"This book doesn't just focus on singular concepts, it also provides end-to-end perspective on building an app in WinRT. It is one of those essential tools for Windows developers that will help you complete your software goals sooner than without it!"

—**Tim Heuer,** Principal Program Manager Lead, XAML Platform, Microsoft Corporation

Programming the Windows® Runtime by Example

A Comprehensive Guide to WinRT with Examples in C# and XAML

Microsoft Windows Development Series

Jeremy **Likness**
John **Garland**

Wintellect
Know how.

Praise for
Programming the Windows Runtime by Example

"This is a great from-the-ground-up, very complete book on building Windows Store Apps. You'll find it on your desk a year from now all dog-eared and marked up from use."

Dave Campbell, MVP, WindowsDevNews.com

"*Programming with Windows Runtime by Example* is a must-have book for any professional developer building apps for WinRT/Win8.1, especially in the LOB space for modern apps on Windows 8.1. For me it is the reference I provide my team building LOB applications for WinRT. Jeremy and John have done a great job putting together a great reference and educational book on professional development for the WinRT platform."

David J. Kelley, CTO, Microsoft MVP

"Jeremy and John are both very much IT masters from the old guard of software development. With countless years of bending, shaping, and influencing the world of software development behind them both, they continue to do so as they push forward into new and emerging technologies.

"As with everything they do, this book also reflects their ongoing dedication and passion for their quest to bring the reader not only the information he or she requires, but far more beyond that, they build knowledge step-by–step, then deliver it to the reader with cutting-edge, ninja-like precision to deliver exactly what knowledge is needed, when it's needed, and where it's needed.

"If you want to learn the Windows Runtime, then I can think of no finer book, and no finer guides to the WinRT landscape. By the end of this book, you'll have the knowledge, the power, and a hefty dose of passion to go out into the new millennium and create some of the best WinRT apps available."

Peter "Shawty" Shaw, LinkedIn .NET User Group manager

"This book is an invaluable resource for budding WinRT developers. It covers the basics to more advanced topics like MVVM. Readers will find the chapter entitled 'Connecting to the Cloud' especially useful in getting up to speed with Azure and creating cloud connected apps."

Daniel Vaughan, President of Outcoder, Microsoft MVP,
Author of *Windows Phone 8 Unleashed*

"There are books that provide reference for a development topic, and others that you will read from cover to end. *Programming the Windows Runtime by Example* by Jeremy Likness and John Garland should be your go-to guide for getting up to speed on WinRT. Jeremy and John wrote this book with the intention of being easy to follow and hard to forget, and they succeeded in both areas. I recommend this book for all developers, whether new to WinRT development, or those like me who just want to fill in the gaps on advanced topics."

Chris Woodruff, DeepFriedBytes.com, Microsoft MVP

Microsoft Windows Development Series

Addison-Wesley

Visit informit.com/mswinseries for a complete list of available publications.

The Windows Development Series grew out of the award-winning Microsoft .NET Development Series established in 2002 to provide professional developers with the most comprehensive and practical coverage of the latest Windows developer technologies. The original series has been expanded to include not just .NET, but all major Windows platform technologies and tools. It is supported and developed by the leaders and experts of Microsoft development technologies, including Microsoft architects, MVPs and RDs, and leading industry luminaries. Titles and resources in this series provide a core resource of information and understanding every developer needs to write effective applications for Windows and related Microsoft developer technologies.

"This is a great resource for developers targeting Microsoft platforms. It covers all bases, from expert perspective to reference and how-to. Books in this series are essential reading for those who want to judiciously expand their knowledge and expertise."

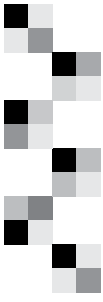
– JOHN MONTGOMERY, Principal Director of Program Management, Microsoft

"This series is always where I go first for the best way to get up to speed on new technologies. With its expanded charter to go beyond .NET into the entire Windows platform, this series just keeps getting better and more relevant to the modern Windows developer."

– CHRIS SELLS, Independent Consultant specializing in Windows, devices, and the cloud

Make sure to connect with us!
informit.com/socialconnect

Addison Wesley | informIT the trusted technology learning source | Safari Books Online



Programming the Windows Runtime by Example

A Comprehensive Guide to WinRT with Examples in C# and XAML

- Jeremy Likness
- John Garland

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Control Number: 2013954295

ISBN-13: 978-0-321-92797-2
ISBN-10: 0-321-92797-4

Text printed in the United States on recycled paper at Edwards Brothers Malloy, Lillington, North Carolina

First printing, June 2014

For Doreen and all her arrows, Lizzie and all her travels,
and Gordon and all his paint.
—Jeremy Likness

To Karen, Callie, Winnie, and Dude,
for the new adventure that is soon to begin.
—John Garland

Contents at a Glance

Contents

This page intentionally left blank

Foreword

The concept of an app has changed dramatically over time, and more increasingly so in the past eight years. The approachability for the masses to have super computers in their pockets has led to the rapid adoption of mobile apps at the fingertips of every user—not just those in cubicles all day long. You can't sit in public transit, walk down a street, or even enjoy a nice meal without looking around and seeing the glow from a screen of some sort on someone's face. Everyone is a part of the app ecosystem now. Whether it is a mobile phone, music device, e-reader, watch, or even glasses, apps are a part of our lives. People desire them to make their lives and jobs more productive or just to have fun. As a software developer, it is hard to ignore this surge in opportunity and the desire to capitalize on this ecosystem.

Microsoft technologies present a large opportunity to software developers to reach a vast ecosystem of traditional users who have used Windows technologies in their personal, educational, and professional lives. These users seek out new ways to accomplish tasks and have fun on their technology devices. Microsoft has computing devices across the various screens presented in our lives in our hands, on our desks, and in our living rooms. All these represent opportunities for you, the developer, to extend your reach and ideas into the world.

As this evolution of mobility, multiple screens, and wearables has increased, so has technology. Microsoft technologies have evolved as well

on the client app areas. Over time Microsoft has delivered various ways to write client applications through standard C++, MFC, Windows Forms, Windows Presentation Foundation (WPF), Silverlight, and HTML. Putting developers on a better path for development, Microsoft introduced the Windows Runtime (WinRT). This technology and principles enable developers to have a single platform to target that extends their potential across the personal, professional, and entertainment endpoints we have in our lives. WinRT enables developers to choose how they can be most productive using their skills in C++, C#, Visual Basic, or JavaScript. Alongside the language of choice, developers have a native UI framework in XAML they can use for the best client app experience on Windows. XAML is everywhere now in Windows, from system shell UI to system apps to key experiences delivered from Microsoft, such as Microsoft Office. When developing an app in C# and XAML, you'll be joining other successful developers in the world and can tap into that ecosystem of knowledge, experience, and examples.

Software is an art. Just like any art project, approaching software development requires thought into the necessary tools, philosophies, and principles you will use to create your app. I still remember one of my earliest "professional" software development jobs, sitting in a meeting listening to the customer describe all these (what was at the time) high-tech requirements of their app, all needing to be done in Internet Explorer 3. I scribbled notes as fast as I could while my dev lead at the time, all too quickly I thought, was busy nodding his head in acceptance of the requirements. As we walked out of the meeting, I expressed my concern about the requirements and available technology at the time. He smiled and shrugged like it was no problem stating, "No worries Tim, we just need the right tools."

One of the key tools is a good guide and mentor. In my early days, for me that was books just like this one you have now. To this day I still prefer books on my shelf when learning new technology concepts. I've had the pleasure of working with Jeremy Likeness over the years in the XAML ecosystem, and I can attest to his expertise in building real-world apps using these technologies. In *Programming the Windows Runtime by Example*, Jeremy and John provide these key tools for any software developer to understand the fundamentals of the Windows Runtime and XAML, and be

successful quickly. This book doesn't try to only focus on singular concepts but also provides an end-to-end perspective on building an app in WinRT. Jeremy and John know that your scenarios are connected ones and deal with web services, data, security, and integration. The book will walk you through understanding how the pieces fit together in WinRT while still providing you the knowledge and tools to be productive at the core concepts of working with C# and XAML in the Windows Runtime. John and Jeremy describe philosophies and different approaches to using WinRT, empowering you with knowledge to make the best decisions for your app. This knowledge will enable you to write the best apps for Windows, Windows Phone, Xbox, and whatever future Microsoft has in store for WinRT areas.

Like any artist, tools are essential. This book is one of those essential tools for Windows developers and will help you complete your software goals sooner than without it! To this day, my bookshelf is filled with books just like this one that I refer to often. Even as your experience grows, you'll find yourself referring back to this book for knowledge when developing, just like I did.

—**Tim Heuer**, Principal Program Manager Lead, XAML Platform, Microsoft Corporation

Preface

In 2011 I heard the first rumors about Windows 8 and knew immediately what my next book would be about. Unlike *Designing Silverlight Business Applications* that captured years of experience writing Line of Business (LOB) apps in Silverlight, this book would be an introduction to an entirely new platform. My goal was to take what I knew and loved about Silverlight, find its similarities in the new platform, and then highlight what I felt were some amazing developer experiences. It was important to get to market fast, so through several iterations of the Windows 8 releases (including changes to terminology) that required substantial rewrites of content and a rapid release cycle, I managed to release *Building Windows 8 Apps with C# and XAML* as Windows 8 was revealed to the world.

By necessity, this book introduced developers to the new platform but didn't dig into best practices (there were none yet) or get very deep (there simply wasn't time). I vowed to release another book that would fill in the missing pieces and provide a comprehensive overview of the entire Windows Runtime. Because anyone can read the documentation and reference the API, my intent with this book was to make it example-driven and provide thousands of lines of code for you to integrate and use to kick-start your own Windows Store apps.

I was relieved at the thought of not rewriting most of the book three times, as I had to do with the first one, but Microsoft once again proved too fast for me. What sounded at first like a relatively minor release (Windows

8.1) managed to integrate enough changes to warrant revisiting every one of the ten chapters I had completed to date. With an eye on //BUILD in 2014, I reached out to Windows Store expert and Wintellect colleague John Garland to help me finish the remaining chapters. John and I have worked on several projects together (and incidentally two of them won awards for their groundbreaking use of XAML for touch and mobile), and he helped write pilot code for several of our customers who were early Windows 8 adopters, so I knew he was the right person to bring a fresh set of example projects and content-rich chapters. As a bonus, he is also well-versed in cloud technology and brought this firsthand knowledge to bear in the chapters that deal with connecting to Azure.

In Windows 8.1 and the Windows Runtime, Microsoft has successfully demonstrated their commitment to the development ecosystem by providing us with a rich, vast array of APIs, SDKs, and tools for building incredible apps that run on a variety of devices. I was absolutely amazed when I discovered how easy it was to connect to a web cam, open a web socket, download files in the background, or profile my app to find "hot spots" that I could target to improve performance using WinRT. I was delighted to find that Portable Class Libraries (PCL), something I evangelized heavy as a solution to target multiple platforms in the Silverlight and WPF days, was evolving to embrace Windows Store apps. The first-class support for mature design patterns like MVVM makes it easier than ever to write stable, reusable code that runs on a variety of target devices.

In *Building Windows 8 Apps with C# and XAML,* I shared my intent to guide you through the process of learning the new territory quickly to begin building amazing new applications using skills you already had with C# and XAML. In this book, it is our goal to take you beyond that initial exposure and help you dive deep into all the various APIs WinRT makes available. Our goal was to hit virtually any scenario possible using the Windows Runtime—not just provide code snippets, but full projects you can use to experiment, learn, and use as a starting point for your own apps. The most rewarding feedback I received from my first book was hearing from authors sharing with me their excitement having their first Windows 8 apps approved for the Store. I hope this book not only helps take those apps to the next level, nor simply inspires your imagination, but

empowers you to implement solutions you only dreamed possible using this incredible new platform. I know I speak for both John and myself when I say we look forward to hearing back from you about what you were able to achieve with Visual Studio, Windows 8.1, and this reference on your desk.

What This Book Is About

The purpose of this book is to explain how to write applications—mainly Windows Store apps—that are based on the Windows Runtime. The intent is to explore every available API, exposing you to possibilities across all areas and diving deep into major areas that are likely common to most apps that will be built. Instead of a traditional reference guide that shares API details and code snippets, this book includes more than 80 sample projects. These projects provide a "by example" approach to learning the various APIs; and the text either walks through how they were built, or breaks apart the code step-by-step to make it easy to understand and use as a template for your own projects.

This book is not an introduction to Windows 8.1. We assume you have some experience working with C# and XAML and are familiar with Windows Store apps. We also assume that you are at least familiar with the concept of design patterns and the notion of decoupled code. Both of these ideas have been core to the success of the applications we've helped build and will be used as foundations for the concepts presented in this book.

Whether you're a Windows 8.1 developer looking to improve an existing app, or an experienced client technologies developer transitioning to the Windows Runtime for the first time, this book will give you the guidance, proven patterns and practices, and example projects you'll need to build functional apps that run well across the myriad Windows 8.1 devices.

This version of the book specifically addresses Windows 8.1 using Visual Studio 2013. At this writing, the Windows 8.1 Update was announced at //BUILD, but fortunately the changes did not impact development as much as use of the OS and deployment options. During the course of this book, several changes have occurred that may not be reflected throughout: Visual Studio 2013 Update 2 was released, the name SkyDrive was changed

to OneDrive, Windows Azure became Microsoft Azure, and Azure Mobile Services are constantly being revised.

Where to Access the Source Code

The source code for this book is open source and will be maintained and updated as needed to match any future revisions that may come out. You can download the code samples from the companion website: winrtexamples.codeplex.com.

How to Use This Book

The aim of this book is to enable you to discover the appropriate APIs to build your Windows Store apps. Each chapter is designed to help you discover what features are available in that area of the framework and how they are applied through example projects. Code examples are provided that demonstrate the features for programming them using C# and XAML. Although different chapters may relate to various parts of a comprehensive project, the individual samples are designed to stand on their own.

Each chapter is similarly structured. The chapters begin with an introduction to a topic and an inventory of the capabilities that topic provides. This is followed by explanations of areas of the framework and runtime and a walkthrough of the target APIs. The code samples are explained in detail, either as a walkthrough "lab" or by analyzing the existing sample, and the topic is summarized to highlight the specific information that is most important for you to consider.

I suggest you start by reading the book from start to finish, regardless of your existing situation. Inexperienced developers will find their understanding grows as they read each chapter and concepts are introduced, reinforced, and tied together. Experienced developers will gain insights into areas they might not have considered or had to deal with in the past, or simply didn't factor into their software lifecycles. Once you've read the book in its entirety, you will then be able to keep it as a reference guide and refer to specific chapters any time you require clarification about a particular topic.

Acknowledgments

Jeremy Likness: Although this is my third book through Pearson and fourth full book I've authored, writing a good book still depends on a solid team. I continue to be grateful for my superhuman Editor, Joan Murray, who has been patient and understanding, encouraging, and continuously provided her support and guidance throughout the process. Once again, Eleanor Bru braved working with me on this very ambitious project and, like Joan, was very patient and understanding while keeping me honest and on target. I can't thank Lori Lyons and the production team (including Krista Hansing and Debbie Williams) enough for taking my rambling and helping turn it into coherent prose.

The content of this book was amazingly enriched by our thorough and passionate technical editors. Thank you, Harry Pierson and Christophe Nasarre, for your incredible attention to detail. If anything was missed, I'll take the blame because Harry and Christophe ran every example, pored over every word, and provided me with volumes of suggestions and feedback that helped shape the book to its present form. It is always a pleasure to work with technical editors who bring strong technical insights to the table and help keep me honest when I want to take a shortcut and leave a thread spinning where it shouldn't.

Many thanks to my boss and friend, Steve Porter, for letting me devote a large chunk of my time to a project that made me disappear for a few hours every day. Thanks to Barbara Keihm for her support and encouragement,

to Todd Fine for always recognizing our hard work and being one of the first to pre-order copies whenever they are available, and Bethany Vananda and Sara Faatz for working tirelessly to help spread the word and share what we're doing.

A special note goes to Dave Baskin, Dave Black, Josh Carroll, Aaron Carta, Phil Denoncourt, Dave Frommer, James Katic, Edward Kim, Wes McCammon, and Dan Sloan. This team worked with me on a major project that has lasted longer than the writing of this book and always understood when I had to turn down dinner or other outings so I could get back to my hotel and write. OK, who am I kidding—sometimes I managed to break away.

My wife and daughter have waited patiently through several books now, so they know the routine. Doreen is always quick to remind me when I need to push away from the dinner table and get back to writing, but Lizzie always noticed when I'd been writing too much and was always ready to have a movie date so I could unwind.

Finally, last but certainly not least, thank you! I appreciate my readers—and of course it is for you this was written—so it is my sincere hope you receive tremendous value from these pages.

John Garland: Like Jeremy, I'd very much like to thank Joan Murray, Eleanor Bru, and Lori Lyons, as well as everyone else at Pearson for their unwavering help and guidance throughout this project. Many thanks go to Harry Pierson and Christophe Nasarre for their invaluable help and insight throughout the technical review process—especially for helping to me find the right mix of code and prose, which invariably was along the lines of less prose and more code.

I'd like to very much thank my friends and colleagues at Wintellect. It is truly a privilege for me to count myself in your company and your passion for your craft is absolutely contagious. Many thanks to Steve Porter and Todd Fine for the continued opportunity, and to Bethany Vananda for all the help in putting my work in the best possible light. Much gratitude is owed to Jeff Richter, Jeff Prosise, and John Robbins for their insights into the writing process and for providing the Wintellect stage that I am fortunate to be able to stand on.

Families often have to take a back seat when these projects are in high gear, and mine was no exception. My wife Karen has been more than understanding and forgiving of many late nights, lost weekends, and grumpy mornings. My daughter Callie continues to be a walking smile that forces me to keep things in perspective, despite our having had to skip a few of our priceless Daddy-Callie days. Now that the book is done and the snow has melted, we can get back to bike rides, games of tag, and swing-pushes in the backyard.

I owe many thanks to the folks on and involved with the Zumo (Azure Mobile Services) team, including Kirill Gavrylyuk, Yavor Georgiev, Merwan Hade, and Heinrich Nielsen, among several others. Your insights into the Mobile Services inner workings, and prompt and helpful replies to my inquiries, have been invaluable both for the content included in this book as well as in my professional endeavors.

Finally, I'd like to thank Jeremy for asking me to come along not only on this ride as his co-author, but also as a technical editor on two of his previous books. The experiences, insights, and most importantly, the friendship, have been both personally and professionally invaluable.

About the Authors

Jeremy Likness is a multi-year Microsoft MVP for XAML technologies. A Principal Consultant for Wintellect with 20 years of experience developing enterprise applications, he has worked with software in multiple verticals ranging from insurance, health and wellness, supply chain management, and mobility. His primary focus for the past decade has been building highly scalable web-based solutions using the Microsoft technology stack with client stacks ranging from WPF, Silverlight, and Windows 8.1 to HTML5 and JavaScript. Jeremy has been building enterprise line of business applications with Silverlight since version 2.0, and he started writing Windows 8 apps when the Consumer Preview was released in 2011.

Prior to Wintellect, Jeremy was Director of Information Technology and served as development manager and architect for AirWatch, where he helped the company grow and solidify its position as one of the leading wireless technology solution providers in the United States prior to their acquisition by VMware. A fluent Spanish speaker, Jeremy served as Director of Information Technology for HolaDoctor (formerly Dr. Tango), where he architected a multilingual content management system for the company's Hispanic-focused online diet program. Jeremy accepted his role there after serving as Development Manager for Manhattan Associates, an Atlanta-based software company that provides supply chain management solutions.

John Garland is a Principal Consultant for Wintellect with more than 15 years of experience developing software solutions. Prior to consulting, he spent much of his career working on high-performance video and statistical analysis tools for premier sports teams, with an emphasis on the NFL, the NBA, and Division 1 NCAA football and basketball. His consulting clients range from small businesses to Fortune-500 companies, and his work has been featured at Microsoft conference keynotes and sessions.

John is a Microsoft Client Development MVP, as well as a member of the Windows Azure Insiders and Windows Azure Mobile Services Advisory Board. He lives in New Hampshire with his wife and daughter, where he is an active speaker and participant in the New England software development community. He is a graduate of the University of Florida with a Bachelor's degree in Computer Engineering and holds Microsoft Certifications spanning Windows, Silverlight, Windows Phone, and Windows Azure. John is the author of the ebook *Windows Store Apps Succinctly* (Syncfusion, 2013).

1
The New Windows Runtime

The Windows Runtime (WinRT) provides developers with an object-oriented, language-independent application programming interface (API) for creating applications that run on the Windows 8.1, Windows RT, and Windows Server 2012 and later operating systems. It is based on existing technologies such as the .NET Framework and the decades-old Common Object Model (COM) specification. Microsoft used the best parts of these existing technologies to create something better. Instead of using the .NET Framework to execute code, WinRT is an unmanaged object-oriented runtime that supports multiple development languages. Managed developers can use WinRT from the .NET Framework, thanks to an updated version of the Common Language Runtime (CLR) that interoperates seamlessly with the WinRT APIs.

Windows Runtime Specifics

The Windows Runtime runs on three fundamental architectures: the Intel-based x86 (32-bit) and x64 (64-bit), and the 32-bit ARM. The Intel-based architecture is the most common, and the majority of modern Windows laptops and desktops are based on it. These devices are capable of running Windows 8.1, the latest version of the Windows operating system. Windows 8.1 can run programs from previous versions of Windows (including Windows 7, Windows Vista, and Windows XP).

The ARM-based chip is designed to allow fewer transistors in the microprocessor, resulting in lower power usage and longer battery life on smaller devices that generate less heat. This has made it popular for use in smaller devices such as tablets and smartphones. To address the growing popularity of this chip, Microsoft created the Windows RT operating system. This is a version of Windows 8.1 that targets the ARM architecture but will not run programs built for previous versions of Windows (although Microsoft has ported some popular software, including Microsoft Office, to the platform). It is available only preinstalled on new PCs or tablets. Although the ARM architecture allows for incredibly thin form factors that produce little to no heat, Intel has begun manufacturing a low-power version of its x86 chips, codenamed Atom, that can run Windows 8.1 and retain backward compatibility.

> **TIP**
>
> The Windows Runtime terminology is often a source of confusion. The Windows Runtime is abbreviated as WinRT and refers to the underlying runtime that powers Windows Store apps and desktop applications. Windows RT is the name of the Windows 8.1 operating system version designed specifically for ARM chipsets. Windows RT is a target platform, whereas Windows Runtime (WinRT) is the framework this book discusses that runs on all the target platforms.

Why should you care about WinRT? Unlike the .NET Framework, which runs on top of an underlying operating system, WinRT is part of the native operating system itself. It exposes a set of native APIs that not only provides direct access to components of the operating system, but also evolves and is built as part of the operating system. This gives you access to native APIs that developers working on both native and managed platforms can immediately consume.

WinRT is available only to programs written for Windows 8.1, Windows RT, and Windows Server 2012. You must have a Windows 8.1 or later machine to develop applications that use WinRT, and those applications

must target Windows 8.1 or Windows RT machines. You can develop two types of applications that use WinRT: desktop applications and Windows Store apps. Desktop applications can target only Windows 8.1 machines and are not supported by Windows RT devices. Windows Store apps are supported by both Windows 8.1 and Windows RT. Table 1.1 summarizes these restrictions.

TABLE 1.1 Windows Runtime Support

Application Type	Windows 8.1 Targets	Windows RT Targets
Windows Store app	x86 (32-bit), x64 (64-bit)	ARM (32-bit)
Desktop application	x86 (32-bit), x64 (64-bit)	None

A powerful benefit of developing WinRT applications is that they can be written in a variety of languages because of the object-oriented and language-independent nature of WinRT. The same APIs are exposed to every supported language and are supported as native constructs of those languages. In addition to C# (the focus of this book), the runtime currently supports VB.Net, C++ (a special version of C++ that includes component extensions to simplify writing applications that interact with the WinRT), and JavaScript. JavaScript relies on HTML markup and a browser host to render content, but the other language options all rely on Extensible Application Markup Language (XAML) for the user interface (UI). It is also possible to use DirectX (including Direct2D and Direct3D) in addition to XAML. Table 1.2 provides a brief overview of the various language options and their implementation. You also can create custom WinRT components to add APIs beyond what the operating system exposes. You cannot create custom WinRT components using the JavaScript language option, and custom components are not available to desktop applications.

TABLE 1.2 WinRT Language Options

Language	Runtime	Rendering Engine	WinRT Components
C++	N/A (native code)–requires specific builds for x86, x64, and ARM	XAML, DirectX	Yes
C#	CLR	XAML, DirectX	Yes
JavaScript	Chakra (JavaScript runtime)	Trident (HTML browser)	No
VB.NET	CLR	XAML, DirectX	Yes

Custom WinRT components can be used only from Windows Store apps. Every Windows Store app is actually a WinRT component itself.

Windows Store Apps

Windows Store apps are unique to Windows 8.1, Windows RT, and Windows Server 2012. They run in a single full-screen window without Chrome or in a smaller snapped view side by side with another Windows Store app. They are specifically designed to support different layouts and views (such as portrait and landscape orientations) out of the box and provide first-class support for various forms of input, including stylus and touch. Windows Store apps enable you to engage your users through unique features such as tiles, contracts, and cloud services.

Creating a Windows Store app is simple. Visual Studio 2013 provides several templates for the task. The templates address various forms of navigation, including hierarchical and flat navigation, as well as a blank template to start your app from scratch. The templates also have built-in features such as commonly used value converters and classes that facilitate use of the Model-View-ViewModel (MVVM) pattern.

The MVVM pattern appears throughout the examples in this book. It is a special pattern that takes advantage of the data-binding features of Windows Store apps. The *model* refers to the data and functionality of the app, and the *view* references the user interface as commonly described by XAML in Windows Store apps. The *viewmodel* is a special class that maintains state and encapsulates presentation logic. You learn more about the parts of the MVVM pattern as the book breaks down the sample app.

Example: Create a Windows Store App

To supplement this book, you can download dozens of example projects that illustrate the Windows Runtime. The solution is available online at http://winrtexamples.codeplex.com/, organized by chapter. The first chapter contains a simple "Hello, World" application (the project is called **HelloWorldGridApp**) that demonstrates one of the default templates for creating a Windows Store app. Figure 1.1 illustrates the **Add New Project** dialog box for a Windows Store app. The template is under the **Visual C#** menu when you select **Windows Store**. This example uses the **Grid App** template.

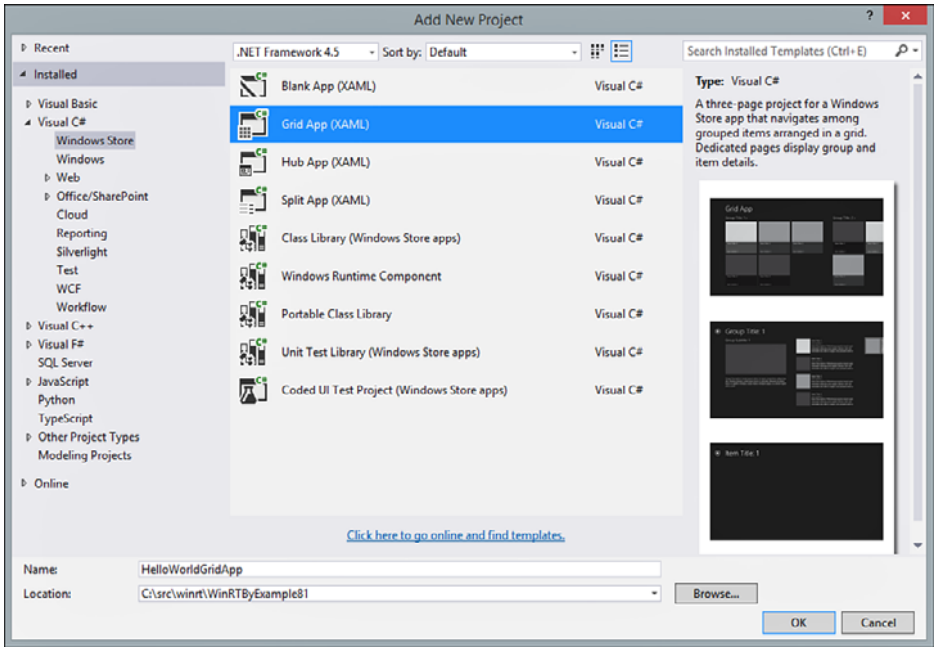


FIGURE 1.1 The Grid App template for a new Windows Store app

The resulting application is fully functional and contains groups of data with subsets of items. The navigation provides the capability to inspect data at the group level, drill into a specific group, and view the details of a specific item. A navigation engine that can persist state is built into the template, along with a base Model-View-ViewModel (MVVM) implementation. The rest of this book makes extensive use of this pattern (Chapter 9, "Model-View-ViewModel [MVVM]," covers it in more detail). The template itself provides several parts you can use to build your own app. You learn more about how to use these parts in Chapter 2, "Windows Store Apps and WinRT Components," which covers the templates in more detail. This is a high-level overview of the key components the template provides.

TIP

If you are creating your own projects, you might notice that they do not exactly match the sample code provided with the book. This is because the sample projects use a technique to share common code and assets. The Common solution folder contains assembly settings that all projects share. To use the `CommonAssemblyInfo.cs` attributes in one of your projects, the file should be added using **Add as Link** from the **Add Item** dialog box (instead of using the default **Add** option). This applies to assets for the Windows Store apps as well. This technique enables you to change common items in one place without having to update each individual project, and it is a good practice to follow for your own applications.

The main application is launched through the `App` class with code in the `App.xaml.cs` code-behind file. Think of that as your "main program" loop. The `OnLaunched` method is called when the app is launched (this can happen in several ways, ranging from tapping on a tile to invoking search). A `Frame` hosts the navigation of the application. The `SuspensionManager` class saves and restores state when the app is paused and resumed. As a default, the app navigates to the `GroupedItemsPage`, which shows you a list of both groups and items that are available.

```
rootFrame.Navigate(typeof(GroupedItemsPage))
```

The pages of the app include the `GroupItemsPage` (all groups and items), the `GroupDetailPage` (all items for a single group), and the `ItemDetailPage` (individual item). All are designed to help the user navigate through the "model" that is exposed through the `SampleDataSource` class in the `DataModel` folder. This class generates several groups that contain items with images (the default shades of gray provided in the `Assets` folder).

Open `GroupDetailPage.xaml.cs` and look at the code-behind. The control is based on the `Page` class, which a `Frame` can use for navigation. The page uses the `DefaultViewModel` viewmodel to synchronize state (notice how it hosts the current group and the items for the group), handle presentation logic, and act as an arbiter between the rest of the application and the XAML view.

The `NavigationHelper` class provides a `LoadState` method. This method takes either a parameter passed through navigation or a parameter that was saved when the app was suspended and then uses that to construct the view. This provides consistent behavior, whether navigating to the page or returning it when you resume the app. The parameter references the group passed (or saved) for the page; then the group and its corresponding list of items are assigned to the viewmodel. You learn more about this helper in Chapter 2.

```
var group = await SampleDataSource
    .GetGroup((String)navigationParameter);
this.DefaultViewModel["Group"] = group;
this.DefaultViewModel["Items"] = group.Items;
```

When an individual item is selected, the identifier is passed to the `Frame` to navigate to the detail page.

```
var itemId = ((SampleDataItem)e.ClickedItem).UniqueId;
this.Frame.Navigate(typeof(ItemDetailPage), itemId);
```

The bulk of the presentation logic is encapsulated in the underlying viewmodel. For example, the `OnNavigatedTo` method of the page is overridden to preserve the navigation request in a "back stack" that can be used to allow backward and forward navigation in the app (this is referred to as the journal). It preserves the navigation parameter and then calls the method on the derived class to `LoadState`, as shown earlier. It also uses the

`SuspensionManager` to save the state to disk when the user navigates away from the page. This preserves the history and restores it when the user returns to the app after it has been suspended.

The XAML view itself also handles layout changes (such as changing the orientation of the page or snapping it). Figure 1.2 shows a general overview of how the template works and its relation to the MVVM pattern.

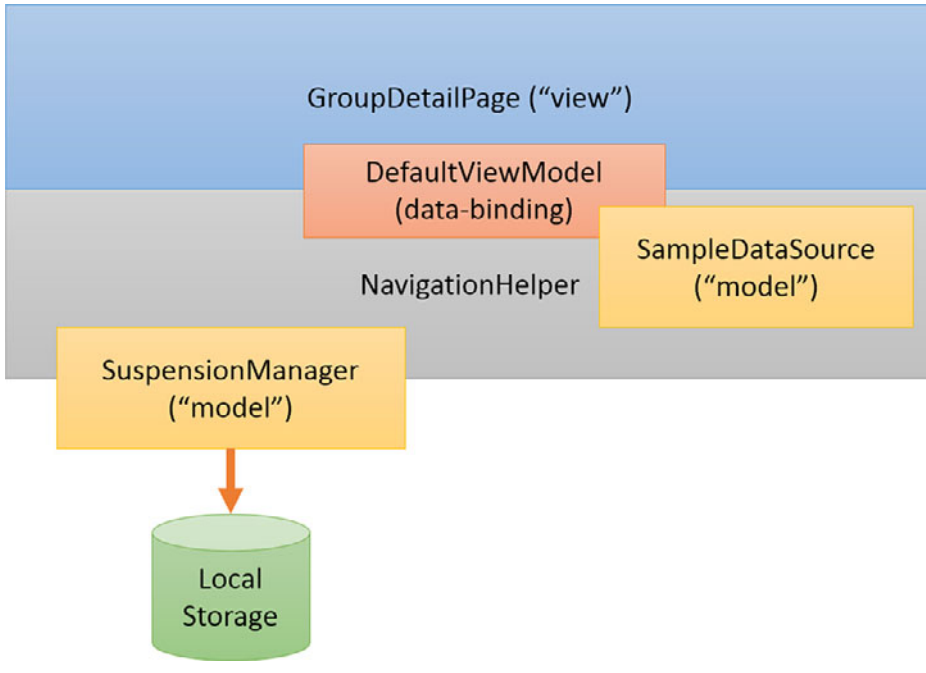


FIGURE 1.2 The sample template and the MVVM pattern

The built-in templates have many parts that you won't always use. The next several chapters provide more details about how the view, viewmodel, and application model work together to provide you with the tools you need to build apps using WinRT. Although WinRT provides its own set of APIs, understanding how WinRT interoperates with the .NET Framework is important.

.NET and WinRT

You might have noticed that the sample app uses C# and references .NET types. When you program Windows Store apps using C#, you interact with both the Windows Runtime and a special .NET Framework 4.5 profile. The profile enables you to create both Windows Runtime components with managed code (only Windows Store apps can consume these components) and Windows Store apps. This profile provides a set of reference assemblies that expose only the relevant types and members allowed to be consumed by Windows Store apps. It removes features from the .NET Framework API that exist in the Windows Runtime and provides direct references to the full set of WinRT APIs available to Windows Store apps.

The .NET Framework 4.5 for Windows Store apps is selected for you when you indicate that you want to build a Windows Runtime component or a Windows Store app using C#. It is important to understand how it is implemented because it uses a combination of type mapping, language projection, compiler-specific features, and CLR enhancements to create the interoperability between the .NET Framework and the Windows Runtime. Although most of the functionality is handled "behind the scenes," you should understand some nuances to avoid side effects or unwanted errors in your applications. For example, although you might reference .NET types in your code, some magic is happening behind the scenes.

Fundamental Types

Fundamental types from the Windows Runtime appear in Visual Studio and source code as their .NET equivalents. For example, the metadata information for a 4-byte integer in WinRT is coded as `ELEMENT_TYPE_I4`. When you are developing in the .NET Framework, it appears as the familiar `System.Int32`. This is done automatically, and you have no way to reference an underlying type directly from your source code. When you use an alias for a common type such as `int`, Visual Studio automatically maps that as `System.Int32`, and the compiler automatically emits the `ELEMENT_TYPE_I4` metadata.

Note that no support for null strings exists in the Windows Runtime. If you attempt to pass a null string to a WinRT component, you receive an `ArgumentNullException`. An empty string should be represented using `String.Empty`.

Mapped Types

A number of types appear as one type in the Visual Studio but are generated as another type by the compiler. Table 1.3 lists these types and their maps.

TABLE 1.3 Mapped Types Between the .NET Framework and WinRT

.NET Type	WinRT Type	Notes
`System: AttributeUsage`	`Windows.Foundation.Metadata: AttributeUsage`	
`System: AttributeTargets`	`Windows.Foundation.Metadata: AttributeTargets`	
`System: DateTimeOffset`	`Windows.Foundation: DateTime`	1
`System: EventHandler<T>`	`Windows.Foundation: EventHandler<T>`	2
`System.Runtime. InteropServices. WindowsRuntime: EventRegistrationToken`	`Windows.Foundation: EventRegistrationToken`	2
`System: Exception`	`Windows.Foundation: HResult`	3
`System: Nullable<T>`	`Windows.Foundation: IReference<T>`	4
`System: TimeSpan`	`Windows.Foundation: TimeSpan`	
`System: Uri`	`Windows.Foundation: Uri`	5
`System: IDisposable`	`Windows.Foundation: IClosable`	6

.NET Type	WinRT Type	Notes
System.Collections.Generic:IEnumerable<T>	Windows.Foundation.Collections:IIterable<T>	
System.Collections.Generic:IList<T>	Windows.Foundation.Collections:IVector<T>	
System.Collections.Generic:IReadOnlyList<T>	Windows.Foundation.Collections:IVectorView<T>	
System.Collections.Generic:IDictionary<K,V>	Windows.Foundation.Collections:IMap<K,V>	
System.Collections.Generic:IReadOnlyDictionary<K,V>	Windows.Foundation.Collections:IMapView<K,V>	
System.Collections.Generic:KeyValuePair<K,V>	Windows.Foundation.Collections:IKeyValuePair<K,V>	
System.Collections:IEnumerable	Windows.UI.Xaml.Interop:IBindableIterable	
System.Collections:IList	Windows.UI.Xaml.Interop:IBindableVector	
System.Collections.Specialized:INotifyCollectionChanged	Windows.UI.Xaml.Interop:INotifyCollectionChanged	
System.Collections.Specialized:NotifyCollectionChangedEventHandler	Windows.UI.Xaml.Interop:NotifyCollectionChangedEventHandler	
System.Collections.Specialized:NotifyCollectionChangedEventArgs	Windows.UI.Xaml.Interop:NotifyCollectionChangedEventArgs	

.NET Type	WinRT Type	Notes
System.Collections.Specialized: NotifyCollectionChanged Action	Windows.UI.Xaml.Interop: NotifyCollectionChanged Action	
System.ComponentModel: INotifyPropertyChanged	Windows.UI.Xaml.Interop: INotifyPropertyChanged	
System.ComponentModel: PropertyChangedEvent Handler	Windows.UI.Xaml.Interop: PropertyChangedEventHandler	
System.ComponentModel: PropertyChangedEventArgs	Windows.UI.Xaml.Interop: PropertyChangedEventArgs	
System: Type	Windows.UI.Xaml.Interop: TypeName	7

The following list summarizes the notes from Table 1.3:

1. Dates are always converted to Coordinated Universal Time (UTC). The time zone is not stored in WinRT, so when the dates are converted back to the .NET Framework equivalent, the local time zone is assumed. If you are using time zones other than the local time zone, you need to apply further conversions when the date is passed back.
2. WinRT event handlers generate a unique token when a subscriber is added. The subscriber uses this token to unsubscribe. In the .NET Framework, you do not have to use these tokens; you can simply add or remove delegates in the standard way using the += and -= operator overloads. The compiler generates the necessary code to call a helper function to maintain a map between the delegates you register and their corresponding tokens.
3. WinRT exceptions are really just a 32-bit integer that, in COM, is known as an HRESULT.[1] The CLR maps well-known exceptions to

[1]HRESULT, http://bit.ly/W8WBxF

the COM equivalent; conversely, an exception returned from WinRT can be converted to a well-known .NET `Exception` type. When there is no well-known mapping, the HRESULT is converted to a generic `Exception` object, with the value of the HRESULT stored in the `HResult` property.

4. WinRT uses `IReference<T>` for two scenarios: boxing values and providing support for `Nullable<T>`. You can read more about how this is implemented in the previous section.
5. The Windows Runtime does not support relative URIs. If you try to pass a relative URI to a WinRT component, an `ArgumentException` is thrown. If you must use a relative URI, the workaround is to provide a base URI to create an absolute URI and then ignore the base portion when the URI is returned.
6. The Windows Runtime requires all I/O operations to be asynchronous. The mapped `Close` method on the WinRT `IClose` interface maps to the `Dispose` method on the .NET Framework `IDisposable` interface. This interface is not marked as asynchronous. Therefore, you should not perform any I/O in your `Dispose` method. You must implement your components so that they flush or close streams before they are disposed.
7. Types in WinRT are far more simplified than in the .NET implementation. A WinRT type is simply the fully qualified name for the type, and the metadata system is used to parse any relevant information about that type.

Note that the restrictions for mapping apply only to calls between the CLR and WinRT. You can still build C# class libraries to use within your projects without any restrictions. In fact, it is recommended that you use C# class libraries instead of WinRT components for reusable code in your applications if you are not planning to consume those components using any other languages. Create a managed WinRT component only when you want to access it from nonmanaged language options such as C++ and JavaScript.

Streams and Buffers

The Windows Runtime contains several interfaces for streams that existing classes in the .NET Framework do not directly support. To close this gap, you can use one of several extension methods provided by the `System.IO.WindowsRuntimeStorageExtensions` class and defined in `System.Runtime.WindowsRuntime.dll`. They effectively convert streams between WinRT and the CLR by performing quite a bit of work, such as ensuring that each stream has its own unique adapter and buffer to provide the best performance possible and ensuring that the stream behaves as expected within your .NET code.

WinRT provides the `IStorageFile` interface to work with local and roaming storage. You can use the two extension methods `OpenStreamForReadAsync` and `OpenStreamForWriteAsync` to generate, from an `IStorageFile` instance, a `Stream` object to pass to your .NET components. Several overloads exist to work with directories and relative paths and specify various options when creating a new file. You learn more about storage in Chapter 4, "Data and Content."

WinRT stream interfaces include `IRandomAccessStream`, `IInputStream`, and `IOutputStream`. The `System.Runtime.WindowsRuntime` namespace provides several extension methods to convert between these stream types and .NET streams: `AsStream`, `AsStreamForRead`, `AsStreamForWrite` to convert from WinRT to .NET Framework streams; and `AsInputStream` and `AsOutputStream` to pass .NET Framework streams to WinRT APIs.

Sometimes you need to pass blocks of data to WinRT APIs. You cannot pass a byte array or memory stream directly to the Windows Runtime, so WinRT provides the `IBuffer` interface. The interface declares only two properties: the total capacity for the buffer and the number of bytes in use. You cannot access, read from, or write to the actual underlying buffer. This makes the interface a safe way to pass buffers among various language implementations. Internally, WinRT can access the buffer with a COM interface that can access its actual contents.

Several extension methods exist in the `System.Runtime.InteropServices.WindowsRuntime.WindowsRuntimeBufferExtensions`[2] class. You can convert a byte array into an `IBuffer` instance by using the `AsBuffer` extension methods. You can also convert a `MemoryStream` instance into an `IBuffer` instance by using the `GetWindowsRuntimeBuffer` extension methods.

Conversely, you can convert an `IBuffer` instance to either a byte array or a `Stream` instance. The same helper class provides the `AsStream` and `ToArray` extension methods to perform this conversion for you. The class contains other methods to parse, compare, and copy data to and from buffers.

Desktop Applications

The Windows Runtime is also available from the desktop-style applications you are likely familiar with, although this is fairly rare. Most of this book focuses on Windows Store apps, but it is important to know that various APIs in WinRT are available to traditional desktop applications.

You can write desktop applications that use the Windows Runtime in C++, C++/CX,[3] or C#. To use C#, you must target the .NET Framework 4.5. The capability to reference WinRT APIs is not provided out of the box. Instead, you must modify your project to change the references manually.

Example: Reference WinRT from a Desktop Application

The sample project **DesktopWinRT** in the **Chapter 1** solution folder demonstrates a desktop application that calls WinRT APIs. To create a similar project yourself, choose the option to add a new project from within Visual Studio 2013. Navigate to **Visual C#** and **Windows**, and then choose the template for a **WPF Application** as in Figure 1.3.

---

[2]WindowsRuntimeBufferExtensions, http://bit.ly/XOBVtL

[3]C++/CX Reference, http://bit.ly/XLP5Yd

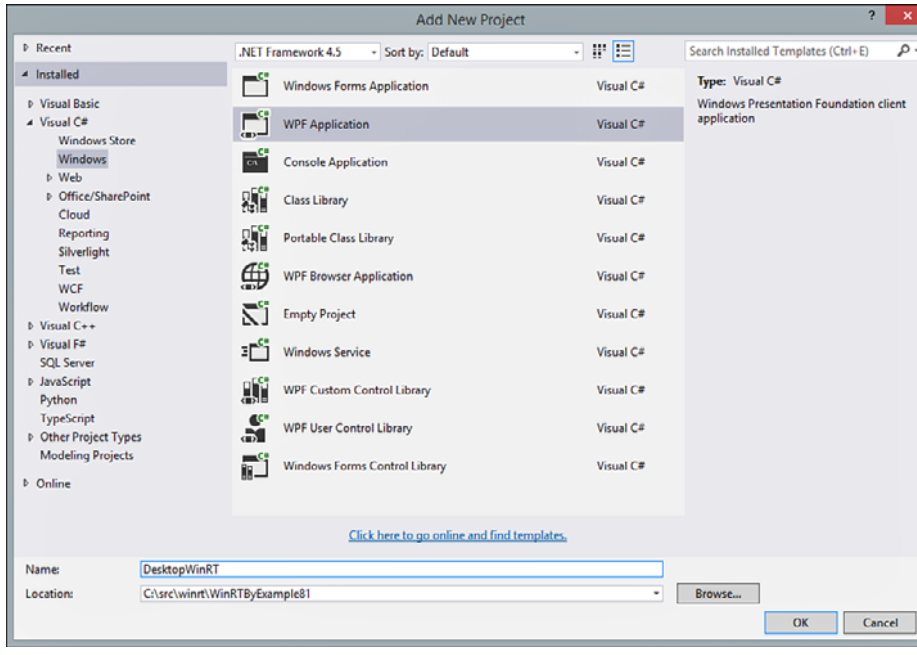


FIGURE 1.3 Creating a desktop application

Make sure that the project targets the .NET Framework 4.5 or later. Although a number of default assembly references are added, accessing WinRT APIs is not yet possible. Doing this involves a few steps. The first step is required because it is not possible to update a project to access WinRT from the IDE. Instead, you must edit the project file directly. First, right-click the project in the **Solution Explorer** and choose the option **Unload Project**. This project should show as "unavailable" in the Solution Explorer (see Figure 1.4).

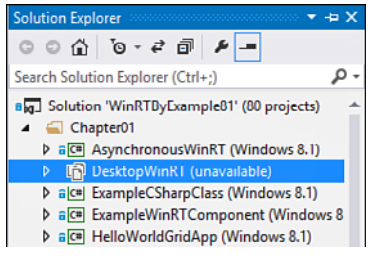


FIGURE 1.4 Unloaded project

When the project is unloaded, you can edit the XML contained in the project file by right-clicking the project node and selecting **Edit**. If you receive a message about inconsistent line endings, feel free to accept the dialog box to clean the document. To allow you to reference WinRT, the project must target Windows version 8.1 or later. This is done by adding the following snippet inside the first `PropertyGroup` element:

```
<TargetPlatformVersion>8.1</TargetPlatformVersion>
```

Note that IntelliSense might suggest `TargetFrameworkVersion`, but that is not the correct tag to update. After you've made the change, save the file, right-click the project node, and select **Reload Project**. If you receive a prompt about the project file already being open, select the option to close it. After the project has reloaded, you are ready to add a WinRT reference.

Right-click the **References** node in the **Solution Explorer** and select **Add Reference**. You should notice a new tab with the text **Windows** and a child tab named **Core**. Select this tab and check the box next to the **Windows** assembly, as in Figure 1.5.

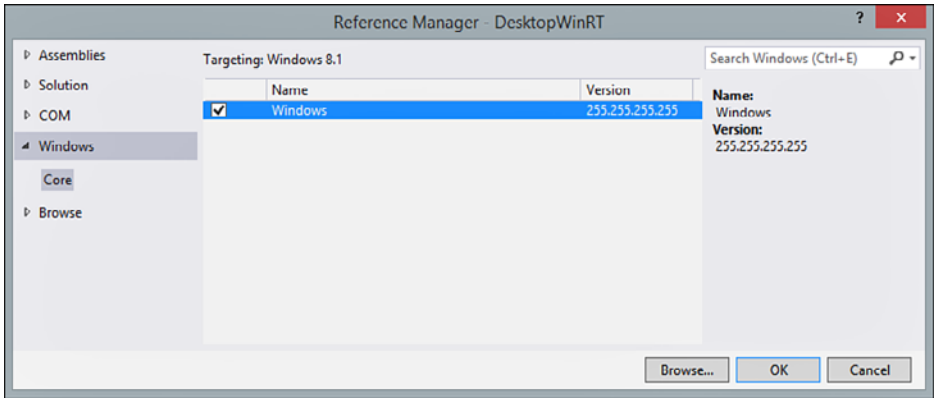


FIGURE 1.5 Adding the WinRT reference

After you click **OK**, you're ready to reference the interfaces. For the example program, the following `using` statement was added to **MainWindow.xaml.cs**:

```
using Windows.Management.Deployment;
```

This statement provides access to a set of WinRT APIs for enumerating packages (installed apps) in the system. The following line inside the event handler for the main window load event references a WinRT component:

```
var packageManager = new PackageManager();
```

A query on the Package Manager returns a list of packages for the user that are added to a list in the WPF control. Listing 1.1 shows the full code for this.

LISTING 1.1 Package Manager List

```
var list = new List<string>();
var packageManager = new PackageManager();
var identity = WindowsIdentity.GetCurrent();
if (identity == null || identity.User == null)
{
    MessageBox.Show(
        "Unable to determine the current user's identity.");
    return;
}
var query = packageManager.FindPackagesForUser(identity.User.Value);
foreach (var package in query)
{
    var name = package.Id.Name;

    try
    {
        list.Add(string.Format("Package {0} at {1}", name,
            package.InstalledLocation.Path));
    }
    catch (FileNotFoundException)
    {
        list.Add(string.Format("Package {0} deleted.", name));
    }
}

Packages.ItemsSource = list;
```

Unfortunately, adding the references and compiling the program isn't enough, and the compiler generates several errors (see Figure 1.6). This is because additional assemblies are needed for the WPF example.

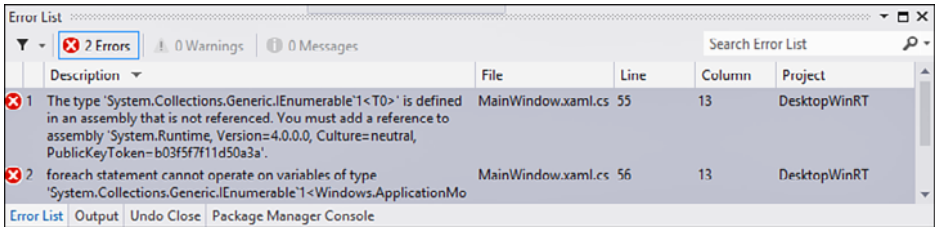


FIGURE 1.6 Compiler errors when attempting to reference WinRT

The dependent assemblies exist in a special path on the system, and references to them are required to successfully call WinRT APIs from the WPF desktop application. To fix this, you add another reference to the project and browse to the following folder:

```
C:\Program Files (x86)\Reference Assemblies\Microsoft\Framework\
➥.NETFramework\v4.5\Facades
```

Select `System.Runtime.dll` and click **Add** to include it within your project references. You should then be able to compile your program. In the reference example, you should see a window with a list of packages on the system, similar to Figure 1.7.

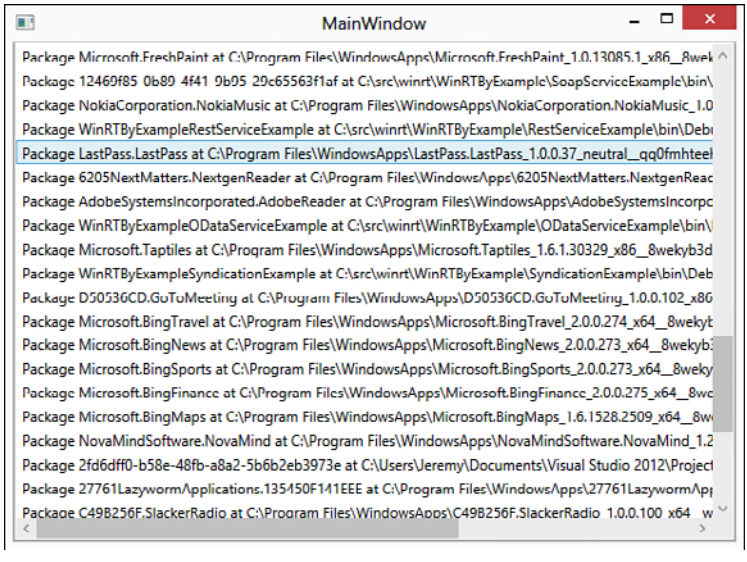


FIGURE 1.7 Package listing from a WinRT API called from a WPF application

Referencing WinRT from a desktop application is not straightforward, but it is possible. Remember, three steps are required:

1. Modify the project file to target the platform version 8.1.
2. Add the Windows Core reference.
3. Browse and add the `System.Runtime.dll` reference.

The MSDN library provides a list of WinRT APIs that you can call from Windows 8.1 desktop applications.[4] The MSDN documentation for each WinRT component includes a section that provides information about the requirements to use a particular API. Figure 1.8 shows an example for the `PackageManager` class.

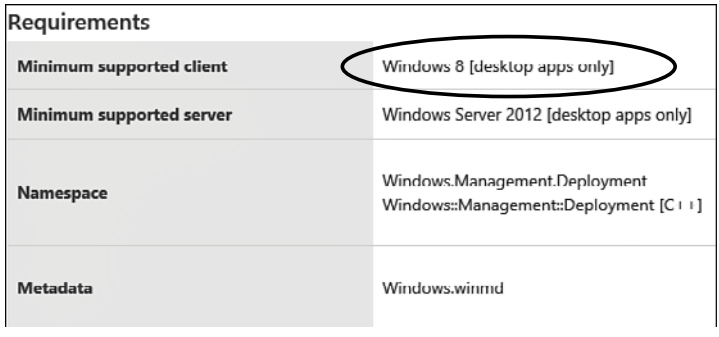


FIGURE 1.8 Determining where and how WinRT APIs can be called

Only a small number of APIs are available for the desktop. Most APIs are for the Windows Store only, and a few are available to both types of apps.

Example: Examine Projections in a WinRT Component

Projections make it possible to reference underlying WinRT types as equivalent .NET types in code, even though the running app will use the

[4]Windows 8.1 API list, http://bit.ly/Slcq53

unmanaged component. The example solution contains a project named **ExampleCSharpClass** that illustrates the differences between C# class libraries and WinRT components and demonstrates how projections work. Figure 1.9 shows the code map for the simple API this class exposes. A method takes in a URI and fires an event that encapsulates the string representation of the URI. This shows the specific differences between how events are handled in the Windows Runtime vs. the .NET Framework and how types are mapped. Keep in mind that you will most likely create simple C# class libraries unless you specifically need to reference your libraries from C++ or JavaScript code.

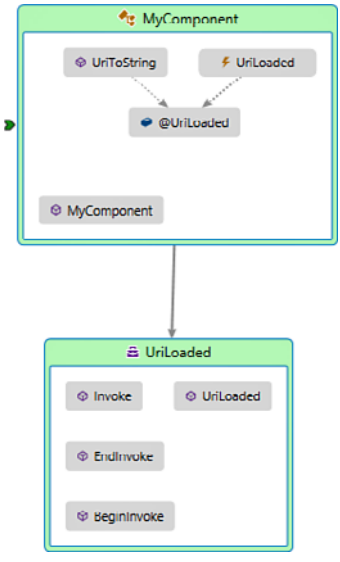


FIGURE 1.9 The code map for the example project

The event arguments simply reference the URI.

```
public sealed class UriLoadedArgs
{
    public string Uri { get; set; }
}
```

The component exposes the signature of the event:

```
public delegate void UriLoaded(object sender,
    UriLoadedArgs args);
```

It also defines the event itself:

```
public event UriLoaded UriLoaded;
```

Compile the C# class library and navigate to the output directory using the Developer Command Prompt link. (If you're not sure how to find the output directory, right-click the project name in the **Solution Explorer** menu and choose **Open Folder in File Explorer** to dig into its **bin\debug** subfolders.) Use the `ildasm.exe` tool to inspect the DLL by typing the following on the command line in the folder that contains the DLL:

```
ildasm ExampleCSharpClass.dll
```

> **TIP**
>
> If you use the `ildasm.exe` tool quite often, it can be useful to associate it with the DLL files you are inspecting. On Windows 8.1 machines, you can right-click a DLL and choose **Open With** for a list of options. The tool will not likely show by default. Instead, choose **More Options** (if that is available), followed by **Look for Another App on This PC**. Be sure to check the box labeled **Use This App for All .dll Files** unless you have another application installed that you prefer to use instead. Simply browse to the folder `C:\Program Files\Microsoft SDKs\Windows\v8.1A\bin\NETFX 4.5.1 Tools` and select `ildasm.exe` to launch directly into the tool from **File Explorer**.

When the component is loaded, you should see three top-level entities defined:

- `MyComponent` is the main class definition.
- `UriLoaded` is the definition of a delegate.
- `UriLoadedArgs` is the class definition for the event arguments.

The signature of the `UriToString` method of `MyComponent` is:

```
UriToString: void(class [System.Runtime]System.Uri)
```

The signature of the event handler methods to subscribe and unsubscribe by passing a delegate is:

```
add_UriLoaded: void(class ExampleCSharpClass.UriLoaded)
remove_UriLoaded: void(class ExampleCSharpClass.UriLoaded)
```

Notice that the methods for the event handlers are empty and take in a single parameter that is the delegate for the event. Now compile the WinRT version of the component by building the **ExampleWinRTComponent** project. The code is identical to the class library but resides in a different namespace. If you browse to the output location, the first thing you'll likely notice is that there is no DLL. Instead, the compiler has produced a WinRT metadata file named **ExampleWinRTComponent.winmd**.

Open this file using `ildasm.exe`. The first thing you'll notice is that the public classes have been renamed with a `<CLR>` prefix and made private. This is how the .NET Framework and WinRT interoperate together. The internal classes are marked private so they are not visible to WinRT, and the projected implementations are made public. In addition, both classes had an interface automatically generated. The main component class implements the `IMyComponentClass` interface. In COM, clients must interoperate with services through interfaces, but the compiler has made this transparent for you by generating the interface if you did not provide one yourself.

Also notice that no "special" class was created for the delegate. The metadata for the base delegate type in WinRT is marked the same way as the CLR type, so no conversion is needed. When you look at the interface `IMyComponentClass`, you'll see that the signature of the `UriToString` method is different:

```
UriToString: void(class [Windows]Windows.Foundation.Uri)
```

The methods to add and remove event handlers have also changed:

```
add_UriLoaded: valuetype [Windows]Windows.Foundation.
   EventRegistrationToken(class ExampleWinRTComponent.UriLoaded)
remove_UriLoaded: void(valuetype [Windows]Windows.Foundation.
   EventRegistrationToken)
```

The URI has been converted to the WinRT type, and the event code has changed to implement the WinRT method of managing registrations with a token. If you want to see how the component is projected back to the CLR, go to the command line and open the file again, but this time add the **/project** command line switch:

```
ildasm /project ExampleWinRTComponent.winmd
```

The same metadata will be parsed, but this time projections are applied so you can see the types as they appear in managed code. For example, the `UriToString` signature on the interface is now this:

```
UriToString: void(class [System.Runtime]System.Uri)
```

You can use `ildasm.exe` to inspect metadata for WinRT types. Using the **/project** switch parses the same metadata but displays the projections and mappings for consumption by managed code.

Asynchronous Functions

In C#, you can expose and consume asynchronous methods using the `async` and `await` keywords. In the .NET Framework, you typically write the implementation of an asynchronous interface using the Task Parallel Library (TPL).[5] The **AsynchronousWinRT** example provides an example of this in the `AddNumbersInternal` method (see Listing 1.2). This is how you typically wrap and return a long-running task.

LISTING 1.2 Using Task for Asynchronous Operations in the .NET Framework

```
private static Task<long> AddNumbersInternal(
    ICollection<int> array)
{
    return Task.Run(
        () =>
            {
```

[5]Task Parallel Library (TPL), http://bit.ly/UcpUhS

```
            long result = 0;
            for (var index = 0; index < array.Count;
                index++)
            {
                result += index;
            }

            return result;
        });
}
```

If you try to make the method public, you will receive an error. This is because WinRT doesn't contain a `Task` type. Instead, it supports four distinct asynchronous operation interfaces. All asynchronous interfaces inherit the `IAsyncInfo` interface, which provides a unique identifier for the operation, a status of the operation, an error code in case the operation aborts, and methods to cancel and close the operation:

- `IAsyncAction` is an asynchronous operation that does not report progress and does not return a result.
- `IAsyncActionWithProgress<TProgress>` is an asynchronous operation that reports progress but does not return a result.
- `IAsyncOperation<TResult>` is an asynchronous operation that does not report progress but returns a specific result when complete.
- `IAsyncOperationWithProgress<TResult, TProgress>` is an asynchronous operation that reports progress and returns a result when complete.

The .NET Framework provides extension methods that enable you to convert `Task` objects to WinRT equivalents or launch asynchronous tasks directly. For example, the following code converts the task-based asynchronous operation from Listing 1.2 into a WinRT asynchronous operation:

```
public IAsyncOperation<long> AddNumbers(
   [ReadOnlyArray] int[] array)
{
    return AddNumbersInternal(array).AsAsyncOperation();
}
```

Notice the use of the `ReadOnlyArray` attribute to flag the array as input-only. This marks the array so that WinRT understands which of the three allowed array-passing methods is being used (passed, filled, or received). The extension method that the framework provides converts the `Task` to an `IAsyncOperation`. To return a task that supports progress, you can use the `AsyncInfo` static class to launch the instance, as in Listing 1.3. This listing uses the TPL to create a long-running task but wraps that in a WinRT asynchronous operation that supports progress reporting.

LISTING 1.3 Creating a WinRT Asynchronous Operation That Reports Progress

```
public IAsyncOperationWithProgress<long, double>
    AddNumbersWithProgress([ReadOnlyArray] int[] array)
{
    return AsyncInfo.Run(
        async (
            CancellationToken cancellationToken,
            IProgress<double> progress) =>
            {
                progress.Report(0);
                return await Task.Run(
                    () =>
                        {
                            long result = 0;
                            for (var index = 0;
                                index < array.Length;
                                index++)
                            {
                                progress.Report(
                                    (double)index /
                                    array.Length);
                                result += index;
                            }

                            return result;
                        });
            });
}
```

Writing asynchronous operations that the Windows Runtime can consume is easy using the extension methods that the framework provides.

Summary

This chapter introduced you to the concept of two types of programs that can interoperate with the Windows Runtime: desktop applications and Windows Store apps. You learned how WinRT uses the best parts of COM and the .NET Framework to allow multiple languages to seamlessly consume components. This projection is coupled with features of the CLR and the compiler that maps WinRT types and classes to their CLR equivalents. Various extension methods make it easy to interoperate with WinRT components and expose managed code to the Windows Runtime by converting between types and generating boilerplate code that handles events, streams, buffers, and asynchronous operations.

In the next chapter, you learn more about Windows Store apps. The Windows Runtime provides a host environment with services for suspending and resuming Windows Store apps, as well as creating interfaces between Windows Store apps, other applications, and the operating system itself. You will also create a Windows Runtime component and learn how to consume it from unmanaged code and JavaScript-based Windows Store apps.

This page intentionally left blank

2

Windows Store Apps and WinRT Components

Windows store apps is a label with a bit of history behind it. The story began in 2010 when Microsoft announced the Windows Phone 7 Series. Microsoft later dropped the *Series* part of the name, but new developers to the platform quickly learned that Windows Phone 7 featured the design philosophy referred to as Metro. Earlier Microsoft projects, including Zune and Windows Media Center, heavily influenced it. Many developers were excited to see the Metro design language appear in the earliest versions of Windows 8.

If you created a Windows Store app in early 2011 when the Developer Preview version of Windows 8 was released, you fired up a copy of the Visual Studio 11 Preview and chose your language and the option for the Windows Metro Style application. By the Consumer Preview, the menu item changed to Metro App for the Metro style framework. Sometime after this, potential trademark infringements led Microsoft to drop the term *Metro* altogether. Documentation began referring to either the "Modern UI" or the "Microsoft design language," and after Visual Studio 11 was renamed to Visual Studio 2012, it provided an option to build a Windows Store app for Windows 8. Eventually, Visual Studio 2013 was released with the capability to build the same apps to target Windows 8.1.

This history might help explain why you can visit the Windows Store to purchase desktop applications (that are not Windows Store apps) or why you can install Windows Store apps without ever visiting the Windows Store. When I refer to a "Windows Store app," think of it as a Windows 8.1 application that is built and deployed as a WinRT component. That should help separate these apps from desktop applications in your mind.

The Windows Runtime provides a special set of services for Windows Store apps. The runtime hosts Windows Store apps as needed, loads the CLR for managed apps, and invokes a web host to render HTML and interpret client script for apps written with JavaScript. Windows Store apps are deployed in special packages that contain unique assets and resources. They have access to local and temporary storage that is sandboxed from the rest of the system and can manipulate roaming settings that automatically synchronize across multiple Windows 8.1 devices. Windows Store apps operate in a special lifecycle designed to maximize battery life and end-user productivity by suspending and resuming applications as they are brought in and out of focus.

In addition to creating special WinRT components that are published as Windows Store apps, you can create standalone WinRT components in C# that other apps written in native C++ or interpreted JavaScript can invoke. Managed components might appear as native WinRT components to their clients but impose special dependencies. An instance of the CLR must exist to host managed code for native code to call it. The fact that managed WinRT components generate Common Intermediate Language (CIL) code means they can be compiled once and run on x86, x64, and ARM systems. Native code must be compiled to target each system it will run on.

Fundamentals of a Windows Store App

Windows Store apps have a unique look and feel, can target multiple devices, and are most often distributed through the Windows Store. The Windows Store is Microsoft's platform for software distribution and supports a variety of targets, including Windows 8.1 and Windows RT. Note that Windows Store apps can be developed only on Windows 8.1 machines; no SDK is available for Windows 7 or other versions (for example, you can

target Windows 8 only on a Windows 8 machine, so it is recommended that you upgrade your environment to 8.1).

Windows Store apps are designed to run either in a single window with no chrome, or arranged horizontally side by side, with other apps taking up the full vertical space available in the display. The intention is to provide maximum focus, avoid distractions, and make the best use of available real estate. A common technique is hiding common commands until they are needed. Temporary panels appear as either modal dialogs or panels that fly out to overlay content until they are dismissed (hence the term *flyout*). The apps are also designed to scale to different form factors, such as portrait and landscape, high resolution and low resolution, and a narrow mode when several apps are run side by side.

Windows Store apps provide first-class support for touch and pen input. All the built-in controls you will learn about have built-in capabilities to recognize touch, taps, and gestures. Although touch is important, apps can also be launched on traditional hardware that uses mouse and keyboard input. You should design your apps to accommodate these modalities and all the built-in controls come out of the box with keyboard and mouse support, in addition to the support for pen and touch.

Windows 8 introduced contracts and extensions. Contracts were considered an agreement between apps that enabled rich content sharing and other features. They allow your app to communicate with other apps you don't know about because the interface is a common contract. Extensions were defined as agreements between Windows Store apps and the operating system that allow apps to participate in file picking operations, camera functions, file associations, and more. These features have evolved in Windows 8.1 and are referred to generically as *declarations.* The capability for an app to use these features is declared in the application's manifest, which you learn more about in this chapter. Chapter 11, "Windows Charms Integration," and Chapter 12, "Additional Windows Integration," cover declarations in more detail.

A variety of Windows Store app templates are available to help you start a new project. In fact, most of the templates generate fully functional and interactive reference applications you can modify to create your own Windows Store apps. Understanding the differences between templates is important to choose the right starting point for your application.

Windows Store App Templates

The Windows Store app templates are accessible from Visual Studio 2013 via the **New Project** menu. Eight templates (plus the "bonus" Portable Class Library template you will learn about) are available for developing Windows Store apps using C#. Figure 2.1 shows the templates.

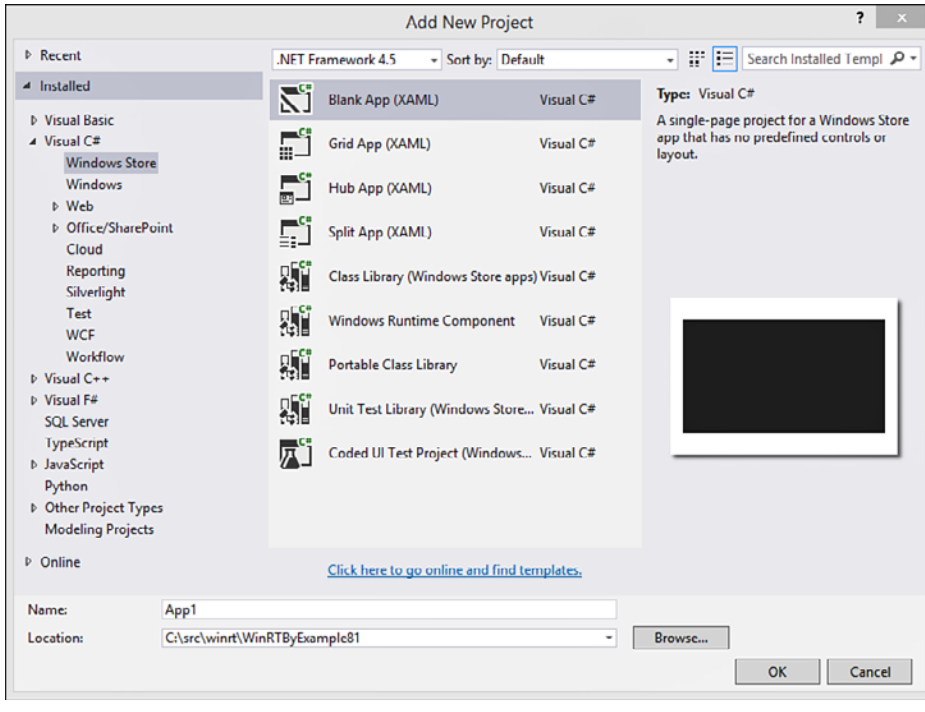


FIGURE 2.1 Available project templates for Windows Store apps

Blank App

This template provides the bare minimum you need to get up and running with a Windows Store app. It provides an application object, a basic set of styles, and an empty main page. Use this template when your application doesn't follow the hierarchical navigation patterns of the grid and split application templates.

Grid App

The **HelloWorldGridApp** project from Chapter 1, "The New Windows Runtime," provides an example for this template. The template is designed for data that can be logically grouped. Use this template if your application provides data in logical groups or categories. Examples include news applications that group items by topic and cooking applications that group recipes by meal or major ingredient. The main page shows a summary of items by group. You can then drill down into the group or category level and/or the item level. Figure 2.2 shows an example.

FIGURE 2.2 Grid-style template

Hub App

The hub app template, introduced with Windows 8.1, demonstrates how to use the new Hub control. This control is ideal for apps that present a view the user can pan through. The user can then pan through different

sections that present different pieces of information. An example of a hub-style app is the free Bing news app included with the Windows 8.1 installation. You can run the included **ReferenceHubApp** project in the `Chapter2` solution folder (available online at http://winrtexamples.codeplex.com/) to see how the default template works (see Figure 2.3).

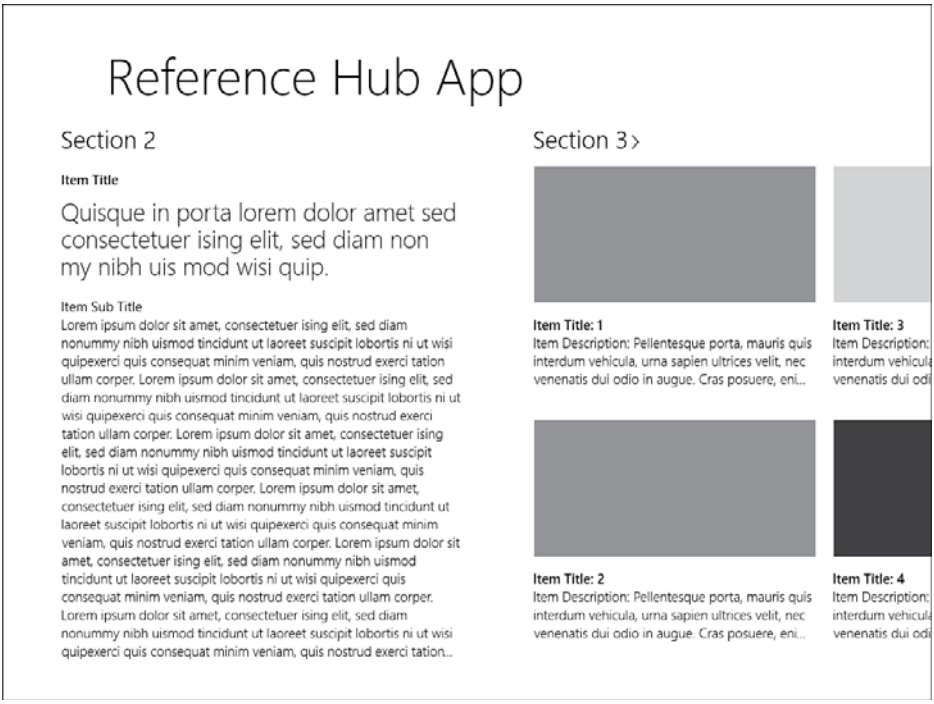


FIGURE 2.3 The hub template

Split App

The split app is similar to the grid app, but it does not provide a single-item view. Instead, it provides a list of groups that enables you to drill down to individual items. The "split page" is a page that lists the items for the group, with details for the item in a side pane. The project **ReferenceSplitApp** demonstrates this template. Figure 2.4 shows the split view.

FIGURE 2.4 The split view

Class Library (Windows Store Apps)

The class library template enables you to create a managed library that other managed Windows Store apps or WinRT components can share. This assembly is not compiled as a WinRT component, so it is available only to other managed code projects (you cannot call a C# class library from C++ or JavaScript Windows Store apps).

Portable Class Library

The Portable Class Library (PCL) provides support for multiple target platforms, including Windows Store apps, .NET Framework desktop applications, Silverlight 5 apps, and even Windows Phone 8 applications. If you don't develop for those platforms, you can skip this section.

The PCL enables you to create a common managed code base that can be shared across platforms. Multiple languages can share WinRT components, but they all must target the same Windows Runtime platform. Portable libraries can be referenced without recompiling on multiple platforms, including the Windows Runtime.

Available platforms include the .NET Framework 4.0 and later, .NET For Windows Store apps (8 and 8.1), Silverlight 5, and Windows Phone 8. The PCL works by defining a list of APIs that are available for each platform. When you select multiple target platforms, a special profile is provided that contains the "lowest common denominator" set of APIs available for those platforms. You can then write a library that uses only references that exist on the target platform. The library is flagged as portable and available to projects you build that target those platforms.

Note that when you select more platforms, the list of supported features narrows dramatically. The key to developing a Portable Library is to pick only platforms that you know you will be targeting, to have the largest API surface area available.

MSDN documentation helps you easily discover which classes and methods the PCL supports. Figure 2.5 shows the documentation for the `WebRequest` object. The icons in the first column indicate whether the given method or constructor is available to the PCL or whether it is accessible from Windows Store apps. You can also scroll to the bottom of the documentation and view the **Versions** section to see whether the given class is available (and for which versions) to the .NET Framework, PCL, and Windows Store apps.

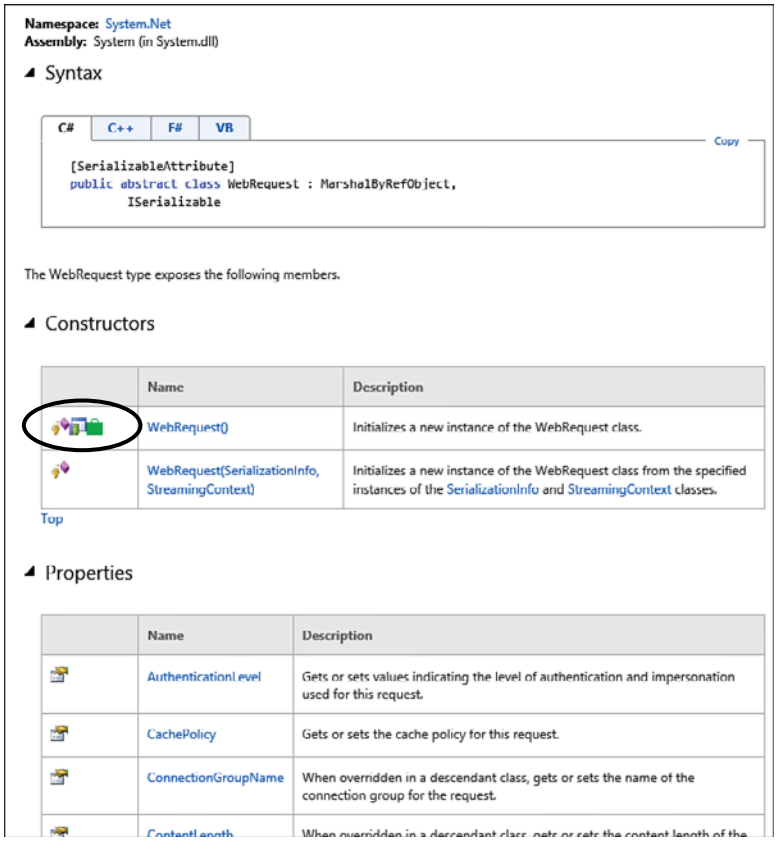


FIGURE 2.5 Documentation for the `WebRequest` object

The PCL helps avoid duplication of code. Instead of copying code for different platforms or using linked files, you can create a single library that is referenced directly. All changes can be consolidated to a single location, and you need to write only one suite of tests against the library. The support for MVVM means you can even include presentation logic in your shared code.

The example solution at http://winrtexamples.codeplex.com/ contains a project called **PortableMvvm** in the `Chapter02` solution folder. To create a Portable Class Library, you add a new project and choose the template for

Portable Class Library. After you click **OK**, the **Add Portable Class Library** dialog appears. For this example, you can choose **.NET Framework 4.5 and Higher**, **Windows Store Apps (Windows 8) and Higher**, and **Silverlight 5**. Uncheck the other options, as in Figure 2.6.

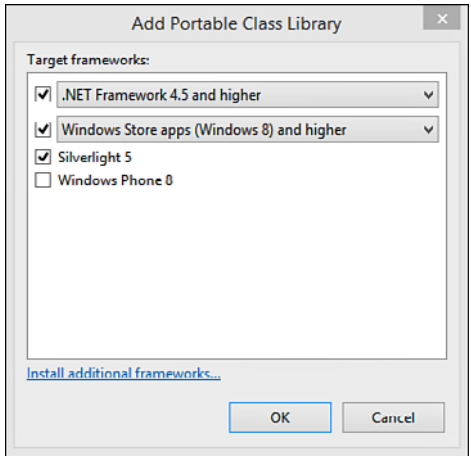


FIGURE 2.6 Selecting target frameworks for a PCL

The PCL template uses a special Platform SDK[1] to create your project. Based on the profile you choose, a reference is made to a set of DLLs that are specially created to expose only the set of functionality that exists across all your target platforms. This enables you to reference components without accidentally including an API that is not compatible.

Listing 2.1 shows the first of two classes the library provides. This demonstrates some presentation logic by exposing a command that can be run only once. It takes in a delegate to perform an action and then sets a flag to indicate that it was run when that action is triggered. It implements the `ICommand` interface that is commonly used with implementations of the MVVM pattern (you learn more about the MVVM pattern in Chapter 3, "Layouts and Controls," and later in Chapter 9, "Model-View-ViewModel [MVVM]").

[1]How to: Create a Software Development Kit, http://bit.ly/SRpH4m

LISTING 2.1 A Portable Class That Implements ICommand

```
public class RunOnceCommand : ICommand
{
    private readonly Action thingToDo
        = delegate { };

    private bool alreadyRan;

    public RunOnceCommand(Action thingToDo)
    {
        this.thingToDo = thingToDo;
    }

    public event EventHandler CanExecuteChanged;

    public bool CanExecute(object parameter)
    {
        return !this.alreadyRan;
    }

    public void Execute(object parameter)
    {
        this.thingToDo();
        this.alreadyRan = true;

        var handler = this.CanExecuteChanged;
        if (handler != null)
        {
            handler(this, EventArgs.Empty);
        }
    }
}
```

This command is perfectly valid to use in multiple platforms. It is also testable without having to invoke a user interface. The second class defined in the portable library uses the command to expose a text property that changes after the command is executed. Listing 2.2 shows that code.

LISTING 2.2 A Portable Class That Implements `INotifyPropertyChanged`

```
public class PortableViewModel : INotifyPropertyChanged
{
    private string tapText;

    public PortableViewModel()
    {
        this.TapCommand = new RunOnceCommand(this.OnTapped);
        this.TapText = "Tap or Click Me.";
    }

    public event PropertyChangedEventHandler PropertyChanged;

    public ICommand TapCommand { get; private set; }

    public string TapText
    {
        get
        {
            return this.tapText;
        }

        set
        {
            if (value == this.tapText)
            {
                return;
            }

            this.tapText = value;
            this.OnPropertyChanged("TapText");
        }
    }

    protected virtual void OnPropertyChanged(string propertyName)
    {
        var handler = this.PropertyChanged;

        if (handler != null)
        {
            handler(this, new PropertyChangedEventArgs(propertyName));
        }
    }

    private void OnTapped()
    {
        this.TapText = "Disabled.";
    }
}
```

The viewmodel simply exposes some text with a call to action. The command is configured so that after it executes, the text changes. This uses data-binding (see Chapter 3) to display the text and invoke the command.

The project includes three sample applications that reference the PCL: **PortableSilverlight** (a Silverlight 5 application), **PortableDesktop** (a WPF application), and **PortableStore** (a Windows Store app). Each project was created by choosing the target platform and template, and then referencing the Portable Library (in the solution it is referenced by project, but you can also generate the PCL assembly and reference that directly). Listing 2.3 shows the XAML used to reference the viewmodel from the Silverlight project. This XAML is almost identical across all three platforms. The only exceptions are how the references are declared and the styles used for the main grid.

LISTING 2.3 Data-Binding Using the Portable viewmodel

```
<Grid x:Name="LayoutRoot" Background="White">
    <Grid.DataContext>
        <portableMvvm:PortableViewModel/>
    </Grid.DataContext>
    <Button
        HorizontalAlignment="Center"
        VerticalAlignment="Center"
        Margin="10"
        Content="{Binding TapText}"
        Command="{Binding TapCommand}"/>
</Grid>
```

Even more incredible is that the logic is all contained within the PCL, so no code-behind exists. I did not have to change a single line of code within the various applications to build the application; all the work was done in XAML by referencing the portable viewmodel and binding to it.

Each program displays a button that asks you to tap or click it. After you complete the requested action (on a touch screen or touch pad if you're tapping, of course), the button is disabled and displays the text Disabled.

Windows Runtime Component

This template enables you to create a managed WinRT component that any Windows Store app or other custom WinRT component can use. The power of this option is that applications written in any language the Windows Runtime supports can then reference your component. The disadvantage is that you can use your component only in Windows Store apps and cannot target multiple platforms the same way you can with PCL.

Unit Test Library

The unit test library enables you to stand up tests that the Windows Runtime runs. The applications are created as a special type of WinRT component that can be called from Visual Studio. Unit tests are written in the same way you do using Microsoft Test System. Consider an example unit test for the `RunOnceCommand` class defined in the **PortableMvvm** project (the test itself is located in the **PortableTests** project).

```
[TestMethod]
public void GivenNeverExecutedWhenCanExecuteCalledThenShouldBeTrue()
{
    var target = new RunOnceCommand(() => { });
    Assert.IsTrue(
        target.CanExecute(null),
        "Test failed: can execute should return true when command
➥ has not been executed.");
}
```

If you expand the **Test** menu and choose **Windows**, **Test Explorer** a dialog opens that lists the available tests (if tests aren't showing and you are in a test project, try rebuilding the project). Figure 2.7 shows the results of running the tests from Visual Studio.

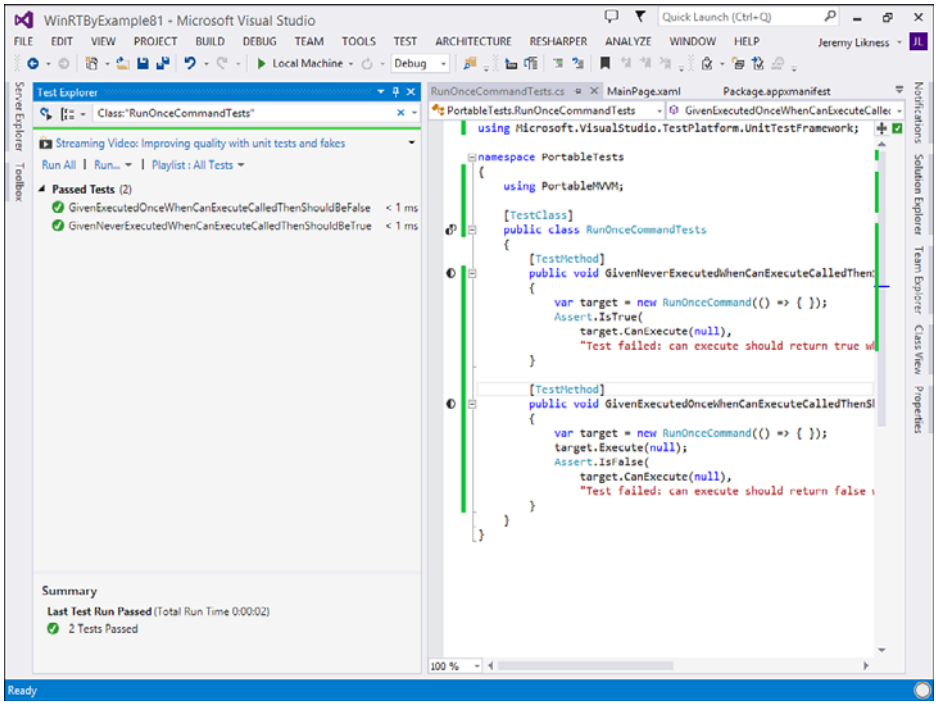


FIGURE 2.7 Running the unit test project

The example project uses the Portable Class Library, but you can reference any valid Windows Store or WinRT component to write your tests against.

Coded UI Test Project

Coded UI tests are automated tests based on the concept of recording a set of actions and playing them back. They are often useful for regression testing an application because you can save a known path through the app and run the test after making changes to ensure that the user interaction is still valid. The MVVM pattern this book advocates enables you to unit-test your logic without having to interface with the UI, so coded UI tests are not covered.

Template Assets

You might have noticed that each project template also provides predefined resources, such as classes, styles, certificates, and images. Some of the item templates also automatically pull in various dependencies as needed to facilitate the functionality required. All Windows Store apps require logos and splash screens, so those images are pulled in automatically. For the examples in this book, most of those assets are shared from a common folder, to avoid duplicating them across projects.

Visual Studio provides a test certificate to sign your app whenever you create a new Windows Store project.[2] For example, the **HelloWorldGridApp** from Chapter 1 contains a file named `HelloWorldGridApp_TemporaryKey.pfx`. The certificate generated is good for one year after the application is generated. When the time period expires, you must either generate a new key or sign the project with a different key (such as one you generate or purchase). You learn more about signing your apps in Chapter 19, "Packaging and Deploying." Individual WinRT components that generate the corresponding metadata files cannot be strong-named and signed.

The more advanced templates include classes to support data-binding and the MVVM pattern. The following list summarizes these classes.

- `NavigationHelper`—Class that aids in navigation between pages and works in conjunction with the `SuspensionManager` to handle process lifetime management.
- `ObservableDictionary`—Class that enables you to keep track of state for views by holding a dictionary of observable objects (that is, objects that implement property and collection change notification).
- `RelayCommand`—Implementation of the command pattern that enables you to specify delegates for execution logic.
- `SuspensionManager`—Helper class for saving values. It is especially useful for restoring state (you learn more about application state later in this same chapter).

[2]Signing an app package, http://bit.ly/XKaJdF

TIP

The **Blank App** template provides only the basic main page and does not include any of the classes that help you implement the MVVM pattern and manage navigation and state. Sometimes you simply don't need the additional pages and code that the more complex **Grid App** and **Split App** templates provide. If you decide to use the **Blank App** template and need the supporting classes as well, adding them to your project is easy. In the project context, go to **Add** and choose **New Item**; then select the template for **Basic Page** instead of **Blank Page**. A dialog appears, prompting you to add dependent files. Click **OK** to add all the supporting files to the project.

Regardless of the template you choose, the end result is a fully functional Windows Store app. One file included with every Windows Store app that this book hasn't yet discussed is the package manifest. The next section covers the contents of the `Package.appxmanifest` file.

Understanding the App Manifest

Listing 2.4 shows the contents of the app manifest for the **HelloWorld GridApp** project. As you can see, the manifest is a simple XML document that helps describe the application. A package is another name for the content of a Windows Store app. The package contains all the code and resources necessary for the application to run. The app manifest describes the content and capabilities of the package.

LISTING 2.4 Contents of `Package.appxmanifest` for HelloWorldGridApp

```
<?xml version="1.0" encoding="utf-8"?>
<Package xmlns="http://schemas.microsoft.com/appx/2010/manifest"
    xmlns:m2="http://schemas.microsoft.com/appx/2013/manifest">
  <Identity Name="WinRTByExampleHelloWorldGridApp"
    Publisher="CN=Jeremy" Version="1.0.0.0" />
  <Properties>
    <DisplayName>Hello World Grid App</DisplayName>
    <PublisherDisplayName>Jeremy</PublisherDisplayName>
    <Logo>Assets\StoreLogo.png</Logo>
  </Properties>
  <Prerequisites>
    <OSMinVersion>6.3.0</OSMinVersion>
    <OSMaxVersionTested>6.3.0</OSMaxVersionTested>
```

```
  </Prerequisites>
  <Resources>
    <Resource Language="x-generate" />
  </Resources>
  <Applications>
    <Application Id="App" Executable="$targetnametoken$.exe"
    EntryPoint="HelloWorldGridApp.App">
      <m2:VisualElements DisplayName="Hello World Grid App"
    Square150x150Logo="Assets\Logo.png"
    Square30x30Logo="Assets\SmallLogo.png"
    Description="An example. . ."
    ForegroundText="light"
    BackgroundColor="#464646">
        <m2:DefaultTile>
          <m2:ShowNameOnTiles>
            <m2:ShowOn Tile="square150x150Logo" />
          </m2:ShowNameOnTiles>
        </m2:DefaultTile>
        <m2:SplashScreen Image="Assets\SplashScreen.png" />
      </m2:VisualElements>
    </Application>
  </Applications>
  <Capabilities>
    <Capability Name="internetClient" />
  </Capabilities>
</Package>
```

You don't have to know the app manifest schema to edit it. Visual Studio 2013 provides a rich user interface for modifying the contents. Double-click the app manifest file in the Solution Explorer to show the **Manifest Designer**.

Application UI

Figure 2.8 shows the Application tab. This tab contains properties that identify and describe the application. The display name is how your app appears in the various dialogs that handle package installation and deployment. The entry point is the class that is called when the app is launched. This is provided by the template and is the class defined in the `App.xaml` and `App.cs` files. The default language[3] is self-explanatory (you learn more about languages in Chapter 18, "Globalization"); it is followed by a description of the application.

[3]Choosing your languages, http://bit.ly/X29FBg

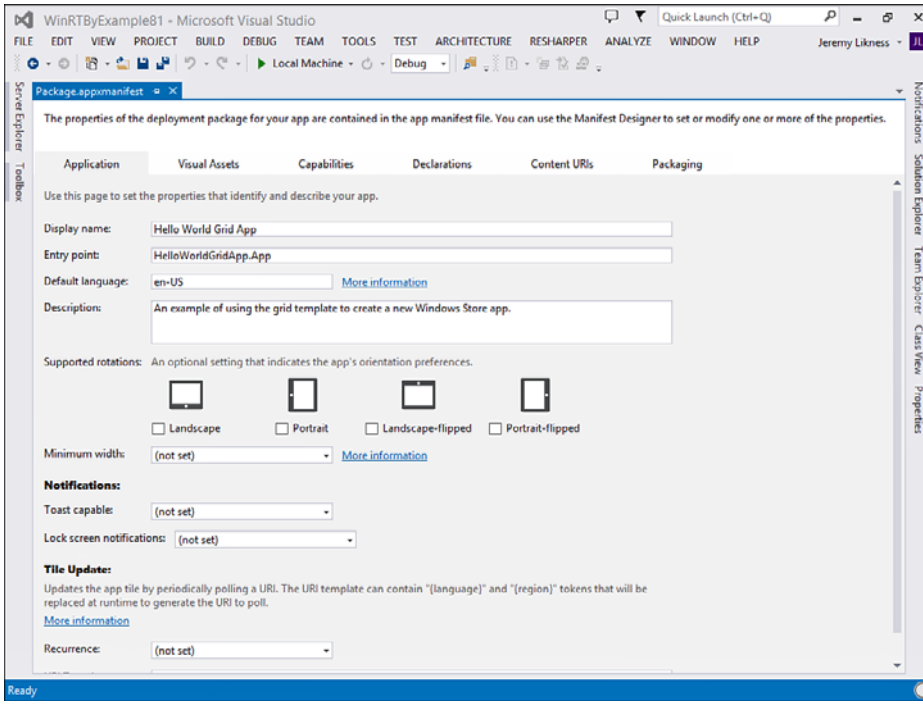


FIGURE 2.8 The Application tab of the manifest

You can use supported rotations to influence the behavior of your app in different orientations. By default, all orientations are supported, and the app resizes to fit the device's orientation when it is rotated. If your app makes sense only in portrait orientation, check the **Portrait** and **Portrait-flipped** options so that the app will not rotate or resize if the user orients the device in landscape mode. (Also note that you can test this feature only on an actual device; the simulator does not support orientation restrictions.)

You can set up **Notifications** for your app to update the live tile (more on this in Chapter 6, "Tiles and Toasts"). You can also specify a URL that dynamically supplies a template for your tile with localization support. In Windows 8.1, the URL is called automatically when the app gets installed, enabling live tiles even before the user starts the app.

The Visual Assets (the next tab) are important because they describe how your app appears to Windows 8.1. For example, the Tile section enables you to enter a Short name that appears on the Start menu and

determine when the name should appear (you can suppress the name if it is already part of the logo graphic, for example). You can indicate the background color and specify whether the font color should be light or dark.

Windows Store apps require Windows to display a splash screen when it gets started. A configurable fixed color is used to fill up the background, while a logo 620x300, 868x420, or 1116x540 pixels wide is centered on the screen. All the assets enable you to provide multiple resolutions to support scaling.[4] The runtime automatically scales your graphics as needed, but providing a native format ensures that it can scale to 140% and 180% of original size without blurring or becoming pixelated. You can also set the logo for a badge icon that appears in notification flyouts and other sized logos used for larger or smaller tiles, and in the preview view when you switch active apps. Windows uses a naming convention to select the appropriate asset. `SplashScreen.png` automatically is scaled, but if you provide a `SplashScreen.scale-100.png` and `SplashScreen.scale-140.png`, Windows automatically picks the appropriate asset to scale at 100% or 140%, respectively.

Capabilities

The Capabilities tab enables you to specific the features and sensors your app will use. Capabilities help inform the Windows Store about what features to test, and having many capabilities can increase the level of testing of your app when submitted.[5] Some capabilities, such as Document Library access, require a Corporate developer account, and Internet access might require your app to include a privacy policy. The manifest also informs users of the capabilities your app needs when listed in the Windows Store. At execution time, the Windows Runtime checks the capabilities you set when the corresponding APIs are called. Table 2.1 lists the available capabilities, along with a brief description.

---

[4]Guidelines for scaling to pixel density, http://bit.ly/111GtSc

[5]App capability declarations, http://bit.ly/Uvcein

TABLE 2.1 Windows Store App Capabilities

Capability	Description
Enterprise Authentication	Connects to intranet resources that require domain credentials.
Internet (Client)	Turned on by default; enables access to the Internet and other networks in public places (when the user has designated the active network as public).
Internet (Client & Server)	Provides both inbound and outbound access to the Internet and other networks when the active network is designated as public.
Location	Accesses the current location (aggregated from various sources, including the network address and GPS sensor, when available).
Microphone	Accesses the audio feed for the attached microphone devices.
Music Library	Adds, changes, and deletes files in the Music Library (applies to both local PC and HomeGroup libraries).
Pictures Library	Adds, changes, and deletes files in the Pictures Library (applies to both local PC and HomeGroup libraries).
Private Networks (Client & Server)	Provides both inbound and outbound access to networks that have been designated as home or work networks, or ones that require domain authentication.
Proximity	Allows Near Field Communications (NFC).
Removable Storage	Adds, changes, and deletes files on removable storage devices. As with the Documents Library, access is restricted by the File Type Associations, and access to HomeGroup is excluded.
Shared User Certificates	Accesses certificates (smart cards, x.509, and so on) used to validate the user's identity.
Videos Library	Adds, changes, and deletes files in the Videos Library (applies to both local PC and HomeGroup libraries).
Webcam	Accesses the video feed from connected webcams.

All capabilities are associated with WinRT APIs and components. For example, use of the `MediaCapture` component implies the webcam capability. You must declare the proximity capability if you will use the `ProximityDevice` class. You learn more about various WinRT APIs in later chapters.

Declarations

Declarations provide extensibility points for your app by allowing it to either connect with other apps through contracts or enhance the OS through extensions.[6] You learn more about declarations in Chapters 11 and 12. Table 2.2 summarizes the available declarations.

TABLE 2.2 Windows Store App Declarations

Declaration	Description
Account Picture Provider	Enables your app to appear as an option and be invoked to provide an account picture when the user is changing settings.
AutoPlay Content	Registers the app to handle events such as the user inserting a DVD.
AutoPlay Device	Registers the app to handle device change events, such as the user connecting a webcam.
Background Tasks	Enables the app to specify the name of a class that can run code in response to triggered events, including audio, notifications, timer, and more.
Cached File Updater	Enables the app to provide files to other Windows 8 apps and provides the triggers to synchronize the files based on local or remote updates.
Camera Settings	Indicates that your app can provide a custom user interface for selecting camera options when the user is taking a picture or recording video.
Certificates	Used to install certificates with the Windows Store app to enable secure communications channels, signing of digital content, and encryption of data.

[6] App contracts and extensions, http://bit.ly/WhFJVa

Declaration	Description
Contact Picker	Enables your app to provide contact information when users access their contacts from any other app.
File Open Picker	Makes your app an option when the user invokes the file picker to browse for files. For example, a cloud storage app such as OneDrive (formerly SkyDrive) can provide a UI that enables users to pick content from their OneDrive account.
File Save Picker	Enables your app to appear as an option when the user is saving content.
File Type Associations	Indicates file types that your app can manage. The Documents Library and Removable Storage capabilities use this declaration.
Print Task Settings	Indicates that your app will supply custom UI for printer settings.
Protocol	Enables your app to manage an existing communications protocol (for example, register the **mailto** protocol to handle email messages) or define a custom protocol.
Search	Registers to participate as a search provider, enabling the user to search your app from the Search Charm while it is active on the screen. Note that using the SearchBox in Windows 8.1 instead of implementing this contract is recommended. If a SearchBox is used in an app that implements the Search contract, an exception is thrown.
Share Target	Indicates that your app is capable of receiving shareable content. Includes the data formats and file types your app can receive.

Another way to think about declarations is as alternative entry points to start your application. For example, when the user invokes the Share Charm or opens a file picker, your app might be activated to handle that specific interaction.