



Windows 10 Development with XAML and C# 7

Second Edition

Jesse Liberty
Jon Galloway
Philip Japikse
Jonathan Hartwell

Apress®

www.allitebooks.com

Windows 10 Development with XAML and C# 7

Second Edition

Jesse Liberty

Jon Galloway

Philip Japikse

Jonathan Hartwell

Apress®

Windows 10 Development with XAML and C# 7

Jesse Liberty
Acton, Massachusetts, USA

Jon Galloway
Spring Valley, California, USA

Philip Japikse
West Chester, Ohio, USA

Jonathan Hartwell
Plainfield, Illinois, USA

ISBN-13 (pbk): 978-1-4842-2933-0
<https://doi.org/10.1007/978-1-4842-2934-7>

ISBN-13 (electronic): 978-1-4842-2934-7

Library of Congress Control Number: 2017962077

Copyright © 2018 by Jesse Liberty, Jon Galloway, Philip Japikse and Jonathan Hartwell

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Cover image by Freepik (www.freepik.com)

Managing Director: Welmoed Spahr
Editorial Director: Todd Green
Acquisitions Editor: Steve Anglin
Development Editor: Matthew Moodie
Technical Reviewer: Fabio Claudio Ferracchiati
Coordinating Editor: Mark Powers
Copy Editor: Kezia Endsley

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit <http://www.apress.com/rights-permissions>.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at <http://www.apress.com/bulk-sales>.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484229330. For more detailed information, please visit <http://www.apress.com/source-code>.

Printed on acid-free paper

To my loving wife, for all of her support over the years.

Table of Contents

- About the Authors.....xi**
- About the Technical Reviewerxiii**
- Chapter 1: Getting Started 1**
 - Background..... 1
 - Versionless Windows 10..... 2
 - The Microsoft Store 2
 - Windows Design Guidelines..... 2
 - Being Fast and Fluid 3
 - Sizing Beautifully 4
 - Using the Right Contracts 4
 - Investing in a Great Tile 4
 - Feeling like It Is Connected and Alive..... 4
 - Roaming to the Cloud 5
 - UX Guidelines 5
 - Tooling..... 5
 - Visual Studio 2017 5
 - Blend for Visual Studio 2017 14
 - Git..... 18
 - Using Git in Visual Studio..... 19
 - NuGet 24
 - Installing Your First Package 25
 - Summary..... 27

TABLE OF CONTENTS

Chapter 2: Building Your First Windows 10 UWP App29

- Creating Your First App..... 29
 - App Project Overview 30
 - App.xaml..... 39
 - MainPage.xaml 39
- Model View ViewModel (MVVM)..... 40
 - The Pattern 40
 - Creating a Model 41
 - Creating the ViewModel 47
 - Updating the Code-Behind 49
 - Creating the View 49
 - Testing the App..... 51
 - Guidance..... 52
- Navigation 52
 - Creating a New Page 53
 - Adding Navigation to the MainPage 55
 - Handling the NavigatedTo Event..... 58
 - The Back Button 58
- Summary..... 59

Chapter 3: Themes, Panels, and Controls61

- Choosing a Theme 61
- Using Panels 64
 - The Canvas Control..... 65
 - The Grid Control..... 69
 - The StackPanel Control 74
 - The RelativePanel Control 76
 - The Border Control 77

Working with Controls	78
TextBlock and TextBox.....	79
Spell Check.....	82
Headers and Watermarks Controls.....	86
The PasswordBox Control.....	89
Buttons and Event Handlers Controls	90
The CheckBoxes, ToggleSwitches, and RadioButtons Controls	95
The ListBox, ListView, and ComboBox Controls.....	98
The Image Control	101
The Slider Control	104
The ProgressBar Control.....	104
The ToolTip Control	106
The DatePickers and TimePickers Controls.....	106
Flyouts	108
Understanding Dependency Properties.....	111
Data Hiding	111
Dependency Properties	113
Summary.....	117
Chapter 4: Binding	119
DataContext.....	120
Creating a Simple Binding	120
Data-Binding Statements.....	123
Binding Errors	125
FallbackValue	126
TargetNullValue.....	127
Binding to Elements.....	127
Binding Modes	130

TABLE OF CONTENTS

UpdateSourceTrigger	131
INotifyPropertyChanged	132
Binding to Collections	136
Creating the Collection	136
Creating a Data-Bound ListView	141
INotifyCollectionChanged	143
Data Converters	144
Summary.....	149
Chapter 5: Local Data	151
Application Data.....	151
Settings Containers	152
Saving, Reading, and Deleting Local Data	152
Creating the Data Layer	158
Creating the Repository Interface.....	159
Creating the DataModel.....	160
Creating the ViewModel	161
Local Data	166
Using JSON to Format Data.....	166
Local Data Containers	167
Creating the File Repository	167
Creating the View	173
Roaming	179
User-Specified Locations	180
Creating the FileOperations Class	180
Adding the File Association for JSON Files.....	184
SQLite.....	187
Summary.....	195

Chapter 6: Application Lifecycle	197
The Three Application States	198
Running	199
Suspended.....	199
Terminated.....	199
State Transitions	200
Launching.....	201
Activating.....	201
Suspending.....	201
Resuming	201
Terminating.....	201
Killing.....	202
Managing the Lifecycle	202
Building the Sample App.....	203
Adding the Navigation Parameter Class	204
Creating the Details Page	205
Creating the Main Page	207
Handling Adding/Editing List Items.....	209
Responding to App Suspension	210
The OnSuspending Event.....	210
The EnteredBackground Event	211
Responding to App Activation	216
Determining the Previous Application State	216
Testing the Restoring Navigation State	216
Testing the Restoring Session Information.....	217
Summary.....	218
Index	221

About the Authors

Jesse Liberty is a master consultant for Falafel Software, a Microsoft MVP, a Telerik MVP, and an author. He creates courses for Pluralsight and hosts the popular *Yet Another Podcast*. His blog is also considered required reading. He was a senior evangelist for Microsoft, a XAML evangelist for Telerik, a Distinguished Software Engineer at AT&T, Software Architect for PBS, and Vice President of Information Technology at Citibank. Jesse can be followed on Twitter at @JesseLiberty.



Philip Japikse is an international speaker, Microsoft MVP, ASPInsider, MCSD, CSM, and CSP, and a passionate member of the developer community. He has been working with .NET since the first betas, developing software for over 30 years, and heavily involved in the Agile community since 2005. Phil is the lead director for the Cincinnati .NET User's Group (<http://www.cinnug.org>), founded the Cincinnati Day of Agile (<http://www.dayofagile.org>), and volunteers for the National Ski Patrol. Phil is also a published author with LinkedIn Learning (<https://www.lynda.com/Phil-Japikse/7908546-1.html>). During the day, Phil works as an enterprise consultant and Agile coach for large to medium firms throughout the United States. Phil enjoys to learn new tech and is always striving to improve his craft. You can follow Phil on Twitter via <http://www.twitter.com/skimedic> and read his blog at <http://www.skimedic.com/blog>.

ABOUT THE AUTHORS

Jon Galloway is a senior technical evangelist for Microsoft. He's been developing applications on Microsoft technologies for 17 years on the desktop and the web, from scrappy startups to large financial companies. Along the way, he's contributed to several open source projects, started the Herding Code podcast, and helped build some of the top keynote demo apps for Microsoft conferences over the past five years. He travels worldwide, speaking at conferences, Microsoft Web Camps, and developer events. Jon tweets as @jongalloway.

Jonathan Hartwell has worked professionally with C# for five years and spent four of those years handling XML. He received his master's degree in computer science from DePaul University and has an affinity for programming languages. When not programming, he is either watching Arsenal play or spending time with his wife and their two dogs. He is the founder of Voltaire Software LLC, which creates software that helps developers be more productive.

About the Technical Reviewer

Fabio Claudio Ferracchiati is a senior consultant and a senior analyst/developer using Microsoft technologies. He works for BluArancio (www.bluarancio.com). He is a Microsoft Certified Solution Developer for .NET, a Microsoft Certified Application Developer for .NET, a Microsoft Certified Professional, and a prolific author and technical reviewer. Over the past 10 years, he's written articles for Italian and international magazines and co-authored more than 10 books on a variety of computer topics.

CHAPTER 1

Getting Started

Windows 10 Universal Windows Platform (UWP) development with C# and XAML carries a lot of similarities with developing Windows Presentation Foundation (WPF) applications. Well, they both use XAML and C#. Many of your existing skills with user interfaces (UIs) and program code can be leveraged for Windows 10 UWP apps. But there are a lot of differences, as well. The Universal Windows Platform is the major difference between UWP applications and WPF applications. An application written on the UWP can run on whatever hardware implements that functionality, which includes Xbox, Surface, PC, mobile, and even the HoloLens. Apps are deployed through a central store (as opposed to click-once deployment or Microsoft Installer packages).

Not a WPF developer? No worries! This book will take you through everything you need to know to build Windows 10 UWP apps.

Background

Microsoft released the latest revision of its Windows operating system, Windows 10, on July 29th, 2015. The release of the Universal Windows Platform is a continuation on the idea of writing code on one platform that could run on all supported devices. In order to get their users on the new operating system, Microsoft offered a free upgrade to Windows 10 to users who have a valid copy of Windows 7 or 8.1 and are not running an Enterprise edition.

Versionless Windows 10

With Microsoft's policy change to auto update whenever there are updates available, Microsoft is moving away from Windows versions such as Windows 8.1 and moving toward build versions. Microsoft has announced that they will be doing feature updates semi-annually every March and September. The goal is to be able to have everybody on the same build, or as close to each other, as possible. If Microsoft is able to pull that off, it will be a major benefit for all Windows developers as it will allow us to better focus on supporting new features rather than worrying about supporting obsolete versions of Windows.

The Microsoft Store

How many times have you had to do tech support for a family member because he clicked on some random pop-up on the Internet or installed some software that a friend told him about? The main mechanism for getting apps is from the Microsoft Store. Having that one central place to get apps for Windows 10 helps prevent rogue software from getting installed, increasing the security and reliability of the device. It also provides a centralized location for developers to place their app for others to find. For more information about submitting your app to the Microsoft Store, see Chapter [12](#).

Windows Design Guidelines

In order to get your apps accepted into the store, you must make sure they meet the seven traits of a great app and follow the five Microsoft design principles. There are additional technical requirements that will be discussed in Chapter [11](#).

Let's look at the seven traits of a great app first. To achieve greatness, it must:

- Be fast and fluid
- Size beautifully
- Use the right contracts
- Invest in a great tile
- Feel like it is connected and alive
- Roam to the cloud
- Embrace modern app design principles

Being Fast and Fluid

Modern apps can run on a variety of devices with a wide range of capabilities. While Microsoft has set minimum standards for all hardware that carries the Windows 10 logo, it's important for the success of your app (as well as the success of Windows 10) that your app doesn't perform poorly or cause the hardware to perform poorly. You will see as you work your way through this book that in order to develop Windows 10 UWP applications, you must use asynchronous programming to ensure a responsive UI. Additionally, the very design of the Windows 10 UWP process lifetime management cycle ensures that background apps don't drain the battery or use up precious system resources.

Use the `async` pattern liberally. If your app is taking a long time to load or to run, people will uninstall it. Or, worse yet, they will write a scathing review and *then* uninstall it.

Sizing Beautifully

Windows 10 devices come in a variety of sizes and screen resolutions. Apps can be run in a landscape or portrait view as well as resized to share the screen with other apps. Your app needs to be able to adjust to different layouts and sizes, not only in appearance but also in usability. For example, if you have a screen showing a lot of data in a grid, when your app gets pinned to one side or the other, that grid should turn into a list.

Using the Right Contracts

Windows 10 introduces a completely new way to interact with the operating system and other applications. Contracts include Search, Share, and Settings. By leveraging these contracts, you expose additional capabilities into your app in a manner that is very familiar to your users.

Investing in a Great Tile

Tiles are the entry point into your applications. A live tile can draw users into an app and increase the interest and time spent using it. Too many updates can lead them to turn off updates, or worse yet, uninstall your app.

Secondary tiles are a great way for users to pin specific information to their Start screens to enable quick access to items of their interest.

Feeling like It Is Connected and Alive

Users are a vital component to Windows 10 UWP apps. It is important to make sure that your app is connected to the world so that it can receive real-time information. Whether that information is the latest stock prices or information on sales figures for your company, stale data doesn't compel users to keep using your app. They already know what yesterday's weather was. The current forecast is much more interesting.

Roaming to the Cloud

Windows 10 allows the user the capability to share data between devices. Not only can application settings be synced but so can the application data. Imagine the surprise for a user who enters some data into your app at work and then picks up another Windows 10 device at home, starts your app, and the data is right there.

It is important to leverage the cloud whenever possible to make transitioning from one device to another as seamless as possible.

UX Guidelines

There are many more guidelines suggested by Microsoft. For the full guidelines, see <http://msdn.microsoft.com/en-us/library/windows/apps/hh465424.aspx>.

Tooling

While you can certainly remain in Visual Studio the entire time you are developing your app, leveraging a combination of the available tooling provides the best experience. For developing Windows 10 UWP applications, the two main tools you will use are Visual Studio 2017 and Blend for Visual Studio 2017.

Visual Studio 2017

In March of 2017, Microsoft released Visual Studio 2017. Among the changes in the latest version of Visual Studio is support for C# 7. While it is possible to use Visual Studio 2015 in order to create Windows 10 UWP applications, the examples in this book may not work with that edition as we will be using new features only available in C# 7.

Versions

If you are reading this book, then you are probably very familiar with Visual Studio. In this section, I'll talk about the different versions of Visual Studio 2017 available to you and some of the differences between them. If you aren't very familiar with Visual Studio, don't worry. As we move through the chapters of this book, the relevant features will be discussed in greater detail.

Visual Studio Community Edition

Microsoft has removed the Express editions of Visual Studio and has moved to a Community Edition as the entry into Visual Studio. It is free and fully featured, but does come with licensing that allows use in only certain situations. Verify that you are following the license agreement by viewing the terms at <https://www.visualstudio.com/license-terms/mlt553321/>.

Visual Studio with MSDN

With an MSDN subscription there are two options for developers when it comes to Visual Studio 2017: Professional and Enterprise. Both versions allow you to create Windows 10 UWP apps. For a full comparison, you can check out Microsoft's comparison at <https://www.visualstudio.com/vs/compare/>.

The Windows 10 Simulator

All versions of Visual Studio come with the ability to run your Windows 10 UWP app in a simulator. This is essentially a remote desktop session to your PC with the added ability to change orientation, form factor, and gesture support and to simulate many factors of a tablet (even if you are developing on a non-touch device).

Creating Your First Windows 10 UWP App

To create a Windows 10 UWP app, create a new project in Visual Studio 2017 by selecting File ► New ► Project. In the left rail, you will see all of the installed templates for your Visual Studio installation (your mileage may vary based on the version you installed and the third-party products you use). Select Installed ► Templates ► Visual C# ► Windows Store, and you will be presented with the dialog shown in Figure 1-1.

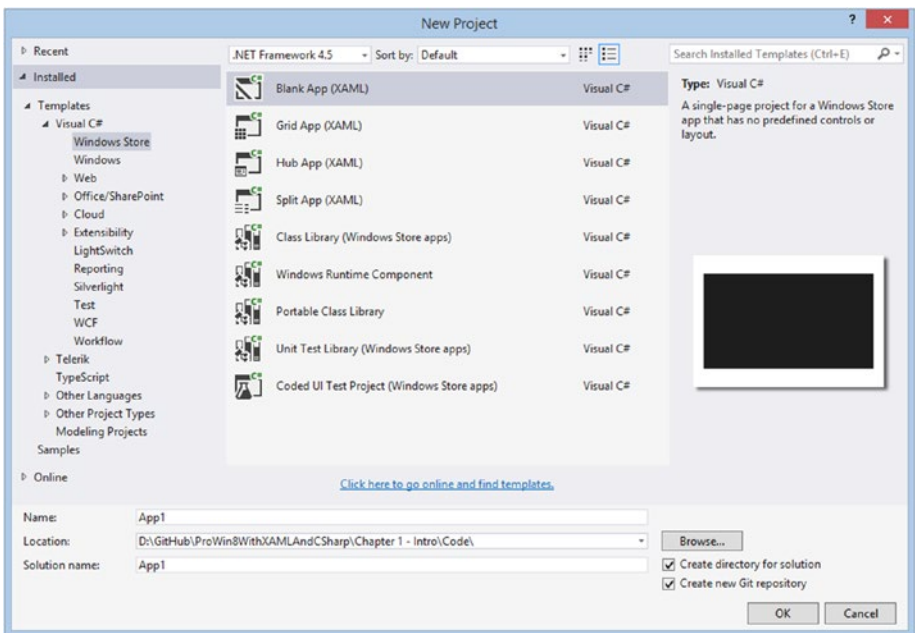


Figure 1-1. New Project templates for Windows 10 UWP apps

In Chapter 4, we will go into great detail for all of the project templates, so for now, select Blank App (XAML). In fact, this will be the starting template for *most* of our projects in this book, and is the template I typically start with when I develop Windows 10 UWP apps. You can leave the project name as the default App1.

After you create your project, take a look at the default solution folder (shown in Figure 1-2). The Blank template actually does a lot for us. In addition to creating the project and bringing in the appropriate references, it supplies several assets—the `App.xaml` file and `MainPage.xaml`. The `Assets` folder contains the images for the splash screen and the default tiles (more on that later in this book), and if you are familiar with WPF, the `App.xaml` and `MainPage.xaml` files should be very familiar. Again, we will spend a lot of time in the book on those files.

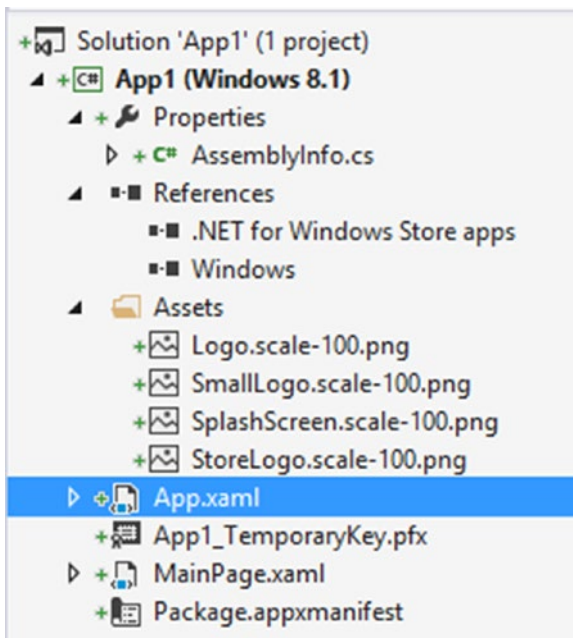


Figure 1-2. *Default Solution Explorer files*

To run the app, you have several options. You can press F5 (to start with debugging), Ctrl-F5 (to start without debugging), click on Debug in the menu (to be presented with the same options), or click the toolbar item with the green arrow (as shown in Figure 1-3).

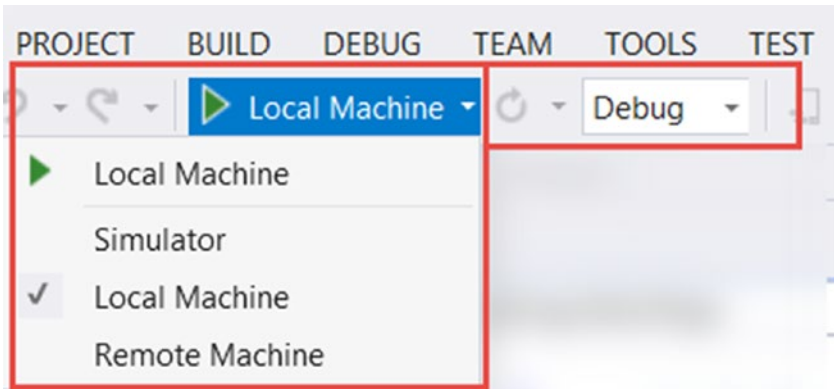


Figure 1-3. Run toolbar utility

By default, Visual Studio will run your app on the local machine in Debug configuration. Go ahead and click on the green arrow (or press F5) to run the app. We would expect to see a completely blank screen, but instead we are presented with some changing numbers (they change as you move the mouse around the screen) in the top corners of the screen, as shown in Figure 1-4. The frame rate counters show you, from left to right, the UI frame rate (frames per second), the App CPU usage of UI thread, the system composition frames per second, and the system UI thread CPU usage. If you run the app without debugging, you will not see the numbers in the corner. This is because all of the Visual Studio-supplied templates enable the frame rate counter display while running in debug mode.



Figure 1-4. Debugging with `FrameRateCounter`

Turning this off is very simple—you just open `App.xaml.cs`, and in the `OnLaunched` event handler, comment out this line of code:

```
this.DebugSettings.EnableFrameRateCounter = true;
```

so that it looks like this:

```
//this.DebugSettings.EnableFrameRateCounter = true;
```

Now, when you run your app in debug mode, the numbers are no longer displayed.

Adding a Basic Page

Even though I typically start with the Blank App template, I rarely keep the supplied `MainPage.xaml` (and its code-behind file `MainPage.xaml.cs`). Visual Studio provides a Basic Page file template that provides a lot of necessary functionality. Delete the `MainPage.xaml` (we will be replacing this) and right-click your project and select **Add ► New Item**. From the **Add New Item—App 1** dialog, select the Basic Page and name the page `MainPage.xaml`, as shown in Figure 1-5.

Note We call it `MainPage.xaml` so we don't have to change `App.xaml.cs`. If you want to call the files something else (or change the page that gets loaded when an app first starts), open `App.xaml.cs`, navigate to the end of the `OnLaunched` event handler and change the following line to the name of the page you added:

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

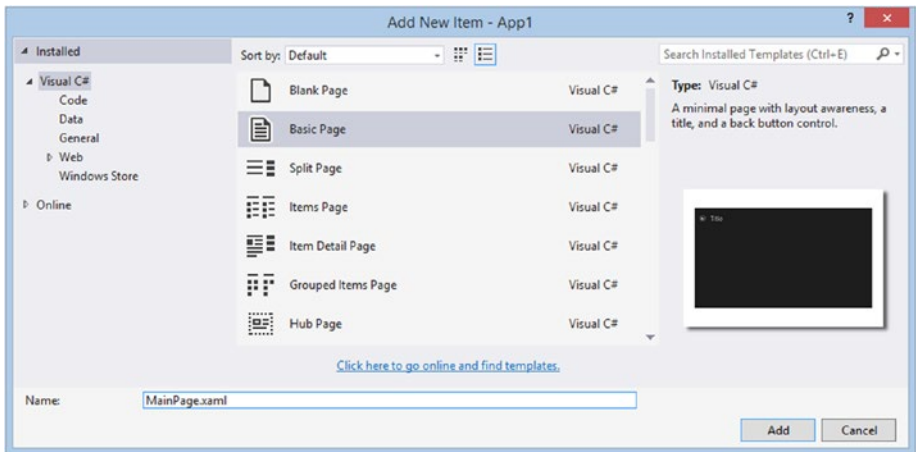


Figure 1-5. Adding a new Basic Page

When you add a new Basic Page, Visual Studio prompts you that it will add several files into your project. Say Yes! These are extremely helpful files and will be used extensively throughout the course of this book. However, for now, we just want to have some text to display. Change the option on the debug location toolbar to run in the simulator, and then press F5 (or click the green arrow to start debugging). You'll now see a title for the app (running in a window that resembles a tablet) and a series of controls on the right rail of the simulator, as shown in Figure 1-6.

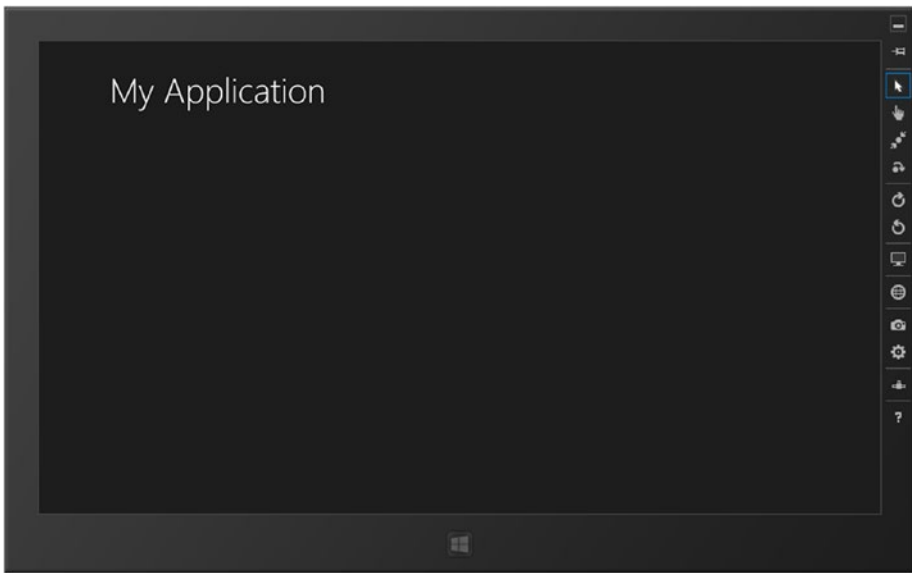


Figure 1-6. *The simulator*

The Simulator Controls

Most of the simulator controls are very self-explanatory, but I struggled in my early days of Windows 8 apps to remember what each icon stood for, so I've listed the explanations here to help you out.



Minimize the simulator



Always keep the simulator on top

The touch modes in the simulator are important to be able to test your app's responsiveness to touch if you don't own (or develop on) a touch device. The mouse mode button takes you back out of touch mode to keyboard and mouse mode.



Mouse mode



Basic touch mode, pinch/zoom touch mode,
and rotation touch mode

The rotation and resolution controls help testing by responding to different orientations and form factors.



Rotate clockwise (90 degrees)/rotate
counterclockwise (90 degrees)



Change the resolution

If you are building a location-aware application, you can test it by setting the location that is sent to the app from the hardware.



Set location

The screenshot commands are invaluable for the submission process, as you will see in Chapter 12. They are also useful to create screenshots for building documentation, advertising your app on your web site, and so on.



Copy screenshot/screenshot settings

The network control allows for testing occasionally connected scenarios, bandwidth usage, and making other networks variables.



Change network properties



Help

Blend for Visual Studio 2017

Expression Blend has long been a staple of the WPF developer. Long sold as a separate product from Visual Studio, it was part of the Expression suite. Starting with Visual Studio 2012, Blend for Visual Studio was released as a free companion application for Visual Studio. To open your project in Blend, you can right-click on any XAML file in Visual Studio 2017 and select Open in Blend. This will open not just the file that you selected, but also the entire project/solution.

Many of the features of Blend are covered in subsequent chapters, but some of the biggest benefits of using Blend are:

- Full control of your UI in a compact layout: The Visual Studio XAML designer pales in comparison to what can be accomplished in Blend. While I am not a designer (and don't make any claims to having design skills), Blend has enabled me to make much-better-looking UIs as well as to make changes much faster than in Visual Studio (regardless of being in design or XAML mode in Visual Studio).
- The ability to easily add animations, gradients, and styles to your app/page.
- The ability to quickly add states to your page (for layout updates) and state recording.

- The ability to view your page in many layouts and form factors (much like the simulator, but without the benefit of the page running—WinJS/HTML developers still have the advantage here).

Additionally, Visual Studio and Blend for Visual Studio keep your files in sync. If you have your project open in both, when you make changes (and save them) to your app/pages in one program, switching to the other program will prompt you to reload. Make sure that you actually save the changes, as making changes in both without saving will result in concurrency problems.

Opening Your Project in Blend for Visual Studio

Visual Studio and Blend work extremely well together. To open your project in Blend, right-click on the `MainPage.xaml` in your project and select `Open in Blend` (see Figure 1-7).

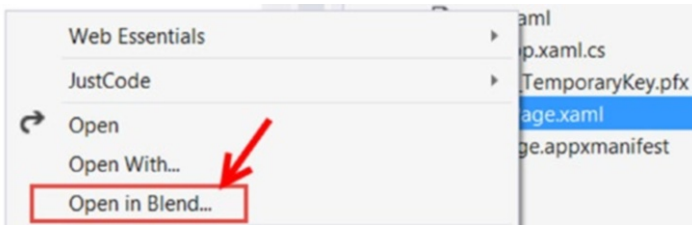


Figure 1-7. *Opening a file in Blend*

Visual Studio invokes Blend, opening your entire project (not just the file you clicked on). Once the file is opened, you will see a screen similar to Figure 1-8. Blend will open the file you right-clicked on in Visual Studio.

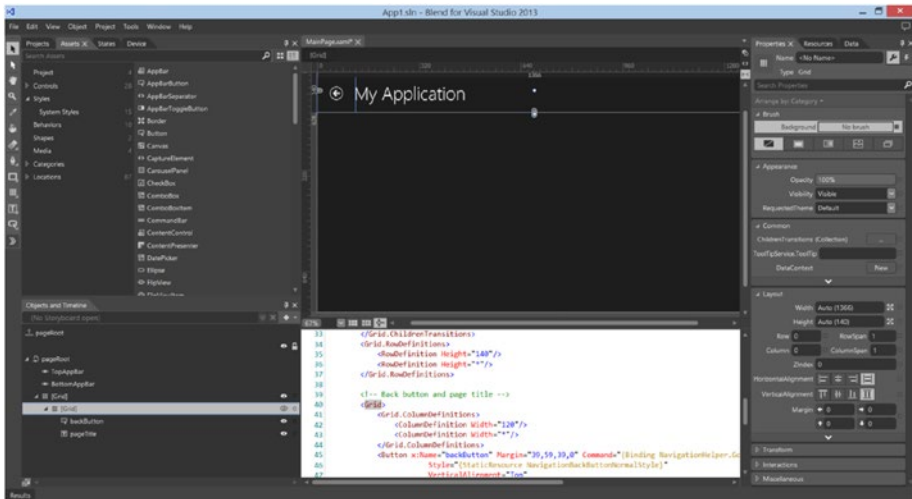


Figure 1-8. *MainPage.xaml* opened in Blend

That’s a lot of windows, but at least in the default layout (much like Visual Studio, you can change the layout to suit your needs). Let’s look at them in a little more detail.

Projects, Assets, States, and Data Tabs

The top-left corner of the window contains the Projects, Assets, States, and Data tabs, which allow you to do the following processes:

- The Projects tab shows all of the files in your solution (much like Solution Explorer in Visual Studio). Nothing too exciting to report here.
- The Assets tab lists all of the assets available to add to your page. Think of this as a turbo-charged Visual Studio Toolbox. In addition to controls and panels that you can add to your page, you can add (and modify) styles, behaviors, and media.

- The States tab allows you to add the Visual State Manager XAML as well as Visual State groups to your page. It also allows for easy addition of transitions for your visual states.
- The Data tab gives you access to the Data Context for the given XAML.

Objects and Timeline

The Objects and Timeline panel (lower left) provides the document outline as well as the ability to add and modify storyboards (to be used in conjunction with the Visual State Manager).

Page Designer, Markup, and Code

The center of the workspace is the designer and code editor. Just like in Visual Studio, you can have a split view, all design, or all markup. You can also load code files into the center pane. While you get features like IntelliSense, the development experience doesn't contain all of the great features of Visual Studio like navigation and refactoring. Plus, you lose any productivity plug-ins like Telerik's JustCode that you might have installed in Visual Studio.

Properties, Resources, and Data Tabs

The right rail of the workspace contains the Properties, Resources, and Data tabs, which can be described as follows:

- The Properties tab is where I spend a significant portion of my time in Blend. In addition to the simple items like Name and Layout properties like Width and Height, there are a host of properties that are difficult to set by hand in markup. Brushes, Transforms, and Interactions can all be set using the Properties panel.

- The Resources tab contains all of the application and page-level resources as well the option to edit and add more resources.
- The Data tab allows you to set the data context for your page, create sample data, and create different data sources. This is helpful to see what the page will look like with data at design time instead of always having to run the app.

Blend for Visual Studio is an extremely powerful tool and it would take an entire book to discuss all of the features. My development workflow involves keeping both Visual Studio and Blend open at the same time, and I switch back and forth depending on what I am trying to accomplish. Explore Blend and see what works best for you.

Git

Software version control has been around for a long time. If you have been in the Microsoft space for a significant length of time, you might remember Visual Source Safe. In the .NET world, the MS developer was left with Team Foundation Server (TFS) as the only integrated source-code-management (SCM) system.

TFS is a powerful application lifecycle management (ALM) tool (it includes project management, bug tracking, SCM, and other components). That is a lot of tooling when you are only looking for SCM. The SCM portion of TFS is Team Foundation Version Control (TFVC) and is a centralized SCM system. This means that a single repository is the source of record, and all developers check their code in and out of this single repository. Later versions of TFVC include the capability to shelve work and create branches, providing some isolation for work in progress.

Git, developed by Linus Torvalds in 2005 for the Linux kernel, is a distributed version control system (DVCS). This means that every developer using Git has a full-fledged repository on his local machine with complete history and tracking capabilities. Many Git users (especially in a team environment) have a central repository in addition to their local repository. This frees the developer to spike different ideas, work on features independent of the rest of the team, and check in rapidly as often as they like without worrying about network latency or affecting other team members.

Which SCM system you choose to use is completely up to you. They both have their merits (and there are many other SCM systems available to you as well that are very effective in what they do). It's more how you work and whom you work with that usually determines which system to use. So why do I bring up Git specifically in this book? Because if you are a single developer creating a Windows 8 app, Git is custom tailored to you, and with Visual Studio 2013 (and updated to Visual Studio 2012), Git support is now included.

There are entire books written about effectively using Git, so this is just a quick look into the Visual Studio integration, and not a treatise on DVCS.

Using Git in Visual Studio

One of the advantages of using Git is its simplicity. A Git repository can be created anywhere—on a local disk, network share, or web site (like GitHub).

GitHub for Windows

The easiest way to start working with Git if you are new to the system is to install GitHub for Windows, which is available from <https://windows.github.com/>. Creating a new repository is as easy as clicking on the Create button in GitHub for Windows. Once Visual Studio is configured to use Git, any projects created inside an existing repo will automatically tie into the Git repo.

Enabling Git in Visual Studio 2017

The first step to using Git with your project is to enable the Microsoft Git Provider. Do this by selecting **Tools** ► **Options** ► **Source Control** ► **Plug-in Selection**, and then select the Microsoft Git provider for the Current source control plug-in, as shown in Figure 1-9.

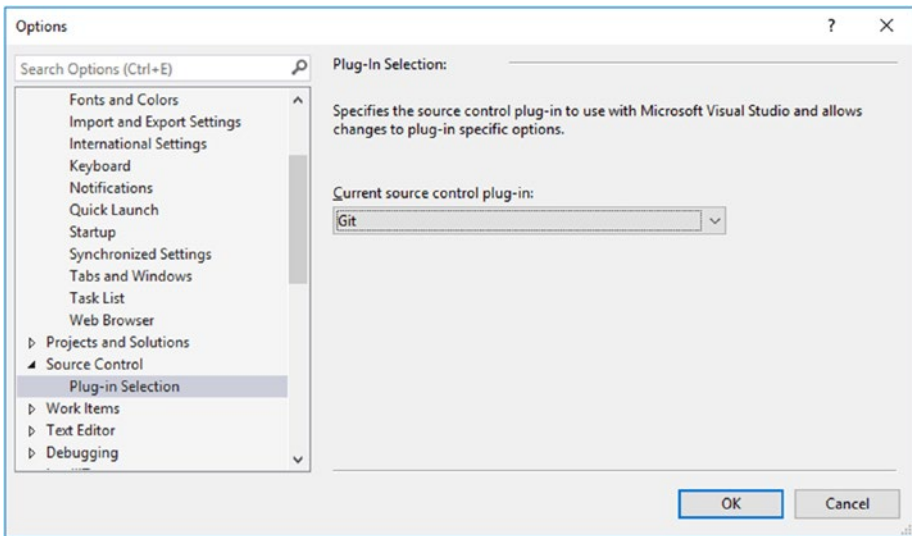


Figure 1-9. Selecting the Microsoft Git provider

Selecting Team Explorer (**View** ► **Team Explorer**) in the right rail of Visual Studio (the default location) allows you to manage your local Git repository. By default, VS 2017 creates the appropriate Git ignore files so local files such as `/bin` and `/obj` files, temp files, user files, and so forth don't appear in the repository. There are also attributes on how Git should handle conflicts in project files. To view both of these files, select Git Settings, as shown in Figure 1-10.

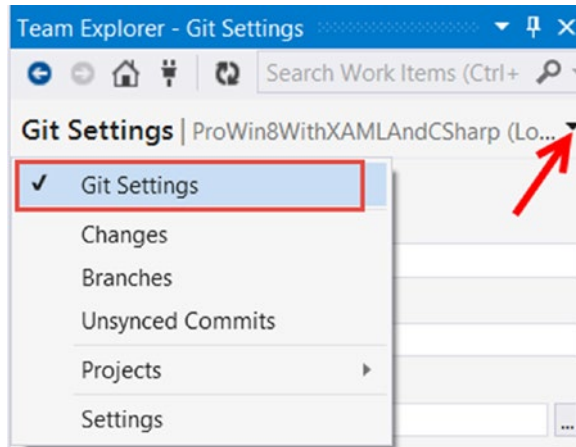


Figure 1-10. Accessing the Git repository settings

This is also where you enter your username and e-mail address as well as the default Git directory, as shown in Figure 1-11.

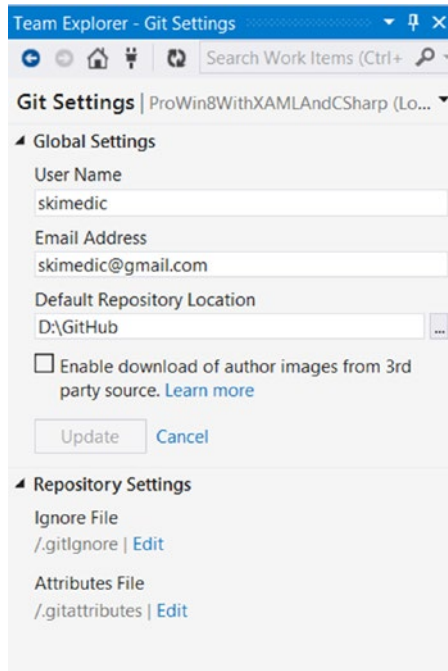


Figure 1-11. Git settings

Checking In Changes

To check in changes, select Changes from the same menu, as shown in Figure 1-10. You will see changes that will be included in this check-in, excluded changes, and untracked files. To commit the changes, enter a comment in the text box with the watermark “Enter a commit message <Required>” and click on Commit. You can also choose Commit and Push to share your changes with a remote repository or choose Commit and Sync to share your changes and get the latest version from the remote repository, as shown in Figure 1-12.

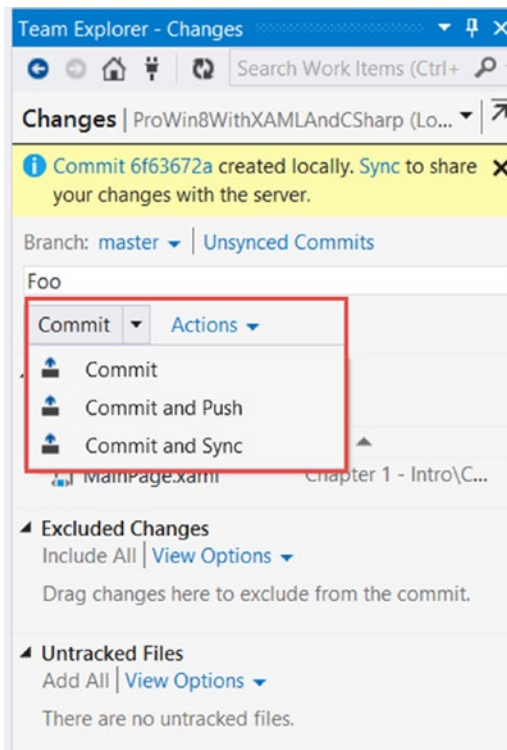


Figure 1-12. Committing changes to the local repository

Remote Repositories

There are many places where you can host remote Git repositories, with the most popular being GitHub (www.github.com). Once you set up a remote repository, you can point your project to it by entering its URL, as shown in Figure 1-13.

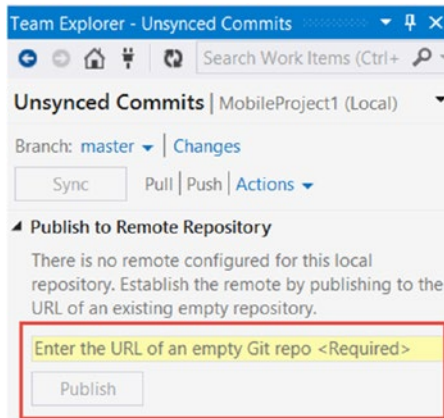


Figure 1-13. Publishing to a remote repository

Reverting Changes

If you totally mess up while developing, Git makes it very easy to restore from the repository. Right-click on your file in Solution Explorer and you will see the Git features exposed—Undo, View History, Compare with Unmodified, and Commit (see Figure 1-14). Undo does just what it says—it throws away your changes and restores the file from the repository. It's like your own personal security blanket!

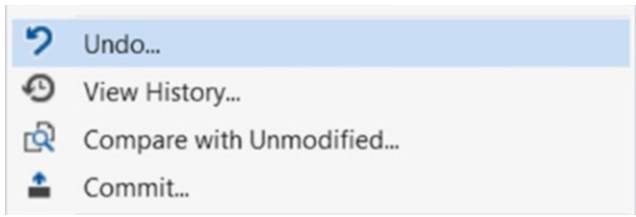


Figure 1-14. *Git functions exposed through Solution Explorer*

Again, this isn't a full explanation of how Git works but a quick overview of the Visual Studio features that support Git. If you've never used source code control systems, Git is an easy first one to use. You'll thank yourself in the end.

NuGet

From the official NuGet site (www.nuget.org): "NuGet is the package manager for the Microsoft development platform including .NET. The NuGet client tools provide the ability to produce and consume packages. The NuGet Gallery is the central package repository used by all package authors and consumers."

Instead of scouring the web for tools to add to Visual Studio, you can use NuGet as a single source to get a wide variety of add-ins for your solution. Rather than installing the tools on your development machine, the packages are installed at the *solution* level. This permits different versions to coexist on the same developer machine.

Another very large advantage to NuGet is that each package lists its dependencies in its package manifest. When a package is installed through NuGet, all of its dependencies are installed as well.

Yet another benefit of NuGet is the ability to create private NuGet package sources. To change the source, select Tools ► Options ► NuGet Package Manager ► Package Sources, as shown in Figure 1-15.

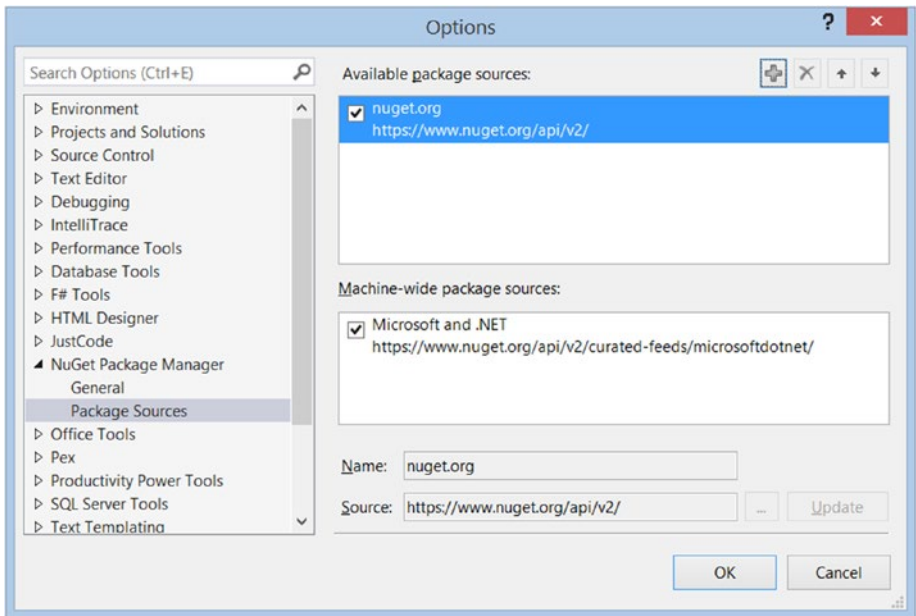


Figure 1-15. NuGet Package Source dialog

Installing Your First Package

One of the “can’t live without” packages for developing Windows 10 UWP apps is Newtonsoft’s Json.NET. We’ll use Json.NET later in this book, but for now, let’s just install it. There are two ways to install packages—by using the Package Manager Console command line or by using the Package Manager GUI.

Installing from the Command Line

Access the Package Manager Console by selecting View ► Other Windows ► Package Manager Console if it isn’t currently visible in the bottom rail of Visual Studio.

Type **install-package newtonsoft.json** and you'll see the dialog shown in Figure 1-16. At the time of this writing, 10.0.2 is the current version. NuGet will install the current version unless you specify a version. Another benefit of using NuGet.

```

PM> install-package newtonsoft.json
Restoring packages for C:\Apress\Windows10\update\source\Chapter 2\App1\App1.csproj...
GET https://api.nuget.org/v3-flatcontainer/newtonsoft.json/index.json
OK https://api.nuget.org/v3-flatcontainer/newtonsoft.json/index.json 35ms
GET https://api.nuget.org/v3-flatcontainer/newtonsoft.json/10.0.2/newtonsoft.json.10.0.2.nupkg
OK https://api.nuget.org/v3-flatcontainer/newtonsoft.json/10.0.2/newtonsoft.json.10.0.2.nupkg 16ms
GET https://api.nuget.org/v3-flatcontainer/system.runtime.windowsruntime/index.json
OK https://api.nuget.org/v3-flatcontainer/system.runtime.windowsruntime/index.json 37ms

```

Figure 1-16. Command-line installation of *Json.NET*

Installing from the Graphical User Interface GUI

Installing from the GUI is very simple and provides a search mechanism if you don't know the exact name of the package that you are looking for. For example, everyone refers to the package as "Json.NET." The actual package name in NuGet is `newtonsoft.json`. This is a great example of where the search in the NuGet GUI is very helpful.

To access the GUI, right-click on your solution and select **Manage NuGet Packages for Solution**, as shown in Figure 1-17.

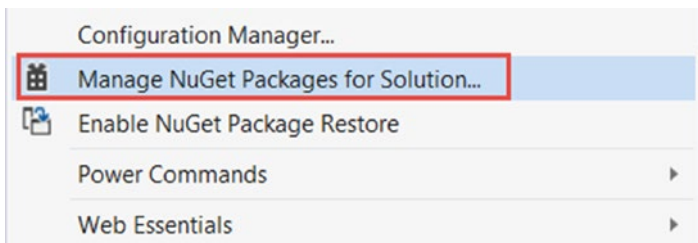


Figure 1-17. Launching the *NuGet GUI*

Select **Online** in the left rail and enter **Json.NET** in the search dialog. You will see results similar to Figure 1-18. Merely click **Install** to install `Json.NET`.

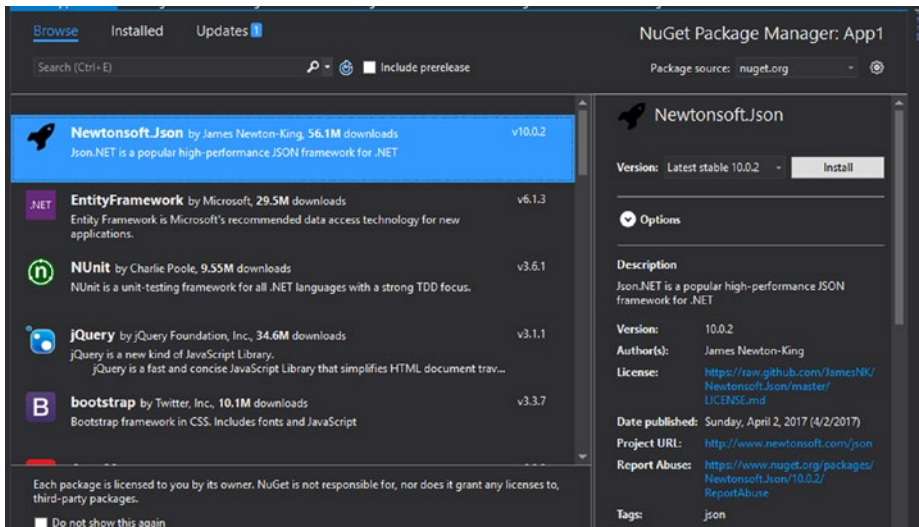


Figure 1-18. *Installing Json.NET with the Package Manager GUI*

Summary

Windows 10 UWP apps represent a very large paradigm shift from traditional Windows desktop applications (such as WPF or WinForm) or web apps (such as ASP.NET WebForms or MVC). Whereas traditional applications were developed with a wide range of tools but no real design guidelines and no expectations of performance, Windows 10 UWP apps must meet a series of expectations, both in terms of UI design and app performance. They are distributed through the Microsoft Store after a stringent certification process.

Developing Windows 10 UWP apps involves a lot more than just Visual Studio. Blend for Visual Studio helps build compelling UIs, Git provides security for your source code, and NuGet enables easy addition of packages and add-ons to Visual Studio.

Now that you know the tools to use, let's build that first app!

CHAPTER 2

Building Your First Windows 10 UWP App

Chapter 1 covered the design guidelines as well as the tooling commonly used to build Windows 10 UWP apps. In this chapter, we cover some of the core principles of Windows 10, including its architecture, all of the many parts of its apps in Visual Studio, the Model-View-ViewModel pattern, and navigation. All in the context of building your first Windows 10 UWP app.

Creating Your First App

To create your first app, start Visual Studio and select File ► New ► New Project. Then select Templates ► Visual C# ► Windows Universal. We'll talk about the different templates in later chapters, but for now just select the Blank App and leave the default name as App1, as shown in Figure 2-1.

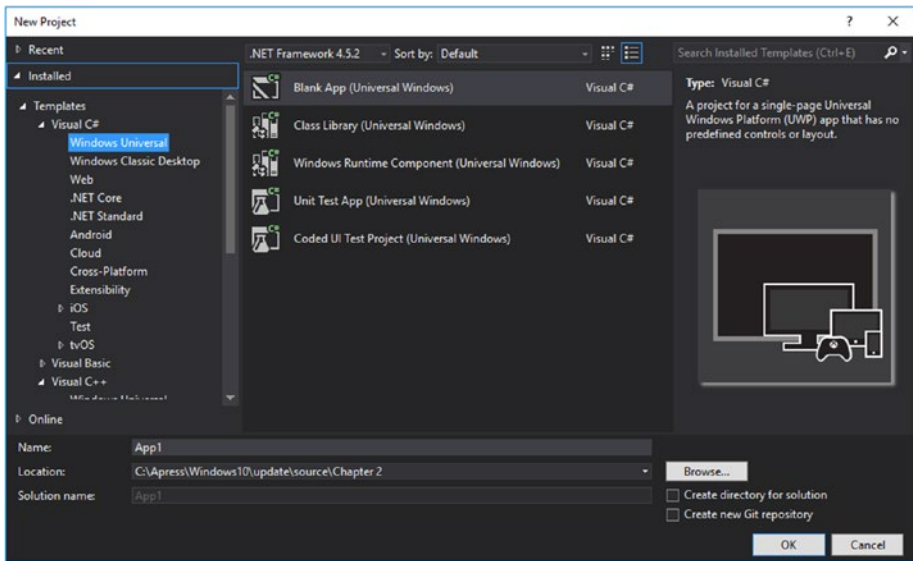


Figure 2-1. *Creating a new Windows 10 UWP app*

Let’s look at the nodes and files that are created as part of the template. Much of the project should be familiar to you.

App Project Overview

When you create a new project, the first thing you will be greeted with is a pop-up asking for the minimum version of Windows 10 required and the recommended version of Windows 10 for this application, as you can see in Figure 2-2. This gives you greater control over what is available to your application as you can ensure that your users have APIs or other features that may be required in your application. For now, we will just use the defaulted selections.

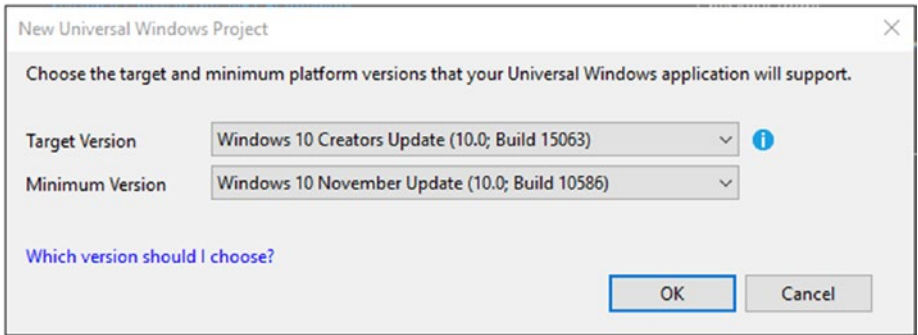


Figure 2-2. *Picking the target and minimum version of Windows 10 for your application*

The New Project template introduces a significant number of folders and files, as shown in Figure 2-3.

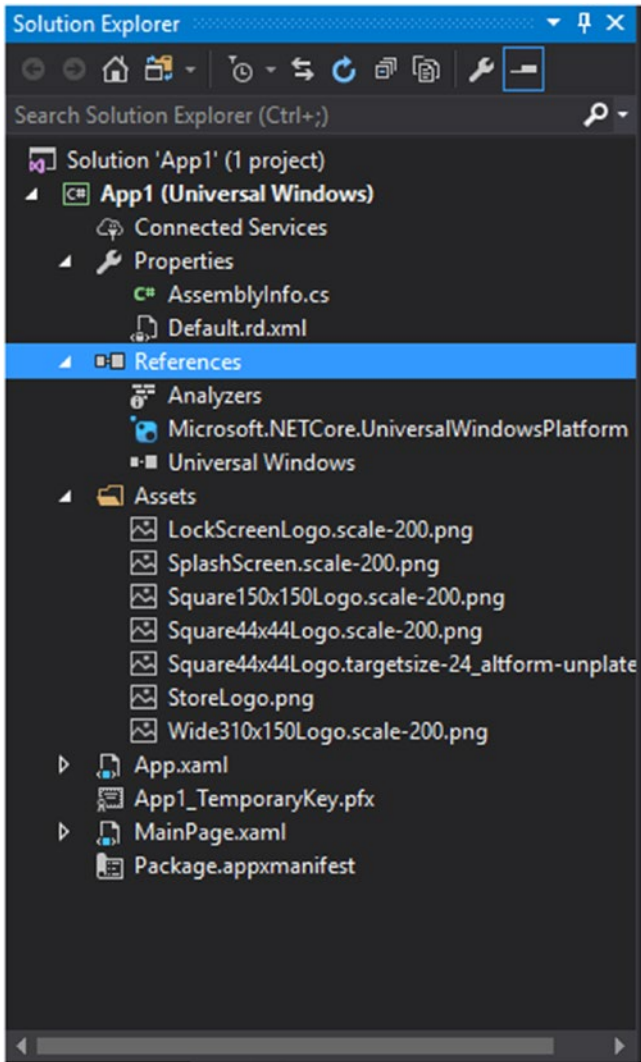


Figure 2-3. Folders and files in the Blank App template

Properties

Under the Properties node in the New Project template is the `AssemblyInfo.cs` file, which is the standard meta-information container for C# projects. Feel free to update the information such as description, copyright, and so on. Most of this information isn't necessary for modern apps, but I tend to update the information anyway out of habit.

References

The template also includes the standard References node, which is prepopulated with two references: `Microsoft.NETCore.UniversalWindowsPlatform` and `Universal Windows`. These references provide the vast majority of functionality and must be included. Throughout this book, we will add references to supplement the default features available to us.

Assets

There is also the Assets folder, which contains all of the images that are part of your application. The tile images and splash screen graphics go in this folder, as well as any other images or assets that need to be packaged with your app when it is deployed. Click on one of the images in Solution Explorer (such as `Logo.scale-100.png`) and press F4 to view the properties. The Build Action for the images is set to Content and set not to copy to the Output directory, as shown in Figure 2-4. Alternatively, you can have the content copied to the Output directory or run a custom tool, although you will not want to do that for the images.

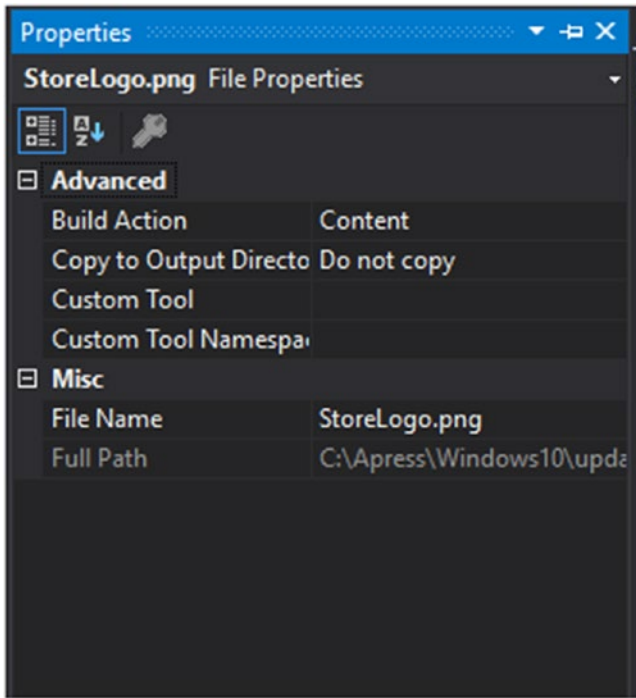


Figure 2-4. Image Asset properties

App1_TemporaryKey.pfx

App1_TemporaryKey.pfx (named after the app—App1 in this case) is the developer license information for the app. We'll update this when we work with push notifications as well as when we get ready to submit our app to the Microsoft Store later in this book.

Package.appxmanifest

Package.appxmanifest contains six tabs that describe your application that we will go on to look at. The actual manifest file is an XML file, but Visual Studio provides a nice GUI to work with the elements in the file, saving us from having to memorize the format or definitions. Double-click on the Package.appxmanifest file to open it in the Visual Studio Editor.

The Application Tab

The Application tab largely replaces `AssemblyInfo.cs`, but also provides many more options, as shown in Figure 2-5. The top section includes the Display Name, Entry Point, Default Language, and Description.

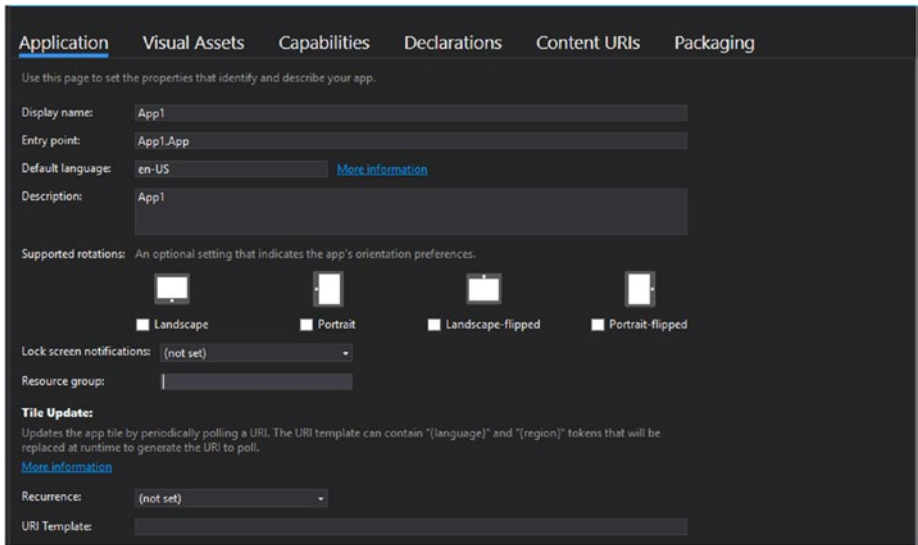


Figure 2-5. Application tab

The next section under the tab is for the supported rotations, or the layout. You can select the rotations as well as the minimum width. For the layout preferences, if all of the options are checked (or none of them checked, as in the default), then all rotations are supported. If only some of the layouts are checked, such as Portrait and Portrait-Flipped, the app will not rotate when a tablet is changed from Portrait to Landscape. If the hardware does not support rotation (such as a traditional laptop), then the setting is essentially meaningless. The following section, Minimum Width, will be covered in detail in Chapter 5.

The Lock Screen notifications, which are covered in Chapter 9 and the next category, Tile Update, provides a mechanism to enter URI details for the source for notifications using a polling mechanism (instead of push notifications). The Resource Group is a name you can give to your application where all processes running under your application will appear under that name.

The Visual Assets Tab

The Visual Assets tab (shown in Figure 2-6) is where you set the Tile Images and Logo, Splash Screen, and Badge Logo, as well as text that can appear on the different tiles. Visual Studio 2017 has the ability to generate the tile images based on a single image and will take that image and generate the tiles of each size. The splash screen is what is shown as your app is activated. The default image is the white box on a dark screen. Select Splash Screen in the left rail to set a new splash screen for your app. By default, any images specified here should be stored in the Assets folder previously discussed.

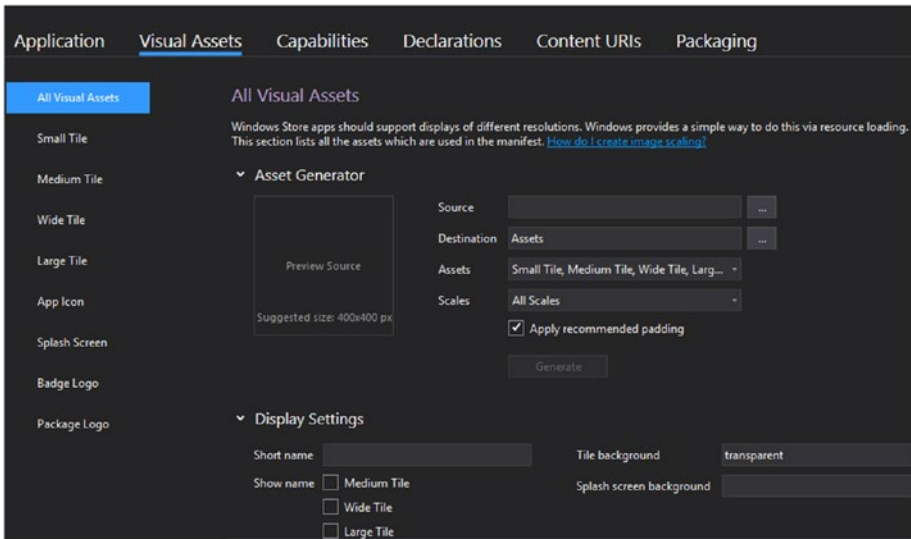


Figure 2-6. Visual Assets tab

The Capabilities Tab

The Capabilities tab (shown in Figure 2-7) is where you specify what features you would like the users to allow when they install your app. By default, Internet (Client) is checked and doesn't require the user to accept the capability (it's assumed that Windows 10 UWP apps can connect to the Internet).

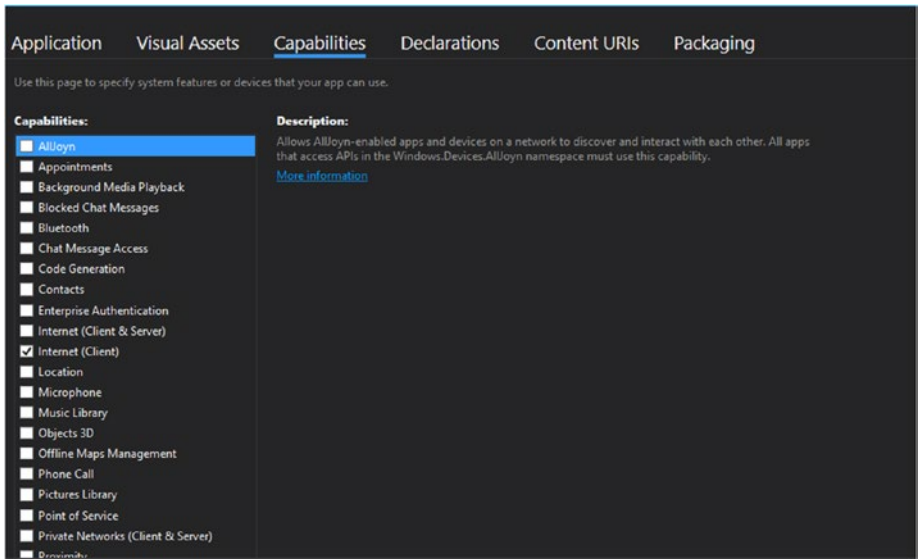


Figure 2-7. Capabilities tab

When users install your app, they will be prompted to allow or deny all of the capabilities (such as the webcam, the libraries, and so forth). Those capabilities will also be placed into the Settings charm under Permissions so that users can change their mind after installation.

The Declarations Tab

The Declarations tab (shown in Figure 2-8) adds capabilities to your app, such as Background Tasks, File Open, and Save pickers, as well as Search and Share Target. Note that Share Target is just below Search, but due to scrolling, it doesn't appear in the image. Many of these features are also covered in later chapters.

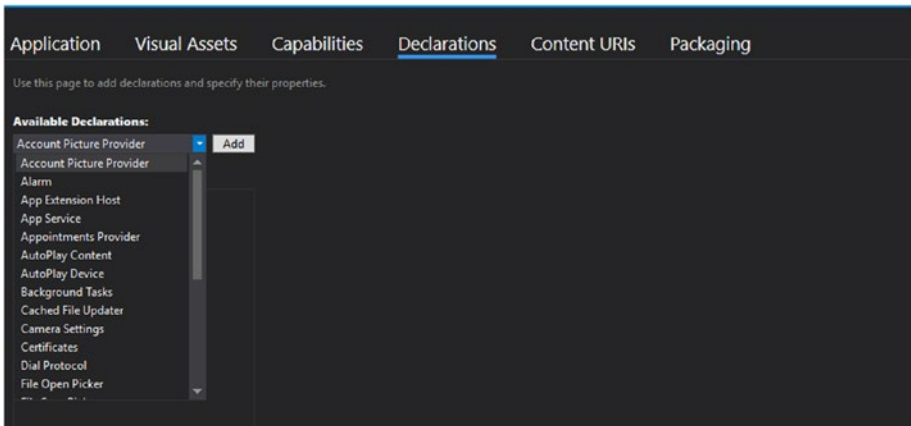


Figure 2-8. Declarations tab

The Content URIs Tab

The Content URIs tab is where you can specify an external web page that is permitted to fire the `ScriptNotify` event. Although we cover push notifications in Chapter 9, we use Azure Mobile Services instead of the mechanisms provided here, so this tab isn't covered in this book.

The Packaging Tab

The final tab, the Packaging tab, is for setting the packaging information. We cover this in depth in Chapter 11.

App.xaml

The `App.xaml` file (and the related `App.xaml.cs` code-behind file) is the entry point for your application. We will spend a lot of time in the code-behind file throughout this book. For this example, we will simply comment out the following line of code in the `OnLaunched` event handler to turn off the frame rate counter.

```
this.DebugSettings.EnableFrameRateCounter = true;
```

Open `App.xaml.cs`, navigate to the `OnLaunched` event handler, and add two slashes to the beginning of the line as such:

```
//this.DebugSettings.EnableFrameRateCounter = true;
```

MainPage.xaml

`MainPage.xaml` is blank and is the default UI page. As in Chapter 1, we want to delete this page and add a new Basic Page to access the additional benefits of the `SuspensionManager`, `NavigationHelper`, and more. Remember to add the new Basic Page with the name `MainPage.xaml`, or alternatively update `App.xaml.cs` to load your new page in the `OnLaunched` event handler. For example, if you named your new page `NewPage.xaml`, change this line

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

to this:

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

Model View ViewModel (MVVM)

The Model-View-ViewModel (MVVM) pattern is wildly popular among XAML developers. Derived from Martin Fowler's Presentation Model pattern, it leverages many Windows 8.1-specific and XAML-specific capabilities to make development cleaner. It is so popular that there are a host of open-source frameworks available, and the pattern has even spilled over from XAML to web developers.

The Pattern

The goal of MVVM is to increase the separation of concerns between the layers of your app, increase testability, and promote code reuse. In this chapter, we just scratch the surface of the pattern, starting with a brief explanation of the parts.

Model

The model is the data for your app. It is not the persistence layer (such as database or web service) but the object representation of your data. The structure of this data is typically in the form of entities or data transport objects (DTOs). They are commonly referred to as POCOs (Plain Old CLR Objects).

View

The view is the window (such as `MainPage.xaml`). The view shows data to the user and takes input from the user. Beyond that, there shouldn't be any other intelligence behind the view. Often, MVVM proponents strive for zero code-behind. My opinion (and this is not meant to start an architectural debate) is that removing code from the code-behind is a pleasant side effect of implementing MVVM properly, but not the goal. But, either way, the view becomes very lightweight.

ViewModel

The ViewModel performs two functions in the MVVM pattern in a XAML world (it's a bit different in the web world):

- The first function is to be a transport mechanism for the model required for the window. There is typically a one-to-one correlation between windows and ViewModels in my code, but architectural differences exist, and your mileage may vary.
- The second job is to act as the controller for the view and the model, receiving the user actions from the view and brokering them accordingly.

ViewModels should also be very lightweight and leverage other classes, such as commands and repositories, handle the heavy lifting.

Creating a Model

Let's start with the model. Add a new folder to your app named `Models` by right-clicking your project and selecting **Add ► New Folder**. Next, add a new Class file named `Customer` by right-clicking on the new project and selecting **Add ► New Item**, as shown in [Figure 2-9](#).

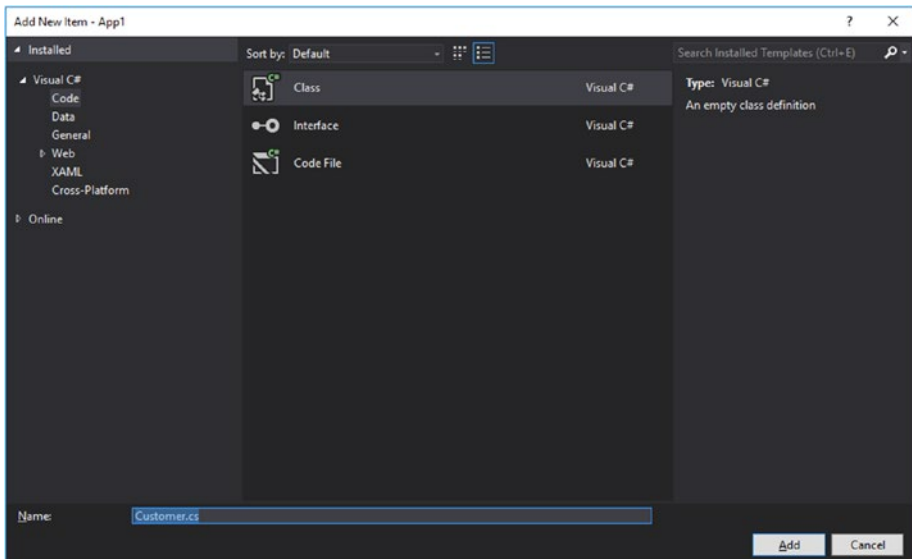


Figure 2-9. Adding the Customer model

Adding Customer Properties

For this simple example, we are only going to have two properties—a first name and a last name. We need to have backing properties to handle `INotifyPropertyChanged` events (as we will see in the next section). To create these properties, open `Customer.cs` and add the following code:

```
public class Customer
{
    private string _firstName;
    private string _lastName;
    public string FirstName
    {
        get
        {
            return this._firstName;
        }
    }
}
```

```
set
{
    if (value != _firstName)
    {
        this._firstName = value;
    }
}
}
public string LastName
{
    get
    {
        return this._lastName;
    }
    set
    {
        if (value != _lastName)
        {
            this._lastName = value;
        }
    }
}
}
```

In the setters, we check to see if the value is different before setting it and updating the backing property. This is to save on calls to the `PropertyChanged` event, as you will see in the next subsection.

INotifyPropertyChanged

The `INotifyPropertyChanged` interface has one event and resides in the `System.ComponentModel` namespace.

```
namespace System.ComponentModel
{
    public interface INotifyPropertyChanged
    {
        event PropertyChangedEventHandler PropertyChanged;
    }
}
```

To implement this interface, add a `using` for `System.ComponentModel` to the `Customer` class, and then add the interface and the event. The resulting code is shown next with the property setters and getters omitted for brevity:

```
using System.ComponentModel;
public class Customer : INotifyPropertyChanged
{
    // omitted for brevity
    public event PropertyChangedEventHandler PropertyChanged;
}
```

Next, we need to implement the event, and we want to make sure something is listening before firing the event off. Add a `using` for `System.Runtime.CompilerServices` as follows:

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

Then, add the code for the `OnPropertyChanged` method:

```
private void OnPropertyChanged([CallerMemberName] string member
= "")
{
    PropertyChanged?.Invoke(this, new PropertyChangedEventArgs
(member));
}
```

The `PropertyChanged` event informs the binding engine to reinspect the data source for the property sent in the event arguments. You can also include the empty string, which will ask the binding engine to refresh all bindings on the `Custom` object. The `CallerMemberName` attribute will pass in the method name that executed the `OnPropertyChanged` method. For our purposes, we want the setters for each of the properties to call `OnPropertyChanged` when a value on the model is updated (but not when it is set to the same value; hence the added check that we did in the previous step). The full code is listed here:

```
public class Customer : INotifyPropertyChanged
{
    private string _firstName;
    private string _lastName;

    public event PropertyChangedEventHandler
PropertyChanged;

    private void OnPropertyChanged([CallerMemberName]
string member = "")
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs
(member));
    }
}
```



```
public string FirstName
{
    get
    {
        return this._firstName;
    }
    set
    {
        if (value != _firstName)
        {
            this._firstName = value;
            OnPropertyChanged();
        }
    }
}
public string LastName
{
    get
    {
        return this._lastName;
    }
    set
    {
        if (value != _lastName)
        {
            this._lastName = value;
            OnPropertyChanged();
        }
    }
}
}
```

Creating the ViewModel

We are going to create a very simple ViewModel that starts by wrapping a Customer instance. In real-world examples, you would have ObservableCollections (discussed later in the book) and probably more than one model type in your window. As we are just showing the pattern here, we are going to keep things very simple.

Create a new folder called ViewModels (just like before) and then add a new class called MainPageViewModel. Note that there aren't firm rules around naming your ViewModels, but a popular convention is to add ViewModel to the end of the view that will be using it.

For starters, we want the ViewModel to instantiate an instance of the Customer class (again, in a real app, this would come from a repository in the data access layer). First, add a using for the models.

```
using App1.Models;
```

And then create the ViewModel:

```
public class MainPageViewModel
{
    public Customer MyCustomer { get; set; }
    public MainPageViewModel()
    {
        MyCustomer = new Customer()
            { FirstName = "Bob", LastName = "Smith" };
    }
}
```

Next, we will create the RelayCommand. The RelayCommand class was added to our project when we added the Basic Page, and it takes care of a lot of plumbing that we would have to do ourselves if we created an instance of ICommand manually. The command (as you will soon see) gets tied to an actionable UI element, such as a button or a menu option. We will bind the command in the next section.

When you create a `RelayCommand`, it takes two parameters: The first is the delegate that is executed when the action is taken. The second is optional and determines if the command is allowed to execute. We are only going to use the first parameter in this example. Add the namespace for the `RelayCommand`, `App1.Common`:

```
using App1.Common;
```

And then add the following code to the `MainPageViewModel` class:

```
private RelayCommand _updateNameCommand;
private void UpdateName()
{
    MyCustomer.FirstName = "Sue";
}
public RelayCommand UpdateNameCommand
{
    get
    {
        if (_updateNameCommand == null)
        {
            _updateNameCommand = new RelayCommand(UpdateName);
        }
        return this._updateNameCommand;
    }
    set
    {
        this._updateNameCommand = value;
    }
}
```

We want to make sure the `RelayCommand` is not null. We could easily do that in the constructor for the `ViewModel`, but I like to do that in the getter so that the related code is grouped together. The action will change

the `FirstName` of the customer to “Sue” from “Bob.” That’s it. We are done with the `ViewModel`.

Updating the Code-Behind

Open `MainPage.xaml.cs` and add the following `using` statements:

```
using App1.Models;  
using App1.ViewModels;
```

Then add the following line of code to the constructor:

```
this.DataContext = new MainPageViewModel();
```

This creates a new instance of our `ViewModel` and sets the `DataContext` for the entire view to the `ViewModel`. Data binding needs two things—the object that is the source of the data and the path to the property that is being bound. If a binding statement doesn’t include a `DataContext`, the element will look up the element tree (to all of its parents, in order) to find a `DataContext`. Once it finds one, it stops and then attempts to bind the element based on the `Path` and the found `DataContext`. When we assign the `DataContext` to the entire view, everything will then use that specified source object.

That’s it. No more code is necessary in the code-behind!

Creating the View

For the view, we are going to create a very simple form that displays the first and last name and has a button to execute the name change. The finished view is shown in [Figure 2-10](#). One thing you’ll notice is the button right in the middle of the view, after all of that talk about content over chrome in [Chapter 1](#). Yes, I broke the rules, but we are going to talk about command bars and app bars later in the book, and I didn’t want to throw too much new content at you.

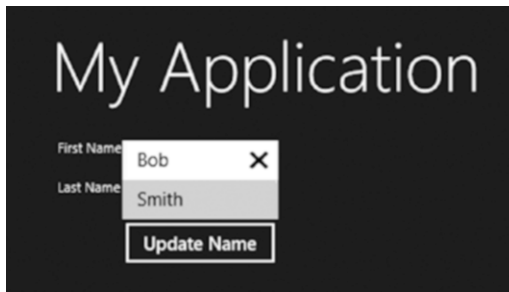


Figure 2-10. View for the MVVM example

Open `MainPage.xaml` and add the following XAML just before the final closing `</Grid>` tag:

```
<Grid Grid.Column="0" Grid.Row="1" Margin="120,0,0,0" Width="Auto">
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto"/>
    <ColumnDefinition Width="Auto"/>
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto"/>
    <RowDefinition Height="Auto"/>
    <RowDefinition Height="Auto"/>
  </Grid.RowDefinitions>
  <TextBlock Grid.Row="0" Grid.Column="0">First Name</TextBlock>
  <TextBlock Grid.Row="1" Grid.Column="0">Last Name</TextBlock>
  <TextBox Grid.Row="0" Grid.Column="1"
    Text="{Binding Path=MyCustomer.FirstName}"></TextBox>
  <TextBox Grid.Row="1" Grid.Column="1"
    Text="{Binding Path=MyCustomer.LastName}"></TextBox>
```

```
<Button Grid.Row="2" Grid.Column="1"  
    Content="Update Name" Command="{Binding  
    Path=UpdateNameCommand}"/>  
</Grid>
```

We will cover the controls and layouts in depth later in this book, so I just want to focus on the binding for the `TextBox` elements as well as the `Command` property of the `Button` element. The `TextBox` controls have a binding statement of `Path=MyCustomer.FirstName` and `Path=MyCustomer.LastName`. Each of these controls will look at the `MainPageViewModel` for the property `MyCustomer` and then look for the correct property on that object. This then binds these controls to the first and last name of the customer.

Buttons also can leverage the command pattern. Instead of double-clicking on a button to create an event handler in the code-behind, we bind the button's click action to a command. In this case, it's the `UpdateNameCommand` we created in the `ViewModel`. This gives a very clean implementation of the view and allows for separation of concerns all the way down.

Testing the App

Press `F5` to run the app (either in the simulator or on your local machine) and click the button. The First Name is changed to Sue (as in Figure 2-11), and the view is automatically updated! That is because `TextBox` bindings, by default, are two-way. If the source changes, the view is updated; if the view changes, the source is updated.

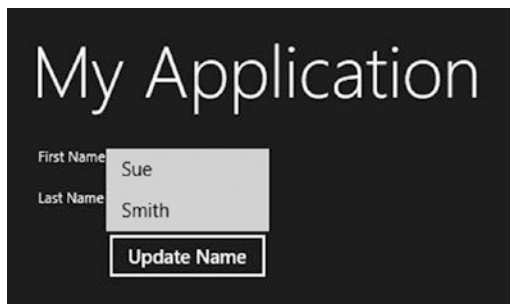


Figure 2-11. *The First Name changed to Sue*

Guidance

The MVVM pattern is very popular and extremely powerful, but it isn't for every app (or every developer). The rest of the code in this book does *not* use the pattern to keep the examples simple and clean, but I recommend that you spend some time learning the pattern so you can make an educated decision for yourself as to when it will help your architecture.

Navigation

The next topic that we explore is navigation. All Windows 10 UWP apps are single-page apps. In XAML, this is implemented by creating a `Frame` that all subsequent pages get loaded into. This code was written for us when we created the new project from the Blank App template.

Open `App.xaml.cs` and examine the following code (much of it I omitted to just show the relevant parts):

```
protected override void OnLaunched(LaunchActivatedEventArgs e)
{
    Frame rootFrame = Window.Current.Content as Frame;
    if (rootFrame == null)
```

```

{
    rootFrame = new Frame();
    rootFrame.NavigationFailed += OnNavigationFailed;
    Window.Current.Content = rootFrame;
}
if (rootFrame.Content == null)
{
    rootFrame.Navigate(typeof(MainPage), e.Arguments);
}
Window.Current.Activate();
}

```

After creating a frame when the app is launched, the method determines if there is anything in the frame. If it is null, then the app is started fresh (and not resumed or restored from termination) and a new frame is created. The current window's content is then assigned this frame. If the content is still null, the app navigates to the Main Page (the default start page for the app).

Navigation is a bit of a misnomer here. This is not like going from one web page to another. It is really swapping out the contents of the frame with the contents of the new page.

To see how this works, we are going to add a new Basic Page to the sample app we started to show the MVVM pattern.

Creating a New Page

Right-click on the project and select Add ► New Item ► Basic Page. Name it PageTwo.xaml, as shown in Figure 2-12.

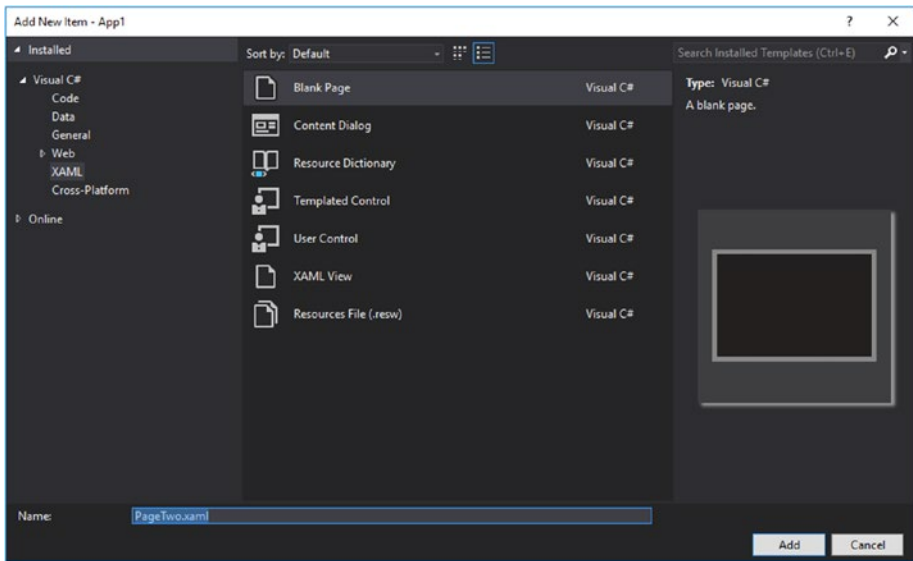


Figure 2-12. Adding PageTwo.xaml

Creating the UI

The UI for the second page will just be a copy of the MainPage without the buttons, as shown in Figure 2-13. Notice the back button? That comes along for free as part of the Basic Page template. More on that shortly.

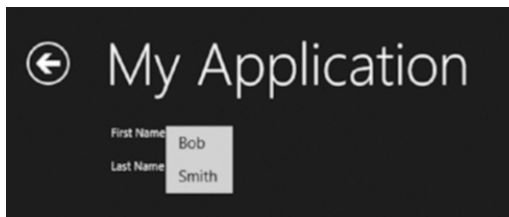


Figure 2-13. PageTwo UI

To create this page, open `PageTwo.xaml` and enter the following XAML just before the final closing `</Grid>` tag:

```
<Grid Grid.Column="0" Grid.Row="1" Margin="120,0,0,0"
Width="Auto">
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto"/>
    <ColumnDefinition Width="Auto"/>
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto"/>
    <RowDefinition Height="Auto"/>
  </Grid.RowDefinitions>
  <TextBlock Grid.Row="0" Grid.Column="0">First Name</TextBlock>
  <TextBlock Grid.Row="1" Grid.Column="0">Last Name</TextBlock>
  <TextBox Grid.Row="0" Grid.Column="1"
    Text="{Binding Path=FirstName}"></TextBox>
  <TextBox Grid.Row="1" Grid.Column="1"
    Text="{Binding Path=LastName}"></TextBox>
</Grid>
```

Notice that the binding statements are exactly the same. This is the advantage of setting the `DataContext` at the window level instead of for each control. We can pass in any data object that has those properties and it will just simply work.

Adding Navigation to the MainPage

Navigation commands belong in the top app bar (or command bar), but for this example, we'll create a button on the Main Page to trigger the navigation. To do so, open `MainPage.xaml` and add the following lines to the grid that we added for the previous example (the last grid in the XAML).

In the RowDefinition block, add:

```
<RowDefinition Height="Auto"/>
```

At the end of the XAML (before the closing </Grid> tag for the same grid we've been working with), add:

```
<Button Name="NavigateBtn" Grid.Row="3" Grid.Column="1"
Content="Navigate" Click="NavigateBtn_Click"/>
```

The full XAML looks like this:

```
<Grid Grid.Column="0" Grid.Row="1" Margin="120,0,0,0" Width="Auto">
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto"/>
    <ColumnDefinition Width="Auto"/>
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto"/>
    <RowDefinition Height="Auto"/>
    <RowDefinition Height="Auto"/>
    <RowDefinition Height="Auto"/>
  </Grid.RowDefinitions>
  <TextBlock Grid.Row="0" Grid.Column="0">First Name</TextBlock>
  <TextBlock Grid.Row="1" Grid.Column="0">Last Name</TextBlock>
  <TextBox Grid.Row="0" Grid.Column="1"
    Text="{Binding Path=MyCustomer.FirstName}"></TextBox>
  <TextBox Grid.Row="1" Grid.Column="1"
    Text="{Binding Path=MyCustomer.LastName}"></TextBox>
  <Button Grid.Row="2" Grid.Column="1"
    Content="Update Name" Command="{Binding Path=
    UpdateNameCommand}"/>
```

```
<Button Name="NavigateBtn" Grid.Row="3" Grid.Column="1"
    Content="Navigate" Click="NavigateBtn_Click"/>
</Grid>
```

We now need to add the event handler for the button click event. Note that if we were sticking with MVVM, we would add a command for this, but to keep things simple (and to match the rest of the examples in this book), we are just going to add an event handler in the code-behind.

To do this, open `MainPage.xaml.cs` and add this code:

```
private void NavigateBtn_Click(object sender, RoutedEventArgs e)
{
    this.Frame.Navigate(typeof(PageTwo),
        ((this.DataContext as MainPageViewModel)!=null) ?
        (this.DataContext as MainPageViewModel).MyCustomer:
        (new Customer() {FirstName="Jane", LastName="Doe"}));
}
```

First and foremost, the code calls the `Navigate` method on the frame. This is the same frame that was created in `App.xaml.cs` (and previously explained in the text). The first parameter is required, and it requires the type of page that is to be loaded into the frame. In our case, it is `typeof(PageTwo)`. The navigation framework will take this type, use reflection to create an instance of the page, and load it into the frame.

The second parameter is optional and will be passed into the page that is being navigated to as the `Parameter` property of the `NavigationEventArgs`. In our case, we want to grab the customer out of the `ViewModel`, and the easiest way to do that is to use the window's `DataContext`. If all went well, the `DataContext` is an instance of the `MainPageViewModel`, and we can call the `MyCustomer` property once we convert it back to the `MainPageViewModel` type. We added some defensive programming to pass in a new `Customer` instance in case there is an error along the way.

Handling the NavigatedTo Event

Open `PageTwo.xaml.cs` and add the following override to the `OnNavigatedTo` event handler.

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    this.DataContext = e.Parameter as Customer;
}
```

This sets the data context for the window as the customer that was passed in from the page that navigated to `PageTwo`, which in this simple case is the `Main Page`. Now the binding statements in the XAML will use the customer passed in to show the values on the model, as shown in [Figure 2-13](#). If the type conversion doesn't work, the `DataContext` will be set to `null`, and the bindings will fail silently, resulting in a blank form.

The Back Button

Now to the back button. Open `PageTwo.xaml` and examine the XAML that created the back button, as shown here:

```
<Button x:Name="backButton" Margin="39,59,39,0"
    Click="backButton_Click"
    Style="{StaticResource NavigationBackButtonNormalStyle}"
    VerticalAlignment="Top" AutomationProperties.Name="Back"
    AutomationProperties.AutomationId="BackButton"
    AutomationProperties.ItemType="Navigation Button"/>
```

We then add the event handler on `PageTwo.xaml.cs` with the following code.

```
private void backButton_Click(object sender,
    RoutedEventArgs e)
{
    Frame rootFrame = Window.Current.Content as Frame;
    if(rootFrame.CanGoBack)
    {
        rootFrame.GoBack();
    }
}
```

As you can see, the navigation framework supplied by Microsoft makes it super simple to load new pages into the mainframe, pass values from page to page, and ensure that users can navigate back through your app.

Summary

In this chapter, we looked at the anatomy of a Windows 10 UWP app project and the myriad of options available to you. We touched on the MVVM pattern and data binding. Finally, we covered the single-page nature of Windows 10 UWP apps and using navigation framework to load different pages into your app.

In the next chapter, we will take a deep look into all of the controls available for Windows 10 UWP app development.

CHAPTER 3

Themes, Panels, and Controls

Themes provide a consistent display for all of the pages in your app. Controls provide a way for users to interact with your app. Panels hold the controls. Together, they help you define your UI. This chapter takes a quick look at the theme options and then dives deep into the panels and controls offered out of the box for the Windows 10 UWP app.

Choosing a Theme

Windows 10 UWP applications use a Light theme by default. Using a Dark theme is typically more user- and battery-friendly for tablet-based applications. However, based on your application, the Light theme might create a better user experience.

The best way to understand the themes is to see them in action. To do so, open Visual Studio and select New Project ► Windows Universal ► Blank App (Universal Windows). Name the project Controls1, as shown in Figure 3-1.

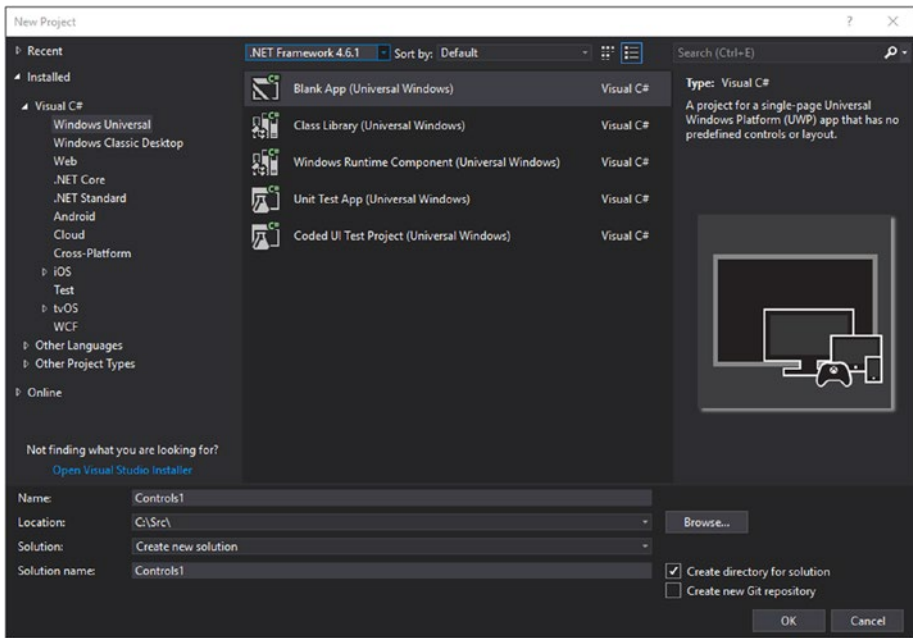


Figure 3-1. *New Project in Visual Studio*

In Windows 8, the application theme can be changed in `App.xaml` by specifying `RequestedTheme="Light || Dark"`, as shown in the following code:

```
<Application
  x:Class="Controls1.App"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:Controls1"
  RequestedTheme="Light">
```

In Windows 10 UWP applications, setting the `RequestedTheme` in `App.xaml` changes the foreground theme but not the background theme for panels.

To see the Dark and Light themes in action, add a `TextBlock` to the `Grid` in `MainPage.xaml`, add a style for `HeaderTextBlockStyle`, and set the text to "Hello, World". The resulting code is shown here:

```
<Grid>
  <TextBlock Text="Hello, World"
    Style="{StaticResource HeaderTextBlockStyle}"
  />
</Grid>
```

When you run the program, "Hello, World" is shown in light text on a dark background, as shown in Figure 3-2.



Figure 3-2. A text block with the default Dark theme

In `App.xaml`, add `RequestedTheme="Light"` to the `<Application />` tag to switch to the Light theme:

```
<Application
  x:Class="Controls1.App"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
  presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:Controls1"
  RequestedTheme="Light">
  <!-- Omitted for brevity -->
</Application>
```

When you run the program again, all that shows is a dark screen. This is because the control is set to use the Light theme. In the Light theme, controls are expecting a light background, so they are configured to use dark fonts. Panels (covered in the next section) must have their background set to use `ApplicationPageBackgroundThemeBrush` in order for `RequestedTheme` to take effect.

To do so, change the Grid so that it uses `ApplicationPageBackgroundThemeBrush`, as in the following code:

```
<Grid Background="{ThemeResource
ApplicationPageBackgroundThemeBrush}">
  <TextBlock Text="Hello, World"/>
</Grid>
```

Run the application again and you will see dark text on a light background, as in Figure 3-3.

Hello, World

Figure 3-3. *A text block with a Light theme and the grid configured to use the requested theme*

Using Panels

You can't discuss controls without first discussing panels. Panels are a special kind of UI control that "holds" other controls, supplies a place to put your controls, and helps manage the layout. Panels are not new to Windows 10 UWP; they have been in XAML since the beginning, with WPF and Silverlight. The type of panel determines the behavior of the contained controls (as will be described). Conceptually, they are very similar to ASP.

NET placeholder controls in that they contain other controls and they can both have controls added through code during runtime. However, while ASP.NET placeholder controls are just another option for creating UIs, the XAML containers are the key components of creating UIs. There are a number of types of panels available out of the box with Windows 10, the most important of which are:

- Canvas
- Grid
- StackPanel
- RelativePanel

The Canvas Control

The Canvas control is used primarily in games and other applications where you need precise (to the pixel) control over the placement of every object in your application. We'll show how to use Canvas, but then we won't return to it for the rest of the book, as our focus is not on games.

To start, open `MainPage.xaml` and remove the `Grid` and the `TextBlock`. Next, add a `Canvas`. Again, to add a `Canvas` (or any other control), you can drag it from the toolbox onto the design surface or the XAML, or you can enter it by hand by typing the following where the `Grid` used to be:

```
<Canvas>  
</Canvas>
```

METHODS FOR ADDING CONTROLS

Depending on how you add controls to your page, the resulting XAML can be very different. If you do so by dragging a control from the toolbox directly into the XAML editor, the resulting XAML will be very clean:

```
<Canvas/>
```

If you do so by dragging a control from the toolbox onto the design surface in Visual Studio, there will be a lot more attributes set on the control, as the design surface interprets where the control is dropped. For example, in my test app, dragging a control onto the design surface resulted in the following XAML:

```
<Canvas HorizontalAlignment="Left" Height="100"  
Margin="221,399,0,0"  
Grid.Row="1" VerticalAlignment="Top" Width="100"/>
```

There are advantages and disadvantages to the different methods of adding controls. The best option is to try them all and determine what works best for you.

You'll place controls within the Canvas control (that is, between the opening and closing tags). If you drag an Ellipse onto the design surface, Visual Studio will fill in a number of properties for you so that the Ellipse is visible:

```
<Ellipse Fill="#FFF4F4F5"  
Height="100"  
Canvas.Left="205"  
Stroke="Black"  
Canvas.Top="111"  
Width="100" />
```

Setting the Height and Width to the same value (in this case, 100) makes the Ellipse a circle. The Canvas.Left and Canvas.Top properties set the location of the Ellipse with respect to the left and top boundaries of the Canvas. In this case, the Ellipse is 205 pixels to the right of the left margin and 111 pixels down from the top boundary.

Change the Fill property to "Red" to see the circle more clearly.

You can place as many objects as you like onto the Canvas. Try the following XAML:

```
<Canvas>
  <Ellipse Fill="Red"
    Height="100"
    Canvas.Left="205"
    Stroke="Black"
    Canvas.Top="111"
    Width="100"
    Canvas.ZIndex="1"/>
  <Rectangle Fill="Blue"
    Height="188"
    Canvas.Left="82"
    Stroke="Black"
    Canvas.Top="40"
    Width="118" />
  <Rectangle Fill="Blue"
    Height="137"
    Canvas.Left="278"
    Stroke="Black"
    Canvas.Top="91"
    Width="140"
    Canvas.ZIndex="0"/>
</Canvas>
```

The result is shown in Figure 3-4.

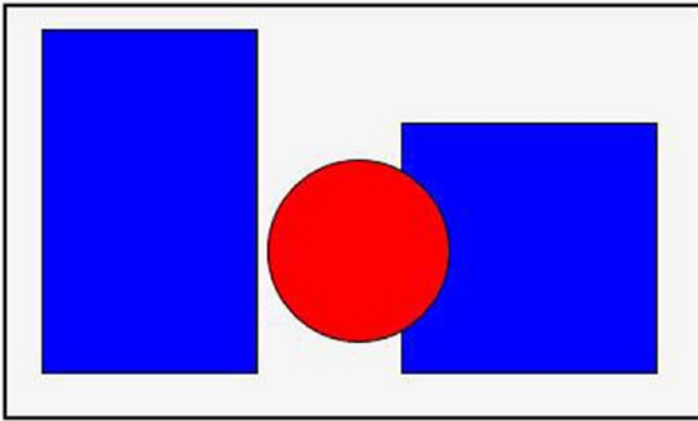


Figure 3-4. *Two rectangles and an ellipse*

Normally, this would create two rectangles and an ellipse (appearing as a circle since the height and width are set to the same value). The second rectangle would partially cover the ellipse since it was declared after the ellipse in the XAML. In this case, however, the ellipse occludes the rectangle because the `ZIndex` was set higher for the ellipse (placing the ellipse “on top of” the rectangle). The `ZIndex` determines the layering as if the shapes were on a three-dimensional surface, as shown in Figure 3-5.

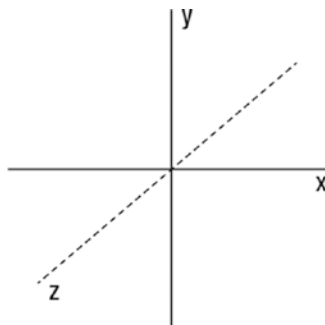


Figure 3-5. *ZIndex*

The Canvas control really comes into its own when you work with animations, a topic beyond the scope of this book.

The Grid Control

The Grid control is the workhorse panel of Windows 10 UWP. It is so commonly used that Microsoft made it the default panel for a new window.

Defining Rows and Columns

As one might expect, grids are composed of rows and/or columns. You can define them using various measurements. The most common way is to define the actual size:

```
<RowDefinition Height="50" />
```

or to define the relative size of two or more rows:

```
<RowDefinition Height="*" />
<RowDefinition Height="2*" />
```

This code indicates that the second row will be twice as big as the first (and will take up two thirds of the available space). An asterisk alone indicates 1*.

The last way to define them is by declaring a size to be "Auto," in which case it will size to the largest object in that row or column.

To see this all at work, create a new project named Controls2 and add the following code in the Grid that is provided:

```
<Grid Background="{StaticResource
ApplicationPageBackgroundThemeBrush}">
  <Grid.RowDefinitions>
    <RowDefinition Height="100" />
    <RowDefinition Height="Auto" />
```

```

    <RowDefinition Height="*" />
    <RowDefinition Height="2*" />
</Grid.RowDefinitions>
<Grid.ColumnDefinitions>
    <ColumnDefinition />
    <ColumnDefinition />
</Grid.ColumnDefinitions>
<Rectangle Fill="Red"
    Height="40"
    Width="20"
    Grid.Row="0"
    Grid.Column="0" />
<Rectangle Fill="Blue"
    Height="40"
    Width="20"
    Grid.Row="1"
    Grid.Column="0" />
<Rectangle Fill="Green"
    Height="40"
    Width="20"
    Grid.Row="2"
    Grid.Column="0" />
<Rectangle Fill="Yellow"
    Height="40"
    Width="20"
    Grid.Row="3"
    Grid.Column="0" />
</Grid>

```

Notice that all the rectangles are the same size (Height="40"), but the rows are of differing sizes. The result is shown in Figure 3-6.

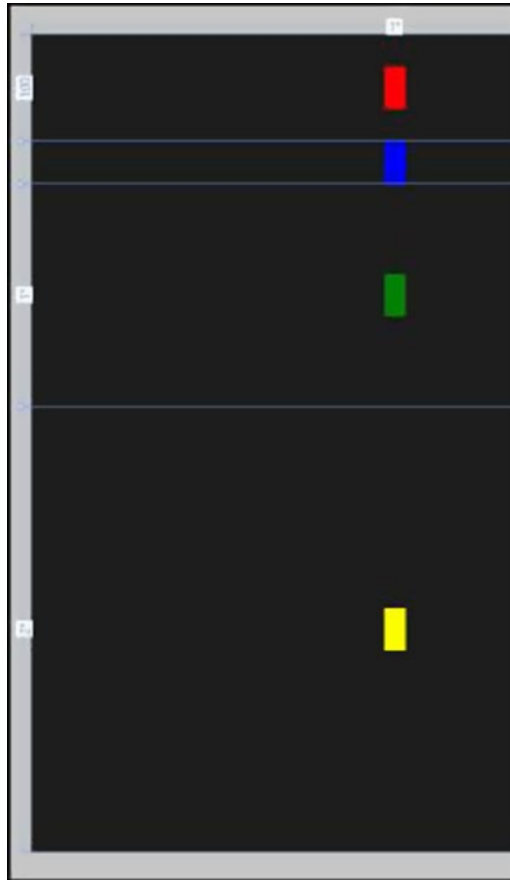


Figure 3-6. *Grid with varying heights*

The things to notice in Figure 3-6 are: The first rectangle is centered in a row that is 100 pixels high. The second row is fit to the size of the rectangle. The next two rows divide the remaining space in the proportion 1:2 and the rectangles are centered in each. The image was cropped to make all of this more obvious.

Notice that the relative sizes are shown (e.g., 100, 1*, 2*, etc.) around the borders of the design surface to make it easier to see (and adjust) the sizes.

In this example, we placed everything in the first column to save space in the figure.

Alignment, Margins, and Padding

In setting objects into rows and columns of the Grid, you often want finer control over their placement. The first properties you might set are `HorizontalAlignment` and `VerticalAlignment`. These are enumerated constants, as shown in Figure 3-7.

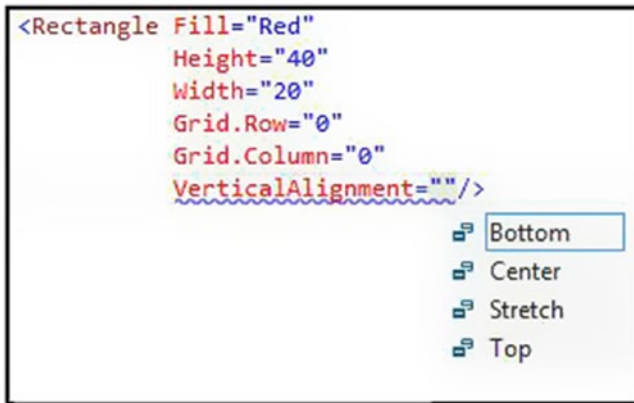


Figure 3-7. *Setting the VerticalAlignment*

Margins are set to further fine-tune the placement of the object within the grid's cell. You can set margins as any of the following three types of values:

- *A single value:* If you set the margin as a single value in XAML, all four margins (top, left, bottom, and right) will be set to the entered value.
- *A pair of values:* If you set the margin as a pair of values, the left and right margins will be assigned the value of the first number, and the top and bottom margins will be set to the second value.

- *Four values*: Finally, if you set the margins as four numbers, they will be assigned left, top, right, and bottom. This is different from cascading style sheets (CSS) in web development, where the numbers are in top-right-bottom-left order.

Thus, if you set

```
Margin="5"
```

you create a margin of five pixels all around the object. But if you set

```
Margin = "10, 20"
```

you create a margin of 10 on the left and right and 20 on the top and bottom. And, finally, if you set

```
Margin = "5, 10, 0, 20"
```

you create a margin of 5 on the left, 10 on the top, 0 on the right, and 20 on the bottom.

Padding refers to the space *within* a control between its border and its contents. You can see padding at work when you create a button and set the padding—the contents are spaced further from the edges. In Controls2a, you see that we create three buttons, setting the padding to be quite small in the second and quite large in the third:

```
<Grid Background="{StaticResource
ApplicationPageBackgroundThemeBrush}">
  <StackPanel Margin="50">
    <Button Content="Hello" />
    <Button Content="Hello"
      Padding="5" />
    <Button Content="Hello"
      Padding="25" />
  </StackPanel>
</Grid>
```

The result is shown in Figure 3-8.



Figure 3-8. *Padding*

You can see in Figure 3-8 that the padding affects the spacing around the content but only within the button, not the spacing between buttons (which is controlled by the margin).

The StackPanel Control

The StackPanel control is both simple and useful. It allows you to “stack” one object on top of another, or one object to the right of another (horizontal stacking). No room is created between the objects, but you can easily add space by setting a margin. You can add any control inside a stack, including another stack.

For example, create a new application (Controls3) and add the following code:

```
<Grid Background="{StaticResource
ApplicationPageBackgroundThemeBrush}">
  <StackPanel HorizontalAlignment="Left" Margin="50">
    <Rectangle Fill="Red">
```

```

    Height="50"
    Width="50" />
<Ellipse Fill="Blue"
    Height="50"
    Width="50" />
<StackPanel Orientation="Horizontal">
    <Ellipse Fill="Green"
        Height="20"
        Width="20" />
    <Ellipse Fill="Yellow"
        Height="20"
        Width="20" />
</StackPanel>
</StackPanel>
</Grid>

```

Here, you have two instances of `StackPanel`—an outer `StackPanel` and an inner `StackPanel`. The outer `StackPanel` consists of a `Rectangle` and an `Ellipse` stacked on top of another. The inner `StackPanel` has its orientation set to `Horizontal` (the default is `Vertical`), and within the inner `StackPanel` are two somewhat smaller ellipses.

The result is shown in Figure 3-9.

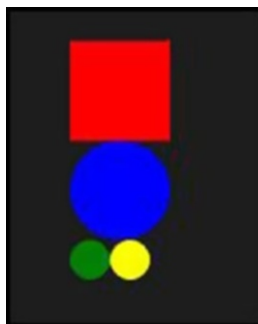


Figure 3-9. *StackPanel objects*

The RelativePanel Control

The `RelativePanel` control lets you arrange elements in relation to each other. Utilizing this panel will allow you to control exactly where you want elements. For example, we can create something that looks like a flag by using the following XAML:

```
<RelativePanel BorderBrush="Gray" BorderThickness="1">
  <Rectangle x:Name="WhiteRect" Fill="White" Height="40"
    Width="44"/>
  <Rectangle x:Name="BlueRect" Fill="Blue"
    Height="10" Width="88"
    RelativePanel.RightOf="WhiteRect" />
  <Rectangle x:Name="RedLine" Fill="Red"
    Height="10" Width="88"
    RelativePanel.RightOf="WhiteRect"
    RelativePanel.Below="BlueRect"/>
  <Rectangle x:Name="BlueLine" Fill="Blue"
    Height="10" Width="88"
    RelativePanel.RightOf="WhiteRect"
    RelativePanel.Below="RedLine"/>
  <Rectangle x:Name="RedLine2" Fill="Red"
    Height="10" Width="88"
    RelativePanel.RightOf="WhiteRect"
    RelativePanel.Below="BlueLine"/>
  <Rectangle x:Name="GreenRect" Fill="Gold"
    Height="44"
    RelativePanel.Below="WhiteRect"
    RelativePanel.AlignLeftWith="WhiteRect"
    RelativePanel.AlignRightWith="BlueRect"/>
</RelativePanel>
```

In Figure 3-10, you will see there is a white box and lines that alternate blue and red all on top of a yellow box.



Figure 3-10. Flag design

The Border Control

The Border control is a container that can draw a border, background, or both around one or more other objects. Technically, the Border control, like many other controls, can only have a single object as its contents, but that object can be something like a StackPanel that can in turn have many objects within it, greatly increasing the utility of the Border. Create another project called Controls4 and add the following XAML to the Main Page:

```
<Border BorderBrush="Red"
  BorderThickness="5"
  Height="150"
  Width="150"
  Background="Wheat">
  <StackPanel VerticalAlignment="Center" >
    <Rectangle Height="50"
      Width="50"
      Fill="Blue"
      Margin="5"/>
    <Rectangle Height="50"
      Width="50"
```

```

    Fill="Black"
    Margin="5"/>
</StackPanel>
</Border>

```

In Figure 3-11, the two squares have 10 pixels between them. That is because each declared a margin of 5, and the bottom margin of the upper rectangle was added to the top margin of the lower rectangle (5 + 5).

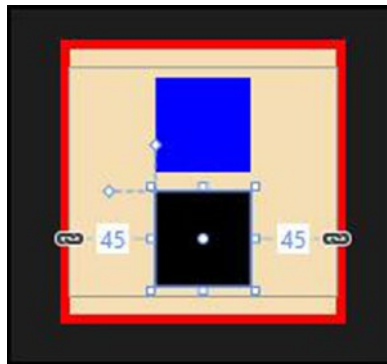


Figure 3-11. Border control

Working with Controls

In addition to the various panels, the Windows 10 UWP toolbox is chock-full of additional controls for gathering or displaying information and/or for interacting with the end user. You can add controls to your page in a number of ways:

- Open Blend and drag a control onto the design surface.
- Open Blend and create the control by hand in XAML.
- Open Visual Studio and drag a control onto the design surface.

- Open Visual Studio and drag a control into XAML.
- Open Visual Studio and create the control by hand in XAML.

Almost too many choices. For simplicity, we assume that you are working in Visual Studio. Blend is a powerful tool for XAML design work (for laying out controls, etc.), and we highly recommend you take a look at it, but the design work can often best be done in Blend.

Dragging a control from the toolbox onto either the design surface or directly into the XAML source itself is a great way to add controls to your page. If your control needs a special namespace, Visual Studio will add it for you automatically if you drag the control into place. If you add the control manually by typing the XAML, you'll have to add the namespace by hand.

This chapter covers some of the more important controls and panels but does not endeavor to be exhaustive. A few controls, such as `GridView`, won't be covered in detail until later in the book.

TextBlock and TextBox

There are a number of simple controls for displaying or retrieving data from the user. For example, text is typically displayed using a `TextBlock` control and is retrieved from the user with a `TextBox` control. In `Controls5`, we create a very simple data entry form as follows:

```
<Grid Background="{StaticResource
ApplicationPageBackgroundThemeBrush}">
  <Grid.RowDefinitions>
    <RowDefinition Height="50" />
    <RowDefinition Height="50" />
    <RowDefinition Height="50" />
  </Grid.RowDefinitions>
```

```
<Grid.ColumnDefinitions>
  <ColumnDefinition />
  <ColumnDefinition />
</Grid.ColumnDefinitions>
<TextBlock Text="First Name"
  FontSize="20"
  Margin="5"
  HorizontalAlignment="Right"
  VerticalAlignment="Center" />
<TextBox x:Name="FirstName"
  Width="200"
  Height="40"
  Grid.Row="0"
  Grid.Column="1"
  HorizontalAlignment="Left"
  Margin="5"
  VerticalAlignment="Center" />
<TextBlock Text="Last Name"
  Grid.Row="1"
  FontSize="20"
  Margin="5"
  HorizontalAlignment="Right"
  VerticalAlignment="Center" />
<TextBox x:Name="LastName"
  Width="200"
  Height="40"
  Grid.Row="1"
  Grid.Column="1"
  HorizontalAlignment="Left"
  Margin="5"
  VerticalAlignment="Center" />
```

```

<TextBlock Text="Job Title"
  Grid.Row="2"
  FontSize="20"
  Margin="5"
  HorizontalAlignment="Right"
  VerticalAlignment="Center" />
<TextBox x:Name="JobTitle"
  IsSpellCheckEnabled="True"
  IsTextPredictionEnabled="True"
  Width="200"
  Height="40"
  Grid.Row="2"
  Grid.Column="1"
  HorizontalAlignment="Left"
  Margin="5"
  VerticalAlignment="Center" />
</Grid>

```

We start by giving the Grid three rows and two columns, and then we populate the resulting cells with TextBlock and TextBox controls. Notice that the default position for a control is in `Grid.Row="0"` and `Grid.Cell="0"`, and so we only need to specify when we want a different row or cell. While it is best programming practice to specify the default as well, it is so common to assume 0 for a cell or row if not specified, that we show it that way here.

The result of this code is shown in [Figure 3-12](#).

Figure 3-12. Simple data entry form

The text boxes in the data entry form have an X that shows up on the right side once the users start typing, allowing them to delete the text and start over, as shown in Figure 3-13.

Figure 3-13. The X in text box for deleting

Also, notice in the XAML that the two `TextBox` controls are assigned names with the `x:Name` attribute:

```
<TextBox x:Name="FirstName"/>
```

This is so that we can refer to them programmatically. That is, we can refer to them in code, such as in an event handler, which we'll see in the next section.

Spell Check

Windows 10 UWP XAML supports system-driven spell checking and auto-complete, both built-in and right out of the box. Figure 3-14 shows spell checking at work. Notice the squiggly underlining of the misspelled word and the options to correct or add to the dictionary or ignore.

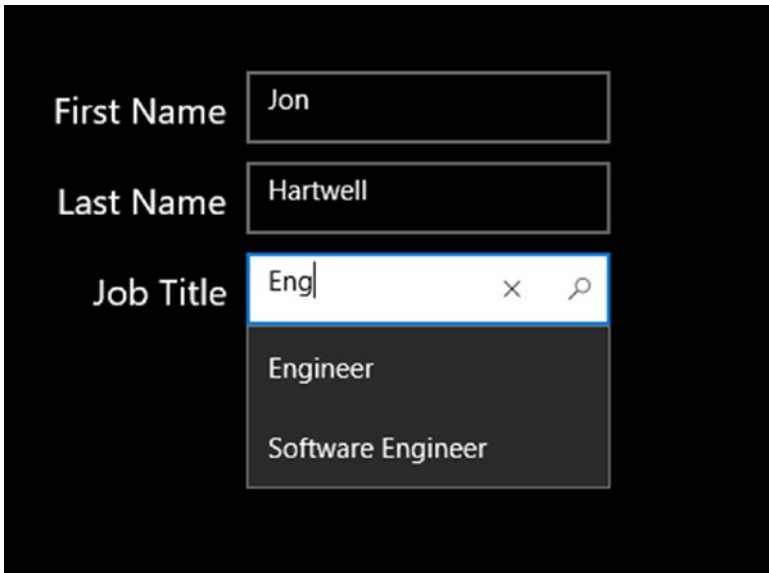


Figure 3-14. Spell check

Unlike Windows 8.1, where you had to explicitly specify you want to add spell check to your text box, in UWP all `TextBox` and `TextBlock` controls have spell check enabled by default. You can get auto-completion by setting just the following property on the `TextBox`:

```
IsTextPredictionEnabled="True"
```

The complete XAML for the Job Title text box is shown here:

```
<TextBox x:Name="JobTitle"
  IsTextPredictionEnabled="True"
  Width="200"
  Height="40"
  Grid.Row="2"
  Grid.Column="1"
  HorizontalAlignment="Left"
  Margin="5"
  VerticalAlignment="Center" />
```

One thing to note is that in Windows 10 UWP, the text prediction does not work on devices that have a physical keyboard. When using a mobile device, the text prediction will work and it uses the device's internal dictionary.

If you want to use text prediction on devices with a keyboard, you should use `AutoSuggestBox`. If you wanted to use the same example on a desktop, you would use the following XAML

```
<AutoSuggestBox x:Name="JobTitle"
    Width="200"
    Height="40"
    Grid.Row="3"
    Grid.Column="1"
    HorizontalAlignment="Left"
    Margin="5"
    VerticalAlignment="Center"
    PlaceholderText="Job Title"
    QueryIcon="Find"
    TextChanged="JobTitle_TextChanged"
    QuerySubmitted="JobTitle_QuerySubmitted"
    SuggestionChosen="JobTitle_SuggestionChosen"/>
```

Notice that there are a few differences here. First, you can use a query icon. That is not necessary but if you look at [Figure 3-14](#) you will notice the icon on the right of the Job Title text box. There are many options (or you can leave it blank if you want) for this icon and you are able to see them if you allow Visual Studio to use auto-complete for you.

Another difference is that there are three different event handlers that are attached to this control. The following code populates the Job Title suggest box.

```

public sealed partial class MainPage : Page
{
    private List<string> jobs;
    public MainPage()
    {
        this.InitializeComponent();
        jobs = new List<string>() { "Engineer", "Software
        Engineer" };
    }

    private void JobTitle_TextChanged(AutoSuggestBox
    sender, AutoSuggestBoxTextChangedEventArgs args)
    {
        if (args.Reason ==
        AutoSuggestionBoxTextChangeReason.UserInput)
        {
            sender.ItemsSource = jobs;
        }
    }

    private void JobTitle_QuerySubmitted(AutoSuggestBox
    sender, AutoSuggestBoxQuerySubmittedEventArgs args)
    {
        if(args.ChosenSuggestion != null)
        {
            // Do something with the text
            if(!jobs.Contains(args.ChosenSuggestion.
            ToString()))
            {
                jobs.Add(args.ChosenSuggestion.ToString());
            }
        }
    }
}

```

```

private void JobTitle_SuggestionChosen(AutoSuggestBox
sender, AutoSuggestBoxSuggestionChosenEventArgs args)
{
    sender.Text = args.SelectedItem.ToString();
}
}

```

The `TextChanged` event is used whenever the user enters text in the text box. This is where we set the source for the auto suggestions. The `QuerySubmitted` event is called when the user actually submits the text box. Right now, we don't do anything with the actual selected text except add it to our list of jobs if it does not already exist in our list. There could be many other things you do, such as compare to a dictionary or add to your custom dictionary. The `SuggestionChosen` event is fired when a user selects one of the drop-down options. In this case, we are setting the text on the `AutoSuggestBox` to what the user selected.

Headers and Watermarks Controls

In the previous examples, we used a two-column `Grid` and created text blocks next to the text boxes in order to create a data entry form. Just like in Windows 8.1, Windows 10 UWP apps allow us to create headers in place with many input controls, including the `TextBox`, the `PasswordBox`, and the `ComboBox` (all shown later in this chapter).

To demonstrate this, create a project called `Controls5a` and add the following row definitions into the `Grid`. The main difference between this `Grid` and the one we created in `Controls5` is that the height is set to "Auto" and column definitions are not included.

```

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

```



```

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

```

To create headers for the controls, add the `Header` attribute into the markup, as in the sample code shown here:

```

<TextBox x:Name="FirstName"
  Header="First Name"
  Width="200"
  Grid.Row="0"
  HorizontalAlignment="Left"
  Margin="5"
  VerticalAlignment="Center" />

```

To add a watermark, use `PlaceholderTextAttribute`, as shown in the following sample code:

```

<TextBox x:Name="FirstName"
  Header="First Name"
  PlaceholderText="[Enter First Name]"
  Width="200"
  Grid.Row="0"
  HorizontalAlignment="Left"
  Margin="5"
  VerticalAlignment="Center" />

```

The full code for the page is listed here:

```

<Grid
  Background="{StaticResource
ApplicationPageBackgroundThemeBrush}">
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto" />

```

```
<RowDefinition Height="Auto" />
<RowDefinition />
</Grid.RowDefinitions>
<TextBox x:Name="FirstName"
  Header="First Name"
  PlaceholderText="[Enter First Name]"
  Width="200"
  Grid.Row="0"
  HorizontalAlignment="Left"
  Margin="5"
  VerticalAlignment="Center" />
<TextBox x:Name="LastName"
  Header="Last Name"
  PlaceholderText="[Enter Last Name]"
  Width="200"
  Grid.Row="1"
  HorizontalAlignment="Left"
  Margin="5"
  VerticalAlignment="Center" />
<TextBox x:Name="JobTitle"
  Header="Job Title"
  PlaceholderText="[Enter Job Title]"
  IsSpellCheckEnabled="True"
  IsTextPredictionEnabled="True"
  Width="200"
  Grid.Row="2"
  HorizontalAlignment="Left"
  Margin="5"
  VerticalAlignment="Center" />
</Grid>
```

When you run the project, you get the result shown in Figure 3-15.

Figure 3-15. Password box

The PasswordBox Control

A variation on the `TextBox` control is the `PasswordBox`, which allows you to collect information from the user that is masked by a “password character”—which is by default a bullet (•). To demonstrate this, create a new project called `Controls5b`. Add a `PasswordBox` into `MainPage.xaml` and substitute a question mark as the password character. Also, turn on the `Reveal` button, which allows users to temporarily see the password in clear text. The XAML is as follows:

```
<Grid Background="{StaticResource
  ApplicationPageBackgroundThemeBrush}">
  <PasswordBox Margin="5"
    Width="200"
    Header="Password"
    PlaceholderText="Please Enter Your Password">
```

```

    IsPasswordRevealButtonEnabled="True"
    PasswordChar="?"
    VerticalAlignment="Top"/>
</Grid>

```

The before-and-after results of typing an entry are shown in Figure 3-16.

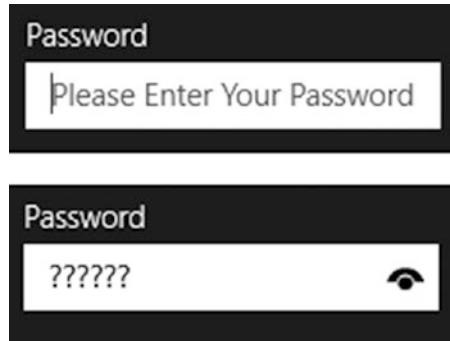


Figure 3-16. Password boxes waiting for input (top) and with input (bottom)

The “eye” on the right side of the password box is the Reveal button. It allows users to see the actual characters being typed. It is not a toggle, as the characters are only revealed while the Reveal button is pressed.

Buttons and Event Handlers Controls

Create a new project named Controls6. The Main Page is a form with two TextBox instances, a Button, and an event handler that will respond to the button being clicked. In that event handler, we can get the values in the two TextBox instances. The result will be displayed in a text block.

```

<Grid Background="{StaticResource
  ApplicationPageBackgroundThemeBrush}">
  <Grid.RowDefinitions>
    <RowDefinition Height="50" />

```

```
<RowDefinition Height="50" />
<RowDefinition Height="50" />
<RowDefinition />
</Grid.RowDefinitions>
<Grid.ColumnDefinitions>
  <ColumnDefinition />
  <ColumnDefinition />
</Grid.ColumnDefinitions>
<TextBlock Text="First Name"
  FontSize="20"
  Margin="5"
  HorizontalAlignment="Right"
  VerticalAlignment="Center" />
<TextBox x:Name="FirstName"
  Width="200"
  Height="40"
  Grid.Row="0"
  Grid.Column="1"
  HorizontalAlignment="Left"
  Margin="5"
  VerticalAlignment="Center" />
<TextBlock Text="Last Name"
  Grid.Row="1"
  FontSize="20"
  Margin="5"
  HorizontalAlignment="Right"
  VerticalAlignment="Center" />
<TextBox x:Name="LastName"
  Width="200"
  Height="40"
  Grid.Row="1"
  Grid.Column="1"
```

```

        HorizontalAlignment="Left"
        Margin="5"
        VerticalAlignment="Center" />
<TextBlock x:Name="Output"
    HorizontalAlignment="Right"
    VerticalAlignment="Center"
    Text=""
    Margin="5"
    FontSize="20"
    Grid.Column="0"
    Grid.Row="2" />
<Button Name="ShowName"
    Content="Show Name"
    HorizontalAlignment="Left"
    VerticalAlignment="Center"
    Margin="5"
    Grid.Row="2"
    Grid.Column="1" />
</Grid>

```

There are numerous ways to add an event handler for the click event. The simplest is to type `Click=`, and when you press space Visual Studio will offer to create (and name) an event handler for you, as shown in Figure 3-17.



Figure 3-17. Event handler

The name created for you by Visual Studio is `<objectName>_Click`, so if your button is named `ShowName` then the event handler will be `ShowName_Click`, although you are free to override this name with anything you like.

Not only does Visual Studio set up the name for your event handler but it also stubs out the event handler in the code-behind (e.g., `MainPage.xaml.cs`).

```
private void ShowName_Click( object sender, RoutedEventArgs e )
{
}
```

All you need to do is fill in the logic. In this case, we'll obtain the string values from the `Text` properties of the `FirstName` and `LastName` text boxes, concatenate them, and then place them into the `Text` property of the `TextBlock Output`, as follows:

```
private void ShowName_Click( object sender, RoutedEventArgs e )
{
    string fn = FirstName.Text;
    string ln = LastName.Text;
    Output.Text = fn + " " + ln;
}
```

You can of course shorten this by leaving out the intermediate string variables:

```
private void ShowName_Click( object sender, RoutedEventArgs e )
{
    Output.Text = FirstName.Text + " " + LastName.Text;
}
```

We will typically show the longer version, as it makes it easier to see what is happening and to debug it if anything goes wrong.

The HyperLinkButton Control

The `HyperLinkButton` control looks like a `HyperLink` but acts like a button. That is, you can assign a click event handler to the `HyperLinkButton` that will be handled before the user navigates to the URL represented by the `HyperLinkButton`. `HyperLinkButtons` are often used inline with text blocks.

Create a new project names `Controls6a` and add the following XAML to show a simple `HyperLinkButton` at work:

```
<Grid Background="{StaticResource
  ApplicationPageBackgroundThemeBrush}">
  <StackPanel Margin="50">
    <TextBlock Name="Message" />
    <HyperlinkButton Content="Phil Japikse"
      NavigateUri="http://skimed.com"
      Click="HyperlinkButton_Click" />
  </StackPanel>
</Grid>
```

Notice that the `HyperLinkButton` has an event handler. This method is implemented in the code-behind, like any event handler.

```
private void HyperlinkButton_Click( object sender,
RoutedEventArgs e )
{
  Message.Text = "Hello Hyperlink!";
}
```

The code will be called before the hyperlink is navigated to, and so in this case the message "Hello Hyperlink!" will flash just before we navigate to the URL. While you can write code that requires user interaction to take place before the URL is navigated to, it is more common to use the `HyperLinkButton` just to navigate to the URL. This gives you the appearance of a normal (HTML) hyperlink within an XAML application.

The CheckBoxes, ToggleSwitches, and RadioButton Controls

There are a number of controls that help the users make selections. Two of the most popular that allow the users to select from a number of options are `CheckBox` and `RadioButton`. `ToggleSwitch` is the Windows 10 UWP control of choice when the user is selecting one of two mutually exclusive choices. `Controls7` shows a small form with all three of these controls added:

```
<Grid Background="{StaticResource
  ApplicationPageBackgroundThemeBrush}">
  <StackPanel Name="OuterPanel" Margin="50">
    <StackPanel Orientation="Horizontal" Name="RadioButtonsPanel">
      <RadioButton Name="Soft"
        Content="Soft"
        GroupName="Loudness"
        Margin="5" />
      <RadioButton Name="Medium"
        Content="Medium"
        GroupName="Loudness"
        Margin="5" />
      <RadioButton Name="Loud"
        Content="Loud"
        IsChecked="True"
        GroupName="Loudness"
        Margin="5" />
    </StackPanel>
    <StackPanel Orientation="Horizontal"
      Name="CheckBoxPanel">
      <CheckBox Name="ClassicRock"
        Content="Classic Rock"
        Margin="5" />
    </StackPanel>
  </StackPanel>
</Grid>
```

```

    <CheckBox Name="ProgRock"
      Content="Progressive Rock"
      Margin="5" />
    <CheckBox Name="IndieRock"
      Content="Indie Rock"
      Margin="5" />
  </StackPanel>
  <ToggleSwitch Header="Power" OnContent="On"
    OffContent="Off" />
  <ToggleButton Content="Toggle me!"
    Checked="ToggleButton_Checked"
    Unchecked="ToggleButton_Unchecked"/>
  <TextBlock Name="Message"
    Text="Ready..."
    FontSize="40" />
</StackPanel>
</Grid>

```

The first thing to notice about this form is that rather than creating the layout with cells in a `Grid`, we laid out the controls in a `StackPanel`. Either way is valid.

The first `StackPanel` control consists of three `RadioButton` controls. `RadioButton` controls are grouped using a `GroupName` (every `RadioButton` in a group is mutually exclusive). We set the `IsChecked` property to `true` for the third button—the default is `false`, and in a `RadioButton` group only one button will have the `IsChecked` value set to `true` at a time.

The second `StackPanel` has three `CheckBox` controls and they are not mutually exclusive; the user is free to pick any or all of the music choices.

Next, we have a `ToggleSwitch`. You can see that it has a header and text for when it is in the on or off position.

Finally, we have a `ToggleButton`, which looks like a button but toggles between a checked and unchecked state and has events that correspond to being checked or unchecked.

Run the program and click on the buttons and check boxes to see how they behave, then toggle the switch to see the text and the switch change, as shown in Figure 3-18.



Figure 3-18. Radio buttons, check boxes, a toggle switch, and a toggle button

The only code for all of the previous controls are the event handlers for clicking the Toggle button, as shown here:

```
private void ToggleButton_Checked( object sender,
RoutedEventArgs e )
{
    Message.Text = "Button was toggled on!";
}
```

```
private void ToggleButton_Unchecked( object sender,
RoutedEventArgs e )
{
    Message.Text = "Button was toggled off!";
}
```

The ListBox, ListView, and ComboBox Controls

Windows 10 UWP continues to support the `ListBox` control, though it has, in many ways, been replaced by `ListView`. They work very similarly, and they both provide a scrolling list of data. The `ListView` is more powerful and more suited to Windows 8.1 Store applications. In their simplest forms, the use and appearance is identical, as shown in Controls8:

```
<Grid Background="{StaticResource
ApplicationPageBackgroundThemeBrush}">
<StackPanel Orientation="Horizontal" Margin="50">
<ListBox Name="myListBox"
    Background="White"
    Foreground="Black"
    Width="150"
    Height="250"
    Margin="5">
<x:String>ListBox Item 1</x:String>
<x:String>ListBox Item 2</x:String>
<x:String>ListBox Item 3</x:String>
<x:String>ListBox Item 4</x:String>
</ListBox>
```

```

<ListView Name="myListView"
  Background="White"
  Foreground="Black"
  Width="150"
  Height="250"
  Margin="5">
  <x:String>ListView Item 1</x:String>
  <x:String>ListView Item 2</x:String>
  <x:String>ListView Item 3</x:String>
  <x:String>ListView Item 4</x:String>
</ListView>
</StackPanel>
</Grid>

```

The output is nearly identical in this case, as shown in Figure 3-19.

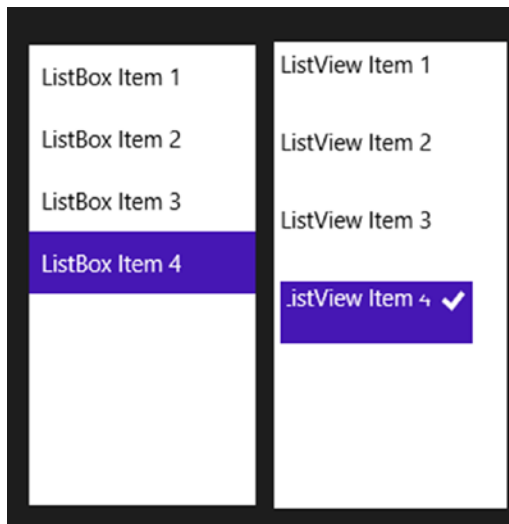


Figure 3-19. *ListBox and ListView controls*

The `ComboBox` is very similar to the `ListBox`. It provides a drop-down list, but you can also provide watermark text and a header (through the `PlaceholderText` and `Header` attributes, respectively), as shown in `Controls8a`:

```
<ComboBox Name="myComboBox"
  Background="White"
  Foreground="Black"
  Width="Auto"
  Height="60"
  Margin="5"
  PlaceholderText="Please Select an Item"
  Header="ComboBox Example">
  <x:String>ComboBox Item 1</x:String>
  <x:String>ComboBox Item 2</x:String>
  <x:String>ComboBox Item 3</x:String>
  <x:String>ComboBox Item 4</x:String>
</ComboBox>
```

The result, with all three controls, is shown in [Figure 3-20](#).

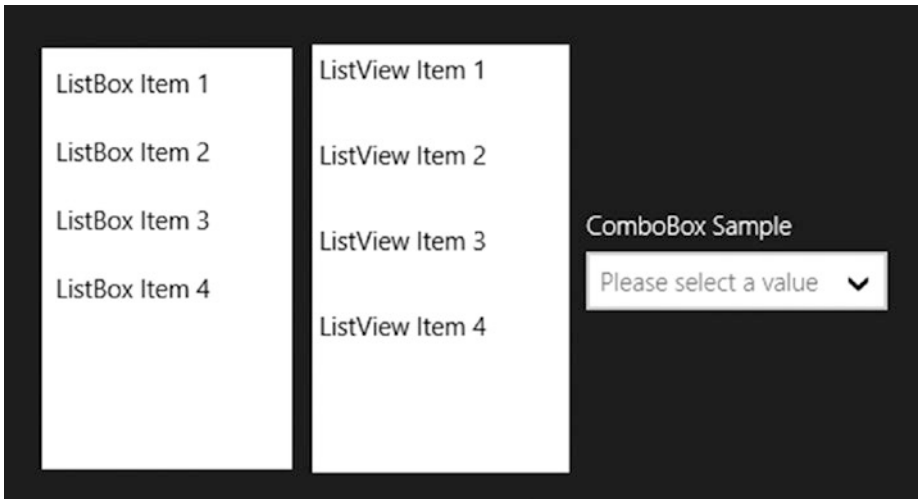


Figure 3-20. *ListBox and ListView controls with a ComboBox control*

Typically, you'll use the ComboBox when space is tight and when users select just one choice at a time. You'll use the ListBox (if you use it at all) when you want to display all the choices at once and when you want to allow multiple selections.

The Image Control

The Image control is used to display (surprise!) an image, typically a jpg or png file. Its most important property is `Source`, which is the URI of the image itself. `Controls9` shows how to add two Image controls to the page:

```
<Grid Background="{StaticResource
  ApplicationPageBackgroundThemeBrush}">
  <StackPanel Margin="50">
    <Image Height="240"
      Width="360"
      Source="Assets/Sheep.jpg"
      Margin="5" />
```

```
<Image Height="240"  
  Width="360"  
  Source="Assets/PaintedDesert.jpg"  
  Margin="5" />  
</StackPanel>  
</Grid>
```

The result is shown in Figure 3-21.



Figure 3-21. Images presented with the image controls

To add the images to your page, copy two images into your Assets folder, click on Add Existing Item to add them to the project, and then update the name in the Source property of the Image. You can use any images that you have on your local system or use the images from the sample code provided along with the book.

If your image does not fit the rectangle described for it by the Image control, you can set the Stretch property, which is an enumerated constant, as shown in Figure 3-22.

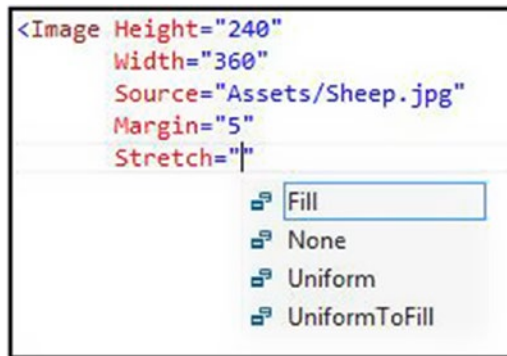


Figure 3-22. Stretch property

The effect of each of these constants can be seen in Figure 3-23.

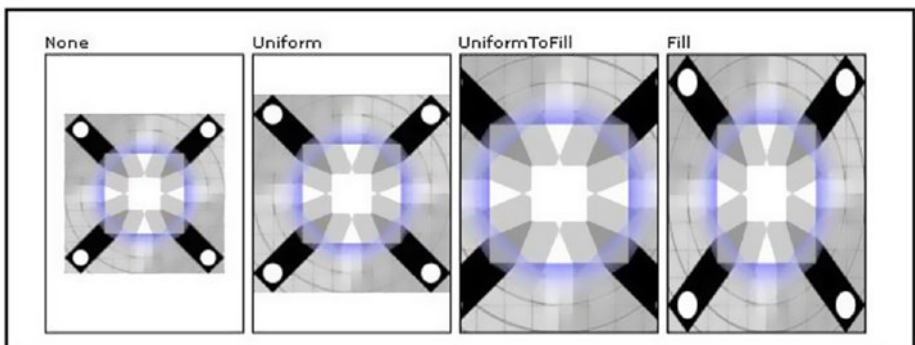


Figure 3-23. Stretch settings in action

The Slider Control

The `Slider` control allows the user to select from a range of values by sliding a thumb control along a track. While the thumb is being moved, the actual value is automatically displayed above the slider, as shown in Figure 3-24.

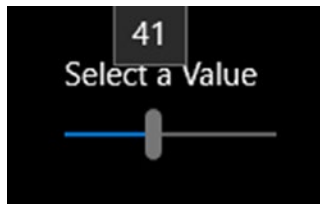


Figure 3-24. *Slider control*

The code for creating the slider is shown in `Controls10`:

```
<StackPanel Margin="50">
  <Slider
    Header="Select a Value"
    Minimum="0"
    Maximum="100"
    Value="50"
    Width="100"/>
</StackPanel>
```

The ProgressBar Control

The `ProgressBar` control is used to indicate progress, typically while waiting for an asynchronous operation to complete. If you know the percentage of progress achieved at any moment, you can use a standard

progress bar, setting its value as you progress toward completion. Otherwise, you will want to use the indeterminate progress bar.

A `ProgressRing` works just like the indeterminate `ProgressBar` but is typically used to indicate waiting for the system rather than for program progress.

The code for creating progress bars is shown in `Controls11`:

```
<StackPanel Margin="50">
  <ProgressBar Value="50" Width="150" Margin="20"/>
  <ProgressBar IsIndeterminate="True"
    Width="150" Margin="20" />
  <ProgressRing IsActive="True" Margin="20" />
</StackPanel>
```

The result is shown in [Figure 3-25](#).

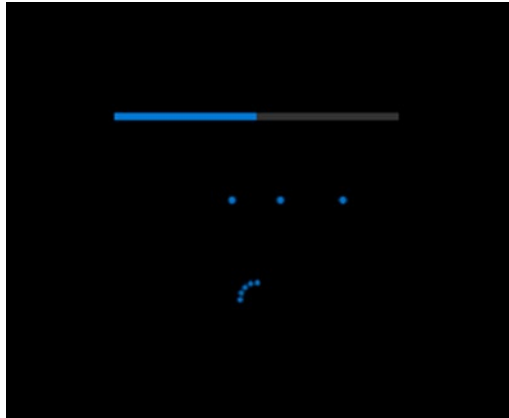


Figure 3-25. A progress bar

The ToolTip Control

Tooltips pop up and display information associated with an element. A tooltip will appear when users hover over the element with a mouse or tap and hold the element with their finger. Controls12 shows the code for adding a ToolTip control to a button:

```
<Grid Background="{StaticResource
  ApplicationPageBackgroundThemeBrush}">
  <StackPanel Margin="50">
    <Button Content="Button"
      ToolTipService.ToolTip="Click to perform action" />
  </StackPanel>
</Grid>
```

The tooltip is shown in Figure 3-26.

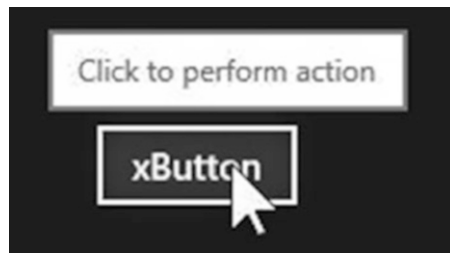


Figure 3-26. *Tooltip*

The DatePicker and TimePicker Controls

Windows 10 UWP allows you to add date and time pickers for XAML-based applications. To do so, create a new project called Controls13 and add the following XAML to demonstrate the core functionality of the controls:

```
<Grid.ColumnDefinitions>
  <ColumnDefinition Width="120"/>
```

```
<ColumnDefinition Width="Auto"/>
<ColumnDefinition Width="40"/>
<ColumnDefinition Width="Auto"/>
<ColumnDefinition Width="*/>
</Grid.ColumnDefinitions>
<TimePicker Grid.Column="1" Header="Select a Time" />
<DatePicker Grid.Column="3" Header="Select a Date"/>
```

The controls are shown in Figure 3-27.

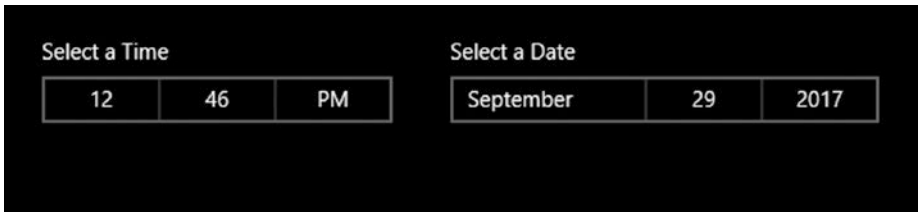


Figure 3-27. *TimePicker and DatePicker controls*

You can see the TimePicker control in action in Figure 3-28.

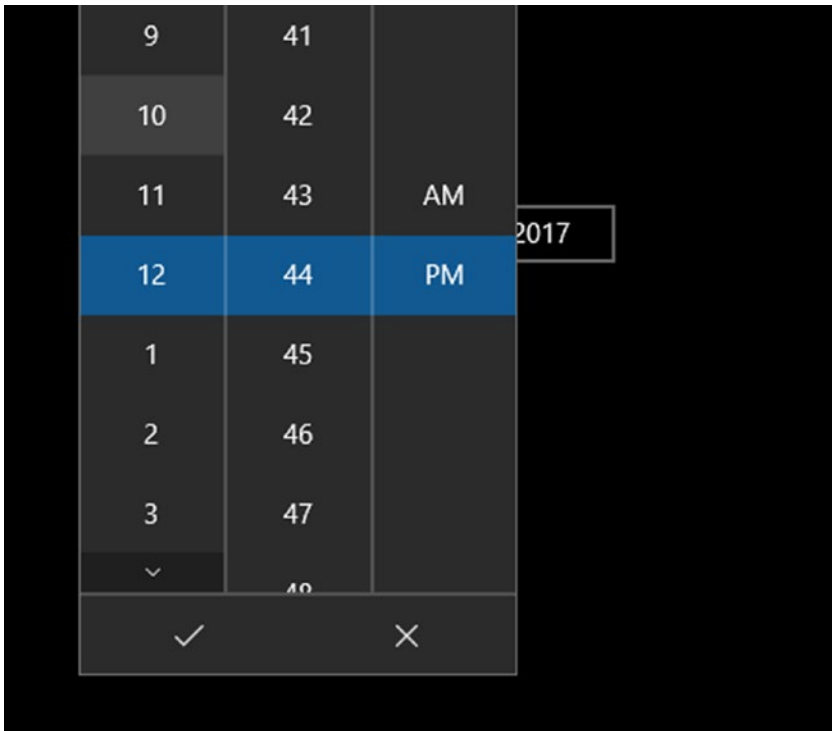


Figure 3-28. Using the TimePicker control

Flyouts

Flyouts have been available for HTML/JavaScript developers since Windows 8. Windows 8.1 adds flyouts for XAML developers. A flyout is an UI element that overlays the current UI to provide additional information to the user or provide a mechanism to get confirmation of an action.

Flyouts are attached to other controls and are shown in response to an action. For example, a flyout attached to a button will show when the button is clicked. Flyouts are container controls themselves and can hold any number of additional controls by using a panel.

To show the different types of flyouts, create another project called Controls14. Add the following column definitions in preparation for the different examples:

```
<Grid.ColumnDefinitions>
  <ColumnDefinition Width="120"/>
  <ColumnDefinition Width="Auto"/>
  <ColumnDefinition Width="Auto"/>
</Grid.ColumnDefinitions>
```

Basic Flyouts

To see how to add a flyout, add the following XAML to the Grid, run the program, and you will see the result shown in Figure 3-29 (after clicking the button).

```
<Button Grid.Column="0" Content="This is a button to launch a
flyout">
  <Button.Flyout>
    <Flyout>
      <StackPanel>
        <TextBlock>This is a flyout message</TextBlock>
        <Button>This is a flyout button</Button>
      </StackPanel>
    </Flyout>
  </Button.Flyout>
</Button>
```

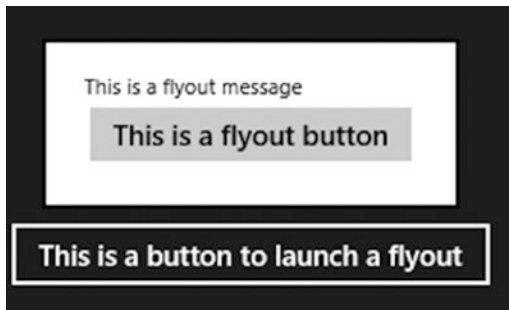


Figure 3-29. Flyout attached to a button

Menu Flyouts

Menu flyouts allow for building menus that remain hidden until needed. Add the following XAML into Controls14 and run the program to get the result shown in Figure 3-30.

```
<Button Grid.Column="1"
  Content="This is a button to launch Menu Flyout">
  <Button.Flyout>
    <MenuFlyout>
      <MenuFlyoutItem Text="First Item"/>
      <MenuFlyoutSeparator/>
      <ToggleMenuFlyoutItem Text="Toggle Item"/>
    </MenuFlyout>
  </Button.Flyout>
</Button>
```

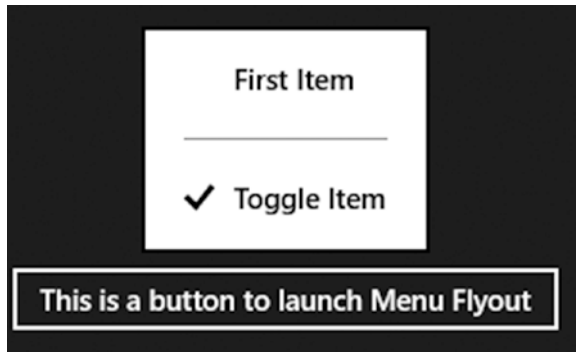



Figure 3-30. *Menu flyout with a regular menu item and a toggle menu item*

Understanding Dependency Properties

Dependency properties are like the electric wires and pipes running under the streets of a big city. You can ignore them, you can even be oblivious to their existence, but they make all the magic happen and when something breaks, you suddenly discover that you need to understand how they work.

Before we begin exploring dependency properties, let's review properties themselves. To do that, we go back into the dark early days of C++ when what we had available in a class were methods and member variables (fields).

Data Hiding

Back in those dark days, we wanted to store values in fields but we didn't want clients (methods that use our values) to have direct access to those fields (if nothing else, we thought we might later change how we store the data). Thus, we used accessor methods.

The accessor methods had method semantics (you used parentheses), and so data hiding was explicit and ugly. You ended up with code that looked like this:

```
int theAge = myObject.GetAge();
```

which just seemed all wrong when what you wanted was

```
int theAge = myObject.age;
```

Enter properties. Properties have the wonderful characteristic of looking like a field to the consumer but looking like a method to the implementer. Now you can change the way the property is “backed” without breaking the code that uses the property.

```
private int _age;
public int Age { get { return _age; } set { _age = value; } }
```

Age is the property; `_age` is the backing variable. The consumer can now write

```
int theAge = myObject.Age;
```

Age is a property and so you can put anything you like in the getter, including accessing a database or computing the age.

If your property getters and setters only return and set the value (in other words, no additional logic), you can use automatic properties to save typing. The private field and public property we looked at could have been written with this single line:

```
public int Age { get; set; }
```

The compiler turns this single statement into a declaration of a private backing variable and a public property whose `get` returns that backing variable, and whose `set` sets that variable.

Dependency Properties

Dependency properties are an extension to the CLR and to C# and were created because normal properties just don't provide what we need: *declarative syntax that supports data binding as well as storyboards and animations.*

Most of the properties exposed by Windows 10 elements are dependency properties, and you've been using these properties without realizing it. That is possible because dependency properties are designed to look and feel like traditional C# properties.

In short, what was needed was a system that could establish the value of a property at runtime based on input from a number of sources. The value of a dependency property is computed based on inputs such as

- User preferences
- Data binding
- Animation and storyboards
- Templates and styles
- Inheritance

A key value of the dependency properties system is the ability to build properties that automatically notify any interested party that is registered each time the value of the property changes. This free, painless, and automatic implementation of the observer pattern¹ is tremendously powerful and greatly reduces the burden on the client programmer (in fact, the data-binding system depends on it!).

You normally will not have to create dependency properties unless you are creating a custom control (a topic beyond the scope of this book).

¹See "Exploring the Observer Design Pattern" on the Microsoft Developer Network for more information: [http://msdn.microsoft.com/en-us/library/Ee817669\(pandp.10\).aspx](http://msdn.microsoft.com/en-us/library/Ee817669(pandp.10).aspx).

The first thing to know is that in order to support a dependency property, the object that defines the property (your custom control) must inherit from the `DependencyObject`. Virtually all of the types you use for a Windows Store app with XAML and C# will be a `DependencyObject` subclass.

You might then declare your property like this:

```
public bool Valuable
{
    get { return (bool) GetValue( ValuableProperty ); }
    set { SetValue( ValuableProperty, value ); }
}
```

It's a pretty standard get and set except that you access your backing variable using `GetValue` and `SetValue` to get and set the value of a dependency property named `ValuableProperty` (and that is the idiom, the CLR property name plus the word *property* equals the name of the dependency property, thus `Valuable` plus `Property` equals `ValuableProperty`).

The declaration of the dependency property itself is a bit weirder:

```
public static readonly DependencyProperty ValuableProperty =
    DependencyProperty.Register(
        "Valuable",
        typeof( bool ),
        typeof( MyCustomControl ),
        new PropertyMetadata( new PropertyChangedCallback(
            MyCustomControl.OnValuablePropertyChanged ) ) );
```

Let's break this down. The first line declares the object (which is really a reference to a dependency property) as public. It must be static and read-only, its type is `DependencyProperty`, and its name (identifier) is `ValuableProperty`.

We set that reference to what we'll get back by calling the static `Register` method on the `DependencyProperty` class. `Register` takes four arguments:

- The name of the dependency property wrapper
- The type of the dependency property being registered
- The type of the object registering it
- The `Callback`

The `Callback` is of type `PropertyMetadata`. You can imagine a world in which there are various pieces of metadata for the `DependencyProperty`. At the moment, however, in Windows 10 UWP, there is only one: the `Callback`.

The constructor for the `PropertyMetadata` takes an object of type `PropertyChangedCallback` that will be called any time the effective property value of the dependency property changes. We pass it a reference to the method to call (which equates to a `Callback`).

The net of all of this is that we present to the world a CLR property (`Valuable`) that is in fact backed by a `DependencyProperty` that will call back to the method `OnValuablePropertyChanged` any time the effective value of the property changes.

The `Callback` method will take two arguments:

- A `DependencyObject` control (the control)
- An object of type `DependencyPropertyChangedEventArgs`

Typically, you'll cast the first argument to be the type of the control that contains the property, and you'll cast the `NewValue` property of the `DependencyPropertyChangedEventArgs` object to the `DependencyProperty` that changed. You can then take whatever action you need to based on the change in the dependency property's value

```

public class MyCustomControl : Control
{
    public static readonly DependencyProperty
        ValuableProperty = DependencyProperty.Register(
            "Valuable",
            typeof( bool ),
            typeof( MyCustomControl ),
            new PropertyMetadata( new PropertyChangedCallback(
                MyCustomControl.OnValuablePropertyChanged ) ) );
    public bool Valuable
    {
        get { return (bool) GetValue( ValuableProperty );}
        set { setValue( ValuableProperty, value );}
    }
    private static void OnValuablePropertyChanged(
        DependencyObject d,
        DependencyPropertyChangedEventArgs e )
    {
        MyCustomControl control = d as MyCustomControl;
        bool b = (bool) e.NewValue;
    }
}

```

The key thing is that when you bind to a property (either through data binding or element binding), it must be a dependency property. Similarly, if you are going to use animations (for example, to change the opacity of a control or the style based on values), the animation storyboard only works on dependency properties. This is normally not a problem, as the properties you will be inclined to animate or bind to are (almost always) dependency properties anyway. You can see that quickly by looking at the

documentation for the properties for any of the UI elements. These will all be UI elements (or the `FrameworkElement`, which derives from `UIElement`) and `UIElement` derives from the `DependencyObject`, which provides the support for dependency properties.

Summary

There are a wide variety of controls available for use in your Windows 10 UWP apps that ship with Visual Studio, plus many more available through the rich third-party ecosystem. Every page starts with a `Panel` control (the most common being the `Grid`), and then additional controls are added to make the desired UI.

Using controls and handling events for those controls is very similar to what you find in other XAML environments such as WPF and Silverlight. This chapter did not cover every type of control available, but provides information about the core controls. The remaining controls are all conceptual extensions of the ones described.

CHAPTER 4

Binding

The *target* of the binding must be a dependency property on a UI element; the source can be any property of any object. That is a very powerful statement; it means that you don't have to create anything special in the source. A simple class such as a POCO (Plain Old CLR Object) will do. The classes that represent the data are typically referred to as *models*.

The essential glue in any meaningful Windows UWP application is showing data in your UI. At its simplest, binding allows you to connect the value of one object to a property on another object. This connection is referred to as a *binding*. The most common use is binding the properties of a data object (a .NET class) to the controls in the UI. For example, a customer's information might be bound to a series of text boxes. This use case is so prevalent that binding is commonly referred to as data binding.

It is important to note that *any* dependency property on the control can be bound to any accessible property on the data source. We'll see an example of this later in this chapter.

Note See Chapter 3 for more information on dependency properties.

DataContext

As stated earlier, data binding needs a model to supply the data and a dependencyproperty to receive the value of one of the model's properties. If the binding statement specifies the information directly, the binding engine sends the data value(s) from the model to the control. However, when multiple controls are bound to the model, *a lot* of typing is required, as each binding statement must provide the model's information. In addition to the additional work required upfront to bind the UI, if anything changes about the model (or if the model must be exchanged for another one), you must update each control with the new information.

If the binding statement does not contain information that specifies how to find the model, the binding engine will use the DataContext. If the control does not have a DataContext specified, the binding engine will walk up the control tree until it finds a DataContext. Once the binding engine finds a control in the UIElement tree that has a DataContext specified, the binding engine will use that definition in the binding statements for all of the controls contained by that element.

Using a DataContext instead of explicitly referencing a model in each of the binding statements leads to a much more supportable UI. It also greatly enhances the ability to change the data source during runtime with a single statement.

Creating a Simple Binding

The best way to move from theory to practice is to write some simple code as follows: Create a new Windows 10 UWP project by selecting the Blank App (Universal Windows) and naming it Binding1. Add a new class to the

project named `Person`. This class will become the model to hold the data and will be bound to the UI. The `Person` class is shown in the following:

```
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

Figure 4-1 shows a prompt with the output we want to display and the value of the first and last name.

```
First Name: Jon
Last Name: Hartwell
```

Figure 4-1. *First name and last name*

We certainly could create this by adding four text blocks like this:

```
<StackPanel>
    <StackPanel Orientation="Horizontal">
        <TextBlock Text="First Name: " Margin="5"/>
        <TextBlock Name="txtFirstName" Margin="5"/>
    </StackPanel>
    <StackPanel Orientation="Horizontal">
        <TextBlock Text="Last Name: " Margin="5"/>
        <TextBlock Name="txtLastName" Margin="5"/>
    </StackPanel>
</StackPanel>
```

We would then populate the text blocks by writing the value from the Person instance into the Text property of the text block:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    txtFirstName.Text = person.FirstName;
    txtLastName.Text = person.LastName;
}
```

The OnNavigated event is raised when a XAML page loads into view. It will be covered later in the book when we talk about navigation. It is used to set up the state for each page.

While this works and is perfectly valid code, it is inflexible and incredibly tedious when working with collections. Data binding is much more powerful, more flexible, easier to maintain, and generally more desirable.

To change the controls to data bind to the model, start by changing the XAML as follows:

```
<StackPanel>
    <StackPanel Orientation="Horizontal">
        <TextBlock Text="First Name: " Margin="5"/>
        <TextBlock Text="{Binding FirstName}" Margin="5"/>
    </StackPanel>
    <StackPanel Orientation="Horizontal">
        <TextBlock Text="Last Name: " Margin="5"/>
        <TextBlock Text="{Binding LastName}" Margin="5"/>
    </StackPanel>
</StackPanel>
```

The data-binding statements are contained as strings enclosed with curly braces and start with the word *binding*. In these examples, the Text properties of text blocks are bound to the FirstName and LastName properties, respectively. This is much better; if nothing else, we've moved

from procedural to declarative code, which is more easily manipulated in tools such as Blend or Visual Studio. We'll look deeper into the binding syntax in the next section, but for now, this amount of description is adequate.

Only the properties on the model are specified, not the model. This causes the data-binding engine to next check the `TextBlock` for a `DataContext`. Not finding one, it would then check the two `StackPanel`s (starting with the inner `StackPanel`), finally moving on to the window itself.

The `DataContext` can be set programmatically (in code) or declaratively (in XAML). In this example, I'll set the `DataContext` at the page level, and I'll do so programmatically by setting the `DataContext` of the class (which is the page itself). The page is the highest-level container, so all of the elements on the page will inherit the `Person` class as the `DataContext`. To do this, replace the assignments to the text blocks in the `OnNavigatedTo` event to the following code:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    DataContext = person;
}
```

When the application runs, the `DataContext` is set to the `Person` instance, and the values are filled into the text blocks.

Data-Binding Statements

As mentioned earlier, a declarative binding statement in its simplest form is as follows:

```
<dependencyobject dependencyproperty="{Binding bindingArgs}" />
```

The `bindingArgs` are a set of name-value pairs that further refine the binding. There are a lot more options to explore, but here are some of the most commonly used binding arguments:

Path. The path defines the property from the model that will be data bound to the dependency property. It can be a simple property or a complex property chain in dot notation, walking down the properties of the model, such as `Address.StreetName`. If there is only one argument and that argument is not a binding property, the binding engine will use that as the path. For example, these two statements are equivalent:

```
{Binding Path=Address.Streetname}
{Binding Address.Streetname}
```

Source. The `Source` property specifies the model that the binding statement will use to get the data. This is unnecessary when the `DataContext` is set.

FallbackValue. The `Binding.FallbackValue` provides a value to display in the control when the binding fails.

TargetNullValue. The `TargetNullValue` provides a value to display in the control when the bound value is `null`.

ElementName. XAML allows a control to bind one of its properties to the property of another control. This is commonly called element binding. Instead of using a data context, the binding is set to the name of the element.

Mode. The binding mode specifies whether the binding is `OneWay`, `TwoWay`, or `OneTime`. If `Mode` is not specified, it will be set to `OneWay`.

UpdateSourceTrigger. The `UpdateSourceTrigger` determines when an update will be triggered in two-way binding situations. The possible values are `Default`, `Explicit`, and `PropertyChanged`.

Converter. Value converters enable binding of disparate types, such as changing the color of an element based on the numeric value of a property on the model.

Binding Errors

If you haven't worked in XAML before, there is a very important fact about data binding that you should know: When a binding statement fails, nothing happens. That's right, the user doesn't get any notification and there aren't any error dialogs—it's just a silent failure.

On one hand, this is very good. In the days of Windows Forms, when there was a binding error, the user would get a modal dialog box. This dialog came from the framework and could be very disruptive. Especially if the binding problem was in a grid, the user would get a steady stream of modal dialog boxes.

Data-binding errors in the XAML world are very different. Nothing is presented to the user (unless you purposely add that interaction), and the default behavior is that the error will *only* be displayed in one place—in the output window of Visual Studio. To illustrate this, update the `Binding` statement for the `LastName` property to this:

```
<TextBlock Text="{Binding LastName}" Margin="5"/>
```

When you run the app, look in the Output window in Visual Studio. There will be a statement similar to the one that follows (it is actually a lot longer and might differ slightly on your machine):

```
Error: BindingExpression path error: 'LastName1' property not found on 'Binding1.Person, Binding1,...
```

Note If you don't have the Output window open, you can find it on the Debug menu under Windows ► Output.

There are two features in data binding specifically for problem situations: The `FallbackValue` provides a safety valve if the binding is in error, and the `TargetNullValue` provides a value to display if the source is `null`.

FallbackValue

The `FallBackValue` gives us the ability to specify a value to be shown if the binding fails. To specify a fallback value, update the `Binding` statement to the following:

```
<TextBlock Text="{Binding LastName1, FallbackValue='Doe'}"
Margin="5"/>
```

Now, when you run the app, the result will be like [Figure 4-2](#).

First Name: Jon

Last Name: Doe

Figure 4-2. Window using `FallbackValue`

It is important to note that the control is not data bound. It is essentially showing a watermark.

TargetNullValue

Another binding option allows you to specify what to show when the bound property is null. To illustrate this, correct the Binding statement to once again bind to LastName and add the TargetNullValue:

```
"{Binding LastName, FallbackValue='Doe',
TargetNullValue='Unknown'}"
```

The next change is to alter the creation of the Person class so that it does not provide a last name.

```
public Person person = new Person { FirstName = "Jon"};
```

When you run the project, you get the result shown in Figure 4-3.

```
First Name: Jon
Last Name: Unknown
```

Figure 4-3. *TargetNullValue*

Just like the FallbackValue, the value shown in the UI is window dressing. It does not affect the value of the model.

Binding to Elements

In addition to binding to objects such as the Person object, you are also free to bind to the value of other controls. For example, you might add a slider to your page and a TextBlock and bind the Text property of the TextBlock to the Value property of the Slider.

To see this, create a new project and name it Binding2. Add the following XAML:

```
<StackPanel Orientation="Horizontal" Margin="100">
  <Slider Name="MySlider"
    Minimum="0"
    Maximum="100"
    Value="50"
    Width="300"
    Margin="10" />
  <TextBlock Margin="10"
    Text="{Binding ElementName=MySlider, Path=Value}"
    FontSize="42" />
</StackPanel>
```

That's all you need; no code required. You've bound the text block to the slider. As the slider changes value, the value in the text block will be updated, as shown in Figure 4-4.



Figure 4-4. Element binding

You might be thinking at this point: “That’s interesting. Why would I use that?” Element binding is a very powerful technique that can make your application’s user interface much more interactive.

A common example is enabling or disabling controls based on the state of other controls. Figure 4-5 is an example where users must check a box to accept the conditions (perhaps an End User License Agreement) before they continue. Selecting the check box should enable the button so the user can continue.

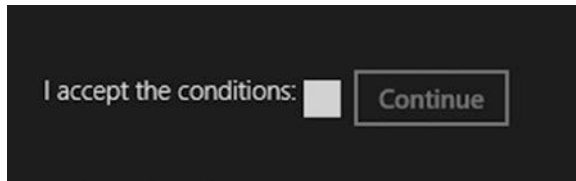


Figure 4-5. *Accept/Continue button appearing after controls have been disabled*

This can certainly be done with code, but it can also be done entirely in markup by leveraging element binding. Simply binding the `IsEnabled` property of the button to the `IsChecked` property of the check box links them together. The Binding statement is written like this:

```
IsEnabled="{Binding ElementName=MyCheckBox,Path=IsChecked}"
```

When the check box is checked, the button becomes enabled, as shown in Figure 4-6.



Figure 4-6. *Button that is enabled after the check box is clicked is checked*

The entire XAML of this example is shown in the following:

```
<StackPanel Orientation="Horizontal">
  <CheckBox Name="MyCheckBox"
    Content="I accept the conditions:"
    FlowDirection="RightToLeft" Margin="5"/>
  <Button Content="Continue"
    IsEnabled="{Binding ElementName=MyCheckBox,Path
    =IsChecked}"/>
</StackPanel>
```

Binding Modes

The binding mode sets the direction(s) in which binding takes place. Binding comes in three “modes”:

- OneWay
- TwoWay
- OneTime

Specifying the mode in a binding statement is optional. If it’s not specified, the mode is set to the default value of `OneWay`. The `OneTime` mode sets the value of the target when the window first loads and then severs the binding. Any changes to the source will not update the target. To see this, update the Binding statement on the text block by adding `Mode=OneTime` like this:

```
Text="{Binding ElementName=MySlider, Path=Value, Mode=OneTime}"
```

Run the project and the value in the text block will be set to 50 (the initial value of the slider). Changing the slider will not change the value in the text block.

`OneWay` binding mode sets the initial value just like `OneTime` but then keeps the connection alive so that changes to the source can be reflected in the target. Changing the mode to `OneWay` (or removing the `Mode` property completely) returns the example back to one direction, and changing the slider will change the value in the text block.

It is important to note that this example works because the target is binding to a dependency property on the `Slider`. If you were to bind to the `Person` class we built for the first binding example and then updated the source, the target value would not change. This is due to the class (as we have written it) missing some needed infrastructure. Fortunately, the fix is to leverage `INotifyPropertyChanged`. This is very simple and is covered in the next section.

TwoWay binding allows for binding back to the source so that user input can update the data source. This is what you should expect when your application is binding to application data. To see TwoWay binding at work, change the TextBlock to a TextBox and set the binding mode to TwoWay as shown:

```
<StackPanel Orientation="Horizontal"
    Margin="100">
    <Slider Name="xSlider"
        Minimum="0"
        Maximum="100"
        Value="50"
        Width="300"
        Margin="10" />
    <TextBox Margin="10"
        Text="{Binding ElementName=xSlider, Path=Value,
        Mode=TwoWay}"
        FontSize="42"
        Height="75"
        VerticalAlignment="Top" />
</StackPanel>
```

Run the application and move the slider; the value in the TextBox updates. Now type a new value between 0 and 100 into the TextBox and press Tab. The value of the slider changes to reflect the value you entered. That is two-way binding at work.

UpdateSourceTrigger

In the previous example, the Slider value didn't change until the TextBox lost focus, but the TextBox updated immediately when the Slider's value changed. Why the different behavior? It is actually very clever how the

framework determines when to send the update. By default, the update is sent when the value changes on a non-text-based control and when a text-based control loses focus. If the screen were to jump around at every keystroke (or send a lot of error messages while the user is still typing), users would not be kept around for long.

There are three options for the `UpdateSourceTrigger` binding property:

- Default
- Explicit
- `PropertyChanged`

Default leaves the behavior the same as previously explained. `Explicit` prevents the binding framework from updating the target, requiring it to programmatically update values with the call to the `UpdateSource` method. `PropertyChanged` will fire every time a property changes (except on lost focus).

Change the `Binding` statement from the previous example by adding `UpdateSourceTrigger=PropertyChanged` as follows:

```
Text="{Binding ElementName=MySlider, Path=Value,  
Mode=TwoWay, UpdateSourceTrigger=PropertyChanged}"
```

Run the program and start typing into the text box. As you type, the slider will update.

INotifyPropertyChanged

It is possible for the value of a property in your data to change after it is displayed to the user. You would like for that value to be updated in the view. For example, you might be retrieving the data from a database and the data gets changed by another instance of the program. It can be imperative that the user interface keep up with these changes.

Imagine that a customer calls a certain store asking for *A History of the English Speaking Peoples* by Winston Churchill. The employee looks on the application to find out how many books are in stock and sees that there is one copy left. While he's negotiating the price with the potential buyer, another employee sells the last copy of the book. If the first employee's screen does not update to show that the store is now sold out of the book, he's in real danger of selling a product he doesn't have.

To prevent this type of situation from occurring, we typically have the classes that will serve as models in the view implement the `INotifyPropertyChanged` interface. This interface consists of exactly one event: `PropertyChanged`. You raise this event each time your property changes, passing in an `EventArgs` that contains the name of your property. The XAML framework listens for this event and will then update any control bound to the property on the model that matches the name in the `EventArgs`. If the `EventArgs` is blank, all properties will be updated.

To illustrate this, create a copy of the `Binding1` project and name it `Binding3`. Add a helper method to the `Person` class called `NotifyPropertyChanged` that wraps the event to cut down on repetitive code.

```
public event PropertyChangedEventHandler PropertyChanged;

private void NotifyPropertyChanged([CallerMemberName]
string caller = "")
{
    PropertyChanged?.Invoke(this, new PropertyChangedEventArgs
    Args(caller));
}
```

The `[CallerMemberName]` is part of the `System.Runtime.CompilerServices` namespace (so be sure to add that to the `Person.cs`) and provides the name of the caller that called it. It is useful for properties with

`INotifyPropertyChanged` so that you can call the `NotifyPropertyChanged` method without having to pass in the parameter name explicitly. To fire this event, change the getters and setters for the `Person` class so that they call the `NotifyPropertyChanged` method from each setter, passing in the name of the property that is changed. That way, each time you set the value of the property for any reason, the UI will be notified.

```
private string firstName;
private string lastName;
public string FirstName
{
    get
    {
        return firstName;
    }
    set
    {
        if (value != firstName)
        {
            firstName = value;
            NotifyPropertyChanged();
        }
    }
}
public string LastName
{
    get
    {
        return lastName;
    }
}
```

```

set
{
    if (value != lastName)
    {
        lastName = value;
        NotifyPropertyChanged();
    }
}
}

```

To test this, add a button to `MainPage.xaml` below the text blocks as follows:

```

<Button Name="cmdChange"
        Content="Change"
        Click="cmdChange_Click" />

```

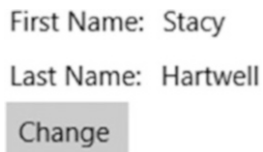
When the button is pressed, modify the `FirstName` of the person object as shown:

```

private void cmdChange_Click( object sender, RoutedEventArgs e )
{
    person.FirstName = "Stacey";
}

```

Because of `INotifyPropertyChanged`, the UI will be updated when the value is changed, as shown in Figure 4-7.



First Name: Stacy
 Last Name: Hartwell
 Change

Figure 4-7. *Person changed*

Binding to Collections

One of the key processes of data binding is showing lists of items. The main control used to show lists is `ListView`. To show binding to collections, we will create an app that takes a list of people and displays it in a `ListView` (shown in Figure 4-8).



Figure 4-8. A data-bound ListView

Creating the Collection

Before we see this at work, we need to create a list. Start by creating a new application named `Binding4`. Instead of manually creating a list of `Person` objects, we are going to create a helper function that will generate names for each of the `Person` objects. In the `Person` class, create arrays to hold some first and last names as well as an array of cities as follows:

```
private static readonly string[] firstNames = { "Adam", "Bob",
"Carl", "David", "Edgar", "Frank", "George", "Harry", "Isaac",
"Jesse", "Ken", "Larry" };
```

```

private static readonly string[] lastNames = { "Aaronson",
"Bobson", "Carlson", "Davidson", "Enstwhile", "Ferguson",
"Harrison", "Isaacson", "Jackson", "Kennelworth", "Levine" };
private static readonly string[] cities = { "Boston",
"New York", "LA", "San Francisco", "Phoenix", "San Jose",
"Cincinnati", "Bellevue" };

```

We can then “assemble” new `Person` objects by randomly selecting from each of the arrays, creating as many people as we need. To keep things simple, we’ll do this in a static method so that we can call it from our `MainPage.xaml.cs` without instantiating an object. `ListView`s bind very well with `IEnumerables`, so we’ll have it return an `IEnumerable<Person>` as shown:

```

public static IEnumerable<Person> CreatePeople( int count )
{
    var people = new List<Person>();
    var r = new Random();

    for ( int i = 0; i < count; i++ )
    {
        var p = new Person()
        {
            FirstName = firstNames[r.Next( firstNames.Length )],
            LastName = lastNames[r.Next( lastNames.Length )],
            City = cities[r.Next( cities.Length )]
        };
        people.Add( p );
    }
    return people;
}

```

The entire Person class now looks like this:

```
public class Person : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler
    PropertyChanged;

    private void NotifyPropertyChanged([CallerMemberName]
    string caller = "")
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEv
        entArgs(caller));
    }

    private string firstName;
    private string lastName;
    public string FirstName
    {
        get
        {
            return firstName;
        }
        set
        {
            if (value != firstName)
            {
                firstName = value;
                NotifyPropertyChanged();
            }
        }
    }
}
```

```
public string LastName
{
    get
    {
        return lastName;
    }
    set
    {
        if (value != lastName)
        {
            lastName = value;
            NotifyPropertyChanged();
        }
    }
}
private string city;
public string City
{
    get
    {
        return city;
    }
    set
    {
        if (value != city)
        {
            city = value;
            NotifyPropertyChanged();
        }
    }
}
```

```

private static readonly string[] firstNames = { "Adam",
"Bob", "Carl", "David", "Edgar", "Frank", "George",
"Harry", "Isaac", "Jesse", "Ken", "Larry" };
private static readonly string[] lastNames = {
"Aaronson", "Bobson", "Carlson", "Davidson",
"Enstwhile", "Ferguson", "Harrison", "Isaacson",
"Jackson", "Kennelworth", "Levine" };
private static readonly string[] cities = { "Boston",
"New York", "LA", "San Francisco", "Phoenix", "San
Jose", "Cincinnati", "Bellevue" };

public static IEnumerable<Person> CreatePeople(int count)
{
    var people = new List<Person>();
    var r = new Random();
    for(int i=0; i <count; i++)
    {
        var p = new Person()
        {
            FirstName = firstNames[r.Next(firstNames.
            Length)],
            LastName = lastNames[r.Next(lastNames.
            Length)],
            City = cities[r.Next(cities.Length)]
        };
        people.Add(p);
    }
    return people;
}
}

```

Creating a Data-Bound ListView

A `ListView` lays out the data vertically in a single column. A `ListView` is most often used when an application is in Portrait or Snapped mode due to the reduced width of the view screen when apps are in those two layouts. A `ListView`s scroll vertically, which is the preferred scrolling direction for Portrait and Snapped views.


To create a new `ListView`, open `MainPage.xaml` and either drag a `ListView` from the toolbox onto the page and name it `MyListView` or enter the following XAML:

```
<ListView Name="MyListView">
</ListView>
```

We need to set the `ItemsSource` property of the `ListView` to the result of calling the static `CreatePeople` function we added to the `Person` class. Add the following line of code in `MainPage.xaml.cs`, in the constructor as follows:

```
public MainPage()
{
    this.InitializeComponent();
    MyListView.ItemsSource = Person.CreatePeople(25);
}
```

When you run this, you won't get quite what you were hoping for, as shown in Figure 4-9.



Binding4.Person
 Binding4.Person
 Binding4.Person
 Binding4.Person

Figure 4-9. Binding without a data template

The binding is working perfectly; there is one entry for each of the 25 Person objects. The problem is that the `ListView` doesn't know how to display a Person object and so it falls back to just showing the name of the type. Data-bound list controls such as the `ListView` require an `ItemTemplate` to define how each item in the list is displayed. The `ItemTemplate` needs to have a `DataTemplate` that contains the defining markup to display each item. Update the `ListView` XAML to match the following:

```
<ListView Name="MyListView">
  <ListView.ItemTemplate>
    <DataTemplate>
      <Border Width="300" Height="Auto"
        BorderBrush="Beige" BorderThickness="1">
        <StackPanel>
          <TextBlock>
            <Run Text="{Binding FirstName}" />
            <Run Text=" " />
            <Run Text="{Binding LastName}" />
            <Run Text=" " />
            <Run Text="(" />
            <Run Text="{Binding City}" />
            <Run Text=")" />
          </TextBlock>
        </StackPanel>
      </Border>
    </DataTemplate>
  </ListView.ItemTemplate>
</ListView>
```

Let's break down the `ItemTemplate`. The main container is the `DataTemplate` and its `DataContext` is the current item in the collection. It's just a matter of formatting the data to your liking. Inside the `TextBlock`

in the sample is a XAML trick that allows for more complex binding into a single TextBlock. By using the <Run/> markup, you can add multiple binding statements to build a single TextBlock. In this case, we are combining the FirstName, LastName, and City data elements with some punctuation. This pattern will be repeated for all 25 entries in the ListView, as shown in Figure 4-10.



Figure 4-10. *ListView*

INotifyCollectionChanged

Just as you want the application to be updated when the value of a property changes, you want it to be notified if the contents of a collection changes (e.g., if an item is added, deleted, etc.). Similar to INotifyPropertyChanged, there is an interface made just for this scenario: INotifyCollectionChanged. This interface has just one event:

```
public event NotifyCollectionChangedEventArgs
CollectionChanged;
```

This event needs to be called each time an item is added or removed from the collection. We could inherit from a specialized collection and code this ourselves (much like we did in the Person class for INotifyPropertyChanged), but Microsoft has already done the work for us and provided a custom class called ObservableCollection<T>. You can use the ObservableCollection<T> class the same way you use other collection classes.

In fact, you can modify `Person.cs` to use an `ObservableCollection<Person>` and the program will continue to work just like it did before with the added feature that the `ListView` will be updated if an item is added or removed from the collection. All we need to do is simply change the return type of the `CreatePeople` method to return `ObservableCollection<Person>` and change the internal variable from `List<Person>` to `ObservableCollection<Person>`:

```
public static ObservableCollection<Person> CreatePeople( int
count )
{
    var people = new ObservableCollection<Person>();
```

Note The `ObservableCollection` class only updates the client if an item is added or deleted but not if the item is changed (e.g., a property on an item in the collection is changed). If you want to receive a notification of the item being changed, you need to implement `INotifyPropertyChanged` on the items.

Data Converters

There are times that you will want to bind an object to a control but the types won't match up correctly. For example, you might want to bind a check box to a value, setting the check mark if the value is anything other than zero. To do this, you need to convert the integer value to a Boolean, and for that you need a `DataConverter`. A `DataConverter` is any class that implements `IValueConverter`, which takes two methods: `Convert` and `ConvertBack`.

To see this at work, create a new project and name it Binding5. In MainPage.xaml, we'll have a CheckBox that will be set if the Num property of the Person object is not zero. A text block will display the Num property, and there will be a button that will generate a new Num property at random.

```
<StackPanel Margin="100"
    Orientation="Horizontal"
    Height="100">
    <CheckBox Name="xCheckBox"
        Content="Is Not Zero"
        Margin="10"
        IsChecked="{Binding Num,
            Converter={StaticResource numToBool}}"/>
    <TextBlock Name="xTextBlock"
        Margin="10"
        FontSize="42"
        VerticalAlignment="Center"
        Text="{Binding Num}"/>
    <Button Name="xButton"
        Content="Generate Number"
        Click="xButton_Click"
        Margin="10" />
</StackPanel>
```

Notice that the IsChecked binding for the CheckBox calls on the DataConverter that was identified in the Resources section:

```
<Page.Resources>
    <local:IntegerToBooleanConverter x:Key="numToBool" />
</Page.Resources>
```

We'll use the Person class we used in previous iterations, adding a Num property to the Person class so that we can bind to it (e.g., number of children, etc.):

```
public class Person : INotifyPropertyChanged
{
    private string firstName;
    public string FirstName
    {
        get { return firstName; }
        set
        {
            firstName = value;
            NotifyPropertyChanged();
        }
    }
    private string lastName;
    public string LastName
    {
        get { return lastName; }
        set
        {
            lastName = value;
            NotifyPropertyChanged();
        }
    }

    private int num;
    public int Num
    {
        get { return num; }
    }
}
```

```

    set
    {
        num = value;
        NotifyPropertyChanged();
    }
}

public event PropertyChangedEventHandler PropertyChanged;
private void NotifyPropertyChanged( [CallerMemberName]
string caller = "" )
{
    if ( PropertyChanged != null )
    {
        PropertyChanged( this, new
        PropertyChangedEventArgs( caller ) );
    }
}
}

```

The key to making this work, of course, is the converter. Create a new class called `IntegerToBooleanConverter` and have it implement `IValueConverter` as shown:

```

public class IntegerToBooleanConverter : IValueConverter
{
    public object Convert( object value, Type targetType,
object parameter, string language )
    {
        int num = int.Parse( value.ToString() );
        if ( num != 0 )
            return true;
    }
}

```

```

        else
            return false;
    }

    public object ConvertBack( object value, Type targetType,
    object parameter, string language )
    {
        throw new NotImplementedException();
    }
}

```

The binding mechanism will take care of passing in the value (in this case `num`), the `targetType` (in this case `Boolean`), a parameter if there is one, and the language. Our job is to convert the value to a `Boolean`, which we do in the body of the `Convert` method. The net effect is to convert any non-zero value to the `Boolean` value `true`, which sets the check box, as shown in Figure 4-11.

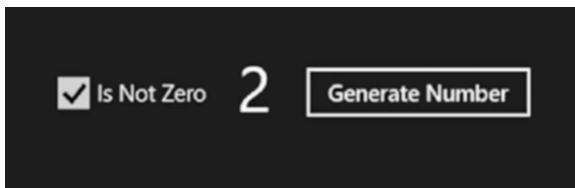


Figure 4-11. Boolean converter

`ConvertBack` is used in `TwoWay` binding scenarios. For example, if you needed to format a number from the model to resemble the user's local currency, you could do that using `ConvertMethod`. In order to save that number back into the model, it would need to be changed back into a number from the string representation that includes the currency symbol and the commas.

Summary

Binding is a key element of programming for Windows 10 UWP. With data binding, you can connect a data source, which can be virtually any object, to a property on a visual element (or you can bind two visual elements together). Along the way, you can convert data from one type to another to facilitate the binding. The `INotifyPropertyChanged` interface allows your view to be updated when your data changes. `TwoWay` binding allows your data object to be changed based on user input.

CHAPTER 5

Local Data

All significant Windows 10 UWP applications manage and store data of some sort. This data can be arbitrarily divided into application data and user data. The former, application data, refers to the state of the application at any given time, including:

- What page the user is on
- Which form fields have been filled out
- Which selections and other choices have been made

User data, on the other hand, is data entered by the user specifically to be processed and stored by the application. User data can be stored in a number of formats, and it can be stored in a number of places on your disk or in the cloud.

This chapter starts off by showing you how to store application data and then explores some of the most popular ways to store user data.

Application Data

While your Windows 10 UWP application is running, you want to be saving data all the time. After all, you can switch to another application at any time and then have only five seconds to store your data at that point.

While it's true that you want to be saving major data points all the time, there are certain "status" values that you want to store only at the last minute. This might include visible information, information that's selected, currently filled-in fields, and so forth.

A great way to store that information is using an `ApplicationDataContainer` control. This can be stored with your local or roaming settings, and it requires no special permission from the operating system; these files are considered safe.

Settings Containers

There are three settings containers that can be used for local storage: local, roaming, and temporary. Local is stored on the device and persists across app launches. Roaming is also stored locally but will sync across devices based on the currently logged-in user. Temporary is, well, temporary. It can be deleted by the system at any time.

Saving, Reading, and Deleting Local Data

Let's create an application that stores, reads, and deletes application data. Our UI is very simple—it consists of a text box to enter information into, a text block to display recalled information, and three buttons, as shown in [Figure 5-1](#).

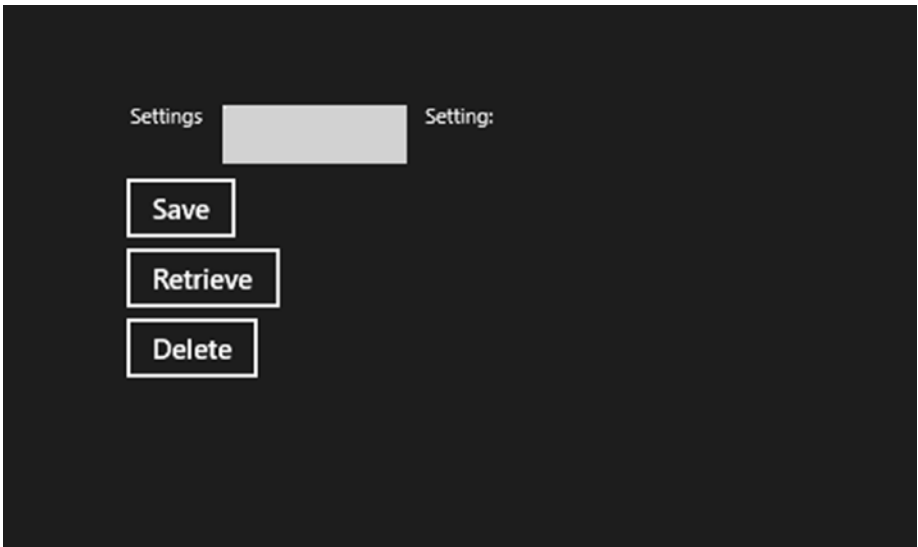


Figure 5-1. *View for saving, retrieve, and deleting data*

Start with a new project based on the Blank App (Universal Windows) template and name your app `AppSettings`. Select the versions of Windows 10 you want to support. In this example, we are going to pick the earliest version of Windows 10 available as well as the most current version of Windows 10, as shown in Figure 5-2. This is not a requirement but it provides the most flexibility when targeting your users. Do note that the broader your minimum and target versions are, the fewer options you have for newer features that may have been released in the newest versions of Windows 10.

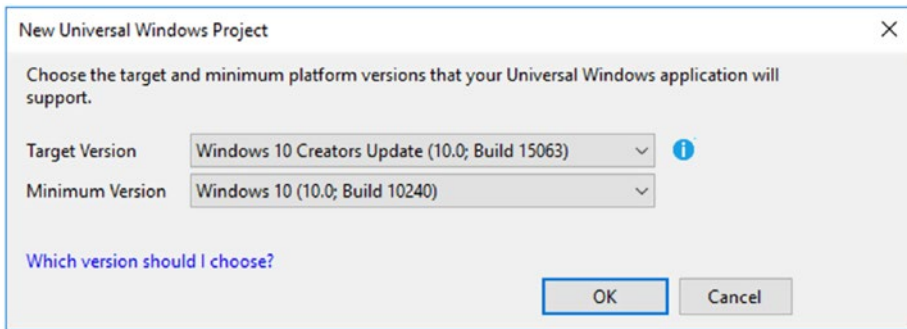


Figure 5-2. Picking the minimum and target versions of Windows 10

Now open `MainPage.Xaml.cs` and add the following using statement to the top of the file:

```
using Windows.Storage;
```

Then, add the following declaration to the top of the class:

```
private ApplicationDataContainer settings =  
    ApplicationData.Current.LocalSettings;
```

This creates a local variable that references the local data storage.

Next, open `MainPage.xaml` and replace the default Grid with the following XAML:

```
<Grid Background="{ThemeResource  
ApplicationPageBackgroundThemeBrush}">  
    <Grid.ColumnDefinitions>  
        <ColumnDefinition Width="120"/>  
        <ColumnDefinition Width="*/>  
    </Grid.ColumnDefinitions>  
    <Grid.RowDefinitions>  
        <RowDefinition Height="140"/>  
        <RowDefinition Height="*/>  
    </Grid.RowDefinitions>
```

```

<StackPanel Grid.Column="1" Grid.Row="1">
  <StackPanel Orientation="Horizontal">
    <TextBlock Text="Settings"
      Margin="5" />
    <TextBox Width="100"
      Height="30"
      Name="txtSettings"
      Margin="5"/>
    <TextBlock Text="Setting: " Margin="5"/>
    <TextBlock Name="txtSettingOutput"
      Text=""
      Margin="5"/>
  </StackPanel>
  <Button Name="SaveSettings"
    Content="Save"
    Click="SaveSettings_Click" />
  <Button Name="RetrieveSettings"
    Content="Retrieve"
    Click="RetrieveSettings_Click" />
  <Button Name="DeleteSettings"
    Content="Delete"
    Click="DeleteSettings_Click" />
</StackPanel>
</Grid>

```

Go on to open the `MainPage.xaml.cs` file again and add another class-level variable. This variable will be used to eliminate the use of “magic strings,” which can lead to typos and runtime errors.

```
private string settingName = "UserSetting";
```

When the user clicks the Save button, the content from the text box is written to the settings collection, and then the text box is cleared. Create the event handler for the SaveSettings button in MainPage.xaml.cs and enter the following code:

```
private void SaveSettings_Click(object sender, RoutedEventArgs e)
{
    string userValue = txtSettings.Text;

    settings.Values[settingName] = userValue;

    txtSettings.Text = string.Empty;
}
```

If a UserSetting already exists in the dictionary, the current value will be overwritten. If it doesn't exist, it will be created.

Retrieving the text is equally simple. We pull the text out of the settings collection as an object, and if it is not null, we turn it into a string. If the setting does not exist, it will come back as null. While still in MainPage.xaml.cs, create the event handler for the RetrieveSettings button and add the following code:

```
private void RetrieveSettings_Click(object sender,
RoutedEventArgs e)
{
    string val = settings.Values[settingName]?.ToString();
    if (!string.IsNullOrEmpty(val))
    {
        txtSettingOutput.Text = val;
    }
}
```

Finally, deleting a setting just requires a call to the `Remove` method. We also clear out the output text box. Create the event handler for the `DeleteSettings` button and add the following code:

```
private void DeleteSettings_Click(object sender,
RoutedEventArgs e)
    {
        settings.Values.Remove(settingName);
        txtSettingOutput.Text = string.Empty;
    }
```

That's all it takes to manage your application state. Piece of cake. Easy as pie. The complete code is listed here:

```
public sealed partial class MainPage : Page
    {
        private ApplicationDataContainer settings =
            ApplicationData.Current.LocalSettings;

        private string settingName = "UserSetting";

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

        private void SaveSettings_Click(object sender,
            RoutedEventArgs e)
        {
            string userValue = txtSettings.Text;
            settings.Values[settingName] = userValue;
            txtSettings.Text = string.Empty;
        }
    }
```

```
private void RetrieveSettings_Click(object sender,
RoutedEventArgs e)
{
    string val = settings.Values[settingName]?.
ToString();
    if (!string.IsNullOrEmpty(val))
    {
        txtSettingOutput.Text = val;
    }
}

private void DeleteSettings_Click(object sender,
RoutedEventArgs e)
{
    settings.Values.Remove(settingName);
    txtSettingOutput.Text = string.Empty;
}
}
```

Creating the Data Layer

Applications that rely on data are wise to separate data-related classes into a separate layer. Often, in large applications, this data-access layer is a separate project. To keep things simple in our examples, we will say that all classes are the same project for simplicity.

To begin, create a new Windows 10 UWP application using the Blank Application template and call it `LocalFolderSample`. In the following sections, you will create the foundation for the rest of the examples in this chapter.

Creating the Repository Interface

Basic data operations include Create, Read, Update, and Delete, commonly referred to as CRUD operations. A very common pattern to use is the repository pattern. While this section does not attempt to fully execute the pattern, we will place these functions into repository classes. We start by creating a common interface for all repositories in the data-access layer. This is a good practice, even though we are only creating a single repository in this example.

To begin, create a new folder in your project named `Data` by right-clicking the project name and selecting `Add ► New Folder`. Next, add a new interface by right-clicking the folder you just created and selecting `Add ► New Item` and then `Interface`. Name the folder `IDataRepository` and enter the following code:

```
public interface IDataRepository<T>
{
    Task Add(T customer);

    Task<ObservableCollection<T>> Load();

    Task Remove(T customer);

    Task Update(T customer);
}
```

To use the `ObservableCollection`, you need to add a `using` to `System.ComponentModel`. At the top of the file, add this statement:

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

There's a lot going on in this interface, and it merits some discussion before we move on. The interface is created using generics so that the implementing classes can be strongly typed. The `ObservableCollection`

class has been around since Windows Presentation Foundation (WPF), and it is a class that implements `INotifyCollectionChanged` and `INotifyPropertyChanged`. These events are raised any time items are added to or deleted from the collection, or when any other changes are made to the collection class. The data-binding engine in Windows 10 UWP XAML apps listens for those events and will automatically update the bound elements to the new values. Additionally, all of the methods have been defined to return a `Task<T>` to enable asynchronous operations.

Creating the DataModel

Next, we need to create the class that will hold and transport the data. Sometimes referred to as Data Transfer Objects (DTOs) or Plain Old CLR Objects (POCOs), the term most commonly used is `Model`. Add a new class by right-clicking the `Data` folder and selecting `Add ► New Item ► Class`. Name the class `Customer.cs` and add the following code:

```
public class Customer
{
    public int Id { get; set; }

    public string Email { get; set; }

    public string FirstName { get; set; }

    public string LastName { get; set; }

    public string Title { get; set; }
}
```


Creating the ViewModel

The final class to create is the view model. This class will act as the data context for the view. To do this, right-click the Data folder and select Add ► New Item ► Class. Name the file `ViewModel.cs`. The `ViewModel` will use an instance of the data repository interface to get the necessary data (as well as to make any updates), so we add a class-level variable to hold the instance. Open `ViewModel.cs` and add a member variable of type `IDataRepository<Customer>`:

```
private IDataRepository<Customer> data;
```

To get an instance of the repository into the view model, we use constructor injection. The constructor takes an `IDataRepository` and initializes the local variable as follows:

```
public ViewModel(IDataRepository<Customer> data)
{
    this.data = data;
}
```

Implementing INotifyPropertyChanged

In this example, we want the view model to implement `INotifyPropertyChanged` instead of the model. Add the interface `INotifyPropertyChanged` to the class and create a `PropertyChanged` event (this is the only item in `INotifyPropertyChanged`).

```
public class ViewModel : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;
```

You will need to add a using to `System.ComponentModel` as follows:

```
using System.ComponentModel;
```

The common implementation is to create a method to encapsulate the raising of the `PropertyChanged` event. To do this, create a new method called `RaisePropertyChanged()`. Add the following code:

```
private void RaisePropertyChanged([CallerMemberName] string
fieldName = "")
{
    PropertyChanged?.Invoke(this, new PropertyChanged
    EventArgs(fieldName));
}
```

You will need to add a using to `System.Runtime.CompilerServices` as follows:

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

This method checks to make sure there is a listener for the event and then fires it with the `fieldname` as the event argument. We use the `CallerMemberNameAttribute` for the variable `fieldname`. This allows us to call `RaisePropertyChanged` inside a setter without having to pass in an argument, as it will use the field name that is calling the method. This will become more important as we develop the rest of the view model. Note that `CallerMemberName` requires the parameter to have a default value. We are also using the null-condition operator, `?`, after the `PropertyChanged` event. This will allow us to call `Invoke`, which actually executes the method tied to the event, if and only if, `PropertyChanged` is not null. We could check for nulls, as done in the past, by using an `if` statement, but using the null-condition operator will cut down on the

boilerplate code that is needed. The following code shows the current status of the view model:

```
public class ViewModel : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    private IRepository<Customer> data;

    public ViewModel(IRepository<Customer> data)
    {
        this.data = data;
    }

    private void RaisePropertyChanged([CallerMemberName] string
        fieldName = "")
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs
            (fieldName));
    }
}
```

Adding the Public Properties

There are two public properties in `ViewModel`: `SelectedItem` and `Customers`. We can't use automatic properties because we want to leverage the `PropertyChanged` notification system. So we have to use property statements with backing fields.

The first property to create is `SelectedItem`. Add the following code to `ViewModel.cs`:

```
private Customer selectedItem;

public Customer SelectedItem
{
    get { return selectedItem; }

    set
    {
        if(value != selectedItem)
        {
            selectedItem = value;
            RaisePropertyChanged();
        }
    }
}
```

In the Setter, if the value has been changed, then the `RaisePropertyChanged` method is called.

The next property is `Customers`, which is an `ObservableCollection` of the `Customer` class. Before adding this property, be sure to add the `using` for `System.Collections.ObjectModel`:

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

Then, add the property:

```
private ObservableCollection<Customer> customers;

public ObservableCollection<Customer> Customers
{
    get {return customers;}
}
```

```
    set
    {
        customers = value;
        RaisePropertyChanged();
    }
}
```

Go on to create an async method to load the data:

```
public async void Initialize()
{
    Customers = await _data.Load();
}
```

Finally, you are ready to implement the CRUD operations, delegating the work to the repository. You do this as follows:

```
internal void AddCustomer(Customer cust)
{
    data.Add(cust);
}

internal void DeleteCustomer(Customer cust)
{
    data.Remove(cust);
}
```

That's it for the ViewModel class. By using a repository, we keep the view model simple, clean, and reusable with different storage approaches.

Local Data

Local data files include any data in your app, including data that the user entered and should be persisted. One option is to persist this data in the local or roaming settings as discussed in the previous section. This can be less than ideal, especially with large amounts of data.

Another option is to store the data in a file located in a folder under the signed-in user's name. This is not the same as using `LocalSettings`. `LocalSettings` is a dictionary object that is stored in the registry, and as such, is limited in size. Local and roaming folders are stored alongside the app on the disk.

Using JSON to Format Data

Before getting into the meat of reading and writing data files, we need to consider the different storage formats available. Local data files are text files, and as such, all of the data stored will be simple strings. There was a time that XML was the predominant format for string data. However, XML is heavy with all of the tags and can be difficult to work with.

The current leader in textual data is JSON—JavaScript Object Notation. Initially championed by the web for its lightweight format and its ease of conversion between text and an object graph, JSON is winning the format war, and most Windows 10 UWP developers are using JSON for local storage of text.

The leading package with which C# developers can work with JSON data is `Json.NET`. One way to add `Json.NET` to your project is to download the package from <http://json.codeplex.com/>, install it (or unzip it based on the download), and reference the correct assemblies. An easier way to add it is to use NuGet to quickly and easily add JSON to your project. See the sidebar for more information.

USING NUGET TO INSTALL JSON.NET

Instead of downloading the Json.NET package from <http://json.codeplex.com>, you can install it through NuGet. To install Json.NET through NuGet, choose Tools ► Library Package Manager ► Package Manager Console. This will open the console window at the bottom of your screen, providing you with the Package Manager prompt (PM>). Then, enter the command to load Json.NET:

```
PM> Install-Package Newtonsoft.Json
```

Remember to include the hyphen between Install and Package. A few seconds later, the package will be installed.

As discussed in the chapter on tooling, you can also use the GUI for NuGet, which includes a nice search feature. Instead of having to know the exact package name, you can search for JSON and pick the correct package.

Local Data Containers

Just like the settings containers, there are three local data containers that can be used for local storage: local, roaming, and temporary. Local is stored on the device and persists across app launches. Roaming is also stored locally but will sync across devices based on the currently logged-in user. Temporary is, again, temporary. It can be deleted by the system at any time.

Creating the File Repository

The actual storage of the file is pretty simple, but we're going to build a fully reusable repository model based on the interface we created earlier. Using the interface as the base allows us to reapply the same pattern to other storage approaches later. We begin with the data file itself. To keep things simple, I'll stay with the idea of storing and retrieving customer data.

To create the file repository, first add a new class file to the Data directory by right-clicking the directory and selecting Add ► New Item ► Class. Name the class FileRepository.

Next, add the `IDataRepository<Customer>` interface to the class and the required members. The class will look like this:

```
public class FileRepository : IDataRepository<Customer>
{
    public Task Add(Customer customer)
    {
        // TODO: Implement this method
        throw new NotImplementedException();
    }

    public Task<ObservableCollection<Customer>> Load()
    {
        // TODO: Implement this method
        throw new NotImplementedException();
    }

    public Task Remove(Customer customer)
    {
        // TODO: Implement this method
        throw new NotImplementedException();
    }

    public Task Update(Customer customer)
    {
        // TODO: Implement this method
        throw new NotImplementedException();
    }
}
```


Finally, add the required usings for `System.Collections.ObjectModel` and `Windows.Storage`:

```
using System.Collections.ObjectModel;
using Windows.Storage;
```

Following this, we need to add the member variables of a string for the file name of our specific storage file, an observable collection of `Customer`, and a `StorageFolder` referencing the `LocalFolder` using the following code:

```
private string fileName = "customers.json";
```

```
ObservableCollection<Customer> customers;
```

```
StorageFolder folder = ApplicationData.Current.LocalFolder;
```

The constructor calls the `Initialize` method, which in this example does nothing. The `Initialize` method will be used when we cover `SQLite` later in this chapter.

```
public FileRepository()
{
    Initialize();
}
private void Initialize()
{
}
```

Next, create two helper methods in the `FileRepository` class to save and read the collection of customer data to and from disk. We first need to add the following using statement:

```
using Newtonsoft.Json;
```

The first helper method serializes the collection of customers and then

```
private Task WriteToFile()
{
    return Task.Run(async () =>
    {
        string JSON = JsonConvert.SerializeObject(customers);
        var file = await OpenFileAsync();
        await FileIO.WriteTextAsync(file, JSON);
    });
}

private async Task<StorageFile> OpenFileAsync()
{
    return await folder.CreateFileAsync(fileName,
    CreationCollisionOption.OpenIfExists);
}
```

Notice that `WriteToFile` converts the `customers` collection to JSON, then asynchronously opens the file to write to, and then finally writes to the file, again asynchronously. To open the file, we add the helper method `OpenFileAsync`. Notice that when opening the file, we handle `CreationCollisions` by saying that we want to open the file if it already exists.

Now it's time to add the methods that handle adding, removing, and updating the customers in the list. The `Add` method adds a customer (passed in as a parameter) to the `customers` collection and then calls `WriteToFile`:

```
public Task Add(Customer customer)
{
    customers.Add(customer);
    return WriteToFile();
}
```

The `Remove` method removes a customer (passed in as a parameter) from the `customers` collection and then calls `WriteToFile` as such:

```
public Task Remove(Customer customer)
{
    customers.Remove(customer);
    return WriteToFile();
}
```

The third interface method is `Update`. Here, we have slightly more work to do: we must find the record we want to update and, if it is not null, we remove the old version and save the new one.

```
public Task Update(Customer customer)
{
    var oldCustomer = customers.FirstOrDefault(
        c => c.Id == customer.Id);
    if (oldCustomer == null)
    {
        throw new System.ArgumentException("Customer not found.");
    }
    customers.Remove(oldCustomer);
    customers.Add(customer);
    return WriteToFile();
}
```

READING AND WRITING THE ENTIRE FILE

So why remove the old record and add the new one? While JSON is a very efficient storage and transport mechanism for text, it is not a relational database. It is much more efficient to simply remove the old record and replace it with the new one than to loop through all of the properties and

update the record. Since there isn't a concept of an auto-increment ID (or any other relational database concept for that matter), we don't lose anything, and gain only speed.

A similar question can be asked as to why save the entire file each time there is a change instead of just updating the individual record. The answer is pretty much the same: The time and effort it would take to replace/add/change individual records compared to writing the entire file on each change makes writing the file each time a much faster and less fragile option.

What if you have a really large text file? Then I suggest looking at a more database-centric solution, such as SQLite, discussed later in this chapter.

Finally, we come to Load. Here, we create our file asynchronously and if it is not null, we read the contents of the file into a string. Then, the string is deserialized into a list of customers and added into the class's `ObservableCollection<Customer>`. Prior to flushing out the Load method, we need to add a using for `System.Collections.Generic` as follows:

```
using System.Collections.Generic;
```

If the Load method isn't marked as `async`, you need to do that now. Then add the following code into the Load method:

```
public async Task<ObservableCollection<Customer>> Load()
{
    var file = await _folder.CreateFileAsync(
        _fileName, CreationCollisionOption.OpenIfExists);
    string fileContents = string.Empty;
    if (file != null)
    {
        fileContents = await FileIO.ReadTextAsync(file);
    }
}
```

```
    IList<Customer> customersFromJSON =  
        JsonConvert.DeserializeObject<List<Customer>>(fileContents)  
            ?? new List<Customer>();  
    _customers = new ObservableCollection<Customer>(  
        customersFromJSON);  
    return customers;  
}
```

You then deserialize the customer from the string of JSON into an `IList` of `Customer` and create an `ObservableCollection` of customer from that `IList`. Now that the repository is complete, it's time to create the view.

Creating the View

The view that we create will be very simple. It will consist of four text boxes with labels to add a new record with the list of current customers following the data entry section. It is not an award-winning UI, but enough to show the concepts of this chapter. The last piece of UI is a command bar that holds the Save and Delete command buttons. The UI is shown in Figure 5-3.

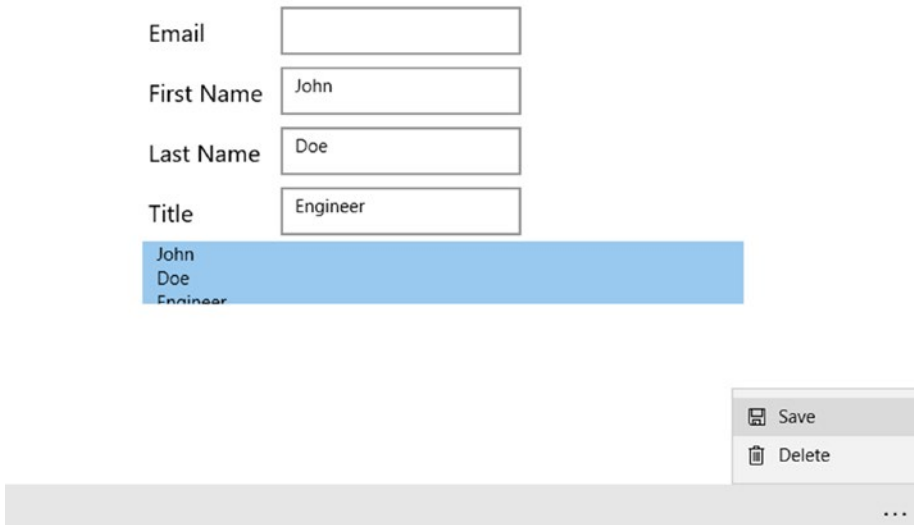


Figure 5-3. UI for the local data example

Open `MainPage.xaml` and add the following styles to the `Page.Resources` section:

```
<Page.Resources>
    <Style TargetType="TextBlock">
        <Setter Property="FontSize"
            Value="20" />
        <Setter Property="Margin"
            Value="5" />
        <Setter Property="HorizontalAlignment"
            Value="Right" />
        <Setter Property="Grid.Column"
            Value="0" />
        <Setter Property="Width"
            Value="100" />
    </Style>
</Page.Resources>
```

```

        <Setter Property="VerticalAlignment"
            Value="Center" />
    </Style>
    <Style TargetType="TextBox">
        <Setter Property="Margin"
            Value="5" />
        <Setter Property="HorizontalAlignment"
            Value="Left" />
        <Setter Property="Grid.Column"
            Value="1" />
    </Style>
</Page.Resources>

```

Next, add a `CommandBar` to the `BottomAppBar` and add the `Save` and `Delete` buttons:

```

<Page.BottomAppBar>
    <CommandBar>
        <CommandBar.SecondaryCommands>
            <AppBarButton x:Name="cmdSave" Label="Save"
                Click="cmdSave_Click" Icon="Save"/>
            <AppBarButton x:Name="cmdDelete" Label="Delete"
                Click="cmdDelete_Click" Icon="Delete"/>
        </CommandBar.SecondaryCommands>
    </CommandBar>
</Page.BottomAppBar>

```

Then, create a set of stack panels to gather the data and a `ListView` to display the data as follows:

```

<StackPanel Margin="150">
    <StackPanel Orientation="Horizontal">
        <TextBlock Text="Email" Margin="5" />
        <TextBox Width="200" Height="40"

```

```

        Name="Email" Margin="5" />
</StackPanel>
<StackPanel Orientation="Horizontal">
    <TextBlock Text="First Name" Margin="5" />
    <TextBox Width="200" Height="40"
        Name="FirstName" Margin="5" />
</StackPanel>
<StackPanel Orientation="Horizontal">
    <TextBlock Text="Last Name" Margin="5" />
    <TextBox Width="200" Height="40"
        Name="LastName" Margin="5" />
</StackPanel>
<StackPanel Orientation="Horizontal">
    <TextBlock Text="Title" Margin="5" />
    <TextBox Width="200" Height="40"
        Name="Title" Margin="5" />
</StackPanel>
<ScrollViewer>
    <ListView Name="xCustomers"
        ItemsSource="{Binding Customers}"
        SelectedItem="{Binding SelectedItem, Mode=TwoWay}"
        Height="400">
        <ListView.ItemTemplate>
            <DataTemplate>
                <StackPanel>
                    <TextBlock Text="{Binding FirstName}" />
                    <TextBlock Text="{Binding LastName}" />
                    <TextBlock Text="{Binding Title}" />
                </StackPanel>
            </DataTemplate>
        </ListView.ItemTemplate>
    </ListView>

```



```
</ScrollViewer>
</StackPanel>
```

Notice the binding for `ItemsSource` and `SelectedItem`.

The code-behind is straightforward. The first thing we do is instantiate an `IDataRepository` and declare the `ViewModel`:

```
public sealed partial class MainPage : Page
{
    private IDataRepository<Customer> data;

    private ViewModel vm;
```

This requires adding the namespace for the `Data` folder:

```
using LocalFolderSample.Data;
```

In the constructor, we create the `ViewModel` passing in the repository, then call `initialize` on the VM, and finally set the VM as the `DataContext` for the page:

```
public MainPage()
{
    this.InitializeComponent();
    data = new FileRepository();
    vm = new ViewModel(data);
    vm.Initialize();
    DataContext = vm;
}
```

All that is left is to implement the two event handlers as such:

```
private void cmdSave_Click(object sender,
    RoutedEventArgs e)
{
    Customer cust = new Customer
```

```

        {
            Email = Email.Text,
            FirstName = FirstName.Text,
            LastName = LastName.Text,
            Title = Title.Text
        };
        vm.AddCustomer(cust);
    }

    private void cmdDelete_Click(object sender,
        RoutedEventArgs e)
    {
        vm?.DeleteCustomer(vm.SelectedItem);
    }

```

When you run the application, you are presented with the view shown earlier in Figure 5-3. Fill in an entry, click the three dots in the lower-right corner (or swipe up from the bottom) to bring up the app bar, and click Save. It immediately appears in the list box.

Once you have saved at least one record, the `Customers.json` file is saved in application data. You can find it by searching under `AppData` on your main drive, which for most people is the C drive (see Figure 5-4).

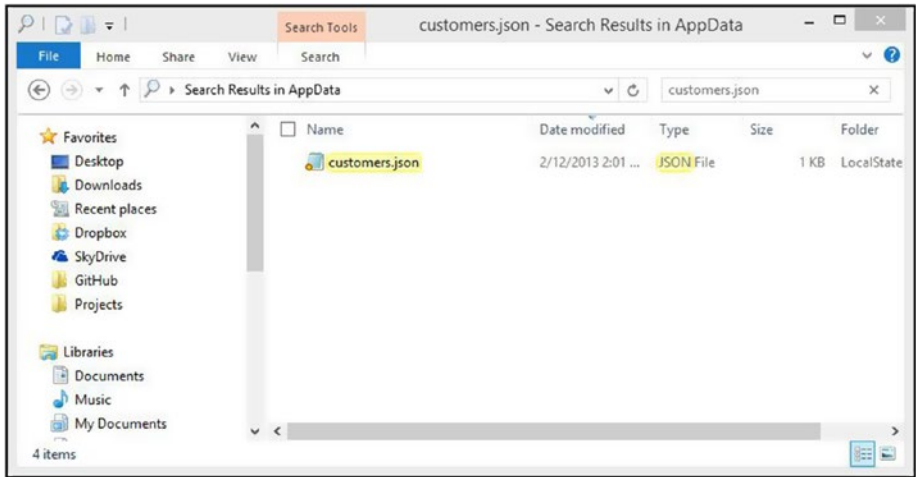


Figure 5-4. The data file in AppData

Double-click that file to see the JSON you've saved:

```
[{"Id":0,"Email":"john@domain.com","FirstName":"John","LastName":"Doe","Title":"Scientist"}]
```

Roaming

To change from storing your data in local storage to roaming storage, you must change *one line* of code. Back in `FileRepository.cs`, change the `LocalFolder` to a `RoamingFolder` (see Figure 5-5).

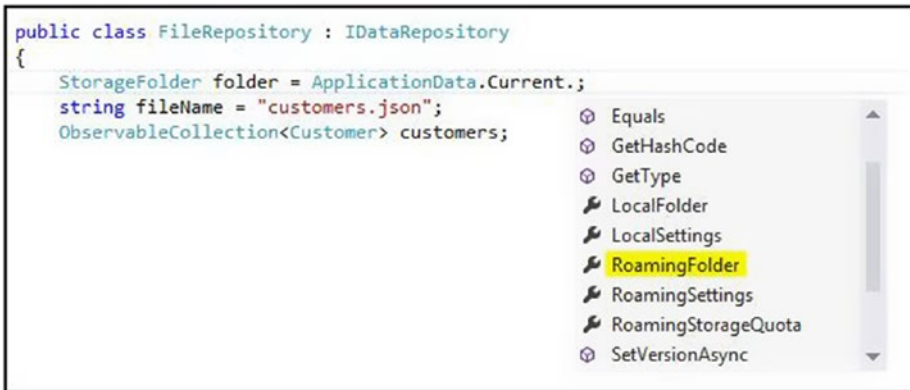


Figure 5-5. Available settings options

Hey! Presto! Without any further work, your application data is now available on any Windows 8 computer that has the application installed.

User-Specified Locations

Local and roaming files are a valid solution for storing application data, but sometimes you want to write your data to a more well-known location. You can do so in Windows 10 by having the user explicitly pick a location at runtime.

Creating the FileOperations Class

To create the FileOperations class, we have to make some modifications to the program we previously wrote. The interface remains the same, but the implementation of the DataRepository changes a bit, and we add another class to encapsulate the utilization of the file pickers.

Start by adding a new class to the `DataModel` folder named `FileOperations` and add three static class-level variables:

```
public static ApplicationDataContainer settings =
ApplicationData.Current.LocalSettings;

private static string mruToken; private static string tokenKey
= "mruToken";
```

We need to add a method to create the file if it doesn't exist:

```
private static async Task<StorageFile> CreateFile(string
fileName)
{
    FileSavePicker savePicker = new FileSavePicker();
    savePicker.SuggestedStartLocation = PickerLocationId.
DocumentsLibrary;
    savePicker.FileTypeChoices.Clear();
    savePicker.FileTypeChoices.Add("JSON", new List<string>() {
        ".json" });
    var file = await savePicker.PickSaveFileAsync();
    return SaveMRU(file);
}
```

You have to add a `using` to include `Windows.Storage.Pickers`. Once the user has selected a file, we want to save the file into the `MostRecentlyUsedList`. This list remembers that the user gave permission to open the file and therefore that the files that are stored in the list are accessible on successive loads. The code to accomplish this is as follows:

```
private static StorageFile SaveMRU(StorageFile file)
{
    if(file != null)
    {
```

```

        mruToken = StorageApplicationPermissions.
            MostRecentlyUsedList.Add(file);
        settings.Values["mruToken"] = mruToken;
        return file;
    }
    else
    {
        return null;
    }
}

```

Make sure that you have added using: `Windows.Storage`.

We also use a `FileOpenPicker` to enable loading a file from the computer in case the MRUToken isn't available:

```

private static async Task<StorageFile> GetFile()
{
    FileOpenPicker openPicker = new FileOpenPicker();
    openPicker.SuggestedStartLocation = PickerLocationId.
        DocumentsLibrary;
    openPicker.ViewMode = PickerViewMode.List;

    // Filter to include a sample subset of file types.
    openPicker.FileTypeFilter.Clear();
    openPicker.FileTypeFilter.Add(".json");

    // Open the file picker.
    var file = await openPicker.PickSingleFileAsync();
    return SaveMRU(file);
}

```

If the MRUToken does exist, we can greatly simplify getting the file by doing the following:

```
private static async Task<StorageFile> GetFileFromMRU()
{
    return
    await StorageApplicationPermissions
        .MostRecentlyUsedList.GetFilesAsync(_mruToken);
}
```

Finally, we create the entry method into the FileOperations:

```
public static async Task<StorageFile> OpenFile(string fileName)
{
    mruToken = settings.Values[tokenKey] != null ?
        settings.Values[tokenKey].ToString() : null;
    if(mruToken != null)
    {
        return await GetFileFromMRU();
    }
    var file = await GetFile();
    if(file != null)
    {
        return file;
    }
    else
    {
        return await CreateFile(fileName);
    }
}
```

With the `FileOperations` class built, there are just a few changes to make to the `FileRepository` class. At the top of the `FileRepository` class, we need to delete the instantiation of the `StorageFolder`:

```
StorageFolder folder = ApplicationData.Current.RoamingFolder;
```

Then, we update the `OpenFileAsync` to the following:

```
private async Task<StorageFile> OpenFileAsync()
{
    return await FileOperations.OpenFile(fileName);
}
```

Finally, we update the `Load` method by deleting the first line:

```
var file = await folder.CreateFileAsync(
    fileName, CreationCollisionOption.OpenIfExists);
```

and replacing it with this one:

```
var file = await FileOperations.OpenFile(fileName);
```

Adding the File Association for JSON Files

To add a file association for JSON files, open the `Package.appxmanifest` file and click the `Declarations` tab. Open the drop-down list for the available declarations, select `File Type Associations`, and click `Add`, as shown in [Figure 5-6](#).

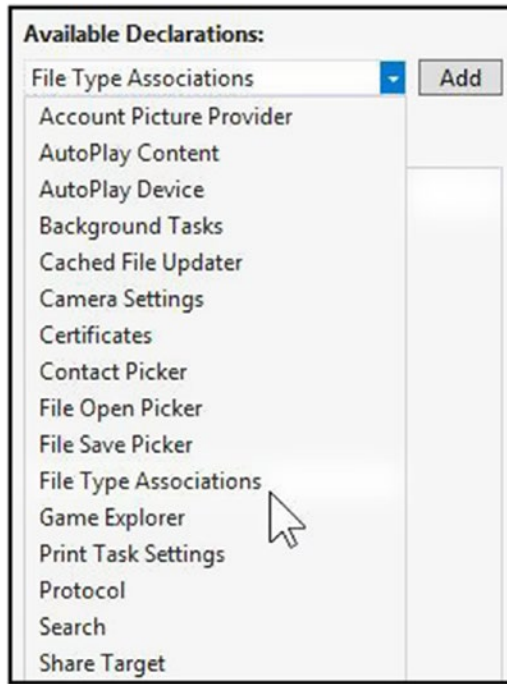


Figure 5-6. Declaring file type associations

Once you have added the file type association, you must also fill in a number of fields in the right rail as follows:

- *Display Name:* json files
- *Name:* json
- *Content Type:* text/plain
- *File Type:* .json
- *Open Is Safe:* checked

The filled-out form is shown in Figure 5-7.

Declarations Packaging

properties.

Description:
Registers file type associations, such as .jpeg, on behalf of the app.
Multiple instances of this declaration are allowed in each app.
[More information](#)

Properties:

Display name: json files

Logo: [x] [...]

Info tip:

Name: json

Edit flags

Open is safe
 Always unsafe

Supported file types

At least one file type must be supported. Enter at least one file type; for example, ".jpg".

Supported file type	Remove
Content type: text/plain File type: .json	

Figure 5-7. Declarations

After adding the file type association declaration, your app can load *.json files from File Explorer, as shown in Figure 5-8.

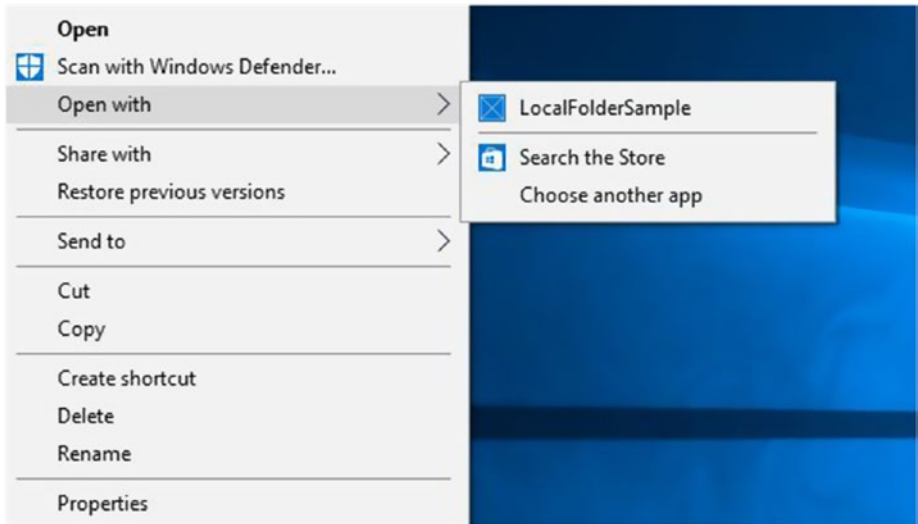


Figure 5-8. Opening .json files with the app

SQLite

One of the main benefits of using a repository is that we can swap in SQLite for our data store. We create a new repository based on the `IDataRepository` interface, and the rest of our code largely remains intact. In fact, the view model remains unchanged, while `Customer.cs` and `MainPage.Xaml.cs` just receive a tiny tweak. To see this in action, create a new project based on the Blank App template named `SQLiteSample`.

We are going to be using Entity Framework Core in order to do the actual communication to the SQLite database. Because of that, there are two prerequisites that we need to download: `Microsoft.EntityFrameworkCore.Sqlite` and `Microsoft.EntityFrameworkCore.Tools`. Once that is installed, you might have to restart Visual Studio. If you don't restart Visual Studio, you will have issues when it comes to setting up your migrations.

Now that we have the prerequisites installed, we need to create our `DataContext`. This is our repository and it follows the repository pattern. We will still continue to use the `IDataRepository<T>` that we created in previous examples as well. We will create our `CustomerContext` class, which inherits from `DbContext`. We also override `OnModelCreating` so that we can use SQLite and we pass it in a data source. In this example, we are using a file named `customer.db`. See Figure 5-9.

```
public class CustomerContext : DbContext
{
    public DbSet<Customer> Customers { get; set; }

    private static readonly string DataSource = "customer.db";
    protected override void OnModelCreating(DbContextOptions
    Builder optionsBuilder)
    {
        optionsBuilder.UseSqlite($"Data Source={DataSource}");
    }
}
```

We have a `DbSet` that has type `Customer`, which we named `Customers`. `DbSet` is the equivalent to a table in SQL. What that code is saying is that we have a table named `Customers`.

Before this code will work, however, we need to use the following namespace:

```
using Microsoft.EntityFrameworkCore;
```

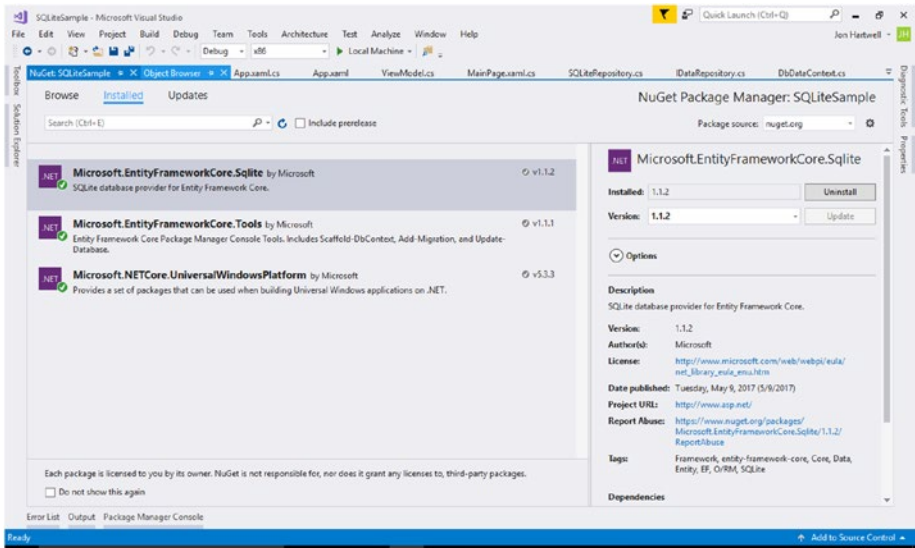


Figure 5-9. SQLite references for Windows 10 UWP

Finally, we need to install the `Microsoft.EntityFrameworkCore.Sqlite` NuGet package. This time, rather than using the console, choose **Tools** ► **Library Package Manager** ► **Manage NuGet Packages for Solution**, which brings up the NuGet GUI. From there, you can search for and install SQLite, as shown in Figure 5-10.

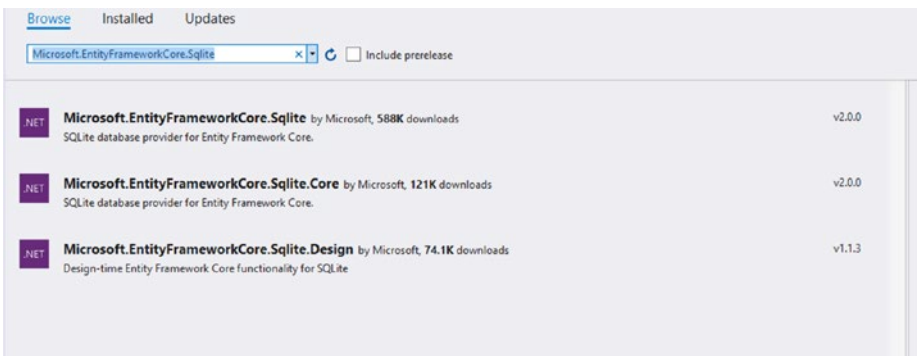


Figure 5-10. `Microsoft.EntityFrameworkCore.Sqlite` NuGet package

Just like with the earlier example, create a folder named `Data` and add the `IDataRepository`, `ViewModel`, and `Customer` classes from the `LocalFolder` sample. Because we can leverage the fact that `SQLite` is a database engine, we need to do some set up with `NuGet` first in order to get our dependencies. You will want to get the following packages—`Microsoft.EntityFrameworkCore.Sqlite` and `Microsoft.EntityFrameworkCore.Tools`—as shown in [Figure 5-11](#).

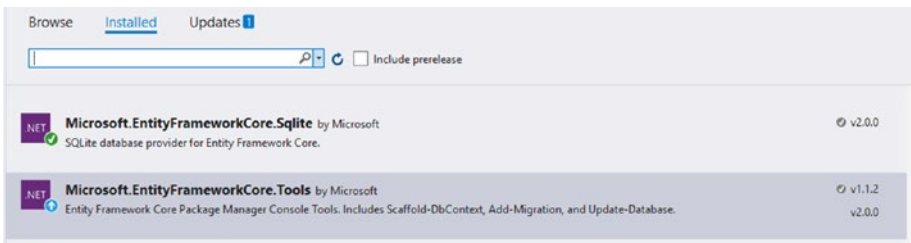


Figure 5-11. *The NuGet packages needed for SQLite*

Once you have the dependencies installed, you need to create a class that inherits from `DbContext`. Name the class `CustomerContext` and implement it as shown here.

```
public class CustomerContext : DbContext
{
    public DbSet<Customer> Customers { get; set; }

    private static readonly string DataSource = "customer.db";
    protected override void OnConfiguring(DbContextOptions
    Builder optionsBuilder)
    {
        optionsBuilder.UseSqlite($"Data Source={DataSource}");
    }
}
```

We inherit from `DbContext` and then create a public property named `Customers` that is of type `DbSet<Customer>`. The `DbSet` class lets the Entity Framework know that this is a table in the database that maps to the type `Customer`, which we will see the implementation of `Customer` next. We add a private string member that gives us our database name and override the `OnConfiguring` method to ensure we use SQLite instead of any other database engine. This `Customer` class looks plain, with no real differences from before.

```
public class Customer
{
    public int Id { get; set; }
    public string Email { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Title { get; set; }
}
```

Now, create a new class in the `Data` directory named `SQLiteRepository`, and implement the `IDataRepository` interface. Start by adding a `using` for `Windows.Storage`, `System.Collections.ObjectModel`, and `System.IO`:

```
using Windows.Storage;
using System.Collections.ObjectModel;
using System.IO;
```

Next, add a class-level variable for the location where we want to store the database:

```
private static readonly string dbPath = Path.Combine(
    ApplicationData.Current.LocalFolder.Path, "app.SQLite");
```

We also declare an `ObservableCollection<Customer>` as follows:

```
private ObservableCollection<Customer> customers;
```

Once again, the constructor calls `Initialize`, but this time `Initialize` has real work to do.

```
public SQLiteRepository()
{
    Initialize();
}
public async void Initialize()
{
    using (var db = new CustomerContext())
    {
        if (!db.Customers.Any())
        {
            db.Database.Migrate();
            db.Customers.Add(new Customer { FirstName = "Phil",
                LastName = "Japikse" });
            db.Customers.Add(new Customer() { FirstName =
                "Jon", LastName = "Galloway" });
            db.Customers.Add(new Customer() { FirstName =
                "Jesse", LastName = "Liberty" });
            db.Customers.Add(new Customer() { FirstName =
                "Jon", LastName = "Hartwell" });
            db.SaveChanges();
        }
        else
        {
            await Load();
        }
    }
}
```


The first step in `Initialize` is to instantiate an instance of the `CustomerContext`, which is the backing of our SQLite database. The `using` statement, of course, ensures that this resource will be released the moment we're done with it. SQLite is built using COM and other unmanaged resources, so it's extremely important to make sure all of the resources are disposed of. We then check to see if there are any customers already in the database. If there are no customers, we call `Migrate()` on the `Database` property of our `CustomerContext` instance. This will create the table and we will then populate it with hardcoded data. If there is already customer data in the SQLite database, we just call `Load` as we did previously.

```
public async Task<ObservableCollection<Customer>> Load()
{
    using (var db = new CustomerContext())
    {
        return new ObservableCollection<Customer>(await
            db.Customers.ToListAsync());
    }
}
```

`Load` will connect to the SQLite instance using the `CustomerContext` class and then make an async call to turn the `Customers` table into a `List<Customer>`. We then use that to populate our new `ObservableCollection<Customer>`, which is bound to the UI.

Similarly, the `Add` and `Remove` methods make a connection to the database and insert or delete the customer, respectively. We want to ensure that we call `SaveChangesAsync` with the parameter of `true` so that we are sure the changes make it back to the database. As of this writing, there are issues with SQLite and Entity Framework where if you call `SaveChangesAsync` without passing in a parameter of `true` for the `acceptAllChangesOnSuccess` parameter, the changes won't get persisted to the database.

```
public Task Add(Customer customer)
{
    using(var db = new CustomerContext())
    {
        db.Customers.Add(customer);
        return db.SaveChangesAsync(true);
    }
}

public Task Remove(Customer customer)
{
    using (var db = new CustomerContext())
    {
        db.Customers.Remove(customer);
        return db.SaveChangesAsync(true);
    }
}
```

Next, the Update method, much as it did in the previous cases, finds the original value for the customer and removes the old value and replaces it with the new value. It then does an update on the customer in the database.

```
public Task Update(Customer customer)
{
    using (var db = new CustomerContext())
    {
        var existingCustomer = db.Customers.Find(customer);
        existingCustomer = customer;
        db.Customers.Update(existingCustomer);
        return db.SaveChangesAsync(true);
    }
}
```

Finally, we copy the XAML after the Page directive from MainPage.xaml in the previous example into MainPage.xaml in this example. Also, we copy all of the code from MainPage.xaml.cs starting with the public-sealed partial class from the previous example and paste it into MainPage.xaml.cs in this example. Copying the XAML and the C# code in this way preserves the namespaces and usings in the files. We then just need to update one line of code where we instantiate the IRepository. The update line should read like this:

```
private IRepository data = new SQLiteRepository();
```

You can see that we were able to carry the repository pattern that we used in the first example all the way forward to the SQLite example successfully, saving us a lot of rethinking and redesign. You have a number of choices for where and how you store user data, but none of them is terribly difficult to implement.

Summary

There are a variety of options to choose from when your app needs to store data locally on the device. Simple data can be stored in application settings or in files on the device. More complex data can be stored in SQLite. One noticeably absent item from the list is the SQL Server. The SQL Server doesn't run on ARM devices or under WinRT.

If your app needs to access the SQL Server, it must be done through a service. In the next chapter, we look into accessing remote data.

CHAPTER 6

Application Lifecycle

In Windows 7, as in all prior versions of Windows (and virtually every other operating system for that matter), the lifecycle of an application was largely determined by the user. The user started the application and, when done, closed the application. The exceptions were crashes and other unforeseen terminations. For Windows 8.1 modern design apps, the process is very different from Windows 7 (or even Windows 8.1 desktop applications). Users can start apps in the usual way by tapping (or clicking) on the app's tile, but they can also start with a number of new methods, including searching and sharing. They can switch from app to app and when they are finished, they typically do not close the application but simply open another one.

With the release of Windows 8, the operating system plays a much larger role in the lifecycle of each application in the new version. When an app no longer has the focus, Windows 8.1 suspends it. A suspended app is not running in the background but truly idle. This means no processor time at all, and only its state is retained in memory. If the user then brings that app back into the foreground, it once again receives processor time and returns to normal operations.

Suspended apps can also be terminated by the operating system. The most common reason for the termination of an app is memory pressure, although there are some other ways termination can occur. Users also do not know that their apps have been terminated other than the fact it is no longer in the list of running apps.

Windows 10 Universal Windows Platform (UWP) builds on what was developed in Windows 8. Like Windows 8, Windows 10 UWP has the operating system playing a larger role. The operating system can suspend or terminate applications and will do so if resources are needed. With the release of Windows 10 Version 1607, however, Microsoft brought back the concept of apps running in the background. Instead of going directly to the suspended state, an app may be changed to running in background. The big difference in Windows 10 UWP running in the background compared to a normal Win32 application running in the background is that Windows 10 will limit the amount of resources that a Windows 10 UWP application can consume while running in the background.

The Three Application States

There are three app states: running, suspended, and terminated. As just mentioned, the operating system (and not the user) plays a key role in determining the state of an application. This was implemented for a number of reasons, but the two that figured most prominently in the decision were to match how users work with mobile devices and to conserve battery power and other resources.

In the case of mobile devices, unlike working with a plugged-in desktop (or laptop), they are designed to be, well, *mobile!* Battery life is a crucial element in deciding which mobile device a consumer selects. Running multiple applications in the foreground would drain the battery in a very short period of time. That being said, as of Windows 10 Version 1607, applications can now run not only in the foreground but also in the background. Applications running in the background are much more limited than the traditional applications. They are given a memory limit and can still be closed if the system is low on resources.

Tablets have been out long enough that the typical usage pattern is well known. When users have an app on the screen, it is the focus of their

attention. When an app is moved to the background, then it is out of sight, out of mind.

Running

If your user can see your app on his device, it is *running*. Running apps get processor time, memory, access to device hardware, and everything else one would expect.

Suspended

When the user moves an app to the background (removes it from view), the app goes into the *suspended* state. As an app moves into the background, it has five seconds to store its state. Once the app is in the background, it is completely isolated from the operating system. Anything in memory in the app will stay there, but it has no access to processor threads or any other resources. Therefore, as a developer, you must treat the suspension event as if your app will be terminated.

The lack of resource utilization minimizes power consumption for apps that are not in the foreground, helping to extend the battery life of the device. It also helps to keep foreground apps much more responsive, as system resources are dedicated to apps in the foreground (which are visible to the user).

By retaining in memory the state of the app, the user can switch back to the app by bringing it into view in a near instant, which gives the illusion that it was running all along in the background.

Terminated

As discussed earlier, the operating system can *terminate* apps when certain conditions exist, most often memory pressure. Since the app does not have access to any system resources (except for the memory that it was using

when it moved to suspended), no notification is given when a termination happens.

While the operating system certainly has an algorithm that it follows in determining which apps to terminate, this information is not available to you as a developer, and you must assume that as soon as your app is suspended it will terminate. If you did not save your app's data and state (including session and navigation), it will be lost. The next time a user launches your app, it will appear as if it crashed. And in reality, if your app doesn't handle saving its state on suspend, it did crash.

State Transitions

App state transitions can happen due to user or operating system action. As shown in Figure 6-1, six actions can affect the state of an app: launching, activating, suspending, resuming, terminating, and killing.

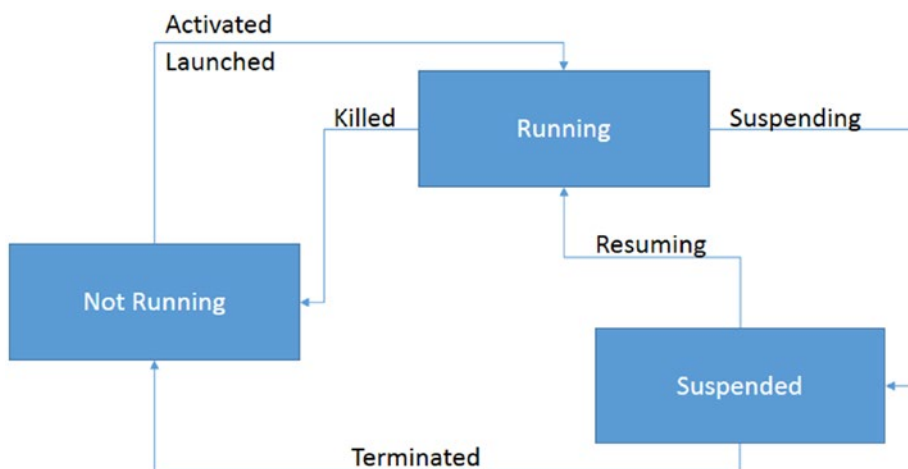


Figure 6-1. State transitions

Launching

Launching an app is a user-driven action. It occurs when a user clicks an app to load it. This places the app in the running state.

Activating

Another way for an app to move to running is activation. Activation occurs through the operating system. Some examples of activation include the Search and Share charms as well as notifications.

During activation, the arguments passed into the `OnLaunched` event provide information on the previous execution state. This enables you to determine how the app transitioned into the not-running state. If the previous execution state was terminated, you need to restore the state that was saved during the suspension process.

Suspending

The operating system will suspend an app when it is no longer in the foreground.

Resuming

When a suspended app returns to the foreground, the app is resumed. Since suspended apps maintain state, there is no need to do anything in code to provide the expected user experience.

Terminating

Your app may be terminated by the operating system for any number of reasons, including memory pressure requiring that your app be closed to allow other apps to run, the app taking too long to start, or the app taking too long to suspend.

Killing

An app changes directly from running to not running (skipping the suspended state) only when it is killed. An app can be killed by any of the following conditions:

- The user swiping down from the top of the screen (or pressing Alt-F4)
- The user logging off, shutting down, or restarting the computer
- The user killing the process in Task Manager
- The app freezing, taking longer than five seconds to suspend, or taking longer than 15 seconds to start
- The app exit being called

If the user performs a close gesture (swiping down from the top of the screen), the app suspends for 10 seconds before it gets killed. This provides a better user experience when an app is accidentally closed. The user can immediately relaunch the app and still maintain the same state.

Managing the Lifecycle

For the sample code in this chapter, we start with the Blank App template. To get started, create a new Blank App project by selecting File ► New Project ► Visual C# ► Windows Universal ► Blank App (XAML) and setting the name of the project to **LifeCycle**, as shown in Figure 6-2.

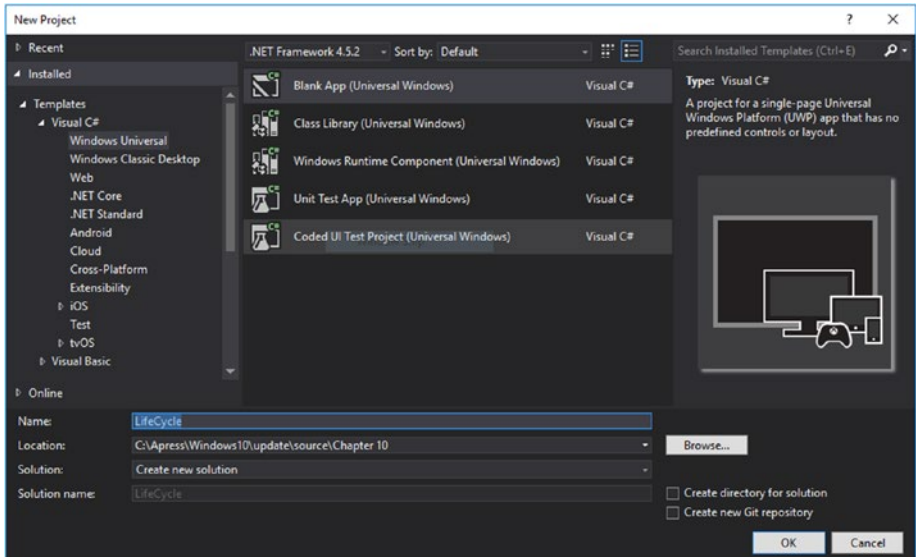


Figure 6-2. Creating the LifeCycle project

Building the Sample App

In order to demonstrate saving and restoring data and state, we need a simple app that has data and more than one page to prove that we can restore data as well as navigation. The Main Page (shown in Figure 6-3) includes a list of tasks that need to be completed. Clicking on the Add button or selecting an item and clicking Edit will bring the user to the Details Page (shown in Figure 6-4), which allows users to add to or edit the list values. This app is very simplistic on purpose because the main goal is to show how to save and restore state and app data.

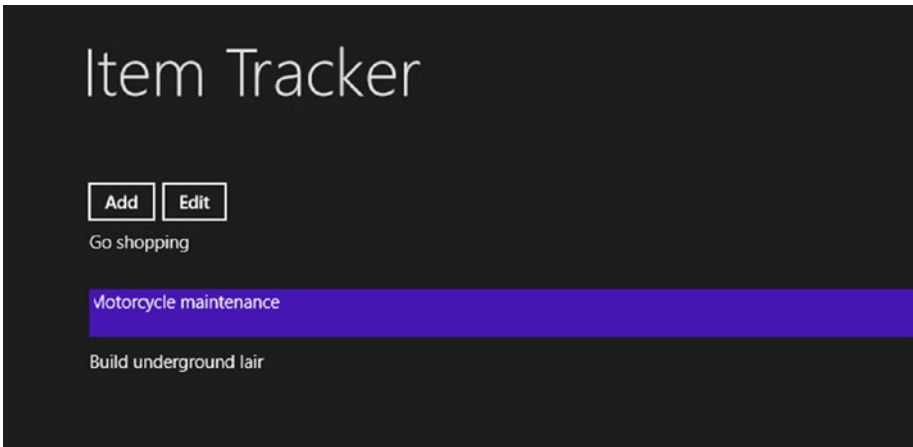


Figure 6-3. Main Page of the Item Tracker app

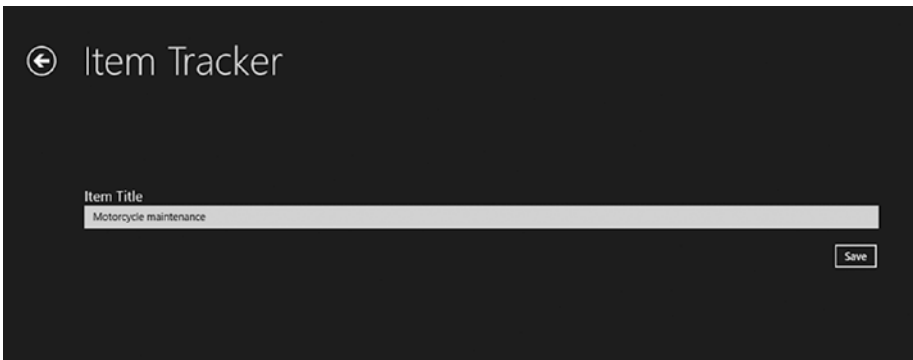


Figure 6-4. Details Page of the Item Tracker app

Adding the Navigation Parameter Class

When navigating from one page to another, any object can be passed as a parameter to the new page. In our sample app, we need a custom object to pass two values back to the Main Page from the Details Page.

To do so, add a new class to the project named `NavigationParameter.cs`. In that class, enter the following code:

```
public class NavigationParameter
{
    public string OriginalItem { get; set; }
    public string UpdatedItem { get; set; }
}
```

Creating the Details Page

The Details Page allows the user to add or edit tasks. Right-click on the project, add a Blank page named `Details.xaml`, and complete the following steps:

1. Open `Details.xaml` and add the following markup just before the closing `</Grid>` tag:

```
<Grid Row="1" Margin="120">
    <StackPanel>
        <TextBlock FontSize="20">Item Title</TextBlock>
        <TextBox Name="ItemTitle" Text="New Item" />
        <Button Name="Submit" Click="Submit_Click"
            HorizontalAlignment="Right" Margin="0,20,0,0">
            Save</Button>
    </StackPanel>
</Grid>
```

2. Add the stub for the `Submit_Click` method in `Details.xaml.cs`:

```
public void Submit_Click(object sender, RoutedEventArgs e)
{
    var param = new NavigationParameter
    {
        OriginalItem = this.OriginalItem,
        UpdatedItem = ItemTitle.Text
    };

    this.Frame.Navigate(typeof(MainPage), param);
}
```

3. Add a variable to hold the original value if the user is editing an existing list item. Add this code just before the constructor:

```
private string OriginalItem = string.Empty;
```

4. Add the following method in `Details.xaml.cs` to handle the parameter data (if any):

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    if(e.Parameter != null)
    {
        string param = e.Parameter as string;
        if(!string.IsNullOrEmpty(param))
        {
            OriginalItem = param;
            ItemTitle.Text = param;
        }
    }
}
```

```

        else
        {
            ItemTitle.Text = "Add a new
            item...";
        }
    }
}

```

Creating the Main Page

The Main Page contains two buttons and a `ListView` control that will be a simple Item Tracker/ToDo app. Open `MainPage.xaml` and complete the following steps:

1. Open `MainPage.xaml` from the project previously created. Add the following XAML just before the closing `</Grid>` tag:

```

<StackPanel Grid.Row="1" Margin="120,30,0,0">
    <StackPanel Orientation="Horizontal">
        <Button Name="Add"
            Content="Add"
            Click="Add_Click" />
        <Button Name="Edit"
            Content="Edit"
            Click="Edit_Click" />
    </StackPanel>
    <ListView Name="ItemsList"/>
</StackPanel>

```

2. Open MainPage.xaml.cs and add the two-button click method handlers and the following code:

```
public void Edit_Click(object sender, RoutedEventArgs e)
{
    this.Frame.Navigate(typeof(Details), ItemsList.
        SelectedValue);
}

public void Add_Click(object sender, RoutedEventArgs e)
{
    this.Frame.Navigate(typeof(Details), string.Empty);
}
```

3. Add starter data to the ListView by copying the following code to the top of the class just before the constructor:

```
string[] _defaultItems = {"Go shopping", "Motorcycle
maintenance", "Build underground lair"};
string[] _items;
```

4. Add a using for Windows.Storage at the top of the file:

```
using Windows.Storage;
```

5. Add an application data container to save the application data:

```
private ApplicationDataContainer _settings =
ApplicationData.Current.RoamingSettings;
```

Handling Adding/Editing List Items

The `OnNavigatedTo` method is executed whenever a page is navigated to. The following code first checks to see if there are items in the application data container's roaming settings, which have been added to the `OnNavigatedTo` method, and then checks to see if a navigation parameter was passed in. If so, the list gets updated accordingly, the app data is saved in the application data container, and the `ItemsSource` property for the `ListView` is set to the item's collection. On the `MainPage.xaml.cs` file, add this code:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    _items = _settings.Values["StoredItems"] as string[] ?? _
    defaultItems;
    var param = e.Parameter as NavigationParameter;
    if(param != null && !string.IsNullOrEmpty(param.
    UpdatedItem))
    {
        List<string> items = new List<string>();
        if(!string.IsNullOrEmpty(param.OriginalItem))
        {
            items.Remove(param.OriginalItem);
        }
        items.Add(param.UpdatedItem);
        _settings.Values["StoredItems"] = items.ToArray();
    }
    ItemsList.ItemsSource = items;
}
```


Responding to App Suspension

As previously mentioned, the operating system notifies apps on suspension, but not when they are killed or terminated. Until the release of Windows 10 Version 1607, you would handle saving your state in the `OnSuspending` event. With the release of Version 1607 came the addition of running in the background and running in the foreground. Since Windows 10 Version 1067, any saving of state should happen in the `EnteredBackgroundEvent`. For more information on saving app data to local or roaming settings, refer to [Chapter 6](#).

The `OnSuspending` Event

If you are using one of the templates provided, the app-suspending event already has a handler called `OnSuspending` in `App.xaml.cs`.

The out-of-the-box version has three lines as shown here:

```
private void OnSuspending(object sender, SuspendingEventArgs e)
{
    var deferral = e.SuspendingOperation.GetDeferral();
    //TODO: Save application state and stop any background
    activity
    deferral.Complete();
}
```

The call to `GetDeferral` returns an object of type `SuspendingDeferral` and it has only one method: `Complete`. You need to sandwich your work between these two calls if you are working with asynchronous methods. As explained in the “Using Async Methods” section, the `SuspensionManager` calls are async calls. When using async, the `OnSuspending` method will run until you call `Complete` or the five seconds allocated is exhausted, whichever comes first. If you are running on versions of Windows 10 prior to Version 1067, this is where you would handle the save data.

The EnteredBackground Event

With Windows 10 Version 1607, saving of state should be all done in the EnteredBackground event. This event signals that the application is now running in the background rather than the foreground. There are extra hoops we must jump through as well, though. When an application is put into the Running in Background state, the amount of memory that is available to that application decreases. Because of that, we will need to handle the memory usage events manually too.

In order to keep track of whether our app is running in the background or not, we need to create a member `bool` as follows:

```
private bool isInBackground;
```

We then need to attach to the following events: EnteredBackground, LeavingBackground, AppMemoryUsageLimitChanging, and AppMemoryUsageIncreased. We do this in the constructor of the App class in App.xaml.cs. We also initialize our `isInBackground` variable to `false` to indicate that we are not running in the background currently.

```
public App()
{
    this.InitializeComponent();
    this.Suspending += OnSuspending;
    this.EnteredBackground += OnEnteredBackground;
    this.LeavingBackground += OnLeavingBackground;
    Windows.System.MemoryManager.AppMemoryUsageLimitChanging +=
    MemoryManager_AppMemoryUsageLimitChanging;
    Windows.System.MemoryManager.AppMemoryUsageIncreased +=
    MemoryManager_AppMemoryUsageIncreased;

    isInBackground = false;
}
```

Because we do not have any heavy objects in this example, `OnEnteredBackground` and `OnLeavingBackground` are rather bare:

```
private void OnEnteredBackground(object sender,
EnteredBackgroundEventArgs e)
{
    isInBackground = true;
}
```

First, the `OnEnteredBackground` event handler. As stated before, we don't have many objects in our application, so there isn't much to save (we are already saving on edit as well). We just need to ensure our application knows we are entering the Running in Background state by setting `isInBackground` to `true`.

While our application is running in the background, we must ensure that it doesn't utilize more resources than it is allowed. We do this with the `AppMemoryUsageLimitChanging` and `AppMemoryUsageIncreased` events. The `AppMemoryUsageLimitChanging` event will fire whenever there is a change in how much memory our application is allowed to consume. This will typically occur when the application is first being run in the background and when the application transitions to running in the foreground again. We are only concerned with when the memory limit decreases because that means we may need to take action. The following code shows the implementations of the `AppMemoryUsageIncreased` and `AppMemoryUsageLimitChanging` events that are in `App.xaml.cs`.

```
private void MemoryManager_AppMemoryUsageIncreased(object
sender, object e)
{
    var memUsage = Windows.System.MemoryManager.
AppMemoryUsageLevel;
```

```

    if (memUsage == Windows.System.AppMemoryUsageLevel.
        OverLimit
        || memUsage == Windows.System.
            AppMemoryUsageLevel.High)
    {
        FreeMemory(Windows.System.MemoryManager.
            AppMemoryUsageLimit);
    }
}

private void MemoryManager_AppMemoryUsageli
mitChanging(object sender, Windows.System.
AppMemoryUsageLimitChangingEventArgs e)
{
    if (Windows.System.MemoryManager.AppMemoryUsage >=
        e.NewLimit)
    {
        FreeMemory(e.NewLimit);
    }
}

```

When the memory usage increases, we want to check our current usage level and then determine if it is over the limit or if it is high. If it is either one of those, we need to take action. Whereas when the usage limit changes, we only need to check to see if our usage is above the new memory limit. If it is, then we must ensure that we do what we can to release memory, using the `FreeMemory` method.

```

private void FreeMemory(ulong size)
{
    if (isInBackground && Window.Current.Content != null)
    {

```

```

        Window.Current.Content = null;
    }
    GC.Collect();
}

```

The `FreeMemory` method determines if we are running in the background and if the current UI is null. If we are running in the background and we haven't disposed of the UI, then we are able to easily take advantage of the space we can gain by disposing of the UI elements. We set the current UI content to `null` and then call the garbage collector. If we had larger objects that we knew we did not need, we could also clean those up in this instance.

When the application is starting to run in the foreground again, the `LeavingBackground` event is fired. Here we set our member variable `isInBackground` to `false` to indicate we are no longer running in the background and then determine if the current UI is null. Remember, there is a chance we could have disposed of the UI elements in the `FreeMemory` method. Therefore, if we need to ensure that we do not reference null objects, we call `CreateRootFrame`.

```

private void OnLeavingBackground(object sender,
LeavingBackgroundEventArgs e)
    {
        isInBackground = false;
        if (Window.Current.Content == null)
        {
            CreateRootFrame(ApplicationExecutionState.
                Running, string.Empty);
        }
    }
}

```

The `CreateRootFrame` method takes part of the `OnLaunched` method and creates the new UI, giving it the default look.

```
private void CreateRootFrame(ApplicationExecutionState
previousState, object arguments)
{
    Frame rootFrame = Window.Current.Content as Frame;

    // Do not repeat app initialization when the Window
    already has content,
    // just ensure that the window is active
    if (rootFrame == null)
    {
        // Create a Frame to act as the navigation
        context and navigate to the first page
        rootFrame = new Frame();

        rootFrame.NavigationFailed +=
        OnNavigationFailed;

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

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

Responding to App Activation

When an app is activated, the `OnLaunched` event is raised. We have already used this when we registered the root frame of the app. This method is also where the app session is restored.

Determining the Previous Application State

The custom event argument for the `OnLaunched` event (`LaunchActivatedEventArgs`) contains a `PreviousExecutionState` property of type `ApplicationExecutionState`. The `ApplicationExecutionState` enumeration contains values representing the five possible states that an app can be in prior to activation:

- `NotRunning`
- `Running`
- `Suspended`
- `Terminated`
- `ClosedByUser`

Testing the Restoring Navigation State

Visual Studio makes it very easy to test termination and activation. While an app is running, locate the Suspend menu. It will vary based on customization, but the default location is the left side of the third row of the toolbars. (If it isn't showing, you can enable the toolbar by selecting `View ► Toolbars ► Debug Location`.) The Suspend menu gives you the option to suspend, resume, or suspend and shut down the app being debugged. This menu is shown in [Figure 6-5](#).

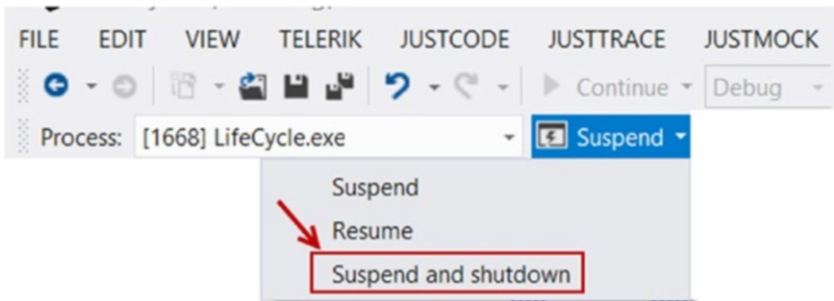


Figure 6-5. *Suspend options in Visual Studio*

Follow these steps to run the test:

1. Run the app.
2. Click Add on the Main Page to navigate to the Details Page.
3. Go back to Visual Studio, and on the Suspend menu, click Suspend and Shutdown. When the app has completed termination, Visual Studio's menu bar returns to the non-running state.
4. Run the app again, and the app will load with the Details Page as the current page.

Testing the Restoring Session Information

To test the restoration of the app state information, use the same process as the one for testing the restoration of the navigation information:

1. Run the app.
2. Select one of the items in the list.

3. Go back to Visual Studio, and on the Suspend menu, click Suspend and Shutdown. When the app has completed termination, Visual Studio's menu bar returns to the non-running state.
4. Run the app again, and the app will load to the Main Page with the previous item selected once again.
5. Click Edit on the Main Page to navigate to the Details Page. The text box is filled in with the selected item. Make an edit to the text box.
6. Go back to Visual Studio and repeat Step 4—click Suspend and Shutdown. When the app has completed termination, Visual Studio's menu bar returns to the non-running state.
7. Run the app again, and the app will load to the Details Page with the text box restored to the previous value.

Summary

Because Windows 10 UWP apps can run on many different devices with many different hardware profiles, the app lifecycle has been updated to ensure that applications can still run in the background but not overconsume resources. One of the mechanisms utilized prevents background apps from stealing precious resources when the user isn't actively engaged. When an app isn't visible on the screen, it may go into a suspended state or it may run in the background. While suspended, your app doesn't have access to any processor cycles. While running in the background, the amount of resources that an app is allowed to consume is drastically limited. If system resources become low enough, Windows will terminate your app.

It is important for your app to gracefully handle app termination. If your app doesn't restore session data and state after termination, then the user will believe (rightly so) that the app crashed and isn't stable. By saving the session data and state when an app is suspended (and subsequently restoring them when an app is started again after termination), the user experience will be seamless and the user will not have any cause for concern.

Index

A

Application lifecycle

activation

ApplicationExecution

State, 216

restoring navigation

state, 216–217

restoring session

information, 217–218

state transitions, 200

activation, 201

killing, 202

launching, 201

resuming, 201

suspend, 201

terminating, 201

suspension

EnteredBackground

event, 211–215

OnSuspending event, 210

SuspensionManager

Blank App, 202–203

Details page creation, 205–207

Item Tracker app, 203–204

Main page creation, 207–208

navigation parameter

class, 204–205

OnNavigatedTo method, 209

Windows 10

running, 199

suspended state, 199

terminate apps, 199–200

B

Binding

collection

array, 136

data-bound ListView, 136

IEnumerable<Person>, 137

INotifyCollection

changed, 143

ListView creation, 141, 143

Person class, 138–141

DataContext, 120

DataConverter

boolean, 148–149

ConvertBack, 149

IValueConverter, 148–149

methods, 145

Num property, 145

Person class, 146, 147

elements

accept/continue

button, 128–129

check box button, 129

XAML, 128

INDEX

Binding (*cont.*)

errors

- FallbackValue, 126
- LastName property, 125
- TargetNullValue, 127

INotifyPropertyChanged

- EventArgs, 133
- Person class, 133, 135
- text blocks, 135
- UI, 134

modes

- OneTime, 130
- OneWay, 130
- TwoWay, 131

Person class creation, 121

statements, 123

text blocks property, 121

UpdateSourceTrigger, 131–132

user interface, 119

XAML, 122–123

Border control, 77, 78

C

CheckBox controls, 95

ComboBox controls, 100

Controls

- CheckBox, 95
- ComboBox, 100–101
- DatePickers and TimePickers, 106
- event handler, 92
- flyouts
 - basic, 109
 - definition, 108
 - menu, 110–111

headers and watermarks, 86

HyperLinkButton, 94

Image controls, 101, 102

ListBox, 98, 99

ListView, 99

PasswordBox, 89

ProgressBar control, 104, 105

RadioButton, 95–96

Slider control, 104

spell check, 83

StackPanel, 96

Stretch property, 103

TextBlock and TextBox, 79

TextBox instances, 90

toggle button, 97

ToggleSwitch, 95

tooltips, 106

Controls5a, 86

CRUD operations, 159

D, E, F

Data binding statements, 123

Data-bound ListView, 141

DataContext, 120

Dependency properties

- Callback method, 115
- data hiding, 111–112
- DependencyObject, 114
- inputs, 113
- Register method, 115

Distributed version control system (DVCS), 19

G

Git

- committing changes, [22](#)
- DVCS, [19](#)
- functions, [23-24](#)
- GitHub for Windows, [19](#)
- Microsoft Git provider
 - selection, [20](#)
- remote repository, [23](#)
- repository settings, [20-21](#)
- SCM system, [19](#)
- TFS, [18](#)
- TFVC, [18](#)

Graphical User Interface (GUI), [26](#)

H

Headers and watermarks

- controls, [86](#)

I

Image control, [101, 102](#)

J, K

JavaScript Object Notation

- (JSON), [166](#)

Json.NET package, [167](#)

L

ListBox control, [98-101](#)

ListView control, [98-101](#)

Local data

application data

- data file, [179](#)
- data storage, [153](#)
- magic strings, [155](#)
- MainPage.xaml.cs
 - file, [154, 155](#)
- Remove method, [157](#)
- retrieving text, [156](#)
- settings containers, [152](#)
- stores, reads, and
 - deletes, [152](#)

file repository

- Add method, [170](#)
- creation, [168](#)
- FileRepository class, [169](#)
- Initialize method, [169](#)
- Load method, [172](#)
- OpenFileAsync method, [170](#)
- Remove method, [171](#)
- System.Collections.
 - ObjectModel and
 - Windows.Storage, [169](#)
- Update method, [171](#)

implement event handlers, [177](#)

JSON, [166](#)

layer

- creating repository
 - interface, [159](#)
- DataModel, [160](#)
- ViewModel creation
 - (*see* ViewModel)

local, roaming, and temporary

- data container, [167](#)

INDEX

Local data (*cont.*)

roaming, [179](#)

Save and Delete command

buttons, [175](#)

SQLite

Add and Remove

methods, [193](#)

advantages, [187](#)

COM and unmanaged

resources, [193](#)

database engine, [190](#)

SQLite NuGet package, [189](#)

Update method, [194](#)

Windows.Storage and

System.Collections.

ObjectModel, [191](#)

XAML and C# code, [195](#)

stack panels, [175](#)

UI, [173-174](#)

user-specified locations

FileOperations

class, [180-183](#)

file associations,

JSON files, [184-186](#)

ViewModel, [177](#)

M

MainPage.xaml file, [8](#)

Microsoft Store, [2](#)

Model-View-ViewModel (MVVM)

app testing, [51](#)

creation, [47](#)

customer model

adding customer

properties, [42](#)

INotifyProperty

Changed, [44-46](#)

DataContext, [49](#)

guidance, [52](#)

patterns, [40-41](#)

view creation, [49](#)

N

NuGet, [24](#)

command line

installation, [25-26](#)

GUI, [26](#)

Json.NET installation, [26-27](#)

package source, [24-25](#)

O

OnNavigatedTo method, [209](#)

P, Q

Package.appxmanifest file

Application tab, [35](#)

Capabilities tab, [37](#)

Content URIs tab, [38](#)

Declarations tab, [38](#)

Packaging tab, [38](#)

Visual Assets tab, [36](#)

Padding, [73-74](#)

Panels

- Border control, 77
- Canvas control
 - open MainPage.xaml, 65
 - rectangles and an ellipse, 68
 - Visual Studio, 66
 - XAML code, 67
 - ZIndex, 68
- flag design, 77
- Grid control
 - four values margins, 73
 - horizontalAlignment, 72
 - margin, 73
 - padding, 73–74
 - pair of values margins, 72
 - rows and columns, 69
 - single value margins, 72
 - VerticalAlignment, 72
- RelativePanel control, 76
- StackPanel control, 74
- PasswordBox control, 89
- Plain Old CLR Object (POCO), 119
- ProgressBar control, 104, 105

R

- RadioButton controls, 96
- RelayCommand class, 47

S

- Slider control, 104
- Source-code-management (SCM)
 - system, 18–19

Spell check, 83**SQLite**

- Add and Remove
 - methods, 193
- advantages, 187
- COM and unmanaged
 - resources, 193
- database engine, 190
- SQLite NuGet package, 189
- Update method, 194
- Windows.Storage and System.
 - Collections.Object
 - Model, 191
 - XAML and C# code, 195
- StackPanel controls, 96

T, U

- Team Foundation Server (TFS), 18
- Team Foundation Version Control
 - (TFVC), 18
- TextBlock and TextBox controls, 79
- Themes, 61
- TimePicker and DatePicker
 - controls, 107
- ToggleSwitch controls, 95
- Tooltips control, 106

V

- ViewModel
 - async method, 165
 - CRUD operations, 165
 - customers property, 164

ViewModel (*cont.*)

- implementing INotifyProperty

- Changed, [161](#)

- MainPageViewModel class, [47, 48](#)

- SelectedItem property, [164](#)

- System.Collections.

- ObjectModel, [164](#)

Visual Studio 2017

- Blend for

- assets tab, [16](#)

- benefits, [14-15](#)

- data tab, [17-18](#)

- MainPage.xaml, [15-16](#)

- objects and timeline, [17](#)

- page designer, markup, and

- code, [17](#)

- projects tab, [16](#)

- properties tab, [17](#)

- resources tab, [18](#)

- states tab, [17](#)

- versions

- Community Edition, [6](#)

- MSDN, [6](#)

- simulator, [6](#)

Windows 10 apps creation

- App.xaml file, [8](#)

- Basic Page file, [10-12](#)

- debugging with

- FrameRateCounter, [9](#)

- new project, [7](#)

- run toolbar utility, [8-9](#)

- simulator controls, [12-14](#)

- solution explorer files, [8](#)

W, X, Y

Windows 10 applications

- creation

- App1_TemporaryKey.pfx, [34](#)

- App.xaml file, [39](#)

- assets, [33-34](#)

- MainPage.xaml, [39](#)

- Package.appxmanifest (*see*

- Package.appxmanifest file)

- properties, [33](#)

- standard References node,

- [33](#)

- folders and files, [31-32](#)

- Git

- committing changes, [22](#)

- functions, [23-24](#)

- GitHub for Windows, [19](#)

- Microsoft Git provider

- selection, [20](#)

- remote repository, [23](#)

- repository settings, [20, 21](#)

- SCM system, [19](#)

- TFS, [18](#)

- Microsoft design

- cloud roaming, [5](#)

- connected and alive, [4](#)

- contracts, [4](#)

- fast and fluid, [3](#)

- sizing beautifully, [4](#)

- tiles, [4](#)

- UX guidelines, [5](#)

- Microsoft Store, [2](#)

MVVM (*see* Model-View-
 ViewModel (MVVM))
navigation
 back button, 58–59
 frame creation, 52–53
 MainPage, 55
 NavigatedTo Event
 handler, 58
 new page creation, 53–54
 UI creation, 54–55
NuGet
 command line
 installation, 25, 26

GUI, 26
Json.NET installation, 26–27
package source, 24–25
target and minimum
 version, 30–31
Visual Studio 2017 (*see* Visual
 Studio 2017)
Windows Presentation Foundation
 (WPF), 160

Z

ZIndex, 68