community Toolkit

The **Windows Community Toolkit** (WCT) is a collection of open source libraries for Windows developers. The toolkit contains controls and libraries that can be leveraged by WinUI, UWP, WPF, and WinForms applications. Among the controls in the toolkit are the **XAML Islands** container controls, which we will explore in more detail in *Chapter 10, Modernizing Existing Applications with WinUI and XAML Islands*. In the Microsoft Store, there is a companion sample application for the toolkit that developers can install to explore the controls and learn how to use them.

In this chapter, we will cover the following topics:

- Learning about the background and purpose of the toolkit
- Using the toolkit sample application to explore the controls available in the toolkit
- Leveraging toolkit controls in a WinUI project
- Exploring the toolkit's helpers, services, and extensions for Windows developers

By the end of this chapter, you will understand the Windows Community Toolkit and how it can boost your productivity when building Windows apps. You will also know how to incorporate its controls into your WinUI applications.

## Technical requirements

To follow along with the examples in this chapter, the following software is required:

- Windows 10 version 1803 (build 17134) or newer
- Visual Studio 2019 version 16.9 or newer with the following workloads: .NET desktop development and Universal Windows Platform development

The source code for this chapter is available on GitHub at <https://github.com/PacktPublishing/-Learn-WinUI-3.0/tree/master/Chapter09>.

## Introducing the WCT

The WCT was created by Microsoft as an open source collection of controls and other helpers, tools, and services for Windows developers. It is primarily used by UWP and WinUI developers but also adds value for WinForms and WPF developers. In fact, developers can use the toolkit's XAML Islands controls to embed WinUI controls into existing WinForms and WPF applications. The toolkit is available to developers as a set of NuGet packages. There are over a dozen toolkit packages available on NuGet that can be installed independently, depending on the needs of your project. We will explore some of these packages throughout this chapter. Let's start by discussing the history of the WCT.

The toolkit was open sourced from the very beginning, is available on GitHub (<https://github.com/windows-toolkit/WindowsCommunityToolkit>), and is open to community contributions. The documentation is available on Microsoft Docs at <https://docs.microsoft.com/en-us/windows/communitytoolkit/>.

## Origins of the WCT

The WCT was first introduced as the **UWP Community Toolkit** in 2016. As the name implies, it was originally a toolkit solely for UWP developers. The toolkit was created to simplify UWP app development by providing controls and helpers that Windows developers frequently create for their own common libraries. The idea of creating a toolkit for XAML development is not a new one. There have been several other similar projects for other platforms:

- **WPF Toolkit** (<https://github.com/dotnetprojects/wpftoolkit>): A set of WPF open source controls and other components, originally hosted by Microsoft on **CodePlex**.
- **Extended WPF Toolkit** (<https://github.com/xceedsoftware/wpftoolkit>): An open source collection of controls maintained by **Xceed Software**, intended to complement the original WPF Toolkit.
- **XamarinCommunityToolkit** (<https://github.com/xamarin/XamarinCommunityToolkit>): An open source collection of Xamarin controls, animations, behaviors, and effects for Xamarin.Forms.

Microsoft, with help from the open source community, released regular updates to the toolkit, adding new and enhanced components and controls multiple times a year. In spring 2018, shortly before the release of v3.0, they announced the toolkit's new name: Windows Community Toolkit. This renaming signaled the team's intent to embrace all Windows developers moving forward.

WCT 3.0 included a new Microsoft Edge-based `WebView` control – not to be confused with `WebView2`, which we will cover later in this chapter – for WPF and WinForms applications. The release also added code samples to Visual Basic, which is still used in many legacy Win32 codebases.

Another purpose of the toolkit is that you can work on new controls with the hope that some will be integrated into the Windows SDK at a later date (or alternatively, the WinUI libraries). This has happened with several controls over the years since the toolkit's introduction, including the `WebView` control.

Subsequent toolkit releases have continued to add value for both UWP and Win32 developers, and have been fueled by community contributions.

## Reviewing recent toolkit releases

At the time of writing, there have been three more major releases of the WCT since version 3.0 and a preview of version 7, which added WinUI 3.0 support.

In August 2018, WCT 4.0 added a `DataGridView` control, a feature long desired by UWP developers who were familiar with the `DataGridView` control available on the Silverlight and WPF platforms. This was quickly followed by a fall 2018 release of version 5. This release brought two major features to the toolkit:

- **WindowsXamlHost:** This control enabled a single UWP control to be wrapped and hosted within a WPF or WinForms control. Later, the `WindowsXamlHost` would be known as XAML Islands, with the hosting API added to the Windows SDK. Several *Wrapped Controls* were also released, including `InkCanvas`, `MapControl`, and an update to the original `WebView` control.
- **TabView:** Behind `DataGridView`, a rich `TabView` was probably the most requested control not yet available to UWP developers. The WCT `TabView` included support for customizing, closing, and dragging and dropping tabs. `TabView` has also graduated to the WinUI library, becoming available in WinUI 2.2 and later. The WCT team has indicated that `TabView` will be removed from the toolkit in a future release.

A year later, in fall 2019, WCT toolkit 6.0 brought XAML Islands controls to all WinForms, WPF, and C++ Win32 developers, adding support for .NET Core 3 clients. The other major improvement in this release was adding ARM64 development support. In June 2020, the team announced WCT 6.1, as well as previews of versions 7 and 8. Windows Community Toolkit 8.0 supports WinUI 3.0, and the preview numbers of WCT align with the supported WinUI 3 preview numbers. Currently, creating WinUI .NET 5 apps for desktop is supported by WCT. Support for WinUI in UWP projects will be added in later preview releases of WCT 8.0. This support is likely to be available after WinUI in UWP projects supports .NET 5 in a post-3.0 release.

Now that we have covered some background and history of the WCT, we will take a closer look at some of the controls and components currently available in the toolkit.

## Exploring the Windows Community Toolkit Sample App

As we mentioned earlier in this chapter, the **Windows Community Toolkit Sample App** is available from the Microsoft Store (<https://www.microsoft.com/p/windows-community-toolkit-sample-app/9nb1ggh4t1cq>). It can be installed on Windows 10 version 17134 or higher or on Xbox One. Like the XAML Controls Gallery we discussed in *Chapter 5, Exploring WinUI Controls*, the toolkit sample app provides us with an easy way to navigate and explore the contents of the WCT.

## Installing and launching the sample app

Let's get started:

1. Open the Microsoft Store app from the Windows Start menu and enter **windows community** in the **Search** box:

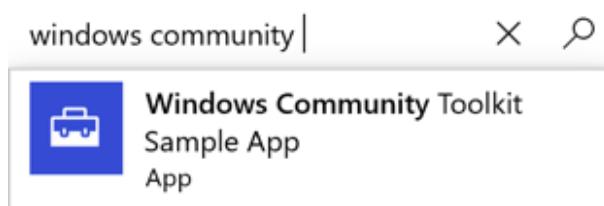


Figure 9.1 – Finding the sample app in the Microsoft Store

2. Select **Windows Community Toolkit Sample App** from the search results and click the **Install** button on the resulting page. After the installation completes, the **Install** button will become a **Launch** button. Open the app from there or the Start menu:

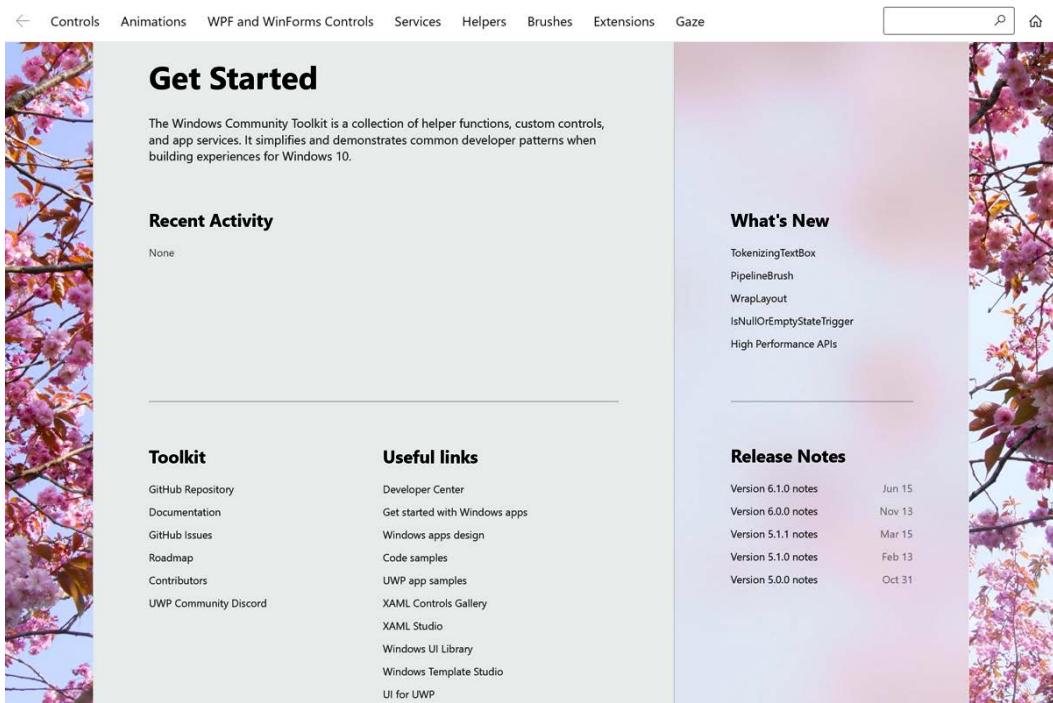


Figure 9.2 – The Windows Community Toolkit Sample App

The app opens on its **About** page, where there are several sections containing helpful information for getting started:

- **Recent Activity:** The app will remember pages you have recently visited to make it easy to get back to them later.
- **Toolkit:** These are links relevant to the WCT, including the GitHub repository, documentation on MS Docs, the GitHub issues page, the product roadmap, open source contributors, and the UWP community **Discord** channel (learn more at <https://uwpcommunity.com/>).
- **Useful links:** A list of links that are not WCT-specific but will be helpful to WinUI, UWP, and other XAML developers.
- **What's New:** A list of recently updated pages in the sample app.
- **Release Notes:** Links to release notes for recent versions of the toolkit.
- **About the App:** Basic version information and links to dependencies.

The controls and other components are divided into eight categories: **Controls**, **Animations**, **WPF and WinForms Controls**, **Services**, **Helpers**, **Brushes**, **Extensions**, and **Gaze**. With so many controls in the toolkit, we will explore just a few of them and leave the rest for you to explore on your own.

## Controls

Click the **Controls** menu item at the top of the app to display the list of controls. This is the largest section of the app, with the controls grouped by category:

- **Developer:** Controls that are useful when building tools for developers (for example, **AlignmentGrid**)
- **Graph:** Controls to display Microsoft Graph data (for example, **PersonView**)
- **Input:** These are custom input controls (for example, **MarkdownTextBlock**)
- **Layout:** Layout panels and related controls (for example, **UniformGrid**)
- **Layout - Items Repeater:** Layout controls that work with an **ItemsRepeater** (for example, **WrapLayout**)

- **Media:** Controls for working with media (for example, **CameraPreview**)
- **Menus and Toolbars:** These are menu and toolbar controls (for example, **TextToolbar**)
- **Status and Info:** Controls for updating the user on progress or status (for example, **RadialProgressBar**):

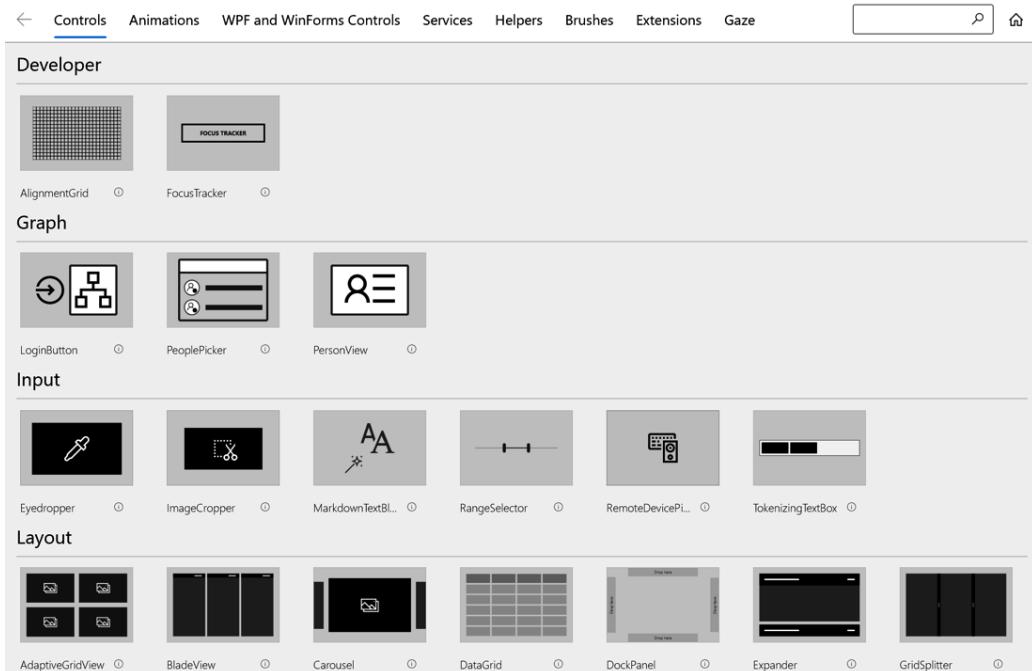
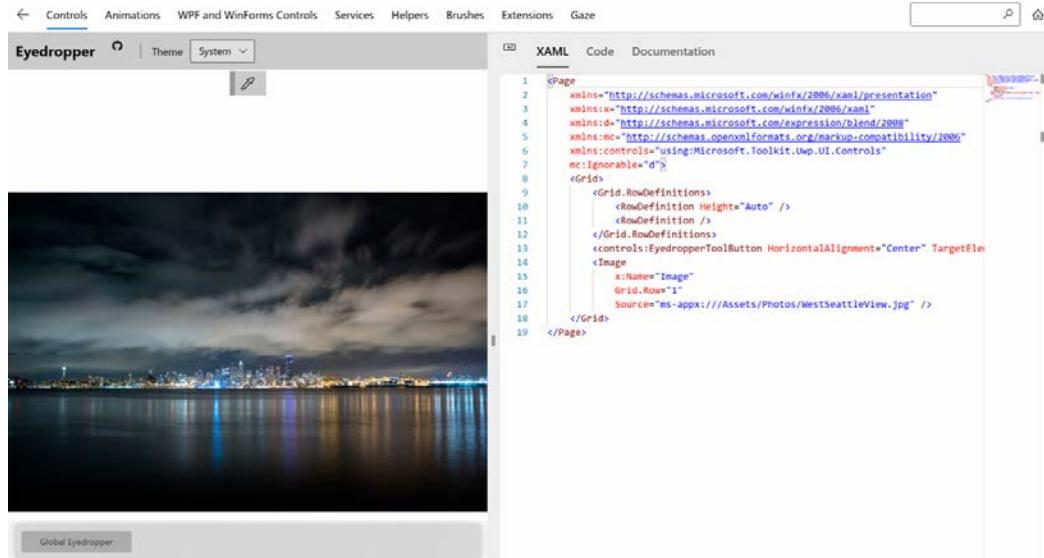


Figure 9.3 – Controls in the Windows Community Toolkit Sample App

Selecting one of these controls will open a page that contains several regions. The left panel is an interactive region where you can interact with the selected control. At the top of this panel, there is a drop-down box where you can select either a **Light**, **Dark**, or **Current** theme, which will update the controls running in the panel.

The right panel contains several tabs, with the number that's available depending on the selected control. The **XAML** tab contains a XAML editor with the markup for the code running in the left panel. You can change the markup here, and your changes will be reflected in the code running in the left panel. The **Code** tab will display sample XAML and C# code for the control. The **Documentation** tab displays the control's documentation from Microsoft Docs:



```
1  <Page
2    xmlns="http://schemas.microsoft.com/winfx/2009/xaml/presentation"
3    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
4    xmlns:d="http://schemas.microsoft.com/expression/blend/2009"
5    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
6    xmlns:controls="using:Microsoft.Toolkit.Uwp.UI.Controls"
7    mc:Ignorable="d">
8
9    <Grid.RowDefinitions>
10      <RowDefinition Height="Auto" />
11      <RowDefinition />
12    </Grid.RowDefinitions>
13    <controls:EyedropperToolButton HorizontalAlignment="Center" TargetFile="Image"
14      <Image
15        x:Name="Image"
16        Grid.Row="1"
17        Source="ms-appx:///Assets/Photos/WestSeattleView.jpg" />
18    </Grid>
19  </Page>
```

Figure 9.4 – Viewing the Eyedropper control in the sample app

Take some time to explore the **Eyedropper** control and **MarkdownTextBlock** in the **Controls** section of the toolkit. Follow these steps:

1. Open the **Eyedropper** control and give it a try. Click the **Eyedropper** button, click an area in the image below, and watch the left-hand side of the **Eyedropper** button change color so that it matches the clicked area.
2. Next, select **Controls | MarkdownTextBlock**. This control's page is similar, but on the left panel, there is a regular **TextBlock** control where markdown can be entered. The markdown in this control is then rendered in **MarkdownTextBlock**, at the bottom of the panel:

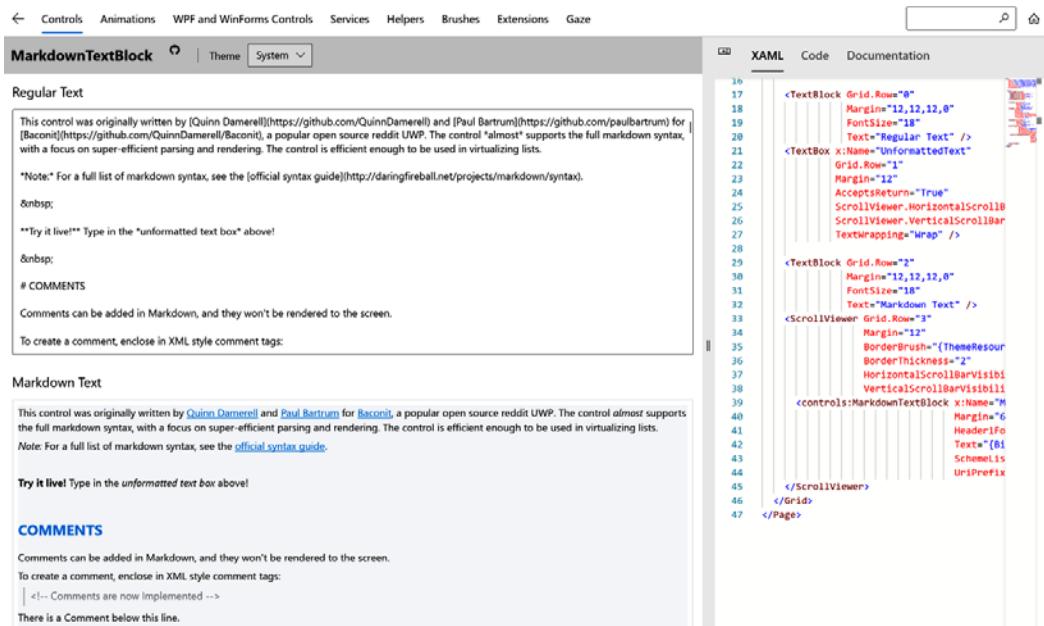


Figure 9.5 – MarkdownTextBlock running in the sample app

3. Switch to the **Code** tab on the right panel to see some sample uses of the control:

```

// How to download .md content from the web and display
it

using (var client = new HttpClient())
{
    try
    {
        MarkdownTextBlockTextblock.Text = await client.
GetStringAsync(feed);
    }
    catch { }
}

// Custom Code Block Renderer
private void MarkdownText_CodeBlockResolving(object
sender, CodeBlockResolvingEventArgs e)
{
    if (e.CodeLanguage == "CUSTOM")
    {
        e.Handled = true;
    }
}

```

```
    e.InlineCollection.Add(new Run { Foreground = new  
        SolidColorBrush(Colors.Red), Text = e.Text, FontWeight =  
        FontWeights.Bold } );  
    }  
}
```

There are many more controls you can explore in this part of the app. You should take some time to find out which might be useful in your next project. Next, we will explore some of the controls for WPF and WinForms developers.

## WPF and WinForms controls

There are currently a handful of controls available for WPF and WinForms developers:

- **InkCanvas**: This brings Windows inking capabilities to Win32 applications.
- **InkToolbar**: This control adds a toolbar that can change how users interact with **InkCanvas** (for example, set the active tool: Pen, Pencil, Eraser, Highlighter, Custom Pen, or Custom Tool).
- **MapControl**: Display an interactive map within your app.
- **MediaPlayerElement**: This allows your Win32 app to display rich media content.
- **WebView**: Hosts an Edge (legacy) web browser control to display online or local web content.
- **WebViewCompatible**: Hosts a web browser compatible with older versions of Windows (prior to Windows 10). Uses Edge (legacy) on Windows 10 and Internet Explorer on older operating systems.
- **WindowsXamlHost**: This control can host a single WinUI or UWP control within a Win32 application.

### Note

There are no interactive demos for these controls in the sample app because it is not a Win32 application.

Selecting one of these controls will open a page containing the documentation from Microsoft Docs, including code samples and API reference information:

The **MapControl** class enables you to display a symbolic or photorealistic map in your Windows Forms or WPF desktop application. This is one of several wrapped Universal Windows Platform controls that are available for Windows Forms and WPF applications as part of a feature called *XAML Islands*. For more information, see [UWP controls in desktop applications \(XAML Islands\)](#).

This control shows rich and customizable map data including road maps, aerial, 3D, views, directions, search results, and traffic. You can also display the user's location, directions, and points of interest.

**Note**  
 This control is currently available as a developer preview for Windows 10, version 1903, and later. Although we encourage you to try out this control in your own prototype code now, we do not recommend that you use it in production code at this time. For more information, see the [XAML Islands feature roadmap](#). If you have feedback about this control, create a new issue in the [Microsoft.Toolkit.Win32 repo](#) and leave your comments there. If you prefer to submit your feedback privately, you can send it to [XamlIslandsFeedback@microsoft.com](mailto:XamlIslandsFeedback@microsoft.com).

**About MapControl**  
 This control wraps an instance of the UWP [Windows.UI.Xaml.ControlsMaps.MapControl](#) class. The WPF version of this control is located in the [Microsoft.Toolkit.Wpf.UI.Controls](#) namespace. The Windows Forms version is located in the [Microsoft.Toolkit.Forms.UITools](#) namespace. You can find additional related types (such as enums and event args classes) in the [Microsoft.Toolkit.Win32.UI.Controls.Interop.WinRT](#) namespace.

**Requirements**  
 Before you can use this control, you must follow [these instructions](#) to configure your project to support XAML Islands.

**Known issues and limitations**  
 See our list of [known issues](#) for WPF and Windows Forms controls in the Windows Community Toolkit repo.

Figure 9.6 – The MapControl documentation for WinForms and WPF

This is the native Windows **MapControl** that's hosted in XAML Islands for use in Win32 applications. The sample code on the page illustrates how to display a geolocation on the map when it has loaded:

```
private async void MapControl_Loaded(object sender,
RoutedEventArgs e)
{
    // Specify a known location.
    BasicGeoposition cityPosition = new BasicGeoposition() {
        Latitude = 47.604, Longitude = -122.329 };
    var cityCenter = new Geopoint(cityPosition);
    // Set the map location.
    await (sender as MapControl).TrySetViewAsync(cityCenter,
12);
}
```

Next, navigate to **WPF and WinForms Controls | MediaPlayerElement** in the app:

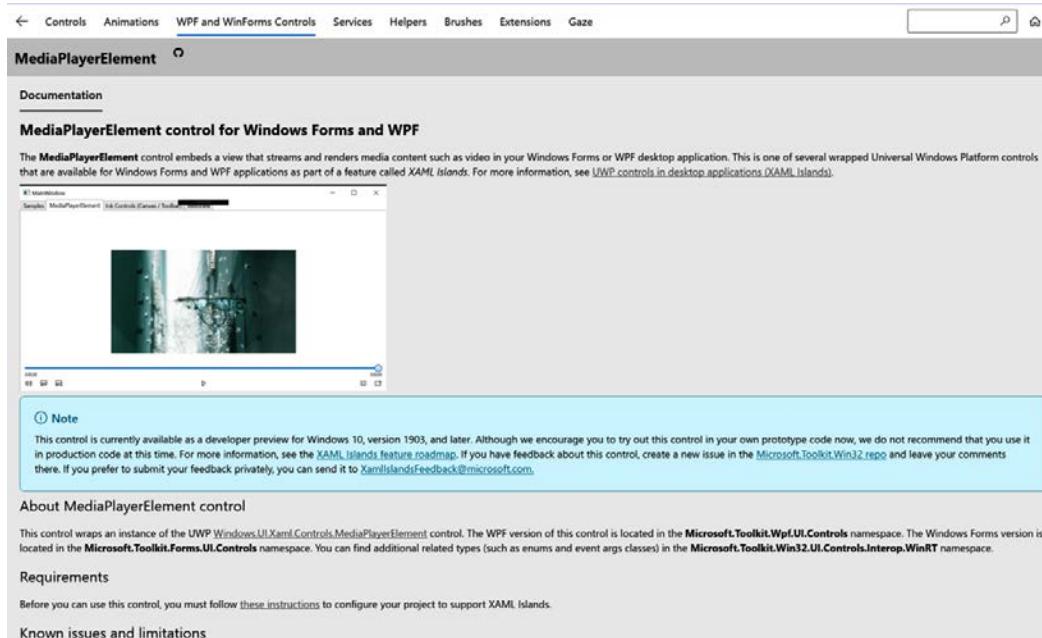


Figure 9.7 – The MediaPlayerElement control page in the sample app

This control embeds the UWP **MediaPlayer** control in XAML Islands, providing Windows media playback capabilities to Win32 applications.

Now that we have explored some of the controls in the sample app, let's try using them in a WinUI project.

## Using controls from the toolkit

We explored a handful of the WCT controls in the sample app in the previous section. Now, it's time to use them in a WinUI project. To demonstrate some of the controls in action, we are going to create a new **WinUI in Desktop** project, built on .NET 5. You should be familiar with this project type from our work in *Chapter 8, Building WinUI Applications with .NET 5*.

### Note

At the time of writing, the WCT controls can only be used with WinUI in Desktop projects built on .NET 5. The compatibility of WCT with WinUI on UWP is not expected until sometime after the WinUI 3.0 RTM release.

## Creating the WinUI in Desktop project

To start our WCT project, you will have to launch Visual Studio and follow these steps:

1. Create a new project. Then, on the **Create a new project** page, enter **WinUI in Desktop** in the search field.
2. Several project types will be displayed, but one of the top results will be **Blank App, Packaged (WinUI in Desktop)**. Select this project template for the language of your choice and click **Next**:

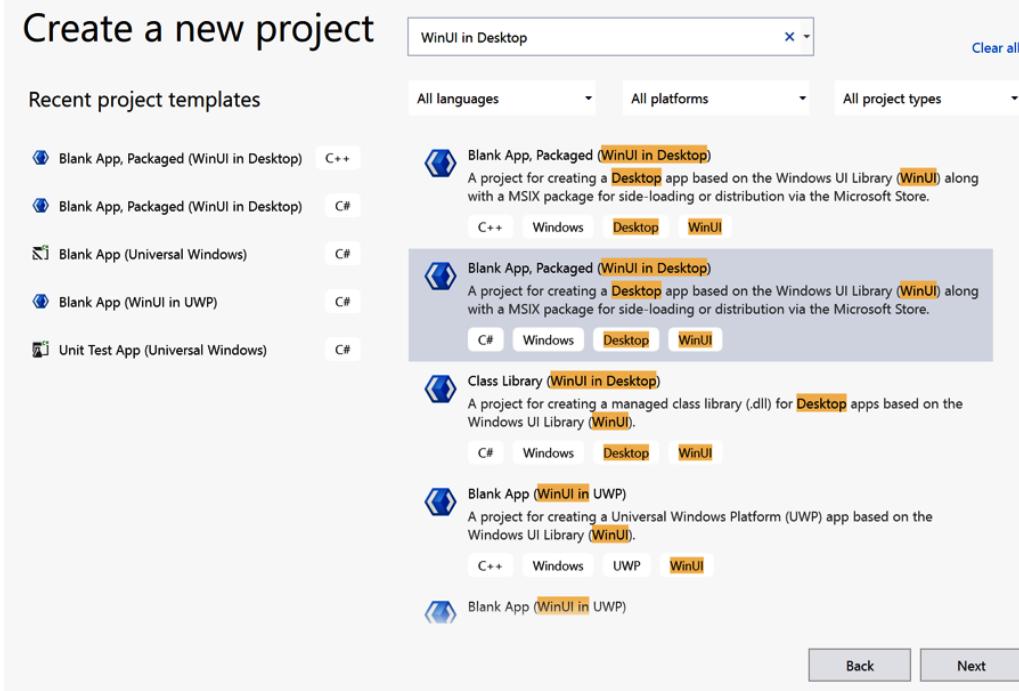


Figure 9.8 – Creating a new WinUI in Desktop project

3. Name the project **HardwareSupplies** and click the **Create** button. You will be prompted to select a **target version** and **minimum version**. Leave the default values as is and select **OK**. The solution will be created and loaded into Visual Studio. The main **HardwareSupplies** project will have some familiar-looking components; that is, **App.xaml** and **MainWindow.xaml** (rather than the usual  **MainPage.xaml**):

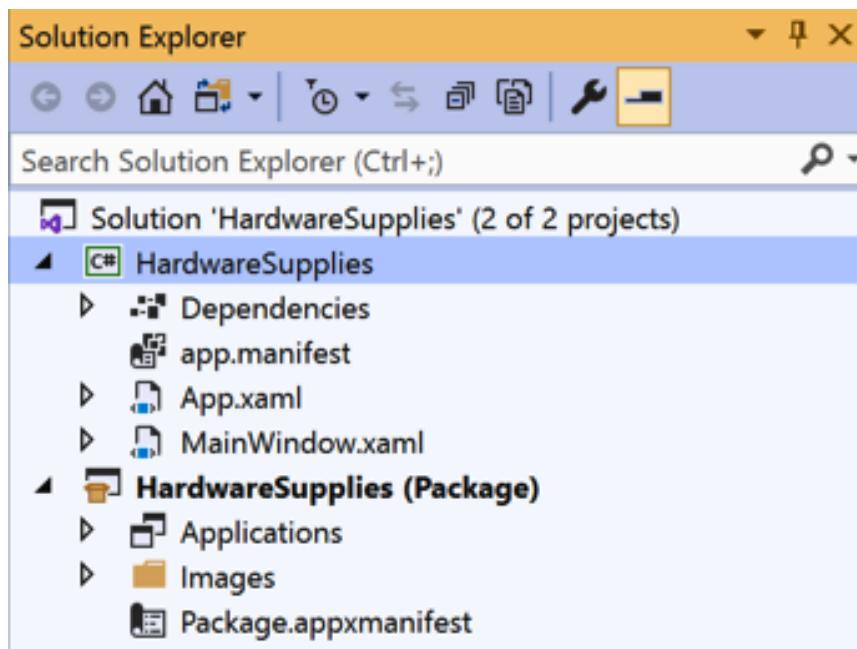


Figure 9.9 – The HardwareSupplies project in Visual Studio Solution Explorer

All you need to know about the second project, **HardwareSupplies (Package)**, for now is that it is for packaging the project into an **MSIX** file for distribution. We will cover this project in more detail in the next chapter, and app distribution will be covered in *Chapter 14, Packaging and Deployment Options and the Microsoft Store*.

4. If you open the **MainWindow.xaml** file, you will see some simple starter markup. There is a **StackPanel** containing a **Button** named **myButton** with **Click Me** as its content:

```
<StackPanel Orientation="Horizontal"
            HorizontalAlignment="Center"
            VerticalAlignment="Center">
    <Button x:Name="myButton"
            Click="myButton_Click">
```

```
    Click Me
  </Button>
</StackPanel>
```

5. The `myButton` variable's `Click` event has a `myButton_Click` event handler in `MainWindow.xaml.cs` that changes the `myButton` variable's content to `Clicked`:

```
private void myButton_Click(object sender,
RoutedEventArgs e)
{
    myButton.Content = "Clicked";
}
```

6. Before we make any changes, run the app and test the button to make sure everything is working as expected:



Figure 9.10 – Running the .NET 5 WinUI app for the first time

Everything is working as expected. Next, we're going to add the WCT package references to the project.

#### Note

At the time of writing, the WCT 8.0 preview NuGet packages are only available through a different package source. You need to open your NuGet options in Visual Studio and add this URL as a new package source: [https://pkgs.dev.azure.com/dotnet/WindowsCommunityToolkit/\\_packaging/WindowsCommunityToolkit-MainLatest/nuget/v3/index.json](https://pkgs.dev.azure.com/dotnet/WindowsCommunityToolkit/_packaging/WindowsCommunityToolkit-MainLatest/nuget/v3/index.json). For more details on adding custom package sources to Visual Studio, visit this Microsoft Docs article: <https://docs.microsoft.com/en-us/azure/devops/artifacts/nuget/consume?view=azure-devops#windows-add-the-feed-to-your-nuget-configuration>.

## Referencing the WCT packages

The primary control we need for the app is a **DataGrid** that displays a list of hardware items. We will also add a **HeaderedContentControl** and a **DropShadowPanel** to get an idea of how those controls can be used. Most of the WCT controls are part of the **Microsoft.Toolkit.Uwp.UI.Controls** package, but **DataGrid** is in the **Microsoft.Toolkit.Uwp.UI.Controls.DataGrid** package. Open **NuGet Package Manager**, search for **Microsoft.Toolkit.Uwp.UI.Controls**, make sure to check the **Include prerelease** checkbox, and add the latest version of these two packages to the project:

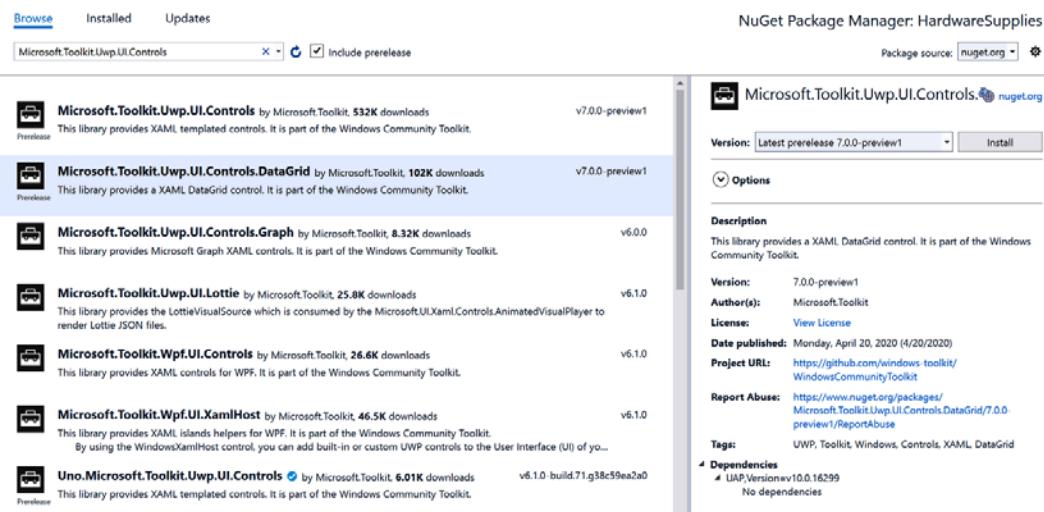


Figure 9.11 – Adding the WCT NuGet packages

After installing these two packages, close the package manager window and compile the project to ensure all the packages are downloaded. Next, we will set up some data for the **DataGrid**.

## Adding data to the DataGrid

The most important part of any **DataGrid** is the data being presented to the user. Before we start building the UI, we're going to build a small inventory of hardware data to display:

1. Start by adding a new class to the **HardwareSupplies** project named **HardwareItem**. The class will have six properties, as shown here:

```
public class HardwareItem
{
```

```
public long id { get; set; }
public string name { get; set; }
public string category { get; set; }
public int quantity { get; set; }
public decimal cost { get; set; }
public decimal price { get; set; }
}
```

2. Next, create a public property named `HardwareItems` and define it as an array of `HardwareItem`:

```
public HardwareItem[] HardwareItems { get; set; }
```

3. Next, open the `MainWindow.xaml.cs` file. In there, create a new method named `PopulateItems`. This method will initialize the `HardwareItems` array and populate it with 12 items:

```
private void PopulateItems()
{
    HardwareItems = new HardwareItem[]
    {
        new HardwareItem { id = 1, name = "Wood Screw",
category = "Screws", cost = 0.02M, price = 0.10M,
quantity = 504 },
        new HardwareItem { id = 2, name = "Sheet Metal
Screw", category = "Screws", cost = 0.03M, price = 0.15M,
quantity = 655 },
        new HardwareItem { id = 3, name = "Drywall
Screw", category = "Screws", cost = 0.02M, price = 0.11M,
quantity = 421 },
        new HardwareItem { id = 4, name = "Galvanized
Nail", category = "Nails", cost = 0.04M, price = 0.16M,
quantity = 5620 },
        new HardwareItem { id = 5, name = "Framing Nail",
category = "Nails", cost = 0.06M, price = 0.20M, quantity
= 12000 },
        new HardwareItem { id = 6, name = "Finishing Nail
2 inch", category = "Nails", cost = 0.02M, price = 0.11M,
quantity = 1405 },
        new HardwareItem { id = 7, name = "Finishing Nail
1 inch", category = "Nails", cost = 0.01M, price = 0.10M,
```

```
        quantity = 1110 },
        new HardwareItem { id = 8, name = "Light Switch
- White", category = "Electrical", cost = 0.25M, price =
1.99M, quantity = 78 },
        new HardwareItem { id = 9, name = "Outlet -
White", category = "Electrical", cost = 0.21M, price =
1.99M, quantity = 56 },
        new HardwareItem { id = 10, name = "Outlet -
Beige", category = "Electrical", cost = 0.21M, price =
1.99M, quantity = 90 },
        new HardwareItem { id = 11, name = "Wire Ties",
category = "Electrical", cost = 0.50M, price = 4.99M,
quantity = 125 },
        new HardwareItem { id = 12, name = "Switch Plate
- White", category = "Electrical", cost = 0.21M, price =
2.49M, quantity = 200 }
    };
}
```

Now, the app has a nice assortment of screws, nails, and electrical items to present in the DataGrid.

4. Finally, call `PopulateItems` at the end of the `MainWindow` constructor:

```
public MainWindow()
{
    this.InitializeComponent();
    PopulateItems();
}
```

The data is ready to go. Let's move on and define the XAML markup for `MainWindow`.

## Adding controls to the MainWindow

The UI for our app will be simple. We want to show the data in a DataGrid with a drop shadow beneath some header text:

1. Start by placing a HeaderedContentControl inside a Grid in MainWindow.xaml. Set the Header attribute to Hardware Inventory. This will display at the top of MainWindow's content. Set Margin to 6 to leave some space around the edges of the control:

```
<Grid>
    <wct:HeaderedContentControl Header="Hardware
        Inventory"
        Margin="6">
        </wct:HeaderedContentControl>
    </Grid>
```

2. Don't forget to add the namespace definition for the WCT controls:

```
<Window
    x:Class="HardwareSupplies.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
    presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/
    xaml"
    xmlns:local="using:HardwareSupplies"
    xmlns:wct="using:Microsoft.Toolkit.Uwp.UI.Controls"
    xmlns:d="http://schemas.microsoft.com/expression/
    blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-
    compatibility/2006"
    mc:Ignorable="d">
```

3. Next, define a DropShadowPanel as the content of HeaderedContentControl. BlurRadius defines the width of the blurred area of the drop shadow. A ShadowOpacity value of 1 indicates that the darkest part of the shadow will be completely opaque. Most of this will be behind the DataGrid. The OffsetX and OffsetY values will shift the drop shadow over and down by 2 pixels. The Color attribute sets the color of the shadow. Setting IsMasked to True creates a more precise shadow but degrades performance. In our case, performance will not be a concern. Finally, we'll set Margin to 6 to leave some space to see the drop shadow:

```
<wct:HeaderedContentControl Header="Hardware Inventory"
    Margin="6"
    x:Name="headerPanel">
    <wct:DropShadowPanel BlurRadius="8"
        ShadowOpacity="1"
        OffsetX="2"
        OffsetY="2"
        Color="Gray"
        IsMasked="True"
        Margin="6">
    </wct:DropShadowPanel>
</wct:HeaderedContentControl>
```

4. Lastly, add DataGrid as a child of DropShadowPanel. The grid will bind to the HardwareItems property we created. The AutoGenerateColumns property will create column headers using the names of the HardwareItem objects' properties. By setting Background to a ThemeResource, the grid will look great for Windows users who use either the Light or Dark theme. If you do not set any background colors, DataGrid will be transparent, and the gray drop shadow will obscure the contents of the grid:

```
<wct:DropShadowPanel BlurRadius="8"
    ShadowOpacity="1"
    OffsetX="2"
    OffsetY="2"
    Color="Gray"
    IsMasked="True"
    Margin="6">
    <wct:DataGrid ItemsSource="{x:Bind HardwareItems}">
```

```

AutoGenerateColumns="True"
Background="{ThemeResource
SystemControlBackgroundAltHighBrush}"/>
</wct:DropShadowPanel>

```

- The app's code is complete. It's time to build and run the app to see how everything looks. Note that running the app will package and install it on Windows using the `HardwareSupplies` (Package) project, which is set as the startup project in the solution:

Hardware Inventory

id	name	category	quantity	cost	price
1	Wood Screw	Screws	504	0.02	0.10
2	Sheet Metal Screw	Screws	655	0.03	0.15
3	Drywall Screw	Screws	421	0.02	0.11
4	Galvanized Nail	Nails	5620	0.04	0.16
5	Framing Nail	Nails	12000	0.06	0.20
6	Finishing Nail 2 inch	Nails	1405	0.02	0.11
7	Finishing Nail 1 inch	Nails	1110	0.01	0.10
8	Light Switch - White	Electrical	78	0.25	1.99
9	Outlet - White	Electrical	56	0.21	1.99
10	Outlet - Beige	Electrical	90	0.21	1.99
11	Wire Ties	Electrical	125	0.50	4.99
12	Switch Plate - White	Electrical	200	0.21	2.49

Figure 9.12 – The HardwareSupplies app running with data

Here, you can see that, with a little bit of code, we have a pretty nice-looking app to display some hardware inventory data. The header text, drop shadow, and rich DataGrid work well together to create our UI.

Let's finish up by looking at some of the other components available in the WCT.

# Exploring the toolkit's helpers, services, and extensions

We have discussed many of the controls in the WCT, but the toolkit contains much more than UI controls. In this section, we will return to the WCT sample app to explore some of the other components available in the toolkit. We'll start with some helper classes.

## Helpers

Next to the controls in the toolkit, the **Helpers** section contains the largest number of components. Like the controls, the helpers are divided into categories in the sample app:

- **Data:** These helpers relate to loading and displaying data. Examples include **ImageCache**, **GroupedObservableCollection**, and **IncrementalLoadingCollection**.
- **Developer:** These are helpers that are useful for developers and include **DispatcherHelper** for updating the UI from a background thread.
- **Notifications:** These helpers provide customized ways of notifying users with Windows notifications and the Start menu. Included are **LiveTile**, **Toast**, and **WeatherLiveTileAndToast**.
- **Parser:** There are two parser helpers included. **MarkdownParser** is used in conjunction with **MarkdownTextBlock**, which we discussed earlier. The other parser is **RssParser**, which parses RSS data from a provided RSS feed and outputs an RSS schema.
- **State Triggers:** There are currently nine state trigger helpers in the toolkit, including **IsNullOrEmptyStateTrigger**, **FullScreenStateTrigger**, and **RegexStateTrigger**.
- **Systems:** The 10 system helpers include **CameraHelper**, **NetworkHelper**, **PrintHelper**, and **ThemeListener**.

Let's take a closer look at a few of the helpers in the toolkit, starting with **ThemeListener**. This is a simple helper class that provides a `ThemeChanged` event. Any time the user's Windows theme changes, the event fires, and your app can take any actions necessary to adapt to the new theme. It is not required to implement this event handler unless your application has custom logic to implement when the theme changes:

```
var listener = new ThemeListener();
listener.ThemeChanged += Listener_ThemeChanged;
...
private void Listener_ThemeChanged(ThemeListener sender)
{
    var theme = sender.CurrentTheme;
    // Update app to handle new theme
}
```

Next, let's see what the `SystemInformation` helper class offers. This is a static class that contains a long list of useful information about the running application and the user's system. These are just a handful of the available properties:

- `ApplicationName`: The application's name
- `ApplicationVersion`: The application version
- `AvailableMemory`: The available system memory
- `Culture`: The current culture set in Windows
- `DeviceFamily`: The name of the user's device family
- `DeviceModel`: The model number of the current device
- `FirstUseTime`: The first time the app was launched
- `IsAppUpdated`: Indicates if this is the first time the app has been run after being updated
- `LaunchCount`: The number of times the app has been launched since a system reset
- `OperatingSystem`: The name of the operating system
- `OperatingSystemVersion`: The operating system version

Finally, the **Toast** helper includes several classes that help create and display toast notifications in Windows. In this toolkit example, `ToastNotificationManager` and `ToastContentBuilder` create a rich calendar notification with options to snooze or dismiss the notification:

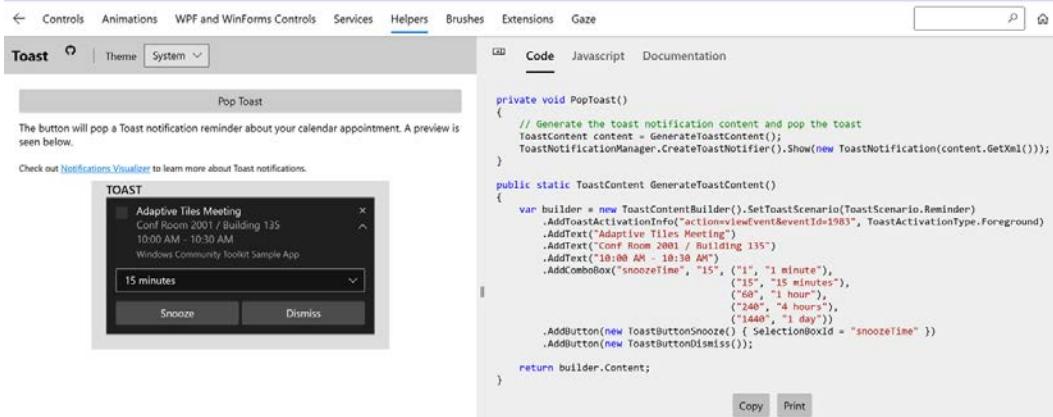


Figure 9.13 – The Toast helper in the WCT sample app

There are many other helpers you can explore in the sample app. Next, we're going to review some of the services in the WCT.

## Services

There are currently seven services showcased in the WCT sample app. It is important to note which of these are dependent on the UWP platform and which are part of the `Microsoft.Toolkit.Service` package. If you try to use the UWP-only features in a WinUI in Desktop project, the application will not compile. This is a **.NET Standard 2.0** package, which means it can be used by any framework implementing .NET Standard 2.0. Some frameworks that implement this standard include .NET Framework version 4.6.1 or later, .NET Core 2.0 or later, and Windows SDK apps targeting version 10.0.16299 or later. You can read more about .NET Standard on Microsoft Docs at <https://docs.microsoft.com/en-us/dotnet/standard/net-standard>. Let's take a look at a few of these services:

- **Facebook Service** (.NET Standard): This service can log into Facebook and fetch and post data to the service.
- **LinkedIn Service** (.NET Standard): The LinkedIn service can retrieve and post information on LinkedIn.

- **Microsoft Graph Service** (UWP only): This service includes `InteractiveProviderBehavior`, which must be added to your app's main page to allow the use of any Graph-enabled XAML controls.
- **Microsoft Translator Service** (.NET Standard): The Translator service enables language translation through **Microsoft Azure Cognitive Services**.
- **OneDrive Service** (.NET Standard): This service can access OneDrive, OneDrive for Business, documents on SharePoint, and Office 365 Groups.
- **Twitter Service** (UWP only): Adds the ability to read data from Twitter and post tweets.
- **Weibo Service** (UWP only): Allows your app to read or publish data on Weibo.

Some of the services have pages containing documentation only, while others have interactive panels where the code can be tested and updated. Let's look at **Microsoft Translator Service**, which has interactive panels:

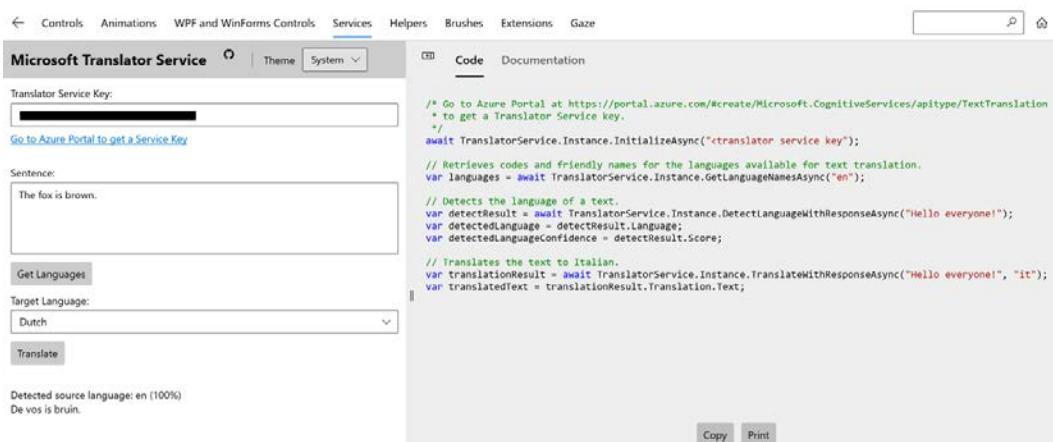


Figure 9.14 – The Microsoft Translator Service

Using the translator service requires an Azure Cognitive Services Text Translation API key. A free Text Translation resource that provides translation for up to 2 million characters per month can be configured at <https://portal.azure.com/#create/Microsoft.CognitiveServices/apitype/TextTranslation>. The code to perform translations with the WCT service is simple and intuitive:

```

await TranslatorService.Instance.InitializeAsync("<insert your
translator service key here>");

// Gets languages info in English
var languages = await TranslatorService.Instance.

```

```
GetLanguageNamesAsync("en");
// Detects the language of a text
var result = await TranslatorService.Instance.
DetectLanguageWithResponseAsync("Hello world!");
var detectedLanguage = result.Language;
var detectedLanguageConfidence = result.Score;
// Translates text to Italian
var translationResult = await TranslatorService.Instance.
TranslateWithResponseAsync("Hello world!", "it");
var translatedText = translationResult.Translation.Text;
```

The translator service and other WCT services can save you a lot of time that can be spent learning the underlying APIs. Next, we will touch on the MVVM library in the WCT.

## MVVM

Version 7.0 of the WCT includes a new MVVM library. The most recent version of the MVVM package, at the time of writing, is **7.0.0** and can be downloaded here:

<https://www.nuget.org/packages/Microsoft.Toolkit.Mvvm/7.0.0.0>.

The **Microsoft.Toolkit.Mvvm** package includes MVVM helpers similar to those we implemented in our `BindableBase` and `RelayCommand` classes in *Chapter 3, MVVM for Maintainability and Testability*, and *Chapter 4, Advanced MVVM Concepts*. This will be a .NET Standard library and will not be specific to WinUI.

Finally, let's review some of the other tools in the **Extensions** area of the WCT sample app.

## Extensions

The **Extensions** menu in the sample app contains several items that add extended properties to WinUI controls and extension methods to other classes. We will review **Mouse Extensions** and **String Extensions** here.

**Mouse Extensions** adds a property to any `FrameworkElement` in order to set the mouse cursor to display when the mouse moves over that element:

```
<Button extensions:Mouse.Cursor="Wait"
Content="Show Wait Cursor" />
<Button extensions:Mouse.Cursor="Hand"
Content="Show Hand Cursor" />
<Button extensions:Mouse.Cursor="UniversalNo"
Content="Show UniversalNo Cursor" />
```

**String Extensions** contains a few extension methods related to string data:

- `IsEmail`: Determines whether a string is a valid email address format.
- `IsDecimal`: Determines whether a string is a decimal value.
- `IsNumeric`: Determines whether a string is a numeric value.
- `IsPhoneNumber`: Determines whether a string contains a valid phone number format.
- `IsCharacterString`: Determines whether a string contains only letters.
- `ToSafeString`: Returns a string from an object.
- `DecodeHtml`: Returns a string with any HTML formatting, tags, comments, scripts, and styles removed.
- `FixHtml`: Similar to `DecodeHtml`, it returns a string with all HTML formatting, comments, scripts, and styles removed.
- `Truncate`: Truncates a string to a specified length, optionally adding an ellipsis.

The `Truncate` extension includes two overloads. This code will truncate the `name` string so that it's no longer than 10 characters. It will truncate the `city` string to seven characters and add an ellipsis to the end of the string to indicate that it was truncated:

```
string name = "Bobby Joe Johnson";
string city = "San Francisco";
name.Truncate(10); // name will be "Bobby Joe "
city.Truncate(7, true); // city will be "San Fra..."
```

I encourage you to explore these extensions, and all the others in the WCT. The sample app is a great way to visually explore the toolkit and get ideas of how to integrate it into your own projects.

## Summary

In this chapter, you learned about the controls, helpers, services, and other components available to WinUI developers in the WCT. You also learned how to leverage the toolkit in WPF and WinForms applications, which will be explored in more detail later in this book. Finally, we installed and used the WCT Sample App to discover the controls and components in the toolkit that we can use in our apps. Adding the WCT packages to your application will provide controls with rich functionality and extensions that save you time.

In the next chapter, we will be working with the XAML Islands controls from the toolkit to show you how you can use WinUI to modernize existing WPF and WinForms applications.

## Questions

1. What was the original name of the Windows Community Toolkit?
2. Which WCT browser control can be used in WPF or WinForms apps that have been deployed to Windows 7?
3. Which WCT control can render markdown output?
4. Which helper in the WCT can manage and group items into an observable collection?
5. What is the name of the project template for running WinUI apps on .NET 5?
6. Which of these does not have a service class in the WCT: LinkedIn, Dropbox, OneDrive, or Twitter?
7. Which helper class in the WCT can raise an event when the current Windows theme changes? (Challenge)
8. Which WCT controls could be leveraged in a WPF app to enable pen input?

# 10

# Modernizing Existing Win32 Applications with XAML Islands

There are many enterprise applications built on WinForms and WPF that are central to operations at companies across the globe. It's not always easy or practical to rewrite an entire application on a new platform. When it comes to migrating legacy Windows applications, there are some incremental migration options available to developers.

By leveraging **XAML Islands** in WinForms and WPF, developers can host WinRT controls in Win32 applications. There are several XAML Islands *wrapped controls* in the **Windows Community Toolkit**, including `WebView2`, `MapControl`, and `InkCanvas`. There is also a generic host control that can be configured to host any other WinRT control.

In this chapter, we will cover the following topics:

- Learning about XAML Islands and why you would want to leverage it in your Win32 applications
- How to add a WinRT control to a WPF project with XAML Islands
- How to add a WinRT Control to a WinForms project with XAML Islands
- How to add the `WebViewCompatible` browser control to a WPF project
- How to add the `WebView2` browser control to a WinForms project
- How the UWP `MapControl` can be added to a WPF project

By the end of this chapter, you will understand how to leverage XAML Islands to enhance existing WPF and WinForms applications. You will also learn how to add controls that have been exposed for use in Win32 applications by the Windows Community Toolkit.

#### Note

At the time of writing, WinUI 3.0 support for XAML Islands is planned to be added sometime in 2021. It will not be supported at RTM. The approach taken with UWP in XAML Islands will work the same with WinUI controls when this support is added. We will be working with controls in the `Windows.UI.Xaml` namespace, but you can replace them with WinUI controls in the `Microsoft.UI.Xaml` namespace later.

You can currently use WinUI 2.x controls in XAML Islands in WPF and WinForms projects if your application is going to be distributed in an MSIX package. For more information, see the *Limitations and Workarounds* section of the XAML Islands docs: <https://docs.microsoft.com/en-us/windows/apps/desktop/modernize/xaml-islands>.

## Technical requirements

To follow along with the examples in this chapter, the following software is required:

- Windows 10 version 1903 (build 18362) or newer.
- Visual Studio 2019 version 16.9 or newer with the following workloads: .NET Desktop Development and Universal Windows Platform Development.
- To create WinUI in Desktop projects, you must also install the latest .NET 5 SDK.

The source code for this chapter is available on GitHub at <https://github.com/PacktPublishing/-Learn-WinUI-3.0/tree/master/Chapter10>.

## What is XAML Islands?

XAML Islands is a layer of technology from Microsoft that ships as part of the Windows Community Toolkit. It allows your WPF, WinForms, and C++ Win32 applications to embed a UWP control in an interop layer on your application's windows. We touched on XAML Islands when we reviewed the controls in the Windows Community Toolkit in *Chapter 9, Enhancing Applications with the Windows Community Toolkit*.

The WCT includes a `WindowsXamlHost` control that can embed any first-party UWP control from Microsoft, as well as any third-party or custom UWP control. Using this control in a WPF or WinForms application is only supported in projects targeting .NET Core 3.x and later.

The other option with XAML Islands is to use one of the *wrapped controls* provided by the WCT. There are a handful of UWP controls that are already wrapped inside a host control and distributed through the toolkit. Providing these controls pre-wrapped in a XAML Islands host saves developers the time and effort of hosting these types of controls themselves. The wrapped controls can be used with .NET Core and .NET Framework WPF and WinForms apps. These are the wrapped controls that are currently available:

- `InkCanvas`: Provides a surface for user input with Windows inking interaction
- `InkToolbar`: Provides a toolbar to be used in conjunction with `InkCanvas`
- `MediaPlayerElement`: Allows embedded streaming content to be displayed in your application
- `MapControl`: Embeds a symbolic or photo-realistic map that users can use to interact in your application
- `WebView`: The legacy Edge-based web browser control only available on Windows 10 (succeeded by `WebView2` in WinUI)
- `WebViewCompatible`: Provides a browser control compatible with more versions of Windows, including Windows 8.x and Windows 7

These controls are included in version 6.0.0 or later of one of these NuGet packages in the WCT:

- `Microsoft.Toolkit.Wpf.UI.Controls`: Contains the wrapped controls for WPF
- `Microsoft.Toolkit.Wpf.UI.Controls.WebView`: Contains the `WebView` and `WebViewCompatible` controls for WPF

- `Microsoft.Toolkit.Wpf.UI.XamlHost`: Contains the `WindowsXamlHost` control for WPF
- `Microsoft.Toolkit.Forms.UI.Controls`: Contains the wrapped controls for WinForms
- `Microsoft.Toolkit.Forms.UI.Controls.WebView`: Contains the `WebView` and `WebViewCompatible` controls for WinForms
- `Microsoft.Toolkit.Forms.UI.XamlHost`: Contains the `WindowsXamlHost` control for WinForms

As we noted previously, the wrapped controls are included in each of the `Controls` packages. Each of these is dependent on the corresponding `XamlHost` package that uses `WindowsXamlHost` internally. If your project will be using wrapped controls and other hosted UWP controls, you should add one of these packages. The `XamlHost` package only needs to be explicitly added to your project if you are using one of the `WindowsXamlHost` controls.

This diagram from Microsoft Docs (available at <https://docs.microsoft.com/en-us/windows/apps/desktop/modernize/xaml-islands>) illustrates how the `WebView` and wrapped controls interoperate with the components in the Windows SDK, which are displayed below the **OS Boundary** line. Here, you can see that the wrapped `WebView` controls do not have the same internal dependency on the `XamlHost` controls as the other wrapped controls. The `WebView` controls operate over a different set of Windows APIs than the rest of the wrapped controls. All of this is obfuscated from the consuming project:

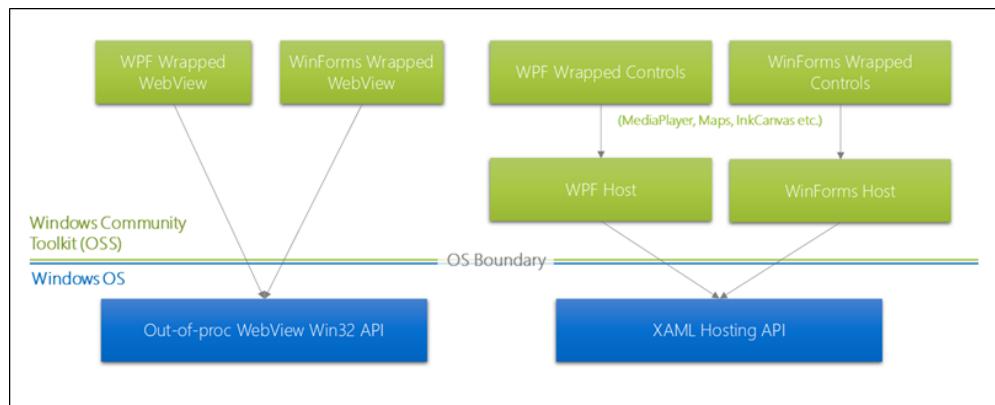


Figure 10.1 – XAML Islands architecture

Now that you have some understanding of what is available in XAML Islands, let's start working with the `WindowsXamlHost` control in a WinForms project.

# Modernizing a WinForms application with XAML Islands

It's time to build a simple WinForms app that will host a custom UWP user control from another project. We will create a UWP project containing a simple input form and add it to a window in a .NET 5 WinForms project. You can use an earlier version of .NET Core if you like, but we will use .NET 5.0 in this example.

## Creating a shared class library project

We will start off by creating the UWP class library. Class libraries help promote code reuse and encapsulate UI appearance and behavior. A NuGet repository is a great way to share code within your company and the OSS community. You can find out more information about NuGet hosting options on Microsoft Docs at <https://docs.microsoft.com/en-us/nuget/hosting-packages/overview>.

### Note

We are creating a UWP class library because XAML Islands does not support WinUI 3.0 projects yet. Once this support is available, you will be able to follow this same technique with a WinUI control library project.

Let's open Visual Studio and step through creating a class library that can be shared across WPF and WinForms:

1. Start by creating a **Blank App (Universal Windows)** project in the language of your choice and click **Next**.
2. Name the project `XamlIslandsSample.UwpApp`, enter `XamlIslandsSample` for **Solution name**, and click **Create**.
3. Make sure both **Target version** and **Minimum version** are set to **Windows 10, version 1903** (currently the minimum supported version) or later and click **OK** to generate the new solution in Visual Studio.
4. Open **NuGet Package Manager** and install `Microsoft.Toolkit.Win32.UI.XamlApplication` version 6.0.0 or later.

5. Open `App.xaml` and replace its contents with the following XAML. By replacing `Application` with `XamlApplication`, you are adding support for the UWP control so that it can be hosted in a Win32 process:

```
<xaml:XamlApplication
    x:Class="XamlIslandsSample.UwpApp.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/pr
    esentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:xaml="using:Microsoft.Toolkit.Win32.UI.XamlHost"
    xmlns:local="using:XamlIslandsSample.UwpApp">
</xaml:XamlApplication>
```

6. Replace the code for the `App` class in `MainPage.xaml.cs` with this code:

```
public sealed partial class App :
    Microsoft.Toolkit.Win32.UI.XamlHost.XamlApplication
{
    public App()
    {
        this.Initialize();
    }
}
```

7. Remove **MainPage.xaml** from the project in **Solution Explorer** and build the project to ensure it compiles successfully.
8. Add a new class named `Entry.cs`. Open the class and update it so that it has the following properties for `FirstName`, `LastName`, `InterviewDate`, and `Accepted`:

```
public class Entry
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime InterviewDate { get; set; }
    public bool Accepted { get; set; }
}
```

9. Next, right-click on the project and select **Add | New Item**.
10. On the **Add New Item** dialog, select **User Control**, enter **EntryForm.xaml** as its **Name**, and click **Add**.
11. Add the following properties to the **Grid** control to create some spacing around it and within the columns and rows:

```
<Grid ColumnSpacing="4" RowSpacing="4" Margin="20">
```

12. Add two columns and five rows to the **Grid** control in **EntryForm.xaml**:

```
<Grid.ColumnDefinitions>
  <ColumnDefinition Width="*"/>
  <ColumnDefinition Width="2*"/>
</Grid.ColumnDefinitions>
<Grid.RowDefinitions>
  <RowDefinition Height="Auto" />
  <RowDefinition Height="Auto" />
  <RowDefinition Height="Auto" />
  <RowDefinition Height="Auto" />
  <RowDefinition />
</Grid.RowDefinitions>
```

13. Add the following markup after the **Grid.RowDefinitions** block to create the data input rows:

```
<TextBlock Text="First name:"/>
<TextBox x:Name="firstNameTextBox"
  Grid.Column="1"/>
<TextBlock Text="Last name:"
  Grid.Row="1"/>
<TextBox x:Name="lastNameTextBox"
  Grid.Row="1" Grid.Column="1"/>
<TextBlock Text="Date of interview:"
  Grid.Row="2"/>
<DatePicker x:Name="interviewDatePicker"
  Grid.Row="2" Grid.Column="1"/>
<TextBlock Text="Accepted:"
  Grid.Row="3"/>
```

```
<ComboBox x:Name="acceptedComboBox"
    Grid.Row="3" Grid.Column="1">
    <x:String>Yes</x:String>
    <x:String>No</x:String>
</ComboBox>
```

14. The last thing we will need on the form are buttons to save or clear the entry:

```
<StackPanel Grid.Row="4" Grid.ColumnSpan="2"
    Orientation="Horizontal"
    HorizontalAlignment="Right">
    <TextBlock x:Name="quantitySavedTextBlock"
        Text="{x:Bind QuantitySaved}"
        Margin="4"/>
    <TextBlock Text="items saved." Margin="4"/>
    <Button Content="Save" x:Name="saveButton"
        Click="saveButton_Click"
        VerticalAlignment="Top" Margin="4"/>
    <Button Content="Clear" x:Name="clearButton"
        Click="clearButton_Click"
        VerticalAlignment="Top" Margin="4"/>
</StackPanel>
```

Notice that the buttons will invoke methods in the code-behind file to when they are clicked, and that there is a label to display how many records have been saved that uses `x:Bind` to get that value. We will create these members in the next step.

15. Open `EntryForm.xaml.cs` and add a `QuantitySaved` property and a private member named `_entries`:

```
private readonly IList<Entry> _entries = new
List<Entry>();
public int QuantitySaved { get; set; } = 0;
```

16. Update the `EntryForm` class so that it implements `INotifyPropertyChanged`:

```
public sealed partial class EntryForm : UserControl,
    INotifyPropertyChanged
{
    private readonly IList<Entry> _entries = new
```

```
    List<Entry>();  
  
    public event PropertyChangedEventHandler  
    PropertyChanged;  
    ...  
}
```

17. Next, add the two click event handlers for **Save** and **Clear**. These call some private methods that will be created in the next step:

```
private void saveButton_Click(object sender,  
    RoutedEventArgs e)  
{  
    SaveEntry();  
    ClearFields();  
}  
private void clearButton_Click(object sender,  
    RoutedEventArgs e)  
{  
    ClearFields();  
}
```

18. Finally, create the private methods that will save the current form entry and clear the form:

```
private void ClearFields()  
{  
    firstNameTextBox.Text = "";  
    lastNameTextBox.Text = "";  
    interviewDatePicker.SelectedDate = null;  
    acceptedComboBox.SelectedIndex = -1;  
}  
private void SaveEntry()  
{  
    var entry = new Entry  
    {  
        FirstName = firstNameTextBox.Text,  
        Lastname = lastNameTextBox.Text,  
    }
```

```
        InterviewDate =
            interviewDatePicker.SelectedDate.HasValue
                ? interviewDatePicker.SelectedDate.Value.
        DateTime
                : DateTime.MinValue,
        Accepted =
            acceptedComboBox.SelectedValue.ToString
            () == "Yes"
        } ;
        _entries.Add(entry);
        QuantitySaved++;
        PropertyChanged?.Invoke(this, new
        PropertyChangedEventArgs(nameof(QuantitySaved))
        );
    }
}
```

Our shared class library with the user entry form is ready to go. This will be added to a XAML Islands host in the WinForms project and to a WPF project that will be created in the next section.

## Creating the WinForms host project

To demonstrate how XAML Islands can be used to modernize your existing WPF and WinForms applications, we need to create WPF and WinForms projects that will host our shared UWP control. In your own existing solutions, you will be referencing the UWP control library in your WPF or WinForms project. This can be done as part of a phased modernization or migration plan.

Follow these steps to create the WinForms project:

1. To start, right-click the **XamlIslandsSample** solution and select **Add | New Project**.
2. In the **Search** field of the **Add a new project** window, search for **windows forms** and select the **Windows Forms App (.NET Core)** template in the language of your choice.
3. Click **Next**, name the project **XamlIslandsSample.WinFormsCore**, and click **Create**.
4. Right-click the WinForms project and select **Manage NuGet Packages**. In **NuGet Package Manager**, search for and add **Microsoft.Toolkit.Forms.UI.XamlHost** version 6.0.0 or later to your project.

5. Right-click on the **Dependencies** node of your WinForms project and select **Add project reference**. From the **Reference Manager** dialog, add a reference to the UWP project and click **OK**.
6. Open **Form1.cs** and change the **ClientSize** and **Text** properties of the form in the constructor that immediately follows the call to **InitializeComponent**:

```
private Form1()
{
    InitializeComponent();
    this.SuspendLayout();
    this.ClientSize = new System.Drawing.Size(1200,
    768);
    this.Text = "Host Form";
    this.ResumeLayout(true);
}
```

7. Update the **Form1** constructor to create an instance of our **EntryForm** and add it as a **Child** to an instance of **WindowsXamlHost**.
8. Set the **Name** and **Location** properties of **WindowsXamlHost** and add it to the **Controls** collection on the form:

```
public Form1()
{
    InitializeComponent();
    this.SuspendLayout();
    this.ClientSize = new System.Drawing.Size(1200,
    768);
    this.Text = "Host Form";
    this.ResumeLayout(true);
    var myHostControl = new
        Microsoft.Toolkit.Forms.UI.XamlHost.
        WindowsXamlHost();
    var entryForm =
        Microsoft.Toolkit.Win32.UI.XamlHost.
        UWPTypeFactory.CreateXamlContentByType(
        "XamlIslandsSample.UwpApp.EntryForm") as
        UwpApp.EntryForm;
```

```
myHostControl.Name = "myUwpAppHostControl";
myHostControl.Child = entryForm;
myHostControl.Location = new
    System.Drawing.Point(0, 0);
myHostControl.Size = Size;
Controls.Add(myHostControl);
}
```

9. Before running the app for the first time, right-click the solution file, select **Configuration Manager**, and ensure that, in the current configuration, the WinForms project builds in **x86** mode. Building WindowsXamlHost host projects as **AnyCPU** is not supported:

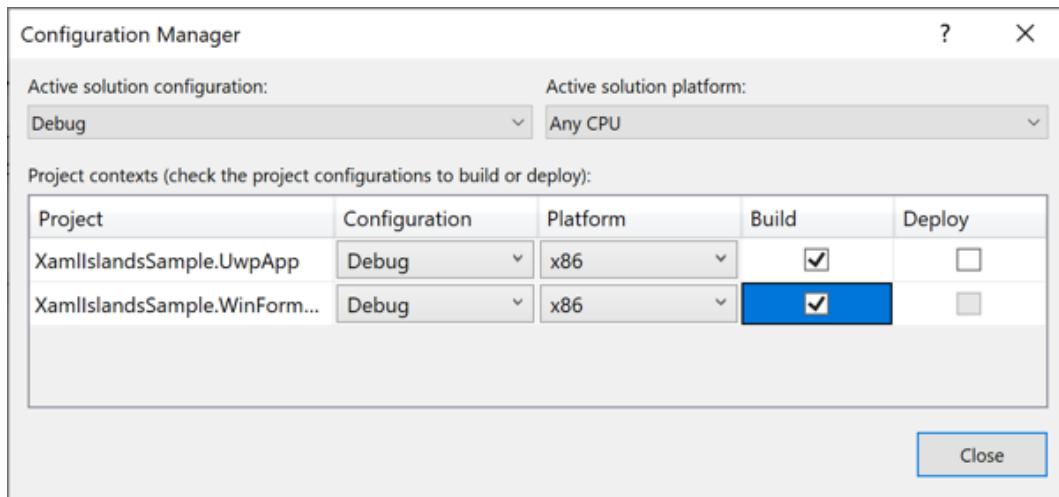


Figure 10.2 – Visual Studio's Configuration Manager window

10. Finally, ensure the WinForms project is set as the start up project in the solution, run the app, and test the entry form to make sure everything works as expected. You should be able to enter some values for each field, and when you click **Save**, the counter will increment the number of **items saved** on the screen:

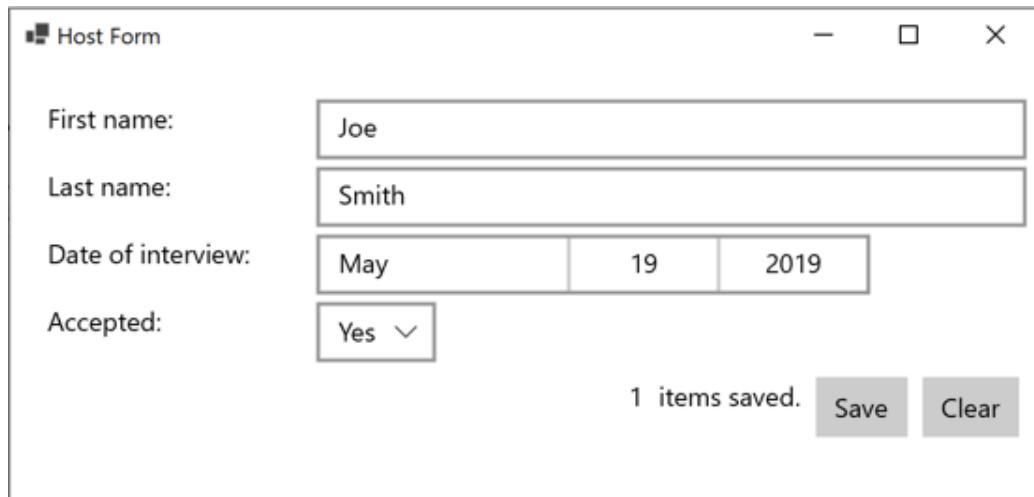


Figure 10.3 – The WinForms window hosting a UWP control

You'll also notice that the same modern-looking controls you would see in a UWP or WinUI application appear in the WinForms window. You can use the tools and techniques we covered in *Chapter 7, Fluent Design System for Windows Applications*, to override the styles in your control or its `App.xaml` file to create beautiful, modern user interfaces inside a WinForms application. `DatePicker` appears like other Windows fluent controls:

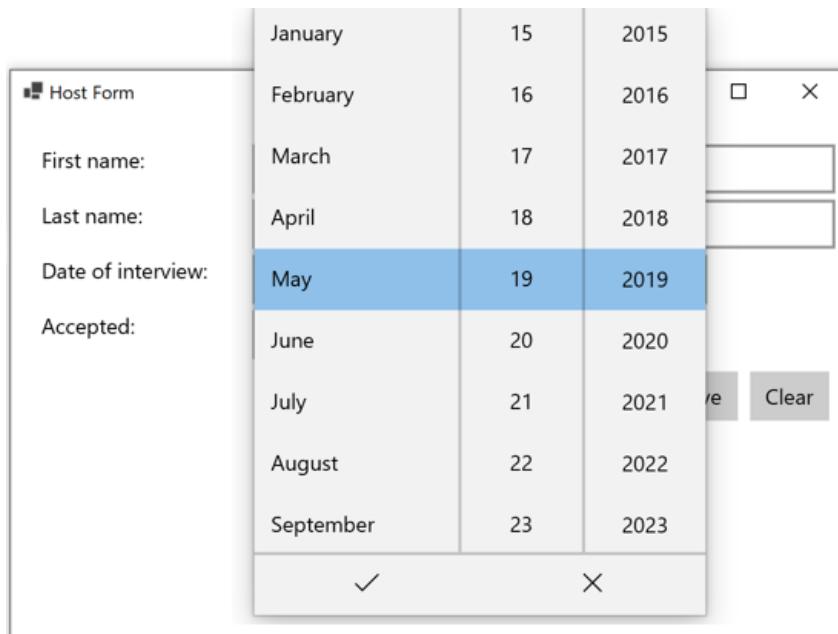


Figure 10.4 – Using a UWP DatePicker in the WinForms host

This data entry form could either be used on a dialog launched by another existing WinForms window, or it could be incorporated into a window containing other WinForms controls. Now that we have the WinForms host working as expected, let's turn our attention to creating a WPF host for the same user control.

## Modernizing a WPF application with XAML Islands

Adding a XAML Island to a WPF project is much like adding one to WinForms. In this section, we will add the `EntryForm` control to a new .NET 5 WPF project. This process will also work with your existing .NET Core and .NET Framework WPF applications, provided they meet the minimum version requirements discussed earlier in this chapter. Let's get started:

1. Start by right-clicking the **XamlIslandsSample** solution and selecting **Add | New Project**.

**Note**

If you didn't complete the previous section with the WinForms example, you can go back and follow the *Creating a shared class library project* section to create your starter solution with the `UwpApp` project.

2. From the **Add a new project** dialog, select the **WPF App (.NET Core)** template in the language of your choice and click **Next**.
3. Name the project `XamlIslandsSample.WpfCore` and click **Create**. The new project will be added to **Solution Explorer**.
4. Open **NuGet Package Manager** for the new WPF project, search for `Microsoft.Toolkit.Wpf.UI.XamlHost`, and add the package with that name and *version 6.0.0 or later* to your project.
5. Configure the WPF project to target **x86** because the **Any CPU** configuration is not supported in projects with XAML Islands. Right-click the solution and select **Configuration Manager**. For the WPF project, select **New** under **Active solution platform**. On the dialog that appears, choose **x86**, make sure the project is set to **Build** and **Deploy**, click **OK**, and close the open dialogs.

6. Add a *project reference* to the **XamlIslandsSample.UwpApp** project and build the solution to make sure everything compiles successfully.
7. Open **MainWindows.xaml** in the WPF project and add a namespace reference to `Microsoft.Toolkit.Wpf.UI.XamlHost`. Also, change the `Title`, `Height`, and `Width` properties so that they match the following values:

```
<Window x:Class="XamlIslandsSample.WpfCore.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/
        xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/
        xaml"
        xmlns:d="http://schemas.microsoft.com/expression/
        blend
        /2008"
        xmlns:mc="http://schemas.openxmlformats.org/
        markup-
        compatibility/2006"
        xmlns:local="clr-
        namespace:XamlIslandsSample.WpfCore"
        xmlns:xaml="clr-
        namespace:Microsoft.Toolkit.Wpf.UI.XamlHost
        ;assembly=Microsoft.Toolkit.Wpf.UI.
        XamlHost"
        mc:Ignorable="d"
        Title="WPF Host" Height="600" Width="1024">
```

8. Add two rows to Grid. The first row will contain the WindowsXamlHost control with its InitialTypeName assigned to the EntryForm control in our UWP project. The second row will contain a Button that we can click to get the current number of entries that have been submitted so far:

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition Height="*"/>
    <RowDefinition Height="Auto"/>
  </Grid.RowDefinitions>
  <xaml:WindowsXamlHost x:Name="xamlHost"
    InitialTypeName="XamlIslandsSample.UwpApp.EntryForm"/>
  <Button x:Name="quantityButton"
    Grid.Row="1"
    Content="Get Total Entries"
    Click="quantityButton_Click"/>
</Grid>
```

9. In **MainWindow.xaml.cs**, add the `quantityButton_Click` event handler. In this method, we will use the `GetUwpInternalObject` method on `xamlHost` to get a reference to the `EntryForm` control. Clicking this button will simply show a `MessageBox` with the total entries submitted so far. This value comes from the public `QuantitySaved` property on `EntryForm`:

```
private void quantityButton_Click(object sender,
  RoutedEventArgs e)
{
  var entryControl = xamlHost.GetUwpInternalObject()
    as UwpApp.EntryForm;
  MessageBox.Show("Total entries: " +
    entryControl.QuantitySaved);
}
```

10. Right-click the WPF project in **Solution Explorer** and select **Set as Startup Project**. Now, run the project, enter a record into the form, save it, and click **Get Total Entries**. You should see a message appear with **Total entries: 1** displayed:

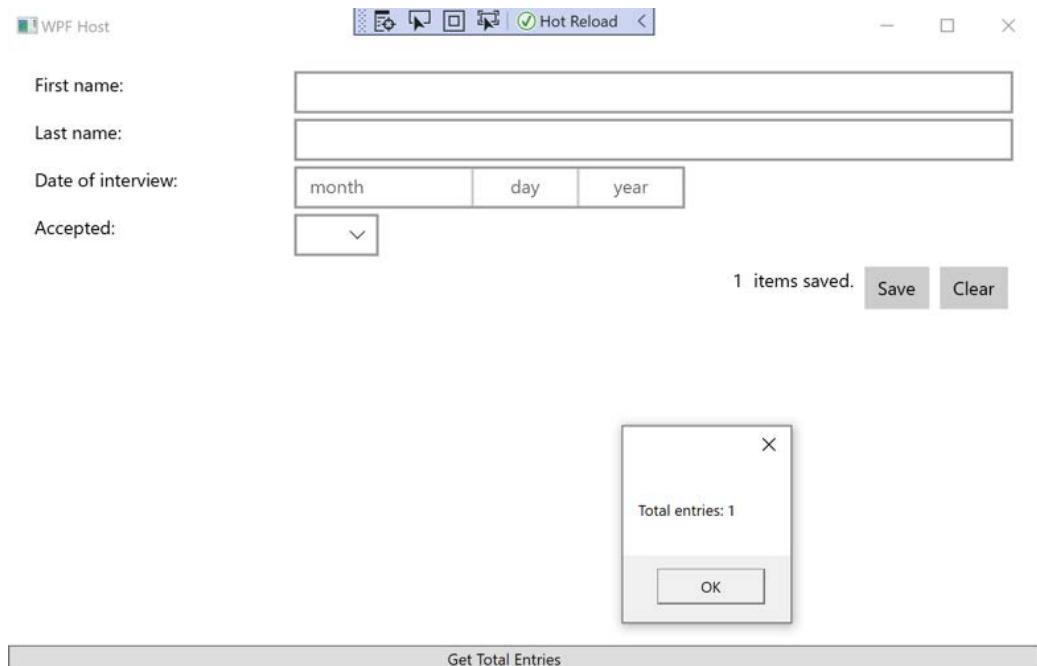


Figure 10.5 – Running the WPF host application

This is a simple example of interop through XAML Islands. A common way to leverage `WindowsXamlHost` would be to set the `DataContext` property or a public `ViewModel` property on the UWP child, which will data bind to a `ViewModel` from the host project.

Let's continue exploring XAML Islands by looking at some of the *wrapped controls* provided by the toolkit.

## Using the UWP MapControl in WPF

In *Chapter 9, Enhancing Applications with the Windows Community Toolkit*, we explored `MapControl` for WPF and WinForms in the WCT sample application. In this section, we are going to add this `MapControl` to a WPF project that allows users to set the current position of the interactive map to several points of interest throughout the world. Let's get started:

1. Add a **WPF Application** or **WPF App (.NET Framework)** project to the solution and name it `XamlIslandsSample.WpfMaps`.
2. Open **NuGet Package Manager** and add the **Microsoft.Toolkit.Wpf.UI.Controls** package to the new project.

3. Before we can begin working with maps, we will need a *map authentication key* from **Bing Maps**. There is no Azure Maps SDK for Windows at the time of writing. The API key must currently be generated in the Bing Maps portal. Get started at <https://www.bingmapsportal.com/>. If you don't have a developer account yet, create one. If you already have one, you can sign in now. Once you have signed into your account, navigate to **My Account | My Keys**. Create a new key by entering the required fields: **Application name**, **Key type**, and **Application type** (this must be set to **Windows Application**):

**My keys**

**Create key**

**Application name** \*

**Application URL**

**Key type** \*

What's This

Basic

**Application type** \*

Windows Application

**Create** **Cancel**

\* Required field

To create Education, Broadcast or Not-for-Profit keys, please contact the Bing Maps account team at [mpnet@microsoft.com](mailto:mpnet@microsoft.com).

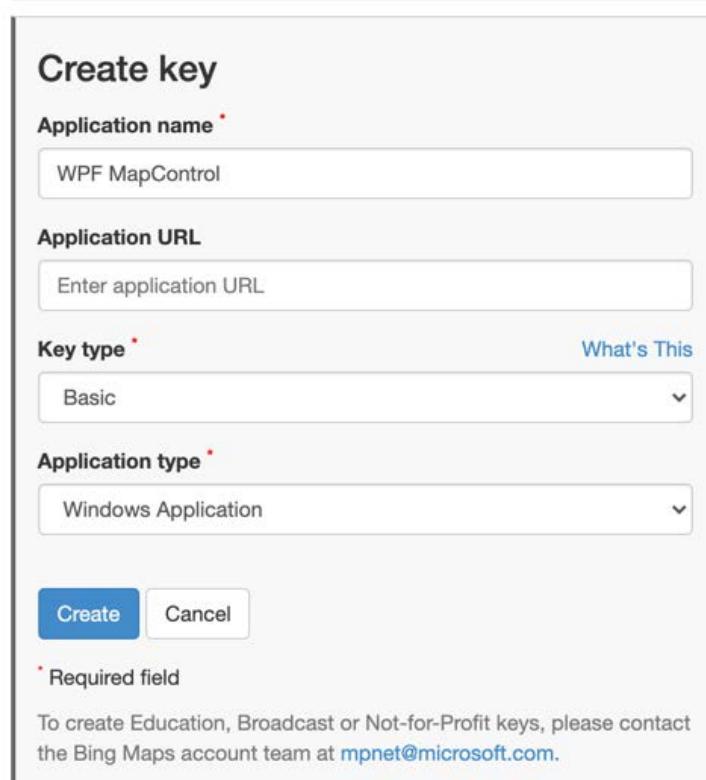


Figure 10.6 – Creating a new maps key

4. Once the key has been created, click the **Copy Key** link to copy your key to the clipboard. Keep this somewhere safe, as we will need it in the steps ahead.

5. Open **MainWindow.xaml** and add a namespace reference to **Microsoft.Toolkit.Wpf.UI.Controls**:

```
xmlns:controls="clr-
    namespace:Microsoft.Toolkit.Wpf.UI.Controls
    assembly:Microsoft.Toolkit.Wpf.UI.Controls"
```

6. Add a **MapControl** to **Grid** and enter your key as the **MapServiceToken** property:

```
<controls:MapControl x:Name="mapControl"
    ZoomInteractionMode="GestureAndControl"
    TiltInteractionMode="GestureAndControl"
    MapServiceToken="EnterYourAuthenticationKeyHere"
    />
```

7. Set **WpfMaps** as the start up project, set the project's run configuration to **x86**, and run it to make sure the project compiles and runs without errors and displays **MapControl** as expected.

**Note**

If you are running an alpha or beta Insiders build of Windows 10, running the app may result in a Catastrophic Failure exception. If this happens, follow the instructions in this **Microsoft Tech Community** article: <https://techcommunity.microsoft.com/t5/windows-dev-appconsult/using-xaml-islands-on-windows-10-19h1-fixing-the-quot/ba-p/376330>.

8. Next, modify **MainWindow.xaml** to add a row of buttons. Each button will set the map's location to a famous landmark. The **Click** event for all the buttons will invoke an event handler named **location\_click**:

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="*"/>
    </Grid.RowDefinitions>
    <StackPanel Orientation="Horizontal"
        HorizontalAlignment="Center">
        <Button x:Name="parisButton" Margin="4"
```

```
        Content="Eiffel Tower"
        Click="location_Click"/>
    <Button x:Name="londonButton" Margin="4"
        Content="London Eye"
        Click="location_Click"/>
    <Button x:Name="newYorkButton" Margin="4"
        Content="Empire State Building"
        Click="location_Click"/>
    <Button x:Name="chinaButton" Margin="4"
        Content="Great Wall"
        Click="location_Click"/>
</StackPanel>
...
</Grid>
```

9. Update MapControl to put it on the second Grid.Row and set LandmarksVisible to True:

```
<controls:MapControl x:Name="mapControl"
    Grid.Row="1"

    ZoomInteractionMode="GestureAndControl"
    TiltInteractionMode="GestureAndControl"
    LandmarksVisible="True"
    MapServiceToken="
        EnterYourAuthenticationKeyHere"
    />
```

10. Finally, in **MainWindows.xaml.cs**, create the location\_Click event handler. This method will create a BasicGeoposition based on the name of the button that's clicked by the user. This will be used to create a Geopoint to pass to the TrySetViewAsync method of MapControl, along with a zoom level of 12:

```
private async void location_Click(object sender,
    RoutedEventArgs e)
{
    var button = e.Source as Button;
    if (button == null) return;
```

```
BasicGeoposition position = new BasicGeoposition()
    { Latitude = 0, Longitude = 0 };
// Specify the location based on the button clicked
switch (button.Name)
{
    case nameof(parisButton):
        position = new BasicGeoposition() {
            Latitude = 48.858242, Longitude =
            2.2949378 };
        break;
    case nameof(londonButton):
        position = new BasicGeoposition() {
            Latitude = 51.502716, Longitude = -
            0.119304 };
        break;
    case nameof(newYorkButton):
        position = new BasicGeoposition() {
            Latitude = 40.748463, Longitude = -
            73.98567 };
        break;
    case nameof(chinaButton):
        position = new BasicGeoposition() {
            Latitude = 40.67693, Longitude =
            117.23193 };
        break;
    }
var point = new Geopoint(position);
// Set the map location
await mapControl.TrySetViewAsync(point, 12);
}
```

Run the app again and try clicking some of the location buttons. The map should reposition to the corresponding coordinates and zoom to the correct level:

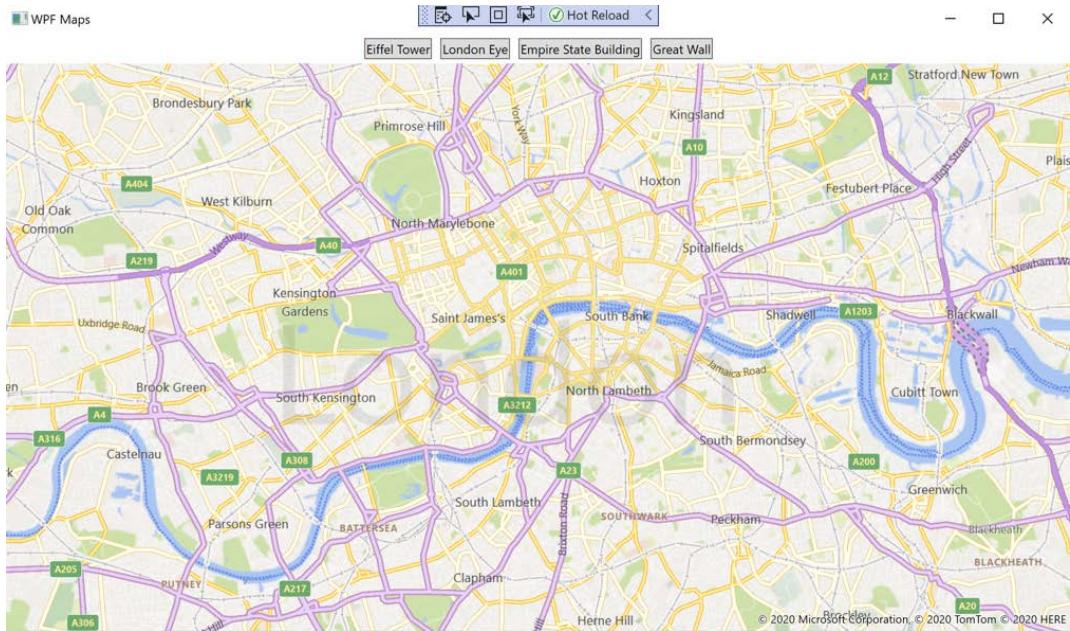


Figure 10.7 – MapControl displays the coordinates of the London Eye

The same technique can be used in any WPF or WinForms project. Now, we will use the `WebView` NuGet package to work with the `WebViewCompatible` browser control in our WPF project.

## Using the `WebViewCompatible` browser control in WPF

The `WebViewCompatible` browser control in the WCT will render web pages in the legacy Edge browser in Windows 10 and in an **Internet Explorer (IE)**-compatible browser in earlier versions of Windows. As an example, if you are building a WPF or WinForms application on .NET Framework, you can deploy to Windows 7 with this browser control and continue to render web pages in the browser control. Keep in mind that many modern websites have compatibility issues with IE.

Now, let's add a pop-up browser window to the WPF project. For this, we're going to modify the **WpfMaps** project and add a **Get Great Wall Info** button to the button bar at the top of the main window. This will launch another window with a browser that will display the **Wikipedia** page for the Great Wall of China:

1. Start by adding a fifth button to **StackPanel** in **MainWindow.xaml**:

```
<Button x:Name="wallInfoButton" Margin="4"
        Content="Get Great Wall Info"
        Click="wallInfoButton_Click"/>
```

2. Next, right-click the **WpfMaps** project and select **Add | New Item**. From the **Add New Item** dialog, select the **Window (WPF)** template, name it **BrowserWindow.xaml**, and click **Add**.
3. In **BrowserWindow.xaml**, add the required namespace reference for the **WebViewCompatible** control and add the control to the main **Grid**. Set the **Source** property to the URL for the Wikipedia page, as shown here:

```
<Window x:Class="XamlIslandsSample.WpfMaps.BrowserWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/
        xaml/pr
        esentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/
        xaml"
        xmlns:d="http://schemas.microsoft.com/expression/
        blend
        /2008"
        xmlns:mc="http://schemas.openxmlformats.org/
        markup-
        compatibility/2006"
        xmlns:local="clr-
        namespace:XamlIslandsSample.WpfMaps"
        xmlns:controls="clr-
        namespace:Microsoft.Toolkit.Wpf.UI.Controls
        ;assembly=Microsoft.Toolkit.Wpf.UI
        .Controls.WebView"
        mc:Ignorable="d"
        Title="BrowserWindow" Height="450"
        Width="800">
```

```
<Grid>
  <controls:WebViewCompatible
    Source="https://en.wikipedia.org/wiki/Great_
    Wall_of_China"/>
</Grid>
</Window>
```

4. Finally, in `MainWindow.xaml.cs`, launch a new instance of `BrowserWindow` from the `wallInfoButton_Click` event handler:

```
private void wallInfoButton_Click(object sender,
  RoutedEventArgs e)
{
  var browserWindow = new BrowserWindow();
  browserWindow.ShowDialog();
}
```

5. Now, run the application and try out the new button. You should see a new window open, displaying the expected Wikipedia page:



Figure 10.8 – The WPF launching a `WebViewCompatible` browser control in a new window

WebViewCompatible works exactly like other WebView controls we have used in this book. If you have access to a Windows 8.x or Windows 7 PC, you can add this control to a .NET Framework project and deploy it to that computer. WebViewCompatible will load the web content in an IE browser control instead of Edge. You can also use this control with a .NET Core 3.x or .NET 5 project.

Let's wrap up this chapter by taking a quick look at using the WebView2 control in a WinForms project.

## Working with the WebView2 browser control in WinForms

The last control we want to explore in this chapter is WebView2. This is the same WebView2 that is available natively in WinUI 3. We have already discussed the basics of WebView2 and added it to a WPF project in the previous chapter, so we won't spend time repeating those details. Instead, let's jump straight into how to use it in WinForms:

1. Add a new **Windows Forms App (.NET Framework)** project to the current solution and name it `XamlIslandsSample.WinFormsBrowser`. Be sure to select .NET Framework version 4.6.2 or later when creating the project.
2. Open **NuGet Package Manager**, make sure to check the **Include prerelease** checkbox, and add the newest prelease version of the **Microsoft.Web.WebView2** package to the new project.

3. Open **Form1.cs** in the WinForms designer and open the Visual Studio **Toolbox** window. You will find the **WebView2** control under the **WebView2Control** section of **Toolbox**:

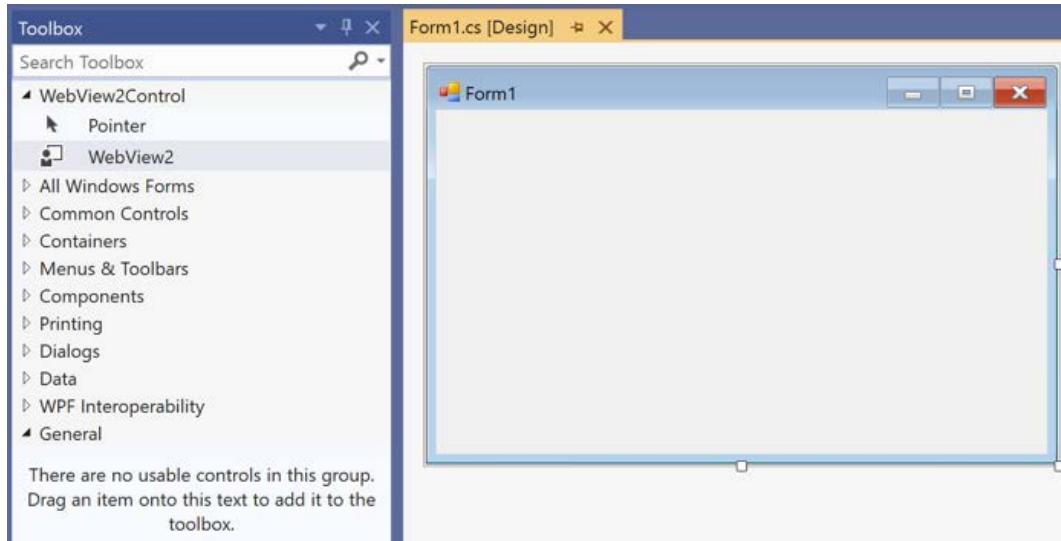


Figure 10.9 – The **WebView2** control in Visual Studio's Toolbox window

4. Drag a copy of **WebView2** to the **Form1** design surface. In the **Properties** window, set **Name** to **webViewBrowser**, **Source** to **https://microsoft.github.io/microsoft-ui-xaml/**, and **Dock** to **Fill**.
5. Set **WinFormsBrowser** as the start up project and run the application. The form will display with **WebView2** and navigate to the WinUI home page:

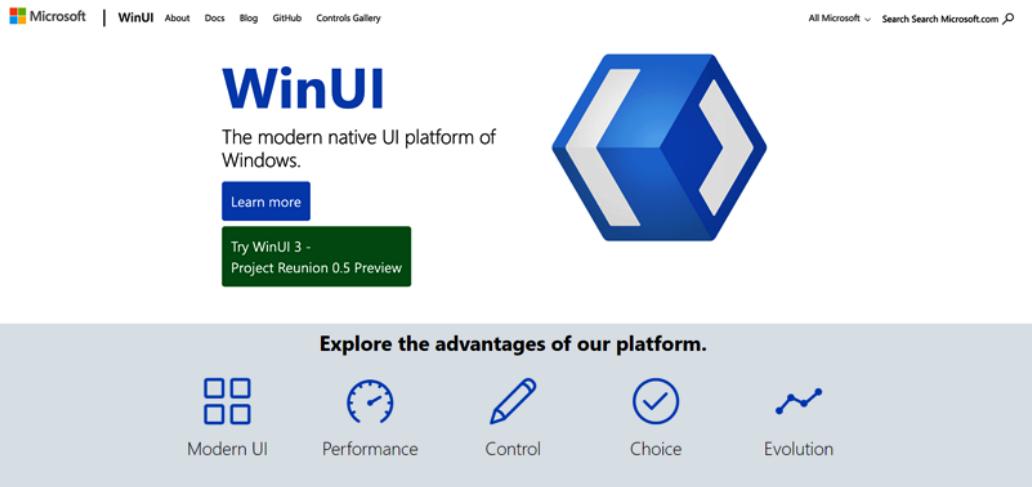


Figure 10.10 – A WebView2 control hosted in a WinForms window

That's all there is to it. If you want to allow users to navigate to other URLs, you can add a navigation toolbar, as we did in the sample applications in *Chapter 9, Enhancing Applications with the Windows Community Toolkit*. Let's wrap up and review what we've learned in this chapter.

## Summary

As you can see, adding modern controls to existing WPF and WinForms applications can be a great first step in an uplift effort. We have seen how to add one of the wrapped controls to a legacy application to bring some specific Windows 10 functionality to those projects. We also wrapped an entire user control from a UWP project and included it in WPF and WinForms projects. The XAML Islands controls can help you modernize your existing projects without the need to rewrite the entire project right away.

In the next chapter, we will shift our focus back to pure WinUI projects and examine some techniques for debugging our applications.

## Questions

1. What is the minimum version of .NET Core required to use XAML Islands in a WPF or WinForms project?
2. What is the minimum .NET Framework version required to consume the `WebView2` control?
3. Which browser control in the WCT can be used on Windows 7?
4. Which XAML Islands control can host a UWP first-party, third-party, or custom user control?
5. Which wrapped control can display interactive maps?
6. Which two wrapped controls are used together to allow user input through touch and pen?
7. What is the difference between `WebView` and `WebView2`?

# Section 3: Build and Deploy on Windows and Beyond

This section rounds out your WinUI knowledge by exploring techniques for debugging, building, and deploying WinUI 3.0 applications. You will explore the extensive debugging tools Visual Studio has to offer WinUI developers. Next, you will see how you can host a web application inside a WinUI application, leveraging Blazor, the Visual Studio Code editor, GitHub Actions, and the new WebView2 WinUI control. Finally, readers will learn about the options for building and deploying WinUI applications to users with Visual Studio, Visual Studio App Center, the Microsoft Store, and Microsoft's new command-line installer called WinGet.

This section includes the following chapters:

- *Chapter 11, Debugging WinUI Applications with Visual Studio*
- *Chapter 12, Hosting an ASP.NET Core Blazor Application in WinUI*
- *Chapter 13, Build, Release, and Monitor Applications with Visual Studio App Center*
- *Chapter 14, Packaging and Deploying WinUI Applications*



# 11

# Debugging WinUI Applications with Visual Studio

Good debugging skills are essential for developers. While .NET developers need to know how to use things such as breakpoints and the **Output** and **Immediate** windows, WinUI debugging adds another set of tools and techniques to learn. There are issues that can arise in the UI layer with data binding, layout, and resources. You will learn how to use the **Live Visual Tree**, the **Live Property Explorer**, and how to discover data binding errors with Visual Studio's **XAML Binding Failures** window.

In this chapter, we will cover the following topics:

- How to debug WinUI applications and work with breakpoints in ViewModels and service classes
- How to debug data binding problems with the **XAML Binding Failures** window in Visual Studio and avoid common problems binding to collections

- How to use the **Live Visual Tree** window in Visual Studio to find layout problems in your XAML
- How to use the Live Property Explorer to get and set real-time information about the data in your XAML elements

By the end of this chapter, you will feel comfortable debugging common problems that WinUI and other XAML developers encounter while developing applications.

**Note**

Visual Studio's XAML designer and many of the debugging tools do not fully support WinUI 3 projects yet. You can enable WinUI support in Visual Studio by going to **Tools | Options | Preview Features** and checking **Enable UI**

**Debugging Tooling for WinUI 3 Projects.** It is anticipated that this support will be fully functional sometime in 2021. In this chapter, we will explore debugging techniques using a UWP project that references the WinUI 2.x NuGet package. The `WinUI2MediaCollection` project on GitHub is a copy of the `MyMediaCollection` project we created in earlier chapters, modified to work with the UWP project structure. All of these examples and instructions will work the same with WinUI 3 when Visual Studio support is generally available.

## Technical requirements

To follow along with the examples in this chapter, the following software is required:

- Windows 10 version 1803 (build 17134) or newer.
- Visual Studio 2019 Version 16.9 or newer with the following workloads: .NET Desktop Development and Universal Windows Platform Development.
- To create desktop WinUI projects, you must also install the latest .NET 5 SDK.

The source code for this chapter is available on GitHub at this URL:  
<https://github.com/PacktPublishing/-Learn-WinUI-3.0/tree/master/Chapter11>.

## Debugging in Visual Studio

There are several fundamental areas of debugging WinUI applications that we will be covering in this section. Some of these techniques are applicable to debugging other types of .NET applications, whereas others are specific to XAML and WinUI development. Throughout the book, we've run our projects with Visual Studio, which is an example of a local debugging session. We will explore other ways to debug local applications as well as remote applications.

Simple mistakes in XAML markup are not always apparent when we're writing it, and this kind of problem is not detected by the compiler. In this section, we will see how to detect and avoid XAML markup issues and how to adhere to best practices.

Let's get started by taking a closer look at debugging local applications.

### Debugging local applications

Open the **WinUI2MediaCollection** project from GitHub and compile it to make sure you have downloaded all the referenced NuGet packages. This UWP project uses WinUI 2.x controls, but otherwise, it is identical to the **MyMediaCollection** project that we last used in *Chapter 7, Fluent Design System for Windows Applications*. Run the application to make sure everything works as expected.

For our first local application debugging walk-through, we will explore the XAML Designer in Visual Studio.

## Using the XAML Designer

Now, open **MainPage.xaml** and you won't notice any differences from the previous WinUI project, except for the different project name. You will notice one very big difference in Visual Studio when opening the XAML file, assuming Microsoft has not added designer support for WinUI 3 by the time you are reading this book. The **XAML Designer** in Visual Studio provides an interactive design surface for your XAML files. Depending on your **XAML Designer** settings in **Tools | Options**, this will appear as either a split-pane view or as **XAML** and **Design** tabs that can be selected to switch between the views:

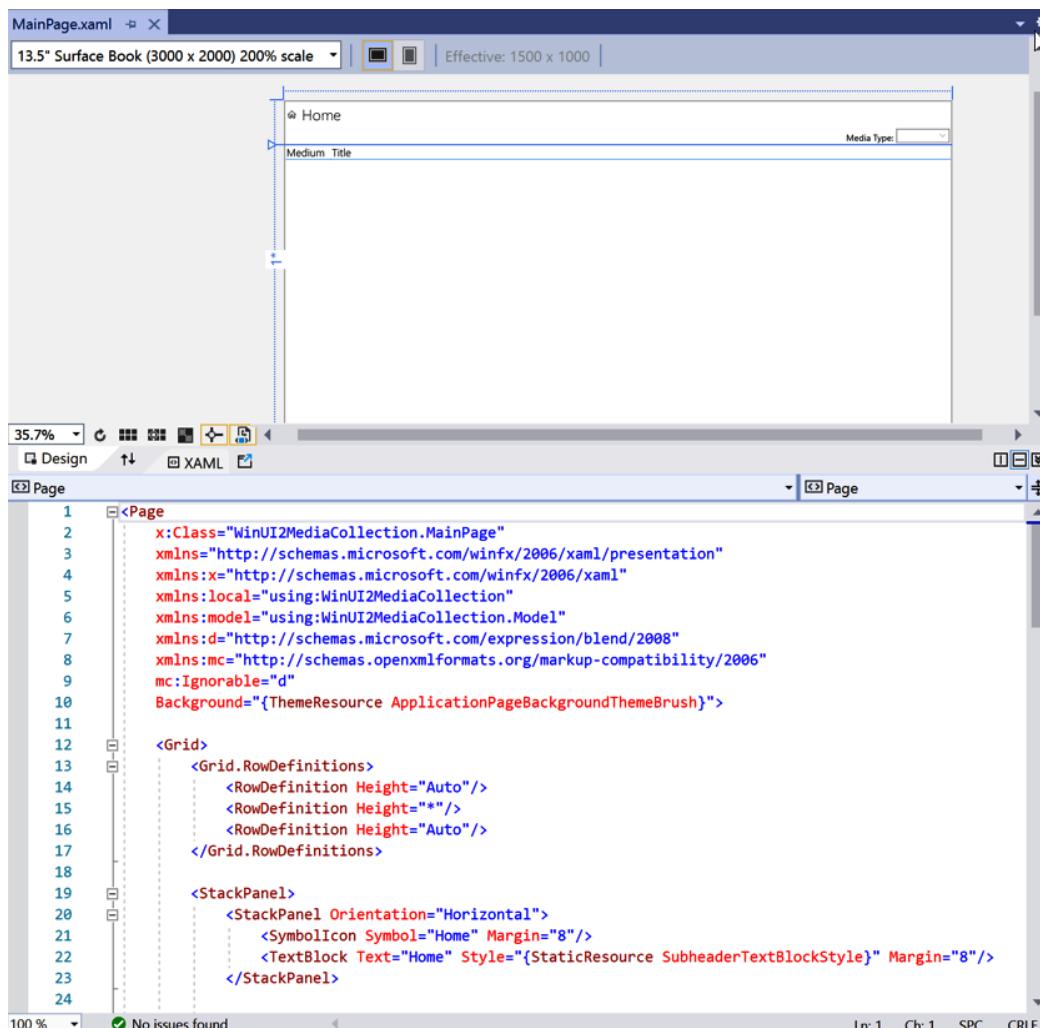


Figure 11.1 – The XAML designer in horizontal split-pane mode

The design surface can be used to lay out elements with drag-and-drop, but be cautious of the XAML markup that is created when adding elements to a page or updating their size and position. The XAML generated by size and position changes in the designer is not always conducive to a responsive layout. The designer is a great way to design **Grid** controls, providing tools to easily add the number of rows and columns your layout requires. You can also select an item in the designer and change its properties in the **Properties** window. Notice that all of the changes are immediately reflected in the designer and the XAML markup.

The designer is a great way to immediately see a preview of the XAML you are writing. It will also highlight the element that has the cursor's focus in the XAML editor. This can help you stay oriented when working on large XAML files.

You can also change the device screen size of the designer. I initially had **13.5" Surface Book (3000 x 2000) 200% scale** selected in the drop-down box in the upper left of the designer. Try changing that to **42" Xbox (1920 x 1080) 200% scale**:

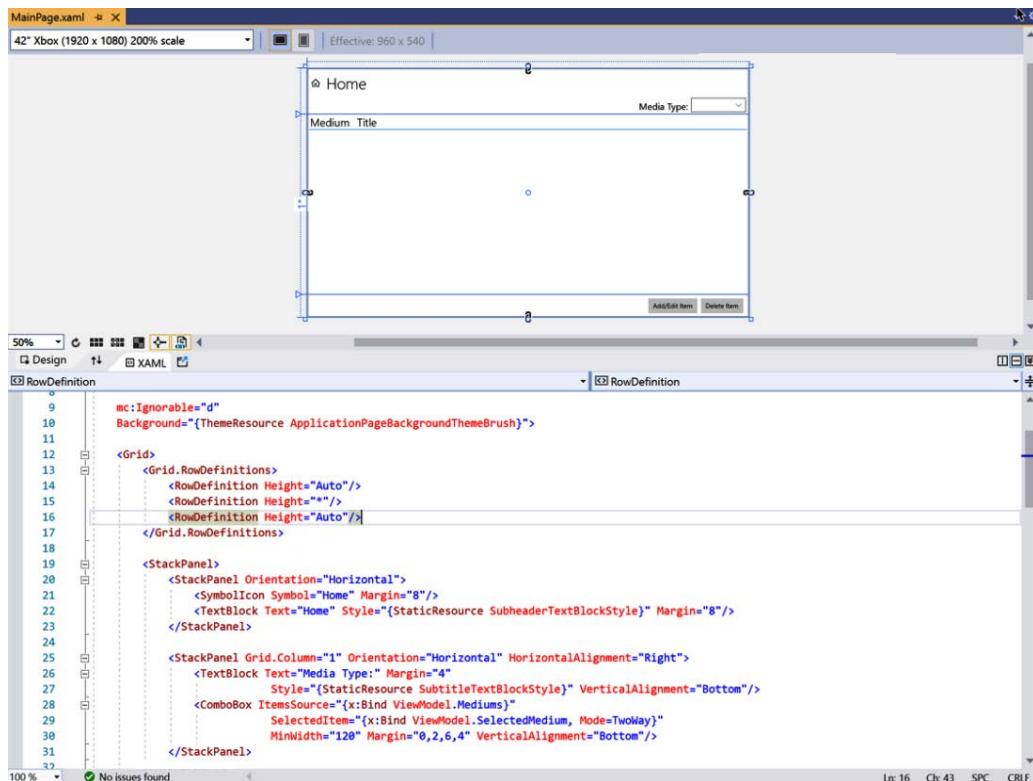


Figure 11.2 – Using the XAML designer to preview Xbox application display

Each selection includes three measurements that impact how your page renders on the target device:

- **Screen size:** This is the physical size of the device screen.
- **Resolution:** This is the screen resolution of the target device.
- **Scale:** This is the scaling factor selected on the device. Different devices have different scaling factors, and Windows 10 now allows users to change the scaling factor in their **Display** settings.

While a television may seem like a very large device, Microsoft classifies them as small devices due to the resolution, scaling, and the distance that users are positioned from the screen. Read more about screen sizes and responsive design here:

<https://docs.microsoft.com/en-us/windows/uwp/design/layout/screen-sizes-and-breakpoints-for-responsive-design>.

Now that you have an idea of how your application would appear if deployed to the Microsoft Store for Xbox, switch your designer back to a PC resolution before continuing.

Next, let's look at how to debug a local application installed on your PC.

## Debugging a locally installed application

We have run and debugged our Visual Studio solutions throughout this book. Now we will see how you can debug an application that you have already installed in Windows. You have previously run several projects while reading this book. Unless you uninstalled them, each should appear as an installed app package that you can debug. Let's start with the steps:

1. Start by selecting **Debug | Other Debug Targets | Debug Installed App Package** in Visual Studio. The **Debug Installed App Package** window will appear:

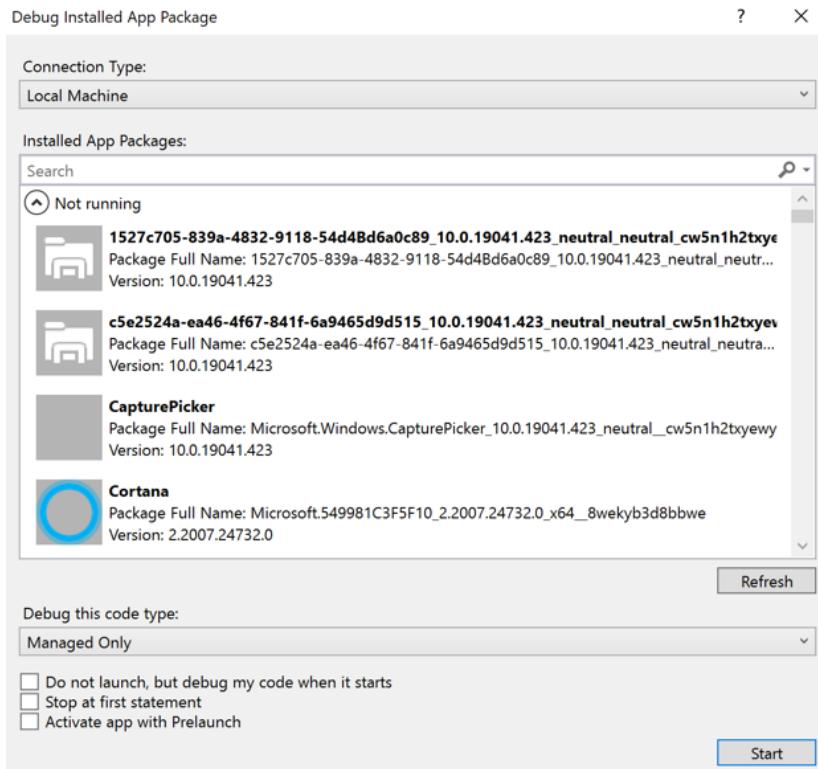


Figure 11.3 – The Debug Installed App Package window

This window will show all the installed packages on your Windows PC. Some names will be familiar, such as **Cortana**, **Fluent XAML Theme Editor**, or **Microsoft Edge**. Others will only be listed by their application ID. You can select some of these other applications to debug, but without debug symbols, you won't be able to hit any breakpoints or step through the code. Let's find one of ours.

2. We have the **WinUI2MediaCollection** project already open so let's search for **WinUI2**. The **WinUI2MediaCollection** application will appear in the search results. If you do not see it, make sure you have run the application from Visual Studio at least once. This step is necessary to deploy the application to Windows.
3. Select it and click **Start**. The application will run, and Visual Studio will start debugging.

If you don't want to start debugging immediately, you can select the **Do not launch, but debug my code when it starts** check box. Now, Visual Studio will start debugging when you start the application from the **Start** menu or some other method.

Another way to start debugging an installed local application is by attaching it to a running application:

1. First, run the app from the **Start** menu, and in Visual Studio go to **Debug | Attach to Process**.
2. In the **Attach to Process** window, find the `ApplicationFrameHost.exe` process with the title that matches the application you want to debug. This is the process that hosts every WinRT application on Windows:

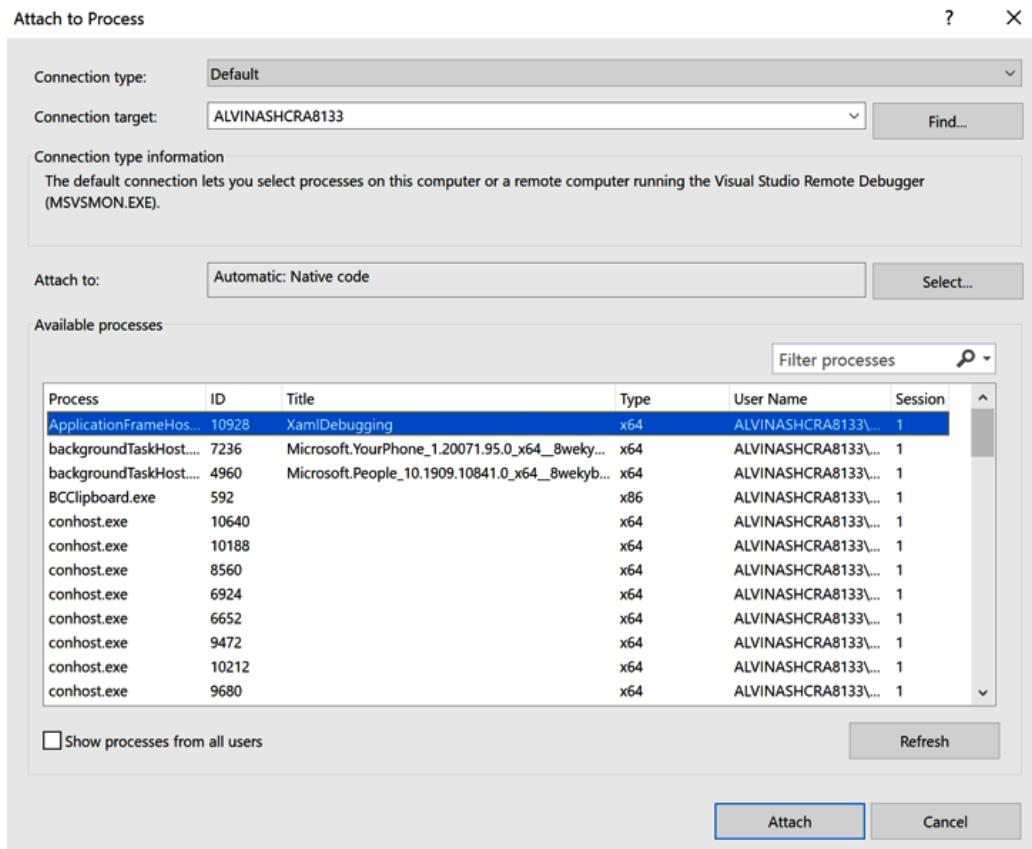


Figure 11.4 – Attaching to a running process to debug an application

3. Click **Attach** and begin debugging as usual.

These are different ways to start a debugging session on a local application, but what if you want to debug your application running on another machine? Let's examine those options next.

## Debugging remote applications

There are many reasons why you might want to debug an application on a remote machine. Sometimes you can only reproduce an error on one device. Perhaps there is an issue that only occurs on a specific device type or screen size. There are some devices, such as Xbox, where you have to use remote debugging.

### Note

Before starting any remote debugging session, ensure that the target device has **developer mode** enabled. For more information about activating developer mode, you can read this Microsoft Docs article: <https://docs.microsoft.com/en-us/windows/uwp/get-started/enable-your-device-for-development>. If the target machine is running a version of Windows 10 prior to the Creator's Update (v1703), you must also install and configure the remote debugging tools: <https://docs.microsoft.com/en-us/visualstudio/debugger/remote-debugging>.

To debug a remote installed application, you will use Visual Studio's **Debug Installed App Package** window again:

1. Open the window from **Debug | Other Debug Targets | Debug Installed App Package**.
2. Change **Connection Type** to **Remote Machine**:

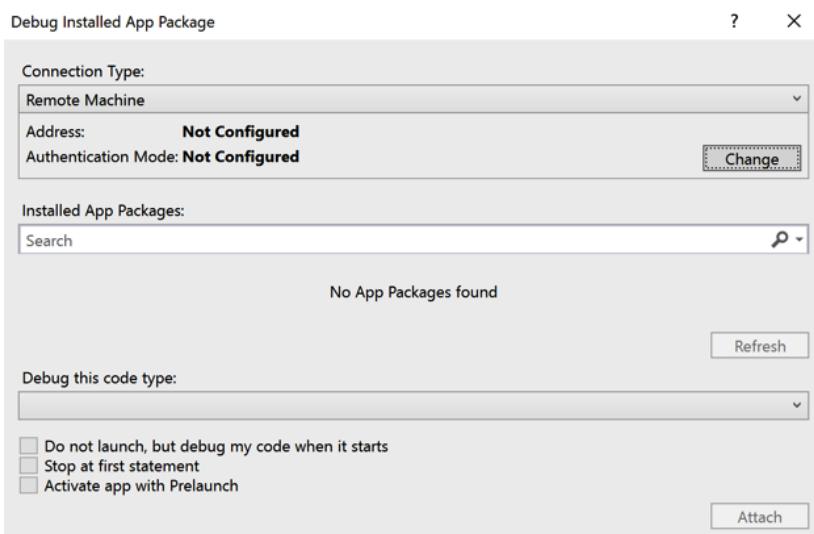


Figure 11.5 – Debugging an application package on a remote machine

3. Click the **Change** button that appears to open the **Remote Connections** window.
4. Visual Studio will attempt to discover other Windows devices and list them in the **Auto Detected** section. If you see the device you want, select it to continue. If the device you want to debug isn't shown, enter its IP address in the **Address** field in the **Manual Configuration** section and click **Select**:

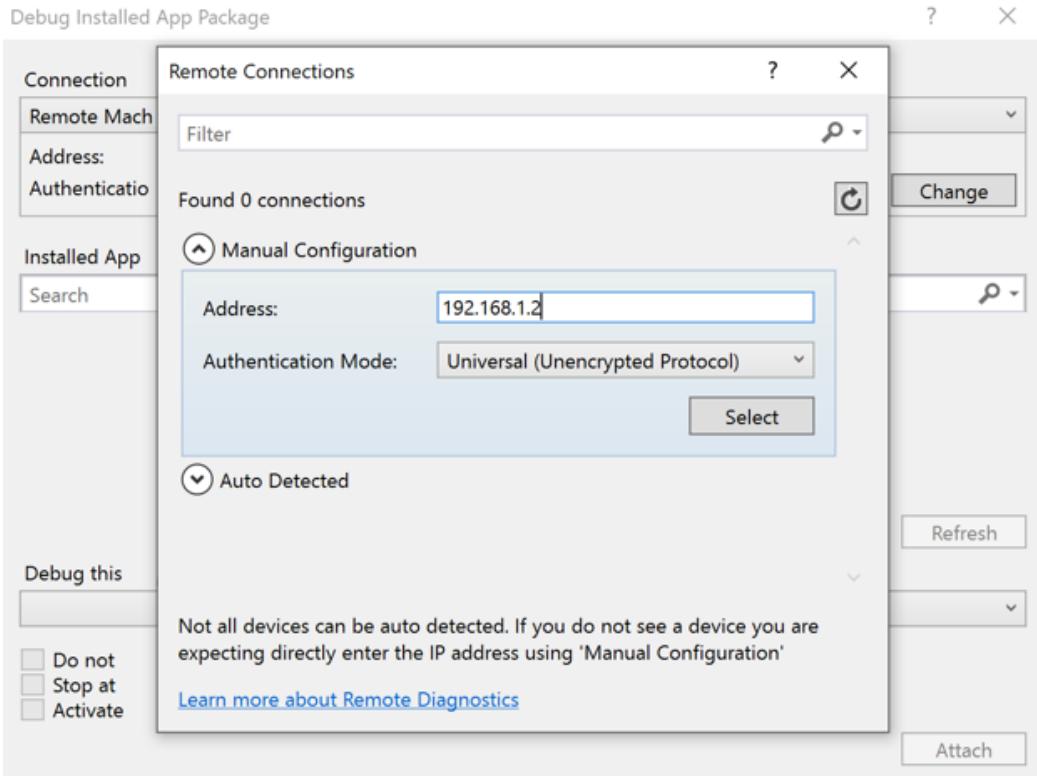


Figure 11.6 – Enter a manually configured remote connection for debugging

5. When you return to the previous window, you can select the application to debug from the list of **Installed App Packages** on the selected device.
6. Click **Start** to begin debugging just as we did with the locally installed application in the previous section.

Another way to debug on a remote machine is to change **Local Machine** to **Remote Machine** on the standard toolbar in Visual Studio:

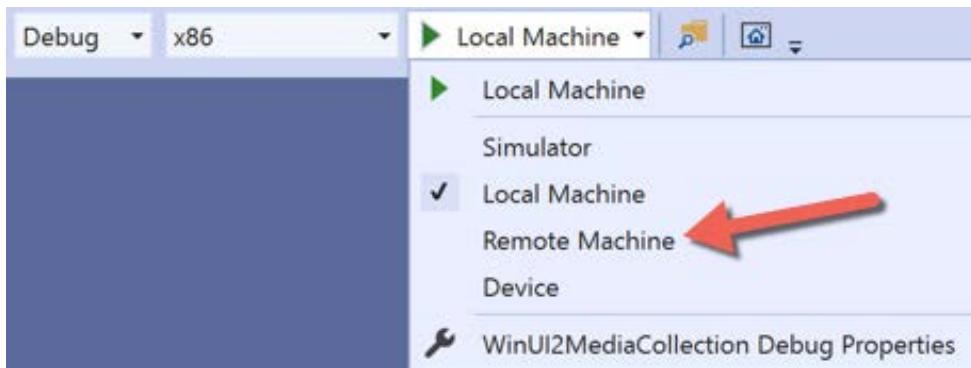


Figure 11.7 – Selecting Remote Machine when starting a new debugging session

This will open the **Remote Connections** window where you can select the remote device for debugging. Remote connections can also be selected from the **Attach to Process** window.

These techniques will allow you to connect to other Windows machines, including other device types such as Xbox, HoloLens, Surface Hub, and Windows IoT devices. To select a device that is physically connected to your machine via USB, you would select **Device** instead of **Remote Machine** when debugging. Remote machine options can also be accessed from the **Debug** tab of the project's **Properties** page.

Let's shift gears and examine some common mistakes that can cause rendering issues in your application's UI.

## Common XAML layout mistakes

There are many kinds of mistakes that XAML developers can make while coding the UI. Many of these will not be caught by the compiler. If you have a syntax error or an invalid `x:Bind` expression, these will fail while compiling, but many other issues will not.

The first source of common XAML layout mistakes we will explore is the `Grid` control.

### Grid layout issues

Some common mistakes related to the commonly used `Grid` control revolve around its rows and columns. Forgetting to set `Grid.Row` or `Grid.Column` on child controls typically leads to overlapping elements at runtime. The same kinds of issues can be seen when setting these values incorrectly or when working with `Grid.RowSpan` and `Grid.ColumnSpan`. One mistake that is not always immediately apparent is setting a `Grid.Row` or `Grid.Column` to a value that's too high.

Open **MainPage.xaml** in our **WinUIMediaCollection** project and remove the `Grid.Row` attribute from the `Border` control containing the two `Button` controls at the bottom of the page. You will immediately see the buttons move to the top of the page in the designer and overlap the controls in the header area.

Now restore the `Grid.Row` attribute but change the value to 5. Everything looks okay again even though the `Grid` only has three rows. Since 5 is greater than the available rows, the control is added to the last row in the `Grid`. However, if you wanted to add a status bar below the buttons by adding a fourth row to the grid, the buttons and status bar would overlap after setting the status bar to `Grid.Row= "3"`.

## Problems when applying style

XAML Style resources are another common source of unintentional UI change. When creating a `Style` in a `Resource`, you should be aware of how it will be applied to controls within the scope of that `Resource`.

Open the **ItemDetailsPage.xaml** and review the `Page.Resources` section. Here we created three `Style` elements for the current page. Each has a different `Target` type: `TextBlock`, `TextBox`, and `ComboBox`. However, they won't be applied to every control of those types on the `Page` because we also gave each `Style` an `x:Key`. That means that the `Style` will only be applied to elements of that type when the `Style` property is explicitly set to that named resource. These are referred to as *explicit styles*:

```
<TextBlock Text="Name :" Style="{StaticResource  
AttributeTitleStyle}"/>
```

If you remove `x:Key` from a `Style` in `Page.Resources`, that *implicit style* will apply to every control of the specified `Target` type on the `Page`, unless those controls have an explicit `Style` set. In a large application with styles declared at different scopes (`Application`, `Page`, `control`), it can sometimes be difficult to determine which `Style` has been applied to a control. We will see how to do this later in the chapter when discussing Visual Studio's **Live Property Explorer** window. It is a best practice that implicit styles should always inherit from an explicit style. This enables developers to inherit from a default style and reduces repeated implicit style attributes across elements.

Next, we will look at a tool that can help find common XAML problems through static code analysis.

## Improving your XAML with static code analysis

There is a free, open source extension for Visual Studio that, among other things, adds support for static code analysis to XAML files. The **Rapid XAML Toolkit** (<https://rapidxaml.dev/>) provides XAML analyzers and code fixes for common issues and provides support for adding your own custom XAML analyzers. Let's install the tool in Visual Studio and see what kind of issues it can identify in our project:

1. In Visual Studio, go to **Extensions | Manage Extensions**.
2. The **Manage Extensions** window will appear. Select **Visual Studio Marketplace** and search for `rapid xaml`:

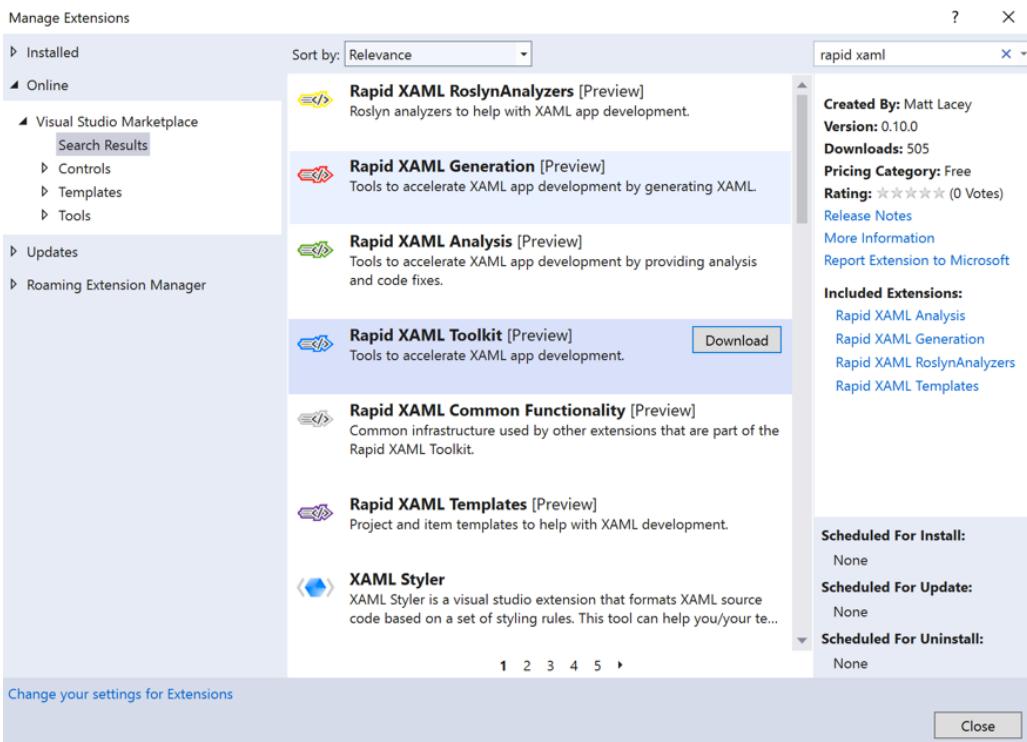


Figure 11.8 – Installing the Rapid XAML Toolkit

3. Click **Download** to queue the extension for installation. When the download completes, you can close the **Manage Extensions** window and restart Visual Studio to complete the installation.

4. Now when you view the **Error List** window in Visual Studio, there are a handful of warnings from Rapid XAML's analyzers:

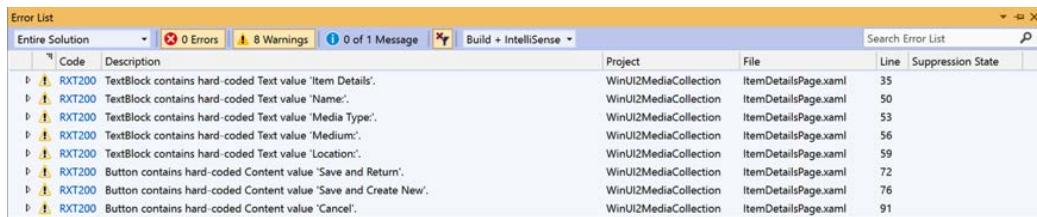


Figure 11.9 – Viewing the new warnings from the Rapid XAML Toolkit

5. Open **ItemDetailPage.xaml**, place your cursor over one of the green squiggles created by the code analyzer, and click the lightbulb icon:

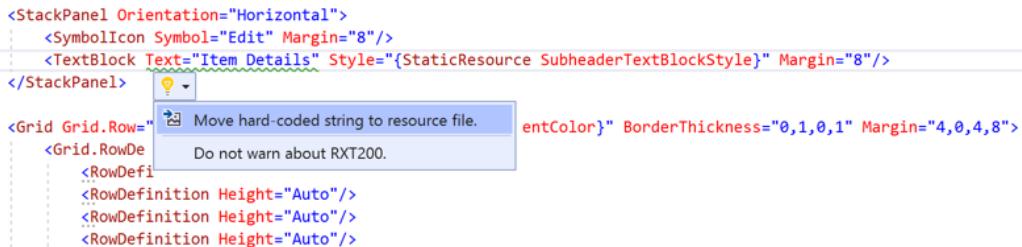


Figure 11.10 – View the quick fix for a code analyzer warning

Alternatively, you can right-click on the XAML and select **Rapid XAML | Move hard-coded string to resource file.** to fix all warnings of this type.

This is a list of the other analyzers that are currently part of the toolkit:

- **RXT101:** Use of a `Grid.Row` value without corresponding `RowDefinition`
- **RXT102:** Use of a `Grid.Column` value without corresponding `ColumnDefinition`
- **RXT103:** Use of a `Grid.RowSpan` value without corresponding `ColumnDefinition`
- **RXT104:** Use of a `Grid.ColumnSpan` value without corresponding `ColumnDefinition`
- **RXT150:** `TextBox` does not have an `InputScope` specified
- **RXT160:** `SelectedItem` binding should probably be `TwoWay`
- **RXT200:** Hard-coded string value that should be a resource
- **RXT300:** `Entry` does not have a `Keyboard` specified

- **RXT300:** Password entry does not have a `MaxLength` specified
- **RXT300:** Image lacks accessibility consideration
- **RXT300:** ImageButton lacks accessibility consideration
- **RXT401:** Handle both `Checked` and `Unchecked` events for a CheckBox
- **RXT402:** Use `MediaPlayerElement` in place of `MediaElement`
- **RXT451:** `x:Uid` should begin with an upper-case character
- **RXT452:** Name should begin with an upper-case character
- **RXT999:** Unknown error – something went wrong when parsing the XAML document

The Rapid XAML Toolkit has many other features outside of analyzers and code fixes, and new features and analyzers are being added frequently. To see a list of upcoming features and fixes being considered, you can view the issues on GitHub:  
<https://github.com/mrlacey/Rapid-XAML-Toolkit/issues>.

**Note**

Community-driven projects like this are always looking for contributors. It's a great way to get started with open source development.

Another related topic that can be a common source of developer angst is debugging data binding. Let's see how we can avoid some common pitfalls in the next section.

## Learning to pinpoint data binding errors

While debugging data binding problems is not as difficult in WinUI and UWP as it is in WPF, thanks to `x:Bind` compiled bindings, there are still some gotchas of which you should be aware. In this section, we will look at what can go wrong in both views and ViewModels and how you can diagnose and fix the problems.

## Common mistakes in data binding

Using `x:Bind` will evaluate whether you're binding to a valid source at compile time and can give you the peace of mind to know that your views and ViewModels are hooked up correctly, but there is still a lot that can go wrong. Let's review a few of the most common mistakes.

## Selecting the best binding mode

We have seen in previous chapters that the default mode for most controls with `x:Bind` is `OneTime`, while the default for `Binding` is `OneWay`. Defaulting to `OneTime` helps with performance as many read-only properties are only ever set when the view is first created. However, if you forget to change this for controls bound to data that changes as the user is interacting with the page, you may not immediately realize why the data doesn't refresh.

When you're binding controls that need data to flow in both directions, remember to set `Mode` to `TwoWay`. We used this in the Media Collection application with the `ComboBox`. `SelectedItem` property to filter the collection by media type.

## Triggering `PropertyChanged` notifications

By binding to a `ViewModel` that properly implements `INotifyPropertyChanged` for all its public properties, issues related to `PropertyChanged` are not common. Problems can arise if `ViewModel` code outside of one of these properties updates the property's value by setting the `private` backing variable. This will update the property's value without notifying the view. You can avoid this by updating the value by setting the `public` property. If there is a good reason for not updating the property directly, then a `PropertyChanged` event should be manually triggered for that property after updating the value. It is a best practice to use the `nameof` method in C# to ensure you use a property name that exists. You can also use `CallerMemberNameAttribute` in .NET: <https://docs.microsoft.com/en-us/dotnet/api/system.runtime.compilerservices.callermembernameattribute?view=net-5.0>. An exception will be raised if your application tries to raise a property change notification for a property that does not exist.

## Working with `ObservableCollection<T>`

`ObservableCollection<T>` serves an important role. Lists in the view will stay synchronized with the collections if they are used correctly. There are a few practices to avoid when using observable collections.

Do not replace the entire value of an observable collection. The element on the view bound to the property will still be bound to the original collection. Any subsequent changes to the collection will not be reflected in the view. While you can work around this by triggering a `PropertyChanged` notification, this can have performance implications with larger collections. It can also be jarring to the user, as most controls will reset the current view to the beginning of the list. Set `ViewModel` properties that use `ObservableCollection<T>` to be read-only to avoid accidentally resetting the entire collection. The one exception to this is if you know the collection will be completely repopulated. Removing and re-adding a large number of items individually in a list can result in a poor user experience.

Do not use LINQ to modify observable collections. LINQ expressions do not operate by calling the `Add` and `Remove` methods on the observable collections. They do not even return `ObservableCollection<T>`. If you use LINQ and convert the results back to `ObservableCollection<T>`, you will be back to replace the entire collection, which was just discussed.

Here is an example using LINQ. This will cause the view to stop receiving `CollectionChanged` notifications because LINQ does not return an `ObservableCollection`:

```
_entryList = from entry in _entryList
              where entry.Lastname.StartsWith("J")
              select entry;
```

This example does the same thing. It removes any entries with a `Lastname` starting with J. By using the `Remove` method and not changing the entire collection, the `CollectionChanged` events are preserved:

```
for (int i = _entryList.Count - 1; i >= 0; i--)
{
    Entry entry = _entryList[i];
    if (!entry.Lastname.StartsWith("J"))
    {
        _entryList.Remove(entry);
    }
}
```

Finally, it is best not to use observable collections for lists that do not change. The extra overhead they bring is not needed for static data. Bind to an `IList<T>`, `IEnumerable<T>`, or better yet, a plain array of items instead.

## Using the XAML Binding Failures window

Visual Studio 2019 version 16.7 introduced the new **XAML Binding Failures** window and a **Binding failures** indicator on the in-app toolbar. These are for failures in Binding, not `x:Bind`, as those errors are checked and caught by the compiler. Clicking the red indicator will take you to the **XAML Binding Failures** window while debugging. Binding failures were always available in the **Output** window while debugging, but they could be difficult to find. The new window provides sorting and searching. Similar failures will also be grouped to make it easy to address related items together:



Figure 11.11 – Checking the in-app toolbar for binding failures

### Note

At the time of this writing, you need to opt in to this preview feature in Visual Studio by going to **Tools | Options** and selecting **XAML Binding Failures Window** under **Environment | Preview Features**.

Let's give it a try. Open the **XamlDebugging** solution from the GitHub repository for this chapter and follow along with these steps:

1. Open **MainPage.xaml** and review the XAML. The Page contains a `StackPanel` with two `TextBox` child elements:

```
<StackPanel DataContext="Test">
  <TextBox x:Name="Text1"
    Text="{Binding Path=SomeText,
    Mode=TwoWay}" />
  <TextBox Text="{Binding ElementName=Text1,
    Path=Text, Mode=OneWay}" />
</StackPanel>
```

The `DataContext` does not exist, so what do you think will be reported in the binding failures? Let's see...

2. Run the application and look at the in-app toolbar. There is an indication that we have one binding failure.

3. Click the red indicator on the toolbar to open the **XAML Binding Failures** window in Visual Studio:



Figure 11.12 – The XAML Binding Failures window

The line in the window tells us **BindingExpression path error: 'SomeText' property not found on 'Windows.Foundation.IReference`1<String>'**. This is useful information. It would be more useful if there was something telling us directly that the data context itself is not valid. However, if you had a dozen controls using this data context, and all the bindings were failing, it would become clear that there is a problem with the data context and not the individual bindings.

4. Let's make sure the other binding is working correctly. Enter some text into the first TextBox:



Figure 11.13 – Binding TextBoxes with ElementName

The text you enter should be duplicated in the second TextBox as your type. Our binding to `ElementName` is working as expected.

I will leave fixing the data source as an exercise for you. Add a valid data source to the code behind and see if the binding error is cleared up.

Now, let's explore a few other XAML debugging tools available from the in-app toolbar.

## Debugging live data with Live Visual Tree and Live Property Explorer

While the **XAML Binding Failures** window is new to Visual Studio, the in-app toolbar has been available to XAML developers since 2015. The toolbar floats over the active window in your application while you are debugging. There are several parts to the toolbar:

- **Go to Live Visual Tree:** Opens the **Live Visual Tree** window.
- **Select Element:** Allows you to select an element in the **Live Visual Tree** by clicking on it in the active window.
- **Display Layout Adorners:** This will highlight the element in the UI that is selected in the **Live Visual Tree**.
- **Display Heatmaps:** This control will indicate which parts of the UI are most active.
- **Track Focused Element:** While the **Live Visual Tree** window is open, toggling this on will indicate in the **Live Visual Tree** which element currently has focus in the UI.
- **Binding failures:** Indicates the number of current binding failures and opens the **XAML Binding Failures** window when clicked.
- **Hot Reload:** Indicates whether **XAML Hot Reload** is currently available.

Let's explore a few of the most commonly used debugging tools. If you wish to get more information on the remaining tools, you can visit this page about XAML tools on Microsoft Docs: <https://docs.microsoft.com/en-us/visualstudio/xaml-tools/> or this Channel 9 video about XAML tools in Visual Studio: <https://channel9.msdn.com/Shows/Visual-Studio-Toolbox/New-XAML-Features-in-Visual-Studio>. We'll start with Hot Reload.

## Coding with XAML Hot Reload

XAML Hot Reload is a simple but powerful feature. Before Visual Studio 16.2, it was known as **XAML C# Edit & Continue**. Hot reload has been available to web developers for a little longer, and the name change for XAML helps to clear up some confusion for developers familiar with the concept in web development. The idea of Hot Reload is that you can make changes to your UI and the changes are reflected in the running application without having to stop debugging and recompile. Let's try it with the XamlDebugging solution:

1. Start by running the application again. Make sure that the **Hot Reload** indicator shows that it is enabled. It should only be disabled if you are running an unsupported project type.
2. Now let's make a change to **MainPage.xaml**. We're going to change the Background color of StackPanel to **LightGray**:

```
<StackPanel DataContext="Test"
            Background="LightGray">
    <TextBox x:Name="Text1"
              Text="{Binding Path=SomeText,
                           Mode=TwoWay}" />
    <TextBox Text="{Binding ElementName=Text1,
                           Path=Text, Mode=OneWay}" />
</StackPanel>
```

3. Save the file and look at the running application:

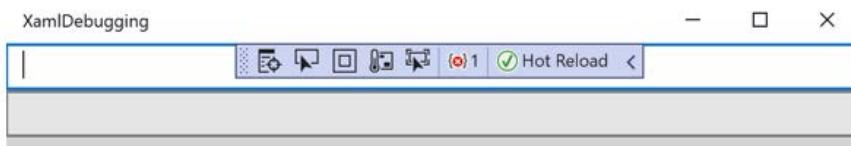


Figure 11.14 – Changing the UI without restarting the application

The background of the window is now light gray.

This is a huge time saver when building a new UI. For a list of known limitations, see this Microsoft Docs article: <https://docs.microsoft.com/en-us/visualstudio/xaml-tools/xaml-hot-reload?view=vs-2019#known-limitations>.

Now let's look at another powerful debugging tool, the **Live Visual Tree** window.

## Debugging with Live Visual Tree and Live Property Explorer

The **Live Visual Tree** window allows you to explore the elements in the current window of your application's XAML **visual tree**. It is available to both UWP and WinUI applications. Let's step through using the Live Visual Tree and related XAML debugging tools:

1. While debugging the XAMLDebugging project, open the Live Visual Tree. The visual tree contains the hierarchy of controls in the window with **Show Just My XAML** selected by default in the **Live Visual Tree** window's toolbar:

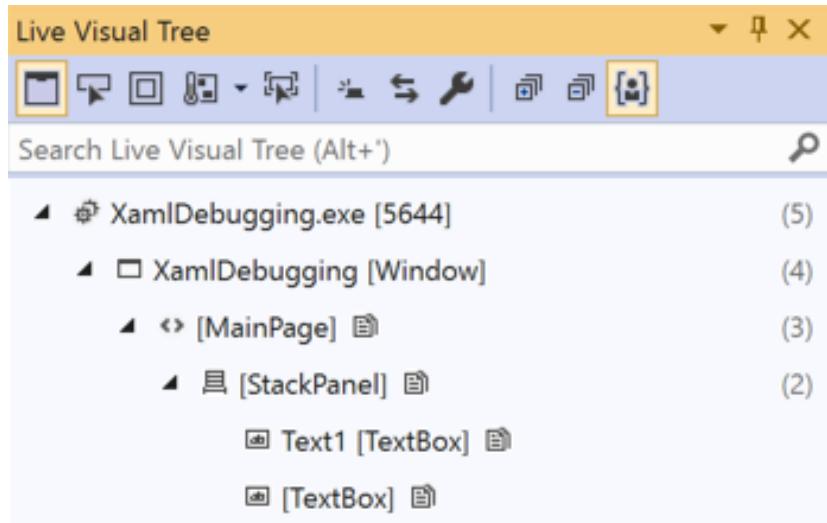


Figure 11.15 – Viewing the Live Visual Tree for the XamlDebugging project (Show Just My XAML)

2. If you want to view the entire visual tree for your window, de-select **Show Just My XAML** on the toolbar and the tree will refresh:

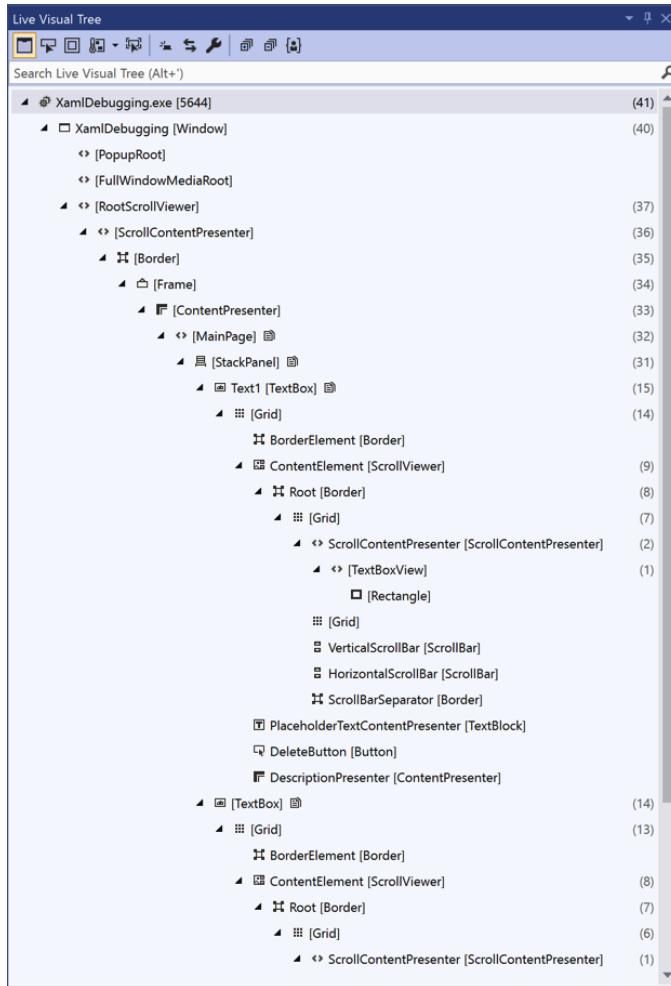


Figure 11.16 – Viewing the Live Visual Tree for the XamlDebugging project (all XAML)

There's a lot more going on here. With **Show Just My XAML** de-selected, you can see the elements that make up each control. These are the contents of the control templates. A `TextBox` is made up of 15 child elements with a `Grid` at the root of its template. This is a great way to learn how the controls we use are composed. Working with and modifying these templates is beyond the scope of this chapter, but I encourage you to explore them on your own. *Chapter 7, Fluent Design System for Windows Applications*, has additional information on using default styles and theme resources. This Microsoft Docs article on control templates is another great place to start: <https://docs.microsoft.com/en-us/windows/uwp/design/controls-and-patterns/control-templates>. For now, let's switch back to the **Show Just My XAML** view.

3. From the **Live Visual Tree**, right-click on a node to navigate to the XAML markup for the selected item by selecting **View Source**.
4. Next, right-click a node and select **Show Properties** to show the **Live Property Explorer** window where you can inspect the current properties of the element. You can see how the properties are grouped based on where they have been set. Here there are some set by **Local** changes and others set based on the **Style** for a **Default TextBox**. If you had multiple levels of inherited explicit styles and an implicit style, those would all be grouped here:

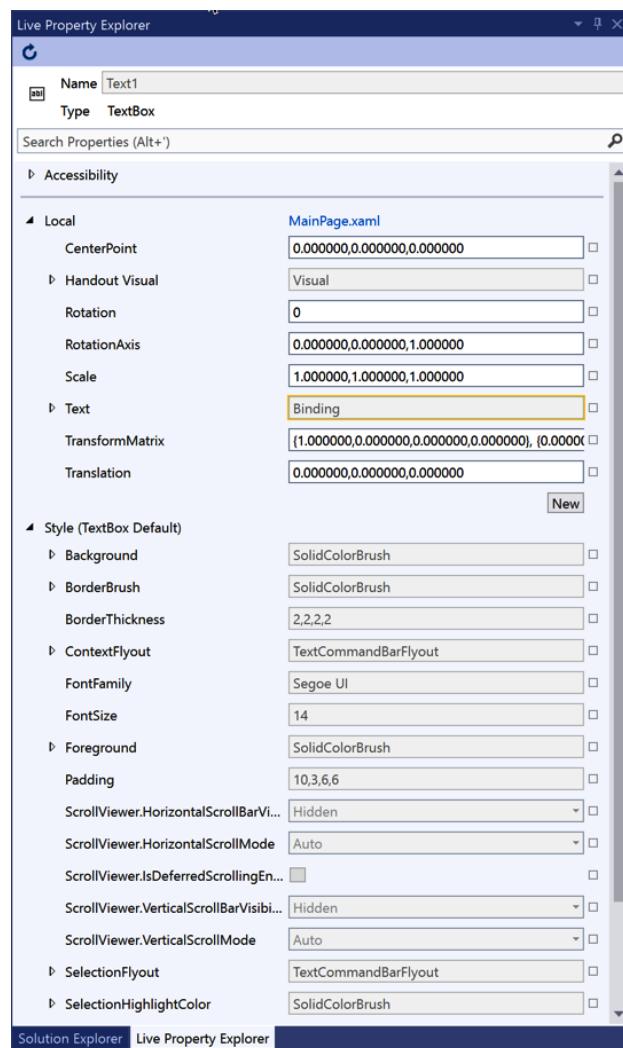


Figure 11.17 – Viewing the Live Property Explorer window for the Text1 TextBox

5. Look at the **Text** property of **Text1**. The value is **Binding**, meaning that the value is set to a binding markup extension. Expand the **Text** property node to view the details of the binding:

▲ Text	Binding
Converter	0
ConverterLanguage	0
ConverterParameter	0
ElementName	0
EvaluatedValue	0
FallbackValue	0
IsBindingValid	False
Mode	TwoWay
Path	SomeText
RelativeSource	0
Source	0
TargetNullValue	0
UpdateSourceTrigger	Default

Figure 11.18 – View the binding details for the Text property of Text1

There is some very helpful information here. We can set the **Path** and **Mode** that we set in the XAML, the ones we did not set, and yet others that are read-only. **IsBindingValid** is **False** in my application, but it may be **True** if you fixed your data context from the previous section. When the binding is valid, **EvaluatedValue** will contain the current value for the **Text** property.

6. Select the **MainPage** element in the tree and view the **Live Property Explorer** window. You can see that the value for **Background** has a green border. This indicates that the value is set by a resource. In this case, a **ThemeResource** is providing the brush.

7. Expand the **Background** node to view the resource details:

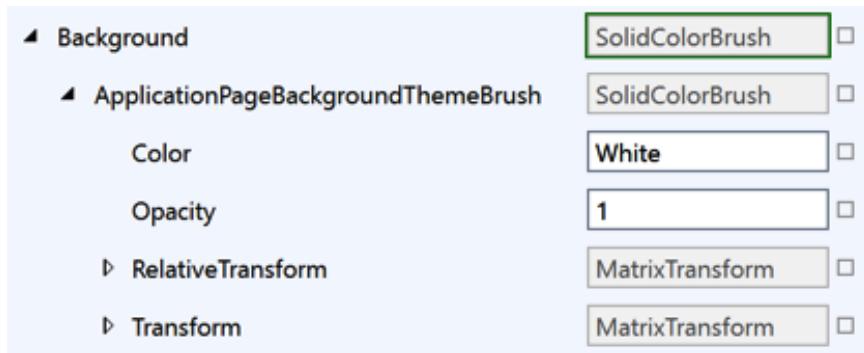


Figure 11.19 – Viewing the property details of the Page.Background

8. Try changing the **Color** property and see what happens in the XAML code and to the running application.
9. Next, try adding markup to `MainPage` to set the `MinWidth` property to 400. What happens in the **Live Property Explorer**? Can you find this new property and value?
10. Returning to the **Live Visual Tree**, you can view the depth of the visual tree to the right of each node's name. Remember that this number is not the true depth of the visual tree when **Show Just My XAML** is enabled. It is the depth of the visible nodes. This is important to note because the depth and complexity of the visual tree impact the performance of the UI rendering. A `Grid` is more complex than a `StackPanel`. You should prefer a `StackPanel` for layout if you are simply laying out a few elements horizontally or vertically. This unidirectional layout is what the `StackPanel` was built to handle. As a rule of thumb, you should use the right control for the job and always keep performance in mind.
11. Then select the active node from the tree in your XAML file and enable **Select Element in the Running Application** from the toolbar. The **Track Focused Element** button on the toolbar will highlight the current tree node in the application's window at runtime. This option is also available from the in-app toolbar.
12. To toggle the in-app toolbar on/off, you can use the **Show Runtime Tools in Application** button on the **Live Visual Tree** toolbar.
13. Finally, if you are working in the active XAML file and want to find the current control in the tree, click the **Find Element from Active Document** button on the toolbar.

I encourage you to spend some time in these windows the next time you are debugging your application. Use them to help find issues with binding, resources, or custom control templates.

Now let's wrap up the chapter and review what we've learned.

## Summary

In this chapter, we have covered some essential tools and techniques for debugging your XAML applications. While the material primarily references UWP, the same will work for WinUI 3 when the Visual Studio XAML tooling is updated with better WinUI support. If you are developing WPF applications, tools such as the Live Visual Tree, Live Property Explorer, and Rapid XAML Toolkit will all work for those projects. Leveraging these tools will shorten the time you spend debugging and help you deliver higher-quality software.

In the next chapter, we will return to the `WebView2` browser control. You will learn how to use `WebView2` to embed an ASP.NET Core **Blazor single-page application (SPA)** inside a WinUI application.

## Questions

1. How can you preview how the current XAML file will appear on an Xbox?
2. How can you debug an application that is currently running on your machine?
3. How can you launch an application package installed on a remote machine for debugging?
4. How can you debug an application on a USB-attached device?
5. Which window will show a hierarchy of the elements in the current window?
6. What is the default binding mode for most control properties with `x:Bind`?
7. Where can you view the runtime properties for the control currently selected in the **Live Visual Tree**?



# 12

# Hosting an ASP. NET Core Blazor Application in WinUI

**Blazor** is a web framework from Microsoft that allows .NET developers to create C# web applications with little to no JavaScript code. Server-side Blazor applications were introduced with ASP.NET Core 3.0, and ASP.NET Core 3.2 added the ability to create client-side Blazor web apps with **WebAssembly (Wasm)**. Wasm (<https://webassembly.org/>) allows runtimes such as .NET and Java to run in web applications in the browser, and it is supported by all the major browser engines. By leveraging the new `WebView2` control in WinUI, Windows developers can run a cloud-hosted Blazor application inside their WinUI client application.

In this chapter, we will cover the following topics:

- Learning some basics of client-side .NET development with ASP.NET Core and Blazor
- Creating a Blazor application with Visual Studio Code
- Deploying Blazor applications to Azure static website hosting
- Creating a WinUI application to host a Blazor application in a `WebView2` browser control

By the end of this chapter, you will understand how to create a new Blazor application, deploy it to the cloud, and use the application within the `WebView2` control in WinUI.

## Technical requirements

To follow along with the examples in this chapter, the following software is required:

- Windows 10 version 1803 (build 17134) or newer.
- Visual Studio 2019 version 16.9 or newer with the following workloads: .NET Desktop Development and Universal Windows Platform Development.
- Visual Studio Code with the following extensions: C# for Visual Studio Code and JavaScript Debugger (Nightly).
- Windows Terminal (which was built with WinUI 2.x) or another command-line tool.
- To create ASP.NET Core Blazor projects, make sure you install .NET Core 3.2 or later. We will be using .NET 5 in the sample code in this chapter.

The source code for this chapter is available on GitHub at this URL: <https://github.com/PacktPublishing/-Learn-WinUI-3.0/tree/master/Chapter12>.

## Getting started with ASP.NET Core and Blazor

Blazor is a web development framework that provides C# developers with an alternative to JavaScript when building client-side web applications. Blazor is part of **ASP.NET Core** and was first introduced with ASP.NET Core 3.0. Let's start by exploring some history of **ASP.NET** and **ASP.NET Core**.

## Exploring some history of ASP.NET and ASP.NET Core

ASP.NET was Microsoft's .NET-based web development framework that was first released in 2002. The early versions of ASP.NET used a client development model called **Web Forms**, which was intended as a web equivalent of Windows Forms client applications. Web Forms was popular with .NET web developers but did not adhere to many web development best practices and patterns. Many were critical of the large amount of **ViewState** data sent over the wire with every server request and response.

In response to Web Forms criticism, the ASP.NET team released ASP.NET MVC in 2009. Web applications built with ASP.NET MVC follow the **Model-View-Controller (MVC)** pattern. The new framework was well-received by the .NET community and is still a popular choice with web developers today. ASP.NET was also one of the first Microsoft frameworks to be released as open source. In 2012, ASP.NET MVC 4 was released as open source under the Apache License 2.0.

As the .NET team continued to embrace open source software, they also decided to make a fresh start with a new, open source and cross-platform version of .NET called **.NET Core**. Microsoft released .NET Core 1.0 in 2016 with runtimes available for Windows, macOS, and Linux. With the release of .NET Core came a new web framework called **ASP.NET Core**. ASP.NET Core 1.0 included project templates to build web applications and Web API projects. The web applications were built with the MVC pattern, the **Razor** syntax for building a rich UI, and CSS for styling pages.

The ASP.NET team continued to add more features to ASP.NET Core over the next several years, including the following:

- **Razor Pages:** Razor Page projects were introduced with ASP.NET Core and offer a simpler alternative to ASP.NET Core MVC.
- **SignalR:** A framework for real-time web communication; SignalR is integral to client-server communication in Blazor server applications.
- **Identity Core:** Supports login functionality in ASP.NET Core applications and manages authentication resources such as users, passwords, roles, tokens, and so on.

#### Note

This book will not provide a detailed tutorial of ASP.NET Core development. If you want to learn more about building web applications with ASP.NET Core, see *ASP.NET Core 5 for Beginners* by *Andreas Helland, Vincent Maverick Durano, Jeffrey Chilberto, and Ed Price*, Packt Publishing (<https://www.packtpub.com/product/asp-net-core-5-for-beginners/9781800567184>).

But where does Blazor fit into the ASP.NET Core development picture? Let's explore that next.

## What is Blazor?

Blazor is a framework for building web applications with .NET Core and C#. There are two hosting models from which developers can choose when starting a new Blazor project:

- **Blazor Server:** Introduced with ASP.NET Core 3.0, the server model executes application logic on the server with UI updates pushed to the client through SignalR connections.
- **Blazor WebAssembly:** Delivered later with ASP.NET Core 3.2, this execution model runs solely on the client, sandboxed and running on the browser's UI thread via Wasm.

So, which hosting model should you choose for your next Blazor application? That decision will depend on your project's requirements. Here are some of the pros and cons of each execution model:

Blazor Hosting Models			
	Blazor Server	Blazor WebAssembly	
<b>Pros</b>	Smaller initial download Full .NET Core API available on server Application code stays on server More robust debugging support Better performance on resource-limited devices	No .NET dependency for server hosting Serverless deployments Workload distributed to clients; higher scalability Leverages client capabilities Offline execution as a Progressive Web Application (PWA)	
<b>Cons</b>	Clients cannot execute offline Higher latency Clients rely on server connections; Less scalability .NET Core server hosting required No serverless	Limited by browser capabilities Wasm support required (No IE) Runtime included in initial download; Slower first load .NET runtime limited on Wasm compared to full .NET Core Debugging tools less mature	

Figure 12.1 – Blazor hosting model pros and cons

The Blazor server hosting model was released first and has the most mature Visual Studio tooling and debugging support. It is a great choice if you plan to host the server on a service that supports ASP.NET Core and your users may be using browsers that do not have Wasm support.

In this chapter, we are going to focus on client-side Blazor applications. So, why choose this model, and how does it work?

## WebAssembly and client-side .NET development

The primary benefits of the client-side hosting model with Wasm are the option of serverless deployment and the ability for clients to work offline. The offline support means your Blazor application can be configured as a **Progressive Web Application (PWA)** and downloaded to PCs, tablets, and phones. Learn more about PWAs from Mozilla's developer documentation: [https://developer.mozilla.org/en-US/docs/Web/Progressive\\_web\\_apps](https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps).

The reasons that Blazor client applications can run as PWAs are the reasons why we want to use it in a `WebView2` control in our WinUI application. Once the web application has loaded in the browser host, all in-memory execution and interactions can occur regardless of any interruptions in network connectivity. If connectivity, scalability, and server hosting are not concerns in your project, then the Blazor server model could certainly be used.

Simple Blazor Wasm applications can be hosted as *static resources* on a web server. You can also host Blazor Wasm applications on ASP.NET Core web hosting solutions. Doing this enables sharing code with other web solutions on the server and more advanced routing scenarios to support deep linking within the application. When clients make their first request to the server, the entire application and .NET runtime are sent to the browser in the response, and the entire application runs on the client side. There is no shared server-side code in this mode. The runtime and application are then loaded on top of Wasm on the UI thread:

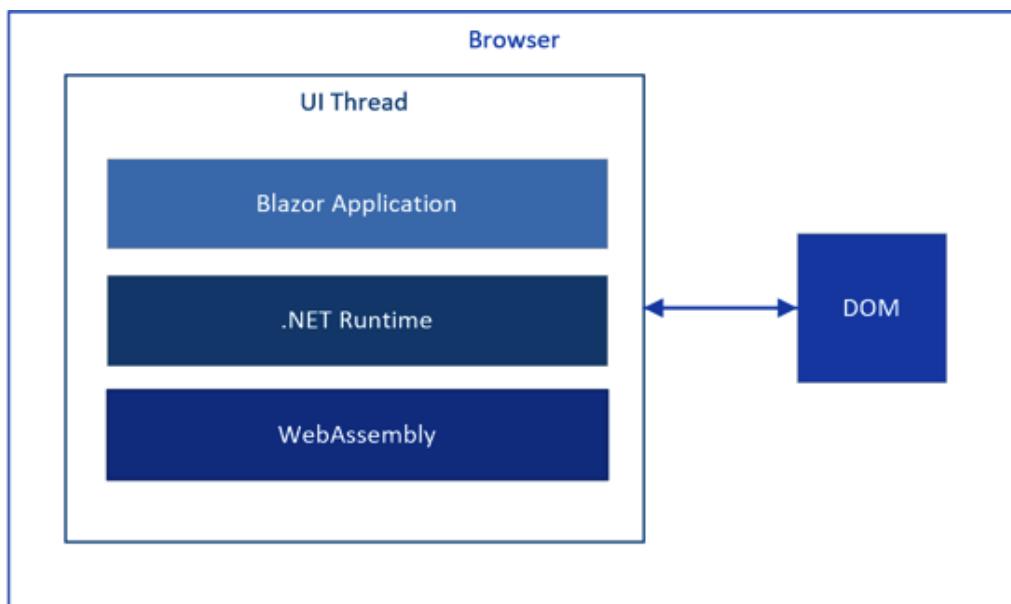


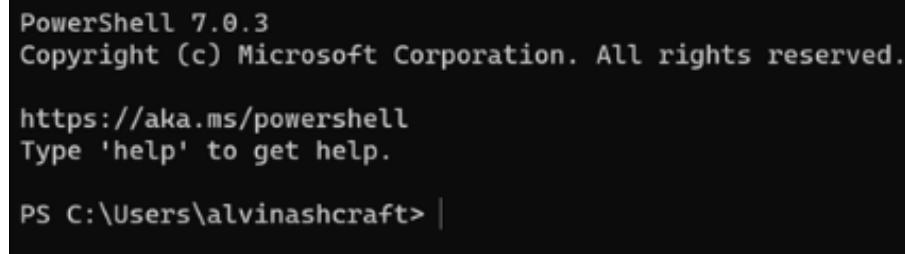
Figure 12.2 – The Blazor Wasm model running in the browser

Now that you have a little background on ASP.NET Core and Blazor applications, let's create a Blazor Wasm project and get some hands-on experience with the framework.

## Creating a Blazor Wasm application

It's time to start building the Blazor application that we'll be running in our WinUI application. We are going to use the .NET **command-line interface (CLI)** and **Visual Studio Code** to create the Blazor project. You can also use Visual Studio 2019 if you prefer the full-featured IDE:

1. Start by opening a Command Prompt with your terminal application of choice. I will be using Windows Terminal (<https://www.microsoft.com/p/windows-terminal/9n0dx20hk701>) with PowerShell 7 (<https://docs.microsoft.com/en-us/powershell/scripting/overview>):



```
PowerShell 7.0.3
Copyright (c) Microsoft Corporation. All rights reserved.

https://aka.ms/powershell
Type 'help' to get help.

PS C:\Users\alvinashcraft> |
```

Figure 12.3 – Running PowerShell 7 in Windows Terminal

2. Use the terminal to change the current folder to the location where you keep your projects. My location will be C:\Users\alvinashcraft\source\repos.
3. Use the following command to create a new Blazor WebAssembly project named `BlazorTasks` and hit *Enter*: `dotnet new blazorwasm -o BlazorTasks`. The .NET CLI will create the new project, and you should see a message indicating it has completed successfully:

```

Telemetry
-----
The .NET tools collect usage data in order to help us improve your experience. The data is anonymous. It is collected by Microsoft and shared with the community. You can opt-out of telemetry by setting the DOTNET_CLI_TELEMETRY_OPTOUT environment variable to '1' or 'true' using your favorite shell.

Read more about .NET CLI Tools telemetry: https://aka.ms/dotnet-cli-telemetry

-----
Installed an ASP.NET Core HTTPS development certificate.
To trust the certificate run 'dotnet dev-certs https --trust' (Windows and macOS only).
Learn about HTTPS: https://aka.ms/dotnet-https

-----
Write your first app: https://aka.ms/dotnet-hello-world
Find out what's new: https://aka.ms/dotnet-whats-new
Explore documentation: https://aka.ms/dotnet-docs
Report issues and find source on GitHub: https://github.com/dotnet/core
Use 'dotnet --help' to see available commands or visit: https://aka.ms/dotnet-cli

-----
Getting ready...
The template "Blazor WebAssembly App" was created successfully.

Processing post-creation actions...
Running 'dotnet restore' on C:\Users\alvinashcraft\source\repos\BlazorTasks\BlazorTasks.csproj...
  Determining projects to restore...
    Restored C:\Users\alvinashcraft\source\repos\BlazorTasks\BlazorTasks.csproj (in 3.86 sec).
Restore succeeded.

PS C:\Users\alvinashcraft\source\repos> |

```

Figure 12.4 – The .NET CLI successfully creates a Blazor WebAssembly project

4. Navigate to the **BLAZORTASKS** folder. If you have **Visual Studio Code (VS Code)** (<https://code.visualstudio.com/>) installed, you can enter code `.` at the command line to open the current folder in Visual Studio Code:

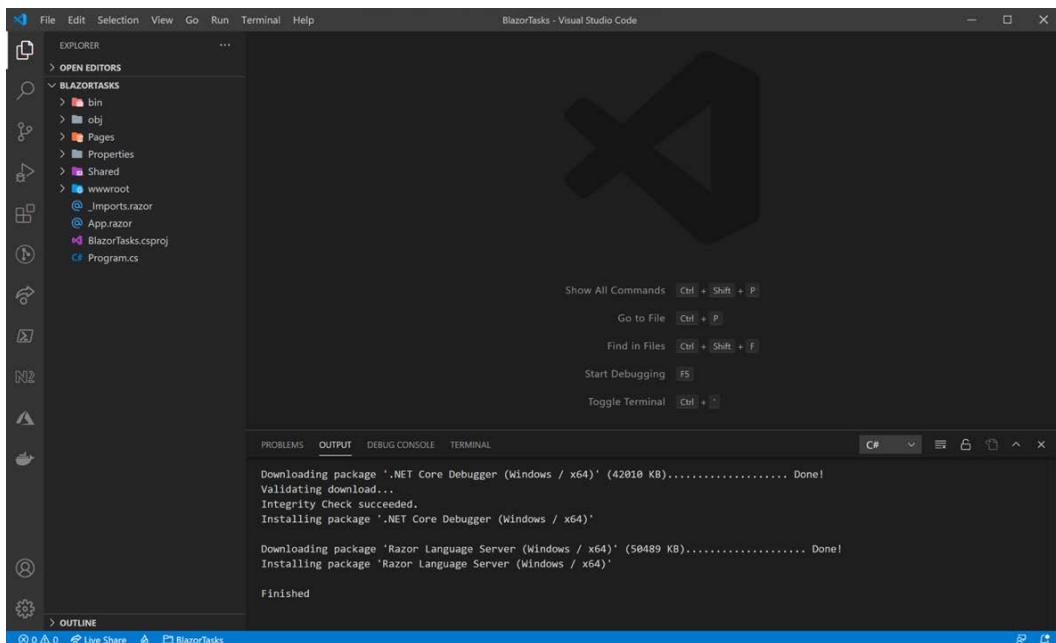


Figure 12.5 – The BlazorTasks project in Visual Studio Code

You may see some activity in the **OUTPUT** window as VS Code downloads some debugging and editing tools relevant to the project.

5. Switch to the **Terminal** window in VS Code. If the window isn't visible at the bottom of your editor, you can click **Terminal | New Terminal** from the menu.
6. Type `dotnet run` in the **Terminal**. (You can also use *F5* to run in VS Code like you are accustomed to using in Visual Studio.) When the compilation completes, you can view the running **BlazorTasks** application by navigating to `https://localhost:5001/` in your browser:

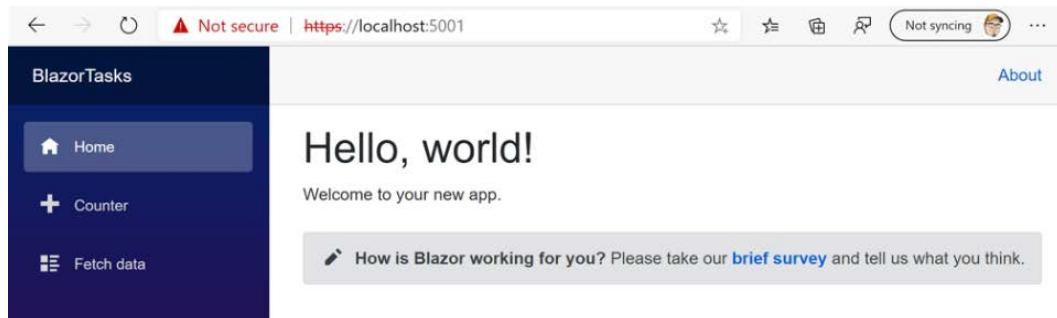


Figure 12.6 – Running the BlazorTasks project for the first time

The default project template has three navigation options in the left pane: **Home**, **Counter**, and **Fetch data**. As you navigate from page to page, all the execution logic is running within the browser.

7. You can open the developer tools in your browser by pressing *F12*. You will see that there is no activity on the **Network** tab of the developer tools while navigating to the **Counter** tab in the application and clicking the **Click me** button several times:

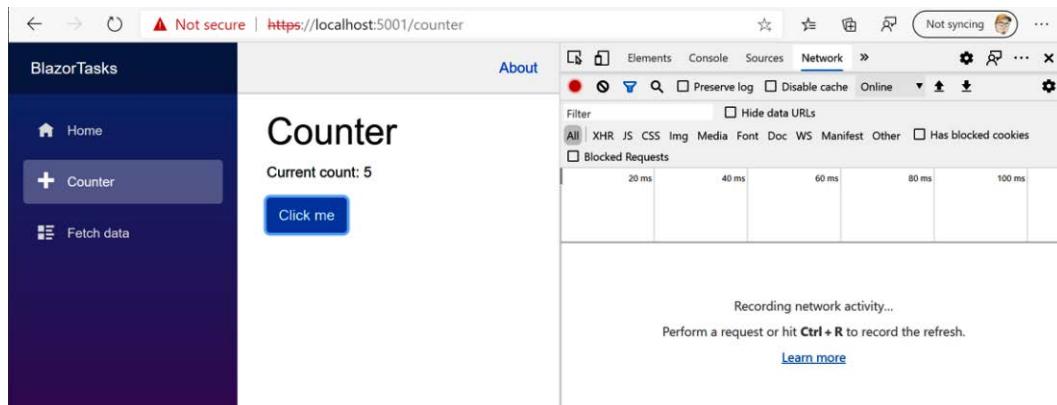


Figure 12.7 – Viewing network activity in the BlazorTasks application

- Finally, when you are done exploring the application, you can close your browser and hit *Ctrl + C* in the VS Code **Terminal** window to stop the application.

Now that we have created and tested the project, let's start coding a new task page for our application.

## Building a simple application for tracking tasks

In this section, we are going to create a new task page for the application that will appear in the left navigation below the **Fetch data** item. If you like, you can remove the other components from the project. I am going to keep them there to test the navigation in the deployed application hosted in WinUI:

- Start by adding a **Tasks** component to the project. Do this by entering `dotnet new razorcomponent -n Tasks -o Pages` in the VS Code **Terminal**. This will create the **Tasks** Razor component in the **Pages** folder.
- Double-click **Tasks.razor** in the **Pages** folder in the **Explorer** window to open it in the editor. The file contains the following code:

```
<h3>Tasks</h3>

@code {

}
```

Razor files contain a combination of HTML markup and C# code, with the HTML at the top of the file, and the C# inside the `@code` block at the bottom of the file. We'll see how these two sections can interact as we move along.

- Add `@page "/tasks"` as the first line of the **Tasks.razor** file. This will enable the app to route to the page using `/tasks` on the URL.
- Before we add the page contents, let's add the new navigation item for it. Open **NavMenu.razor** from the **Shared** folder in **Explorer**.
- Inside the unordered list, add a new list item before the closing `</ul>` tag:

```
<div class="@NavMenuCssClass"
    @onclick="ToggleNavMenu">
    <ul class="nav flex-column">
        ...
        <li class="nav-item px-3">
```

```
<NavLink class="nav-link" href="tasks">
    <span class="oi oi-list-rich" aria-
        hidden="true"></span> Tasks
</NavLink>
</li>
</ul>
</div>
```

6. Run the application with `dotnet run` to make sure the new menu option appears and can navigate to the new page with the **Tasks** header:

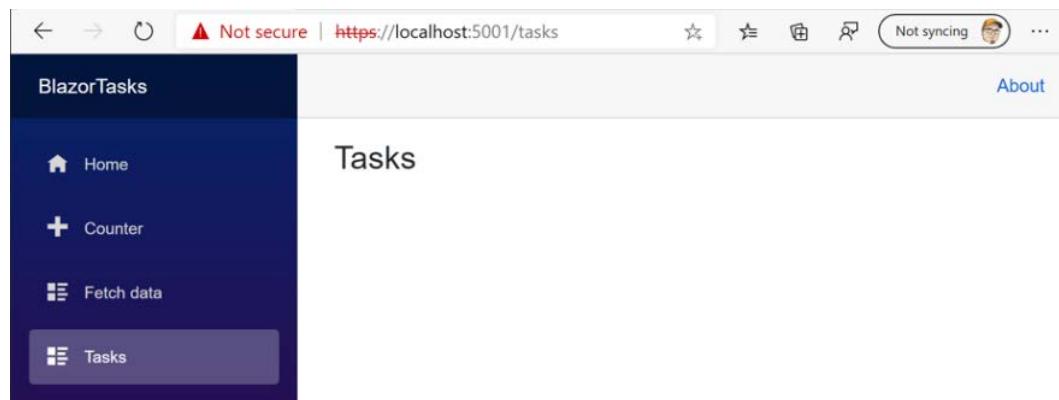


Figure 12.8 – Navigating to the new Tasks page

7. Next, use **File | New File** at the root of the project and name the file `TaskItem.cs`. This will be the model class for tasks:

```
namespace BlazorTasks
{
    public class TaskItem
    {
        public string Name { get; set; }
        public bool IsComplete { get; set; }
    }
}
```

8. Open **Tasks.razor** and add the following code to create an unordered list of tasks by iterating over a list of tasks contained in the `@code` block:

```
@page "/tasks"

<h3>Tasks</h3>

<ul>
    @foreach (var task in taskList)
    {
        <li>@task.Name</li>
    }
</ul>
<input placeholder="Enter new task..." />
<button>Add task</button>

@code {
    private IList<TaskItem> taskList = new
    List<TaskItem>();
}
```

Notice how Razor files allow you to blend C# code and HTML markup. We have a C# `foreach` within `<ul>`, and inside the `foreach`, we're adding `<li>` elements that again contain C# code to get each `task.Name`. This is powerful stuff. We've also added an `input` field to enter a new task and a `button` to add the task. We'll add some code to make the `button` functional next.

9. Add a `newTask` private variable and a new method to the `@code` block named `AddTask`. This method will add a new task to the `taskList` collection:

```
private string newTask;
private void AddTask()
{
    if (!string.IsNullOrWhiteSpace(newTask))
    {
        taskList.Add(new TaskItem { Name = newTask });
        newTask = "";
    }
}
```

10. Finally, add some data binding code to the `input` and `button` elements on the page. The `input` will bind to the `newTask` variable, and the `onclick` event of the `button` will trigger the `AddTask` method to run:

```
<input placeholder="Enter new task..." @bind="newTask"
/>
<button @onclick="AddTask">Add task</button>
```

11. Now, run the application and test the controls. You should be able to add some tasks to the list:

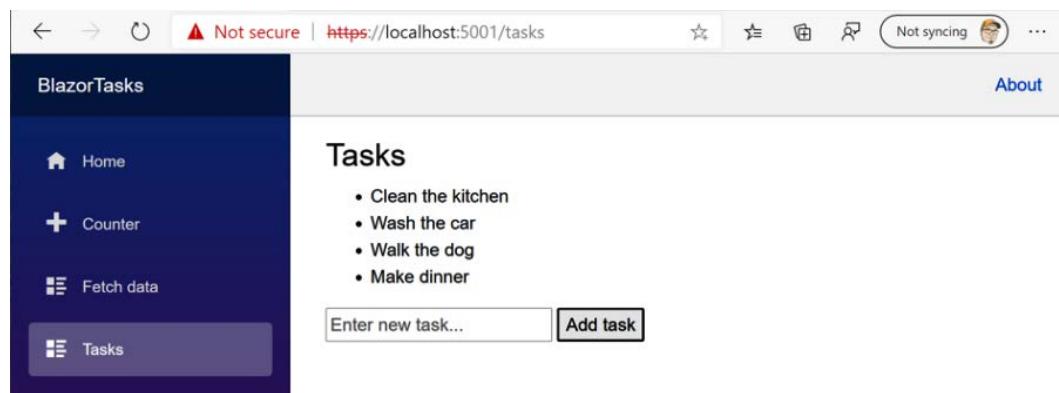


Figure 12.9 – Adding some tasks to the task list in BlazorTasks

This works great, but now that we have some tasks to do, we don't have any way to mark them as done. Let's take care of that next:

1. The first step is to make each list item a `checkbox` that users can check when they complete a task. We are also binding the `task.Name` to an `input` field so users can edit the name of each task:

```
<ul>
    @foreach (var task in taskList)
    {
        <li>
            <input type="checkbox"
                @bind="task.IsComplete" />
            <input @bind="task.Name" />
        </li>
    }
</ul>
```

```

    </li>
}
</ul>

```

2. Next, in case the list gets lengthy, let's use some data binding to display the number of incomplete tasks as part of the page header:

```

<h3>Tasks - (@taskList.Count(task =>
    !task.IsComplete)) Incomplete</h3>

```

3. Run the application again, and start working on your task list:

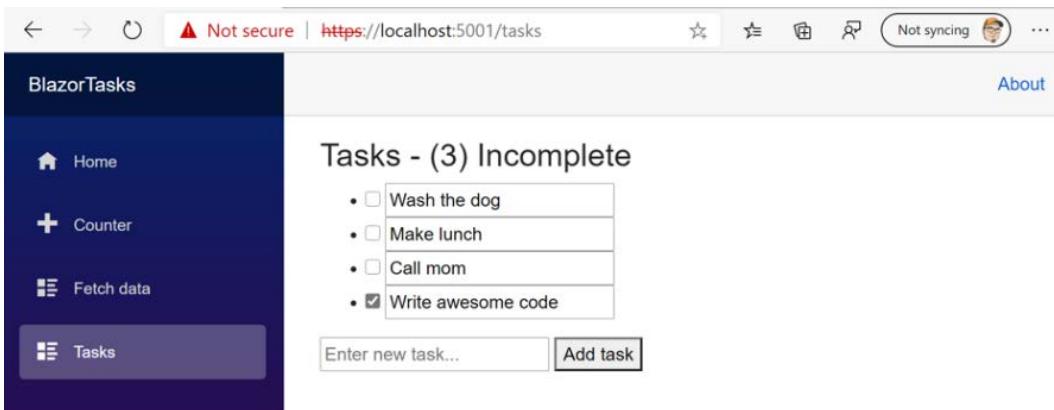


Figure 12.10 – Adding and completing tasks in the BlazorTasks application

You may have noticed that the tasks do not save between sessions. The `taskList` is an in-memory collection for now. To persist it between sessions, you would need to add service calls to save the data in a server-side data store. Creating this service is beyond the scope of this chapter, and I will leave it as an exercise for you.

#### Note

All of these steps can also be taken with Visual Studio or Visual Studio for Mac, if you prefer using a full-featured IDE. This Microsoft Docs page has information on how to debug a Blazor Wasm app in both of these tools, as well as VS Code: <https://docs.microsoft.com/en-us/aspnet/core/blazor/debug>.

Now that we have a functional task tracking web client, we can move on to the next step. It's time to deploy our Blazor app to the cloud.

## Exploring Blazor Wasm deployment options

Running and debugging the Blazor project locally is great while we're developing the solution, but when it's time to share your application with the world, we will need to host it in the cloud. There are many cloud hosting options for typical ASP.NET Core applications, and Blazor Wasm applications have even more. Sites that run entirely on the client can be hosted as static files on the server, meaning that the server simply serves up the files when it receives a request. There is no server-side execution required.

Let's start by reviewing some of the available hosting options for Blazor WebAssembly deployments.

## Deployment options for Blazor Wasm projects

There are several hosting options for our Blazor project. We are going to discuss a few of the most popular solutions today: **GitHub Pages**, **Azure App Service**, **Azure Static Web Apps**, and two options on **Amazon Web Services (AWS)**. For an in-depth exploration of options either hosted with ASP.NET Core or as static files, Microsoft Docs has a great article: <https://docs.microsoft.com/en-us/aspnet/core/blazor/host-and-deploy/webassembly>.

### Amazon Web Services (AWS)

With AWS, a Blazor Wasm site can be hosted with ASP.NET Core in **Elastic Container Service (ECS)** (<https://aws.amazon.com/ecs/>) and **Fargate**. The ECS solution uses Docker to create the container to be hosted in the cloud. The site is then served through Fargate (<https://aws.amazon.com/fargate/>), the AWS compute engine for containers. To read more about this solution for ASP.NET Core projects, the AWS blog has a great article detailing the steps: <https://aws.amazon.com/blogs/compute/hosting-asp-net-core-applications-in-amazon-ecs-using-aws-fargate/>.

For a static hosting option with AWS, static pages can be hosted using **Amazon S3** storage (<https://aws.amazon.com/s3/>) and **CloudFront** (<https://aws.amazon.com/cloudfront/>). An Amazon S3 bucket is a cloud file storage solution. Your wwwroot folder will be copied to S3 storage, and CloudFront handles serving the static files from the S3 bucket. This article details how to create and deploy a Blazor Wasm application in AWS: <https://aws.amazon.com/blogs/developer/run-blazor-based-net-web-applications-on-aws-serverless/>.

Now let's see how to serve static files through **GitHub**.

## GitHub Pages

**GitHub Pages** (<https://pages.github.com/>) are static websites served directly from GitHub repositories. You can maintain your site on GitHub and configure **GitHub Actions** to deploy the site to GitHub Pages. Microsoft MVP Davide Guida has a step-by-step guide for deploying Blazor Wasm projects to GitHub Pages on his blog: <https://www.davideguida.com/how-to-deploy-blazor-webassembly-on-github-pages-using-github-actions/>. GitHub Pages are free, but *standard* user accounts can only host pages from the *default* GitHub branch.

In the next section, we will be using GitHub Actions with our project to deploy to Azure. But now, let's review two of the available Azure hosting solutions.

## Azure App Service

**Azure App Service** (<https://azure.microsoft.com/en-us/services/app-service/>) is a great option to use if you want your Blazor app hosted on an ASP.NET Core web server. There are Windows and Linux servers available with App Service. Microsoft Docs has extensive documentation on deploying ASP.NET Core applications to App Service: <https://docs.microsoft.com/en-us/aspnet/core/host-and-deploy/azure-apps/>.

Now, let's look at another Azure option. This one is specifically for deploying static sites such as Blazor Wasm.

## Azure Static Web Apps

**Azure Static Web Apps** (<https://azure.microsoft.com/en-us/services/app-service/static/>) is a service to host and serve static web applications such as Blazor Wasm. It offers easy deployment through GitHub Actions, free SSL certificates, custom domains, and integration with **Azure Functions**.

**Note**

Azure Static Web Apps is still in preview at the time of writing.

For full documentation on Static Web Apps, including information about using it with other **Single-Page Application (SPA)** websites, Microsoft Docs has guides and **Learn** (<https://docs.microsoft.com/en-us/learn/>) content available: <https://docs.microsoft.com/en-us/azure/static-web-apps/>.

We are going to use Static Web Apps to host our Blazor application. Let's do that now!

## Publishing Blazor to Azure Static Web Apps hosting

In this section, we are going to host our BlazorTasks application in the cloud by pushing the source to GitHub, creating an Azure Static Web App, and configuring GitHub Actions to publish the app to Azure with every commit to the main branch. Let's start by pushing our code to GitHub.

### Pushing the project to GitHub

To push your code to a GitHub repository, you can either use the Git CLI (<https://git-scm.com/downloads>) or the GitHub Desktop (<https://desktop.github.com/>) application. We will use GitHub Desktop in this example:

1. Download and install GitHub Desktop. When the installation completes, launch the application.

2. If your local project is not part of a Git repository yet, choose **File | New Repository**. If you already have a local repository for your project, you can skip to the next step:

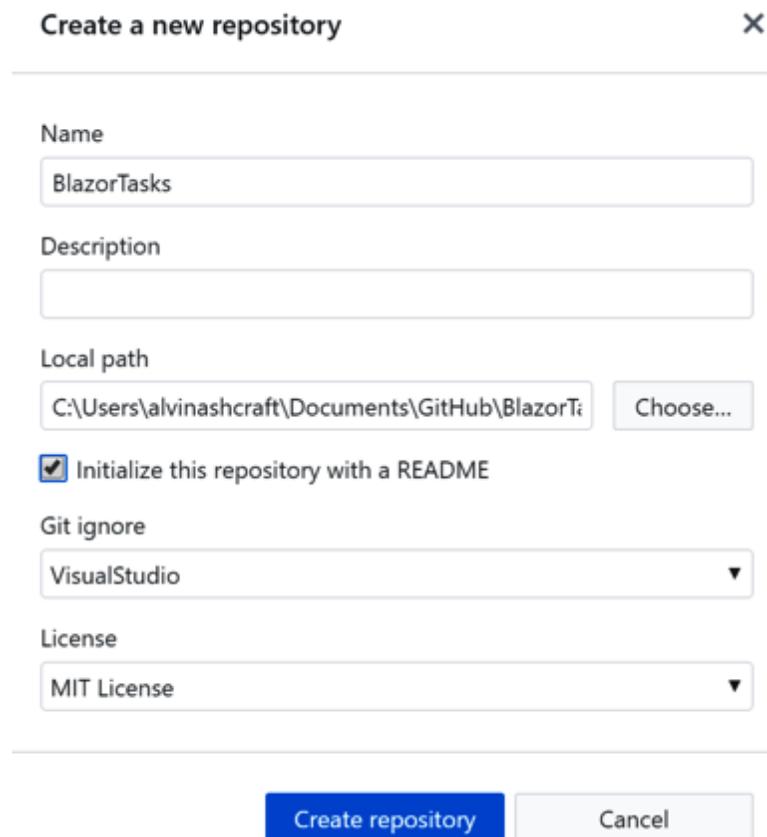


Figure 12.11 – Creating a new local GitHub repository for the BlazorTasks application

Name the repository **BlazorTasks**, optionally add a description, and browse to **Local Path** for your project. It is a good practice to have a **README**, **Git ignore** file, and **License**. So, choose each of these options. When you're done, click **Create repository**. After completing this step, move ahead to *Step 4*.

3. If you created your Blazor project in a local Git repository, you can select **File | Add Local Repository**. Browse to the folder where you have the **BlazorTasks** project and select it. If you do not have a Git repository there yet, the application will prompt you to create one:

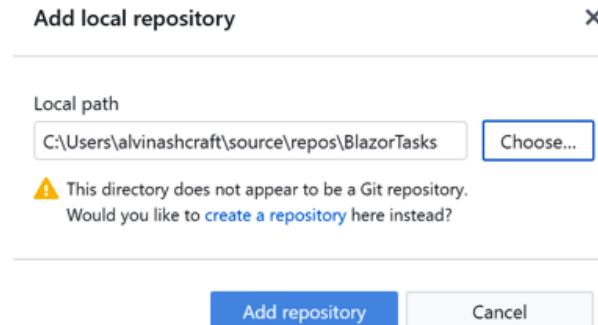


Figure 12.12 – Adding a local repository

4. In this step, we will publish the local repository to GitHub. If you don't have a GitHub account, you can create one at <https://github.com/>. When you're ready to go, make sure **BlazorTasks** is selected as the **Current repository** and click the **Publish repository** button:

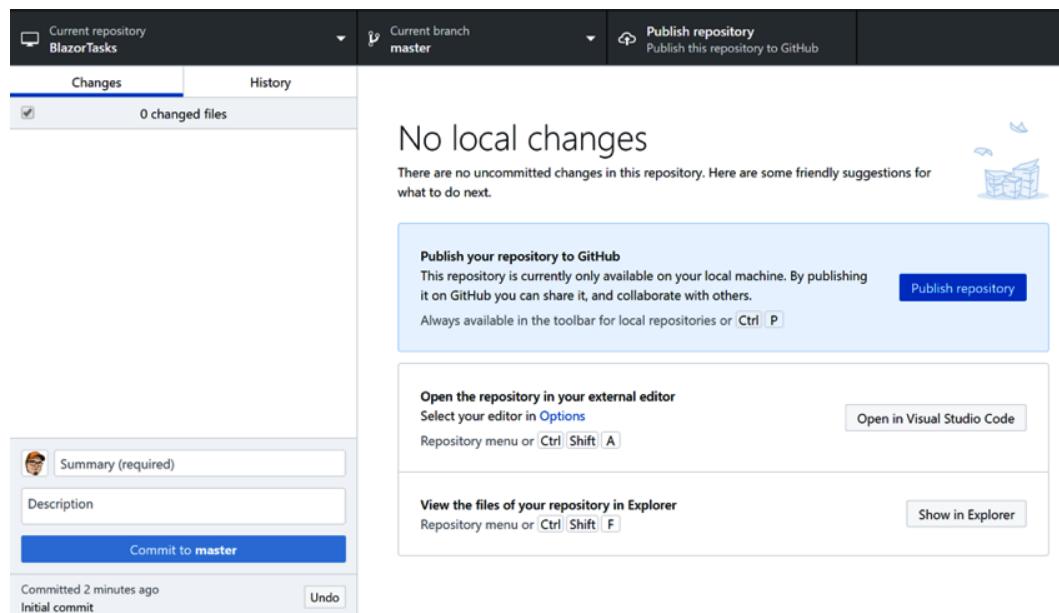


Figure 12.13 – Publishing the local repository to GitHub

5. View the repository on GitHub to make sure it has been published correctly:

The screenshot shows the GitHub repository for 'BlazorTasks'. The repository has 1 commit, 1 star, and 0 forks. The 'About' section describes it as a 'Blazor WebAssembly site to create and manage tasks'. The 'Languages' section shows HTML at 57.9%, CSS at 33.3%, and C# at 8.8%.

Figure 12.14 – The BlazorTasks code on GitHub

Now the code is ready to be published to Azure. Let's do that next.

## Creating an Azure Static Web Apps resource

Let's walk through creating a new Azure Static Web App:

1. To start, if you don't have an Azure account yet, you can create a free trial account at <https://azure.microsoft.com/>. The site will walk you through the steps to create a new account.
2. Log in to the Microsoft account associated with your Azure account at <https://portal.azure.com/>.
3. From the portal home page, click **Create a resource** under **Azure services**.

4. On the **New** page, search for **static** and select **Static Web App (preview)**:

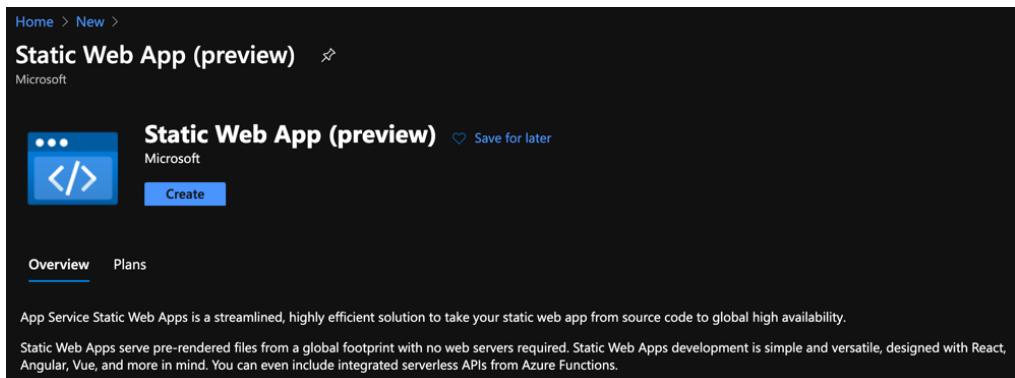


Figure 12.15 – Creating a new Static Web App

5. Click **Create**. On the **Create Static Web App (Preview)** page, select your free trial subscription. You will also need to select a **Resource Group**. Resource groups are a way to group a related set of Azure resources. Create a new **Resource Group** and name it **BlazorTasks**. Give the resource a **Name**, select a **Region** that makes sense for you or your users, and select the **Free** option for **SKU**:

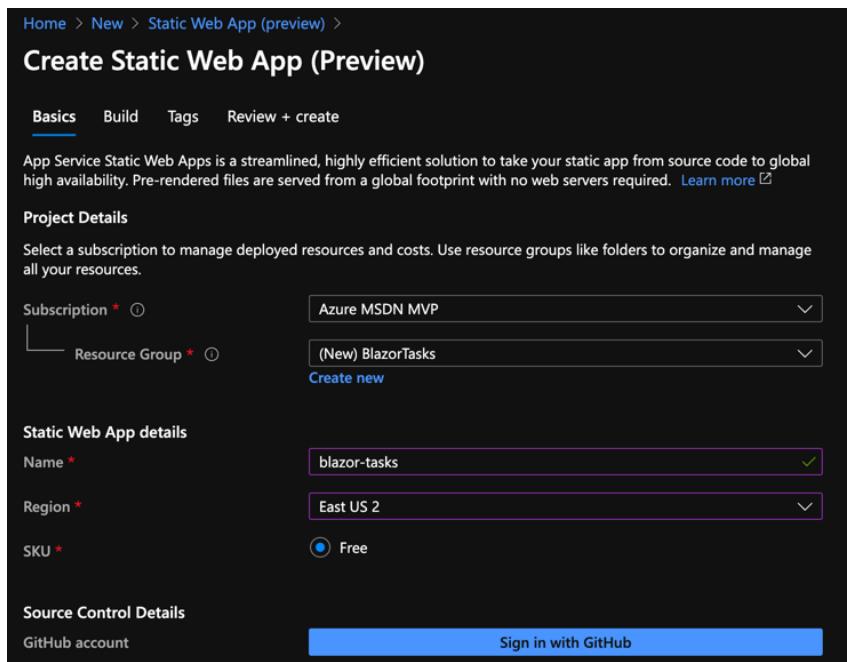


Figure 12.16 – Configuring the new Static Web Apps resource

6. Next, click the **Sign in with GitHub** button to link your GitHub account. Linking the accounts is necessary for the GitHub files to be deployed to the Azure site. After it has linked your accounts, select the **Organization**, **Repository**, and **Branch** names for your BlazorTasks repository:

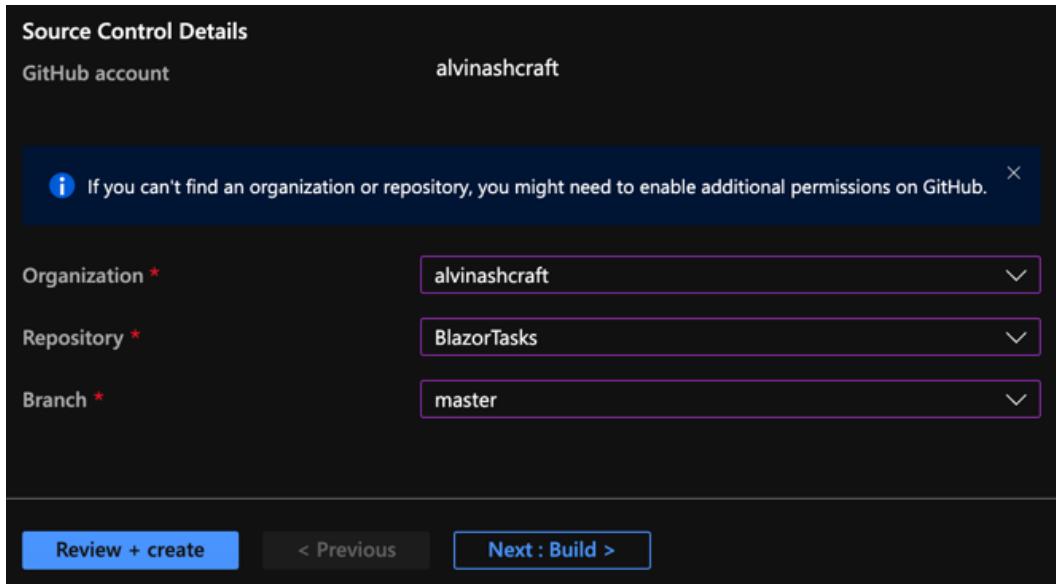


Figure 12.17 – Enter GitHub details for the Static Web Apps resource

7. Click **Next: Build** and enter the **Build Details**. The **App location** will be `bin/Release/netstandard2.1/publish/wwwroot`. You can leave the optional **Api location** and **App artifact location** fields blank.
8. Click **Review + create**. Review the **Summary** page to make sure everything looks correct and click **Create**. Azure will take a couple of minutes to create the resource.

With the Static Web Apps resource ready to go, the next thing we need to do is publish it from GitHub to Azure.

## Publishing an application with GitHub Actions

Now we are ready to configure GitHub Actions to build our project in the GitHub repository and publish it to the Azure resource:

1. Navigate to your project on GitHub and click the **Actions** tab. You will see that Azure has created a workflow named **Azure Static Web Apps CI/CD** and the first build attempt has failed. Let's fix that:

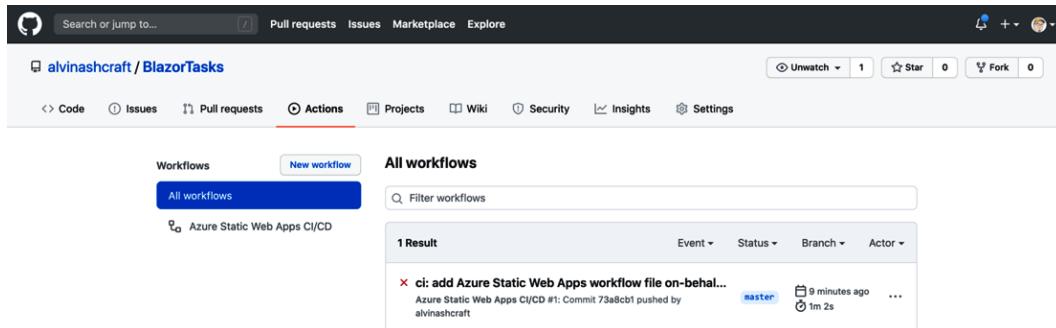


Figure 12.18 – Viewing the workflows for BlazorTasks

2. Select the **Azure Static Web Apps CI/CD** workflow and click the **Workflow file** tab. The **.yml** file will appear in an editor.
3. Edit the file by clicking the pencil icon at the upper right of the editor. We are going to add some steps to `build_and_deploy_job`:

```

jobs:
  build_and_deploy_job:
    if: github.event_name == 'push' ||
      (github.event_name == 'pull_request' &&
       github.event.action != 'closed')
    runs-on: ubuntu-latest
    name: Build and Deploy Job
    steps:
      - uses: actions/checkout@v2
        with:
          submodules: true
      - name: Setup .NET Core
        uses: actions/setup-dotnet@v1.6.0
        with:

```

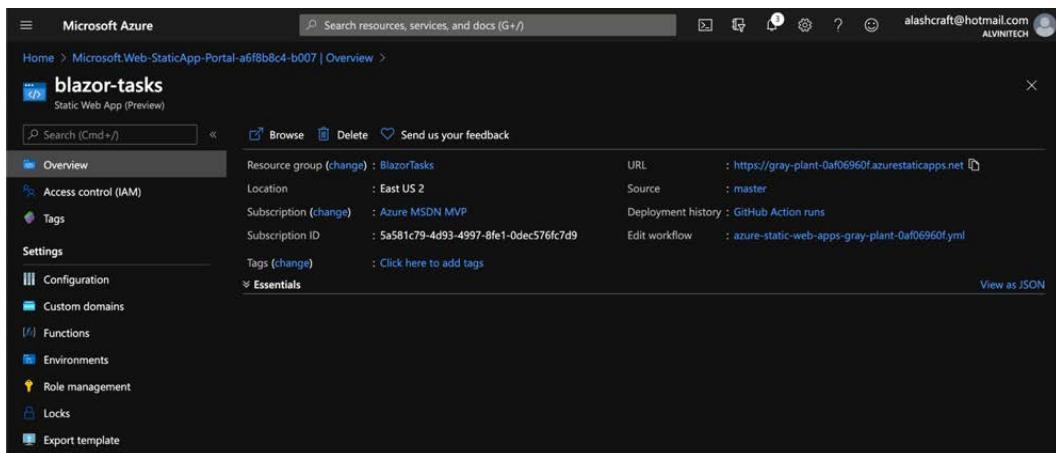
```

dotnet-version: 3.1.x
  - name: Build with dotnet
    run: dotnet build --configuration Release
  - name: Publish with dotnet
    run: dotnet publish --configuration Release
  - name: Publish artifacts
    uses: actions/upload-artifact@master
    with:
      name: webapp
      path:
        bin/Release/netstandard2.1/publish/wwwroot
  - name: Build and Deploy
    id: builddeploy
    uses: Azure/static-web-apps-deploy@v0.
      0.1-preview
    with:
      ...
      ##### End of Repository/Build Configurations
#####

```

After you make the edits, commit the file. Another build will queue. When it completes, it should be successful.

4. You can verify that the site has published to Azure by navigating to the **BlazorTasks** resource in the Azure portal and clicking on the URL for the site:



The screenshot shows the Microsoft Azure portal interface. At the top, there's a navigation bar with 'Microsoft Azure' and a search bar. Below the search bar, the URL 'https://gray-plant-0af06960f.azurestaticapps.net' is shown. The main content area is titled 'blazor-tasks' and is described as a 'Static Web App (Preview)'. On the left, there's a sidebar with 'Overview' selected, along with other options like 'Access control (IAM)', 'Tags', 'Settings', 'Configuration', 'Custom domains', 'Functions', 'Environments', 'Role management', 'Locks', and 'Export template'. The main content area shows the following details:

Setting	Value
Resource group (change)	BlazorTasks
Location	East US 2
Subscription (change)	Azure MSDN MVP
Subscription ID	5a581c79-4d93-4997-8fe1-0dec576fc7d9
Tags (change)	Click here to add tags
Deployment history	GitHub Action runs
Edit workflow	azure-static-web-apps-gray-plant-0af06960f.yml

Figure 12.19 – The BlazorTasks resource home page in the Azure portal

5. The Blazor site will open in a new tab in your browser. Click the **Tasks** item in the navigation menu and verify that the application works as expected:

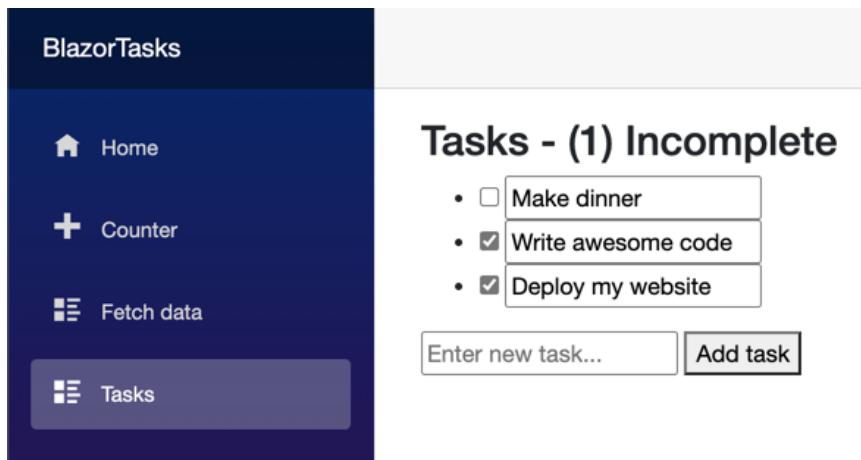


Figure 12.20 – Running BlazorTasks in the static cloud

We have a public-facing static website running the Blazor Wasm application. Now we're ready to run the web application inside a WinUI project.

## Hosting your Blazor application in the WinUI WebView2

We're on the home stretch. We created a Blazor Wasm application, pushed the source code to GitHub, and configured GitHub Actions to publish the application to Azure Static Web Apps with every commit. The last step is to create a simple WinUI project and add a **WebView2** control to the **MainPage**:

1. You can start by either creating a new **Blank App (WinUI in UWP)** named **BlazorTasksHost** in Visual Studio or opening the starter project from GitHub: <https://github.com/PacktPublishing/-Learn-WinUI-3.0/tree/master/Chapter12/Start/BlazorTasksHost>.
2. Open **MainPage.xaml** and update the page to contain a **Grid** that contains the **WebView2** control. Set the **Source** property to the URL of your BlazorTasks site:

```
<Grid>
    <WebView2 Source="https://gray-plant-
        0af06960f.azurestaticapps.net/" />
</Grid>
```

3. Remove the unused button click event handler in **MainPage.xaml.cs** to prevent compilation errors.
4. Finally, run the application, and you will see your BlazorTasks application load as if it were a Windows application:

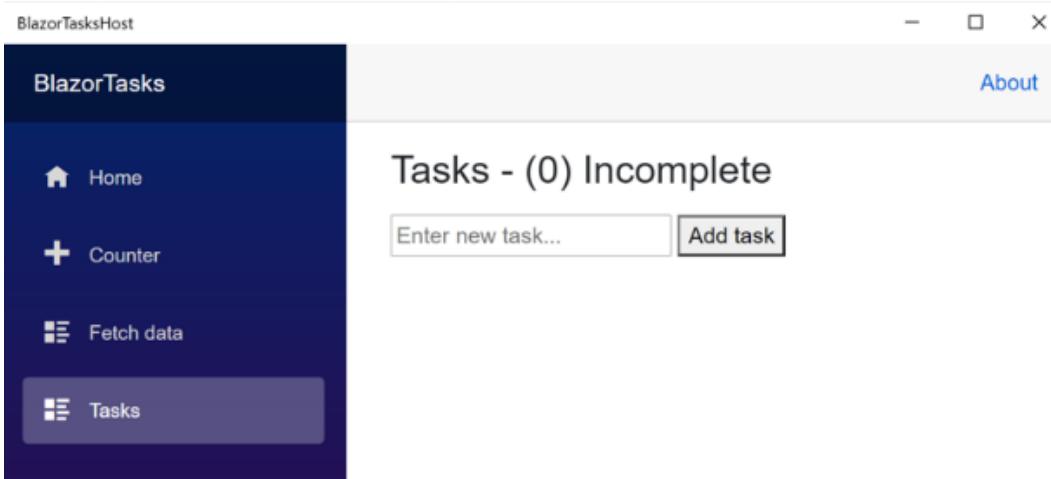


Figure 12.21 – Running BlazorTasks in a WinUI application

Now any updates you make to the Blazor application will be immediately pushed to all your users when you commit to GitHub. This is a compelling way for web developers to reach more Windows users.

#### Note

If you would like to explore Blazor and WinUI integration further, you can check out this blog post by Thomas Claudius Huber. In the post, he experiments with calling a method in the Blazor app from the WinUI host application by executing a script through the WebView2 control: <https://www.thomasclaudiushuber.com/2020/02/18/hosting-blazor-app-in-winui-3-with-webview2-and-call-blazor-component-method-from-winui/>.

Let's wrap up with a summary of what we've covered in this chapter.

## Summary

In this chapter, we learned about ASP.NET Core Blazor. You created a simple task-tracking application with Blazor Wasm and published it to Azure Static Web Apps with GitHub Actions. From here, you could use ASP.NET Core Identity to integrate an application login and save the task data to **SQL Azure** or **Azure Cosmos DB**, personalizing the task list for each user. We created a WinUI client application to run the Blazor client on Windows, but you could also send users directly to your site or create a PWA for desktop and mobile clients. For more information about creating a PWA with Blazor WASM, check out this Microsoft blog post: <https://devblogs.microsoft.com/visualstudio/building-a-progressive-web-app-with-blazor/>.

In the next chapter, we will explore the developer services offered by **Visual Studio App Center**.

## Questions

1. What is the name of the Blazor hosting mode that runs all application logic in the browser?
2. Which Blazor hosting mode is less scalable?
3. What is the name of the syntax used in Blazor UI files?
4. Which .NET CLI command will compile and run the project in the current folder?
5. What is the name of GitHub's product that hosts static websites?
6. Which Azure product hosts static websites?
7. What WinUI 3 control can load web content in a Chromium-based browser control?
8. What is the name of GitHub's **Continuous Integration/Continuous Delivery (CI/CD)** solution?

13

# Building, Releasing, and Monitoring Applications with Visual Studio App Center

**Visual Studio App Center** provides a central place in the cloud to build, test, deploy, and monitor applications online. WinUI developers can add **Continuous Integration (CI)** capabilities to their projects, push beta versions to users with **Continuous Delivery (CD)**, and manage release delivery through the App Center portal. Instrumenting your code to get real-time analytics is a breeze too. With a few lines of code, App Center can provide real-time app monitoring and analytics as well as access to crash reports from beta or production users.

In this chapter, we will cover the following topics:

- Learning to set up an App Center account and create your first project
- Learning to integrate App Center with code repositories to set up CI builds
- How to get your application to beta testers for early feedback
- How to integrate App Center monitoring and analytics with your application to collect real-time data regarding app stability and feature use

By the end of this chapter, you will understand how to get Visual Studio App Center configured to build, deploy, and monitor your applications.

## Technical requirements

To follow along with the examples in this chapter, the following software is required:

- Windows 10 Version 1803 (build 17134) or newer.
- Visual Studio 2019 Version 16.9 or newer with the following workloads: .NET Desktop Development, and Universal Windows Platform Development.
- To create desktop WinUI projects, you must also install the latest .NET 5 SDK.

The source code for this chapter is available on GitHub at this URL: <https://github.com/PacktPublishing/-Learn-WinUI-3.0/tree/master/Chapter13>.

## Getting started with Visual Studio App Center

Visual Studio App Center is Microsoft's cloud **DevOps** solution for app developers targeting Windows, macOS, iOS, and Android devices. Within App Center, developers can configure builds, run automated tests, deploy to app stores and beta users, and review app analytics and crash reports. App Center is designed to help developers quickly build, deploy, and monitor their applications. Over time, most applications will likely migrate to Azure DevOps or GitHub Actions, which both provide a more comprehensive solution.

App Center has free and paid feature levels available. Free App Center accounts have the following limitations:

- **Build:** Limited to 240 build minutes per month, and each build has a maximum 30-minute run time.
- **Test:** Free accounts have a 30-day free trial of App Center's automated UI testing solution.

- **Distribution:** Unlimited distributions and users are available to free accounts.
- **Analytics:** All features are available to free accounts with no limits.
- **Crash reports:** All features are available to free accounts with no limits.

The only major limitation for individuals or small organizations with a limited number of applications is the 30-day trial for UI automation testing. Adding unlimited build time and concurrent builds currently costs \$40 per month, and cloud automation testing costs \$99 per concurrent test device per month.

Let's get started by reviewing the website and creating an account.

## Creating an App Center account

We're going to start our App Center journey by creating a new account on the site. From there, we'll be able to add applications to build, test, and deploy:

1. Start by navigating to the App Center home page (<https://visualstudio.microsoft.com/app-center/>):

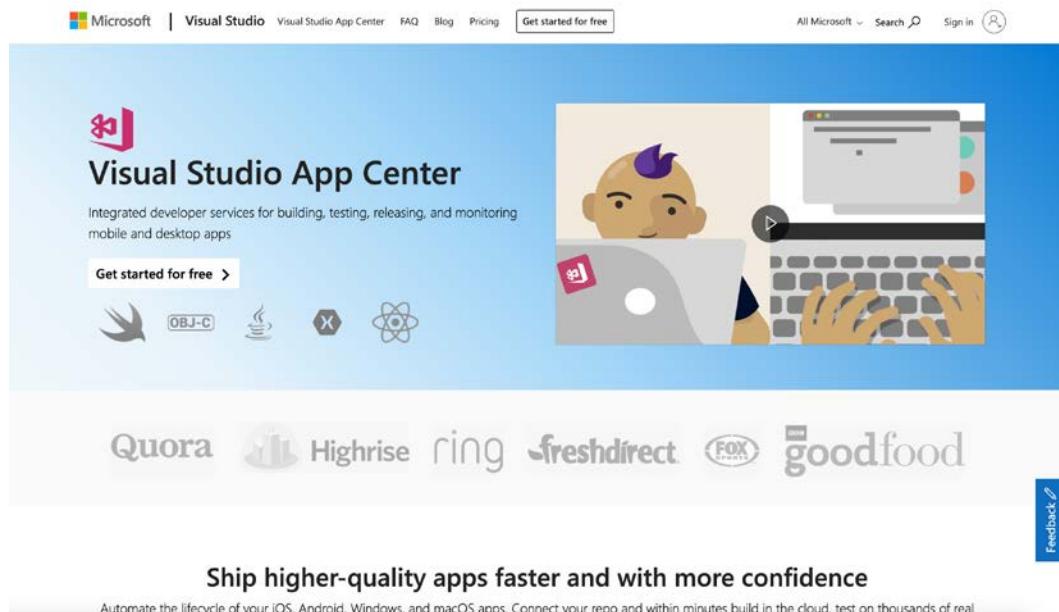


Figure 13.1 – The Visual Studio App Center home page

2. Click **Get started for free** to navigate to the account creation page.

3. The **Create your free account** page has options to create an account with GitHub, Microsoft, Facebook, or Google logins. Select the one you would like to use to continue. If you already have an App Center account, you can click **Sign in**:

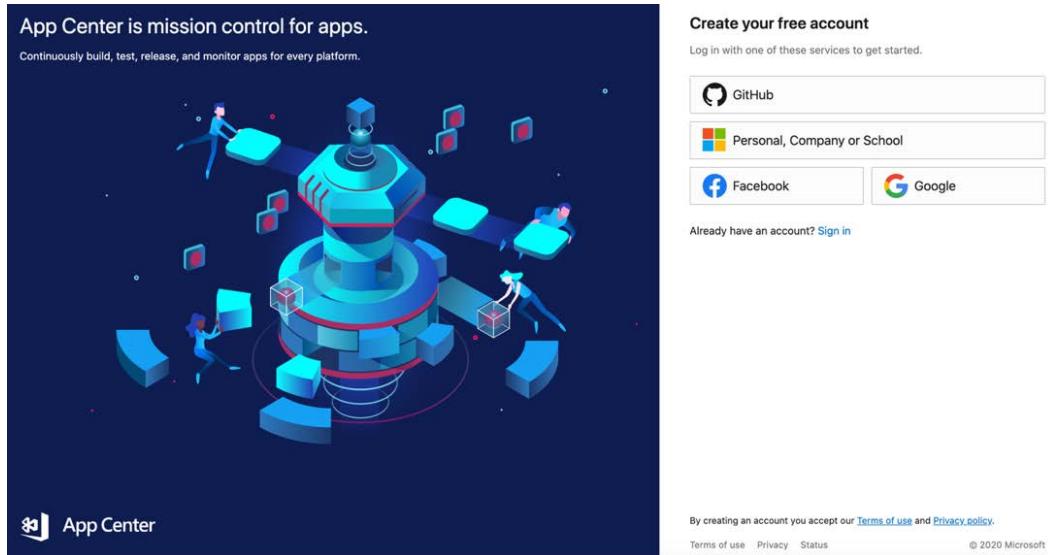


Figure 13.2 – App Center sign-in options

**Note**

For organizations with **Azure Active Directory (AAD)** linked to their App Center organization, users can sign in with their AAD email address and skip the sign-up process. For more information regarding business accounts in App Center, refer to the following link: <https://docs.microsoft.com/en-us/appcenter/general/account#azure-active-directory-backed-work-email-addresses>.

4. After signing in with the account of your choice, you will be prompted to create an App Center username. Enter the name you would like to use and click **Choose username** to continue.
5. This will complete the sign-up process. Then, you will be taken to the App Center dashboard and will be greeted by a welcome message:

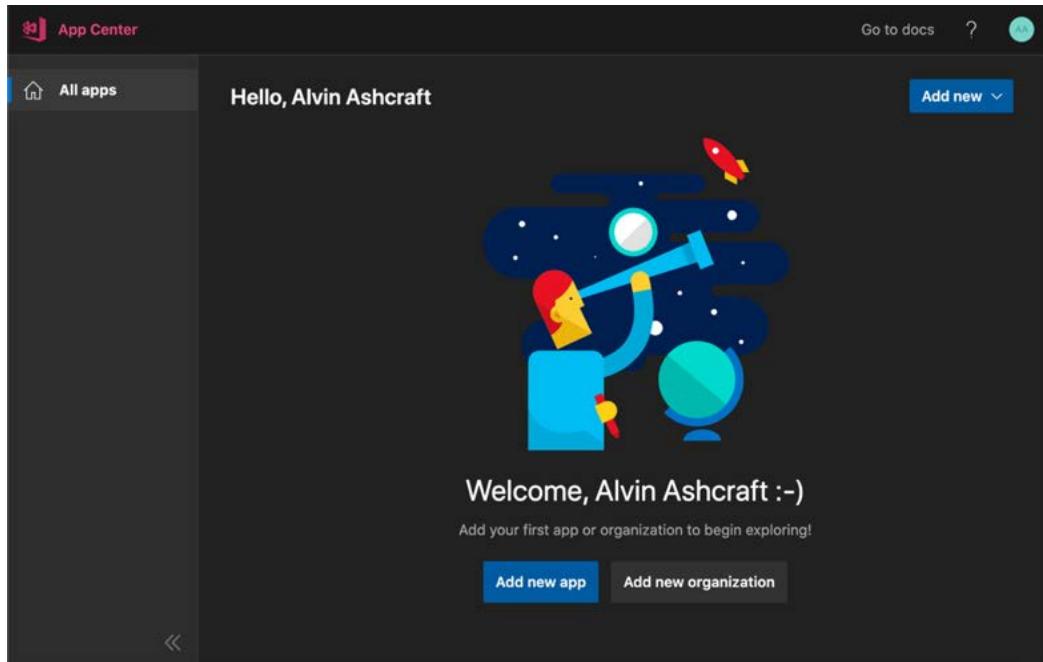


Figure 13.3 – The App Center dashboard

From here, you can add a new *application* or *organization*. Adding an organization is recommended if you plan to collaborate on your applications with others. An organization allows you to invite others and manage their access. Within an organization, you have the following roles available:

- **Admin:** Has full access to the entire organization, including managing apps, users, and distribution groups and teams
- **Collaborator:** Can create apps, see apps in the organization to which they have been added, and manage distribution groups and teams
- **Member:** Can create apps and see apps in the organization to which they have been added

We are not going to use an organization for our application in this chapter. To read more about creating and managing organizations in App Center, you can read this Microsoft Docs article: <https://docs.microsoft.com/en-us/appcenter/dashboard/creating-and-managing-organizations>.

To manage your user settings, you can click your avatar in the upper-right corner of the dashboard and select **Account Settings**. On the **Settings** page, you can manage the following settings:

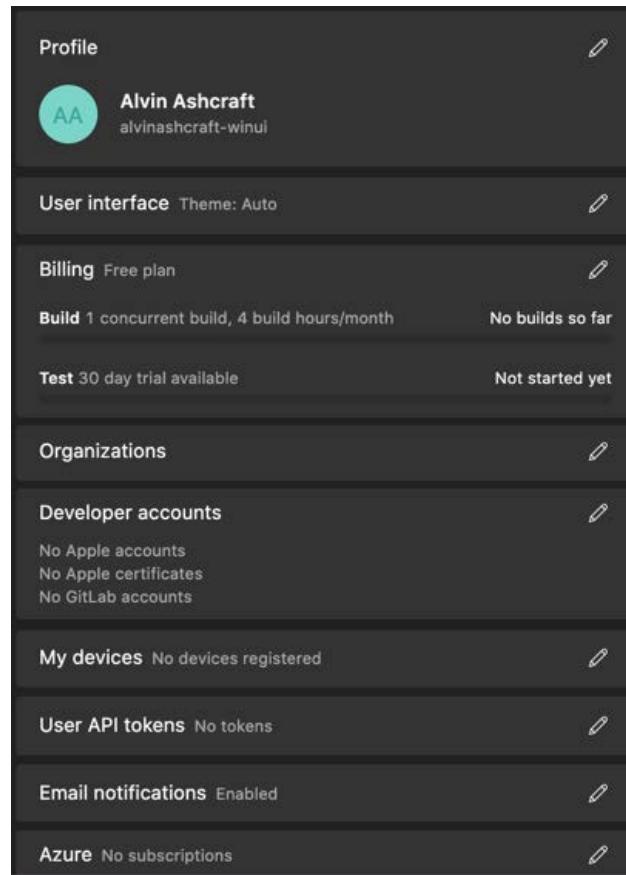


Figure 13.4 – The user settings page in App Center

Here is the list of settings:

- **Profile:** Update your avatar with **Gravatar** or change your name, username, or email address.
- **User interface:** Select a light, dark, or auto theme for the App Center site.
- **Billing:** Manage your current plan and view current usage statistics.
- **Organizations:** Manage your App Center organizations.
- **Developer accounts:** Link Apple accounts, Apple certificates, and GitLab accounts here.

- **My devices:** Manage linked devices.
- **User API tokens:** Manage API tokens for services used by your applications.
- **Email notifications:** Manage your email notifications.
- **Azure:** Link an Azure subscription to manage your payment options for premium App Center services. This also allows for role management via AAD.

Now that we have taken a brief tour of the App Center dashboard and settings, let's return to the dashboard by clicking **All apps** in the left navigation pane. From here, we will create our first app to manage in App Center.

## Creating your first App Center application

In this section, we are going to create an App Center app to manage the **MyMediaCollection** project that we created in the first half of this book. Once it is configured in App Center, we will configure a CI build, make builds available to beta testers, and create deployments for end users:

1. Start by clicking **Add new | Add new app** in the upper-right portion of the main dashboard panel.
2. On the **Add New App** page, enter the name of the app, select an icon if you like, enter **Beta** for **Release Type**, and select **Windows** for **OS** and **UWP** for **Platform**. The project is a WinUI in the UWP project type, so we will want to indicate UWP for **Platform**:

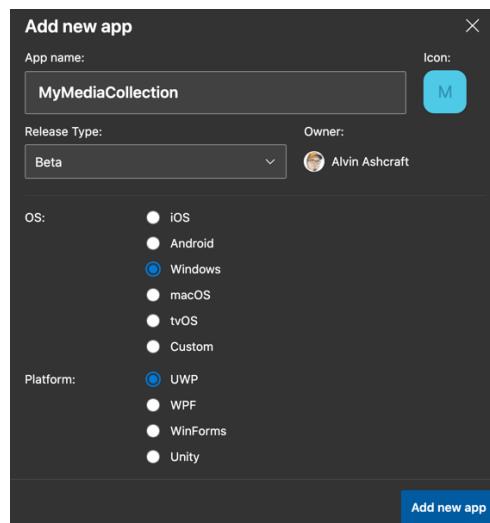


Figure 13.5 – Add a new app to App Center

3. After the app has been created, you will be taken to the **Overview** page for the **MyMediaCollection** app. On this page, you will be given some steps to follow to add App Center analytics crash reporting code to the project. We will hold on to this information until the *Application monitoring and analytics* section of this chapter.

Let's review the **Settings** page for the app before we move ahead with creating a build configuration in the next section. Click **Settings** and review the settings you can configure for each application in App Center:

The screenshot shows the 'App details' page for the 'MyMediaCollection' app. The top section displays the app's name, developer, and basic metadata: OS Windows, Platform UWP, and Release Type Beta. Below this, the page is organized into several sections, each with an edit icon (pencil icon):

- People:** 2 collaborators (with icons for a person and a user)
- Data management:** 90 days
- Services:** None
- Webhooks:** None
- Export:** None
- Email notifications:** New version distributed and adding new devices to a release failed
- App API tokens:** No tokens

Figure 13.6 – The app settings page in App Center

- **App details:** Update the application details that we just entered when creating the app.
- **People:** Manage app collaborators.
- **Data management:** Configure how long App Center should retain diagnostic information about your application (the default is 90 days).
- **Services:** Link bug trackers in Azure DevOps, GitHub, or Jira.
- **Webhooks:** Configure webhooks to send data from App Center to outside applications such as Microsoft Teams or Slack.
- **Export:** Configure the continuous export of data to Azure for analysis.
- **Email notifications:** Configure application-level email notifications.
- **App API tokens:** Configure any API tokens specific to this application.

That was a quick tour of our new application in App Center. In the next section, we will add our project to GitHub and set up a build in App Center.

## Setting up builds in App Center

It's time to push the **MyMediaCollection** app to a repository in GitHub and link it to a new build in App Center. In the previous chapter, we added a project to GitHub with GitHub Desktop. In this section, we will use the Visual Studio 2019 GitHub extension to push our code to the cloud:

1. Open the **MyMediaCollection** project in Visual Studio. If you have your own copy of the project handy, you can use that. Otherwise, get the code from the **Start** folder of this chapter's repository, using the link in the *Technical requirements* section of this chapter.
2. Go to **Extensions | Manage Extensions** and search for GitHub. Select **GitHub Extension for Visual Studio** and install it. You will need to restart Visual Studio to complete the installation.

3. When Visual Studio restarts, go to the **Git Changes** window and click the **Create Git Repository** button. If you don't see the Git Changes window, you can open it from **View | Git Changes**:

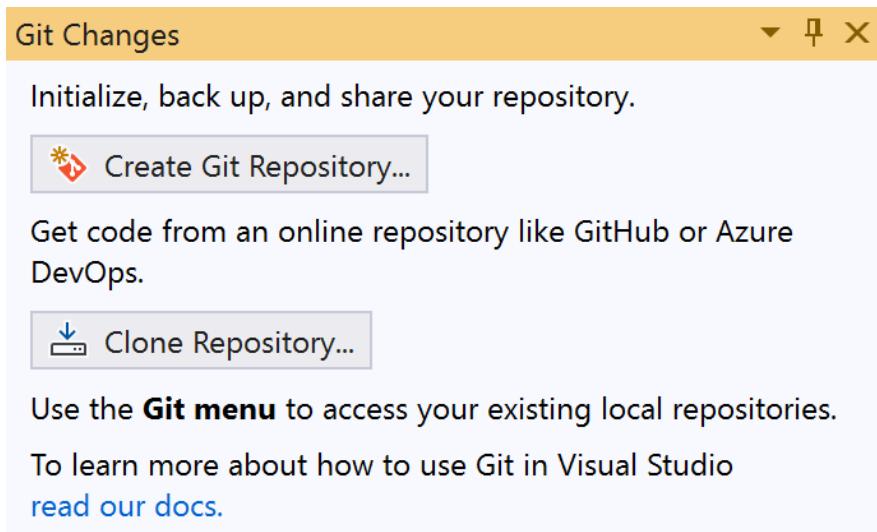


Figure 13.7 – Connecting to a GitHub repository

4. On the **Connect to GitHub** window that appears, click **Sign in with your browser** or **Sign up** if you don't already have a GitHub account. Follow the steps in your browser to sign in to your GitHub account and return to Visual Studio when prompted.
5. In **Solution Explorer**, right-click the **MyMediaCollection** solution and click **Create Git Repository**.
6. On the **Create a Git Repository** window, select **GitHub** under the **Push to a new remote** section, enter a repository name and description, and then click **Create and Push**. You can make the repository private if you so wish. This is selected by default:

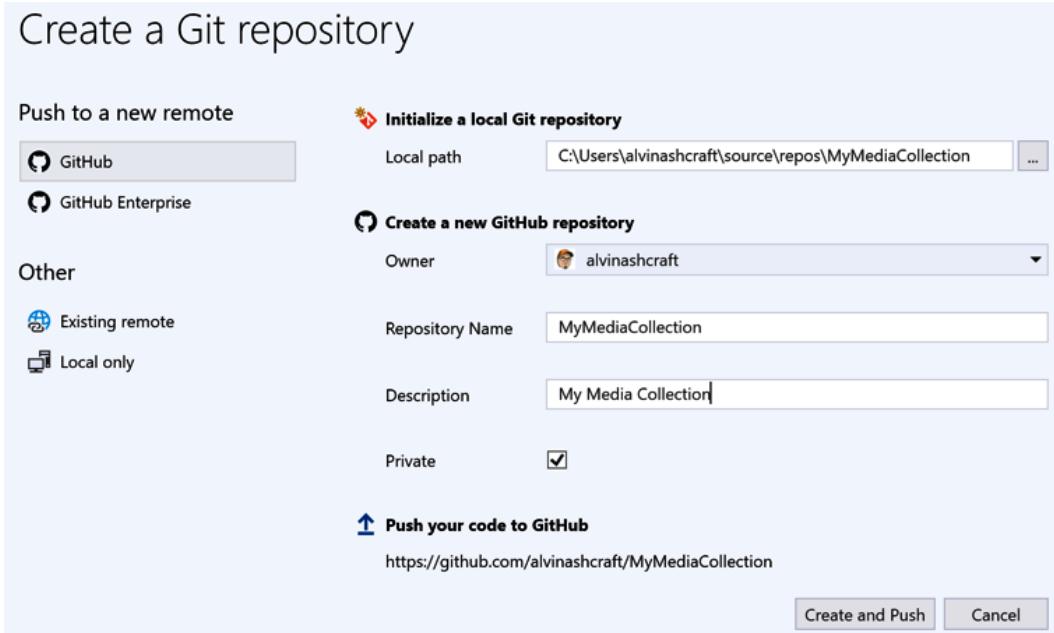


Figure 13.8 – Creating a new GitHub repository in Visual Studio

The repository tools will now appear in **Team Explorer**. Git-specific actions, such as viewing pending changes, performing commits, and pushing commits, can be accomplished in the **Git Changes** window.

The code for **MyMediaCollection** is now available on your GitHub repository. Whether you chose to make the repository public or private, you will be able to integrate it with App Center. In the next section, we will integrate the repository and create a build.

## Integrating App Center with a GitHub repository

Now that we have the **MyMediaCollection** project committed to GitHub, we're ready to add a CI build in App Center. Having a CI build is important for every project, especially when multiple developers are collaborating on the project. It is important to get immediate feedback if there are any compilation issues or unit test failures:

1. Start by opening App Center, select the **MyMediaCollection** app, and then select the **Build** item in the left navigation menu.
2. Under **Select a service**, choose **GitHub**.

3. You will be prompted to connect App Center to GitHub. Read the web page carefully to understand the access you are granting and click **Authorize VSAppCenter**:

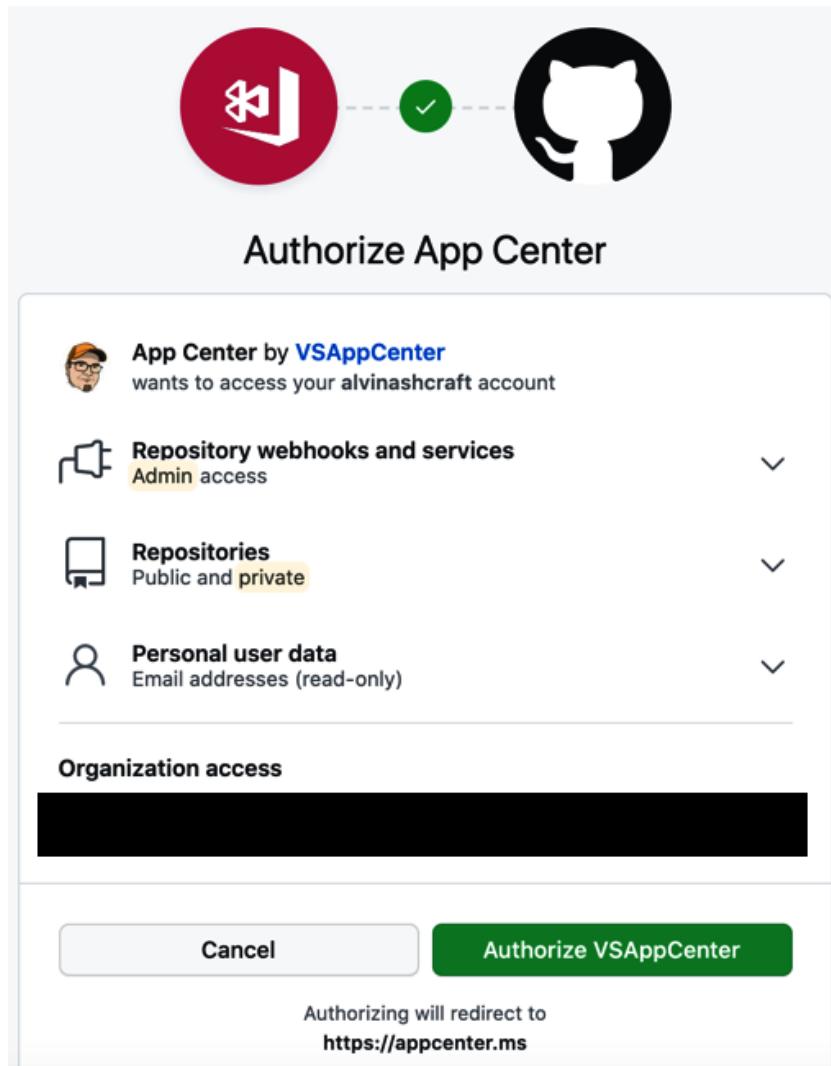


Figure 13.9 – Connecting App Center to GitHub

4. Once you are returned to App Center, select the **MyMediaCollection** repository.
5. Select the branch you want to build—**master** or **main**—for your primary branch, and click the **Configure build** button. The **Build configuration** screen will appear as follows:

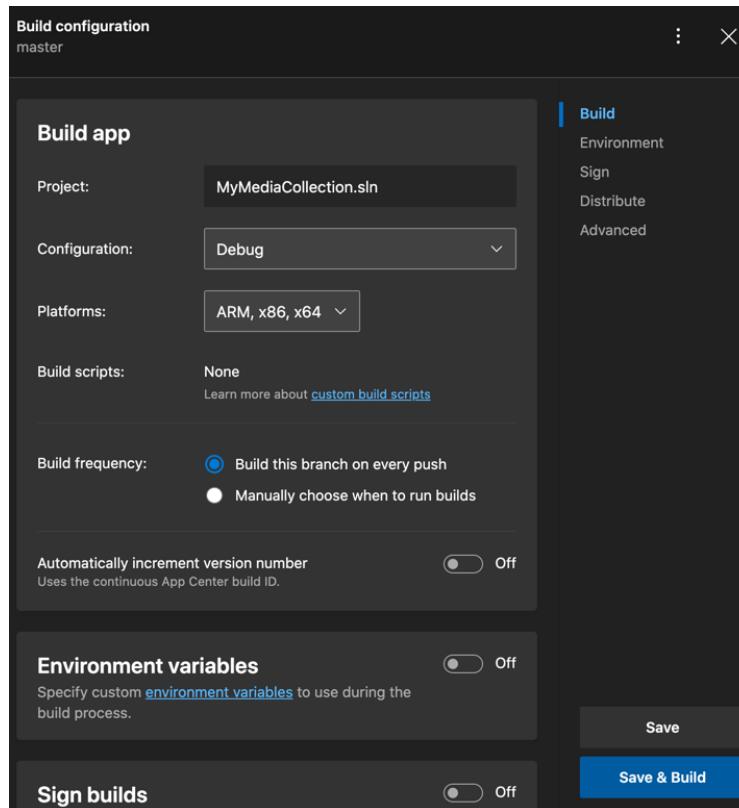


Figure 13.10 – Configuring the MyMediaCollection build

6. Change the **Configuration** field to **Release** and remove **ARM** from the selected platforms because this WinUI project with its current dependencies does not compile successfully with the **ARM** configuration. The **Build this branch on every push** selection for **Build frequency** will set this up as a CI build, enabling us to create a new build with each push to the branch. Keep the rest of the default settings and click **Save & Build**.

If the build fails due to issues with the unit test project, you can update the solution to only build the **MyMediaCollection** project in the **Release** configuration.

**Note**

It is not a good practice to exclude a test project from the build, as you want your CI build to run the unit tests. However, this sample project using a preview build of WinUI is not able to reference **MyMediaCollection** via a project reference without encountering compilation errors. It is likely that the final release of WinUI 3 will not have the same limitations.

Right-click the solution to change this in **Configuration Manager**. Uncheck **Build** and **Deploy** for **MyMediaCollection.Tests** in the **Release** configuration. *Commit* and *push* your changes after making any changes to kick off another build in App Center.

7. When you have a successful build, you will see a green checkmark and options to distribute or download the build output:

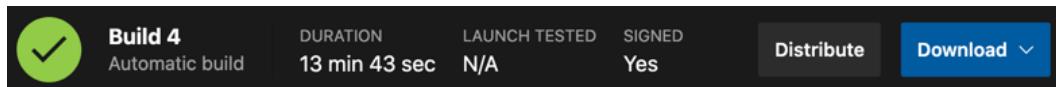


Figure 13.11 – Viewing a successful build in App Center

Now that we have a successful build, let's look at some deployment options.

## Deploying your application with App Center

The next step in this DevOps process is to create a *release* that can be installed by testers or end users. A release could be some type of test release (alpha, beta, release candidate), or it might be a stable release for all users.

Access to each release is managed by creating **Distribution Groups** in App Center. These groups are granted access to specific releases. For example, you might have an **Internal** distribution group for testing early alpha releases, a **Preview** distribution group of select customers or other outside users to test beta releases, and an **All Customers** group to receive only the stable application releases.

That's right, you can distribute any kind of release through App Center. If you don't intend to make your application available to the public through an app store like the Microsoft Store, you can manage these releases solely through App Center.

Let's now review the process of creating a release for our application.

## Creating early releases of your application or beta testers

We are going to use the distribute feature in App Center to create a beta release of **MyMediaCollection**:

1. Start by clicking **Distribute** in the left navigation panel of the App Center portal. This will take you to the **Releases** page.
2. We don't have any releases yet, so you will only see a **New release** button. Click that to continue.

3. On the **New release** page that appears, there is a button to upload a build. This allows you to upload an existing .appx, .msix, .msi, or other types of packages for distribution (refer to the following screenshot for a full list of package types supported):

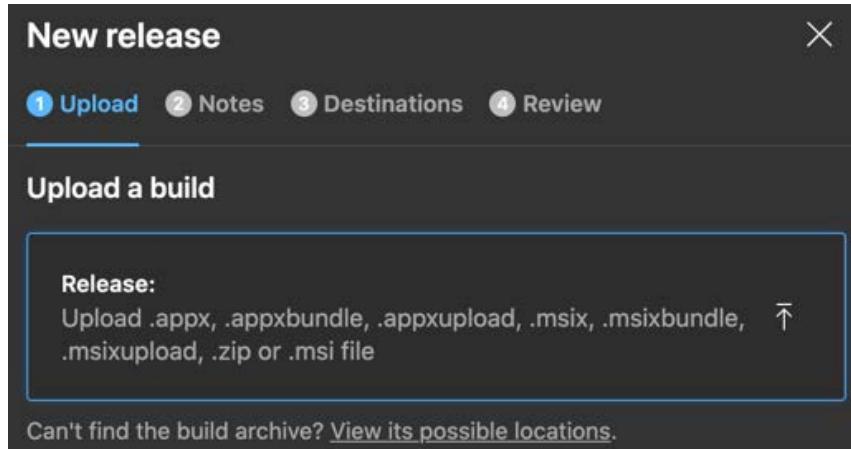


Figure 13.12 – Options for uploading a package for distribution

4. We already have a build available in App Server to distribute. To select the output from the build, click the **Distribute an Existing Build** link at the bottom of the page.
5. The page will now display the branches with builds configured. Select the branch for your application's build and click **Next**:

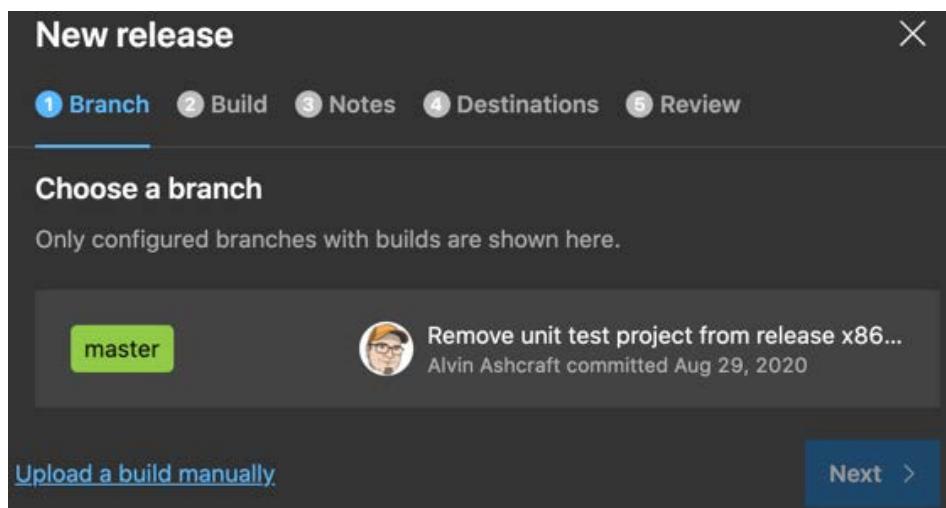


Figure 13.13 – Selecting a branch for distribution

6. The **Build** step will display a list of the five most recent successful signed builds. Select the one you want to use and click **Next**.
7. The **Notes** step allows you to enter any release notes for the users that will be receiving this release. If there are known bugs, fixes, or new features to test, you can enter that information here. By default, the notes you entered with the most recent GitHub commit will be displayed in this field. You can leave these or enter your own and click **Next**.
8. The **Destinations** step is where you assign access to the release. We haven't created any release groups, so we will select the **Collaborators** user group and click **Next**. The **Collaborators** group is the default group that is created, and it includes all users with access to the application in App Center. This will give your own user account access to the release.
9. On the final **Review** step, you will see some summary information regarding the release and two additional options. You can flag this release as a mandatory update for users in the assigned group(s). You use the App Center SDK to add code to your application to check for any mandatory updates and prevent users from using the application until they take the update. There is also a **Do not notify testers** option. If this is a very minor release, you may choose to have App Center not send out notifications when the release is available. Click **Distribute** when you are finished:

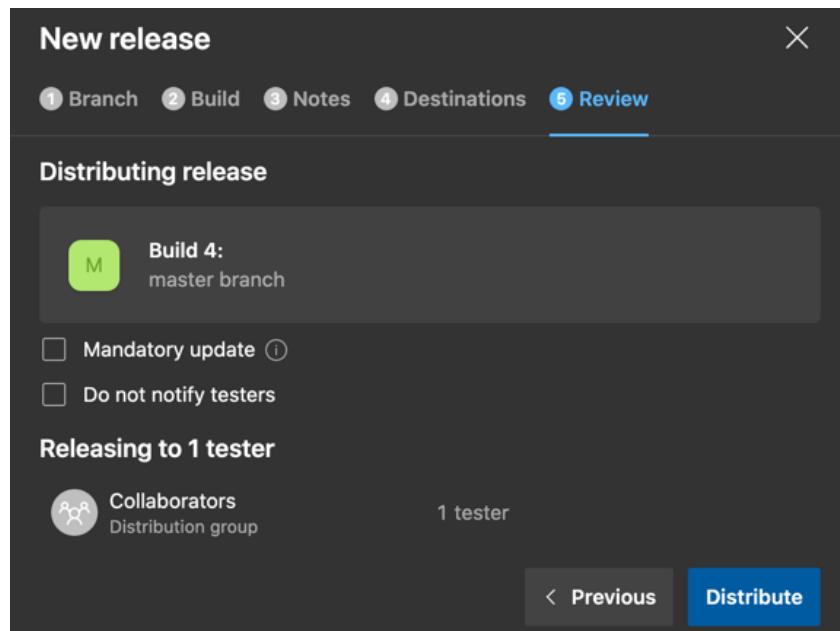
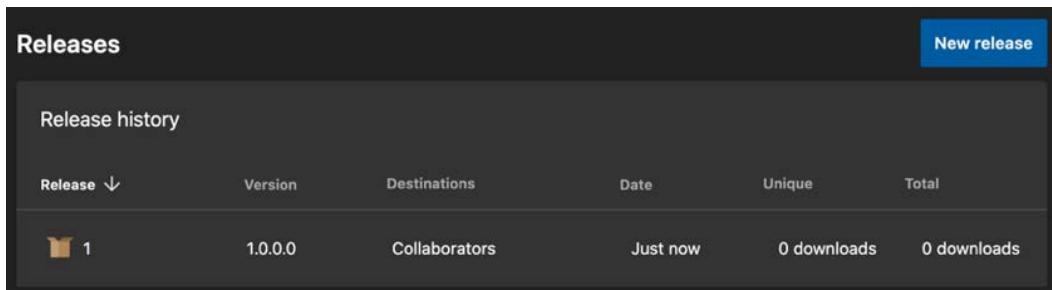


Figure 13.14 – Review and distribute a release in App Center

The **Releases** tab will now show **Release history**, with the release we just created as release 1:



The screenshot shows the 'Releases' tab in the App Center interface. At the top right is a blue 'New release' button. Below it is a table titled 'Release history' with the following data:

Release ↓	Version	Destinations	Date	Unique	Total
1	1.0.0.0	Collaborators	Just now	0 downloads	0 downloads

Figure 13.15 – Listing the available releases for your application in App Center

If you click on the release, you will be taken to its details page where you can download it or create another release. Users in the specified group(s) for the release will receive an email notification to download the new release from App Center.

It is also possible to automate the release and distribution process, generating a new release upon successful completion of a build. To read more about application distribution with App Center, check out this page on Microsoft Docs: <https://docs.microsoft.com/en-us/appcenter/distribution/>.

Now that we understand how to create and distribute releases, let's update our code to enable instrumentation so we can view some analytics in App Center.

## Application monitoring and analytics

The App Center SDK allows you to instrument your code to determine how the application is used, how frequently features are used, and where errors are occurring. If you are distributing an iOS or Android app, you can also use the SDK to require users to take new releases that are flagged as *mandatory*.

Let's start by referencing the SDK and instrumenting our code.

## Instrumenting your code

In this section, you will add some instrumentation code to the **MyMediaCollection** project to capture usage and crashes. When these changes are committed to GitHub, another App Center build will be triggered, and we will create a new mandatory release for users:

1. In Visual Studio, open **NuGet Package Manager** for **MyMediaCollection** and search for **AppCenter**. Install the **Microsoft.AppCenter.Analytics** and **Microsoft.AppCenter.Crashes** packages in the project.
2. Open **App.xaml.cs** and add three using statements:

```
using Microsoft.AppCenter;  
using Microsoft.AppCenter.Analytics;  
using Microsoft.AppCenter.Crashes;
```

3. In the constructor in that same file, add a line of code to start AppCenter's monitoring. You can get the value for **Your App Key** from the **Overview** page of **MyMediaCollection** in the App Center dashboard. Following **Your App Key**, we will pass the types of App Center services we will be opting into. The possible types provided by Microsoft are **Analytics**, **Crashes**, **Auth**, **Push**, and **Distribute**. We will be using the **Analytics** and **Crashes** types provided by the two NuGet packages that were imported into our project:

```
AppCenter.Start("{Your App Key}", typeof(Analytics),  
    typeof(Crashes));
```

4. Now, build and run **MyMediaCollection**. If you open App Center and navigate to the **Analytics** page for the app, you should see an active user and session:

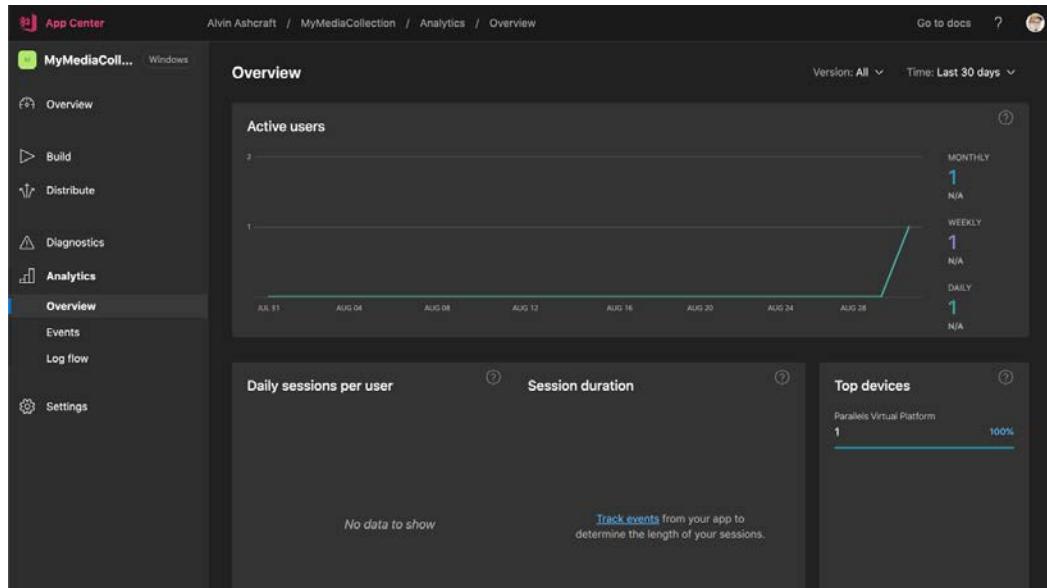


Figure 13.16 – Viewing activity on the App Center Analytics dashboard

Let's add some code to track some of the user activity within the application. Open **MainViewModel.cs** and add this code to the **SelectedMediaItem** property setter:

```
public MediaItem SelectedMediaItem
{
    get => selectedMediaItem;
    set
    {
        if (SetProperty(ref selectedMediaItem, value))
        {
            ((RelayCommand)DeleteCommand).
            RaiseCanExecuteChanged();
            Analytics.TrackEvent("Item selected", new
                Dictionary<string, string> {
                    { "MediaType",
                        selectedMediaItem.MediaType.
                        ToString() },
                    { "ItemName", selectedMediaItem.Name } });
        }
    }
}
```

```

    }
}
```

5. Don't forget to add a using statement to the MainViewModel class:

```
using Microsoft.AppCenter.Analytics;
```

6. Add one more line to DeleteItemAsync to track when users delete an item from their collection:

```

private async Task DeleteItemAsync()
{
    await _dataService.DeleteItemAsync(SelectedMediaItem);
    Items.Remove(SelectedMediaItem);
    allItems.Remove(SelectedMediaItem);
    Analytics.TrackEvent("Delete item clicked", new
        Dictionary<string, string> {
            { "MediaType",
                SelectedMediaItem.MediaType.ToString()
            },
            { "ItemName", SelectedMediaItem.Name }
        });
}
```

7. Now run the application, select and edit a few items, and try deleting an item. When you return to App Center to review **Analytics**, you should see that the activity was tracked and recorded in real time on the **Log flow** page:

Log flow		
Id	Description	Time
• 527c7d56	STARTSERVICE	15:27:55
• 527c7d56	EVENT - Delete item clicked - {"MediaType":"Video","ItemName":"Shark Hunt 3"}	15:30:23
• 527c7d56	STARTSERVICE	15:37:33
• 527c7d56	EVENT - Item selected - {"MediaType":"Book","ItemName":"Doctor Cheese"}	15:37:49
• 527c7d56	EVENT - Item selected - {"MediaType":"Book","ItemName":"Doctor Cheese"}	15:37:49

Figure 13.17 – Tracking user activity in App Center

**Note**

Make absolutely certain that no personal identifiable data is included with analytics data collected by your application. Information such as account IDs, passwords, or anything that can identify individuals should be scrubbed from any data sent to App Center.

8. View the **Events** tab to see a summary of the activity recorded:

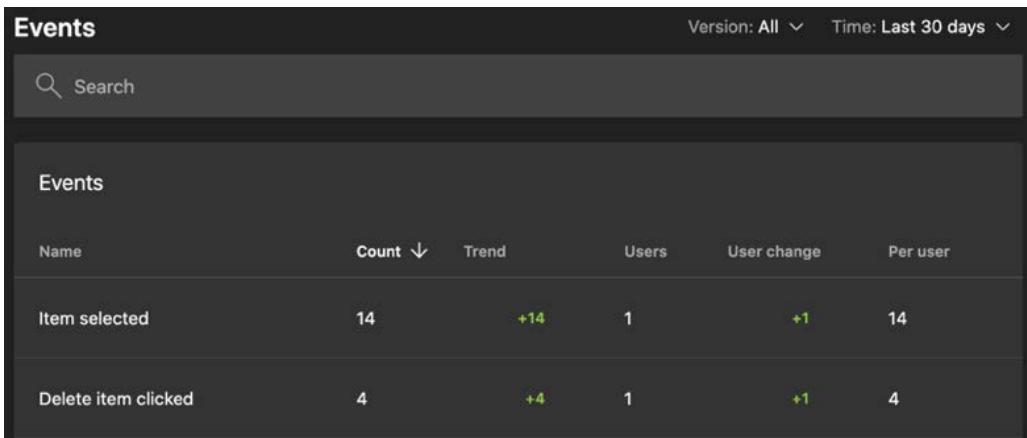


Figure 13.18 – Viewing a list of tracking events and their trends

9. Finish by committing your changes to GitHub. This will trigger another build in App Center.

If you want to share the app with others to track analytics with multiple users, you can create another release from the new build, add users to the distribution group, and have your friends install this package from App Center.

Let's finish up with some information regarding crash reports in App Center.

## Gathering and analyzing App Center crash reports

The last topic we will cover in this chapter is monitoring application crashes with App Center. Because we added `Crashes` to the call to `AppCenter.Start` in `App.xaml.cs`, any crashes will automatically be reported to App Center's analytics. But how can we simulate a crash while we're developing an application?

To simulate a crash, add the following line of code to any application sometime after the `AppCenter.Start` call has been completed. Let's add it to the beginning of the `SelectedMediaItem` property setter in `MainViewModel`. Clicking on any media item in the collection will generate a crash for us. Use the `#if DEBUG` preprocessor directive to ensure that this will not happen in your application in production:

```
#if DEBUG
    Crashes.GenerateTestCrash();
#endif
```

Don't forget to add a `using` statement to the class as well:

```
using Microsoft.AppCenter.Crashes;
```

**Note**

`GenerateTestCrash` will only execute in *debug* builds, so you don't have to be concerned about accidentally crashing users running *release* builds of your application. To be absolutely certain of this, we have used the `#if DEBUG` directive in our code.

You can track exceptions with App Center by using the `Crashes.TrackError` method and passing in the current `Exception` object. Let's add it to a `catch` statement inside the `UpdateMediaItemAsync` method in `SQLiteDataService`. Don't forget to add your `using` statement:

```
private async Task UpdateMediaItemAsync(SqliteConnection
    db, MediaItem item)
{
    try
    {
        await db.QueryAsync(
            @"UPDATE MediaItems
            SET Name = @Name,
                ItemType = @MediaType,
                MediumId = @MediumId,
                LocationType = @Location
            WHERE Id = @Id;", item);
    }
    catch (Exception e)
```

```
{
    Crashes.TrackError(e);
}
}
```

Now, any exceptions caught in this data access method will be tracked by App Center. Note that in your own applications, you will also want to present a user-friendly error message after sending it off to App Center. I will leave it to you as an exercise to go back and add the same error handling and tracking to the rest of the data access code in `SqlLiteDataService`. Don't forget to commit and push all your changes to GitHub when you are finished instrumenting your code.

You can view all crashes and errors on the **Diagnostics | Issues** page in App Center:

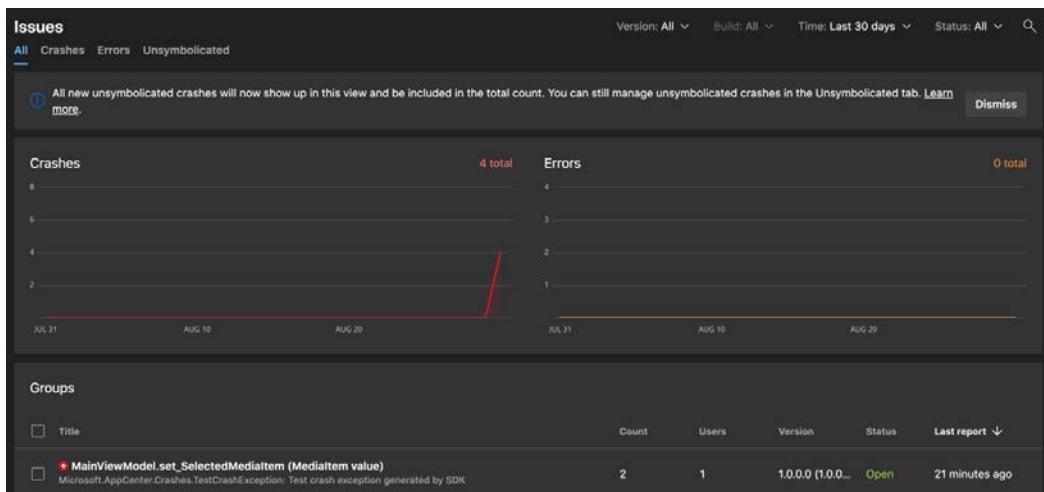


Figure 13.19 – Viewing a summary of issues in your application in App Center

If you want to learn more about the App Center crash reporting capabilities for UWP and WinUI, you can visit this Microsoft Docs page: <https://docs.microsoft.com/en-us/appcenter/sdk/crashes/uwp>.

That wraps up our section on App Center crashes. Let's now review what we have learned in this chapter.

## Summary

In this chapter, we learned how to build, deploy, instrument, and track analytics and crashes for our projects with App Center. We toured the App Center dashboard and explored its features. We saw how to commit changes to GitHub and configure App Center to automatically compile new builds and release them to testers and end users. Finally, we saw how to add some code to track feature use, exceptions, and crashes with the App Center SDK packages. These techniques will help any size of project deliver higher quality code and find bugs earlier in the application's life cycle. Whether you are an individual developer building your own apps or part of a large company, App Center has tools that can help improve your products.

In the next chapter, we will focus on other ways of distributing your WinUI applications.

## Questions

1. Do you need to have a Microsoft account to sign up for App Center?
2. What is the maximum number of applications that a free account can add to App Center?
3. What limits are placed on application builds for free accounts in App Center?
4. How can you create different groups of users to assign to specific releases in App Center?
5. Which bug tracking software can integrate with App Center?
6. Where can you view real-time analytics events flowing into App Center?
7. How can you track exceptions through App Center?

# 14

# Packaging and Deploying WinUI Applications

We have seen how Windows developers can take advantage of **Visual Studio App Center** to package and deploy applications. WinUI developers choosing not to use App Center have several other options for packaging and deployment. Developers can create an account on the **Microsoft Store** and upload a packaged app to be released for public consumption through **Microsoft Partner Center**. App packages can also be created to be distributed by organizations through **Microsoft Endpoint Manager** and **Microsoft Intune** or sideloaded by individuals on Windows PCs.

In this chapter, we will cover the following topics:

- Discovering application packaging and **MSIX** basics
- Getting started with application packaging in Visual Studio
- Deploying applications with Windows Package Manager
- Distributing applications with the Microsoft Store
- Sideloaded WinUI applications with MSIX

By the end of this chapter, you will understand the methods available to package and distribute your WinUI applications and how to use each of them.

## Technical requirements

To follow along with the examples in this chapter, the following software is required:

- Windows 10 Version 1803 (build 17134) or newer.
- Visual Studio 2019 Version 16.9 or newer with the following workloads: .NET Desktop Development; Universal Windows Platform Development.
- To create desktop WinUI projects, you must also install the latest .NET 5 **software development kit (SDK)**.

The source code for this chapter is available on GitHub at this **Uniform Resource Locator (URL)**:

<https://github.com/PacktPublishing/-Learn-WinUI-3.0/tree/master/Chapter14>

## Discovering application packaging and MSIX basics

For most of this book, we have been building and running our applications locally. We have touched on some packaging concepts in *Chapter 8, Building WinUI Apps with .NET 5*, and deployment with App Center in *Chapter 13, Building, Releasing, and Monitoring Applications with Visual Studio App Center*. Now, it's time to build on those concepts and gain a deeper understanding of packaging and deploying WinUI applications.

Why package your application? Well, an application package is currently required for WinUI applications to be installed in Windows. This requirement will likely change as Project Reunion and WinUI applications evolve. Today, when you run a WinUI project in Visual Studio, the **integrated development environment (IDE)** creates a package and deploys it locally. Packaging serves several other important purposes, outlined as follows:

- **Providing a clean uninstall:** A packaging system ensures that any files installed or updated with an application are removed or restored to their previous state when the application is uninstalled.
- **Bundling dependencies:** The application package will bundle and deliver all your application's dependencies, optimizing disk space by sharing files across installed applications when possible.

- **Facilitating updates:** Differential updates are optimized to only deliver files that require updating, based on the manifest of the original package and the updated one.
- **Declaring capabilities:** By declaring your application's capabilities in the manifest, users know what types of access your app requires before they choose to install it.
- **Verifying integrity and authenticity:** In order to install a Windows 10 application, the application package must be digitally signed with a valid certificate from a trusted signing authority.

If you are going to distribute your WinUI applications, packaging is required, and the packaging format you will use is **MSIX**. What is MSIX? Let's find out.

## What is MSIX?

MSIX is a new standard proposed by Microsoft for packaging applications. MSIX is not only for Windows. The **MSIX SDK** (<https://docs.microsoft.com/en-us/windows/msix/msix-sdk/sdk-overview>) is an open source project that can be used to create application packages for any platform. You can use the SDK for Windows, Linux, macOS, iOS, and Android. We will stay focused on delivering WinUI applications to Windows users in this chapter, but you can learn more about the MSIX SDK on its GitHub repository at <https://github.com/Microsoft/msix-packaging>.

On Windows, the following platforms currently support MSIX format:

- Windows 10 Version 1709 and later
- Windows Server 2019 **Long-Term Servicing Channel (LTSC)** and later
- Windows Enterprise 2019 LTSC and later

Earlier versions of Windows 10 require **APPX** packages, which were the predecessor to MSIX packages. Microsoft introduced MSIX packages in 2018 as an evolution of APPX, intending to fill the needs of APPX as well as legacy **Windows Installer (MSI)** packages. MSIX is an open standard and, as such, it can be used to distribute applications to any platform. MSI has been the standard for packaging and installing Windows desktop applications since 1999. Installing applications with MSI packages is supported in Windows 95 and later operating systems.

With the new MSIX packaging standard, **Universal Windows Platform (UWP)** applications delivered to Windows users run inside a lightweight app container. The Windows App Container provides a sandbox for the execution of an application, restricting access to the registry, filesystem, and other system capabilities, such as the camera. Any capabilities that the application needs to access must be specified in the manifest file. Applications installed with MSIX have *read access* to the Windows registry by default. In this configuration, any data written to the virtualized registry will be completely removed if the application is uninstalled or reset. The same is true of data written to the virtual filesystem.

As documented on Microsoft Docs (<https://docs.microsoft.com/en-us/windows/msix/overview#inside-an-msix-package>), an MSIX package's contents are grouped into *application files* and *footprint files*, as illustrated in the following diagram:

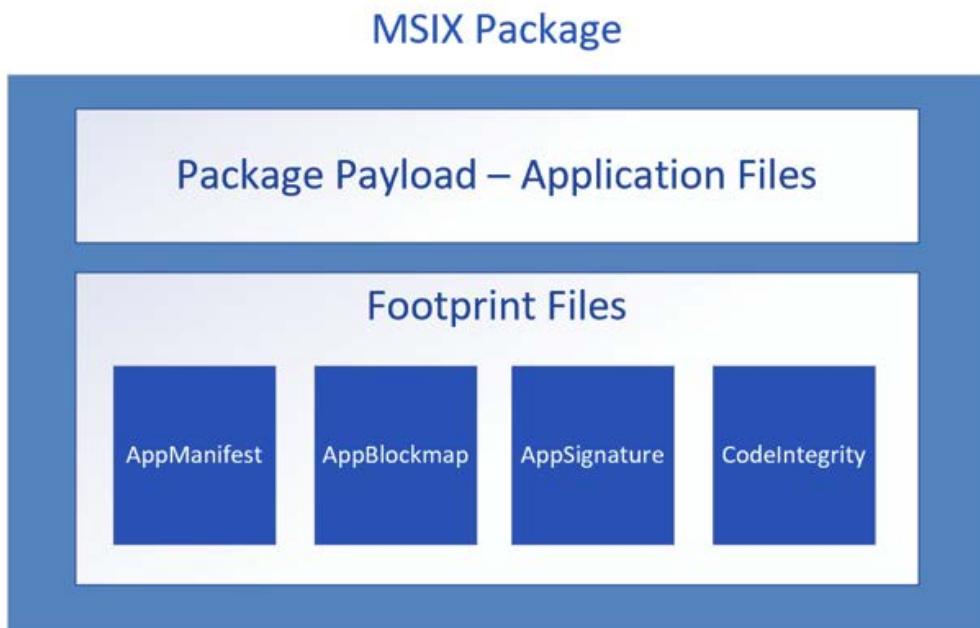


Figure 14.1 – The contents of an MSIX package

The application files are the payload of code files and other assets delivered to the user. The footprint files are the metadata and other resources that the package needs to ensure the application files are delivered as intended. This metadata includes the following:

- **AppManifest:** The manifest (`AppxManifest.xml`) includes information about the application's identity, dependencies, capabilities, extensibility points, and its visual elements. This is generated from the `Package.appxmanifest` file in your WinUI in UWP project or the packaging project in your WinUI in Desktop solution.
- **AppBlockmap:** The `AppxBlockMap.xml` file includes an indexed and cryptographically hashed list of the files in the package, digitally signed to ensure its integrity when the package is signed.
- **AppSignature:** The `AppxSignature.p7x` file in the package is generated when the package is signed. This allows the operating system to validate the signature during installation.
- **CodeIntegrity:** Code integrity is ensured by verifying information about the package in `AppxManifest.xml`, `AppxBlockMap.xml`, and `AppxSignature.p7x`.

The application files in the package will be installed to `C:\Program Files\WindowsApps\<package_name>` with the application executable found at `C:\Program Files\WindowsApps\<package_name>\<app_name>.exe`. Note that you can't directly execute this EXE file. Data created by the application after installation will be created under `C:\Users\<user_name>\AppData\Local\Packages\<package_name>`. All the application files, dependencies, and data will be removed during uninstallation.

Now that we have a little background and history of MSIX, let's review a few of the tools available to developers and IT pros.

## Reviewing MSIX tools and resources

Before we start using MSIX to package our own application, we are going to review a couple of other tools and resources available, as follows:

- **MSIX Toolkit:** The MSIX Toolkit is an open source collection of MSIX tools and scripts maintained by Microsoft on GitHub:  
<https://github.com/microsoft/MSIX-Toolkit>
- **MSIX Labs:** Microsoft maintains a set of hands-on tutorials for developers and IT pros interested in leveraging MSIX for packaging and distributing their applications:  
<https://github.com/Microsoft/msix-labs>
- **MSIX Packaging Tool:** The MSIX Packaging Tool is an application to repackage classic applications in MSIX format. Existing EXE, **Microsoft Software Installer (MSI)**, and **Application Virtualization (App-V)** install packages can be converted to MSIX packages with the tool via its interactive **user interface (UI)** or command-line tooling. It is available on the Microsoft Store at <https://www.microsoft.com/p/msix-packaging-tool/9n5lw3jbcxkf>.
- **MSIX videos:** Microsoft Docs has a series of introductory videos on MSIX packaging here: <https://docs.microsoft.com/en-us/windows/msix/resources#msix-videos>. This is a great way to get started on your MSIX journey.
- **MSIX community:** **Microsoft Tech Community** has a group of discussion spaces dedicated to MSIX packaging and deployment. Join the community and get involved here: <https://techcommunity.microsoft.com/t5/msix/ct-p/MSIX>.

These tools and resources will assist you in your journey while learning the ins and outs of WinUI application deployment. It is important to remember that MSIX is an area of continual investment by Microsoft. It is the go-forward strategy and recommendation for packaging all applications. WinUI developers don't need to have a deep understanding of MSIX. You will only need a basic knowledge of MSIX and the properties that are relevant to our applications. Now, let's get started and go hands-on, creating our MSIX package in Visual Studio.

# Getting started with application packaging in Visual Studio

In the previous chapter, we saw how App Center could automatically create and deploy application packages. Now, we will see how we can manually package our applications with Visual Studio. Visual Studio 2019 includes two WinUI project templates capable of creating MSIX deployment packages, outlined as follows:

- **Blank App (WinUI in UWP):** Creates a WinUI project with a package.appxmanifest file for generating an MSIX package
- **Blank App, Packaged (WinUI in Desktop):** Creates a solution with two projects—a desktop WinUI project and a packaging project that contains the package.appxmanifest file

We are going to work with the WinUI in UWP **MyMediaCollection** solution again. You can either use your own solution from previous chapters or download a copy from this chapter's GitHub repository at <https://github.com/PacktPublishing/-Learn-WinUI-3.0/tree/master/>. Let's see how to generate an MSIX package for the application, as follows:

1. Start by opening the solution in Visual Studio.
2. If you want to review the manifest data for the project, you can open the package.appxmanifest file and review the settings on each tab, as illustrated in the following screenshot:

Figure 14.2 – Reviewing the application manifest

3. Next, right-click the **MyMediaCollection** project in **Solution Explorer** and select **Publish | Create App Packages**. The **Create App Packages** window will appear, as illustrated in the following screenshot:

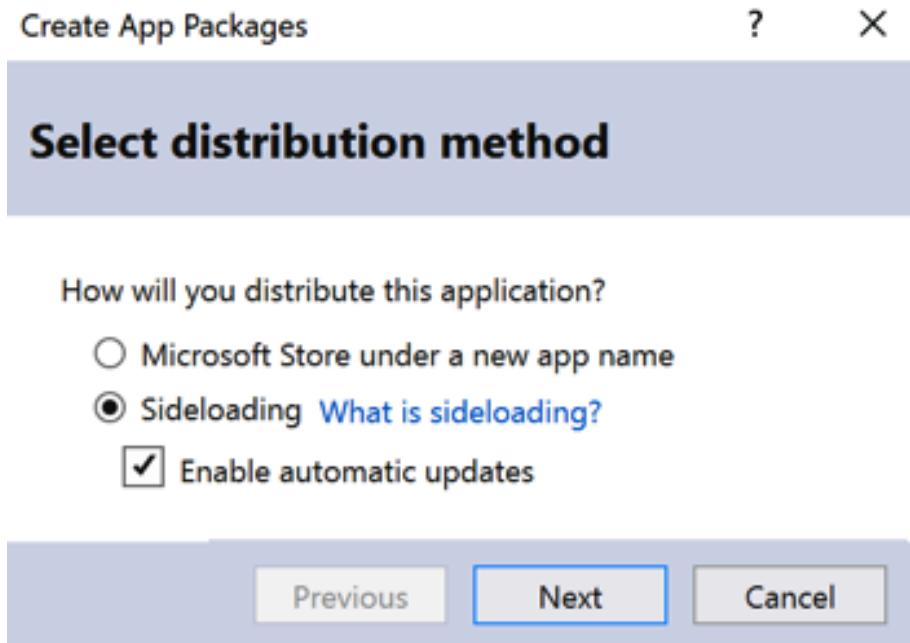


Figure 14.3 – The Create App Packages window

4. Select **Sideloading**, leave **Enable automatic updates** selected, and click **Next**. This is the MSIX equivalent of ClickOnce deployment (<https://docs.microsoft.com/en-us/visualstudio/deployment/clickonce-security-and-deployment>).
5. On the **Select signing method** page, remove the current certificate. You will now see options to add a certificate from **Azure Key Vault**, the **Store**, or a local file, as illustrated in the following screenshot:

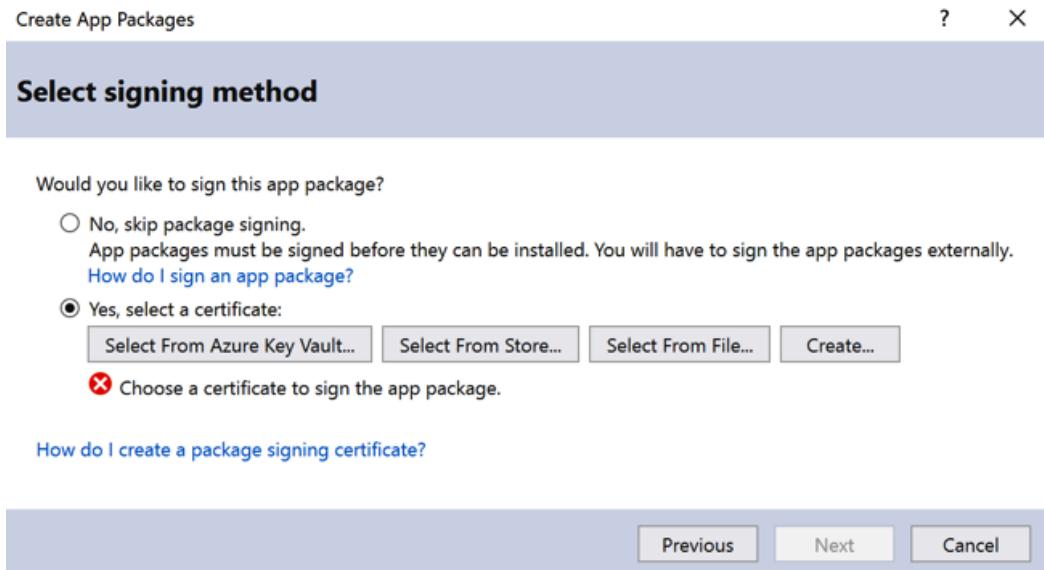


Figure 14.4 – Selecting a signing method for the package

6. We are going to create another self-signed certificate here. If you wanted to create a package with a certificate from a trusted authority, you would choose the **Import** button here to import the .pfx file. For this exercise, select **Create**.
7. Enter and confirm a secure password on the **Create a Self-Signed Certificate** dialog and click **OK**. You will be prompted to overwrite the existing certificate. Select **Yes** to continue.
8. When you return to the **Select signing method** page, click **Next**.
9. On the **Select and configure packages** page, leave all the defaults and click **Next**.

10. On the **Configure update settings** page, an **Installer location** path is required. For apps, you will be installing a workstation on an internal network, so you could enter the network path to the package here. For our test purposes, we are going to enter a local path, as illustrated in the following screenshot. For this to update correctly on other machines, the same local path to the installer would need to exist:

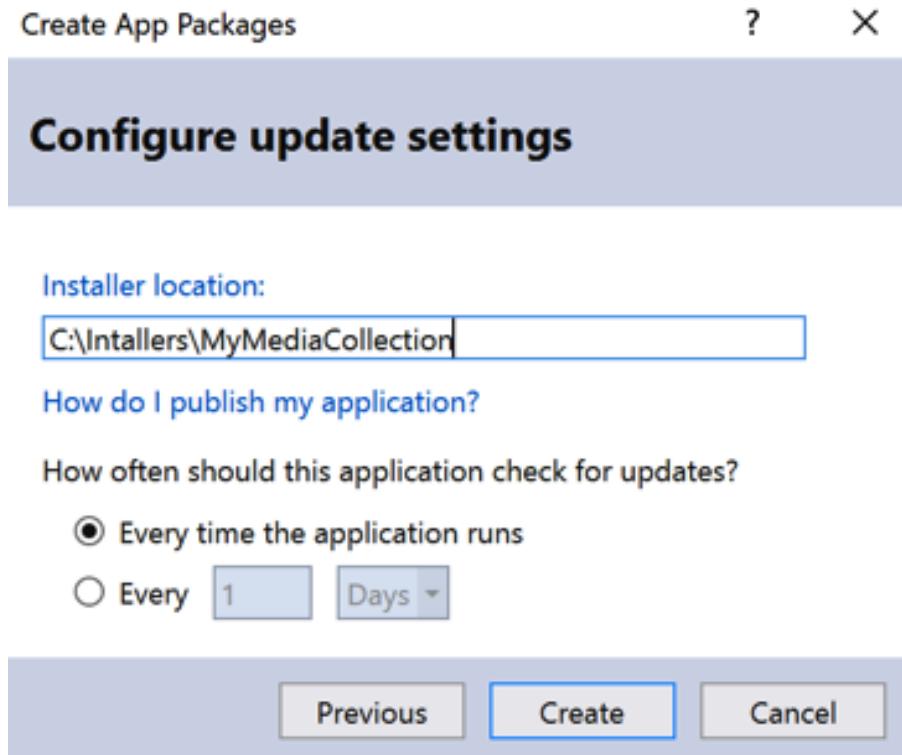


Figure 14.5 – Configuring update settings for the package

11. Click **Create**. The solution will build, and the package will be created. When it completes, click the **Copy and Close** button to copy the installer to your selected **Installer location**.
12. Open the `MyMediaCollection_1.0.0.0_Debug_Test` folder to see the generated MSIX bundle file, as illustrated in the following screenshot:

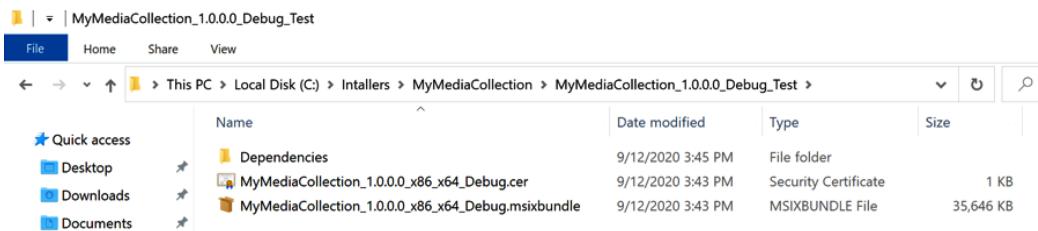


Figure 14.6 – The MSIX bundle created for MyMediaCollection

**Note**

An MSIX bundle is a bundle of MSIX packages that allows you to bundle multiple application architectures into a single distribution. This allows you to distribute them all together.

13. Double-click the bundle file to install it. If you try to install the bundle on the same machine where it has already been installed and run, you will be prompted to either re-install or launch the app, as illustrated in the following screenshot:

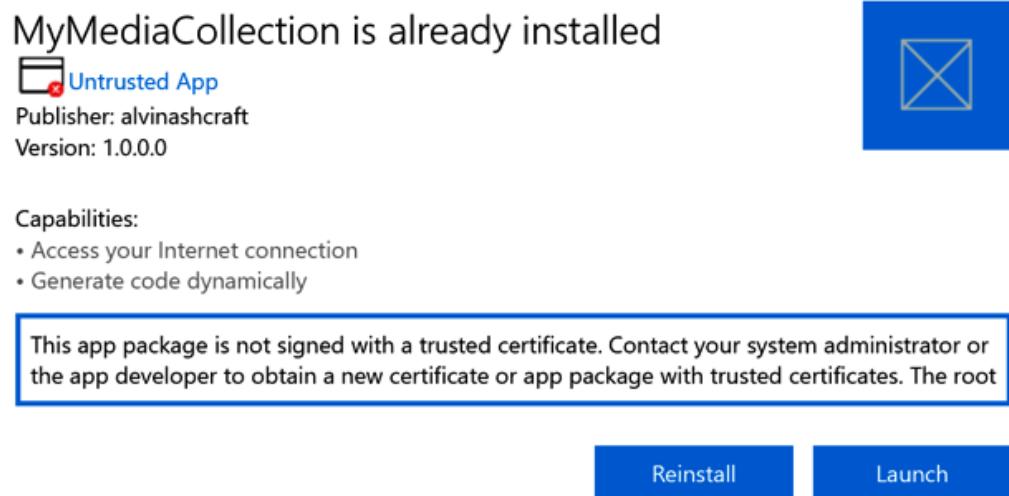


Figure 14.7 – Installing the MyMediaCollection MSIX bundle

Creating a package with Visual Studio is how most IDE users will choose to generate their installers. There are also tools for creating MSIX packages and bundles at the command line, which are documented on Microsoft Docs here: <https://docs.microsoft.com/en-us/windows/msix/package/manual-packaging-root>

Next, let's look at a new distribution method, Windows Package Manager.

## Deploying applications with Windows Package Manager

Windows Package Manager (WinGet) is a new, open source command-line package management tool from Microsoft. It is currently in preview, and the public 1.0 release date is not yet known. In this section, we will cover how to use the `WinGet` command to install published packages and the steps to add your own MSIX packages to the **Windows Package Manager community repository**. This is where `WinGet` finds available packages to install. The goal is to have `WinGet` eventually include all applications published to the Microsoft Store. So, if you plan to publish your application to the store, it is not necessary to also publish it to the `WinGet` repository.

Let's start by reviewing the steps to add a package to the community repository.

### Note

Because Windows Package Manager is still in preview, these steps may change. You can find current guidance on the community repository at <https://github.com/microsoft/winget-pkgs>.

## Adding a package to the community repository

To make your applications available to Windows users with the `WinGet` command, they must be published to Microsoft's Package Manager community repository. Any application published to this repository can be discovered and installed through the `WinGet` command in Windows.

### Note

This method of distribution is inherently less secure than distribution through the Microsoft Store. In theory, anyone with access to the public repository could extract the package and decompile your application.

To add your existing MSIX bundle to the repository, we will need to make it publicly available, create a WinGet manifest, and submit a **pull request (PR)** to have the manifest added to the community repository, as follows:

1. Start by pushing the contents of the installer folder we created (`C:\Installers\MyMediaCollection`) to a public URL. To do this, you could create a static website in Azure. This Microsoft Docs page walks through this process:

<https://docs.microsoft.com/en-us/azure/app-service/quickstart-html>

The files can be made available on any public URL. Another option with Azure is to host the files in Blob Storage.

2. After the application resource has been created in Azure, use the **Deployment Center** blade to choose one of these methods to get your deployment package files to the site: **OneDrive**, **Dropbox**, **FTP**, and so on. If you have an FTP client such as **FileZilla** (<https://filezilla-project.org/>), this is the simplest method for a one-time transfer. The **OneDrive** and **Dropbox** options will keep your site in sync with any changes in the file-sharing service. The deployment options can be seen in the following screenshot:

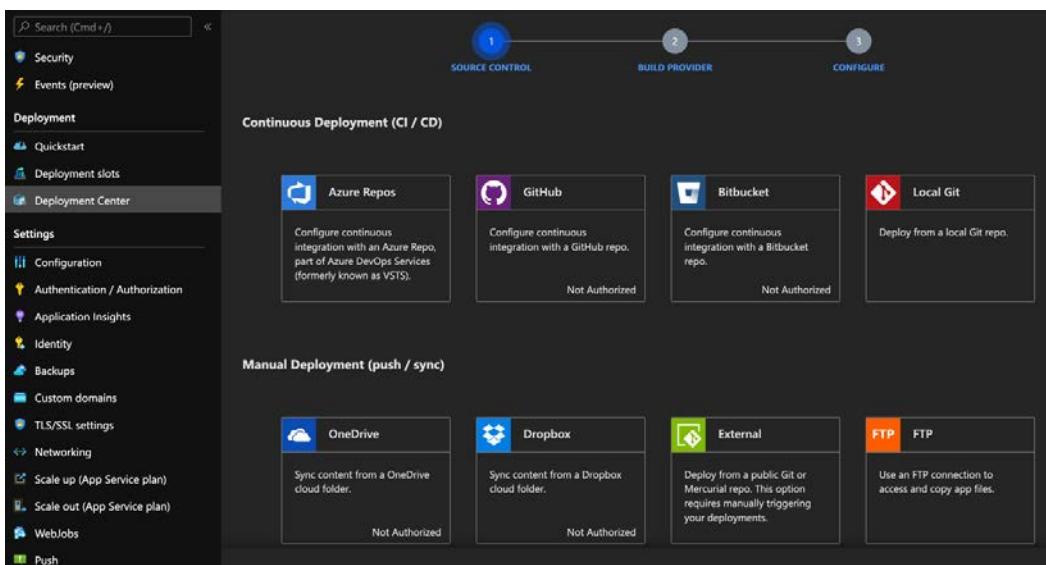


Figure 14.8 – Deployment options in Azure

When your package is in the cloud, you will be ready to create your manifest file for the community repository. The file is created in **Ain't Markup Language (YAML)** format. YAML files are becoming very popular for DevOps workflows.

For detailed instructions on creating a manifest and links to learn more about YAML, check out this Microsoft Docs page:

<https://docs.microsoft.com/en-us/windows/package-manager/package/manifest>

1. Name your YAML file `1.0.yaml`, and the contents should look like this:

```
Id: YourCompany.MyMediaCollection
Publisher: Your Company Name
Name: My Media Collection
Version: 1.0
License: Copyright (c) Your Company Name. All rights reserved.
InstallerType: msix
Installers:
- Arch: x64
  Url: https://mymediacollectionmsix.azurewebsites.net/MyMediaCollection_1.0.0.0_Debug_Test/
    MyMediaCollection_1.0.0.0_x86_x64_Debug.msixbundle
  Sha256: 712f139d71e56bfb306e4a7b739b0e1109abb662d
    fa164192a5cf6adb24a4e1 # SHA256
    calculated from installer
ManifestVersion: 0.1.0
```

2. To get the Sha256 value for your `.msixbundle` file, you can use this command:

```
certUtil -hashfile
C:\Installers\MyMediaCollection\
  MyMediaCollection_1.0.0.0_Debug_Test\
    MyMediaCollection_1.0.0.0_x86_x64_Debug.msixbundle SHA256
```

3. Next, test your manifest with WinGet. To install WinGet, you must join the preview program for the App Installer app at <http://aka.ms/winget-InsiderProgram>. You can then install the App Installer app from the Microsoft Store at <https://www.microsoft.com/en-us/p/app-installer/9nblggh4nns1>.
4. Test the syntax in your manifest with the following command:

```
winget validate <manifest-file-name>
```

5. Next, test installing the app from your manifest with this command:

```
winget install -m <manifest-file-name>
```

6. If your tests succeed, you're ready to submit a PR to the community repository. Use the following path structure when you submit the PR. Your publisher name must be unique. If you have an account on Partner Center for publishing apps, you should use the same publisher name here. package is the name of your application:

```
manifests\<publisher>\<package>\<version>.yaml
```

After the PR has been accepted, any users with WinGet will be able to install your application from the command line. Let's see how to use WinGet next.

## Using WinGet for package management

WinGet is the command-line client for Windows Package Manager. If you are familiar with other application package managers such as Chocolatey for Windows (<https://chocolatey.org/>) or Homebrew for macOS (<https://brew.sh/>), WinGet will feel familiar to you. A package manager allows you to install, list, and update applications on your operating system. In this section, we are going to see how to use WinGet to install **Windows Terminal**, Microsoft's modern command-line app, built with WinUI!

### Note

To read about how the Windows Terminal team created the application with WinUI, check out this blog post:

<https://devblogs.microsoft.com/commandline/building-windows-terminal-with-winui/>

Let's now look at the steps:

1. Now that you have the App Installer preview installed, open Command Prompt.
2. Test WinGet with the following command:

```
winget -?
```

The WinGet help and available commands should appear in your Command Prompt.

3. Now, run the `search` command to find the application we want to install, as follows:

```
winget search windowsterminal
```

You will see two results, as illustrated here:

```
PS C:\Users\alvinashcraft> winget search windowsterminal
Name Id Version
-----
Windows Terminal Preview Microsoft.WindowsTerminalPreview 1.4.2652.0
Windows Terminal Microsoft.WindowsTerminal 1.3.2651.0
PS C:\Users\alvinashcraft> |
```

Figure 14.9 – Performing a search with WinGet

4. To get more information about the package, use the `show` command, as follows:

```
winget show Microsoft.WindowsTerminal
```

This will return the following from the application's manifest:

```
PS C:\Users\alvinashcraft> winget show Microsoft.WindowsTerminal
Found Windows Terminal [Microsoft.WindowsTerminal]
Version: 1.3.2651.0
Publisher: Microsoft
Author: Microsoft
AppMoniker: Terminal
Description: The new Windows Terminal and the original Windows console host, all in the same place!
Homepage: https://github.com/microsoft/terminal
License: MIT
License Url: https://github.com/microsoft/terminal/blob/master/LICENSE
Installer:
  Type: Msix
  Download Url: https://github.com/microsoft/terminal/releases/download/v1.3.2651.0/Microsoft.WindowsTerminal_1.3.2651.0
  _8wekyb3d8bbwe.msixbundle
  SHA256: c91777b63e10e212ce098646bb685421185b259fe10cb37b2e093c2673d55a37
PS C:\Users\alvinashcraft>
```

Figure 14.10 – Viewing manifest information for a WinGet package

5. Next, enter this command to install Windows Terminal:

```
winget install Microsoft.WindowsTerminal
```

You should see a message that Windows Terminal was found and installed on your PC. If you already have Windows Terminal installed, you could try another app such as powertoys (**Windows PowerToys**—<https://github.com/microsoft/PowerToys>).

That's all there is to using WinGet. Because it is a command-line tool, you can build scripts that install all the software you need to get a new PC or **virtual machine (VM)** up and running.

Let's move along to our final section, where we will learn about distributing applications in the Microsoft Store.

## Distributing applications with the Microsoft Store

We have seen how to deliver WinUI applications to users through packages that can be sideloaded and with WinGet. There are a couple of other distribution channels available to Windows developers: **Microsoft Intune** for enterprise application distribution, and the Microsoft Store for consumer apps.

We briefly touched on Intune in the previous chapter when discussing App Center releases. A deeper dive into **Microsoft Endpoint Configuration Manager** and Intune is beyond the scope of this book, but if you are interested in learning how to distribute **line of business (LOB)** applications through them, you can read this Microsoft Docs article: <https://docs.microsoft.com/en-us/windows/uwp/publish/distribute-lob-apps-to-enterprises>

The Microsoft Store is the consumer app store for Windows users. The store accepts submissions for free and paid apps. Additional monetization options such as in-app purchases, sale pricing, and paid apps with a free trial period can also be configured.

In this section, we will cover the basics of submitting a free application to the store. If you want to learn more about monetizing your app, you can start here: <https://docs.microsoft.com/en-us/windows/uwp/publish/set-app-pricing-and-availability>

Let's see how to submit **MyMediaCollection** to the Microsoft Store.

## Preparing a free application for the Microsoft Store

In this section, we are going to use Visual Studio to prepare and submit the **MyMediaCollection** application to the Microsoft Store. The Microsoft Store is the primary delivery outlet for consumer applications. Before you start this process, you will need to have an account on the **Windows Dev Center**. Let's do this now, as follows:

1. First, visit the Windows Dev Center (<https://developer.microsoft.com/en-us/store/register/>) and click **SIGN UP**.
2. Sign in with a Microsoft account and select the country or region where you or your company is located.

**Note**

This account will be the owner of the Store account and cannot easily be changed. If you are creating the Store account for an organization, it is recommended to use a separate Microsoft account that is not tied to any individual at the organization or company.

3. Choose **Individual or Company** as your **Account Type**.
4. Enter your **Publisher Display Name**. This is the name that will be seen by users on your public application store listings, so choose this carefully.
5. Enter your contact details—this information will be used by Microsoft to contact you if there are any issues or updates with your store listings. Click **Next** to continue, to enter payment information.
6. Pay a one-time store registration fee. Microsoft charges a small fee the first time you register for an account on the store. This helps to prevent fraudulent and malicious accounts from being created. In most countries, the fee is *\$19 US Dollars (USD) for individuals and \$99 USD for a company account*.

The full listing of fees by country is available here:

<https://docs.microsoft.com/en-us/windows/uwp/publish/account-types-locations-and-fees#developer-account-and-app-submission-markets>

7. When you are finished, click **Review**.
8. Review the details for your account and the **App Developer Agreement** and click **Finish** to confirm your registration. Payment will be processed at this time and you will receive a confirmation email.

Now that you have an account on the Microsoft Store, we can proceed with submitting our first app, as follows:

1. Return to Visual Studio and right-click on the **MyMediaCollection** project in **Solution Explorer**.
2. Select **Publish | Create App Packages**. This process starts out the same as when creating a package for sideloading.
3. On the **Select distribution method** screen, select **Microsoft Store under a new app name** and click **Next**.
4. On the **Select an app name** screen, ensure the Microsoft account linked to your store account is selected and enter your desired application name in the **Reserve a new app name** field. Click **Reserve** to check if the name is available. Each app name must be unique across the Microsoft Store. The app name will appear in your list of apps, as illustrated in the following screenshot:

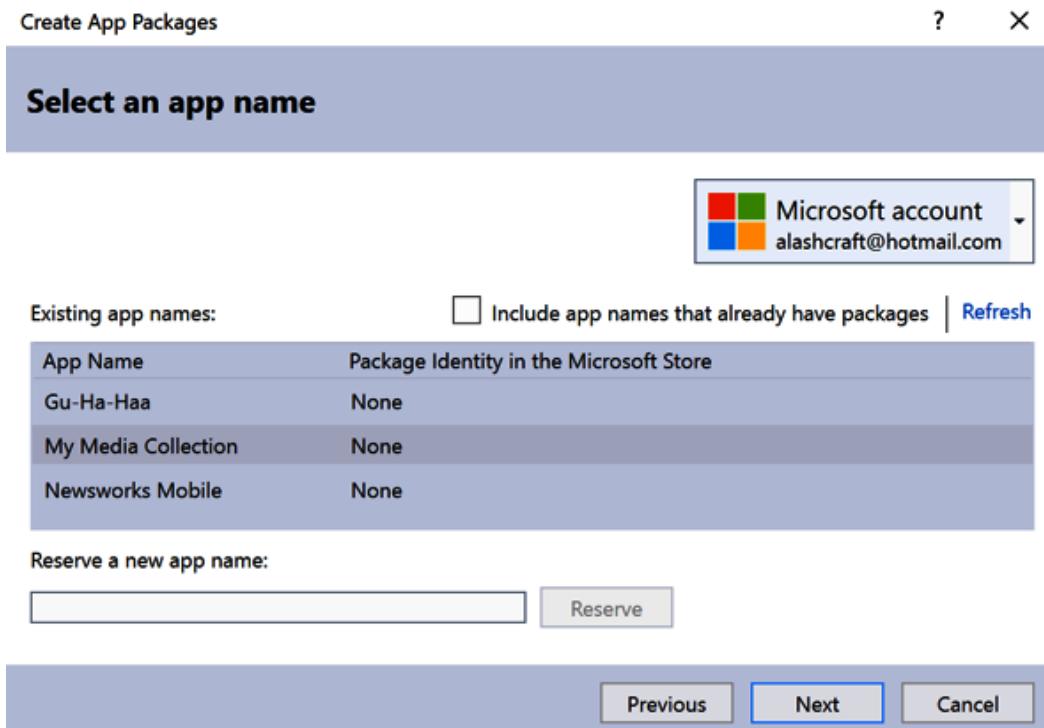


Figure 14.11 – Selecting an app name for the Microsoft Store listing

5. Select the app name in your list of **Existing app names** and click **Next**.
6. You can leave the default values on the **Select and configure packages** screen unless you want to update **Version**. Click **Create** at the bottom of the screen, as illustrated in the following screenshot:

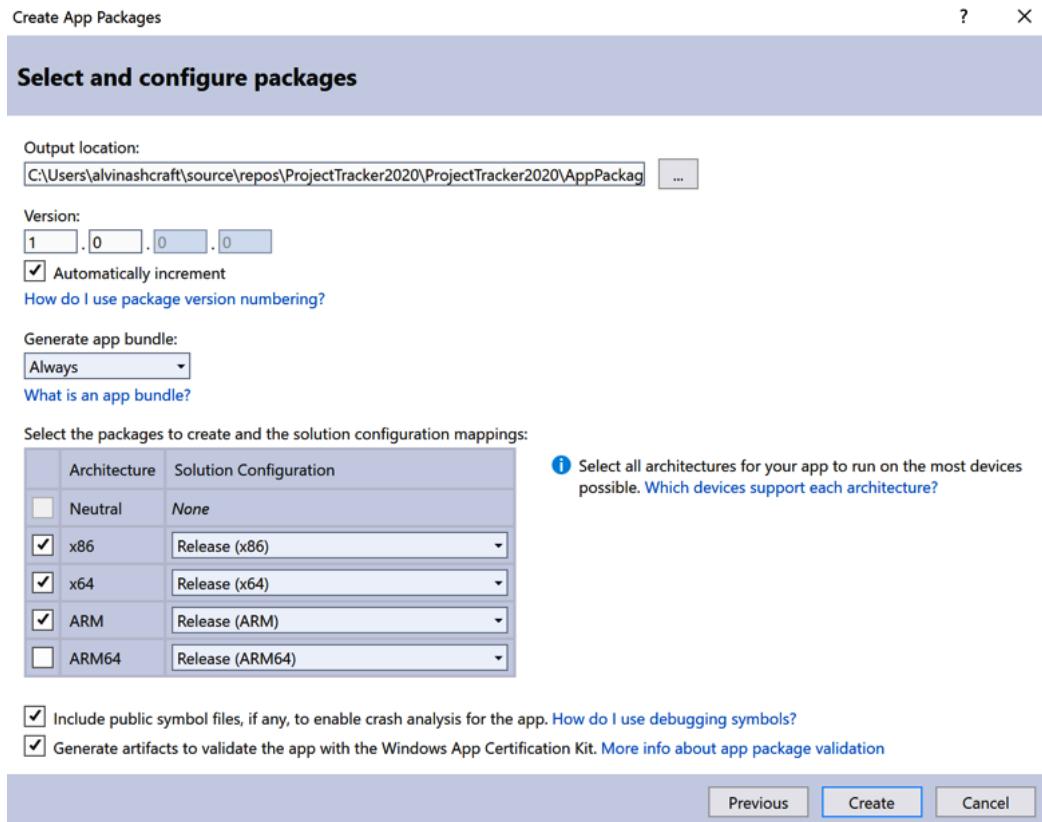


Figure 14.12 – The Select and configure packages screen

7. Visual Studio will build your project and prepare the package for the store. Next, your app needs to pass a local validation process. On the **Finished creating package** screen that appears, click the **Launch Windows App Certification Kit** (a.k.a the **WACK** test) button to run the validation on your local machine. If you feel your submission is ready to go after reviewing the submission checklist (<https://docs.microsoft.com/en-us/windows/uwp/publish/app-submissions#app-submission-checklist>), you can check the **Automatically submit to the Microsoft Store** checkbox before starting the validation. This will submit your package to the store upon a successful validation run.
8. When the certification kit launches, keep all the tests selected and click **Next**. The tests will take several minutes to complete, and the application will launch several times during the validation process, as illustrated in the following screenshot:

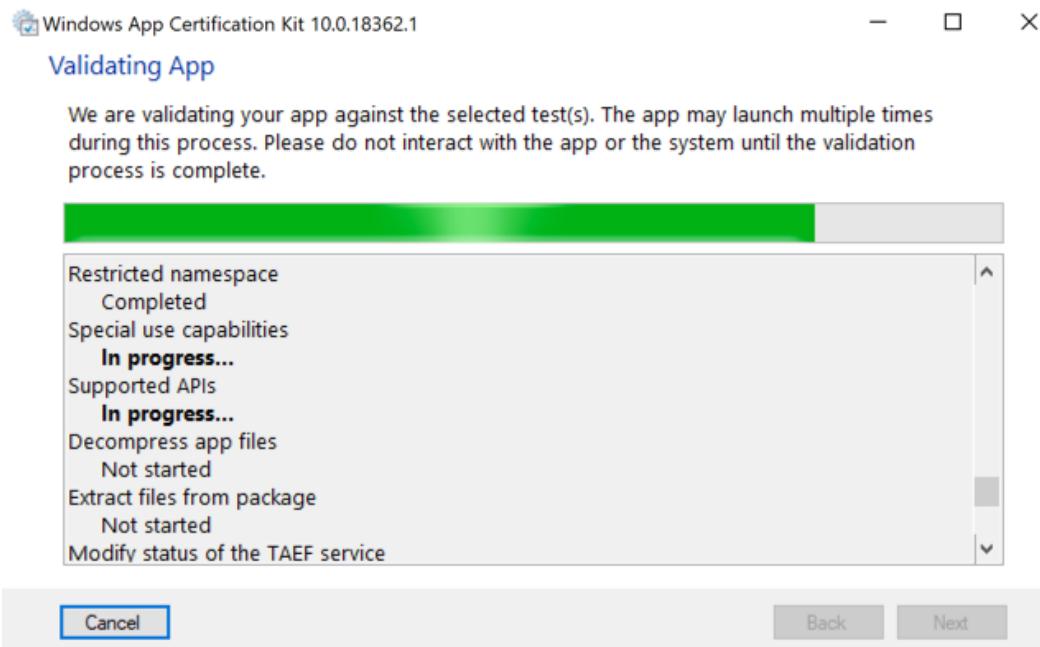


Figure 14.13 – Microsoft Store application validation in process

9. If you run the process with a WinUI 3.0 preview, you will likely get some failures in the validation. Upon completion, you can click the **Click here to view the results** option to see which tests passed or failed. All failures should be addressed before continuing with the store submission. These tests will be run during the Microsoft Store approval process, and any failures here are likely to cause the submission to be rejected. For details on the tests and corrective actions that can be taken for failures, you can review this Microsoft Docs page: <https://docs.microsoft.com/en-us/windows/uwp/debug-test-perf/windows-app-certification-kit-tests>

When you have a validated application ready to submit to the store, you can continue the process. Let's walk through how to submit an application through the Microsoft Partner Center website.

## Uploading a package to the Store

In this section, we will walk through submitting a package created by Visual Studio to the Microsoft Store using the Partner Center dashboard. To do this, proceed as follows:

1. Start by logging in to the Partner Center with your Microsoft account at the following URL:  
<https://partner.microsoft.com/en-us/dashboard/home>
2. Click **Windows and Xbox**, to be taken to the **Overview** page for submitting apps to these platforms.
3. You will see a list of reserved app names and submitted apps on your account here. If you do not yet have an app name reserved, you can click **Create a new app**. I am going to select **Simple Task Tracker** from my apps to submit an initial version of this application, as illustrated in the following screenshot:

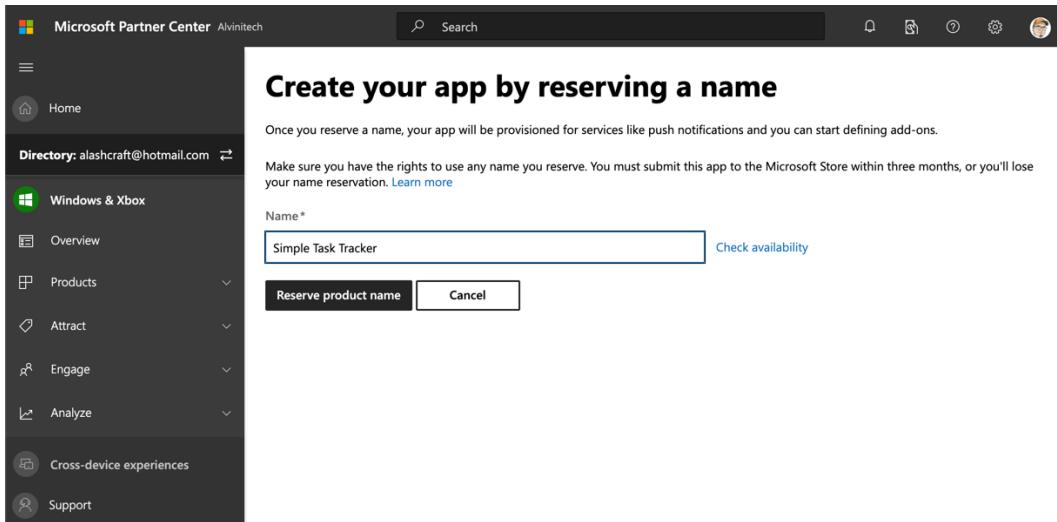


Figure 14.14 – Reserving a new name for your application

4. On the **Application overview** page for your selected app, click **Start your submission**, as illustrated in the following screenshot:

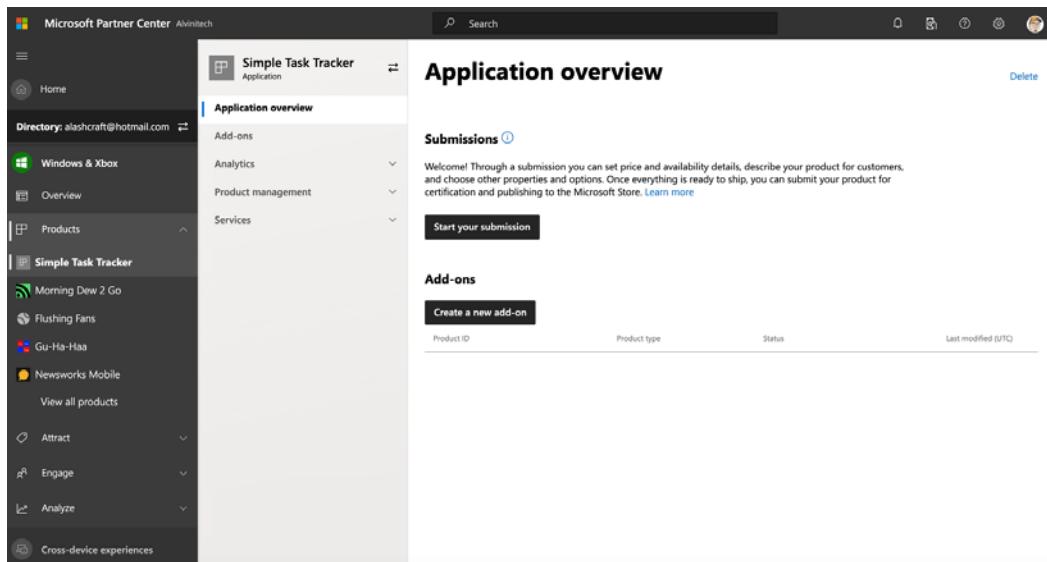


Figure 14.15 – Starting your new application submission

5. Begin the submission by selecting the **Pricing and availability** section.
6. We will keep all the default settings in this section, except for the **Base price** option. A selection must be made here. Choose the **Free** option or select a base price from the list. When you are done, select **Save draft**.
7. Next, select the **Properties** section. Select a category and enter your **Support info** data. Enter any relevant data for your app in the **Display mode**, **Product declarations**, and **System requirements** sections, and click **Save**.

The **Properties** section can be seen in the following screenshot:

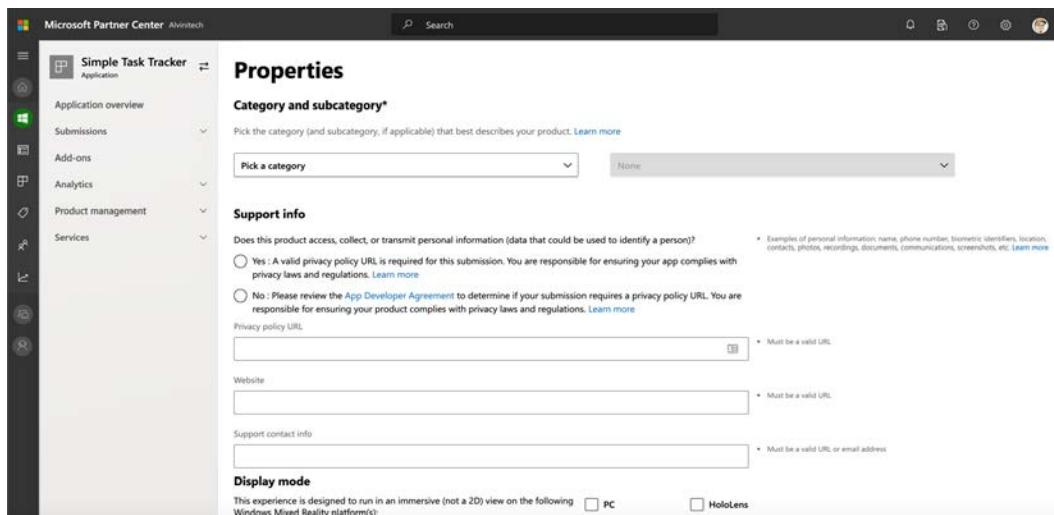


Figure 14.16 – The Properties section of the application submission

8. Complete the questionnaire on the **Age ratings** page and click **Save and generate**. This page determines if your app should be restricted to certain age groups based on the data it collects or exchanges with other users. If everything looks good after generation, click **Continue**.
9. Select the **Packages** page. On this page, you can browse to your **.msixupload** file and upload it to the site for submission. Choose the compatible device families and click **Save**.

10. Next, select a language under the **Store listings** section. Supporting multiple languages and specifying them here makes it more likely that your application will be installed in different countries. Add an application **Description** and at least one screenshot of your application. The remaining fields are optional, but the more you complete, the easier it will be for customers to find your app and the more likely they are to try it. When everything is completed, click **Save**.

Some example language options are shown in the following screenshot:

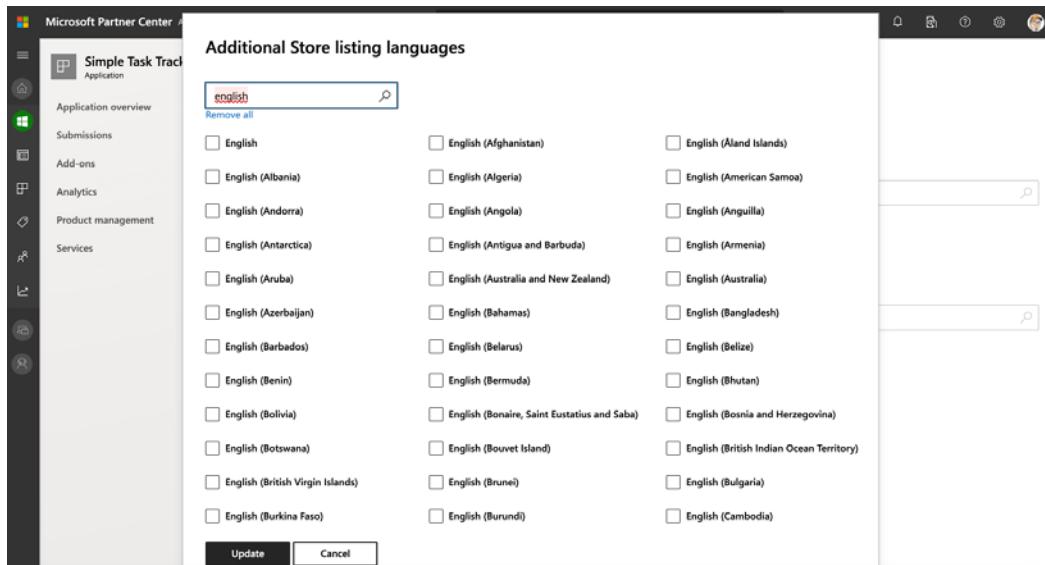


Figure 14.17 – Selecting languages for the application submission

11. Completing the **Submission options** page is optional. By default, your app will be published immediately after passing certification. I am going to select **Don't publish this submission until I select publish now** because I do not want this app to be available in the store until I make some additional backend changes.
12. Finally, click **Submit to the Store**. Your app will be submitted for certification. If it passes, it will be available in the Microsoft Store when you have indicated this on the **Submission options** page. If your app fails validation, you will receive a list of issues to address before attempting another submission.

Those are the basic steps for submitting a new application to the Microsoft Store. For more detailed scenarios, and information about updates and add-ons, you can review the Microsoft Docs *App submissions* documentation at <https://docs.microsoft.com/en-us/windows/uwp/publish/app-submissions>.

Next, we are going to cover how to sideload applications in Windows with MSIX.

## Sideload WinUI applications with MSIX

In this section, we will create an MSIX package for a WinUI project and learn how to sideload it on Windows 10. When you sideload an application, you install it directly with the MSIX UI or with PowerShell commands. This method of installation is important to understand as it is frequently used by enterprises to distribute applications internally.

We are going to start by creating a package for sideloading.

### Creating an MSIX package for sideloading

In this section, we will create a package for a WinUI project with Visual Studio. You can start by either opening an existing WinUI project or by creating a new, empty WinUI project. I have created a new project named **ProjectTracker**. Proceed as follows:

1. First, right-click the project in **Solution Explorer** and select **Publish | Create App Packages**.
2. On the **Select distribution method** screen of the **Create App Packages** window, leave the **Sideload** radio button and **Enable automatic updates** checkbox selected. Click **Next**.
3. On the next page, you select a signing method. Select **Yes, select a certificate** and click the **Create** button. Here, you will create a self-signed certificate. By using a self-signed certificate, any users who install the app will need to trust your package and import the certificate from the MSIX package. We will explain this process in the next section when we sideload the package. Enter a name and password for the certificate, as illustrated in the following screenshot:

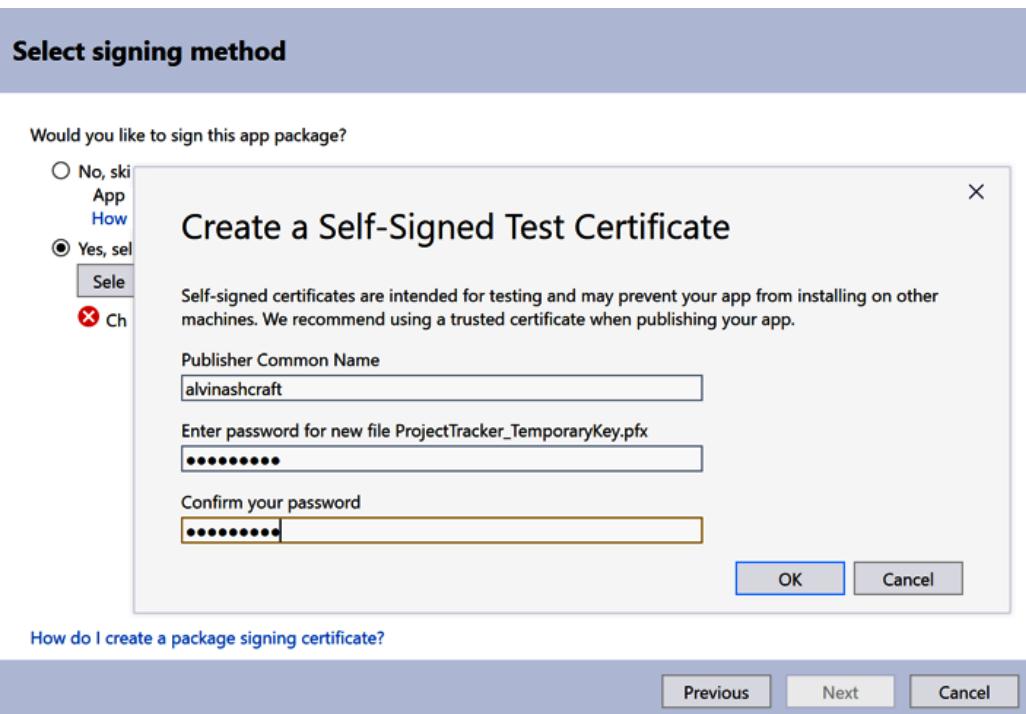


Figure 14.18 – Creating a self-signed certificate for the package

4. Select **Next** and leave the default settings on the **Select and configure packages** page. Click **Next**.
5. Enter the **Installer path**. This can be a local file path or a network location. Click **Create**. Your project will compile and the package will be created in the specified location.

Now that we have an MSIX package for our project, we're ready to sideload it. Let's walk through this process.

## Sideload an MSIX package

In this section, we will learn how to sideload a WinUI application with MSIX. We created a new MSIX package in the previous section. Navigate to the folder where the package was created inside your project's folder and review the files in the package folder, named `ProjectTracker_1.0.0.0_Debug_Test` in my case. The following screenshot illustrates this:

Name	Date modified	Type	Size
Add-AppDevPackage.resources	10/17/2020 1:22 PM	File folder	
Dependencies	10/17/2020 1:22 PM	File folder	
TelemetryDependencies	10/17/2020 1:22 PM	File folder	
Add-AppDevPackage.ps1	9/19/2020 9:31 AM	Windows PowerShell ...	36 KB
Install.ps1	9/19/2020 9:31 AM	Windows PowerShell ...	14 KB
ProjectTracker_1.0.0.0_ARM_Debug.appxsym	10/17/2020 1:22 PM	APPXSYM File	6 KB
ProjectTracker_1.0.0.0_x64_Debug.appxsym	10/17/2020 1:22 PM	APPXSYM File	6 KB
ProjectTracker_1.0.0.0_x86_Debug.appxsym	10/17/2020 1:22 PM	APPXSYM File	6 KB
ProjectTracker_1.0.0.0_x86_x64_arm_Debug.cer	10/17/2020 1:22 PM	Security Certificate	1 KB
ProjectTracker_1.0.0.0_x86_x64_arm_Debug.msixbundle	10/17/2020 1:22 PM	MSIXBUNDLE File	5,753 KB

Figure 14.19 – Reviewing the package files

Let's now look at the steps:

1. To install this package on another Windows PC, start by copying the `ProjectTracker_1.0.0.0_x86_x64_arm_Debug.msixbundle` file to the machine. This file contains all of the necessary information and files to install the application.
2. First, we need to install the self-signed certificate used to sign the package. You can install it by running the `Install.ps1` PowerShell script, but you can also install it at the time it was created if you are installing it on the same machine. We will install it by right-clicking the MSIX bundle file and selecting **Properties**.
3. Click the **Digital Signatures** tab and select the certificate in the **Signature list** box, as illustrated in the following screenshot:

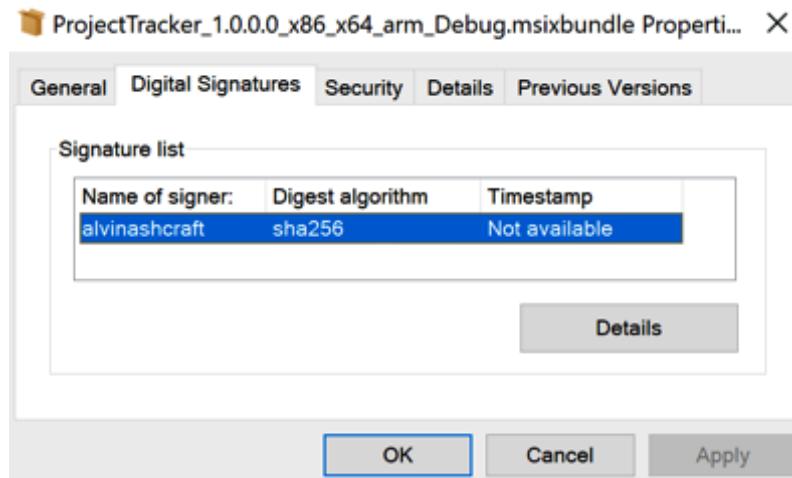


Figure 14.20 – Selecting the certificate in the MSIX package properties

4. Click **Details** to open the **Digital Signature** screen **Details**.
5. Click the **View Certificate** button. On the **Certificate** page that opens, click **Install Certificate**.
6. When completing **Certificate Import Wizard**, import the certificate to **Local Machine**, and on the **Certificate Store** page, select **Place all certificates in the following store**, as illustrated in the following screenshot:

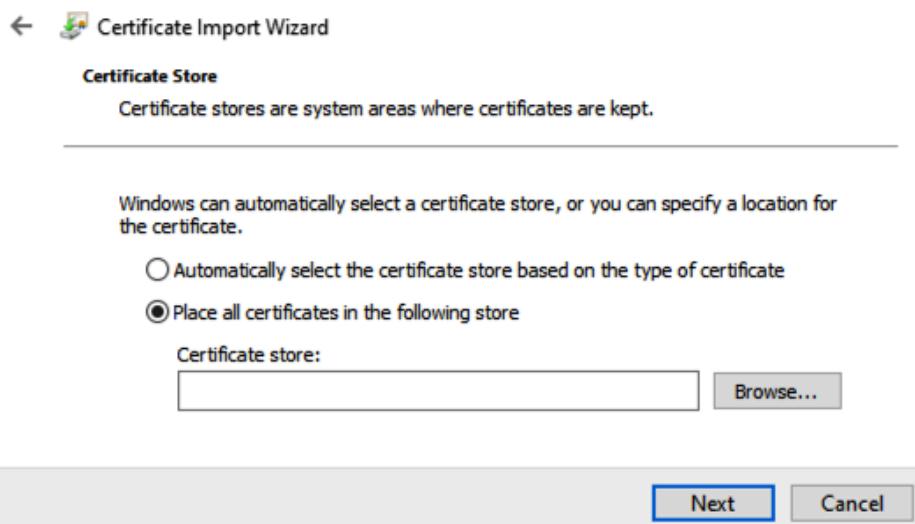


Figure 14.21 – Importing the package certificate

7. Click **Browse...** and select the **Trusted Root Certification Authorities** folder. Click **OK** on the dialog and click **Next** on the wizard.
8. After clicking **Finish**, the certificate will be imported. If all goes well, you will receive a message that the certificate was successfully imported. You can close the **Properties** page and continue with the package installation.
9. Now, double-click the **MSIXBUNDLE** package file. The installer will open a window with some of the application's manifest information. Click **Install** and leave the **Launch when ready** checkbox selected, as illustrated in the following screenshot:

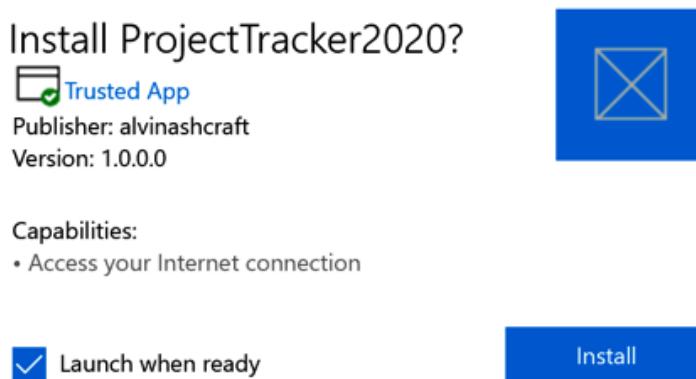


Figure 14.22 – Installing a trusted application from its MSIX package

The application will install and launch, and you're ready to go. Notice that the package is trusted because we imported the certificate to the **Trusted Root Certification Authorities** folder.

This installation can be automated with **PowerShell** if the MSIX certificate is already trusted on the target machines. Use the **Add-AppPackage** command to install an MSIX package or MSIX bundle from a PowerShell prompt, as illustrated in the following code snippet:

```
Add-AppPackage -path C:\Installers\ProjectTracker_1.0.0.0_  
x86_x64_arm_Debug.msixbundle
```

If you have several packages to distribute, you can create a custom PowerShell script to iterate over all of the MSIX packages in a given folder. For more information about PowerShell scripting with MSIX, check out the following Microsoft Docs page: <https://docs.microsoft.com/en-us/windows/msix/desktop/powershell-msix-cmdlets>

Let's wrap up, and review what we have learned in this chapter.

## Summary

In this chapter, we reviewed various methods of delivering WinUI applications to consumers. We learned the basics of MSIX packages and how to create packages to sideload our applications. We also covered the process of creating an account on the Microsoft Partner Center in order to create application submissions on the Microsoft Store.

Next, we validated and submitted an application package to the Store. Finally, we learned how to manually sideload MSIX packages and how PowerShell can be leveraged to automate the sideloading process. These concepts will help you when you are ready to create your own WinUI applications for enterprise or consumer use.

This is the final chapter of our book. I hope that the concepts covered in each chapter will help you succeed in your quest to become a WinUI application developer.

## Questions

1. What are some of the application installer formats that preceded MSIX?
2. Is MSIX only for UWP and WinUI apps?
3. In a WinUI project, which file contains the application manifest data?
4. Which command is used to install packages with Windows Package Manager?
5. How can you make your application available with WinGet?
6. What is the name of the online portal for submitting applications to the Microsoft Store?
7. How many screenshots are required in a Microsoft Store listing?



# Assessments

## Chapter 1

1. Windows 10
2. MVVM
3. False—the schema and underlying APIs are different
4. WPF
5. 2.0
6. Rendering the UI on the GPU—more secure by default
7. No—C++ is also available
8. `<Style TargetType="Button" />`

## Chapter 2

1. Features and workloads can be added or removed from Visual Studio with the Visual Studio Installer.
2. The minimum version of Windows to target for WinUI 3.0 apps is version 17134.
3. Application-wide resources can be added in `App.xaml`.
4. `MainPage`
5. `Grid`
6. `StackPanel`
7. `MessageDialog`
8. `Canvas`

## Chapter 3

1. Model-View-ViewModel
2. The Model layer
3. Prism, MVVM Light, or MVVMCross
4. `INotifyPropertyChanged`
5. `ObservableCollection<T>`
6. `SelectedItem`
7.  `ICommand`
8. `Assert.IsNotNull(object)`

## Chapter 4

1. Dependency Injection is one form of Inversion of Control
2. Call the `GoBack()` method on the current `Window.Content` `Frame`
3. **A DI container**
4. `GetService()`
5. **LINQ**
6. Mocking frameworks use interfaces to imitate parts of objects required for tests to run successfully. Fake objects are alternate interface implementations used in place of an application's implementation of the same interface.
7. `SetupAllProperties()`

## Chapter 5

1. `AnimatedVisualPlayer`
2. `WebView2`
3. **XamlDirect**
4. `TeachingTip`
5. **XAML Controls Gallery**
6. `Windows.Storage.ApplicationData.Current.RoamingSettings`
7. Windows 10 Creator's Update (released in spring 2017)

## Chapter 6

1. When it has been in the background for a few seconds.
2. This can be handled in the `EnteredBackground` (recommended) or `Suspending` events.
3. `CommandLineLaunch`.
4. A Micro ORM is a framework that provides mapping between objects and a data source.
5. Track changes to objects.
6. `Dapper.Contrib`.
7. `MinLength` and `MaxLength`.

## Chapter 7

1. iOS, Android, macOS, Windows, web, and cross-platform (React Native)
2. A group of controls that work together to perform a user interaction
3. Segoe UI
4. Reveal Focus
5. Standard and Compact
6. RequestedTheme
7. Figma, Sketch, Adobe XD, Adobe Illustrator, and Adobe Photoshop

## Chapter 8

1. The Windows Application Packaging Project
2. In the **Capabilities** section of the `package.appxmanifest` in the packaging project
3. Create a .NET class library and reference it from both desktop projects
4. A control library (WinUI in Desktop)
5. `DependencyProperty`
6. In the `OnLaunched` method in `App.xaml.cs`
7. In the **Visual Assets** section of `package.appxmanifest` in the packaging project

## Chapter 9

1. UWP Community Toolkit
2. WebViewCompatible
3. MarkdownTextBlock
4. GroupedObservableCollection
5. WinUI on Desktop
6. Dropbox
7. ThemeListener
8. InkCanvas and InkToolbar

## Chapter 10

1. Both WPF and WinForms on .NET Core require version 3.0 or later.
2. If your WPF or WinForms project uses .NET Framework, the minimum version to use WebView2 is 4.6.2.
3. The WebViewCompatible control can be used on Windows 8.x and Windows 7.
4. WindowsXamlHost.
5. MapControl.
6. InkCanvas and InkToolbar.
7. WebView renders content with the legacy Microsoft Edge browser, while WebView2 renders content with the new Chromium-based Edge browser.

## Chapter 11

1. Change the XAML designer's screen size to the 42" Xbox resolution
2. **Debug | Attach to Process**
3. **Debug | Other Debug Targets | Debug Installed App Package**; then select the remote machine and the package to debug; click **Start**
4. Select **Device** on the **Debug Installed App Package** window
5. **Live Visual Tree**
6. **OneTime**
7. **The Live Property Explorer**

## Chapter 12

1. Blazor WASM
2. Blazor Server
3. Razor
4. `dotnet run`
5. GitHub Pages
6. Azure Static Web Apps
7. `WebView2`
8. GitHub Actions

## Chapter 13

1. No. You can also create an App Center account with other providers: Facebook, Google, or GitHub.
2. Unlimited.
3. Free users are limited to 240 build minutes per month.
4. Create distribution groups and assign a group to a release.
5. Azure DevOps, GitHub, and Jira.
6. On the **Analytics | Log flow** page.
7. By adding `Crashes .TrackError` calls to the exception handlers in your code and passing the `Exception` object.

## Chapter 14

1. APPX for Windows Store apps and MSI for desktop apps
2. No—MSIX can also package desktop apps for distribution
3. `package.appxmanifest`
4. WinGet
5. Create a publicly available MSIX package and submit a PR with your manifest file
6. The Microsoft Partner Center dashboard
7. At least one





Packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

## Why subscribe?

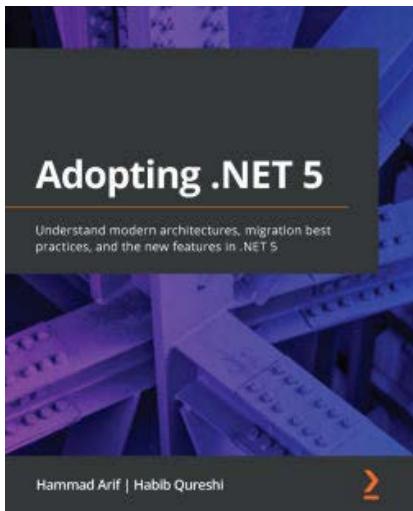
- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at [packt.com](http://packt.com) and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at [customercare@packtpub.com](mailto:customercare@packtpub.com) for more details.

At [www.packt.com](http://www.packt.com), you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

# Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

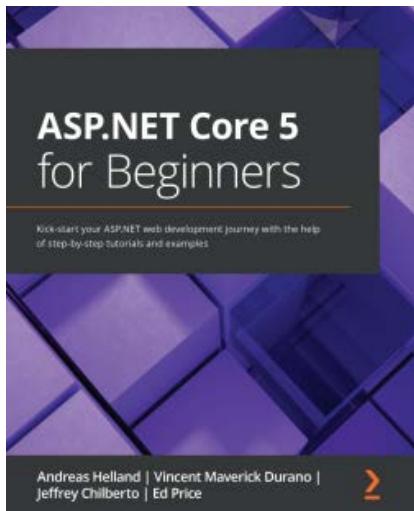


## **Adopting .NET 5**

Hammad Arif, Habib Qureshi

ISBN: 978-1-80056-056-7

- Explore the key performance improvement areas when migrating to modern architectures
- Understand app design and development using .NET 5
- Discover how to shift from legacy to modern application design using microservices and cloud-native architecture
- Explore common migration pitfalls and make the right decisions in situations where multiple options are available
- Understand the process of deploying .NET 5 code on serverless and containerized hosts, along with its benefits
- Find out what ML.NET has to offer and build .NET apps that use machine learning services



### **ASP.NET Core 5 for Beginners**

Andreas Helland , Vincent Maverick Durano , Jeffrey Chilberto, Ed Price

ISBN: 978-1-80056-718-4

- Explore the new features and APIs introduced in ASP.NET Core 5 and Blazor
- Put basic ASP.NET Core 5 concepts into practice with the help of clear and simple samples
- Work with Entity Framework Core and its different workflows to implement your application's data access
- Discover the different web frameworks that ASP.NET Core 5 offers for building web apps
- Get to grips with the basics of building RESTful web APIs to work with real data
- Deploy your web apps in AWS, Azure, and Docker containers
- Work with SignalR to add real-time notifications to your app

## Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit [authors.packtpub.com](http://authors.packtpub.com) and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

## Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

# Index

## Symbols

.NET 5  
WinUI project, creating with 204  
.NET 5 libraries  
referencing, form WinUI  
desktop project 215-218  
sharing, with WPF application  
218, 220-222  
.NET Core 317  
.NET Standard  
reference link 215, 252  
.NET Standard 2.0 package 252

## A

ActivationKind  
examples 151

Adobe Illustrator (AI)  
about 202  
URL 202

Adobe Photoshop  
about 202  
URL 202

Adobe XD  
about 201  
URL 201

Advanced RISC Machines (ARM) 6  
Ain't Markup Language (YAML) 378  
Amazon S3  
URL 329  
Amazon Web Services (AWS) 328  
AnimatedVisualPlayer control 125  
App Center SDK  
code, instrumenting 358, 360, 361  
Apple's Human Interface Guidelines  
reference link 181  
application  
analytics 357  
monitoring 357  
application deployment, with App Center  
beta testers, creating 354-357  
early releases of application,  
creating 354-357  
application programming  
interfaces (APIs) 5, 77  
applications  
deploying, with Windows  
Package Manager 376  
distributing, with Microsoft Store 381  
applications packaging  
discovering 366  
in Visual Studio 371-374, 376

- purposes 366, 367
- Application Virtualization (App-V) 370
- app life cycle events
- application state, managing with 148
- artificial intelligence (AI) 183
- ASP.NET
- history 316, 317
- ASP.NET Core
- about 316
  - history 316, 317
- ASP.NET Core, features
- Identity Core 317
  - Razor Pages 317
  - SignalR 317
- Azure Active Directory (AAD) 344
- Azure App Service
- URL 329
- Azure Static Web Apps
- URL 330
- Azure Static Web Apps resource
- creating 333, 335
- 
- ## B
- Binding markup extension
- about 78, 79
  - properties 78
- Blazor
- about 316, 318
  - code, pushing to GitHub
    - repository 330, 332, 333
  - publishing, to Azure Static Web Apps hosting 330
- Blazor Server 318
- Blazor Wasm application
- building, to track tasks 323-328
  - creating 320-323
  - deployment options, exploring 328
- 
- ## C
- Blazor WebAssembly 318
- Blend 20
- 
- Cascading Style Sheets (CSS) 18
- central processing unit (CPU) 31
- Chocolatey
- URL 379
- Chromium project
- URL 209
- CloudFront
- URL 329
- code-behind file 43
- Codeplex 231
- coding
- with XAML Hot Reload 307
- collection data
- updating, with
    - INotifyCollectionChanged interface 81, 82
- commands
- ICommand, implementing 86, 87
  - using, in ViewModel 88, 89
- View, updating 90, 91
- working with 86
- compiled bindings 29
- Continuous Integration (CI) 20
- controls
- adding, to MainWindow 247-249
- controls, adding to project
- about 139
- SplitButton control, using 140, 141
- TeachingTip control, adding
- to save button 142-144
- Create, Read, Update, Delete (CRUD) 158

**D**

Dapper  
 adding, to project 159-163  
 lambda expression, assigning 160  
 reference link 158

data  
 adding, to DataGrid 244-246  
 retrieving, via services 166-171

data binding errors  
 identifying 301

data binding, in WinUI  
 about 77  
 Binding markup extension 78, 79  
 collection data, updating with  
`INotifyCollectionChanged`  
 interface 81, 82  
 markup extension 78  
 view data, updating with  
`INotifyPropertyChanged`  
 interface 80, 81  
`x:Bind` markup extension 79

data binding, mistakes  
 about 301  
 best binding mode, selecting 302  
`ObservableCollection<T>`,  
 working with 302, 303  
`PropertyChanged` notifications,  
 triggering 302

DataGrid  
 data, adding to 244-246

data service  
 creating 109-112  
 initialization, updating 163-165

DB Browser for SQLite 171

debugging  
 in Visual Studio 289

with Live Property Explorer  
 308, 309, 311-313  
 with Live Visual Tree 308, 309, 311-313

Dependency Injection (DI)  
 about 77  
 basics 96, 97  
 using, with ViewModel classes 97-100

deployment options, Blazor Wasm projects  
 about 328  
 Amazon Web Services (AWS) 328  
 Azure App Service 329  
 Azure Static Web Apps 330  
 GitHub Pages 329

Desktop project  
 WinUI, creating in 241-243

developer mode activation  
 reference link 295

Distribution Groups 354

Domain-Driven Design (DDD) 75

dots per inch (DPI) 31

dynamic link libraries (DLLs) 45

**E**

Elastic Container Service (ECS)  
 URL 328

Entity Framework Core (EF Core) 158

events  
 working with 86

explicit styles 298

Extended WPF Toolkit  
 reference link 231

Extensible Application Markup  
 Language (XAML)  
 about 11-13  
 adaptive UI, creating for devices 14, 15  
 basics 46

data-binding initialization, completing 53, 55  
DataTemplate, creating 55-57  
improving, with static code analysis 299-301  
initial UI, building 52, 53  
model, building 46, 48, 49  
powerful data, binding 15-18  
sample data, creating 50-52  
UI, binding 55-57  
UI, styling with 18-20  
Extensible Markup Language (XML) 11

## F

Fargate  
  URL 328  
Figma tool  
  about 201  
  URL 201  
Fluent Design  
  exploring, for Windows applications 181  
  incorporating, in WinUI  
    applications 186  
  resources, designing for 201  
  toolkits, designing for 201  
Fluent Design aspects, for  
  Windows applications  
  about 181, 186  
  controls 182  
  input 184, 185  
  layout design 184  
  patterns 183  
  style 185, 186  
Fluent Design, in WinUI applications  
  Style of ItemDetailsPage,  
    changing 193, 194  
  style of MainPage, changing 188-192

title bar, updating 186-188  
Fluent Design System 180, 181  
Fluent Design System, core principles  
  engaging and immersive 181  
  intuitive and powerful 181  
  natural on every device 181  
Fluent XAML Theme Editor  
  Colors tab 196, 197  
  reference link 194  
  Shapes panel 197  
  using 194-196  
forms pattern  
  reference link 183  
free App Center accounts  
  limitations 342  
free applications  
  preparing, for Microsoft Store 382-386

## G

GIMP  
  URL 202  
GitHub Actions  
  about 329  
  applications, publishing with 336-338  
GitHub Pages  
  reference link 329  
GitHub repository  
  Visual Studio App Center,  
    integrating with 351, 352, 354  
Google Material Design  
  URL 181  
graphics processing unit (GPU) 31

## H

Homebrew  
  URL 379

**I**

implicit style 298

INotifyCollectionChanged interface

- about 82
- used, for updating collection data 81, 82

INotifyPropertyChanged interface

- used, for updating view data 80, 81

input injection

- reference link 185

integrated development

- environment (IDE) 35, 366

interfaces and services

- adding 106

Internet Explorer (IE) 278

Inversion of Control (IoC) 76, 96

Inversion of Control (IoC),

- implementing ways

about 97

delegate 97

event 97

Service Locator Pattern 97

ItemDetailsPage

- adding 102-104, 106
- parameters, handling 116

ItemDetailsViewModel class

- creating 116-120

**L**

Language Integrated Query (LINQ) 111

Learn

- URL 330

life cycle events, WinUI applications

- about 149, 150
- apps, running in foreground
- and background 152

OnActivated method 151, 152

OnLaunched method 151

resuming, from Suspended state 153

Suspended state 152

line of business (LOB) 381

ListView class

- reference link 52

ListView control 160

Live Property Explorer

- used, for debugging 308, 309, 311-313
- used, for debugging live data 306

Live Visual Tree

- used, for debugging 308, 309, 311-313
- used, for debugging live data 306

locally installed WinUI application

- debugging 292-294

local WinUI applications

- debugging 289
- debugging, with XAML
- Designer 290-292

Long-Term Servicing Channel (LTSC) 367

**M**

MainWindow

- controls, adding to 247-249

Manifest Designer

- visual assets 213, 214

markup extensions

- about 78
- reference link 57

master/details pattern

- reference link 183

Micro-ORM

- leveraging, to simplify data access 158

Microsoft Azure Cognitive Services 253

Microsoft Edge Insider

- download links 210

- Microsoft Endpoint Configuration Manager 381
- Microsoft Intune 381
- Microsoft Software Installer (MSI) 370
- Microsoft Store
- applications, distributing with 381
  - free application, preparing for 382-386
  - package, uploading to 386, 388, 389
- Microsoft Tech Community 370
- Microsoft Tech Community article
- reference link 275
- Microsoft.Toolkit.Mvvm package 254
- middleware authors
- XamlDirect APIs, exploring for 137-139
- Model layer 74
- Model-View-Controller (MVC) 317
- Model-View-ViewModel
- (MVVM) pattern
  - about 16, 20, 74
  - components 75, 76
  - data validation, performing with 172-174
  - implementing, in WinUI application 82-85
  - Model layer 74
  - presentation, separating from business logic 20
  - View layer 74
  - ViewModel layer 75
- Mouse Extensions 254
- MSIX community
- about 370
  - reference link 370
- MSIX labs
- about 370
  - reference link 370
- MSIX package management
- WinGet, using for 379-381
- MSIX packages
- about 211, 367-369
  - adding, to Microsoft Package Manager
  - community repository 376-379
  - basics 366
  - contents 368, 369
  - creating, for sideloading WinUI applications 390, 391
  - sideloading 392-394
  - WinUI applications, sideloading with 390
- MSIX packaging tool
- about 370
  - reference link 370
- MSIX resources
- reviewing 370
- MSIX SDK
- reference link 367
- MSIX toolkit
- about 370
  - reference link 370
- MSIX tools
- reviewing 370
- MSIX videos
- about 370
  - reference link 370
- MVVMCross
- about 77, 86
  - reference link 77
- MVVM framework, for WinUI
- MVVMCross 77
  - Prism library 76
  - selecting 77
  - WCT 76
- MVVM libraries
- for WinUI application 76
- My Media Collection application
- building 38

ComboBox filter, creating 62-68  
 ListView header, adding 61, 62  
 new item button, adding 68-70

## N

navigation service  
 creating 106, 108  
 NavigationView 126, 127  
 NuGet package 76

## O

object injection, ways  
 method injection 96  
 property injection 96  
 constructor injection 96  
 Object Relational Mappers (ORMs) 158  
 ObservableCollection<T>  
 working with 302, 303  
 Open source software (OSS) 21

## P

package  
 uploading, to Microsoft  
 Store 386, 388, 389  
 packaged desktop applications  
 execution, on Windows  
 reference link 212  
 page navigation, with MVVM and DI  
 about 102  
 data service, creating 109-112  
 interfaces and services, adding 106  
 ItemDetailsPage, adding 102-104, 106  
 ItemDetailsViewModel class,  
 creating 116-120

maintainability, increasing by  
 consuming services 113-115  
 navigation service, creating 106, 108  
 parameters, handling in  
 ItemDetailsPage 116

Paint.NET  
 URL 202  
 parallax scrolling 128  
 Patterns & Practices 76  
 Presentation Model pattern 74  
 Prism library 76  
 Progressive Web Application (PWA)  
 reference link 319  
 Project Reunion, and WinUI 3.0  
 about 28  
 versions 28  
 PSD Plugin, for Paint.NET  
 URL 202  
 pull request (PR) 377

## R

Rapid XAML Toolkit  
 analyzers 300, 301  
 URL 299  
 Razor syntax 317  
 React Native for Windows  
 reference link 8  
 Release To Manufacturing (RTM) 27  
 remote WinUI applications  
 debugging 295-297  
 resources  
 designing, for Fluent Design 201  
 responsive layouts, with XAML  
 reference link 184  
 row sizing  
 reference link 52

## S

ScrollViewer control 132, 133  
search pattern  
    reference link 183  
Single-Page Application (SPA) 330  
Sketch tool  
    about 201  
    URL 201  
SplitButton control  
    using 140, 141  
SQLite  
    about 154  
    adding, to DataService 154-158  
    reference link 154  
SQLite data store  
    creating 154  
static code analysis  
    XAML, improving with 299-301  
String Extensions 255  
style, Fluent Design aspects  
    acrylic 185  
    color 185  
    corner radius 185  
    icons 185  
    reveal focus 185  
    reveal highlight 185  
    spacing 185  
    typography 185

## T

TabView 232  
target framework metadata (TFM) 205  
TeachingTip control  
    adding, to save button 142-144  
toolkit releases  
    features 232

reviewing 231, 232  
toolkits  
    designing, for Fluent Design 201  
TwoPaneView control 129, 130  
type ramp  
    reference link 185

## U

Uniform Resource Identifiers (URIs) 214  
unit test framework  
    selecting 92  
unit test framework, for .NET developers  
    MS Test 92  
    NUnit 92  
    xUnit 92  
Universal Windows Platform  
    (UWP) 57-59, 76, 368  
    versus WinUI 29  
Uno Platform 30  
URI activation, handling  
    reference link 214  
User Account Control (UAC) 29  
user experience (UX) 20  
user interface (UI)  
    about 370  
    styling, with XAML 18-20  
UWP application development  
    about 7, 8  
    app restrictions, lifting 9  
    language, selecting for 8  
UWP applications  
    backward compatibility, with  
        Windows versions 10  
    reference link 60  
UWP app model 59, 60  
UWP community Discord channel  
    reference link 234

UWP Community Toolkit 231

UWP MapControl

using, in WPF 273-278

UWP project

properties, reviewing 45, 46

references, reviewing 45

WinUI anatomy 42

UWP Resources Gallery

using 199-201

## V

view data

updating, with `INotifyPropertyChanged` interface 80, 81

View layer 74

ViewModel classes

Dependency Injection (DI),  
using with 97-100

ViewModel layer 75

ViewState 316

virtual machine (VM) 381

Visual Studio

application packaging 371-374, 376  
installing 35, 36

tooling 134

WinUI app templates, adding for 36, 37

Visual Studio 2019

download link 35

Visual Studio App Center

application, deploying with 354  
builds, setting up 349-351

crash reports, analyzing 361-363

crash reports, obtaining 361-363

integrating, with GitHub

repository 351, 352, 354

working with 342

Visual Studio App Center account

creating 343-347

Visual Studio App Center account,  
organization roles

admin 345

collaborator 345

member 345

Visual Studio App Center application

creating 347-349

Visual Studio Code (VS Code)

reference link 321

Visual Studio Preview version

installation link 35

Visual Studio Preview version installation,

for WinUI development

workloads 35

Visual Tree 149

## W

WACK test 385

WCT packages

reference link 244

WebAssembly

and client-side .NET development 319

Web Forms 316

WebView 135, 136

WebView2 browser control

using, in WinForms 281, 282

WebView2 control

adding, in WebViewBrowser

desktop project 209-211

WebViewBrowser desktop project

structure, exploring 208, 209

WebView2 control, adding 209-211

WebViewBrowser (Package) project

exploring 211, 212

- visual assets, in Manifest Designer 213, 214
- WebViewCompatible browser control using, in WPF 278-281
- Windows 8 XAML applications
  - about 5
  - UI design 6
- Windows 10 7, 8
- Windows application life cycle events exploring 148, 149
- Windows Application Packaging Project 211
- Windows applications
  - Fluent Design, exploring for 181
- Windows Community Toolkit, controls using 240
- Windows Community Toolkit Sample App
  - about 232
  - controls 234-238
  - installing 233, 234
  - reference link 232
  - WPF and WinForms controls 238-240
- Windows Community Toolkit (WCT)
  - about 76, 230
  - extensions 254, 255
  - helpers 250-252
  - MVVM library 254
  - origin 231
  - reference link 230
  - services 252-254
- Windows development workloads installing 35, 36
- Windows Forms (WinForms)
  - advantages 31
  - WebView2 browser control, using 281, 282
- Windows Installer (MSI) 367
- Windows Package Manager
  - applications, deploying with 376
- Windows Package Manager
  - community repository
  - about 376
  - MSIX packages, adding to 376-379
- Windows PowerToys
  - reference link 381
- Windows Presentation Foundation (WPF)
  - about 11, 57, 74
  - advantages 30
  - UWP MapControl, using 273-278
  - versus WinUI 29
- WebViewCompatible browser control, using 278-281
- Windows RT 6
- Windows Runtime (WinRT) 6, 39
- WindowsXamlHost 232
- WinForms application, modernizing with XAML Islands
  - about 261
  - shared class library project, creating 261-266
  - WinForms host project, creating 266-270
- WinGet
  - using, for MSIX package management 379-381
- WinUI
  - about 21, 57-59
  - advantages 29, 31
  - creating, in Desktop project 241-243
  - first release 22, 23
  - versions 23
  - versus UWP 29
  - versus Windows Forms (WinForms) 31
  - versus WPF 29
- WinUI 2.1 24

- WinUI 2.2 24
- WinUI 2.2, enhancements
  - about 24
  - NavigationView 24
  - Visual Styles 25
- WinUI 2.3
  - about 25
  - new controls 25
- WinUI 2.4 25, 26
- WinUI 2.5 26
- WinUI 3.0
  - about 26, 28
  - backward compatibility 134
  - input validation 134, 135
  - new features 28
  - reviewing 133
  - Visual Studio tooling 134
  - WebView 135, 136
- WinUI 3.0 issues
  - reference link 82
- WinUI anatomy, in UWP project
  - about 42
  - App.xaml.cs, reviewing 43
  - App.xaml, reviewing 42, 43
  - MainPage.xaml.cs, reviewing 45
  - MainPage.xaml, reviewing 44
- WinUI applications
  - about 38
  - debugging 289
  - features, reviewing 38
  - Fluent Design, incorporating 186
  - life cycle events 149
  - MVVM libraries 76
  - MVVM pattern, implementing in 82-85
  - My Media Collection
    - application, building 38
  - sideloading, with MSIX
- WinUI app templates
  - adding, for Visual Studio 36, 37
- WinUI, controls
  - about 124, 125
  - AnimatedVisualPlayer control 125
  - NavigationView 126, 127
  - ParallaxView control 128
  - rating controls 129
  - TwoPaneView control 129, 130
  - working with 60
- WinUI, control library
  - creating 223-227
- WinUI desktop project
  - .NET 5 libraries, referencing
    - from 215-218
- WinUI, events
  - working with 60
- WinUI in Desktop project
  - about 205, 206
  - creating 206, 207
- WinUI, in UWP
  - versus WinUI in Desktop projects 38
- WinUI, patterns
  - form 183
  - master/details 183
  - search 183
- WinUI project
  - creating 39, 41
  - creating, with .NET 5 204
- WinUI properties
  - working with 60
- WinUI WebView2
  - Blazor application, hosting 338, 339
- WPF application
  - modernizing, with XAML Islands 270, 272, 273
- .NET 5 libraries, sharing

with 218, 220-222  
WPF Toolkit  
reference link 231  
wrapped controls 259

## X

x:Bind  
leveraging, with events 100, 101  
x:Bind markup extension  
about 79  
properties 80  
reference link 80  
XamarinCommunityToolkit  
reference link 231  
XAML Binding Failures window  
using 304, 305  
XAML C# Edit & Continue 307  
XAML Controls Gallery Windows app  
exploring 130, 131  
XAML Designer  
about 290  
used, for debugging local WinUI  
applications 290-292  
XamlDirect APIs  
exploring, for middleware  
authors 137-139  
XAML Hot Reload  
about 306  
coding with 307  
XAML Islands  
about 259, 260  
architecture 260  
WinForms application,  
modernizing with 261  
WPF application, modernizing  
with 270, 272, 273  
XAML layout, mistakes

about 297  
grid layout issues 297  
style, applying problem 298  
XAML Styler  
reference link 56  
XAML tools  
reference link 306  
Xceed Software 231  
xUnit  
reference link 92

