"Jeremy builds real apps for real customers. That's why I can heartily recommend this book. Go out and write some great apps...and keep this book handy."

—From the Foreword by **Jeff Prosise**

Building Windows® 8 Apps *with* **C#** and **XAML**

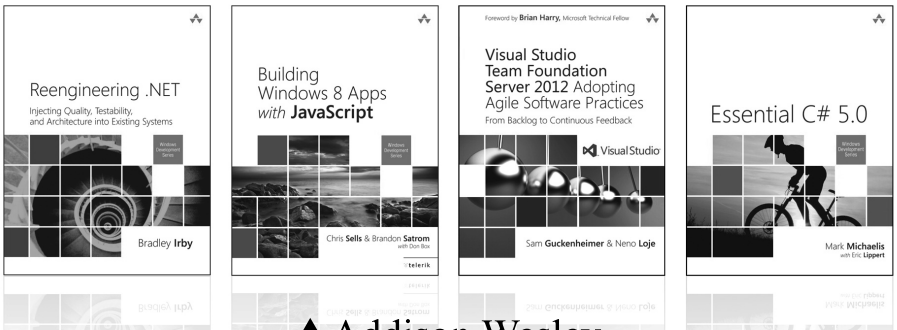
Windows Development Series

Jeremy **Likness**

Wintellect
Know how.

Building Windows 8 Apps with C# and XAML

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."

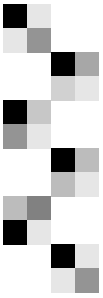
– JOHN MONTGOMERY, Principal Director of Program Management, Microsoft

"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, Vice President, Developer Tools Division, Telerik

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

Addison Wesley | informIT.com THE TRUSTED TECHNOLOGY LEARNING SOURCE | Safari Books Online



Building Windows 8 Apps with C# and XAML

Jeremy Likness

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 Cataloging-in-Publication Data is on file and available upon request.

ISBN-13: 978-0-321-82216-1
ISBN-10: 0-321-82216-1

Text printed in the United States on recycled paper at RR Donnelley and Sons, Crawfordsville, Indiana.

First printing, October 2012

To Ma: Your support and encouragement
have always been a blessing to me.
I will miss not being able to share
that I finished this book with you.

Contents at a Glance

Contents

Foreword

The life of the software developer isn't an easy one. Every ten years or so, he has to throw away everything he knows and start all over again. Times change, and technologies change even faster. A decade ago, developers had to retool their skill sets for the move from Win32 to .NET and C#. Today, there's a new platform in town. It's called Windows 8, and with it comes a profound shift in the way Windows apps are conceived and executed.

Windows 8 is like no Windows the world has seen before. The new Windows programming model favors simplicity, security, and battery efficiency above all else. Modern Windows apps run full-screen, single-instance, and one at a time. Their UIs can be built in XAML, HTML, or DirectX. They run in a sandbox that stops malicious code in its tracks, and they're inspected before they're published in the Windows Store to make sure they don't violate the sandbox. They prefer touch screens but play equally well with mice and other input devices. Moreover, they install with a single click and uninstall without leaving a trace.

Underneath the new user interface is a new API: The Windows Runtime API, better known as WinRT. WinRT represents a rethinking of what the Windows API would look like if it were redesigned from the ground up. The old Windows API is outdated, overly complex, and tied to a specific language. The WinRT API, by contrast, is thoroughly modern and can be

called from a variety of languages. Indeed, one of the most remarkable aspects of Windows 8 is that for the first time in history, a developer versed in HTML and JavaScript enjoys the same ability to write Windows apps as developers who speak XAML and C#.

What it means for the developer is—you guessed it—time to start over again. WinRT *is* the Windows API now, and the new UI layer, formerly known as "Metro," is the new face of Windows apps. Be bold or be left behind.

Becoming a Windows 8 developer means learning WinRT. It means getting comfortable with asynchronous programming. It means understanding that a Windows app that isn't visible to the user is suspended and that an app that's suspended can be unceremoniously terminated by the operating system at any time. It means learning about contracts, which allow apps to integrate with the charms that slide out from the right side of the screen. It means learning about live tiles, push notifications, and other features that make an app a first-class citizen in the Windows 8 environment. It means understanding the Windows 8 design philosophy and how to use XAML to craft compelling, fluid, and responsive Windows UIs.

When you're in the wilderness, it helps to have a guide who has been there before. I can't think of anyone more qualified to lead you on the journey to Windows 8 enlightenment than Jeremy Likness. Jeremy is the only person I know who works 32 hours a day. (He sleeps the other four.) I used to say that I might work with people a lot smarter than me, but none of them can work more hours than me. I've had to reconsider that with Jeremy. Shoot him an e-mail at 3:00 a.m., and you'll have a reply by 3:02. That's why he's a Principal Consultant at Wintellect and why we turn to him to architect and implement Windows 8 solutions for our customers. A teacher can be only so effective if he isn't out there working in the trenches. Jeremy builds real apps for real customers. That's why I can heartily recommend this book and why I'm excited to see how it's received by the community.

Windows 8 is a bold move on Microsoft's part—perhaps the boldest move the company has made since the introduction of Windows itself. But it's the right move at the right time. The action in software development for the next ten years won't revolve around traditional PCs. It'll be in writing

apps for tablets, phones, and other mobile devices. Companies will be built and millionaires will be made from apps for devices with portable form factors, including Microsoft's new Surface tablet. To ignore WinRT is to ignore the part of the Microsoft stack that lets you write for these devices.

Learn WinRT. Go out and write some great apps. Help make this platform a success. And keep this book handy. When you run into problems, it's the next best thing to an instant response to a 3:00 a.m. e-mail. From Jeremy's perspective, it's even better.

—*Jeff Prosise*, Co-Founder, Wintellect

Preface

The first whispers about Windows 8 surfaced in early 2011. Widespread speculation swept the Internet as developers began to question what the new platform would look like. The rumors included a new platform that wouldn't support the .NET Framework, was based solely on C++ or HTML5 and JavaScript, and wouldn't run any existing software. Early builds and screenshots leaked over Twitter but this only fueled speculation. Finally Steven Sinofsky, President of the Windows Division at Microsoft, took the stage on September 13, 2011 and released an early build of Windows 8 to the world.

I was one of the first eager programmers to download the early build, and I installed it in a virtual machine. It didn't take long for me to realize that the .NET Framework was alive and well, I could run my existing Silverlight applications on the new platform, and C# and XAML were tools available to build the new "Metro-style" applications (this name was changed to Windows Store applications with the RTM version of Windows). I didn't make it to the //BUILD conference hosted in California to release Windows 8, but the sessions were made available almost immediately after they were presented, and I watched them every evening, morning, or while I was traveling by plane.

The Windows 8 platform features the Windows Runtime, a new framework for building applications that provides capabilities never before available on a Windows machine. I was building applications within

days and was delighted to find that my existing C# and XAML skills from Silverlight and Windows Presentation Foundation (WPF) applied to the runtime, while a new set of components made it easier than ever to developer rich, touch-based applications. It wasn't long before I reached out to the publisher of my book, *Designing Silverlight Business Applications*, and said, "I want to write my next book on Windows 8."

I was fortunate to get involved in an early adoption program with Microsoft. The consulting and training firm I work for, Wintellect, was hired to provide some hands-on labs and workshops specifically targeted to new developers who want to learn how to build applications for Windows 8. This gave me critical access to early builds of the product and enabled me to start writing about the various features that would ultimately become part of the final release. As I built samples that covered manipulating objects on screen with touch, sharing rich content between applications, and providing live interactive tiles on the start menu with at-a-glance content, my excitement quickly grew.

As part of writing this book, I wrote an article that shares what I believe are the top 10 reasons developers will love building Windows 8 applications. You can read the full article online at:

http://www.informit.com/articles/article.aspx?p=1853667

In summary, these are the reasons I think you will enjoy the new platform:

- **Programming language support**—It is possible to write Windows 8 applications with VB, C#, C++, and XAML, or a special stack that includes HTML5 and JavaScript.
- **XAML**—Developers familiar with the power and flexibility of XAML who have written Silverlight and/or WPF applications in the past will be very comfortable with the XAML used to develop Windows 8 applications.
- **HTML5**—The rich support for HTML5 as a markup option will appeal to web developers crossing over to tablet and touch-based development, although this book will deal primarily with the C# and XAML option.

- **Windows Runtime (WinRT)**—The Windows Runtime provides a number of controls, components, classes, and method calls that make performing complex tasks both consistent and easy using just a few lines of code.
- **Contracts**—The new system of "Contracts" enables a new level of sharing and integration between applications and the end user.
- **Asynchronous support**—The introduction of the `await` and `async` keywords makes programming multi-threaded code more straightforward than it has even been.
- **Touch**—Touch-based input is first-class in Windows 8 applications with out-of-the-box support from all of the available controls and a straightforward API to interface with touch events and manipulations.
- **Settings**—The settings experience (provided via a Contract) provides a very consistent and familiar way for developers to allow end users to configure their application preferences.
- **Roaming profiles**—Building code that synchronizes across Windows 8 machines through the cloud is simple and easy. (You can literally share a data file with a single line of code.)
- **Icons**—Windows 8 features a rich set of pre-existing icons that you can use to provide consistent interfaces for commands within your applications.

To avoid confusion, I refer to the special new programs built specifically for Windows 8 as "Windows 8 applications" throughout this book. The templates to create these new applications in Visual Studio 2012 are grouped under the name "Windows Store." Although these applications can be distributed through the Windows Store, you can also distribute traditional desktop-style applications through the store. Therefore, I will only use "Windows Store" when I refer to the Visual Studio 2012 templates, or when I compare the newer style of application to the traditional desktop style. Everywhere else, you will see them referred to as "Windows 8 applications."

The top 10 items just scratch the surface of the new platform. Windows 8 is definitely different than previous releases of Windows and does require change. You will need to adopt a new interface that elevates touch to a first-class citizen but always provides ways to navigate using the mouse and keyboard. You will have to get used to code that calls native unmanaged components in a way that is almost transparent, and deal with a new set of controls and components that previously did not exist. This book is intended to guide you through the process of learning the new territory quickly so you can begin building amazing new applications using skills you already have with C# and XAML.

What This Book Is About

The purpose of this book is to explain how to write Windows 8 applications using the C# programming language, Extensible Application Markup Language (XAML), the Windows Runtime, and the .NET Framework. For this book, I assume you have some development experience. While I do cover basic topics related to C# and XAML, I try to focus on the areas specific to building Windows 8 applications. Where I introduce more advanced concepts specific to C# or XAML that are not specific to the Windows 8 platform, I reference other books, articles, or online resources so you can further explore those fundamentals.

Whether you're an existing Silverlight of WPF developer looking to migrate an existing application, or a developer who is transitioning the Windows 8 platform for the first time, this book will give you the guidance and resources to quickly learn what you need to go from a new project to a published application in the application store.

How to Use This Book

The goal of this book is to enable you to write Windows 8 applications with C# and XAML. Each chapter is designed to help you move from a fundamental understanding of the target platform to building your first application. Code examples are provided that demonstrate the features and best practices for programming them. Most chapters build on previous content

to provide a continuous narrative that walks through all of the components that make up a typical Windows 8 application.

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 code samples and walk-throughs to demonstrate the application of the topics. The code samples are explained in detail and the topic is summarized to highlight the specific information that is most important for you to consider.

I suggest you read the book from start to finish, regardless of your existing situation. You will find that your understanding grows as you read each chapter and concepts are introduced, reinforced, and tied together. After 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.

My Experience with the Microsoft Stack

My first computer program was written in BASIC on a TI-99/4A. From there I programmed assembly language for the Commodore 64, learned C and C++ on Unix-based systems, and later wrote supply chain management software on the midrange AS/400 computer (now known as iSeries). For the past 20 years, my primary focus has been developing scalable, highly concurrent web-based enterprise applications.

I started my work with Silverlight right before the 3.0 release. At the time, I led a team of 12 developers working on an ASP .NET mobile device management platform that relied heavily on AJAX to provide a desktop-like user experience. When it was evident that the team was spending more time learning various web technologies such as CSS and JavaScript and testing the application on multiple browsers and platforms than focusing on core business value, I began researching alternative solutions and determined that Silverlight was the key our team was looking for.

Since that transition, I worked on XAML applications in the enterprise along with large-scale web applications built with the ASP.NET MVC framework. In addition to the mobile device management software, I helped build the health monitoring system for the back-end data

centers that provided video streams (live and on demand) during the 2010 Vancouver Winter Olympics. I worked on a major social media analytics project that used Silverlight to present data that was mined from social networks and analyzed to provide brand sentiment. I worked with a team that built a slate-based sales interface for field agents to close sales and integrate with their point of sale system. I was on the team that produced the Silverlight version of a major eBook reading platform designed for accessibility and customized to provide interactive experiences and audio for children.

All of this work has been with the company Wintellect, founded by well-known .NET luminaries Jeffrey Richter, Jeff Prosise, and John Robbins. All three have produced countless books about the Microsoft stack, .NET Framework, and Core Language Runtime (CLR). They have trained thousands of Microsoft employees (some teams at Microsoft are required to take their courses as a prerequisite to working on their projects) and contributed to the runtime itself by writing and designing portions of the framework. The company has provided me with unique access to industry leaders and architects and their best practices and solutions for creating successful enterprise applications.

I am certified in various XAML technologies, including Microsoft Silverlight developer (MCTS) and WPF Developer (MCP). I was recognized as a Microsoft Most Valuable Professional® (MVP) for Silverlight in July of 2010 and was re-awarded the title in 2011 and 2012. This was due mostly to my efforts to blog, tweet, and speak about XAML technologies at various user group meetings and conferences around the country. I have conducted hands-on labs and training for Windows 8, worked on its earliest builds, and continue to blog and write about the platform as it develops. It is my depth of experience working with XAML and understanding how to build server and web-based software that has provided me with valuable insights into how to build Windows 8 applications.

Acknowledgments

I've learned that a technical book like this is written by a team even though the author gets the credit. Joan Murray once again has helped drive this book to completion and provided incredible support and encouragement throughout the process. She works with an amazing team. Special thanks to development editor Ellie Bru for staying on top of every draft, figure, revision, edit, and feedback loop. Thanks to Lori Lyons and Christal White for keeping me consistent and correcting all of my bad grammar habits to help me present like a polished writer.

I have enormous gratitude to my co-workers at Wintellect for their support. Steve Porter and Todd Fine once again helped support my efforts to balance long evenings and early mornings writing with my daily activities. Jeffrey Richter and Jeff Prosise gave me plenty of insights and wisdom based on their years of writing incredible books before me. Jeffrey's code samples and Jeff's labs were priceless tools that helped me learn the new platform and evolve the scenarios I share in this book. John Garland was a companion on this journey of learning a new platform and technology, and served once again as a brilliant technical editor and helped me shape and organize the content.

Special thanks to Telerik for supporting me at many levels. I appreciate Jesse Liberty bringing me onto his podcast, Chris Sells for his assistance on the HTML and JavaScript side, and Michael Crump for not only

supporting this book at multiple levels but also taking the time to provide valuable feedback as a technical editor.

Thanks to the Microsoft team who worked with me through multiple iterations of the Windows 8 platform, from Developer Preview to Consumer Preview, onto Release Preview and beyond. Thanks to Jaime Rodriguez, Tim Heuer, Joanna Mason, Jennifer Marsman, and Layla Driscoll for all of your knowledge and insights. David Kean, I appreciate you patiently explaining the portable class library at the MVP summit and all of your patience and support afterward. Daniel Plaisted, you've been a tremendous help along the way.

These acknowledgments wouldn't be complete without a nod to my fellow MVPs and online supporters who have actively promoted and supported this book. Special thanks to Davide Zordan, Shawn Wildermuth, Jeff Albrecht, Roberto Baccari, David J. Kelley, Zubair Ahmed, and Ginny Caughey. Thanks to the Linked In .NET Users Group (LIDNUG), and especially Peter Shawn and Brian H. Madsen for your support and for providing me with a platform to share my excitement about Windows 8. Thanks to Chris Woodruff and Keith Elder for helping me get "deep fried" on their show.

Thanks to everyone who shared my tweets or visited the Facebook page for this book. Thanks to all of the early readers who provided feedback through the rough cuts of this book, and thanks to *you*, kind reader, for being you!

Last but certainly not least, thanks once again to my superstar wife and incredible daughter for understanding why Dad had to lock himself in the office late at night and in the very early hours of the morning. I couldn't have done this without my girls!

About the Author

Jeremy Likness is a principal consultant at Wintellect, LLC. He has worked with enterprise applications for more than 20 years, 15 of those focused on web-based applications using the Microsoft stack. An early adopter of Silverlight 3.0, he worked on countless enterprise Silverlight solutions, including the back-end health monitoring system for the 2010 Vancouver Winter Olympics and Microsoft's own social network monitoring product called "Looking Glass." He is both a consultant and project manager at Wintellect and works closely with Fortune 500 companies, including Microsoft. He is a three-year Microsoft MVP and was declared MVP of the Year in 2010. He has also received Microsoft's Community Contributor award for his work with Silverlight. Jeremy is the author of *Designing Silverlight Business Applications: Best Practices for Using Silverlight Effectively in the Enterprise* (Addison-Wesley). Jeremy regularly speaks, contributes articles, and blogs on topics of interest to the Microsoft developer community. His blog can be found at http://csharperimage.jeremylikness.com.

6
Data

Data is central to most applications, and understanding how to manage data and transform it into information the user can interact with is critical. Windows 8 applications can interact with data in a variety of ways. You can save local data, retrieve syndicated content from the Web, and parse local resources that are stored in JSON format. You can query XML documents, use WinRT controls to direct the user to select files from the file system, and manipulate collections of data using a structured query language.

In this chapter, you learn about the different types of data that are available to your Windows 8 application and techniques for manipulating, loading, storing, encrypting, signing, and querying data. You'll find that the WinRT provides several ready-to-use APIs that make working with data a breeze. This chapter explores these APIs and how to best integrate them into your application.

Application Settings

You were exposed to application settings in Chapter 5, *Application Lifecycle*. Common cases for using application settings include

- Simple settings that are accessed through the **Settings** charm and can be synchronized between machines (Roaming)

- Local data storage persisted between application sessions (Local)
- Local persistent cache to enable occasionally disconnected scenarios (Local)
- Temporary cached data used as a workspace or to improve performance of the application (Temporary)

The settings use a simple dictionary to store values and require the values you store to be basic WinRT types. It is possible to store more complex types. In Chapter 5, you learned how to manually serialize and de-serialize an item by writing to a file in local storage. You serialize complex types using a serialization helper. An example of this exists in the `SuspensionManager` class that is included in the project templates. You can search for the file **SuspensionManager.cs** on your system to browse the source code.

The `SuspensionManager` class uses the `DataContractSerializer` to serialize complex types in a dictionary:

```
DataContractSerializer serializer =
   new DataContractSerializer(typeof(Dictionary<string, object>),
      knownTypes_);
serializer.WriteObject(sessionData, sessionState_);
```

The serializer (in this case, the `DataContractSerializer` class) automatically inspects the properties on the target class and composes XML to represent the class. The XML is written to a file in the folder allocated for the current application. Similar to the various containers for application settings (local, roaming, and temporary), there is a local folder specific to the user and application that you can use to create directories and read and write files. Accessing the folder is as simple as

```
StorageFile file =
   await ApplicationData.Current.LocalFolder.
CreateFileAsync(filename,
   CreationCollisionOption.ReplaceExisting);
```

You can access a roaming or temporary folder as well. The `CreateCompletionOption` is a feature that allows you generate filenames that don't conflict with existing data. The options (passed in as an `enum` to the file method) include:

- `FailIfExists`—The operation will throw an exception if a file with that name already exists.
- `GenerateUniqueName`—The operation will append a sequence to the end of the filename to ensure it is a unique, new file.
- `OpenIfExists`—If the file already exists, instead of creating a new file, the operation will simply open the existing file for writing.
- `ReplaceExisting`—Any existing file will be overwritten. The example will always overwrite the file with the XML for the dictionary.

After the dictionary has been written, the serialization helper is used to de-serialize the data when the application resumes after a termination:

```
DataContractSerializer serializer =
   new DataContractSerializer(typeof(Dictionary<string, object>),
      knownTypes_);
sessionState_ = (Dictionary<string, object>)serializer
   .ReadObject(inStream.AsStreamForRead());
```

The local storage can be used for more than just saving state. As demonstrated in Chapter 5, you may also use it to store data. It can also be used to store assets like text files and images. A common design is to use local storage to save cloud-based data that is unlikely to change as a local cache. This will allow your application to operate even when the user is not connected to the Internet and in some cases may improve the performance of the application when the network is experiencing high latency. In the next section, you learn more about how to access and save data using the Windows Runtime.

Accessing and Saving Data

Take a moment to download the **Wintellog** project for Chapter 6, *Data,* from the book website at http://windows8applications.codeplex.com/.

You may need to remove TFS bindings before you run the project. This is a sample project that demonstrates several techniques for accessing and saving data. The application takes blog feeds from various Wintellect employees and caches them locally on your Windows 8 device. Each time you launch the application, it scans for new items and pulls those down.

These blogs cover cutting-edge content ranging from the latest information about Windows 8 to topics like Azure, SQL Server, and more. You may recognize some of the blog authors including Jeff Prosise, Jeffrey Richter, and John Robbins.

You learned in Chapter 5 about the various storage locations and how you can use either settings or the file system itself. The application currently uses settings to track the first time it runs. That process takes several minutes as it reads a feed with blog entries and parses the web pages for display. An extended splash screen is used due to the longer startup time. You can see the check to see if the application has been initialized in the `ExtendedSplashScreen_Loaded` method in **SplashPage.xaml.cs**:

```
ProgressText.Text = ApplicationData.Current.LocalSettings.Values
    .ContainsKey("Initialized") && (bool)ApplicationData.Current.
LocalSettings.Values["Initialized"]
    ? "Loading blogs..." :
"Initializing for first use: this may take several minutes...";
```

After the process is completed, the flag is set to true. This allows the application to display a warning about the startup time the first time it runs. Subsequent launches will load the majority of data from a local cache to improve the speed of the application:

```
ApplicationData.Current.LocalSettings.Values["Initialized"]
   = true;
```

There are several classes involved with loading and saving the data. Take a look at the `StorageUtility` class. This class is used to simplify the process of saving items to local storage and restoring them when the application is launched. In `SaveItem`, you can see the process to create a folder and a file and handling potential collisions as described in Chapter 5 (extra code has been removed for clarity):

```
var folder = await ApplicationData.Current.LocalFolder
                    .CreateFolderAsync(folderName,
   CreationCollisionOption.OpenIfExists);
var file = await folder.CreateFileAsync(item.Id.GetHashCode().
➥ToString(),
   CreationCollisionOption.ReplaceExisting);
```

Notice that the method itself is marked with an `async` keyword, and the file system operations are preceded by `await`. You learn about these keywords in the next section. Unlike the example in Chapter 5 that manually wrote the properties to storage, the `StorageUtility` class takes a generic type to make it easier to save any type that can be serialized. The code uses the same engine that handles complex types transmitted via web services (you will learn more about web services later in this chapter). This code uses the `DataContractJsonSerializer` to take the snapshot of the instance that is saved:

```
var stream = await file.OpenAsync(FileAccessMode.ReadWrite);
using (var outStream = stream.GetOutputStreamAt(0))
{
    var serializer = new DataContractJsonSerializer(typeof(T));
    serializer.WriteObject(outStream.AsStreamForWrite(), item);
    await outStream.FlushAsync();
}
```

The file is created through the previous call and used to retrieve a stream. The instance of the `DataContractJsonSerializer` is passed the type of the class to be serialized. The serialized object is written to the stream attached to the file and then flushed to store this to disk. The entire operation is wrapped in a `try … catch` block to handle any potential file system errors that may occur. This is common for cache code because if the local operation fails, the data can always be retrieved again from the cloud.

GENERICS 101

Generics are a very important feature of the C# language. They provide the ability to create a template for type-safe code without committing to a specific type. The `SaveItem<T>` method is a template for saving an instance of an unknown type. When it is called from the `BlogDataSource` class, the compiler inspects the type that is passed and generates code specific for that type. The definition is an open generic and the call closes the generic with a specific type. You may be familiar with generics in collections, like List<T>, but they can be used in far more powerful and flexible solutions. Learn more about generics online at: http://bit.ly/csharpgenerics.

To see how the serialization works and where the files are stored, run the application and allow it to initialize and pass you to the initial grouped item list. Hold down the **Windows Key** and press **R** to get the run dialog. In the dialog, type the following:

```
%userprofile%\AppData\Local\Packages
```

Press the **Enter** key, and it will open the folder.

This is where the application-specific data for your login will be stored. You can either try to match the folder name to the package identifier or type **Groups** into the search box to locate the folder used by the Wintellog application. When you open the folder, you'll see several folders with numbers for the name and a single folder called **Groups**, similar to what is shown in Figure 6.1.

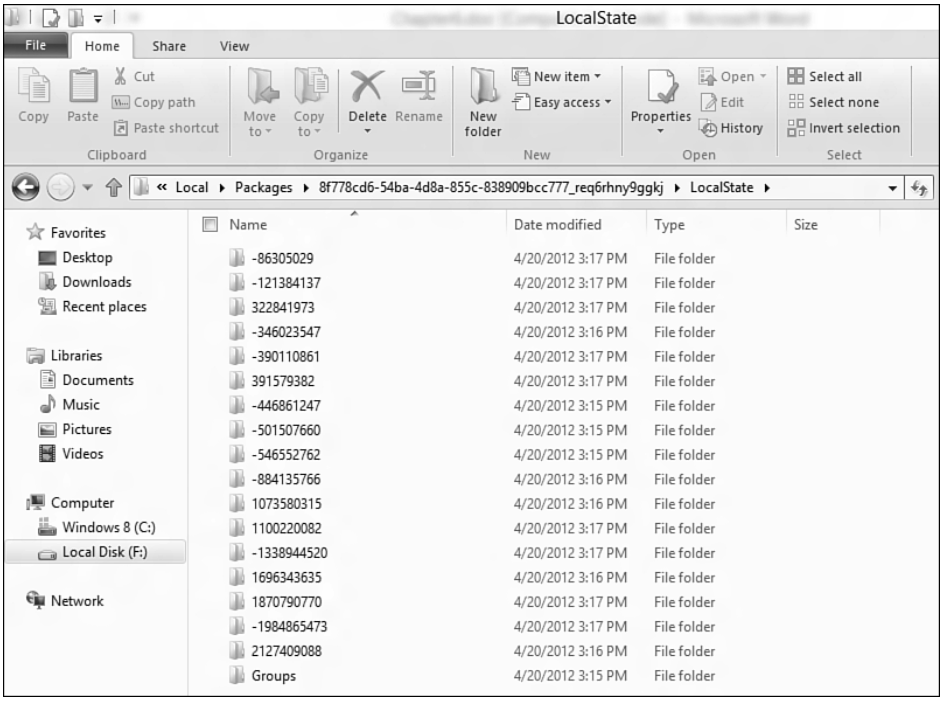


FIGURE 6.1: The local cache for the Wintellog application

To simplify the generation of filenames, the application currently just uses the hash code for the unique identifier of the group or item to establish

a filename. A hash code is simply a value that makes it easier to compare complex objects. You can read more about hash codes online at http://msdn.microsoft.com/en-us/library/system.object.gethashcode.aspx.

Hash codes are not guaranteed to be unique, but in the case of strings, it is highly unlikely that the combination of a group and a post would cause a collision. The **Groups** folder contains a list of files for each group. Navigate to that folder and open one of the items in **Notepad**. You'll see the JSON serialized value for a `BlogGroup` instance.

JAVASCRIPT OBJECT NOTATION (JSON)

JSON is an open standard for storing text-based information. The default serialization engine for web services uses Extensible Markup Language (XML) to store values in a structured document. JSON uses a different approach that has become popular because it takes less space and is easier to read and understand than XML. It uses valid JavaScript syntax to describe objects, so the code can be executed by a JavaScript interpreter to easily create an object. You can search for the keywords "JSON Visualizer" to find several websites that will allow you to paste JSON and see a visual interpretation of the object it represents. The standard is defined online at http://json.org/.

The JSON is stored in a compact format on disk. The following example shows the JSON value for my blog, formatted to make it easier to read:

```
{
   "Id" : "http://www.wintellect.com/CS/blogs/jlikness/default.aspx",
   "PageUri" :
   "http://www.wintellect.com/CS/blogs/jlikness/default.aspx",
   "Title" : "Jeremy Likness' Blog",
   "RssUri" : "http://www.wintellect.com/CS/blogs/jlikness/rss.aspx"
}
```

The syntax is straightforward. The braces enclose the object being defined and contain a list of keys (the name of the property) and values (what the property is set to). If you inspect any of the serialized posts (those are contained in a folder with the same name as the group hash code), you will notice the `ImageUriList` property uses a bracket to specify an array:

```
"ImageUriList" : [
   "http://www.wintellect.com/.../Screen_thumb_42317207.png",
   "http://www.wintellect.com/.../someotherimage.png" ]
```

You may have already looked at the `BlogGroup` class and noticed that not all of the properties are being stored. For example, the item counts are always computed when the items are loaded for the group, so they do not need to be serialized. This particular approach requires that you mark the class as a `DataContract` and then explicitly tag the properties you wish to serialize. The `BlogGroup` class is tagged like this:

```
[DataContract]
public class BlogGroup : BaseItem
```

Any properties to be serialized are tagged using the `DataMember` attribute:

```
[DataMember]
public Uri RssUri { get; set; }
```

If you have written web services using Windows Communication Foundation (WCF) in the past, you will be familiar with this format for tagging classes. You may not have realized it could be used for direct serialization without going through the web service stack. The default `DataContractSerializer` outputs XML, so remember to specify the `DataContractJsonSerializer` if you want to use JSON.

TIP

It is common to put code that initializes a class in the constructor for that class. When you use the serialization engines provided by the system, the constructor is not called. This actually makes sense because the implication is that the class was already created and is serialized in a state that reflects the initialization. If you do need code to run when the class is deserialized, you can specify a member for the engine to call by tagging it with the `OnDeserialized` attribute. In the Wintellog example, you can see an instance of this in the `BlogItem` class. This ensures an event is registered regardless of whether the class was created using the `new` keyword or was deserialized.

The process to restore is similar. You still reference the file but this time open it for read access. The same serialization engine is used to create an instance of the type from the serialized data:

```
var folder = await ApplicationData.Current.LocalFolder
   .GetFolderAsync(folderName);
var file = await folder.GetFileAsync(hashCode);
var inStream = await file.OpenSequentialReadAsync();
var serializer = new DataContractJsonSerializer(typeof(T));
var retVal = (T)serializer.ReadObject(inStream.AsStreamForRead());
```

You can see when you start the application that the process of loading web sites, saving the data, and restoring items from the cache takes time. In the Windows Runtime, any process that takes more than a few milliseconds is defined as asynchronous. This is different from a synchronous call. To understand the difference, it is important to be familiar with the concept of *threading*.

The Need for Speed and Threading

In a nutshell, threading provides a way to execute different processes at the same time (concurrently). One job of the processor in your device is to schedule these threads. If you only have one processor, multiple threads take turns to run. If you have multiple processors, threads can run on different processors at the same time.

When the user launches an application, the system creates a main application thread that is responsible for performing most of the work, including responding to user input and drawing graphics on the screen. The fact that it manages the user interface has led to a convention of calling this thread the "UI thread." By default, your code will execute on the UI thread unless you do something to spin off a separate thread.

The problem with making synchronous calls from the UI thread is that all processing must wait for your code to complete. If your code takes several seconds, this means the routines that check for touch events or update graphics will not run during that period. In other words, your application will freeze and become unresponsive.

The Windows Runtime team purposefully designed the framework to avoid this scenario by introducing asynchronous calls for any methods

that might potentially take longer than 50 milliseconds to execute. Instead of running synchronously, these methods will spin off a separate thread to perform work and leave the UI thread free. At some point when their work is complete, they return their results. When the new `await` keyword is used, the results are marshaled automatically to the calling thread, which in many cases is the UI thread. A common mistake is to try to update the display without returning to the UI thread; this will generate an exception called a cross-thread access violation because only the UI thread is allowed to manage those resources.

Managing asynchronous calls in traditional C# was not only difficult, but resulted in code that was hard to read and maintain. Listing 6.1 provides an example using a traditional event-based model. Breakfast, lunch, and dinner happen asynchronously, but one meal must be completed before the next can begin. In the event-based model, an event handler is registered with the meal so the meal can flag when it is done. A method is called to kick off the process, which by convention ends with the text `Async`.

LISTING 6.1: Asynchronous Meals Using the Event Model

```
public void EatMeals()
{
    var breakfast = new Breakfast();
    breakfast.MealCompleted += breakfast_MealCompleted;
    breakfast.BeginBreakfastAsync();
}
void breakfast_MealCompleted(object sender, EventArgs e)
{
    var lunch = new Lunch();
    lunch.MealCompleted += lunch_MealCompleted;
    lunch.BeginLunchAsync();
}
void lunch_MealCompleted(object sender, EventArgs e)
{
    var dinner = new Dinner();
    dinner.MealCompleted += dinner_MealCompleted;
    dinner.BeginDinnerAsync();
}
void dinner_MealCompleted(object sender, EventArgs e)
{
    // done;
}
```

This example is already complex. Every step requires a proper registration (subscription) to the completion event and then passes control to an entirely separate method when the task is done. The fact that the process continues in a separate method means that access to any local method variables is lost and any information must be passed through the subsequent calls. This is how many applications become overly complex and difficult to maintain.

The Task Parallel Library (TPL) was introduced in .NET 4.0 to simplify the process of managing parallel, concurrent, and asynchronous code. Using the TPL, you can create meals as individual tasks and execute them like this:

```
var breakfast = new Breakfast();
var lunch = new Lunch();
var dinner = new Dinner();
var t1 = Task.Run(() => breakfast.BeginBreakfast())
   .ContinueWith(breakfastResult => lunch.
BeginLunch(breakfastResult))
   .ContinueWith(lunchResult => dinner.BeginDinner(lunchResult));
```

This helped simplify the process quite a bit, but the code is still not easy to read and understand or maintain. The Windows Runtime has a considerable amount of APIs that use the asynchronous model. To make developing applications that use asynchronous method calls even easier, Visual Studio 2012 provides support for two new keywords called `async` and `await`.

Understanding `async` and `await`

The `async` and `await` keywords provide a simplified approach to asynchronous programming. A method that is going to perform work asynchronously and should not block the calling thread is marked with the `async` keyword. Within that method, you can call other asynchronous methods to launch long running tasks. Methods marked with the `async` keyword can have one of three return values.

All `async` operations in the Windows Runtime return one of four interfaces. The interface that is implemented depends on whether or not the operation returns a result to the caller and whether or not it supports tracking progress. Table 6.1 lists the available interfaces.

TABLE 6.1: Interfaces Available for `async` Operations

	Reports Progress	Does Not Report Progress
Returns Results	`IAsyncOperationWithProgress`	`IAsyncOperation`
Does Not Return Results	`IAsyncActionWithProgress`	`IAsyncAction`

In C#, there are several ways you can both wrap calls to asynchronous methods as well as define them. Methods that call asynchronous operations are tagged with the `async` keyword. Methods with the `async` keyword that return `void` are most often event handlers. Event handlers require a void return type. For example, when you want to run an asynchronous task from a button tap, the signature of the event handler looks like this:

```
private void button1_Click(object sender, RoutedEventArgs e)
{
    // do stuff
}
```

To wait for asynchronous calls to finish without blocking the UI thread, you must add the `async` keyword so the signature looks like this:

```
private async void button1_Click(object sender, RoutedEventArgs e)
{
    // do stuff
    await DoSomethingAsynchronously();
}
```

Failure to add the `async` modifier to a method that uses `await` will result in a compiler error. Aside from the special case of event handlers, you might want to create a long-running task that must complete before other code can run but does not return any values. For those methods, you return a `Task`. This type exists in the `System.Threading.Tasks` namespace. For example:

```
public async Task LongRunningNoReturnValue()
{
    await TakesALongTime();
    return;
}
```

Notice that the compiler does the work for you. In your method, you simply return without sending a value. The compiler will recognize the method as a long-running `Task` and create the `Task` "behind the scenes" for you. The final return type is a `Task` that is closed with a specific return type. Listing 6.2 demonstrates how to take a simple method that computes a factorial and wrap it in an asynchronous call. The `DoFactorialExample` method asynchronously computes the factorial for the number 5 and then puts the result into the `Text` property as a string.

LISTING 6.2: Creating an Asynchronous Method That Returns a Result

```
public long Factorial(int factor)
{
    long factorial = 1;

    for (int i = 1; i <= factor; i++)
    {
        factorial *= i;
    }

    return factorial;
}

public async Task<long> FactorialAsync(int factor)
{
    return await Task.Run(() => Factorial(factor));
}

public async void DoFactorialExample()
{
    var result = await FactorialAsync(5);
    Result = result.ToString();
}
```

Note how easy it was to take an existing synchronous method (`Factorial`) and provide it as an asynchronous method (`FactorialAsync`) and then call it to get the result with the await keyword (`DoFactorialExample`). The `Task.Run` call is what creates the new thread. The flow between threads is illustrated in Figure 6.2. Note the UI thread is left free to continue processing while the factorial computes, and the result is updated and can be displayed to the user.

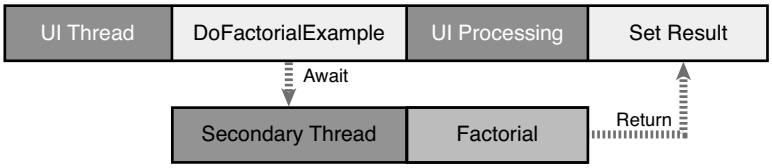


FIGURE 6.2: Asynchronous flow between threads

The examples here use the Task Parallel Library (TPL) because it existed in previous versions of the .NET Framework. It is also possible to create asynchronous processes using Windows Runtime methods like `ThreadPool.RunAsync`. You can learn more about asynchronous programming in the Windows Runtime in the development center at http://msdn.microsoft.com/en-us/library/windows/apps/hh464924.aspx. For a quickstart on using the await operator, visit http://msdn.microsoft.com/en-us/library/windows/apps/hh452713.aspx.

Lambda Expressions

The parameter that was passed to the `Task.Run` method is called a *lambda expression*. A lambda expression is simply an anonymous function. It starts with the signature of the function (if the `Run` method took parameters, those would be specified inside the parenthesis) and ends with the body of the function. I like to refer to the special arrow `=>` as the *gosinta* for "goes into." Take the expression from the earlier code snippet that is passed into `Task.Run`:

```
()=>Factorial(factor)
```

This can be read as "nothing goes into a call to Factorial with parameter factor." You can use lambda expressions to provide methods "on the fly." In the previous examples showing lunch, breakfast, and dinner, special methods were defined to handle the completion events. A lambda expression could also be used like this:

```
breakfast.MealCompleted += (sender, eventArgs)
                =>
            {
                // do something
            };
```

In this case, the lambda reads as "the `sender` and `eventArgs` goes into a set of statements that do something." The parameters triggered by the event are available in the body of the lambda expression, as are local variables defined in the surrounding methods. Lambda expressions are used as a short-hand convention for passing in delegates.

There are a few caveats to be aware of when using lambda expressions. Unless you assign a lambda expression to a variable, it is no longer available to reference from code, so you cannot unregister an event handler that is defined with a lambda expression. Lambda expressions that refer to variables within the method capture those variables so they can live longer than the method scope (this is because the lambda expression may be referenced after the method is complete), so you must be aware of the side effects for this. You can learn more about lambda expressions online at http://msdn.microsoft.com/en-us/library/bb397687(v=vs.110).aspx.

IO Helpers

The `PathIO` and `FileIO` classes provide special helper methods for reading and writing storage files. The `PathIO` class allows you to perform file operations by passing the absolute path to the file. Creating a text file and writing data can be accomplished in a single line of code:

```
await PathIO.WriteTextAsync("ms-appdata:///local/tmp.txt", "Text.");
```

The `ms-appdata` prefix is a special URI that will point to local storage for the application. You can also access local resources that are embedded in your application using the `ms-appx` prefix. In the sample application, an initial list of blogs to load is stored in JSON format under **Assets/Blogs.js**. The code to access the list is in the `BlogDataSource` class (under the **DataModel** folder)—the file is accessed and loaded with a single line of code:

```
var content = await PathIO
   .ReadTextAsync("ms-appx:///Assets/Blogs.js");
```

The `FileIO` class performs similar operations. Instead of taking a path and automatically opening the file, it accepts a parameter of type `IStorageFile`. Use the `FileIO` helpers when you already have a reference to

the file or need to perform some type of processing that can't be done by simply referencing the path.

Table 6.2 provides the list of available methods you can use. All of the methods take an absolute file path for the `PathIO` class and an `IStorageFile` object (obtained using the storage API) for the `FileIO` class:

TABLE 6.2: File Helper Methods from the `PathIO` and `FileIO` Classes

Method Name	Description
`AppendLinesAsync`	Appends lines of text to the specified file
`AppendTextAsync`	Appends the text to the specified file
`ReadBufferAsync`	Reads the contents of the specified file into a buffer
`ReadLinesAsync`	Reads the contents of the specified file into lines of text
`ReadTextAsync`	Reads the contents of the specified file into a single string as text
`WriteBufferAsync`	Writes data from a buffer to the specified file
`WriteBytesAsync`	Writes the byte array to the specified file
`WriteLinesAsync`	Writes the text lines to the specified file
`WriteTextAsync`	Writes the text to the specified file

Take advantage of these helpers where it makes sense. They will help simplify your code tremendously.

Embedded Resources

There are several ways you can embed data within your application and read it back. A common reason to embed data is to provide seed values for a local database or cache, configuration items, and special files such as license agreements. You can embed any type of resource, including images and text files. The applications you have worked with already include image resources.

To specify how a resource is embedded, right-click the resource name in the **Solution Explorer** and select **Properties** or select the item and press **Alt** + **Enter**. Figure 6.3 shows the result of highlighting the file **Blogs.js** in the **Assets** folder and selecting the **Properties** dialog. Note the **Build Action** and **Copy to Output Directory** properties.

FIGURE 6.3: Properties for a resource

When you set the action to `Content`, the resource is copied into a folder that is relative to the package for your application. In addition to the storage containers you learned about in Chapter 5, every package has an install location that contains the local assets you have specified the `Content` build action for. This will include resources such as images.

You can find the location where the package is installed using the `Package` class:

```
var package = Windows.ApplicationModel.Package.Current;
var installedLocation = package.InstalledLocation;
var loc = String.Format("Installed Location: {0}",
   installedLocation.Path);
```

An easier way to access these files is to use the `ms-appx` prefix. Open the **BlogDataSource.cs** file. The **Blogs.js** file is loaded in the `LoadLiveGroups` method. It is loaded by using the special package prefix, like this:

```
var content = await PathIO.ReadTextAsync(
    "ms-appx:///Assets/Blogs.js");
```

It is also possible to embed resources directly into the executable for your application. These resources are not visible in the file system but can

still be accessed through code. To embed a resource, set the **Build Action** to **Embedded Resource**. Accessing the resource is a little more complex.

To read the contents of an embedded resource, you must access the current assembly. An assembly is a building block for applications. One way to get the assembly is to inspect the information about a class you have defined:

```
var assembly = typeof(BlogDataSource).GetTypeInfo().Assembly;
```

The assembly is what the resource is embedded within. Once you have a reference to the assembly, you can grab a stream to the resource using the `GetManifestResourceStream` method. There is a trick to how you reference the resource, however. The resource will be named as part of the namespace for your assembly. Therefore, a resource at the root of a project with the default namespace `Wintellog` will be given the path:

```
Wintellog.ResourceName
```

The reference to the **ReadMe.txt** file in the **Common** folder is therefore `Wintellog.Common.ReadMe.txt`. This file is not ordinarily embedded in the project; the properties have been updated to illustrate this example. After you have retrieved the stream for the resource, you can use a stream reader to read it back. When the assembly reference is obtained, you can return the contents like this:

```
var stream = assembly.GetManifestResourceStream(txtFile);
var reader = new StreamReader(stream);
var result = await reader.ReadToEndAsync();
return result;
```

You will typically use embedded resources only when you wish to obfuscate the data by hiding it in the assembly. Note this will not completely hide the data because anyone with the right tools will be able to inspect the assembly to examine its contents, including embedded resources. Embedding assets using the `Content` build action not only makes it easier to inspect the assets from your application, but also has the added advantage of allowing you to enumerate the file system using the installed location of the current package when there are multiple assets to manage.

Collections

Collections are the primary structures you will use to manipulate data within your application. These classes implement common interfaces that provide consistent methods for querying and managing the data in the collection. Collections are often bound to UI controls. In the **Wintellog** example, a collection of blogs provides the grouped few and is bound to the `GridView` control. A collection of posts within the blogs feed the detail view within a group.

The Windows Runtime has a set of commonly used collection types. These types are mapped automatically to .NET Framework types by the CLR. In code, you won't reference the Windows Runtime types directly. Instead, you manipulate the .NET equivalent, and the CLR handles conversion automatically. Table 6.3 lists the Windows Runtime type and the .NET equivalent along with a brief description and example classes that implement the interface.

TABLE 6.3: Collection Types in the Windows Runtime and .NET

WinRT	.NET Framework	Example	Description
`IIterable<T>`	`IEnumerable<T>`	Most collection types	Provides an interface to support iteration for a collection
`IIterator<T>`	`IEnumerator<T>`	Exposed via collection type	The interface for performing iteration over a collection
`IVector<T>`	`IList<T>`	`List<T>`	A collection that can be individually accessed via an index
`IVectorView<T>`	`IReadOnlyList<T>`	`ReadOnly Collection<T>`	Version of an indexed collection that cannot be modified
`IMap<K,V>`	`IDictionary<K,V>`	`Dictionary<K,V>`	A collection of values that are referenced by keys

IMapView<K,V>	IReadOnly Dictionary<K,V>	ReadOnly Dictionary<K,V>	Version of a collection with key/value pairs that cannot be modified
IBindableIter-able	IEnumerable	Exposed via non-generic collections	Supports iteration over a non-generic collection
IBindableVector	IList	Custom classes that implement IList	Supports a non-generic collection that can be referenced by index

One important list that is not mapped to the Windows Runtime is the `ObservableCollection<T>`. This is a special list because it works with the data-binding system you learned about in Chapter 3, *Extensible Application Markup Language (XAML)*. The `ObservableCollection<T>` implements the `INotifyCollectionChanged` interface, which is designed to notify listeners when the list changes—for example, when items are added or removed or the entire list is refreshed.

For performance, the data-binding system does not constantly examine the lists you bind to UI controls. Instead, the initially bound list is used to generate the controls on the display. When you manipulate the list, the data-binding system receives a notification through the `CollectionChanged` event and can use the list of added and removed items to refresh the controls being displayed. Without the interface, the only way to have a list refresh the UI is to raise a `PropertyChanged` event for the property that exposes the list. This is inefficient because it results in the entire list being refreshed rather than only the items that changed.

Language Integrated Query (LINQ)

One major advantage of using collections is the ability to write queries against them using Language Integrated Query (LINQ). This feature extends the language syntax of C# to provide patterns for querying and updating data. LINQ itself works with providers for different types of data storage, such as a database backend (SQL) or an XML document. The

LINQ to Objects provider supports classes that implement the `IEnumerable` interface and therefore can be used with most collections.

LINQ to Objects is implemented as a set of extension methods to the existing `IEnumerable` interface. These extension methods are declared in the `System.Linq` namespace. Extension methods enable you to add methods to existing types without having to create a new type. They are a special type of static method that use a special `this` modifier for the first parameter. You can learn more about extension methods online at http://msdn.microsoft.com/en-us/library/bb383977(v=vs.110).aspx.

There are three fundamental steps involved with a LINQ query. The first step is to provide the data source or collection you will query against. The second step is to provide the query, and the final step is to execute the query. It's important to understand that creating a query does not actually invoke any action against the data source. The query only executes when you need it and then only processes results as you obtain them. This is referred to as *deferred execution*.

LINQ supports a variety of query operations. It also supports multiple syntaxes for querying data. The `BlogDataSource` class in the **Wintellog** project has a method called `LinqExamples`. This method is never called, but you can use it to see the various types of LINQ queries and syntaxes. The first syntax is referred to as *LINQ query syntax* and resembles the T-SQL syntax you may be used to working with in databases. The second syntax is method-oriented and is referred to as *method syntax*. The method syntax is constructed using lambda expressions.

The following series of examples shows both syntaxes, starting with the query syntax.

Queries

You can use simple queries to parse collections and return the properties of interest. The following examples produce a list of strings that represent the titles from the blog groups:

```
var query = from g in GroupList select g.Title;
var query2 = GroupList.Select(g => g.Title);
```

Filters

Filters allow you to restrict the data returned by a query. You can filter using common functions that compare and manipulate properties. In the following examples, the list is filtered to only those groups with a title that starts with the letter "A."

```
var filter = from g in GroupList
               where g.Title.StartsWith("A")
               select g;
var filter2 = GroupList.Where(g => g.Title.StartsWith("A"));
```

Sorting

You can sort in both ascending and descending order and across multiple properties if needed. The following queries will sort the blogs by title:

```
var order = from g in GroupList
            orderby g.Title
            select g;
var order2 = GroupList.OrderBy(g => g.Title);
```

Grouping

A powerful feature of LINQ is the ability to group similar results. This is especially useful in Windows 8 applications for providing the list for controls that support groups. The following queries will create groups based on the first letter of the blog title:

```
var group = from g in GroupList
            group g by g.Title.Substring(0, 1);
var group2 = GroupList.GroupBy(g =>
            g.Title.Substring(0, 1));
```

Joins and Projections

You can join multiple sources together and project to new types that contain only the properties that are important to you. The following query syntax

will join the items from one blog to another based on the date posted and then project the results to a new class with source and target properties for the title:

```
var items = from i in GroupList[0].Items
            join i2 in GroupList[1].Items
            on i.PostDate equals i2.PostDate
            select new
            {   SourceTitle = i.Title, TargetTitle = i2.Title };
```

Here is the same query using lambda expressions:

```
var items2 = GroupList[0].Items.Join(
    GroupList[1].Items,
    g1 => g1.PostDate,
    g2 => g2.PostDate,
    (g1, g2) => new { SourceTitle = g1.Title,
        TargetTitle = g2.Title });
```

This section only touched the surface of what is possible with LINQ expressions. You can learn more about LINQ by reading the articles and tutorials available online at http://msdn.microsoft.com/en-us/library/bb383799(v=vs.110).aspx.

Web Content

The Windows Runtime makes it easy to download and process web content. To access web pages, you will use the `HttpClient`. The class is similar to the `WebClient` class that Silverlight developers may be familiar with. This class is used to send and receive basic requests over the HTTP protocol. It can be used to send any type of standard HTTP request including `GET`, `PUT`, `POST`, and `DELETE`. The client returns an instance of `HttpResponseMessage` with the status code and headers of the response. The `Content` property contains the actual contents of the web page that was retrieved if the operation was successful.

NETWORK ACCESS

Using the `HttpClient` requires that you provide the appropriate capabilities to your application. There are different capabilities to use based on the location of the target web server. If the web server is located on your home or private network, you will set the **Private Networks (Client & Server)** capability. In most cases, you will be accessing a server that is located on the public Internet. This will require you to set the **Internet (Client)** capability, which is set by default when you create a new project.

The `BlogDataSource` class contains a helper method that provides an instance of `HttpClient`. The method sets a buffer size to allow for large pages to be loaded and provides a user agent for the request to use. User agents are most often used to identify the browser making the web request. In the case of programmatic access, you can pass an agent that provides information about the application and expected compatibility. Passing an agent that is compatible with mobile devices may result in the web server returning a page that is optimized for mobile browsing.

The Windows Runtime makes it easy to fetch a page asynchronously and process the results. The following two lines of code fetch the client and retrieve the page:

```
var client = GetClient();
var page = await client.GetStringAsync(item.PageUri);
```

Images are not always embedded within the RSS feed, so the code retrieves the target page for the entry and then parses it for images. This is done using regular expressions. The syntax for a regular expression provides a concise way to match patterns in strings of text. This makes it ideal for parsing tokens like HTML tags out of the source document.

REGULAR EXPRESSIONS

Regular expressions provide a powerful syntax for finding and replacing patterns in text. The first regular expression parsers were provided as part of early Unix-based distributions and were integrated into text editors and command-line utilities to parse large amounts of data. The .NET Framework provides the `System.Text.RegularExpressions.Regex` object to process text with regular expressions. Most operations involve two strings: a target string containing the text or process and a pattern string that contains the regular expression itself. To learn more about the regular expression syntax and how to use it in .NET, visit http://msdn.microsoft.com/en-us/library/hs600312.aspx.

The first expression parses all `image` tags from the source for the web page:

```
public const string IMAGE_TAG = @"<(img)\b[^>]*>";
private static readonly Regex Tags = new Regex(IMAGE_TAG,
   RegexOptions.IgnoreCase | RegexOptions.Multiline);
var matches = Tags.Matches(content);
```

Each tag is then parsed to pull the location of the image from the `src` attribute. This is used to construct an instance of an `Uri` that is added to the `ImageUriList` property of the blog post. This property is implemented as an `ObservableCollection` to provide notification when new images are added. A random image is displayed for each post. The image is hosted on the Internet, but Windows 8 will use a cached copy of the image when the user is offline if it has been downloaded previously.

Syndicated Content

Syndicated content is information that is available to other sites through special feeds. These feeds are most often presented in an XML format using either RSS (stands for RDF Site Summary, although it is commonly referred to as Real Simple Syndication—RDF is an abbreviation of Resource Description Framework) and Atom. Both formats have evolved

as standard XML-based ways for blogs, websites, and other content providers to expose data in a consistent way so that other programs can download and consume the data.

The RSS specification is available online at http://www.rssboard.org/rss-specification.

The Atom publishing protocol is available online at http://atompub.org/.

Both formats provide a way to specify a *feed*, which is a set of related entries (topics, articles, or posts). Each entry may have a post date, information about the author, a set of links that reference the original source, and rich content such as images and videos. To parse the data in the past, you would either have to read the specifications and write your own special XML parser or find a third-party parser that would do it for you.

The Windows Runtime provides the `SyndicationClient` class to make it easy for you to interact with feeds. This class exists in the `Windows.Web.Syndication` namespace. The class can be used to asynchronously retrieve feed information and can be provided credentials to connect with sources that require authentication. When passed a URL, it is capable of parsing feeds in Atom (0.3 and 1.0) and RSS (0.91, 0.92, 1.0, and 2.0) format and presenting them using a common object model.

The sample program retrieves the feed just using four lines of code. Two lines are not required and are used to take advantage of the browser cache and provide a custom user agent to the host website when requesting the data. A helper method named `GetSyndicationClient` returns the client with some default properties set in the `BlogDataSource` class:

```
private static SyndicationClient GetSyndicationClient()
{
    var client = new SyndicationClient
                        { BypassCacheOnRetrieve = false };
    client.SetRequestHeader("user-agent", USER_AGENT);
    return client;
}
```

Using the client is as simple as calling a method to retrieve the feed by passing the location of the feed:

```
var client = GetSyndicationClient();
var feed = await client.RetrieveFeedAsync(group.RssUri);
```

The result of the operation, if successful, is a `SyndicationFeed` object. The instance contains information about the location of the feed, categories or tags hosted by the feed, contributors to the feed, links associated with the feed, and of course the items that are posted to the feed. Each `SyndicationItem` in the feed hosts the location of the item, categories or tags specific to that item, the title and content of the item, and an optional summary.

You can follow the code in the example to see how easy it is to parse the feed and retrieve the necessary data. There is no need for you to specify the format of the feed because the class will figure this out automatically from the feed itself. Syndication is a powerful way to expose content and consume it in Windows 8 applications.

Streams, Buffers, and Byte Arrays

The traditional method for reading data from a file, website, or other source in .NET is to use a stream. A stream enables you to transfer data into a data structure that you can read and manipulate. Streams may also provide the ability to transfer the contents of a data structure back to the stream to write it. Some streams support seeking, finding a position within a stream the same way you might skip ahead to a different scene on a DVD movie.

Streams are commonly written to byte arrays. The byte array is the preferred way to reference binary data in .NET. It can be used to manipulate data like the contents of a file or the pixels that make up the bitmap for an image. Many of the stream classes in .NET support converting a byte array to a stream or reading streams into a byte array. You can also convert other types into a byte array using the `BitConverter` class. The following example converts a 64-bit integer to an array of 8 bytes (8 bytes x 8 bits = 64 bits) and then back again:

```
var bigNumber = 4523452345234523455L;
var bytes = BitConverter.GetBytes(bigNumber);
var copyOfBigNumber = BitConverter.ToInt64(bytes, 0);
Debug.Assert(bigNumber == copyOfBigNumber);
```

The Windows Runtime introduces the concept of an `IBuffer` that behaves like a cross between a byte array and a stream. The interface itself

only provides two members: a *Capacity* property (the maximum number of bytes that the buffer can hold) and a *Length* property (the number of bytes currently being used by the buffer). Many operations in the Windows Runtime either consume or produce an instance of `IBuffer`.

It is easy to convert between streams, byte arrays, and buffers. The methods to copy a stream into a byte array or send a byte array into a stream already exist as part of the .NET Framework. The `WindowsRuntimeBufferExtensions` class provides additional facilities for converting between buffers and byte arrays. It exists in the `System.Runtime.InteropServices.WindowsRuntime` namespace. It provides another set of extension methods including `AsBuffer` (cast a `Byte[]` instance to an `IBuffer`), `AsStream` (cast an `IBuffer` instance to a `Stream`), and `ToArray` (cast an `IBuffer` instance to a `Byte[]` instance).

Compressing Data

Storing large amounts of data can take up a large amount of disk space. Data compression encodes information in a way that reduces its overall size. There are two general types of compression. *Lossless compression* preserves the full fidelity of the original data set. *Lossy compression* can provide better performance and a higher compression ratio, but it may not preserve all of the original information. It is often used in image, video, and audio compression where an exact data match is not required.

The Windows 8 Runtime exposes the `Compressor` and `Decompressor` classes for compression. The **Compression** project provides an active example of compressing and decompressing a data stream. The project contains a text file that is almost 100 kilobytes in size and loads that text and displays it with a dialog showing the total bytes. You can then click a button to compress the text and click another button to decompress it back.

The compression task performs several steps. A local file is opened for output to store the result of the compressed text. There are various ways to encode text, so it first uses the Encoding class to convert the text to a UTF8 encoded byte array:

```
var storage = await ApplicationData.Current.LocalFolder
   .CreateFileAsync("compressed.zip",
    CreationCollisionOption.ReplaceExisting);
var bytes = Encoding.UTF8.GetBytes(_text);
```

You learned earlier in this chapter how to locate the folder for a specific user and application. You can examine the folder for the sample application to view the compressed file after you click the button to compress the text. The file is saved with a `zip` extension to illustrate that it was compressed, but it doesn't contain a true archive, so you will be unable to decompress the file from Windows Explorer.

ENCODING

Text and characters in various cultures and languages is stored internally as a series of bits and bytes. The way these bits are encoded can vary between different encoding schemes. One of the earliest schemes is known as ASCII and uses a sequence of 7-bit characters to encode text. More modern schemes include UTF-8 (uses sequences of 8-bit bytes, sometimes up to 3 bytes when needed) and UTF-16 (uses sequences of 16-bit integers). The .NET Framework provides the `Encoding` class to make it easier to work with various formats. You can read more about the class online at http://msdn.microsoft.com/en-us/library/86hf4sb8(v=vs.110).aspx.

The next lines of code open the file for writing, create an instance of the `Compressor`, and write the bytes. The code then completes the compression operation and flushes all associated streams:

```
using (var stream = await storage.OpenStreamForWriteAsync())
{
    var compressor = new Compressor(stream.AsOutputStream());
    await compressor.WriteAsync(bytes.AsBuffer());
    await compressor.FinishAsync();
}
```

When the compression operation is complete, the bytes are read back from disk to show the compressed size. You'll find the default algorithm cuts the text file down to almost half of its original size. The decompression operation uses the `Decompressor` class to perform the reverse operation and retrieve the decompressed bytes in a buffer (it then saves these to disk so you can examine the result).

```
var decompressor = new Decompressor(stream.AsInputStream());
var bytes = new Byte[100000];
var buffer = bytes.AsBuffer();
var buf = await decompressor.ReadAsync(buffer, 999999,
   InputStreamOptions.None);
```

When you create the classes for compression, you can pass a parameter to determine the compression algorithm that is used. Table 6.4 lists the possible values.

TABLE 6.4: Compression Algorithms

CompressAlgorithm Member	Description
InvalidAlgorithm	Invalid algorithm. Used to generate exceptions for testing.
NullAlgorithm	No compression is applied, and the buffer is simply passed through. Used primarily for testing.
Mszip	Uses the MSZIP algorithm.
Xpress	Uses the XPRESS algorithm.
XpressHuff	Uses the XPRESS algorithm with Huffman encoding.
Lzms	Uses the LZMS algorithm.

The Windows Runtime makes compression simple and straightforward. Use compression when you have large amounts of data to store and are concerned about the amount of disk space your application requires. Remember that compression will slow down the save operation, so be sure to experiment to find the algorithm that provides the best compression ratio and performance for the type of data you are storing. Remember that you must pass the same algorithm to the decompression routine that you used to compress the data.

Encrypting and Signing Data

Many applications store sensitive data that should be encrypted to keep it safe from prying eyes. This may be information about the user or internal data for the application itself. Other information may need to be signed. Signing generates a specialized hash of data that provides a unique signature. If the original data is tampered with, the signature of the data will change. You can verify the signature against the original to determine if the data was modified in any way.

Encryption and signing is handled in the Windows Runtime by the `CryptographicEngine` class. This class provides services to encrypt, decrypt, sign, and verify the signature of digital content. The **EncryptionSigning** project contains some simple examples of performing these operations. The main code is located in the **MainPage.xaml.cs** file.

Encryption and decryption operations require a special *key*. Think of a key as a password for the encryption and decryption operations. There are two types of keys you can use. The most straightforward is called a *symmetric key*, which uses the same password or "secret" to both encrypt and decrypt the information.

To produce a key, you use the `SymmetricKeyAlgorithmProvider` class. You initialize the class by calling `OpenAlgorithm` with the name of the algorithm you wish to use. You then call `CreateSymmetricKey` to generate the key for the encryption operation. This same key must be used to decrypt the data later on. You can read the list of valid algorithms in the MSDN documentation at http://msdn.microsoft.com/en-us/library/windows/apps/windows.security.cryptography.core.symmetrickeyalgorithmprovider.openalgorithm.aspx.

In the example application, the **RC4** stream cipher is used to encrypt and decrypt the data. The user is prompted for one of two passwords, and then the passwords are repeated 100 times to fill a buffer. You can use any source data that can be converted to an array of bytes for the key. A helper utility is included in the code to help convert a string to an instance of `IBuffer`:

```
var buffer = CryptographicBuffer
    .ConvertStringToBinary(str.Trim(),
    BinaryStringEncoding.Utf8);
return buffer;
```

The `CryptographicBuffer` class provides a set of helper utilities for encryption, decryption, and signing operations. It supports comparing two instances of a buffer, converting between strings and binary arrays using various encodings, decoding and encoding using **Base64**, and generating a buffer of random data. In this example, it is used to encode the string using UTF8 to a buffer that is returned.

BASE64

Base64 is an encoding scheme that supports converting binary data to ASCII. This enables you to transmit binary data in a medium that supports only strings and text. If you want to store the bitmap for an image in a JSON object, you can encode it using Base64 and then decode it when you deserialize the JSON string. The size of the encoded text will always be greater than the source image because the conversion to ASCII requires that it does not use all of the bits available in a character byte. The most common scheme is to encode 3 bytes (24 bits) to 4 bytes (32 bits). For this reason, Base64 is sometimes referred to as 3-to-4 encoding.

Using the helper method, the code produces the key like this:

```
var result = await GetPassword();
var provider = SymmetricKeyAlgorithmProvider.OpenAlgorithm("RC4");
var key = provider.CreateSymmetricKey(AsBuffer(result));
```

When the key is generated, it is a simple step to encrypt the source text with the key. The result is encoded using Base64 so that it can be updated to the `TextBlock` in the right column for display:

```
var encrypted = CryptographicEngine.Encrypt(key,
    AsBuffer(BigTextBox.Text), null);
_encrypted = encrypted.ToArray();
BigTextBlock.Text = CryptographicBuffer
    .EncodeToBase64String(encrypted);
```

When you encrypt the text, the decrypt button is enabled. The user is given the option to select a password again for the decryption. If the user chooses a password that is different from the one used in the encryption operation, the decrypt process will fail or produce illegible output. The decryption process produces a key the same way the encryption process does and then simply calls the `Decrypt` method on the `CryptographicEngine` class:

```
var decrypted = CryptographicEngine.Decrypt(key,
    _encrypted.AsBuffer(), null);
BigTextBox.Text = AsText(decrypted).Trim();
```

It is also possible to encrypt and decrypt using an *asymmetric key*. The `AsymmetricKeyAlgorithmProvider` is used to generate asymmetric keys. Asymmetric encryption uses two different keys, a "public" key and a "private" key to perform encryption and decryption. This allows you to encrypt the data with your private secret but provide a public key for decryption. It allows third parties to decrypt the data while keeping your secrets safe.

You can learn more about asymmetric keys and see sample code online at http://msdn.microsoft.com/en-us/library/windows/apps/windows.security.cryptography.core.asymmetrickeyalgorithmprovider.openalgorithm.aspx.

The key used to sign data can be generated using the `MacAlgorithmProvider` class. This class represents a Message Authentication Code (MAC). You can create the key using any one of the popular algorithms including Message-Digest Algorithm (MD5), Secure Hash Algorithm (SHA), and Cipher-based MAC (CMAC). The key is generated much the same way as the encryption key. In the example project, a default password is used to generate the key for the signature:

```
var provider = MacAlgorithmProvider.OpenAlgorithm("HMAC_SHA256");
var key = provider.CreateKey(
    AsBuffer(MakeBigPassword(PASSWORD1)));
```

The signing process generates a buffer the same way the encryption process does. The difference is that you can use the buffer output by encryption to decrypt the message and produce the original. The signature is one-way—you cannot recreate the message from the signature. It's only

function is to compare against an existing message to determine whether or not it has been tampered with.

The signature is generated with a call to the `Sign` method on the `CryptographicEngine` class:

```
_signature = CryptographicEngine.Sign(key,
    AsBuffer(BigTextBox.Text)).ToArray();
```

The signature is verified with a call to `VerifySignature` that will return true if the text has not been altered since the signature was generated:

```
var result = CryptographicEngine.VerifySignature(key,
    AsBuffer(BigTextBox.Text),
    _signature.AsBuffer());
```

To see how this works, launch the sample application and tap the **Sign** button. Now tap the **Verify** button to see that the text has not been altered. Now add a space or other character to the text in the left column and tap **Verify** again. This time you should receive a message that the text has been tampered with.

Windows 8 provides a set of powerful algorithms for encrypting, decrypting, and signing data. The process is made extremely simple through the use of the `CryptographicEngine`, `CryptographicBuffer`, and key provider classes. Use encryption to secure data both internal to your application and for transport over the Internet and use signatures to verify that data has not been tampered with in-transit.

Web Services

A web service is a method for communication between devices over the Internet. The most common protocol for communication is the Simple Object Access Protocol (SOAP) that was designed in 1998. If you've worked with SOAP, you know there is nothing simple about it, and a new protocol known as Representational State Transfer (REST) is quickly gaining popularity.

Web services are important for communications in applications. Many enterprise systems expose web services for consumption by client software

like this Windows 8 application. One advantage of using SOAP is that it provides a discovery mechanism through the Web Services Description Language (WSDL) that allows the client application to determine the signature and structure of the API. You can learn more about the WSDL specification online at http://www.w3.org/TR/wsdl.

Open the **WeatherService** project to see an example of using web services. The example uses a free web service to obtain weather information. Connecting to the service was as simple as right-clicking the **References** node of the **Solution Explorer**, choosing **Add Service Reference**, and entering the URL for the service. The result is shown in Figure 6.4.

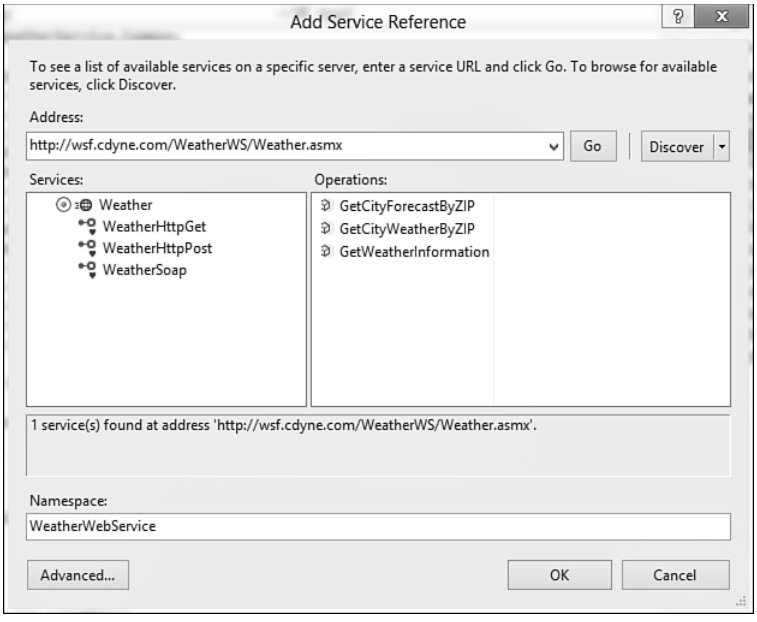


FIGURE 6.4: Adding a SOAP-based web service

When the service is added, a client proxy is generated automatically. The proxy handles all of the work necessary to make a request to the API and return the data and presents these as asynchronous implementations of the server interfaces. In the example application, the user is prompted to enter a zip code. When the button is clicked, the zip code is validated and, if there are no errors, passed to the web service. The following line of code

is all that is needed to create a proxy for connecting to the web service and to call it with the zip code (note the convention of using `Async` at the end of the method name):

```
var client = new WeatherWebService.WeatherSoapClient();
var result = await client.GetCityForecastByZIPAsync(
    zip.ToString());
```

If the result does not indicate a successful call, a dialog is shown that indicates there was a problem. Otherwise, the results are bound to the grid and shown through data-binding. This is all it takes to data-bind the results from the web service call:

```
ResultsGrid.DataContext = result;
```

The XAML is set to show the city and state:

```
<StackPanel Orientation="Horizontal">
    <TextBlock Text="{Binding City}"/>
    <TextBlock Text=","/>
    <TextBlock Text="{Binding State}"/>
</StackPanel>
```

Listing 6.3 shows the full XAML for the individual forecast items. The "description" field is purposefully misspelled because this is how it came across in the web service as of the time of this writing.

LISTING 6.3: Data-Binding the Results from a Weather Service

```
<ListView Grid.Row="1" ItemsSource="{Binding ForecastResult}">
    <ListView.ItemTemplate>
        <DataTemplate>
            <StackPanel Orientation="Horizontal">
                <TextBlock Text="{Binding Date,
➥Converter={StaticResource ConvertDate}}"
                           Width="200"/>
                <TextBlock Text="{Binding Description}"
➥Width="150" Margin="5 0 0 0"/>
                <Image Source="{Binding Description,
➥Converter={StaticResource ConvertImage}}"/>
                <TextBlock Text="{Binding Temperatures.MorningLow}"
➥Margin="5 0 0 0" Width="50"/>
                <TextBlock Text="{Binding Temperatures.DaytimeHigh}"
➥Margin="5 0 0 0" Width="50"/>
            </StackPanel>
```

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

The weather service documentation provided a set of icons that correspond to the description. The `ImageConverter` class takes the description and translates it to a file name so it can return the image:

```
var filename =
    string.Format("ms-appx:///Assets/{0}.gif",
        ((string) value).Replace(" ", string.Empty).ToLower());
return new BitmapImage(new Uri(filename, UriKind.Absolute));
```

Figure 6.5 displays the result of my request for the weather forecast of my hometown (Woodstock, Georgia) via its zip code.

FIGURE 6.5: The weather forecast for Woodstock, Georgia

OData Support

The Open Data Procotol (OData) is a web protocol used for querying and updating data. It is a REST-based API built on top of Atom that uses JSON

or XML for transporting information. You can read more about OData online at http://www.odata.org/.

Windows 8 applications have native support for OData clients once you download and install the client from:

http://go.microsoft.com/fwlink/?LinkId=253653

To access OData services, you simply add a service reference the same way you would for a SOAP-based web service. A popular OData service to use for demonstrations is the Netflix movie catalog. You can browse the service directly by typing http://odata.netflix.com/catalog/ into your browser.

In most browsers, you should see an XML document that contains various tags for collections you may browse. For example, the collection referred to as **Titles** indicates you can browse all titles using the URL, http://odata.netflix.com/catalog/Titles.

The **Netflix** project shows a simple demonstration of using this OData feed. The main URL was added as a service reference the same way the weather service was added in the previous example. The first step in using the service is to create a proxy to access it. This is done by taking the generated class from adding the service and passing in the service URL:

```
var netflix =
    new NetflixCatalog(
        new Uri(
            "http://odata.netflix.com/Catalog/",
            UriKind.Absolute));
```

Next, set up a collection for holding the results of an OData query. This is done using the special `DataServiceCollection` class:

```
private DataServiceCollection<Title> _collection;
...
_collection = new DataServiceCollection<Title>(netflix);
TitleGrid.ItemsSource = _collection;
```

Finally, specify a query to filter the data. This query is passed to the proxy and will load the results into the collection. In this example, the query will grab the first 100 titles that start with the letter "Y" in order of highest rated first:

```
var query = (from t in netflix.Titles
                where t.Name.StartsWith("Y")
                orderby t.Rating descending
                select t).Take(100);
_collection.LoadAsync(query);
```

Finally, as data comes in, you have the option to page in additional sets of data. This is done by checking the collection for a *continuation*. If one exists, you can request that the service load the next set. This allows you to page in data rather than pull down an extremely large set all at once:

```
if (_collection.Continuation != null)
{
    _collection.LoadNextPartialSetAsync();
}
```

Run the included sample application. You should see the titles and images start to appear asynchronously in a grid that you can scroll through. As in the previous example, the results of the web service are bound directly to the grid:

```
<Image Stretch="Uniform" Width="150" Height="150">
    <Image.Source>
        <BitmapImage UriSource="{Binding BoxArt.LargeUrl}"/>
    </Image.Source>
</Image>
<TextBlock Text="{Binding Name}" Grid.Row="1"/>
```

The Windows 8 development environment makes it easy and straightforward to connect to web services and pull data in from external sources. Many existing applications expose web services in the form of SOAP, REST, and OData feeds. The built-in support to access and process these feeds makes it possible to build Windows 8 applications that support your existing functionality when it is exposed via web services.

Summary

This chapter explored a variety of ways you can deal with data in your Windows 8 applications. You learned how to save and retrieve data from file storage, access it over the Web, and syndicate it through RSS and Atom

feeds. You learned about the built-in tools that make it easy to encrypt and sign data. Finally, you saw how easy it is to connect to existing SOAP and OData web services by generating proxies and retrieving data asynchronously from external APIs.

In the next chapter, you will learn how to keep your application alive even when it is not running through the use of tiles and notifications. Tiles provide information to the user at a glance on their Start screens and can be refreshed even when the application is not running. Notifications can be generated from within the application or by an external source to inform the user when important events happen and provide a contextual link back into the application.

Index

A

B

C

D

E

F

G

H

I

J–K

L

M

N

O–P

Q–R

S

T

U

V

W

X–Y–Z